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


### Problemstellung und Erkenntnisinteresse

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.

### Aktueller Forschungsstand

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.

## Zielsetzung

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.

# Inhaltsverzeichnis

# 1 Graph-Datenbanken - Grundlegende technologische Aspekte

## 1.1 Modell

Ein Modell ist eine vereinfachte bzw. abstrahierte Darstellung von realen Gegenständen, Sachverhalten oder Problemen. Durch die Modellierung soll die Realität auf die wichtigsten Einflussfaktoren reduziert werden. Die Graphentheorie spielt eine zentrale Rolle bei der Modellierung von Problemen und Sachverhalten, da sich Graphen sehr gut zur Darstellung vernetzter Daten eignen. Aus diesem Grund haben Graphen in den letzten Jahren auch eine wichtige Rolle in der Datenbankwelt erhalten. Im folgenden sollen verschiedene Arten von Graphen kurz vorgestellt werden.

### 1.1.1 Graph

Ein Graph ist mathematisch folgendermaßen definiert:

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

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

*Ein Knoten $a \in V$ und eine Kante $e \in E$ heißen inzident (incident) genau dann wenn $a$ entweder Anfangs- oder Endecke von $e$ ist. Es gilt $a \in \gamma(e)$. Zwei Knoten $a, b \in V$ heißen adjazent(adjacent) genau dann wenn es eine Kante $e$ gibt die zu $a$ und $b$ inzident ist. Es gilt $\exists e \in E : \gamma(e) = \{a, b\}$.*[1]

Ein Knoten representiert ein Element in einem Graphen. Die Kanten stellen die Beziehung zwischen den einzelnen Knoten her. In einem einfachen Graphen kann eine Kante immer nur jeweils zwei Knoten miteinander verbinden.

[1] [Bec18, Seite 21]

Graphen können gerichtet oder ungerichtet sein. Gerichtete Graphen zeichnen sich dardurch aus, dass die Kanten eine zugewiesene Richtung besitzen. 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. 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. Schlingen und Multikanten dürfen in einem einfachen Graphen nicht auftauchen.[2] Multigraphen hingegen erlauben Multikanten. In Pseudographen sind sowohl Multikanten als auch Schleifen möglich.

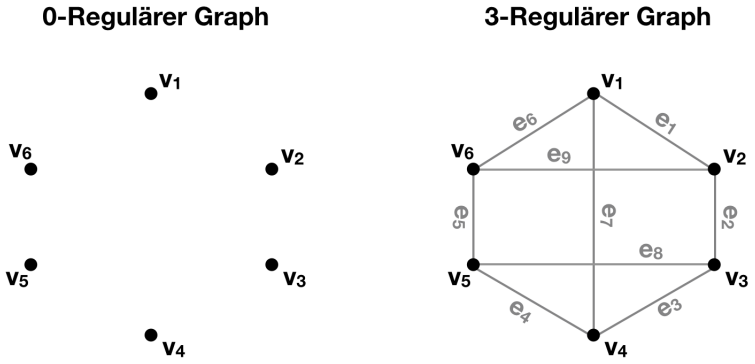
Werden Kanten und Knoten eines Graphs vertauscht entsteht der Kantengraph bzw. Line-Graph des jeweiligen Graphen L(G). Zwei Graphen können isomorph sein. Der Grad eines Knoten bezeichnet die Anzahl der inzidenten Kanten des Knoten. Dabei werden Schleifen doppelt gezählt.[3] Sind bei einem Graphen alle Knoten mit allen übrigen Knoten verbunden spricht man von einem vollständigen Graphen:

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

Ein ungerichteter Graph heißt zusammenhängend, falls zwischen zwei beliebigen Knoten $a$ und $b$ aus $V$ es einen ungerichteten Weg mit $a$ als Startknoten und $b$ als Endknoten gibt.[4]

### 1.1.2 Reguläre Graphen

Bei regulären Graphen haben alle Knoten den selben Knotengrad. Als Knotengrad wird die Anzahl direkter Nachbarn, also alle Knoten die über eine Kante direkt mit dem betrachteten Knoten verbunden sind, bezeichnet.[5] Abbildung 1.1.2 zeigt einen regulären Graphen mit Knotengrad null und einen mit einem Grad von drei.

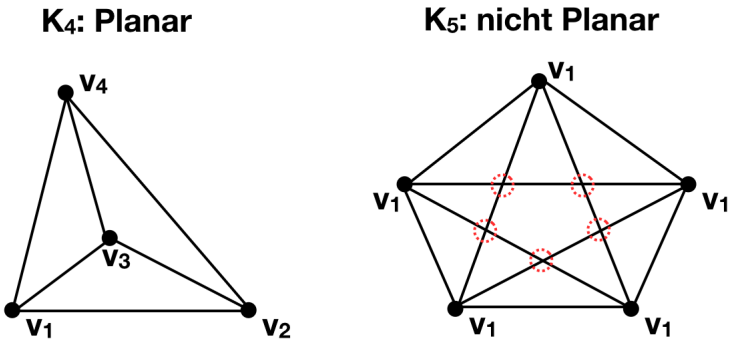



[2] Vgl. [SF05]
[3] Vgl. [Rah17, Seite 13]
[4] [KN12, 36-38]
[5] [Fel03]

### 1.1.3 Planare Graphen

Planare Graphen lassen sich in der Ebene ohne Überschneidung der Kanten zeichnen.[6] Eine Darstellung eines Graphen $G$ in der Ebene ohne Kantenüberkreuzungen wird planare Einbettung von $G$ genannt.

In Abbildung 1.1.3 sind die vollständigen Graphen $K_4$ und $K_5$ abgebildet, wobei es sich bei $K_4$ um einen planaren Graphen und bei $K_5$ um einen nicht planaren Graphen handelt. $K_5$ ist nicht planar, da sich dieser in der Ebene nicht ohne Überschneidungen der Kanten zeichnen lässt. Die Überschneidungen sind in der Abbildung rot markiert.




#### Eulerscher Polyedersatz

Für zusammenhängende planare Graphen besagt der Eulersche Polyedersatz, dass die Anzahl der Knoten minus die Anzahl der Kanten plus die Anzahl der Gebiete zwei ergibt:

$$n - m + f = 2$$

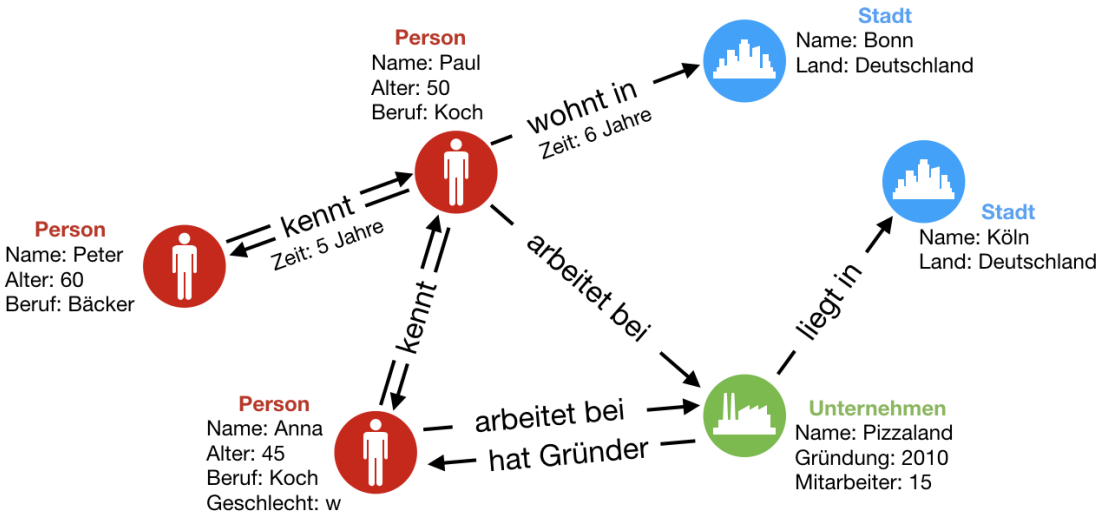
### 1.1.4 Property Graphen

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 1.1.4 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 Attri-

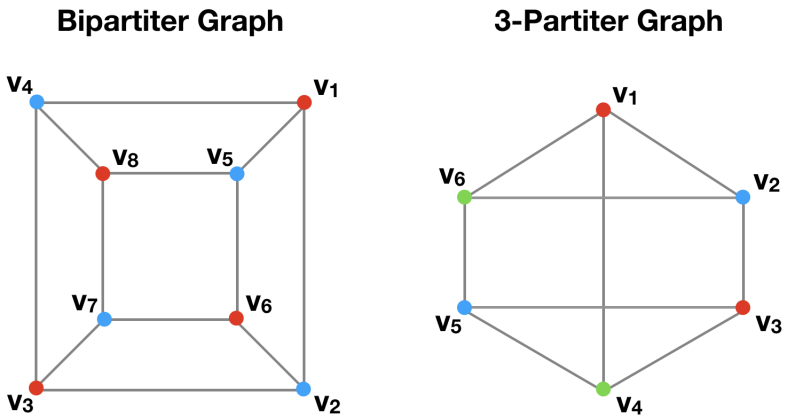
[6] [TT16]

bute, wie beispielsweise der Information über die Dauer der bisherigen Beziehung, genauer definiert werden.




### 1.1.5 k-Partite Graphen

Die Knoten können in k Partitionen unterteilt werden, sodass die Knoten in einer Gruppe keine direkten Nachbarn sind. Für $k = 2$ werden diese Graphen Bipartite Graphen genammt. Die folgende Abbildung 1.1.5 zeigt zwei k-Partite Graphen, wobei die Partitionen durch die unterschiedlichen Farben der Knoten gekennzeichnet sind. Die Darstellung verdeutlicht auch, dass die Anzahl der Knoten eines Graphs keinen direkten Einfluss auf die Anzahl der Partitionen hat.



### 1.1.6 Hypergraphen

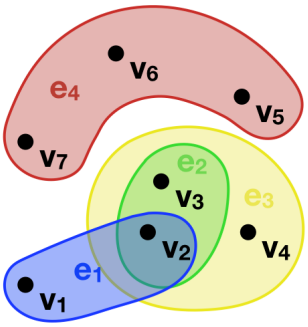
Hypergraphen haben die Eigenschaft, dass Kanten im Gegensatz zu klassischen Graphen mehr als zwei Knoten miteinander verbinden können.
Mathematisch ist ein Hypergraph folgendermaßen definiert:

**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* $H$*.*[7]

Abbildung 1.1.6 zeigt einen Hypergraphen. Die Kante $e_4$ verbindet in diesem Graphen die Knoten $v_5$, $v_6$ und $v_7$ miteinander.




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. In einem normale Graphen sind die Kanten, eine in einem Intervall festgelegte Menge von Knoten:

$$X = \{v_1, v_2, v_3, v_4, v_5, v_6, v_7\} \text{ Knoten}$$

$$E = \{e_1, e_2, e_3, e_4\} \text{ Kanten}$$

$$E = \{e_1, e_2, e_3, e_4\} = \{\{v_1, v_2, v_3\}, \{v_2, v_3\}, \{v_3, v_5, v_6\}, \{v_4\}\}$$

[7] Vgl. [ZSHZ18, Seite 2]

# 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.

### 3.1.1 Traversierung in PostgresSQL

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 5 Methoden vorgesehen:

- Graphtraversierung mit Hilfe von Standard SQL (SELECT RECURSIVE WITH UNION)
- Graphtraversierung mit Hilfe von verschachteltem SELECT Statement
- Graphtraversierung mit Hilfe von rekursiven INNER JOIN
- Graphtraversierung mit Hilfe von selbstgeschriebenen Stored Procedure
- Graphtraversierung mit Hilfe von dynamisch generiertem SQL

### 3.1.2 Installation

PostgreSQL kann unter Ubuntu über die Paketverwaltung Advanced Package Tool (APT) installiert werden. Weiterhin wird eine Installation über die Red Hat Packa-

ge 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 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.

### 3.1.3 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]

### 3.1.4 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).

### 3.1.5 Erstellen von Fremdschlüsseln

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).

### 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 (siehe A.8). Der SQL Befehl WITH RECURSIVE sorgt dafür, dass die Abfrage sich mit der Ergebnismenge wieder selber aufruft. 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 überprüft ob Duplikate vorhanden sind [1] - Duplikate werden gelöscht, die Ergebissmenge wird in die Intermediate Tabelle geschrieben, die Ergebnisse aus der Intermediate Tabelle werden in die Working Tabelle kopiert. Im Vergleich zu den restlichen Methoden garantiert das standard SQL falls man den UNION Operator verwendet, dass Kreise [2] aus dem Graphen entfernt werden. Die Abbruchbedingung für die Rekursion greift, sobald die Working Tabelle keine Einträge mehr hat. Folgende Grafik beschreibt die Rekursion angewendet auf ein relation Tabelle:

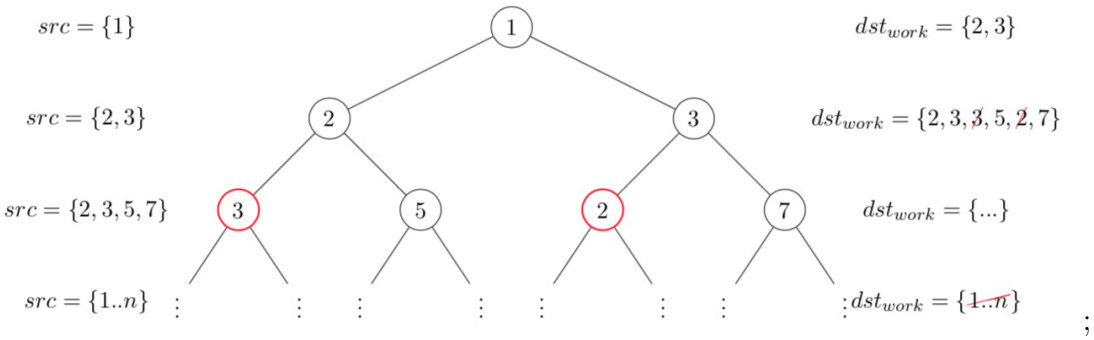



Abbildung 3.1: Traversierung Standard SQL

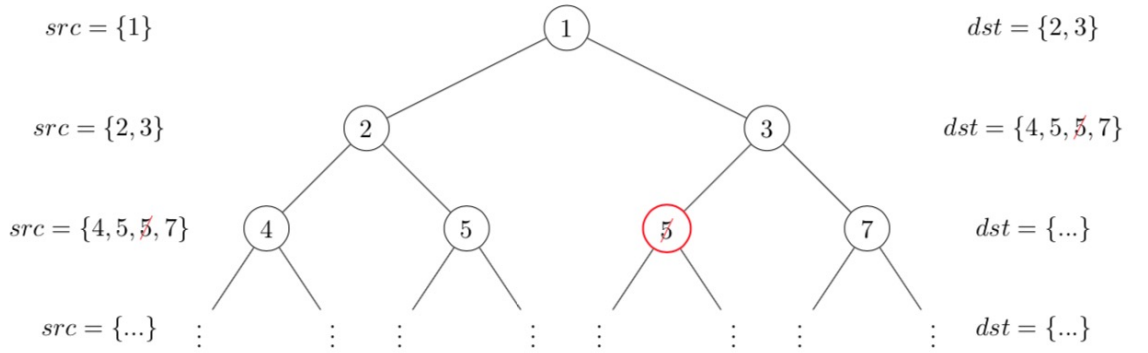
Es wurde sich dafür entschieden den UNION Operator zu verwenden, um Duplikate innerhalb der Working Tabelle auszuschließen. Für die Ergebnismenge der Abfrage auf der relation Tabelle bedeutet dies, dass die Ergebnismenge ungefähr die Anzahl an Profilen haben sollte. Wohingegen die Abfrage mit UNION ALL, das Duplikate aus der Working Tabelle nicht entfernt ([Grob]), eine Ergebnismenge zurückliefert, die genauso viele Elemente hat wie die jeweilige relation Tabelle.

### 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.5). Auf der obersten Rekursionsstufe wird der Startknoten

[1] Die Überprüfung von Duplikaten erfolgt bezogen auf alle vorherigen Rekursionsstufen
[2] Ein Weg, bei dem der Anfangsknoten gleich dem Endknoten ist [Bec18, S.48]

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 3.2: 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.

### 3.1.8 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.6).

### 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.7). Die Abbruchbedingung wird dem Stored Procedure in Form einer Rekursionsstiefe mitgegeben. In jeder Rekursionsstufe erstellt das Script 2 temporäre Tabelle. Eine temporäre Tabelle [3] 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.

[3] 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])

# 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.

**selectSourceCodeGenerator**

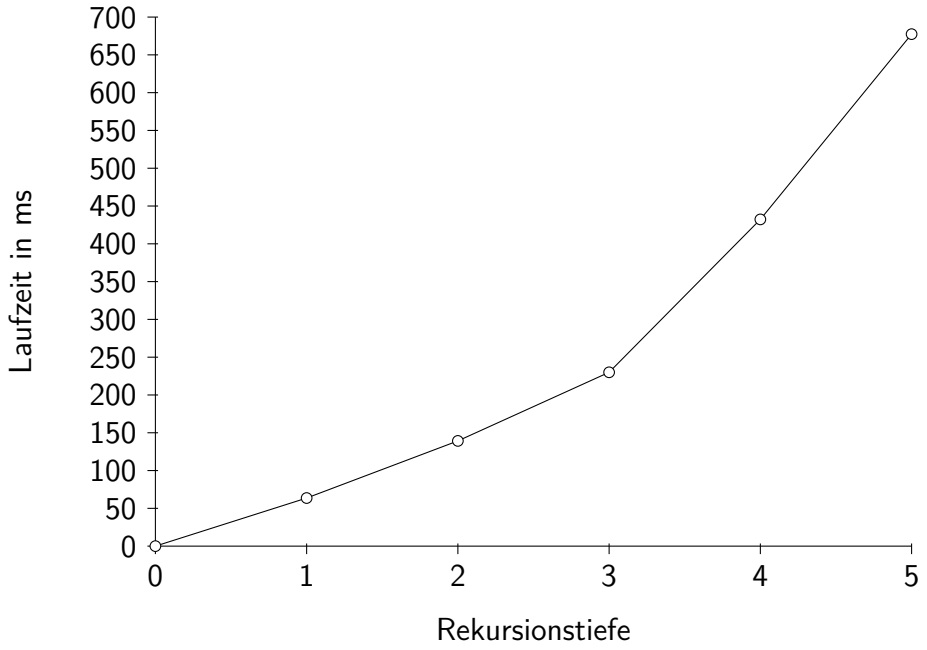



Abbildung 4.1: public_epinions

| Rerkusionstiefe | Laufzeit in ms |
|---|---|
| 1 | 63.636 |
| 2 | 139.225 |
| 3 | 229.997 |
| 4 | 432.297 |
| 5 | 677.420 |

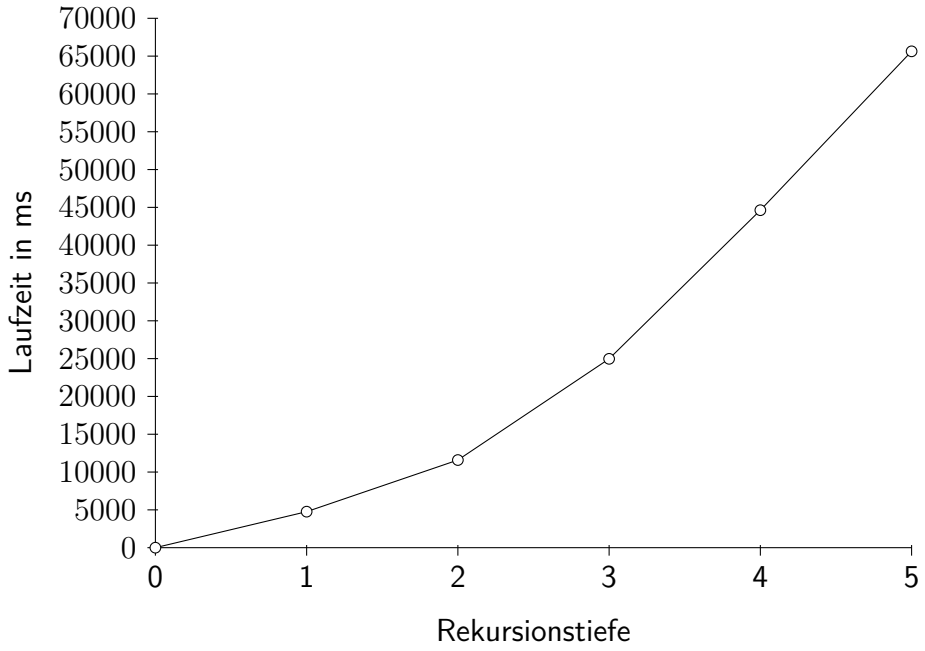



Abbildung 4.2: public_livejournal

### 4.1.2 Standard SQL

### 4.1.3 Stored Procedures

### 4.1.4 PL/SQL-Recursion

### 4.1.5 Datenbankzugriffe

### 4.1.6 Zugriffsart Aggregation

### 4.1.7 Zugriffsart Traversierung

### 4.1.8 Interpretation der Ergebnisse

# A Anhang

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

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

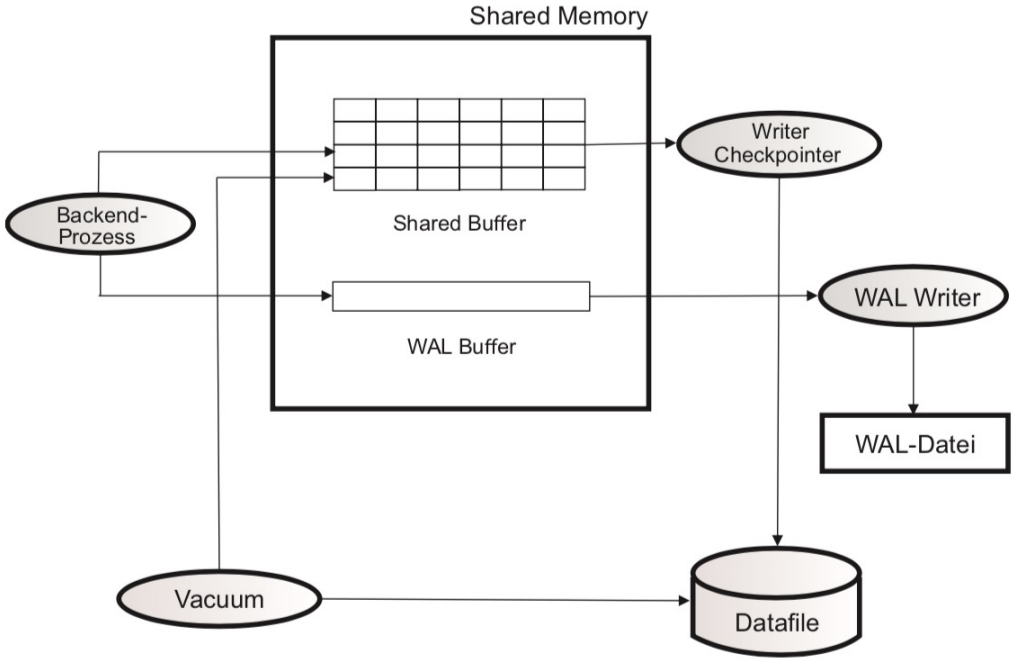



Abbildung A.1: Postgres Architektur

## 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: 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.6: 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.7: 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.8: SQL Standard SQL Mittel

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

## 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 |
| **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


[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

[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

[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*. `https://www.postgresql.org/docs/11/queries-with.html`

[Groc] 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

[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

[Rah17] 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

[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 6. Januar 2019*
Ort, Datum

Jennifer Wittling, Rolf Kimmelmann, Jan Löffelsender