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 Kimmelmann, 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 mittes 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 relationale 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. Insbesonder 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	Kimmelmann	Kimmelmann	Kimmelmann	Kimmelmann
Systemadministration		Kimmelmann	-	-

Tabelle 0.1: Aufgabenverteilung

Die packages

- amsthm,
- lstautogobble,
- multirow

wurden zusaätzlich zur Standardvorlage verwendet.

Inhaltsverzeichnis

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. 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+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 Operation 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]. Im folgenden sollen verschiedene Aspekte der Graphentheorie, die zur Modellierung von Graphdatenbanken von Bedeutung sind, kurz vorgestellt werden.

1.1.1 Graph

Das simpelste Graphdatenbankmodell ist ein einfacher Graph. Ein Graph ist mathematisch folgendermaßen definiert:

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)*

- γ *, einer Inzidenzabbildung (incidence relation), mit* $\gamma : E \longrightarrow \{X | X \subseteq V, 1 \leq |X| \leq 2\}$

[Bec18, Seite 21]

Ein Knoten representiert ein Element in einem Graphen und die Kanten stellen die Beziehung zwischen den einzelnen Knoten her. In einem einfachen Graphen kann eine Kante immer nur jeweils zwei Knoten miteinander verbinden. Zwei Konten heißen adjazent, wenn diese über eine Kante direkt miteinander verbunden sind. Eine Kante die mit einem Knoten verbunden is wird als inzident zu diesem Knoten bezeichnet [KK15].

Graphen können gerichtet oder ungerichtet sein. Gerichtete Graphen zeichnen sich dardurch 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 das Modells erlaubt.

Um die Beziehung zwischen zwei Knoten genauer zu definieren, lassen sich die Kanten gewichten. Dabei werden den Kanten in der Regel nummerische 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, Seite 13] 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:

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

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 weisen 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. Ein Baum mit n Knoten hat immer $n-1$ Kanten.[Rah17a] 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. Bei einem gewurzelten Baum werden die Ausgangsknoten als Eltern und die Zielknoten jeweils als Kinder der Ausgangsknoten bezeichnet. Hat in einem gewurzelten Baum jeder Knoten maximal zwei Kinder, wird dieser als Binärbaum bezeichnet.[Rah17a]

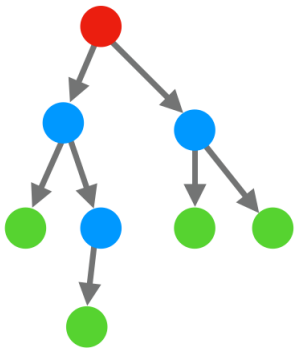


Abbildung 1.1: gewurzelter Baum

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. 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 Graphdatenbanken.

Abbildung 1.1.3 zeigt einen Property Graphen. Die Knoten sind den drei Labeln 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.

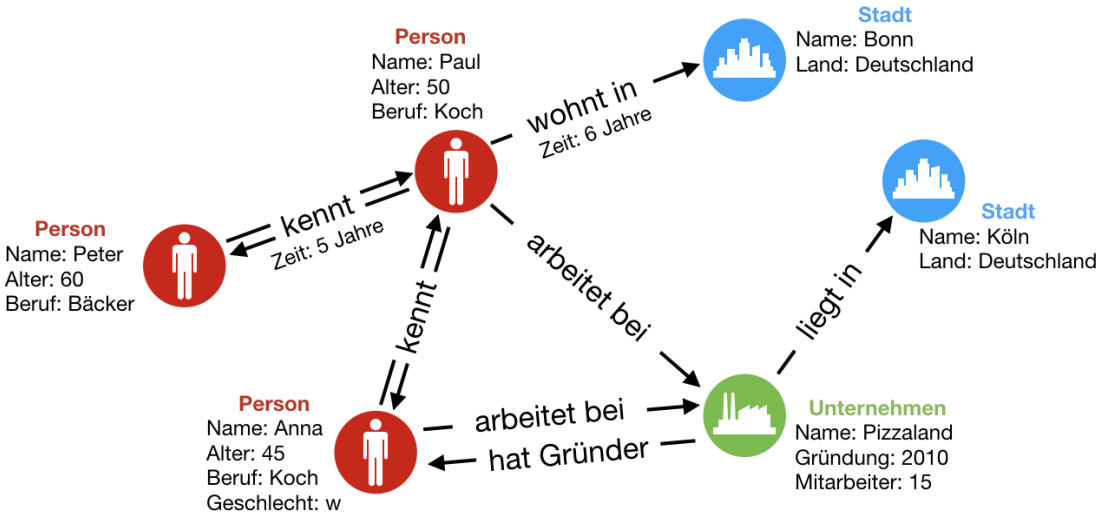


Abbildung 1.2: Property Graph

1.1.4 Hypergraphen

Hypergraphen stellen eine generalisierung von vormalen Graphen dar und ermöglichen die modelierung komplexer Beziehungen [AG18]. Hypergraphen haben die Eigenschaft, dass Kanten im Gegensatz zu klassischen Graphen mehr als zwei Knoten miteinander verbinden können. Die Kanten des Hypergraphen werden auch als Hyperkanten bezeichnet. Im Falle eines gerichteten Hypergraphen verbindet die Hyperkante den Ausgangsknoten direkt mit allen Zielknoten.
Mathematisch ist ein Hypergraph folgendermaßen definiert:

Definition. *Let* $X = \{v_1, v_2, ..., v_n\}$ *be a finite set, and let* $E = \{e_1, e_2, ..., e_m\}$ *be a family of subsets of* X *such that*

$$e_i \neq \varnothing (i = 1, 2, ..., m) \cup_{i=1}^{m} e_i = X.$$

The pair $H = (X, E)$ *is called a hypergraph with vertex set* X *and hyperedge set* E*. The elements* $v_1, v_2, ..., v_n$ *of* X *are vertices of hypergraph* H*, and the sets* $e_1, e_2, ..., e_m$ *are hyperedges of hypergraph* H*.[ZSHZ18, Seite 2]*

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.

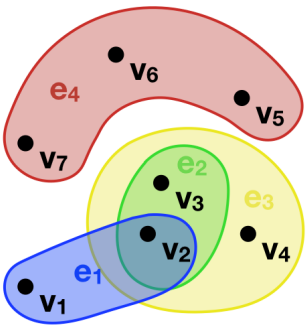


Abbildung 1.3: Hypergraph

Im Vergleich zum normalen Graphen können die Kanten eines Hypergraphen eine beliebige Kardinalität haben (siehe Definition Graph Kapitel 1.2.1). Beim normalen Graphen können die Kanten nur die Kardinalität $1 \leq |X| \leq 2$ haben. Die Hyperedges in einem Hypergraphen sind somit eine beliebige Menge von Knoten.

Da die Hypergraphen eine flexiblere Struktur als das einface Graphen-Modell bietet, werden diese oft zur Modellierung in Graphdatenbanken verwendet.[Goe15]

1.1.5 Hypernode

Hypernodes ermöglichen ermöglichen die Verschachtelung von Graphen.

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, pgtclng, 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)
- Anfragemethoden
  - 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), besoners 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 PostgresSQL: 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 Traversierung 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 Hilve 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 *initdb* ausgeführt werden. Über *initdb* wird ein PostgreSQL-Cluster angelegt. Als Parame-

ter 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 Fremdschlüssel 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 [Grob] . 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 disuktiert. Im Anhang befinden sich Scripte für die Erstellung des Index für alle

relation Tabellen, sowie die Erstellung von partitionierten Tabellen mit Index für alle relation Tabellen.

3.1.5 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 bis zur Rekursionsstufe 5 traversiert (siehe A.20), sowie ein Script, welches den Standard SQL Source Code dynamisch generiert (siehe A.19). Der SQL Befehl WITH RECURSIVE sorgt dafür, dass die Abfrage sich mit der Ergebnismenge wieder selber aufruft. Die Struktur einer WITH Query sieht folgedermaßen aus: Zuerst wird der nicht rekursive Teil der Query angegeben, anschließend wird der rekursive Teil beschrieben:

Listing 3.1: 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 gelöscht, die Ergebissmenge wird in die Intermediate Tabelle geschrieben, die Ergebnisse aus der Intermediate Tabelle werden in die Working Tabelle kopiert ([Groc]). Die Abbruchbedingung für die Rekursion wird innerhalb der Funktion definiert. Ob die Abbruchbedingung erreicht ist, wird im rekursiven Teil der Funktion überprüft:

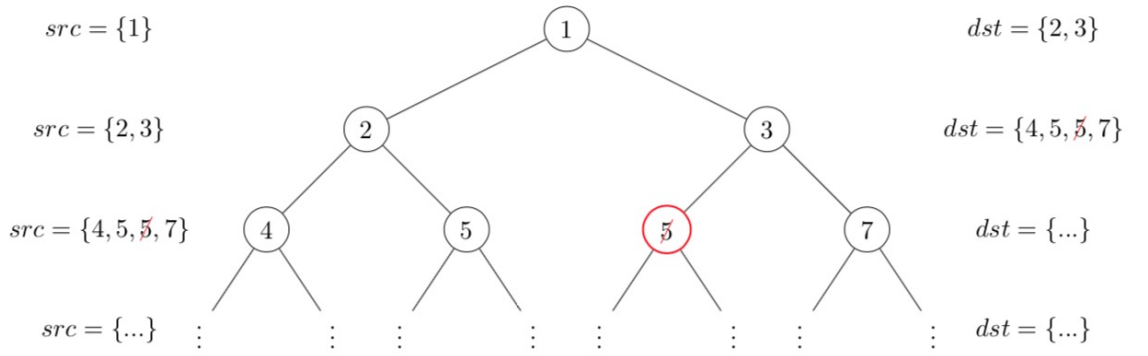
Listing 3.2: Überprüfen der Abbruchbedingung

```
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
```

3.1.6 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 Rekursionstufe 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 Ergebnissmenge der momentanen Rekursionnstufe 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.

3.1.7 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).

3.1.8 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 [1] 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 [2] erstellt.

[1] 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])

[2] "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

4.1.1 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 Anzahl der 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 verewendet.

- -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.

Antwortzeiten ohne Indices

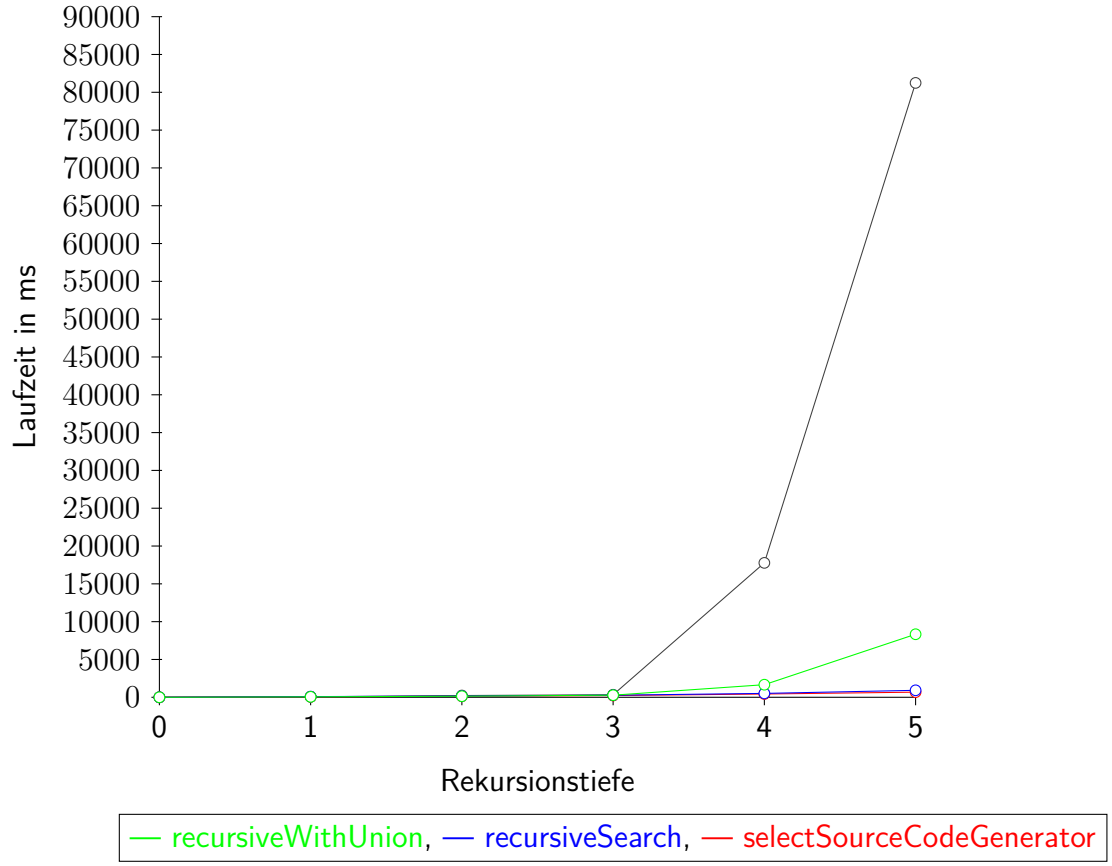


Abbildung 4.1: relation_epinions

	Laufzeit in MS			
Rerkusions-tiefe	selectSource CodeGenerator	recursiveSearch	recursive WithUnion	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

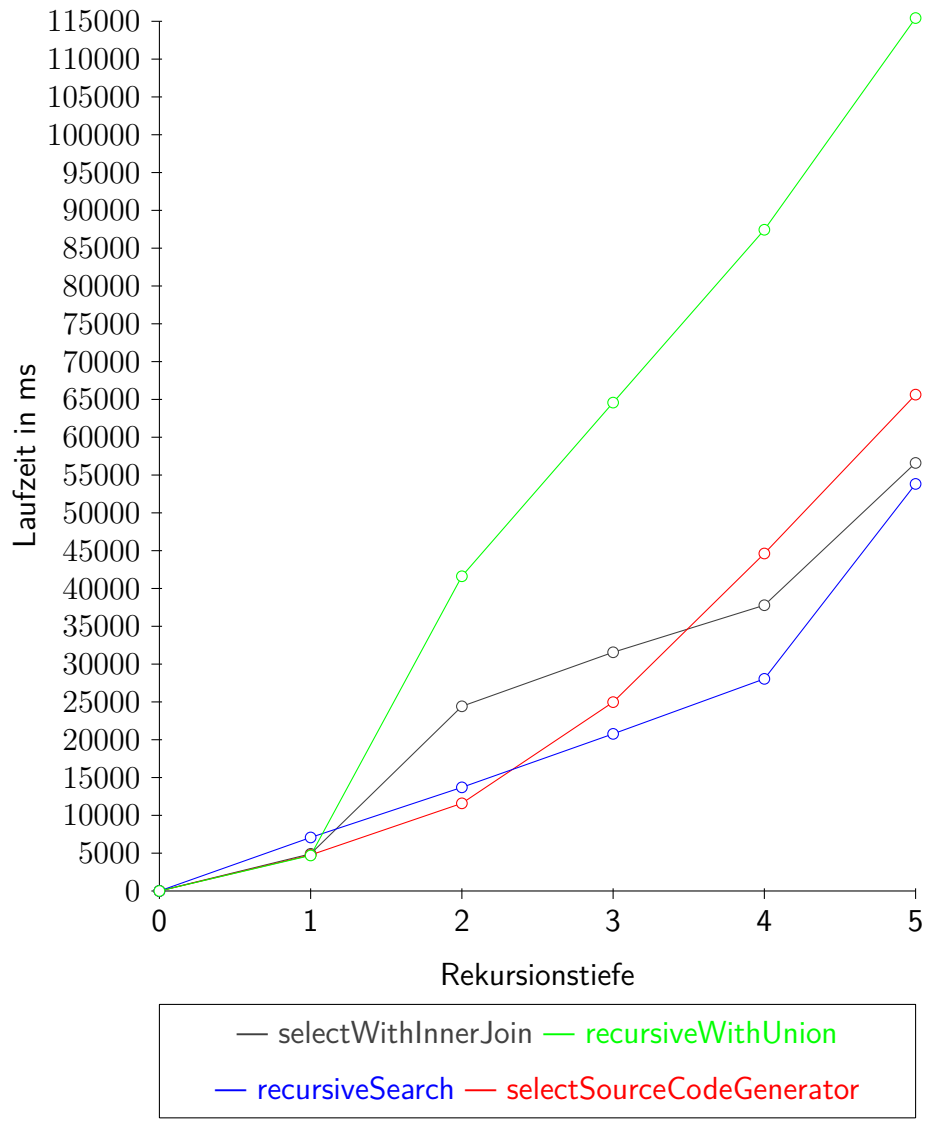


Abbildung 4.2: relation_livejournal

	Laufzeit in MS			
Rerkusions-tiefe	selectSource CodeGenerator	recursiveSearch	recursive WithUnion	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

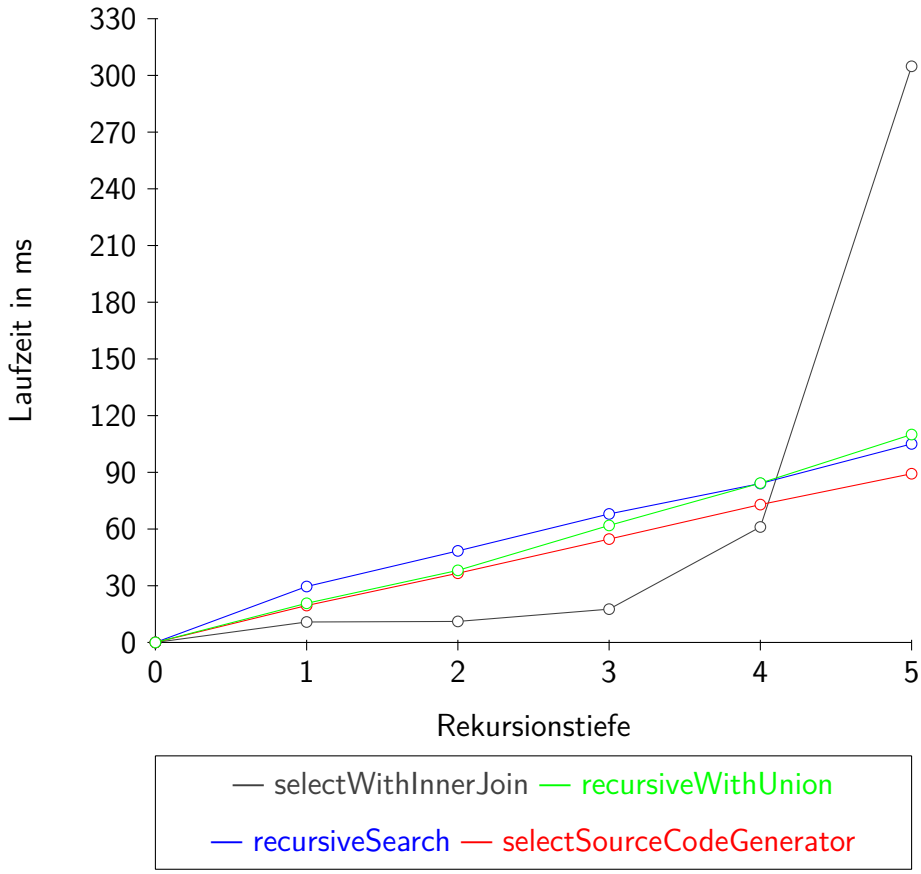


Abbildung 4.3: relation_facebook

	Laufzeit in MS			
Rerkusions-tiefe	selectSource CodeGenerator	recursiveSearch	recursive WithUnion	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

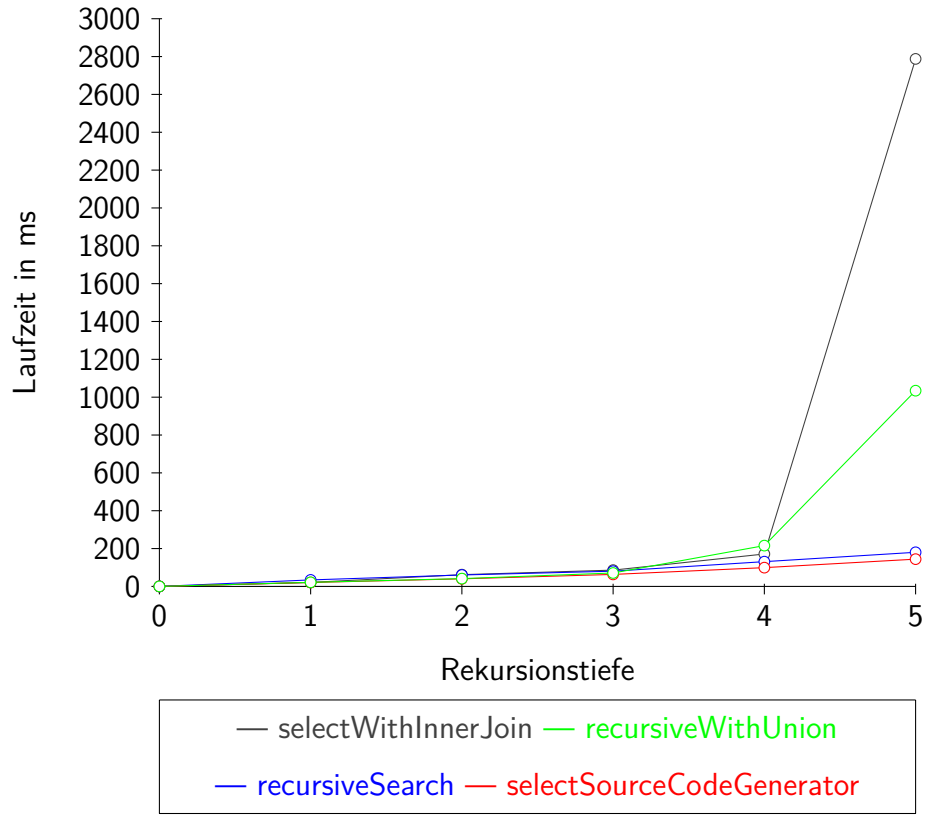


Abbildung 4.4: relation_wiki_vote

	Laufzeit in MS			
Rerkusions-tiefe	selectSource CodeGenerator	recursiveSearch	recursive WithUnion	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

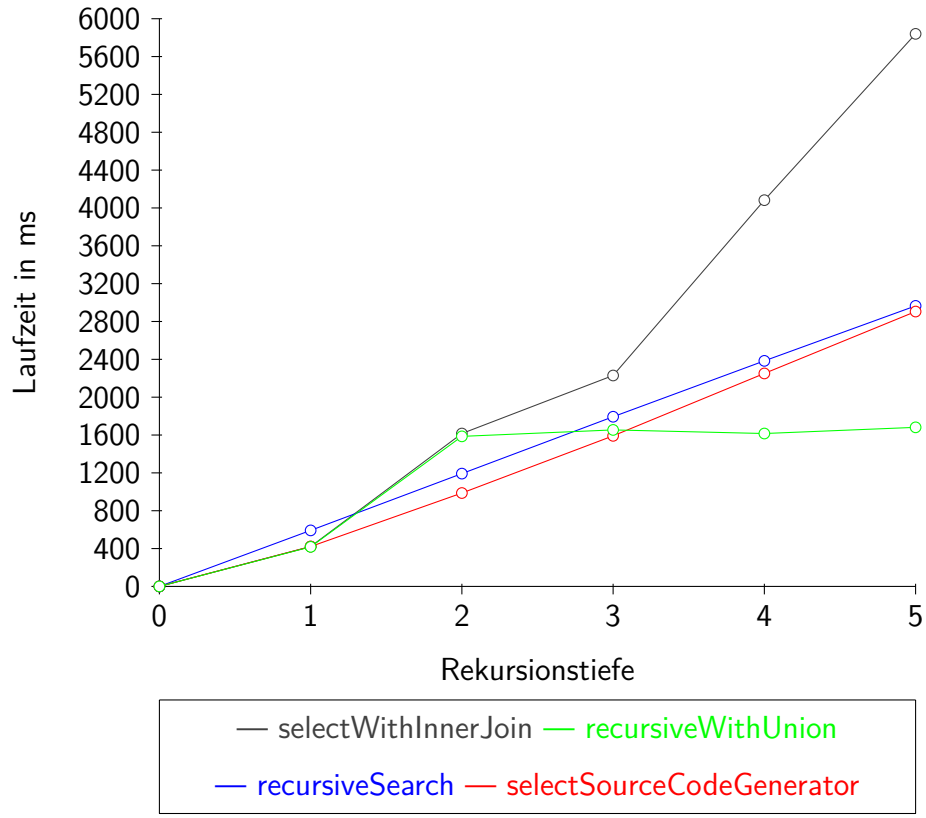


Abbildung 4.5: relation_youtube

	Laufzeit in MS			
Rerkusions-tiefe	selectSource CodeGenerator	recursiveSearch	recursive WithUnion	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

Antwortzeiten mit Indices

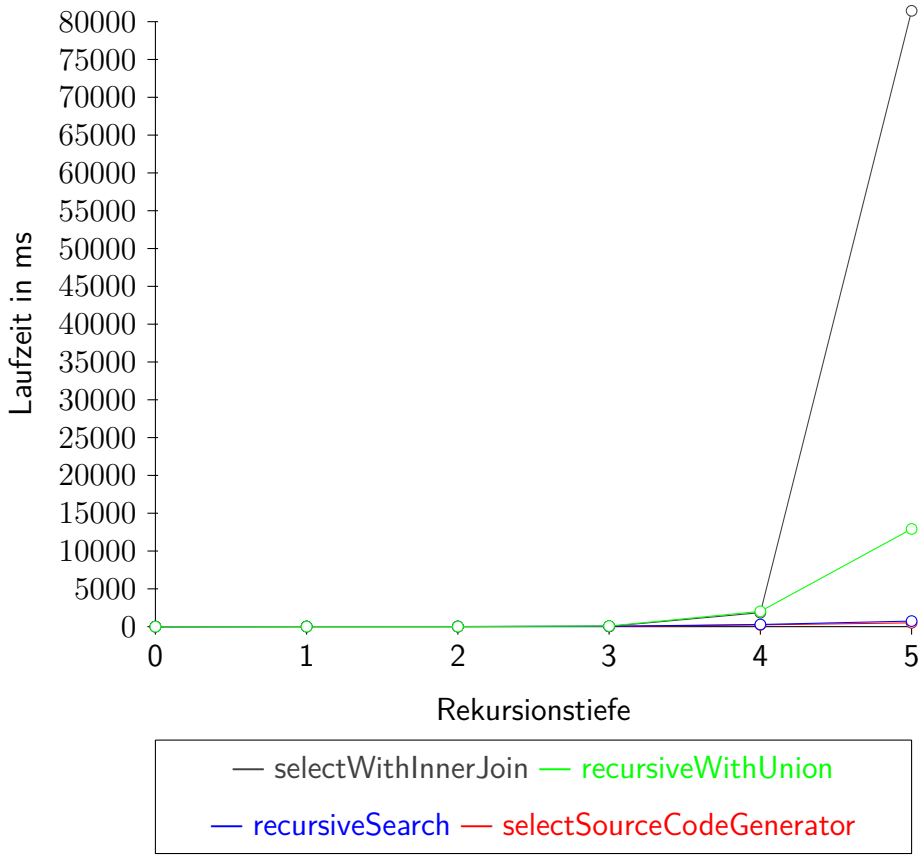


Abbildung 4.6: relation_epinions_with_index

	Laufzeit in MS			
Rerkusions-tiefe	selectSource CodeGenerator	recursiveSearch	recursive WithUnion	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

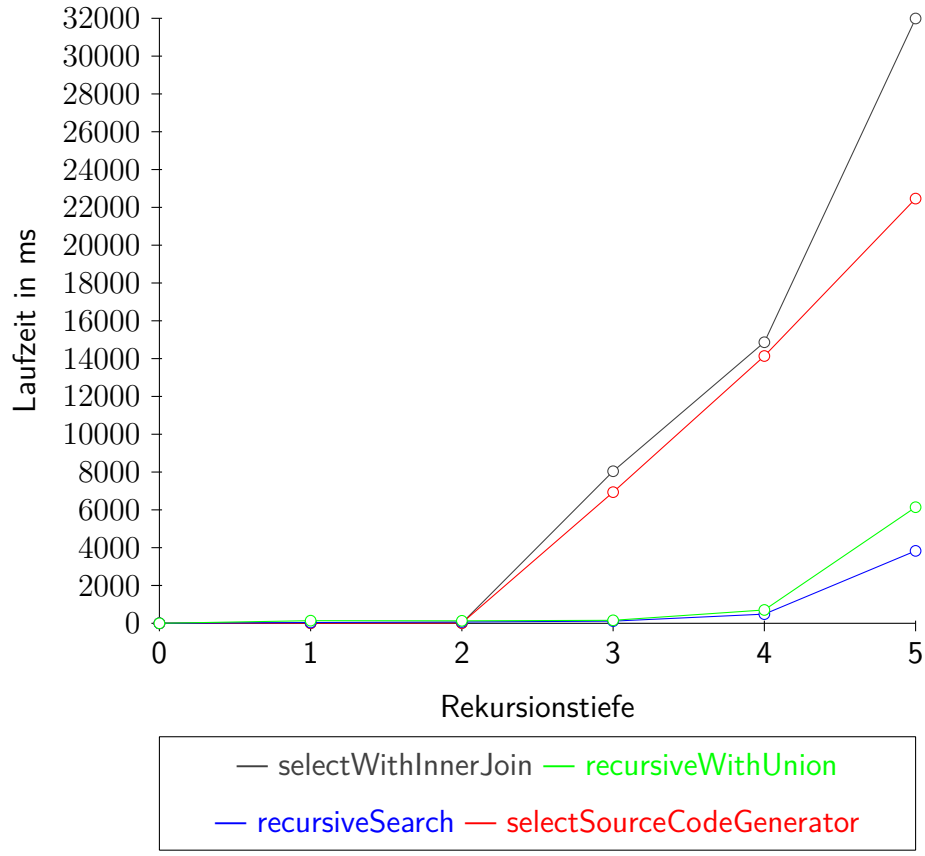


Abbildung 4.7: relation_livejournal

	Laufzeit in MS			
Rerkusions-tiefe	selectSource CodeGenerator	recursiveSearch	recursive WithUnion	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

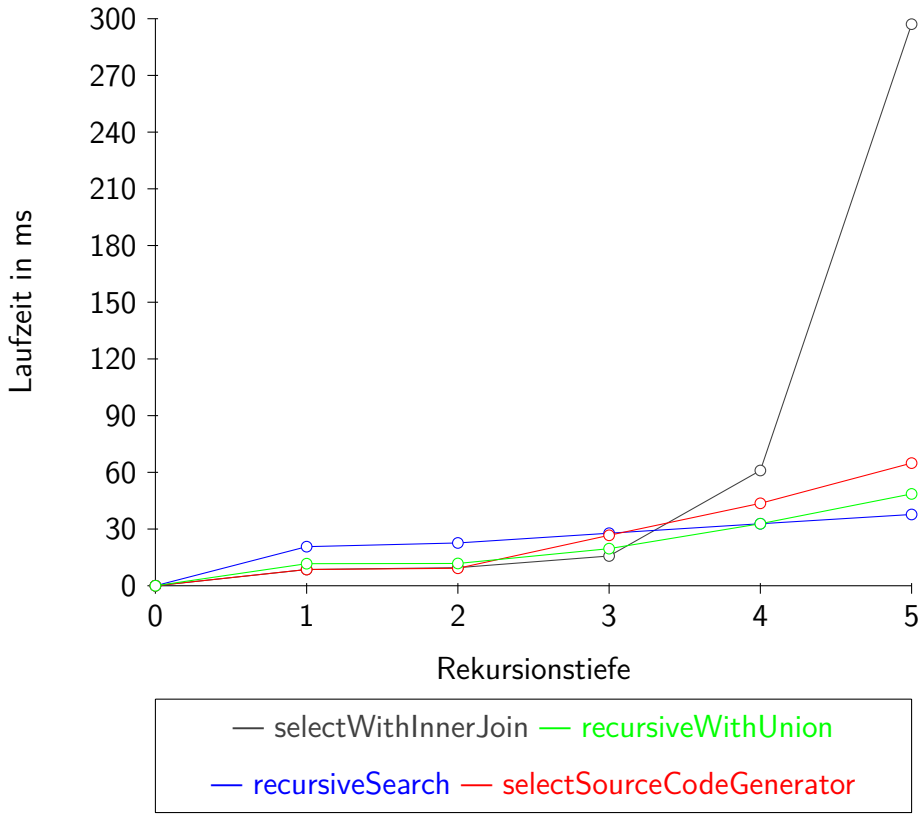


Abbildung 4.8: relation_facebook_with_index

	Laufzeit in MS			
Rerkusions-tiefe	selectSource CodeGenerator	recursiveSearch	recursive WithUnion	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

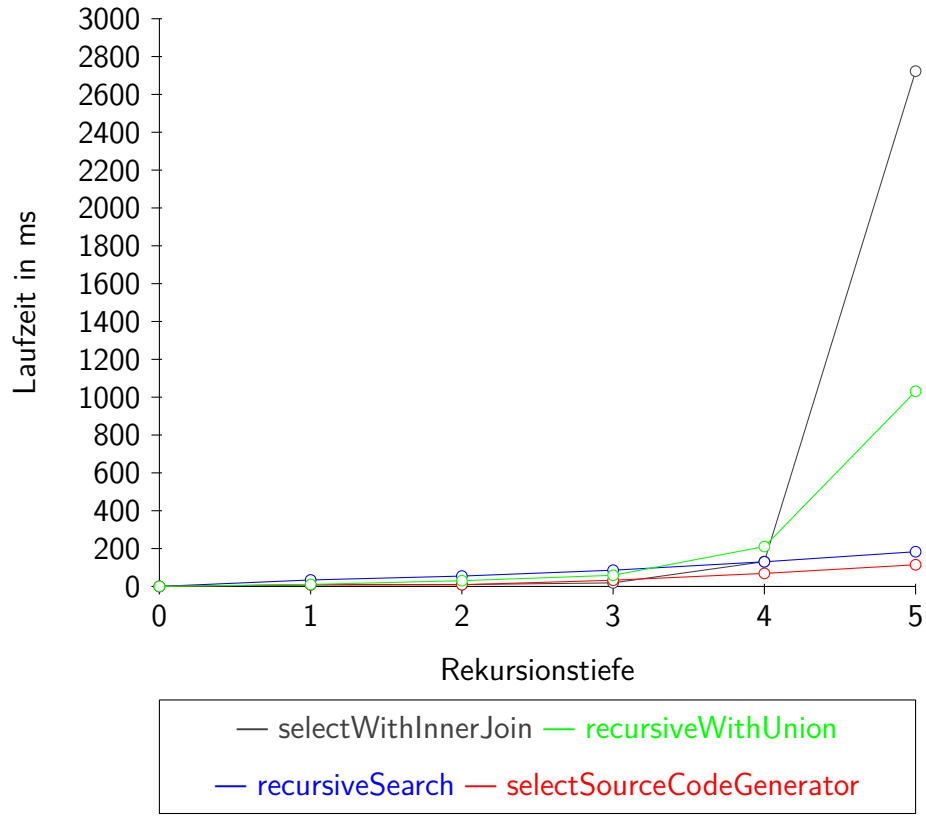


Abbildung 4.9: relation_wiki_vote

	Laufzeit in MS			
Rerkusions-tiefe	selectSource CodeGenerator	recursiveSearch	recursive WithUnion	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

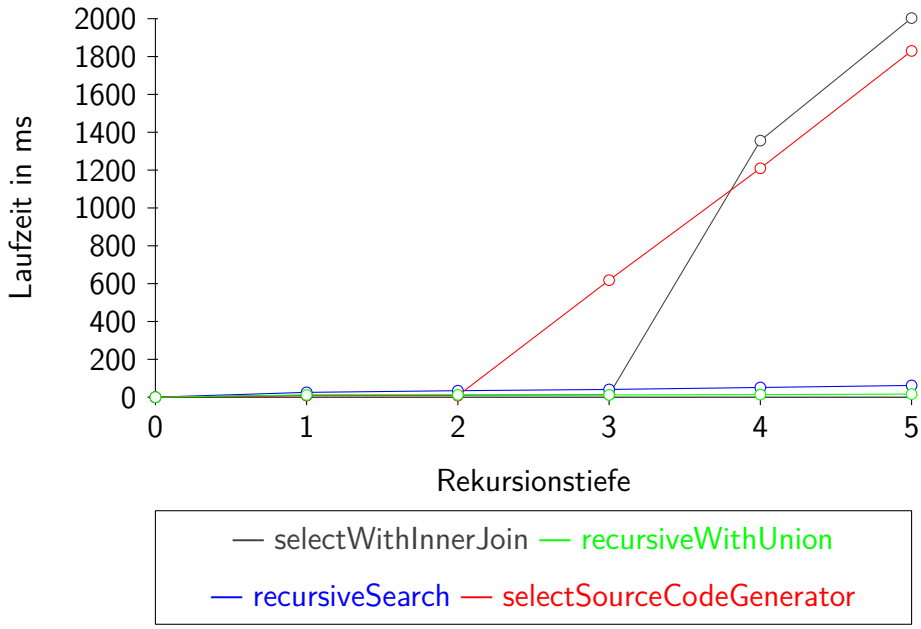


Abbildung 4.10: relation_youtube_with_index

	Laufzeit in MS			
Rerkusions-tiefe	selectSource CodeGenerator	recursiveSearch	recursive WithUnion	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

4.1.2 Antwortzeiten mit partitionierten Tabellen und Indices

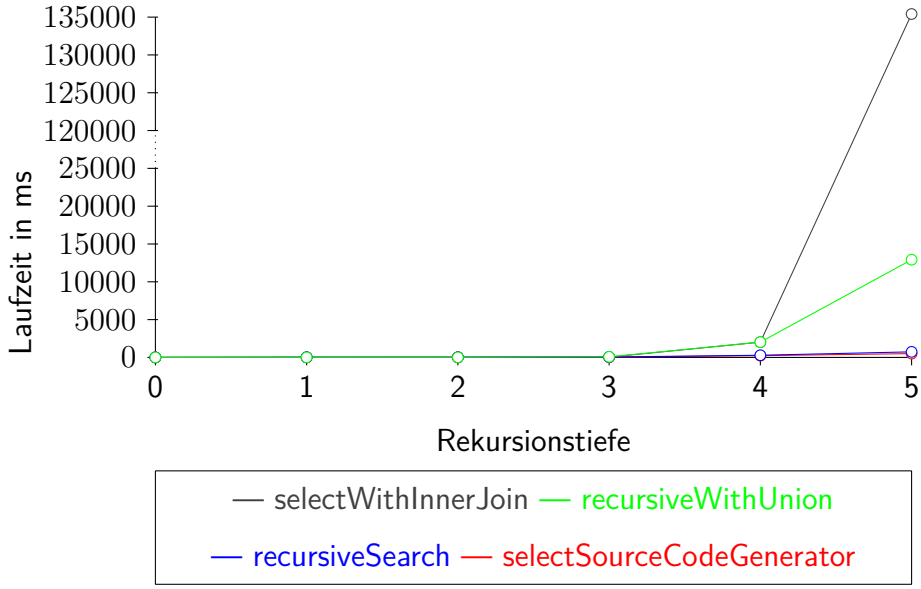


Abbildung 4.11: relation_epinions_partitioned

	Laufzeit in MS			
Rerkusions-tiefe	selectSource CodeGenerator	recursiveSearch	recursive WithUnion	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

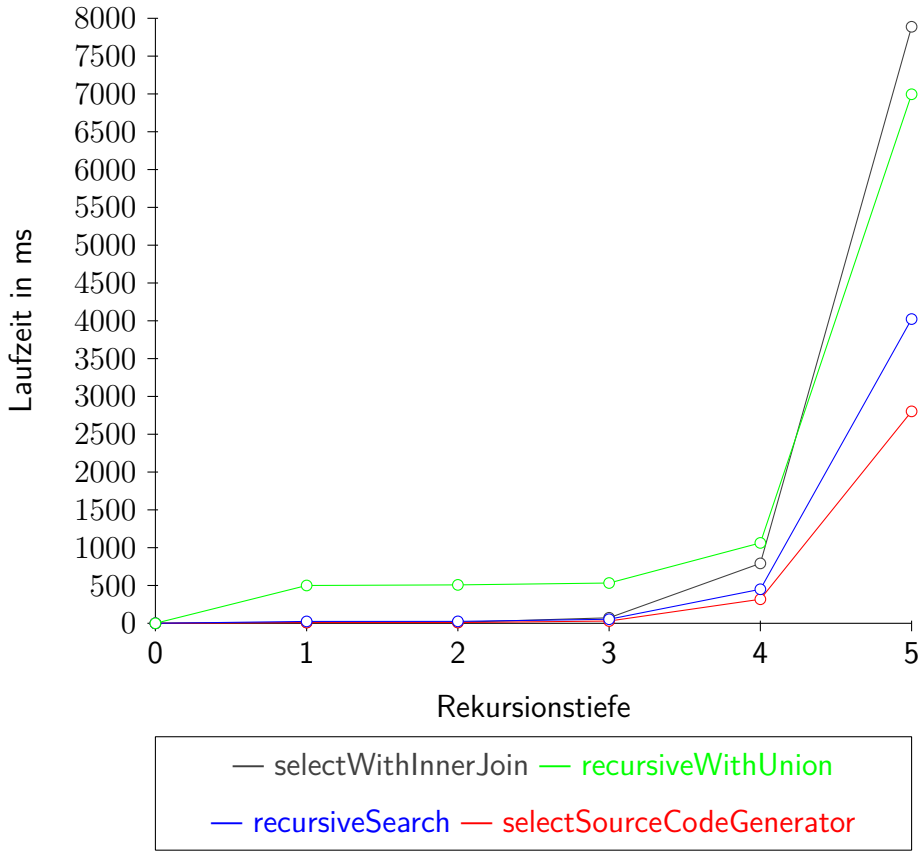


Abbildung 4.12: public_livejournal

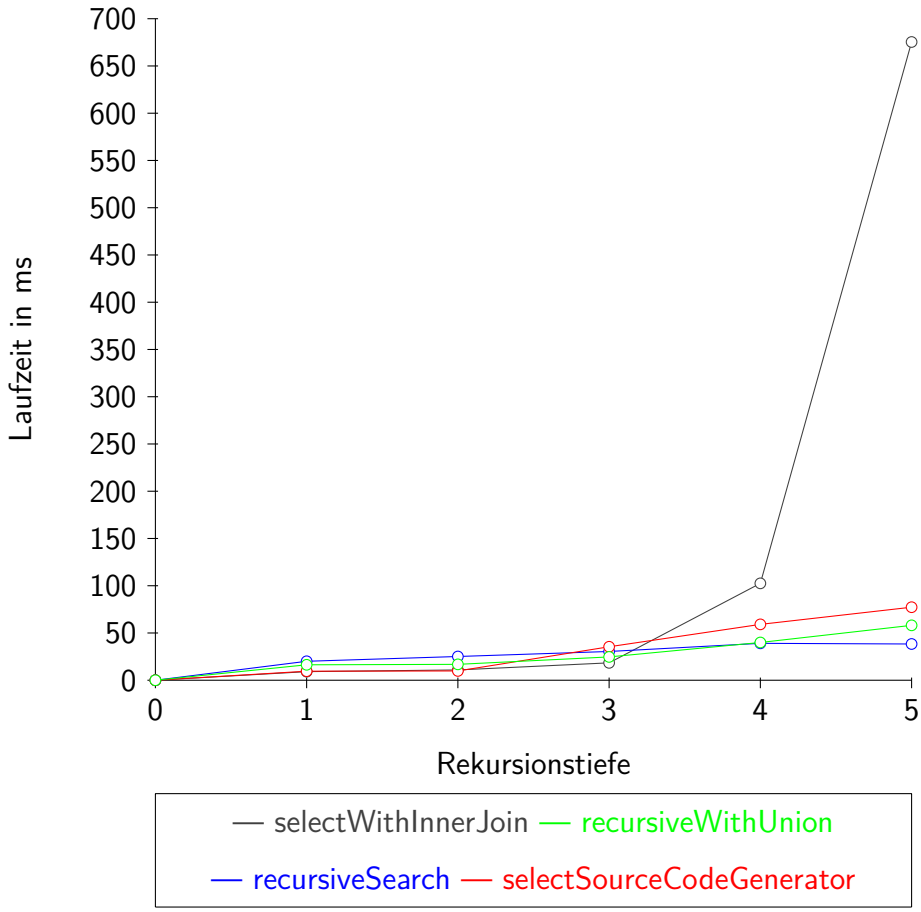


Abbildung 4.13: public_facebook

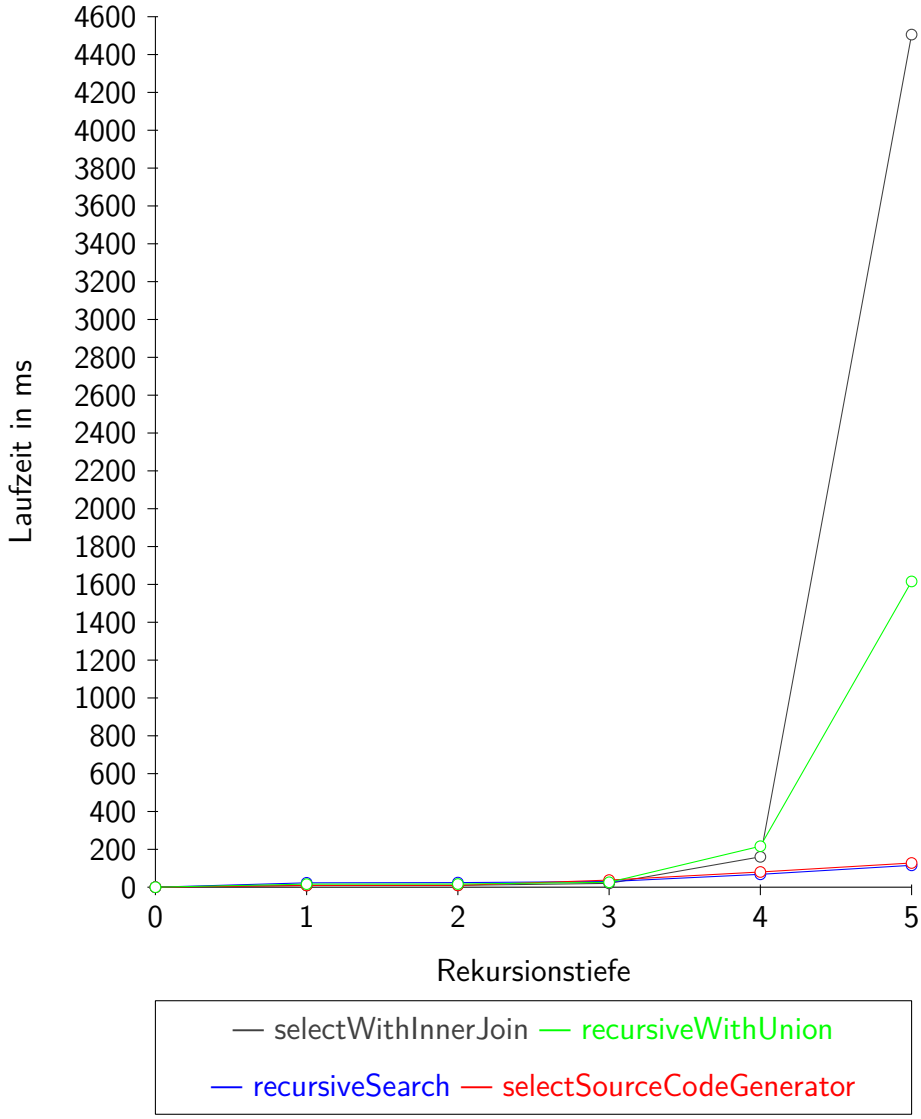


Abbildung 4.14: public_wiki_vote

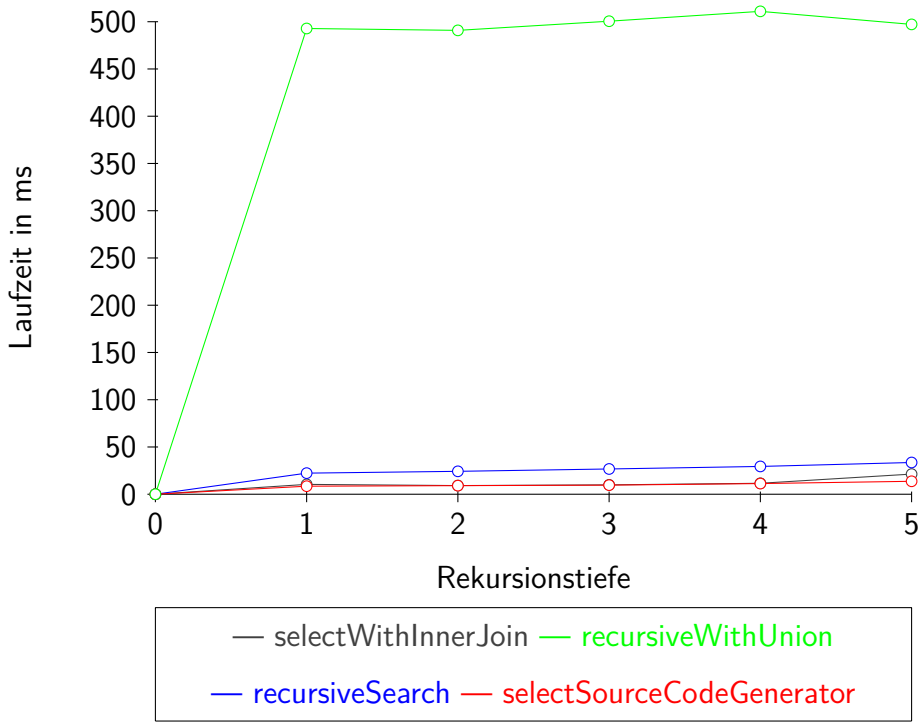


Abbildung 4.15: public_youtube

4.1.3 Standard SQL

4.1.4 Stored Procedures

4.1.5 PL/SQL-Recursion

4.1.6 Datenbankzugriffe

4.1.7 Zugriffsart Aggregation

4.1.8 Zugriffsart Traversierung

4.1.9 Interpretation der Ergebnisse

5 Fazit

5.1 Fazit

A Anhang

A.1 Graph-Datenbanken - Grundlegende technologische Aspekte

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

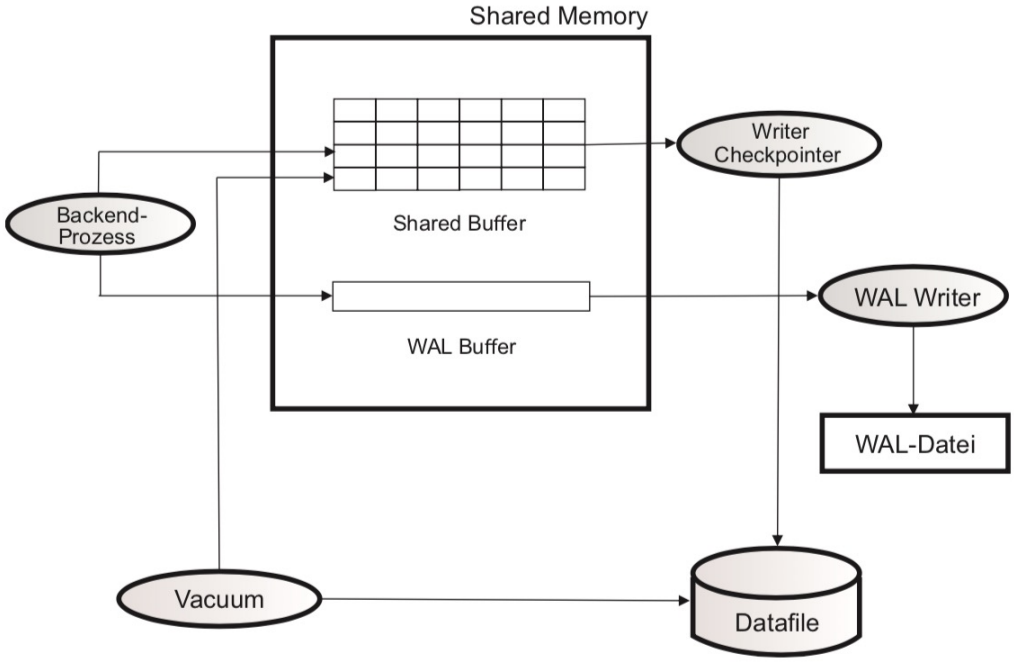


Abbildung A.1: Postgres Architektur

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 wv_part_src ON relation_wiki_vote_partitioned (src);
CREATE INDEX wv_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 wv_src ON relation_wiki_vote_with_index (src);
CREATE INDEX wv_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;
```

A.4 Graph-Datenbanken im praktischen Einsatz: OLAP

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Abkürzungsverzeichnis

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

[Groa] 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*. url=https://www.postgresql.org/docs/10/ddl-partitioning.html

[Groc] Group, The PostgreSQL Global D.: *PostgreSQL 11.1 Documentation*. https://www.postgresql.org/docs/11/queries-with.html

[Grod] 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

[KHA+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

[Kud15] Kudrass, Thomas: *Taschenbuc Datenbanken*. Fachbuchverlag Leipzig im Carl Hanser Verlag, 2015

[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*. Berlin, Heidelberg : Springer, 2017. – ISBN 978–3–319–49475–3

[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+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 15. Januar 2019
Ort, Datum

Jennifer Wittling, Rolf Kimmelmann, Jan Löffelsender