Friedrich-Alexander-Universität Erlangen-Nürnberg
Technische Fakultät, Department Informatik

YU HEYDEMANN

BACHELOR THESIS

PRODUCT ARCHITECTURE MODELING WITH POSTGRESQL AND GRAPHQL

Eingereicht am 15. Dezember 2020

Betreuer:
Andreas Bauer,M.Sc.
Prof. Dr. Dirk Riehle, M.B.A.
Professur für Open-Source-Software
Department Informatik, Technische Fakultät
Friedrich-Alexander-Universität Erlangen-Nürnberg

Versicherung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, 15. Dezember 2020

License

Erlangen, 15. Dezember 2020

Abstract

In today's software development, Free / Libre and Open Source Software (FLOSS) is becoming more and more common used. The management of complex dependencies between the open source components used is an essential part of this. Without such management, license compliance will be impossible, export restrictions will not be upheld, and security vulnerabilities will remain unknown to vendors. The Product Model Toolkit helps you to manage third-party open source dependencies. It combines a product's architecture with licence compliance information of each incorporated component, to derive license compliance relevant artifacts. But is is unclear how such a graph of components should be represented in a relational database.

In this thesis we design a PostgreSQL database schema that can represent components and its dependencies to other components, as well as additional metadata for a component or dependency. It tested it using a GraphQL interface, where PostGraphile provides the GraphQL interface for the data.

Zusammenfassung

In der heutigen Software-Entwicklung werden Freie / Libre und Open Source Software (FLOSS) immer häufiger eingesetzt. Das Management komplexer Abhängigkeiten zwischen den verwendeten Open-Source-Komponenten ist dabei ein wesentlicher Bestandteil. Ohne ein solches Management ist die Einhaltung von Lizenzen nicht möglich, Exportbeschränkungen werden nicht eingehalten und Sicherheitslücken bleiben den Anbietern unbekannt. Das Product Model Toolkit hilft Ihnen bei der Verwaltung von Open-Source-Abhängigkeiten von Third-Party Produkten. Es kombiniert die Architektur eines Produkts mit Informationen zur Lizenz-Konformität, um lizenzkonformitätsrelevante Artefakte abzuleiten. Es ist jedoch unklar, wie ein solcher Graph von Komponenten in einer relationalen Datenbank dargestellt werden soll.

In dieser Arbeit entwerfen wir ein PostgreSQL-Datenbankschema, das Komponenten und ihre Abhängigkeiten zu anderen Komponenten, sowie zusätzliche Metadaten für eine Komponente oder eine Abhängigkeit darstellen kann. Es wird mit einer GraphQL-Schnittstelle getestet, wobei PostGraphile die GraphQL-Schnittstelle für die Daten bereitstellt.

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Die menschliche Gesellschaft wird zunehmend digital, wobei Software unverzichtbar geworden ist. Es ist eine offensichtliche Tatsache, dass die Menschen ohne jede Art von Software leben und arbeiten. In der heutigen Zeit ist moderne Software nicht mehr auf einzeiligen Code angewiesen, egal wie neu und einzigartig eine ausgereifte Anwendung ist, die zu 80% aus Repositories oder Komponenten von Third Party Anbietern stammt. Komponenten von Third Party Anbietern (kommerziell lizenzierte,proprietäre und quelloffene Software) und FLOSS-Komponenten liefern die notwendigen Bausteine, die es Unternehmen ermöglichen, Werte zu liefern, die Qualität zu verbessern, Risiken zu verringern und die Zeit bis zur Markteinführung zu verkürzen. Es besteht kein Zweifel, dass sich FLOSS zu einem strategischen Vorteil in der Softwareentwicklung etabliert hat. Ein Wechsel von kommerzieller Software zu Open Source Software beeinflusst schon die Unternehmensstrategie im IT-Bereich allgegenwärtig, und die kommerzielle Nutzung von FLOSS nimmt auch stetig zu, mit einem geschätzten Anteil von 95% aller kommerziellen Software FLOSS. Immer mehr Unternehmen haben die Vorteile der Verwendung von FLOSS-Komponenten in ihren Produkten erkannt.

Dies wirft eine dringende Frage auf. Ist es bekannt, welche in den von den Unternehmen verwendeten und entwickelten Anwendungen verwendet werden, auf dem neuesten Stand sind und alle wichtigen Sicherheitspatches installiert sind? Dies muss ein Dilemma sein, wenn wir nicht einmal wissen, welche Software unser Unternehmen verwendet und mit welchen Systemen entwickelt wird. Wie könen wir dann über die Installation von Sicherheitspatches für diese Systeme sprechen. Der Grund dafür ist, dass die von vielen Unternehmen verwendeten FLOSS-Komponenten keine genaue, umfassende und aktuelle Liste der Vermögenswerte führen. Daher besteht der erste Schritt zu einer guten Komponentensicherheit darin, eine klare Liste der Komponenten zu haben und diese automatisch im Hintergrund in Echtzeit zu pflegen. Wir konzentrieren uns nicht nur auf extern eingeführten Code, sondern unterscheiden auch zwischen Versionen und Verwend-

barkeiten von kommerziellen, FLOSS- und internen Komponenten. Eine besonderes Augenmerk muss auf das Problem der Abhängigkeiten der im Produkt verwendeten Open-Source-Komponenten gelegt werden. Ohne dieses Management wäre die Einhaltung von Lizenzen unmöglich, die Exportbeschränkungen könnten nicht aufrechterhalten werden, und Sicherheitslücken blieben dem Anbieter unbekannt.

Mit dem Product Model Toolkit[1]ist es möglich, Metadaten zu speichern und zu verwalten, die sich auf die Architektur eines Produkts in Bezug auf die Lizenzkonformität beziehen. Product Model Toolkit (PMT) ist eine Open-Source-Software, die an der Professur für Open-Source-Software der Friedrich-Alexander-Universität Erlangen-Nürnberg entwickelt wird. Das PMT verwendet derzeit PostgreSQL als Datenbank zur Speicherung aller gesammelten Informationen über ein Softwareprodukt.

1.2 Ziel der Arbeit

Mit dem Product Model Toolkit wird es möglich sein, die Architektur eines Produktes in Kombination mit lizenzkonformitätsrelevanten Metadaten zu speichern. Es verwendet PostgreSQL zur Speicherung der Daten und PostGraphile zur Bereitstellung einer GraphQL-Schnittstelle für die Daten. Es ist jedoch unklar, wie das Datenbankschema aussehen soll, um Informationen über die Komponenten eines Produkts als Graph und die zugehörigen Metadaten abzuleiten. In dieser Arbeit soll der Student ein Datenbankschema entwerfen, das einen Komponentengraphen darstellen kann, und diesen mit der GraphQL-Schnittstelle testen.

1.3 Aufbau der Arbeit

Insgesamt beinhaltet die vorliegende Arbeit 7 Kapitel, die sich wie folgt aufteilen. Nach dieser Einleitung werden in Kapitel 2 die Anforderungen an diese Arbeit festgelegt. Im Anschluss werden in Kapitel 3 die Grundlagen, welche zum Verständnis der Arbeit benötigt werden, erläutert. Kapitel 4 stellt drei mögliche Varianten für diese Bachelorarbeit dar. Das für die Umsetzung entwickelte Konzept wird in ihnen auch dargestellt und erläutert. Es wird darauf eingegangen, warum die Entscheidung für die entsprechende Variante des Konzepts getroffen wurde. Die mit dem Konzept durchgeführte Implementierung wird in Kapitel 5 erläutert. Die Bewertung der so erstellten Methoden findet in Kapitel 6 statt. Zum Abschluss wird in Kapitel 7 ein Fazit gezogen und ein Ausblick für Zukunft gegeben.

[1]https://github.com/osrgroup/product-model-toolkit

2 Anforderungen

In diesem Kapitel sollen die Anforderungen an eine GraphQL-Schnittstelle, sowie das zugrundeliegende relationale Datenbankschema, erläutert werden.

2.1 Relationales Datenbankschema zur Speicherung von Graphen

A1: Relationale Abbildung von Graphen

Es ist ein Datenbankschema zu entwerfen, welches es ermöglicht einen Komponetengraphen zu speichern und zu lesen. Hierbei besteht der Komponenetengraph aus mindestens den folgenden Attributen:

- **Der Komponente selbst.** Oder eine Referenz zu einer Komponente.
- **Der Beziehung zwischen Komponenten.** Welche Abhängigkeiten hat eine Komponente?
- **Die Eigenschaften einer Beziehung zwischen Komponenten.** Wie hängt eine Komponente von einer anderen ab? Beispielsweise wird oft zwischen statischem und dynamischen Linking von Komponenten unterschieden.

A2: Kompatibilität zu PostGraphile

Die eigentliche Bereitstellung der GraphQL-Schnittstelle erfolgt durch die Software PostGraphile. Das heißt es muss die Kompatibilität des entworfenen Datenbankschemas mit PostGraphile sichergestellt werden.

2.2 GraphQL-Schnittstelle

A3: CRUD-Operation

Die vier grundlegenden Operation des Datenzugriff Create, Read, Update und Delete (CRUD) bilden die minimale Menge an Operationen, die für eine Res-

source bereitgestellt werden sollten. Die zu entwickelnde GraphQL-Schnittstelle muss die Anwendung von CRUD-Operationen auf Komponenten des Komponentengraphen ermöglichen. Im Konkreten sind dies:

- Erstellung einer neuen Komponente
- Lesen einer existierenden Komponente und dessen Attribute
- Update einer Komponente und dessen Attribute
- Löschen einer Komponente

A4: Querying des gesamten Komponentengraphens

Der Komponentengraph soll als Ganzes durch die GraphQL-Schnittstelle ausgelesen werden können. Dies beinhaltet alle Komponenten, die Beziehungsinformation zwischen den Komponenten, sowie die Attribute der Komponenten.

A5: Querying eines Teils des Komponentengraphens

Ergänzend zu A2, soll es auch möglich sein, nur einen Teil des Komponentengraphens auslesen zu können. Ein Beispiel hierfür ist das Auslesen einer spezifischen Komponente und all ihrer abhängigen Komponenten.

A6: Querying von Lizenzinformation

Die GraphQL-Schnittstelle soll es ermöglichen, alle Komponenten welche eine bestimmte Lizenz haben, zu identifizieren und zurückzugeben.

3 Grundlagen

In diesem Kapitel wird ein Überblick über die Grundlagen dieser Arbeit gegeben. Dabei wird zunächst auf die relationalen Datenbanken eingegangen, um anschließend einen Überblick über PostgreSQL zu geben. Es werden auch die Grundlagen der Graphentheorie beschrieben. Schließlich wird die Abfragesprache GraphQL eingeführt.

3.1 Grundlagen relationaler Datenbanksysteme

3.1.1 Einführung in relationale Datenbanken

Seit 1970, als Edgar F. Codd, ein Mathematiker und ein Forscher am San Jose Research Laboratory von IBM, erstmals die relationale Modell von Datenbanksystemen vorschlug[17], haben sich relationale Datenbanksysteme im Laufe der Jahrzehnte rasant entwickelt und großartige Ergebnisse erzielt. RDMBS (Relationale Datenbankmanagementsysteme) ist heute die Haupttechnologie zur Speicherung strukturierter Daten. Diese Datenbanksysteme werden allgemein als SQL-Datenbanken bezeichnet, die auf der verwendeten Structured Query Language (SQL) basieren. Die derzeit wichtigsten relationalen Datenbanken sind My SQL, Microsoft Access, Oracle undPostgreSQL.

Die Daten im relationalen Modell werden im Allgemein durch eine Datenbanktabelle (Schema) dargestellt, in der Objekte desselben Typs (das heißt dieselbe Anzahl von Attributen desselben Typs) in einer Tabelle kombiniert werden, um die Daten zu strukturieren. Eine relationale Datenbank speichert Daten in Zeilen, und jede Zeile hat die gleiche Anzahl und Art von Spalten. Darüber hinaus werden die Daten in der Regel normalisiert, was zur Erstellung mehrerer Tabellen führt. Die Abfrage von über mehrere Tabellen verteilten Daten erfordert das Lesen und Integrieren von Informationen aus einer oder mehreren verschiedenen Tabellen. Der Prozess der Integration von Informationen basiert auf dem Abgleich der Primär- und Fremdschlüsselwerte mehrerer Tabellen in einer relationalen Datenbank, ein Prozess, der als Join (verbundene) Tabellen bezeichnet wird.

Lese- und Schreiboperationen auf traditionellen relationalen Datenbanken sind transaktional und haben ACID-Eigenschaften. Transaktion definiert Regeln für eine Reihe von Transaktionen, um die Integrität der Daten in der Datenbank sicherzustellen. ACID ist die Abkürzung für die vier Grundelemente der korrekten Ausführung von Datenbanktransaktionen, zu denen folgende gehören: Atomarität (Atomicity), Konsistenz (Consistency), Isolierung (Abgrenzung) und Persistenz (Dauerhaftigkeit). Sie sind in den folgenden vier Erklärungen kurz zusammengefasst [13]:

- **Atomicity(Atomarität)**

  Abschließen alle Vorgänge ab, um eine Transaktion abzuschließen. Wenn beispielsweise das Einfügen aller zugehörigen Objekte abgeschlossen ist, wird das Einfügen eines Objekte als abgeschlossen betrachtet, andernfalls wird es auf den Status vor dem Start der Transaktion zurückgesetzt (Rollback), als ob die Transaktion nie ausgeführt worden wäre.

- **Consistency(Konsistenz)**

  Eine Transaktion kann die Datenbank nicht zum Absturz bringen. Dies bedeutet, dass vor dem Start der Transaktion und nach dem Ende der Transaktion die Integritätsbeschräkungen der Datenbank nicht zerstört werden. Wenn beispielsweise beim Einfügen eines Objekts ein oder mehrere verwandte Objekte nicht in der Datenbank gefunden werden können, wird der Vorgang als unzulässig angesehen. In diesem Fall wird ein Rollback durchgeführt.

- **Isolation(Abgrenzung)**

  Alle Transaktionen sind unabhängig voneinander und können sich nicht gegenseitig beeinflussen. Wenn beispielsweise eine Zeile in einer Tabelle aktualisiert wird, wirkt sich dies nicht auf das Einfügen anderer Zeilen in derselben Tabelle aus.

- **Durability(Dauerhaftigkeit)**

  Wenn eine Transaktion begangen wurde, dann kann sie nicht rückgängig gemacht werden. Wenn beispielsweise ein Objekt gelöscht wird, kann die Aktion nicht rückgängig gemacht werden.

Ein weiteres Merkmal einer relationalen Datenbank ist es, dass sie eine feste tabellarische Struktur hat. Das Funktionsprinzip wird in 3.1.2 erläutern.

3.1.2 Funktionsprinzip relationaler Datenbanken

Um das Funktionsprinzip relationaler Datenbanken zu verstehen, müssen die grundlegenden Begriffe dieses Datenbankmodells bekannt sein. Bei einer rela-

tionalen Datenbank handelt sich um eine Ansammlung von Tabellen mit Datensätzen. Jede Tabelle ist auch eine Relation. Jede Zeile wird auch als Tupelbezeichnet, die einen Datensatz darstellt. Eine Datenbank muss konsistent und auch redundanzfrei sein. Das heißt, jeder Datensatz muss immer eindeutig identifizierbar sein, was Mithilfe eines oder mehrerer Schlüssel (Primärschlüssel oder Fremdschlüssel) sichergestellt wird.

- **Primärschlüssel**

  Er kann innerhalb einer Tabelle nur einmal verwendet werden und dient zur Identifikation der Tupel; die Schlüsselwerte dürfen sich nicht ändern.

- **Fremdschlüssel**

  Er ist ein Verweis auf einen Primärschlüssel einer anderen Tabelle oder innerhalb derselben Tabelle.

In der Abbildung 3.1 wird eine Relation mit dem Namen "Kundendaten"beispielsweise wie folgt aussehen:

Ein Datensatz wird als Tupel bezeichnet und besteht aus mehreren Datenfeldern wie zum Beispiel dem Vornamen (Vname) und dem Nachnamen (Nname). Jeder Datensatz muss über einen Primärschlüssel eindeutig identifizierbar sein. Zum Beispiel mit einer eindeutigen Kundennummer (Kd-Nr).

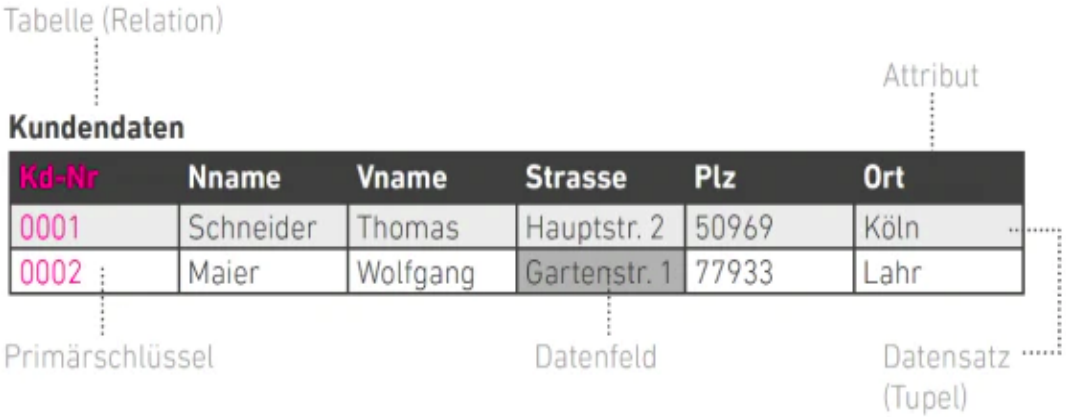

Abbildung 3.1: Beispiel Tabelle für Kundendaten[18]

3.1.3 PostgreSQL

PostgreSQL ist ein objektrelationales Datenbankmanagementsystem (ORDBMS). Es verfügt über eine unterstützende Klasse, Vererbung und anderen Funktionen.

Das System ist eine voll funktionsfähige relationale OSS-Datenbank, die in mancher Hinsicht sogar kommerzielle Datenbanken übertrifft. PostgreSQL wurde ursprünglich 1986 von Professor Michael Stonebraker der University of California in Berkeley gegründet. Professor Stonebraker veröffentlichte 1989 die erste Version[6]. Nach der vierten Version wurde Projekt PostgreSQL offiziell beendet. Obwohl Projekt PostgreSQL beendet wurde, ermöglichte die BSD-Lizenz Liebhabern von OSS, die Software weiterzuentwickeln.

PostgreSQL kann auf fast allen verschiedenen ORDBMS-Plattformen ausgeführt werden, in mancher Hinsicht sogar in der Nähe von Datenbanken auf Unternehmensebene. PostgreSQL unterstützt asynchrone Replikation, vollständige SQL-Standardunterstüzung, Tabellenbereichsmechanimus und Parallelitätskontrolle für mehrere Versionen. PostgreSQL lässt sich im Big-Data-Management leicht erweitern und bietet eine hervorragende Leistung bei Mehrbenutzer-Parallelität. PostgreSQL hat viele effektive Eigenschaften:

- freie Lizenz
- Plattformunabhängigkeit
- hohe Sicherheit durch Security-Features

3.2 Grundlagen Graphentheorie

In diesem Abschnitt werden die Grundlagen der Graphentheorie behandelt, indem wichtige Darstellung und verschiedene Arte von Graphen gezeigt werden.

Die Graphentheorie ist ein Zweig der Mathematik. Sie verwendet Graphen als Forschungsobjekt. Die Graphentheorie hat ihren Ursprung in dem berühmten Königsberger Brückenproblem (Eulerkreisproblem) vom Mathematiker Leonhard Euler[12].

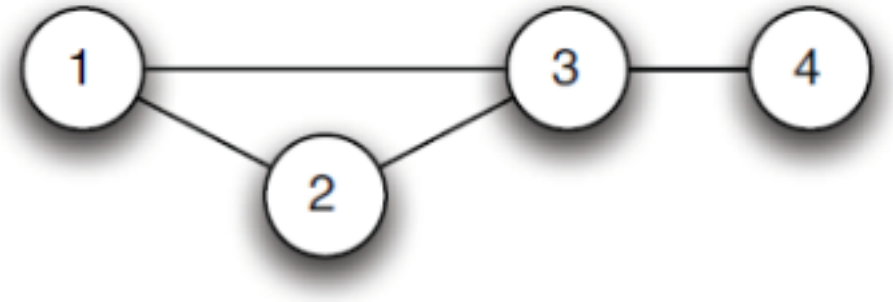

Abbildung 3.2: Darstellung eines Graphen mit vier Knoten und vier Kanten[2]

Im Allgemeinen besteht ein Graph aus zwei Elementen: den Knoten und deren Verbindungen(Kanten), die jeweils zwei Knoten verbinden. Allgemein wird ein **Graph *G= (V,E)*** in einer mathematische Struktur durch eine Menge von Knoten $(v_1, \ldots, v_n)$ und eine Menge von Kanten $(e_1, \ldots, e_n)$, die jeweils zwei Knoten (v_i, v_j) *2* V miteinander oder einen Knoten mit sich selbst verbinden, dargestellt. Durch Graphen lassen sich Objekte als Knoten und ihre Beziehungen untereinander als Kanten zwischen den Knoten darstellen.

Grundsätzlich kann zwischen gerichteten und ungerichteten Graphen unterschieden werden. Dabei unterscheiden sich die Graphen in der Eigenschaft der jeweiligen Kanten zwischen den Knoten. Abbildung 3.3 zeigt die Unterschiede beider Arten von Graphen.

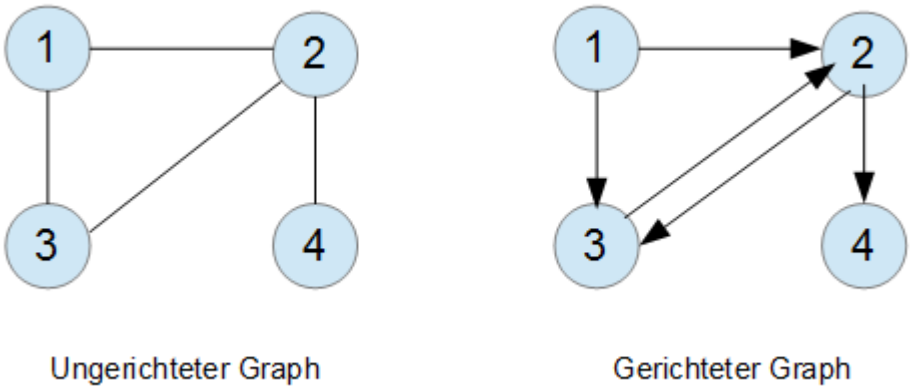

Abbildung 3.3: Ungerichteter und Gerichteter Graph[7]

- **Gerichtete Graphen**

  Die Kanten, welche die einzelnen Knoten verbinden, sind als Pfeil dargestellt. Somit kann die Kante nur entsprechend der Pfeilrichtung genutzt werden.

- **Ungerichtete Graphen**

  Hierbei können die Kanten des Graphen als Verbindung zwischen den Knoten in beide Richtungen genutzt werden.

- **Multigraphen**

  Sind zwei Knoten durch mehrere Kanten verbunden, so spricht man von "Mehrfachkantenünd nennt die zugehörigen Graphen Multigraphen[15]. Ein Graph ohne Mehrfachkanten wird als ëinfacher Graph"bezeichnet.

3.3 Software Componente

Moderne Softwaresysteme, insbesondere große Unternehmenssysteme, werden oft immer komplexer, erfordern dabei jedoch erhöhte Flexibilität. Die Anwendungen werden auf der Grundlage ihres funktionalen Verhaltens in wichtige Teile zerlegt und aus unabhängig voneinander arbeitenden Teilen zusammengebaut, die als Komponenten bezeichnet werden.

Das dieser Bachelorarbeit zugrundeliegende Komponentenmodell wurde an der Professur für Open Source Software an der Friedrich-Alexander-Universität Erlangen-Nürnberg entwickelt und heißt Product Model Toolkit. Mit dem Product Model Toolkit können Sie Open Source-Abhängigkeiten von Third-Party Anbietern in ihrem Produkt verwalten.

Durch die Kombination von Lizenzinformationen und Architekturinformationen in einem gemeinsamen Modell, wird eine umfassende Betrachtung eines Produktes ermöglicht [4]. Der Ausgangspunkt dieses Modells ist das Softwareprojekt selbst. Ein Projekt kann mehrere verschiedene Stammkomponenten enthalten, die wiederum von mehreren verschiedenen anderen Komponenten abhängen können. Diese Abhängigkeitsbeziehungen zwischen Komponenten können unterschiedlicher Art sein (z.B. dynamisch oder statisch verknüpft). Es wird auch hier als linking benannt. Es ist eine Eigenschaft(Attribut) der Beziehung. Die Komponenten bilden immer eine hierarchische Struktur, die einen azyklis6chen Graphen darstellt. Jede Komponente enthält Informationen über das Software-Artefakt, das sie repräsentiert und eine Liste von Metadaten, wie z.B. Name-, Lizenz-, Package- und Linkingdaten. Die Abbildung 3.4 zeigt die Beziehung zwischen Komponenten (Abhängigkeiten).

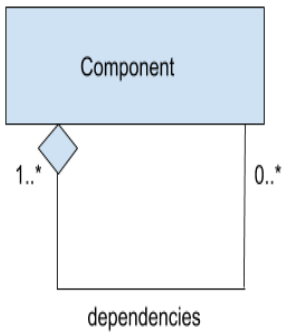

Abbildung 3.4: Darstellung von Component(Dependencies)

3.4 GraphQL

GraphQL wurde von Facebook entworfen und seit 2012 für ihre Web- und Mobilanwendungen verwendet. GraphQL wurde entwickelt, um die Probleme der APIs der damaligen Zeit zu lösen, insbesondere bei übermäßigen Abrufen [3]. 2015 wurde es von Facebook veröffentlicht und wird seitdem als offene Spezifikation weiterentwickelt. Dies ermöglicht es, die Interaktionen und Anforderungen ihrer Datenmodelle präzise zu beschreiben [8]. Im Gegensatz zu dem Ansatz, den REST verfolgt, ist GraphQL eine datenzentrische Spezifikation zur Erstellung von APIs. Genau wie es in der offiziellen Dokumentation steht

> **"ask exactly what you want"[5]**

Das heißt, das Entwickler mit GraphQL gezielt Daten abfragen können. GraphQL ist JSON sehr ähnlich und verfügt über Operationen zum Lesen (Queries) und Schreiben (Mutation) von Daten. Bei den Operationen handelt es sich um Strings, auf die der GraphQL-Dienst dann im gewünschten Format, oft JSON, antworten kann.

Wie wir alle wissen, treten im eigentlichen Entwicklungsprozess häufig Service-Updates und -Upgrades auf. Wenn dies geschieht, wird sich aufgrund der Kopplung zwischen dem Frontend und dem Server über die Datenschnittstelle die zugehörige Datenschnittstelle häufig ändern. Diese Veränderungen können zu einer Vielzahl von Problemen führen. Um Kompatibilität zu erreichen, besteht die aktuelle allgemeine Lösung darin, dass der Server eine neue Version des Datenschnittstellendienstes auf der Grundlage der Beibehaltung der ursprünglichen Datenschnittstelle bereitstellt. Die Koexistenz der beiden Schnittstellen vermeidet bis zu einem gewissen Grad das Risiko einer Inkompatibilität, erschwert jedoch die Wartung. Eine andere Lösung besteht darin, dieselbe Datenschnittstelle für die Datenanforderungen derselben Datenquelle bereitzustellen. Dies reduziert zwar die Anzahl der Datenschnittstellen, bringt jedoch viele redundante Daten mit sich und verschwendet Datenverkehr.

Eine gute Lösung ist die Bereitstellung einer einheitlichen Abfragesprache und einer Plattform für die Datenschnittstelle, für die die Server-Seite beschreibt, welche Daten sie zur Verfügung stellen kann, und das Frontend beschreibt, welche Daten es über die einheitliche Abfragesprache benötigt. Das Frontend und die Server-Seite können über eine solche Plattform vereinheitlicht werden. Die Serverseite muss nur kompatibel sein und ihre eigene Beschreibungen aktualisieren, und die Aktualisierungen der müssen nur die Abfrageanweisung aktualisieren. Diese Änderungen sind für einander transparent und können asynchron durchgeführt werden.

GraphQL bietet eine Plattform zur Entkopplung von Datenschnittstellen zwischen Frontend und Backend. Der Server muss ein Schema zur Beschreibung

der Daten bereitstellen, die über eine Datenschnittstelle bereitgestellt werden, einschließlich des Typs und Formats; das Frontend fragt den GraphQL mit der erforderlichen Abfrageanweisung ab, um die erforderlichen Daten abzurufen oder die Zieldaten zu manipulieren.

3.4.1 Einführung von GraphQL

Im folgenden sind die wichtigsten Konzepte von GraphQL aufgelistet [9]:

- **Schema**

  Sie ist das Rückgrat der Abfrage- oder Mutationsausführung. Die Anfrage wird immer gemäß der Struktur und dem Kontext des Schemas ausgeführt. Ein GraphQL-Schema besteht aus speziellen Wurzeltypen oder Einstiegspunkten: **query, mutation ,subscriptions**

- **Query**

  In GraphQL werden die Lese-Operationen als Querie bezeichnet. Er wird zum Abrufen von Daten verwendet und ist nicht mit der Manipulation von Daten am Backend verbunden. Er holt Daten entsprechend den darunter liegenden Feldern ab.

- **Mutation**

  Sie wird verwendet, um die Daten im Backend zu manipulieren. Sie erstellt und aktualisiert Einträge im Backend entsprechend den darunter liegenden Feldern.

- **Object Types**

  Diese sind die grundlegendste Komponente des GraphQL-Schemas, da sie die Art des Objekts bestimmt, das vom Dienst geholt werden soll. Sie bestimmen auch, welche Felder der GraphQL-Dienst anbietet. Der Objekttyp hat einen Namen und Felder; diese Felder müssen irgendwann aufgelöst werden. Die aufgelösten konkreten Daten sind vom skalaren Typ und stellen die Blätter der Abfrage dar. Mutationen und Queries sind spezielle Objekttypen, die als Einstiegspunkt für jede GraphQL-Abfrage dienen. Die GraphQL-Schemasprache unterstützt die skalaren Typen String, Int, Float, Boolean und ID.

- **Resolvers**

  Dies sind Funktionen, die zum Abrufen der Daten der Felder verwendet werden. Jede Funktion entspricht genau einem Feld der Payload.

- **Fields**

Bei GraphQL geht es darum, nach bestimmten Feldern auf Objekten zu fragen. Diese Felder können entweder skalare Datentypen oder Objekte darstellen. Jedes Feld wird entsprechend den darunter liegenden Resolvers ausgeführt.

- **Arguments**

  In GraphQL kann jedes Feld und verschachtelte Objekt seinen eigenen Satz von Argumenten haben, was bei der Erstellung verschiedener API-Abrufe hilfreich ist.

- **Subscription**

  Dies wird nur verwendet, wenn eine Echtzeit-Interaktion mit dem Server erforderlich ist; um sofort über wichtige Ereignisse informiert zu werden.

Im Folgenden Listing 3.1 wird die Funktionsweise von GraphQL **Schema** kurz anhand einiger Beispiele in ihren Grundzügen erläutern.

```
var schema =buildSchema(`
   type Query {
     components(type:Int):[Component]
     component(id:Int): Component
   }

   type Component{
      id: ID!
      name: String!
      type: Int!
      info: String!
    }
`);
```

Listing 3.1: Beispiel für GraphQL Query-Anweisung

In diesem Schema können Sie sehen, dass die Datenschnittstelle zwei Wurzel-Abfragen, component und components enthält. Die Rückgabetypen sind Component-Typ und ein Array dieses Typs. Eine Klammer kennzeichnet die Parameter und seinen Typ, der durch die Abfrage ausgewählt werden kann, und ein Ausrufezeichen weist darauf hin, dass das Element nicht leer sein kann. Der Typ Component enthält eine Reihe von Attributen, deren Name sowie deren Typ und Darstellung ebenfalls nicht null sein können, wenn sie mit einem Ausrufezeichen enden. Nach der Beschreibung dieses Schema ist es möglich, sich einen klaren Eindruck von den Diensten zu machen, die diese Datenschnittstelle bieten kann.

Im Gegensatz dazu sind hier, Listing 3.2 , die folgenden Aussagen zur Abfrage von Informationen über eine Component mit id von 3 aufgeführt. Man muss die Stamm-Abfrage, das heißt die component, im Hauptteil der Abfrage mit dem Parameternamen und dem Wert in Klammern angeben. Da es sich bei der com-

ponent der Abfrage um einen benutzerdefinierten Type handelt, muss man diese erweitern, um die erforderlichen Eigenschaften aufzulisten. Ebenso muss man die Beschreibung erweitern, wenn das Attribut andere benutzerdefinierte Typen enthält. Die Anfrage wird als String über eine HTTP-Anforderung an den Server übergeben, um die erforderlichen Daten im JSON-Format zurückbekommen.

```
query{
    component(id:"3"){
      name
      info
    }
 }
```

Listing 3.2: Beispiel eines GraphQL-Query

```
{
  "data":{
      "component":{
        "name": "Comp C",
        "info": "Comp C is a subtree"
      }
    }
 }
```

Listing 3.3: Responese von Beispiel für GraphQL Query-Anweisung

3.4.2 GraphiQl

GraphiQL [1]ist die integrierte GraphQL-Entwicklungsumgebung (IDE). Es ist ein leistungsstarkes Werkzeug, das oft beim Erstellen von Gatsby-Websites verwendet wird. Es ist eine Alternative zu Client-Anwendungen. GraphiQL wird üblicherweise während der Test- und Entwicklungsphase verwendet, sollte aber in der Produktion standardmäßig deaktiviert werden.

[1]https://github.com/graphql/graphiql

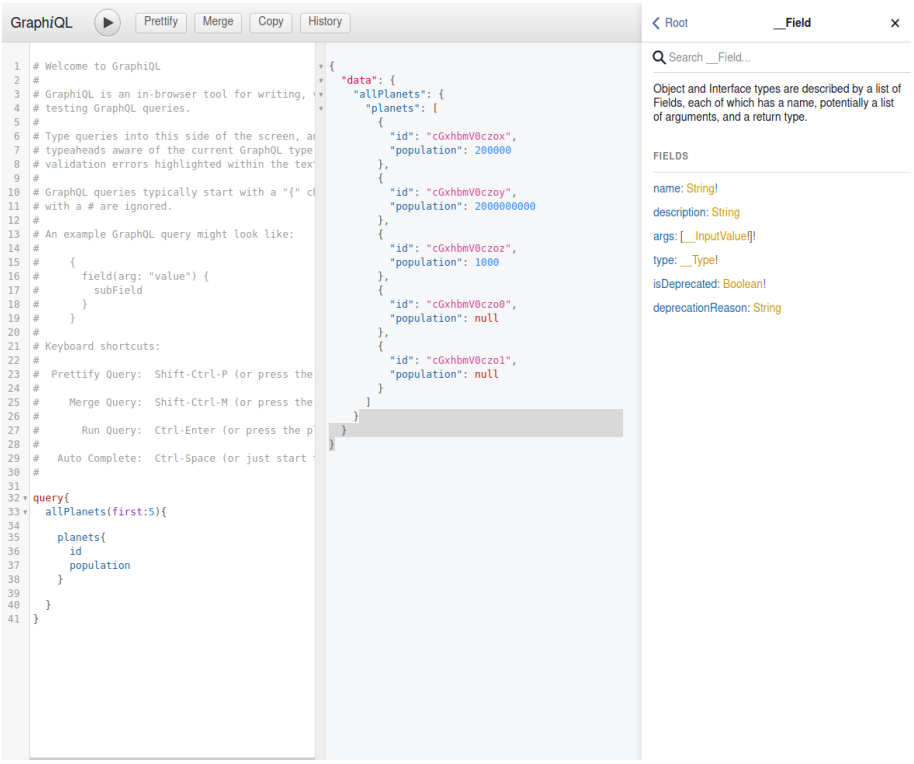

Abbildung 3.5: Schnittstelle von GraphiQL

GraphQL-Backend stellen ihre API über HTTP zur Verfügung, wo **queries** und **mutations** im Body einer POST-Anfrage gesendet werden können[10]. Beispielsweise kann mit express-graphql [2] ein Endpunkt auf einen GraphQL-Server gemountet und eine HTTP POST-Anforderung an ihn gesendet werden. Diese Operation kann auf verschiedene Weise durchgeführt werden. Mit der Entwickler Konsole vom Browser aus oder mit curl von der Befehlszeile aus. GraphiQL ist sich der Semantik der Daten bewusst und bietet Möglichkeiten zur Untersuchung und Fehlersuche, wo andere Alternativen keine haben. Es unterstützt das Debugging, indem es Hinweise gibt und auf Fehler hinweist, die der Benutzer eintippt. Darüber hinaus ist GraphQL gut in der Dokumentation, was GraphiQL nutzen kann. Die Antwort von GraphQL muss nicht nur JSON sein, sondern GraphiQL wird mit einem JSON-Viewer geliefert, der bevorzugt wird. GraphiQL wird während des Prototyps und der Validierung dieser Forschungsarbeit gut verwendet, und ein Beispiel dafür ist in Abbildung 3.4 dargestellt. Star Wars Data verfügt über eine API, die derzeit bei swapi [3] gehostet werden. Diese IDE verfügt über Syntax-Highlighting, intelligente Typisierung vor Feldern, Argumenten und Typen, Dokumentationsexplorer, Echtzeit-Fehler-Highlighting

[2] `https://github.com/graphql/express-graphql`
[3] `https://swapi.dev/`

und -Berichterstellung, automatische Abfragevervollständigung und Tools zum Ausführen und Überprüfen von Abfrageergebnissen.

3.5 PostGraphile

PostGraphile[4] ist ein Tool, welches die Generierung einer GraphQL-API aus einem PostgreSQL-Schema ermöglicht und als Server vor unserer Datenbank läuft. Es verfügt über einen sehr hohen Automatisierungsgrad, der eine automatische Zuordnung von Tabellenstruktur, Beziehungen, Einschränkungen, Zugriffsberechtigungen usw, ermöglicht, wodurch die serverseitige Entwiklungseffizienz effektiv verbessert werden kann. Die Entwickler entwerfen die PostgreSQL-Datenbank: die Struktur der Tabellen, die Beziehungen zwischen den Tabellen und die Berechtigungen des Benutzers. Ebenso kann PostGraphile diese Beziehungen automatisch auf die endgültige GraphQL-Schnittstelle abbilden.

Eine PostGraphile-API bietet wahrscheinlich eine leistungsfähigere und standardkonformere GraphQL-API als jede intern erstellte und kann in einem Bruchteil der Zeit erstellt werden. Konzentrieren wir uns auf Ihr Produkt und lassen Sie PostGraphile sich um die API-Ebene kümmern. Es stellt damit einen GraphQL-Server bereit, der die Daten hoch-intelligent behandelt und der sich automatisch und ohne Neustart aktualisiert, wenn sich die Datenbank ändert. Die Anmerkungen in den Tabellen dienen zur Beschreibung der Schnittstelle, und intelligente Anmerkungen bieten zusätzliche Funktionalität. Dies wird von den Entwicklern als DDGD (Database-Driven GraphQL Development) oder datenbankgesteuerte GraphQL-Entwicklung bezeichnet. In der Abbildung 3.5 ist ein Beispiel einer PostGraphile-Schnittstelle gezeigt.

Die linke Seite der Schnittstelle zeigt den graphiql-explorer. Es ist ein Plugin für GraphiQL, das eine neue Technik zur Untersuchung und Erstellung von GraphQL-Abfragen hinzufügt. Es fügt grafische Felder und Eingaben hinzu, die in Abfragen verwendet werden können. Es ermöglicht auch die Erstellung vollständiger Abfragen durch Anklicken der verfügbaren Felder und Eingaben, ohne dass diese Abfragen wiederholt manuell eingegeben werden müssen. Auf der rechten Seite sieht man, dass PostGraphile automatisch CRUD-Mutationen für jedes Tabellenschema hinzugefügt hat.

[4] https://www.graphile.org/

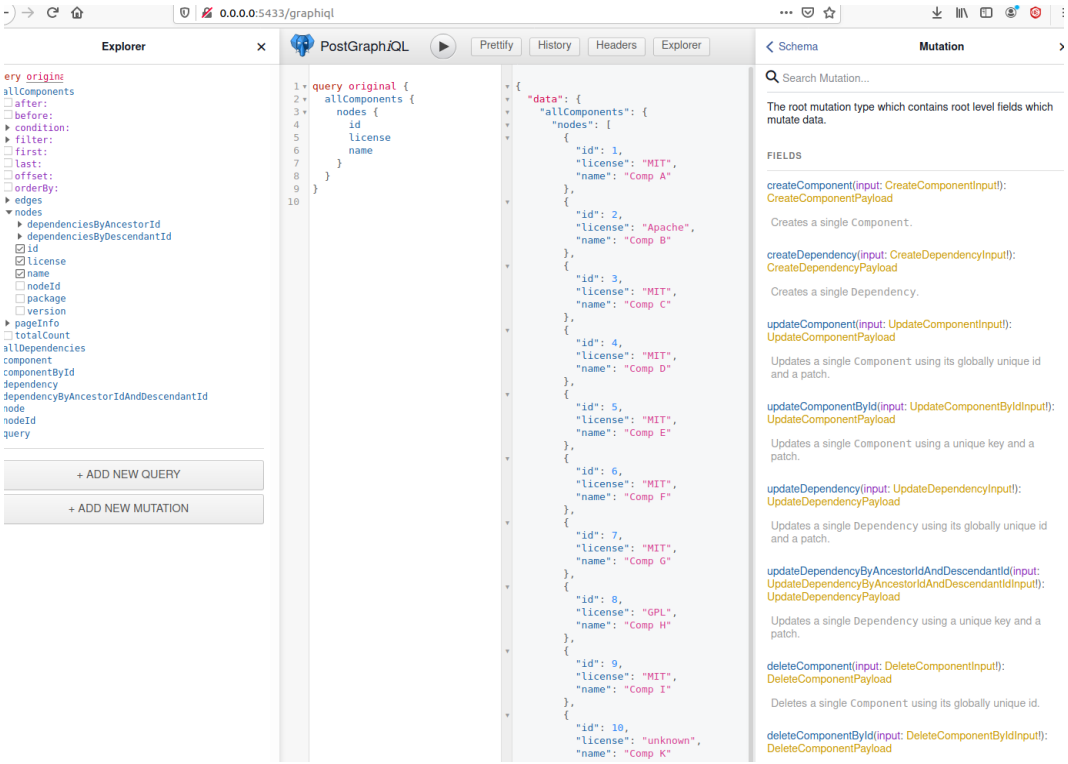

Abbildung 3.6: Schnittstelle von PostGraphile-GraphiQL

4 Konzept zur Speicherung von Graphen in relationalen Datenbank

Es existieren verschiedene Variante, um Graphenstrukturierte Daten in relationale Datenbank abzubilden. Als Vorleistung zu dieser Arbeit wurden einige Variante in der Literatur "SQL Antipatterns "[11] behandelt und können dort detaillierter nachgelesen werden. An dieser Stelle soll nur eine Kurzfassung der drei Variante(Adjacency List, Path Enumeration und Closure Table) mit einem Blick auf die Grundoperationen auf Graphenstruktur, wie das Suchen/Anfragen, Einfügen und Löschen erfolgen.

- Variante 1: Adjacency list(Adjazenz-Liste)
- Variante 2: Path Enumeration
- Variante 3: Closure Table

4.1 Variante 1: Adjacency List (Adjazenz-Liste)

Das Adjacency list Modell ist der einfachste und unkomplizierteste Weg, um einen Graphen darzustellen. Dies ist die typische Speicherung mittels Selbstreferenz. Es erfolgt ein Verweis auf den Vorgänger durch eine ausgezeichnete Spalte in derselben Relation. Einfach ausgedrückt bedeutet dies, dass jede Zeile über genügend Informationen verfügen muss, um die Nachbarschaft abzuleiten. Deshalb wird dies eine Adjacency list genannt. Die referentielle Integrität ist in dieser Variante gegeben.

Zu Veranschaulichung von Variante 1 verwenden wir Abbildung 4.2 aus einer einfachen Graphen Struktur. Dieser Graph wird sowohl als gerichtet als auch als ungerichteter Graph interpretiert.

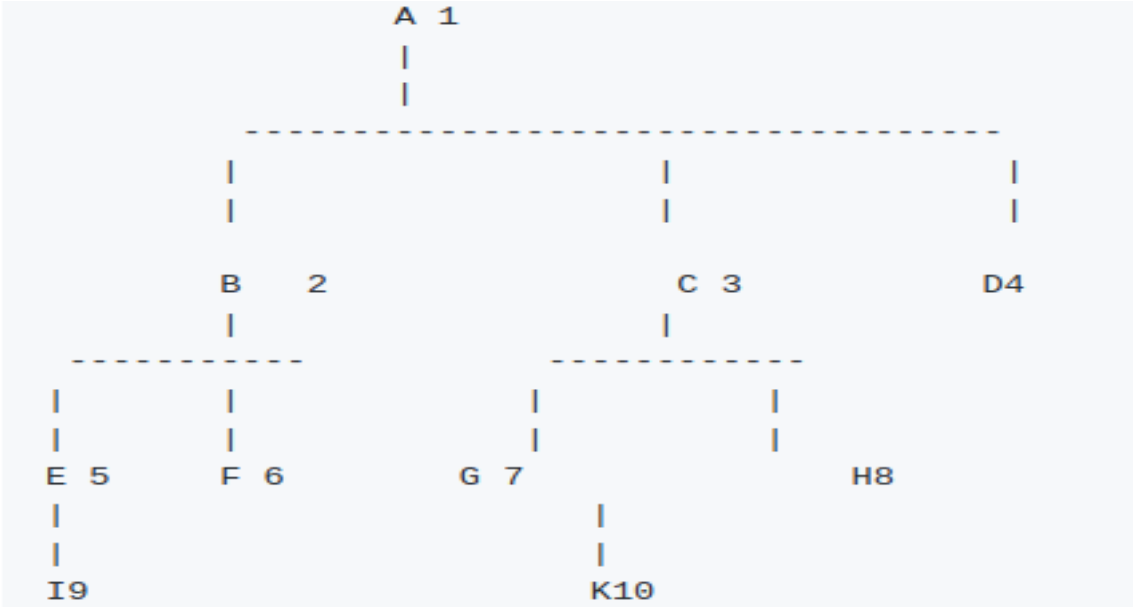

Abbildung 4.1: Beispiele Graph von Adjacency List

Schema erstellen

Anhand der Abbildung 4.1 unter diesem Konzept wird gezeigt, wie eine einfache Sicht in PostgreSQL erstellt wird. Das Beispiel basiert auf einem Schema (Tabelle) mit der folgenden Definition:

```
1 create table myschema.component(
2    id integer  primary key,
3    name character varying,
4    parent_id integer
5 );
```

Tabellename component mit Tabellenstruktur werden mit drei Attributen gebaut. Das erste Attribut *id* ist ein primärer Schlüssel, es hat den Datentyp Integer. *name* ist das zweite Attribute mit dem Datentyp character varying (String). Das dritte Attribute ist *parent_id* , es hat den Datentyp Integer. Es erfolgt ein Verweis auf den Vorgänger (*id*) durch eine ausgezeichnete Spalte (*parent_id*) in derselben Relation.

Einfügen-Operation

Die Tabelle hat nur Verweise auf sich selbst. Der Wurzelknoten A enthält dann einen leeren Wert (NULL) für ihren Elternteil. Die folgende SQL-Anweisung (Abbildung 4.3) fügt Datensätze in die Tabelle component ein:

```
1  INSERT INTO public.component(
2      id, name, parent_id)
3      VALUES (1,'Comp A',NULL);
4  INSERT INTO public.component(
5      id, name, parent_id)
6      VALUES (2,'Comp B',1);
7  INSERT INTO public.component(
8      id, name, parent_id)
9      VALUES ((3,'Comp C',1);
10 INSERT INTO public.component(
11     id, name, parent_id)
12     VALUES  (4,'Comp D',1);
13 INSERT INTO public.component(
14     id, name, parent_id)
15     VALUES (5,'Comp E',2);
16 INSERT INTO public.component(
17     id, name, parent_id)
18     VALUES (6,'Comp F',2);
19 INSERT INTO public.component(
20    id, name, parent_id)
21     VALUES (7,'Comp G',3);
22 INSERT INTO public.component(
23    id, name, parent_id)
24    VALUES (8,'Comp H',3);
25 INSERT INTO public.component(
26    id, name, parent_id)
27    VALUES  (9,'Comp I',5);
28 INSERT INTO public.component(
29     id, name, parent_id)
30     VALUES (10,'Comp K',7);
```

Abbildung 4.2: Einfügen-Operation (Adjacency list)

Anfragen-Operation

Es wird der komplette Graph aus der Tabelle component angefragt.

```
1   SELECT * from component ORDER BY id ASC;
```

Data Output | Explain | Messages | Notifications

	id [PK] integer	name character varying	parent_id integer
1	1	Comp A	[null]
2	2	Comp B	1
3	3	Comp C	1
4	4	Comp D	1
5	5	Comp E	2
6	6	Comp F	2
7	7	Comp G	3
8	8	Comp H	3
9	9	Comp I	5
10	10	Comp K	7

Abbildung 4.3: Anfragen des gesamten Komponengraphen von Adjacency list)

Löschen-Operation

Um an einem Knoten die Löschen-Operation durchzuführen, muss man rekursiv alle Kinder-Knoten dieses Knoten löschen. Das heißt, man muss die Teil-Graphen (Kinder-Knoten) des gelöschten Graphen-Knoten herausfinden und diese zuerst löschen,sonst bekommt man eine Error-Anzeige! Zum Beispiel, möchte man Knoten C mit dem Wert 3 löschen, sollte man die Kinder-Knoten G mit Wert 7 und H mit Wert 8 löschen. Aber um den Knoten G zu löschen, sollte man auch erst alle Kinder-Knoten K von G mit Wert 10 löschen, erst dann löscht man Knoten C mit Werte 3. Die folgende SQL-Anweisung folgt:

Abbildung 4.4: Result von Löschen(Adjacency list)

Zu den Vorteilen Adjacency list gehören:

- intuitive Darstellung der Daten entsprechen direkt den Graphen-Kanten
- es gibt keine Datenredundanz, und es besteht keine Notwendigkeit für nicht referenzielle Integritätsbeschränkungen
- schnelles und einfaches Einfügen, Löschungen sind möglich, da sie keine zusätzlichen Tabellen betreffen

Zu den Nachteilen von Adjacency list gehören:

- Das Finden aller Nachfolger oder Vorgänger bzw. eines Telegraph oder das Löschen von inneren Knoten sind komplexere Operationen.

4.2 Variante 2: Path Enumeration

Die Idee der Path Enumeration besteht darin, eine ausgezeichnet Spalte (*path*) speichert den Pfad von der Wurzel bis zum jeweiligen Knoten als String. Der Pfad

setzt sich aus den Primärschlüsseln und Trennzeichen zusammen. Die referentielle Integrität ist in dieser Variante nicht gegeben.

Einfach ausgedrückt sieht diese Variante aus wie eine **webpage breadcrumb**. Man geht von einem Standpunkt aus, den man meistens als Startseite bezeichnet und geht schritt für Schritt zur aktuellen Seite über. Ein bisschen so wie Startseite *Nachrichten IT Aktuelle*.

Zu Veranschaulichung von Variante 2 verwenden wir Abbildung 4.6 mit einer einfachen Graphen Struktur. Dieser Graph wird als gerichteter Graph interpretiert.

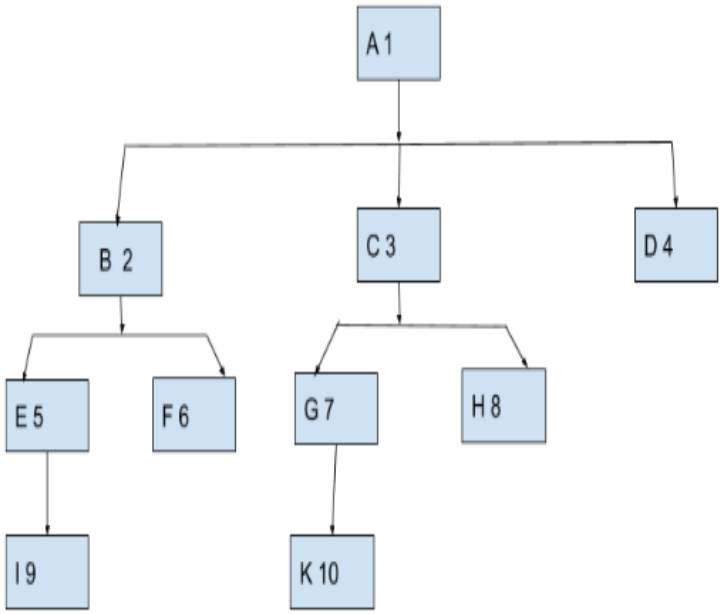

Abbildung 4.5: Beispiel Graph von Path Enumeration

In unserem Fall ist die Wurzel "Comp A"mit einer ID-Nummer 1. Danach haben wir drei Kategorie"Comp B"mit einer ID-Nummer 2, "Comp C"mit einer ID-Nummer 3 und "Comp D"mit einer ID-Nummer 4. Um diese Beziehung als Pfadaufzählung zu schreiben, verknüpfen wir alle IDs in aufsteigender Reihenfolge. Das heisst, vom Wurzel-Knoten Comp Abis zur "Comp F"und trennen sie durch ein anderes Zeichens als eine Zahl (zum Beispiel :.). Es sollte folgendes Ergebnis ergeben: **1.2.6** . Dieses Ergebnis ist unsere Path Enumeration.

Schema erstellen

Anhand der Abbildung 4.6 unter diesem Konzept wird gezeigt, wie eine einfache

Sicht in PostgreSQL erstellt wird. Das Beispiel basiert auf einem Schema (Tabelle) mit der folgenden Definition:

```
1 create table myschema.component(
2    id integer primary key,
3    name character varying,
4    path character varying
5 );
```

Einfügen-Operation

Die folgende SQL-Anweisung fügt einige Datensätze in die Tabelle "component" ein:

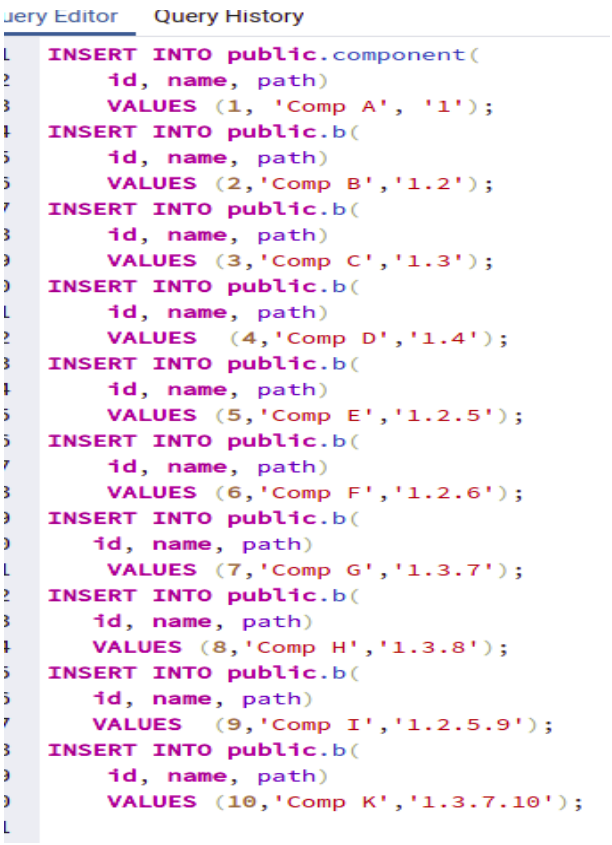

Abbildung 4.6: Einfügen-Operation (Path Enumeration)

Anfrage-Operation

Es werden alle Datensätze aus der Tabelle component angefragt.

```
1   select * from component ORDER BY id ASC;
```

Data Output Explain Notifications

	id [PK] integer	name character varying	path character varying
1	1	Comp A	1
2	2	Comp B	1.2
3	3	Comp C	1.3
4	4	Comp D	1.4
5	5	Comp E	1.2.5
6	6	Comp F	1.2.6
7	7	Comp G	1.3.7
8	8	Comp H	1.3.8
9	9	Comp I	1.2.5.9
10	10	Comp K	1.3.7.10

Abbildung 4.7: Anfragen des Table von Path Enumeration

Löschen-Operation

Wenn man die 'Comp B' löschen möchte, sollte man dies mit der **LIKE**-Klausel versuchen zu realisieren. Die folgende SQL-Anweisung folgt:

```
1  delete from public.component
2  where path like '1.2' || '%';
```

Data Output Explain Notifications

	id [PK] integer	name character varying	path character varying
1	1	Comp A	1
2	3	Comp C	1.3
3	4	Comp D	1.4
4	7	Comp G	1.3.7
5	8	Comp H	1.3.8
6	10	Comp K	1.3.7.10

Abbildung 4.8: Result von Löschen(Path Enumeration)

Zu den Vorteilen von Path Enumeration gehören:

- intuitive Darstellung der Graphenstruktur

- Abfrage ist sehr einfach

Zu den Nachteilen von Path Enumeration gehören:

- Das Verschieben einer ganzen Graphenstruktur ist sehr teuer
- Je nachdem, wie man den Pfad speichert, kann man nicht direkt in SQL damit arbeiten, man muss immer den Pfad holen und analysieren, wenn man ihn ändern möchte.

4.3 Variante 3: Closure Table

Bei dieser Variante werden die Pfad-Informationen in einer separaten Tabelle gespeichert. Dabei werden alle Pfade von einem Knoten zu all seinen Nachfolgern notiert. Die referentielle Integrität ist in dieser Variante gegeben.

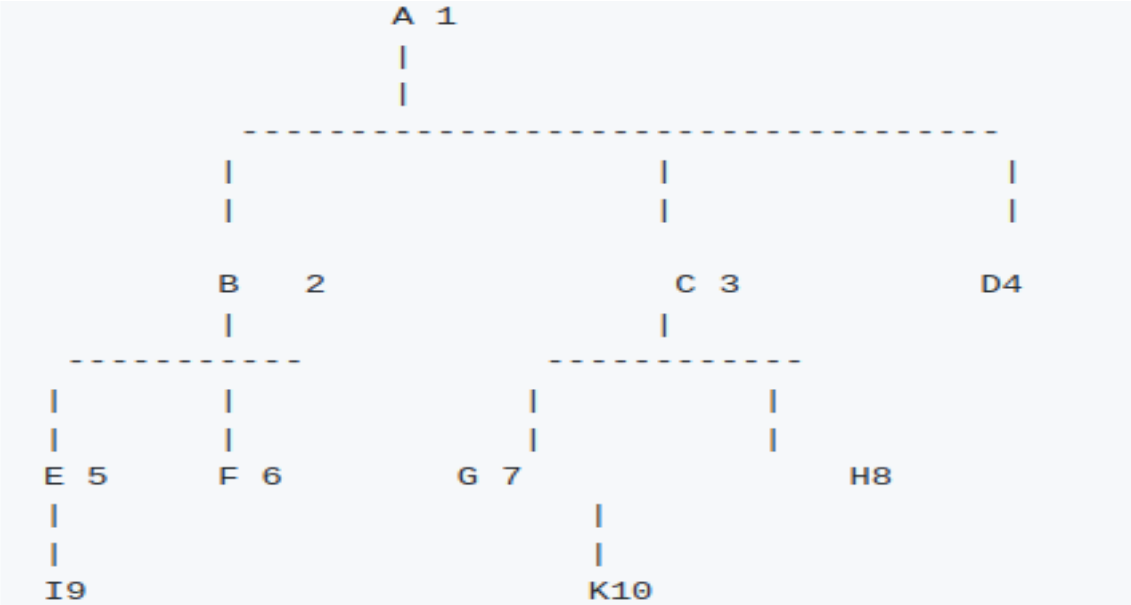

Abbildung 4.9: Beispiele Graph von Closure Table

Schema erstellen

Anhand der Abbildung 4.2 unter diesem Konzept wird gezeigt, wie eine einfache Sicht in PostgreSQL erstellt wird. Das Beispiel basiert auf einer Schema (Tabelle) mit der folgenden Definition:

```
1 create table myschema.component(
2    id integer  primary key,
3    name character varying
```

```
4 );
5 create table myschema.dependency(
6  ancestor_id integer not null,
7  descendant_id integer not null,
8  depth integer not null,
9  primary key (ancestor_id,descendant_id),
10  foreign key (ancestor_id) references public.component(id),
11  foreign key (descendant_id) references public.component(id)
12 );
```

Der myschema.component (Tabelle) besteht aus drei Attributen. Das erste Attribut *id* ist ein primär Schlüssel, es hat den Datentyp Integer. Das zweite Attribute ist *name* mit einem Datentyp character varying (String). Das dritte Attribut ist *parent_id*, es hat den Datentyp Integer. Die zweite Tabelle mit dem Namen myschema.dependency enthält die Attribute *ancestor_id* und *descendant_id*, welche die Knoten in myschema.component *id* referenzieren. (Es enthält zudem das) ??? Attribut *depth* aus der myschema.dependency Tabelle, die die Tiefe eines Knotens darstellt, von dem aus seine Kinder gespeichert werden.

Einfügen-Operation

Die Tabelle hat nur Verweise auf sich selbst. Der Wurzelknoten A enthält dann einen leeren Wert (NULL) für seinen Elternteil. Die folgende SQL-Anweisung (Abbildung 4.3) fügt Datensätze in die Tabelle component ein:

```
1 INSERT INTO public.component(
2   id,name)
3   VALUES (1, 'Comp A');
4 INSERT INTO public.component(
5     id,name)
6   VALUES (2, 'Comp B');
7 ........
8
9 INSERT INTO public.dependency(
10   ancestor_id, descendant_id, depth)
11   VALUES (1, 1, 0);
12
13 INSERT INTO public.dependency(
14   ancestor_id, descendant_id, depth)
15   VALUES (1, 2, 1);
16 ..........
```

Anfrage-Operation

Es werden alle Vorgänger von Component mit Id gleich 7 angefragt.

Data Output | Explain | Messages

	id [PK] integer	name character varying
1	1	Comp A
2	2	Comp B
3	3	Comp C
4	4	Comp D
5	5	Comp E
6	6	Comp F
7	7	Comp G
8	8	Comp H
9	9	Comp I
10	10	Comp K

Abbildung 4.10: Result1 von Einfügen-Operation (Closure Table)

Data Output | Explain

	ancestor_id [PK] integer	descendant_id [PK] integer	depth integer
1	1	1	0
2	1	2	1
3	1	3	1
4	1	4	1
5	1	5	2
6	1	6	2
7	1	7	2
8	1	8	2
9	1	9	3
10	1	10	3
11	2	2	0
12	2	5	1
13	2	6	1
14	2	9	2
15	3	3	0
16	3	7	1
17	3	8	1
18	3	10	2
19	4	4	0
20	5	5	0
21	5	9	1
22	6	6	0
23	7	7	0
24	7	10	1
25	8	8	0

Abbildung 4.11: Result2 von Einfügen-Operation (Closure Table)

```
1  select  c.*
2  from public.component as c JOIN public.dependency as t
3  on (c.id= t.ancestor_id)
4  where t.descendant_id= 7;
```

Data Output | Explain

	id [PK] integer	name character varying
1	3	Comp C
2	7	Comp G
3	1	Comp A

Abbildung 4.12: Result von Anfragen des Closure Table

Löschen-Operation

Es wird angefragt, die zu Id= 7 gehörige Information zu löschen. Die folgende SQL-Anweisung folgt:

```
1 delete from public.dependency
2 where ancestor_id =7 || descendant_id =7;
```

Zu den Vorteilen von Closure Table gehören:

- Unterstützt die referentielle Integrität.
- Einfache Abfrage von Teil-Graphen
- Mehrere Hierarchien können existieren.

Zu den Nachteilen von Closure Table gehören:

- Die Gesamtzahl der Zeilen kann bei großen Datensätzen hoch sein
- Schwer löschbare Datensätze aus dem Graphen

4.4 Vergleich der Ansätze mit den Anforderungen

Die drei Konzepte, die in den Abschnitten 4.1, 4.2 und 4.3 erläutert wurden, wurden je nach Anforderungen verglichen, um das am Besten geeignete Konzept für diese Bachelorarbeit auszuwählen. Anforderungen aus Kapitel 2 werden mithilfe einer Checkliste (SQL-Statement) daraufhin überprüft, ob sie erfüllt werden.

- **A4: Querying des gesamten Komponentengraphen**

  Der Komponentengraph soll als Ganzes durch SQL ausgelesen werden können. Dies beinhaltet alle Komponenten, die Beziehung-Information zwischen den Komponenten, sowie die Attribute der Komponenten.

- **A5: Querying eines Teil des Komponentengraphens**

  Er soll es auch ermöglichen alle Teile eines Komponentengraphens auslesen zu können. Ein Beispiel hierfür ist das Auslesen einer spezifischen Komponente und all ihrer anhängigem Komponenten.

Variante 1: Adjacency List

A4: Querying des gesamten Komponentengraphen

Die SQL-Anweisung folgt:

```
1  WITH  RECURSIVE component_path (id,name,path) as
2  (
3  select id,name,name as path
4    from public.component
5    where parent_id is null
6    union all
7    select c.id,c.name,concat(cp.path,'-',c.name)
8    from component_path as cp join public.component as c
9    on cp.id= c.parent_id
10 )
11 select * from component_path
12 order by path
```

	id integer	name character varying	path character varying
1	1	Comp A	Comp A
2	2	Comp B	Comp A-Comp B
3	5	Comp E	Comp A-Comp B-Comp E
4	9	Comp I	Comp A-Comp B-Comp E-Comp I
5	6	Comp F	Comp A-Comp B-Comp F
6	3	Comp C	Comp A-Comp C
7	7	Comp G	Comp A-Comp C-Comp G
8	10	Comp K	Comp A-Comp C-Comp G-Comp K
9	8	Comp H	Comp A-Comp C-Comp H
10	4	Comp D	Comp A-Comp D

Abbildung 4.13: Result des gesamten Komponengraphen von Adjacency list)

A5: Querying eines Teils des Komponentengraphens

Die folgende Abfrage ruft den Teil-Graph mit dem Namen 'Comp B' ab, dessen ID 2 ist.

```
1  WITH  RECURSIVE Teilcomponent_path (id,name,path) as
2   (
3   select id,name,name as path
```

```
4     from public.component
5     where parent_id = 2
6       union all
7   select c.id,c.name,concat(cp.path,'-',c.name)
8     from Teilcomponent_path as cp join public.component as c
9     on cp.id= c.parent_id
10  )
11    select * from Teilcomponent_path
12    order by path
```

Data Output | Explain | Notifications

	id integer	name character varying	path character varying
1	5	Comp E	Comp E
2	9	Comp I	Comp E-Comp I
3	6	Comp F	Comp F

Abbildung 4.14: Result des Teil-Graph Komponengraphen von Adjacency list)

Variante 2: Path Enumeration

A4: Querying des gesamten Komponentengraphen

Mit der Variante 2 Path Enumeration ist es sehr leicht, den gesamten Komponentgrahpen zu veranschaulichen.

```
1   select * from public.component
```

Data Output Explain Notifications

	id [PK] integer	name character varying	path character varying
1	1	Comp A	1
2	2	Comp B	1.2
3	3	Comp C	1.3
4	4	Comp D	1.4
5	5	Comp E	1.2.5
6	6	Comp F	1.2.6
7	7	Comp G	1.3.7
8	8	Comp H	1.3.8
9	9	Comp I	1.2.5.9
10	10	Comp K	1.3.7.10

Abbildung 4.15: Anfragen des Table von Path Enumeration

A5: Querying eines Teils des Komponentengraphens

Anfrage, welche alles Vorgänger von 'Comp F' sind.

```
1 select *
2 from public.dependency
3 where '1.2.6' like (path || '%')
4 order by length(path);
```

Data Output Explain Notifications

	id [PK] integer	name character varying	path character varying
1	1	Comp A	1
2	2	Comp B	1.2
3	6	Comp F	1.2.6

Abbildung 4.16: Result des Teil-Graph Komponengraphen von Path Enumeration

Variante 3: Closure Table

A4: Querying des gesamten Komponentengraphen

In der Closure Table wird jeder Pfad in einem Graph gespeichert, nicht nur direkt die parent-kind-relationship, sondern auch die Großeltern-Enkelkinder-relationship und auch jeder andere Pfad.

Die SQL-Anweisung folgt:

```
1 select array_agg(c.name order by c.id ,' - ') as path
2 from public.dependency d
3 join public.dependency a
4 on (a.descendant_id= d.descendant_id)
5 join public.component c on
6 (c.id = a.ancestor_id)
7 where d.ancestor_id = 1
8 and d.descendant_id != d.ancestor_id
9 group by d.descendant_id;
```

A5: Querying eines Teil des Komponentengraphens

Die SQL-Anweisung folgt:

```
1 SELECT t.*, q.ancestor_id as parent_id
2   FROM public.component as t
3   JOIN public.dependency as q
4   on (t.id = q.descendant_id)
5   WHERE q.ancestor_id = 2
```

	id integer	name character varying	parent_id integer
1	2	Comp B	1
2	5	Comp E	1
3	6	Comp F	1
4	9	Comp I	1

Abbildung 4.17: Result von Teil des Graph (Closure Table)

Durch die Vergleiche in Kapitel 4.1 bis Kapitel 4.3 wird die Variante 3 als das Konzept für diese Abschlussarbeit gewählt. Abbildung 4.17 aus [1] soll einen

Überblick über die Stärken und Schwächen der jeweiligen Operationen der Variante liefern. Klassifiziert wurde anhand der Art, Anzahl und der Verwendung von Funktionen in den Operationen.

Design	Adjacency List	Path Enumeration	Nested Sets	Closure Table
Tables	1	1	1	2
Query Child	Easy	Easy	Hard	Easy
Query Tree	Hard	Easy	Easy	Easy
Insert	Easy	Easy	Hard	Easy
Delete	Easy	Easy	Hard	Easy
Ref. Integ.	Yes	No	No	Yes

Abbildung 4.18: Tabellarische Gegenüberstellung[1]

Im vorherigen Teil habe ich drei Varianten zur Speicherung von Graphenstrukturen in einer relationale Datenbank, sowie die Anforderungen 4 und 5 aus der Aufgabenstellung zum Abfragen ganzer Graphen und auch von Teilen des Graphen beschrieben. Im Folgenden werde ich meine Auswahl angeben und erläutern. Die Entscheidung für das Konzept meiner Bachelorarbeit sollte immer unter dem Gesichtspunkt der Aufgabenstellung, der Anforderungen, sowie anhand der Häufigkeit der erwarteten Operationen fallen.

Die erste Variante **Adjazenz List** ist nativ und das einfachste Modell. Alle Anfragen und Schreibvorgänge sind sehr einfach und schnell. Sie ist auch mit der referenziellen Integrität kompatibel. Das Anfragen aller Nachfolger oder Vorgänger bzw. eines Teil-Graphen oder das Löschen von inneren Knoten, sind aber komplexere Operationen im Vergleich zu den anderen zwei Varianten. Das Abfragen von Component, die tiefer liegen als die unmittelbaren untergeordneten Component, ist jedoch nahezu unmöglich, ohne die Rekursion auf der Skriptseite oder die **WITH RECURSIVE**-Anweisung zu verwenden. Dieser verhängnisvolle Fehler steht im Widerspruch zu den Anforderungen dieser Abschlussarbeit, da diese im zweiten Kapitel (A4 bis A7), Anforderungen an die Schwierigkeit der Abfrage haben. Darüber hinaus wird die relative Zusammenarbeit mit GraphQL auch eine potentielle Schwierigkeit/Hürde bereiten können.

Die zweite Variante **Path Enumeration** ist eigentlich eine gute Wahl, aber Abfragen zur Ermittlung von Nachkommen erfordern String-Parsing, was insbesondere bei größeren Datensätzen ein Leistungshindernis darstellen kann. Variante

zwei funktioniert für alle Arten von Abfragen, ist aber an der Kompatibilität gescheitert, da sie nicht der Anforderung der referentiellen Integrität nachkommt. Dies widerspricht der Anforderung A1 in meiner Abschlussarbeit.

Die Variante 3 **Closure Table** ist, wie oben bereits erwähnt, ein gutes Datenbankmodell für den Umgang mit graphartigen Daten. Die Vorteile liegen in der Möglichekti der CRUD-Operation, als auch in der einfachen Implementierung. Außerdem ist diese Variante auch rekursionsfrei und erlaubt die Verwendung von referentieller Integrität. Ein Nachteil der Variante 3 ist die hohe Festplattennutzung. Dies ist aber kein Nachteil, auf den wir in dieser Bachelorarbeit Rücksicht nehmen müssen.

Vom Ausgangspunkt dieser Arbeit aus wird deutlich, dass die Größe der Graphenstruktur das Design des Datenbankmodells beeinflusst. Wenn die Graphenstruktur sehr groß ist, sind die Effizienzanforderungen, wie Abfrage und Positionierung, hoch. Die Closure Table ist für den Einsatz mit großen Graphen besser geeignet, als die anderen dargestellten Varianten. Die gute Lesbarkeit des gesamten Graphen wurde bei der Auswahl auch berücksichtigt. Eine gründlichere Vollpfadstruktur, bei der die vollständige Ausdehnung der relevanten Knoten im Pfad separat aufgezeichnet wird, ist besonders mit einer Closure Table möglich. Die Closure Table kommt zudem ohne Rekursionen aus, was beim Einsatz einer GraphQL-Schnittstelle vorteilhaft ist, da Rekursionen in GraphQL-Schnittstellen nicht unterstützt werden. Was wir auch berücksichtigen müssen, ist wie gut der Graph erweitert werden kann. Dies kann in der Abbildung 4.18 abgelesen werden. Ein logisches Datenmodell soll eine oder mehrere many-to-many-Beziehungen enthalten. Physikalische Datenmodellierungstechniken verwandeln eine many-to-many-Beziehung in eine one-to-many-Beziehung, indem weitere Tabellen hinzugefügt werden. Die Closure Table bietet gute Möglichkeiten, die Erweiterbarkeit von gespeicherten Graphen sicherzustellen. Dies ist gegeben, da die Closure Table zwei Tabellen hat, eine mit den grundlegenden Informationen über die Komponente und die andere mit dem Pfad zu der gespeicherten Komponente. Die Variante Closure Table ist in ihrem Umgang intuitiver als die beiden anderen Varianten Adjacency List und Path Emutation.

Wie bereits im Kapitel 3.2 gezeigt wurde, kann ein Graph zwischen gerichteten und ungerichteten Graphen unterschieden werden. Eine Komponente benutzt Funktionalitäten einer anderen Komponente. Dadurch entsteht ein gerichteter Graph. Wenn beide Komponenten die Funktionalitäten des anderen verwenden, wird ein ungerichteter Graph erzeugt. Dies führt auch in der Graphenstruktur zu einer Mischung von gerichteten und ungerichteten Graphen. Closure-Tabelle ist eine gute Möglichkeit, eine solche Datenstruktur zu speichern. Die anderen beiden Konzepte haben keine Möglichkeit, dieses Speicherproblem zu lösen.

Im Rahmen dieser Arbeit soll aus den oben dargestellten Gründen das Konzept des Closure Table für die weitere Umsetzung verwendet werden.

5 Umsetzung

Auf Grundlage des im vorherigen Kapitel entworfenen Konzeptes wird das Speicherkonzept des Closure Table als Speichermethode für die PostgreSQL-Datenbank angewandt. Um die Abhängigkeit von Komponenten anzuzeigen, wird GraphQL als Abfragesprache verwendet, sowie PostGraphile als anzupassenden GraphQLs API, zur Verfügung gestellt. Zuerst wird das zur Datenspeicherung benötigte Datenbankschema von PostgreSQL vorgestellt. Beim Implementierungsteil kamen Container-Technologien zum Einsatz. Es soll eine Beschreibung angegeben werden, wie man ein Netzwerk von Docker-Containern auf einem lokalen Rechner betreibt, einschließlich eines Containers für eine PostgreSQL-Datenbank und eines Containers für PostGraphile. Anschließend wird der Queries von PostGraphile vorgestellt. In den beiden Abschnitten wird auch nach den einzelnen Implementierungsschritten und Features chronologisch geordnet.

5.1 Datenbankschema von PostgreSQL

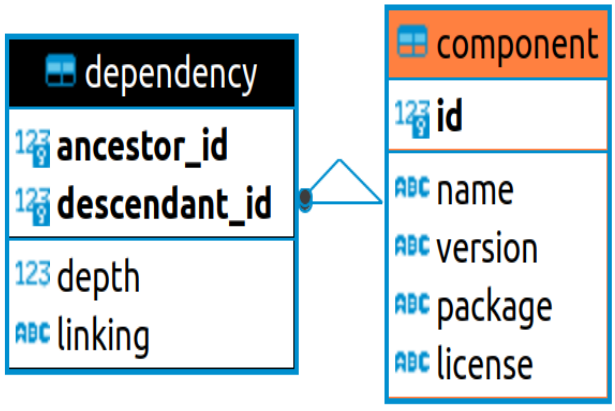

Abbildung 5.1: ER-Diagramm von Komponenten und der Beziehung zwischen Komponenten

Das Datenbankschema in Abbildung 5.1 für die Verwaltung von Artenlisten besteht aus zwei Relationen. Die Component-Relation enthält die Komponenten-Informationen. Eine Dependency-Relation enthält alle Pfad (Kanten)- Informationen.

Das Schema kann in PostgreSQL mit folgender SQL-Anweisung erzeugt werden:

```
1  create table public.component(
2     id serial primary key,
3     name text,
4     version text,
5     package text,
6     license text
7  );
8  create table public.dependency(
9   ancestor_id integer not null,
10  descendant_id integer not null,
11  depth integer not null,
12  linking text,
13  primary key (ancestor_id,descendant_id),
14  foreign key (ancestor_id) references public.component(id),
15  foreign key (descendant_id) references public.component(id)
16 );
```

Zu beachten ist, dass sich die Attribute der hier aufgeführten Component etwas von denen in Kapitel 4 unterscheiden. Hier sind zusätzliche Attribute hinzugefügt worden; es sind version, package und license, die alle als text-Type angegeben werden.

Um die Datenbank in einem initialen Zustand anzugeben, sollten einige Dummy-Daten eingefüllt werden. In der Abbildung sind die Dummy-Daten, die gefüllt wurden.

```
1  INSERT INTO public.component(
2    id, name, version, package, license)
3    VALUES (1, 'Comp A', '1.0.0', 'org.a', 'MIT');
4  INSERT INTO public.component(
5    id, name, version, package, license)
6    VALUES (2, 'Comp B', '1.0.0', 'org.b', 'Apache');
7  INSERT INTO public.component(
8    id, name, version, package, license)
9    VALUES (3, 'Comp C', '1.2.0', 'org.c', 'MIT');
10 INSERT INTO public.component(
11   id, name, version, package, license)
12   VALUES (4, 'Comp D', '1.0.0', 'org.a', 'MIT');
13 INSERT INTO public.component(
14   id, name, version, package, license)
15   VALUES (5, 'Comp E', '1.0.0', 'org.a', 'MIT');
16 INSERT INTO public.component(
```

```
  id, name, version, package, license)
  VALUES (6, 'Comp F', '1.0.0', 'org.a', 'MIT');
INSERT INTO public.component(
  id, name, version, package, license)
  VALUES (7, 'Comp G', '1.0.0', 'org.a', 'MIT');
INSERT INTO public.component(
  id, name, version, package, license)
  VALUES (8, 'Comp H', '1.0.0', 'org.a', 'GPL');
INSERT INTO public.component(
  id, name, version, package, license)
  VALUES (9, 'Comp I', '1.0.0', 'org.l', 'MIT');
INSERT INTO public.component(
  id, name, version, package, license)
  VALUES (10, 'Comp K', '1.0.0', 'org.k', 'unknown');
```

```
INSERT INTO public.dependency(
  ancestor_id, descendant_id, depth,linking)
  VALUES (1, 1, 0,'STATIC_LINKED');
INSERT INTO public.dependency(
  ancestor_id, descendant_id, depth,linking)
  VALUES (1, 2, 1,'STATIC_LINKED');
INSERT INTO public.dependency(
  ancestor_id, descendant_id, depth,linking)
  VALUES (1, 3, 1,'STATIC_LINKED');
INSERT INTO public.dependency(
  ancestor_id, descendant_id, depth,linking)
  VALUES (1, 4, 1,'STATIC_LINKED');
INSERT INTO public.dependency(
  ancestor_id, descendant_id, depth,linking)
  VALUES (1, 5, 2,'STATIC_LINKED');
INSERT INTO public.dependency(
  ancestor_id, descendant_id, depth,linking)
  VALUES (1, 6, 2,'STATIC_LINKED');
INSERT INTO public.dependency(
  ancestor_id, descendant_id, depth,linking)
  VALUES (1, 7, 2,' STATIC_LINKED');
INSERT INTO public.dependency(
  ancestor_id, descendant_id, depth,linking)
  VALUES (1, 8, 2,' STATIC_LINKED');
INSERT INTO public.dependency(
  ancestor_id, descendant_id, depth,linking)
  VALUES (1, 9, 3,' STATIC_LINKED');
INSERT INTO public.dependency(
  ancestor_id, descendant_id, depth,linking)
  VALUES (1, 10, 3,' STATIC_LINKED');
INSERT INTO public.dependency(
  ancestor_id, descendant_id, depth,linking)
  VALUES (2, 2, 0,'STATIC_LINKED');

INSERT INTO public.dependency(
  ancestor_id, descendant_id, depth,linking)
```

```
37    VALUES (2, 5, 1,'STATIC_LINKED');
38  INSERT INTO public.dependency(
39    ancestor_id, descendant_id, depth,linking)
40    VALUES (2, 6, 1,'STATIC_LINKED');
41
42  INSERT INTO public.dependency(
43    ancestor_id, descendant_id, depth,linking)
44    VALUES (2, 9, 2,' STATIC_LINKED');
45
46  INSERT INTO public.dependency(
47    ancestor_id, descendant_id, depth,linking)
48    VALUES (3, 3, 0,'STATIC_LINKED');
49  INSERT INTO public.dependency(
50    ancestor_id, descendant_id, depth,linking)
51    VALUES (3, 7, 1,'STATIC_LINKED');
52  INSERT INTO public.dependency(
53    ancestor_id, descendant_id, depth,linking)
54    VALUES (3, 8, 1,'STATIC_LINKED');
55  INSERT INTO public.dependency(
56    ancestor_id, descendant_id, depth,linking)
57    VALUES (3, 10, 2,'STATIC_LINKED ');
58
59  INSERT INTO public.dependency(
60    ancestor_id, descendant_id, depth,linking)
61    VALUES (4, 4, 0,'STATIC_LINKED');
62
63  INSERT INTO public.dependency(
64    ancestor_id, descendant_id, depth,linking)
65    VALUES (5, 5, 0,'STATIC_LINKED');
66  INSERT INTO public.dependency(
67    ancestor_id, descendant_id, depth,linking)
68    VALUES (5, 9, 1,'STATIC_LINKED');
69
70  INSERT INTO public.dependency(
71    ancestor_id, descendant_id, depth,linking)
72    VALUES (6, 6, 0,'STATIC_LINKED');
73
74  INSERT INTO public.dependency(
75    ancestor_id, descendant_id, depth,linking)
76    VALUES (7, 7, 0,'STATIC_LINKED');
77  INSERT INTO public.dependency(
78    ancestor_id, descendant_id, depth,linking)
79    VALUES (7, 10, 1,'STATIC_LINKED');
80
81  INSERT INTO public.dependency(
82    ancestor_id, descendant_id, depth,linking)
83    VALUES (8, 8, 0,'STATIC_LINKED');
```

Um einen Docker-Container für PostgreSQL-Datenbank zu erstellen, muss man im Stammverzeichnis des Repositorys eine neue Datei **.env** mit folgendem Inhalt

erstellen. Diese Datei wird von Docker verwendet, um die Konfigurationsparameter in die Umgebungsvariablen zu laden.

```
1 # DB
2 # Parameters used by db container
3 POSTGRES_DB=product_model
4 POSTGRES_USER=postgres
5 POSTGRES_PASSWORD=change_me
```

Listing 5.1: PostgreSQL-Teil von .env Datei

Insbesondere:

- **POSTGRES_DB**: Name der Datenbank, die im PostgreSQL-Container erstellt werden soll.
- **POSTGRES_USER**: Standard-Admin-Benutzer, der bei der Datenbankinitialisierung erstellt wird.
- **POSTGRES_PASSWORD**: Das Passwort des Standard-Admin-Benutzers.

Um die für die Erstellung eines PostgreSQL-Containers benötigten Dateien zu speichern, muss man einen neuen Ordner mit dem Namen **db** im Stammverzeichnis des Repositorys erstellen. Im Ordner **db** wird ein neuer Unterordner mit dem Namen **init** erstellt. Er wird die SQL-Dateien enthalten, die zur Initialisierung der PostgreSQL-Datenbank verwendet werden. Wenn PostgreSQL die Datenbank initialisiert, werden die Dateien, die sich im Init-Ordner befinden, der Reihenfolge nach ausgeführt. Unter dem Init-Ordner enthält man zwei sql-Dateien, eine ist das Datenbankschema, das in Kapitel 5.1 erstellt wurde, das andere sind die eingeführten initialen Daten, die ich als Dummy Daten eingeführt habe. Das Datenbankschema hat den Namen 00-database.sql und die Daten den Namen 01-data.sql.

Die Ordnerstruktur ist gemäß Abbildung 5.4 aufgebaut. Eine Dockerfile sollte für PostgreSQL-Datenbank erstellt werden, das Dockerfile wird vom Docker als Blauprint zur Erstellung von Docker-Images verwendet. Eine neue Datei mit dem Namen Dockerfile sollte im Ordner **db** erstellt werden, Inhalt wie folgt:

```
1 FROM postgres:alpine
2 COPY ./init/ /docker-entrypoint-initdb.d/
```

Listing 5.2: PostGraphile-Teil von .env Datei

```
/
├─ db/
|  ├─ init/
|  |  ├─ 00-database.sql
|  |  └─ 01-data.sql
|  └─ Dockerfile
├─ .env
└─ docker-compose.yml
```

Abbildung 5.2: Ordnerstruktur

Die erste Zeile FROM **postgres:alpine** gibt an, das Docker-Image auf der Basis des offiziellen PostSQL-Docker-Images zu erstellen, das in einem Alpine Linux-Container

läuft. Die zweite Zeile **COPY ./init/ /docker-entrypoint-initdb.d/** kopiert die Datenbankinistialisierungsdateien in den Ordner docker-entrypointer-initdb.d, der sich im Docker-Container befindet. Dieser Ordner wird bei der Datenbankinitialisierung von PostgreSQL gelesen und sein gesamter Inhalt wird ausgeführt.

Da Docker-Befehlszeilen sehr ausführlich mit vielen Parametern sein können, wird **Docker-Compose** verwendet, um Konfigurationsdateien dabei zu helfen, ein Netzwerk aus mehreren Containern gleichzeitig zu betreiben, anstatt alle Argumente in der Befehlszeilenschnittstelle (command line Interface) anzugeben. Deshalb müssen wir also eine neue Datei namens **docker-compose.yml** im Stammverzeichnis erstellen, um Docker-compose die gleichzeitige Ausführung mehrerer Container zu ermöglichen. Inhalt wie folgt:

```
1  version: "3.3"
2  services:
3    db:
4      container_name: product-model-db
5      restart: "no"
6      image: product-model-db
7      build:
8        context: ./db
9      # volume mount disabled for development
10     # volumes:
11     #   - db:/var/lib/postgresql/data
12     env_file:
13       - ./.env
14     ports:
15       - 5432:5432
```

Listing 5.3: docker-compose Datei

5.2 Integration der PostGraphile

Beim Implementierungsteil, damit PostGraphile eine Verbindung mit der PostgreSQL-Datenbank herstellen kann, muss die **.env**-Datei aktualisiert werden, indem DATABASE_URL hinzufügt wird.

In der **.env**-Datei soll DATABASE_URL der Syntax postgres:

```
1  DATABASE\_URL = postgres://<user>:<password>@db:5432/<db\_name>
```

Um die für die Erstellung des PostGraphile-Containers benötigten Dateien zu speichern, erstellen Sie einen neuen Ordner mit der Name **graphql** im Stammverzeichnis des Repositorys. Ähnlich wie bei PostgreSQL-Datenbank wird im Ordner

graphql eine neue Datei Dockerfile erstellt, die auch das hervorragende Plugin **Connection Filter** enthält.

Die Inhalt des Dockerfile folgt:

```
graphql:
    container_name: product-model-graphql
    restart: "no"
    image: product-model-graphql
    build:
      context: ./graphql
    env_file:
      - ./.env
    depends_on:
      - db
    ports:
      - 5433:5433
    command:
      [
        "--enhance-graphiql",
        "--connection",
        "${DATABASE_URL}",
        "--port",
        "5433",
        "--schema",
        "public",
        "--append-plugins",
        "postgraphile-plugin-connection-filter",
      ]
```

Listing 5.4: Datenbank-Teil von.env Datei

Gleichzeitig soll die Datei Docker-compose.yml auch unter Abschnitt servies aktualisieren, um den GraphQL-Dienst aufzunehmen.

5.3 GraphQL Queries

5.3.1 Querying des gesamten Komponentengraphens

Die folgende Abfrage gibt die Abhängigkeit des gesamten Komponentengraphens mit ihren id, name, version und package zurück. Um redundante Daten zu vermeiden, wird ein Filter $tiefe > 0$ (Zeile 8 und 13) verwendet.

```
query firstQuery{
 allDependencies{
  nodes{
  info:componentByDescendantId{
    id
```

```
6       name
7       license
8       dependencies1:dependenciesByAncestorId(filter: {depth: {
            greaterThan: 0}}){
9         nodes{dependencies1:componentByDescendantId{
10        id
11        name
12        license
13          dependencies2:dependenciesByAncestorId(filter: {depth: {
                greaterThan: 0}}){
14            nodes{dependencies2:componentByDescendantId{id name
                  license}}
15          }
16        }}
17      }}
18    }
19  }
20  }
```

Listing 5.5: Query von complete component graph

Das Ergebnis sieht man unter Anhang 8.1.

5.3.2 Querying eines Teil des Komponentengraphens

Die folgende Abfrage gibt die Abhängigkeit von Komponente mit id =3 und auch die dazugehörige Informationen.

```
1  query secondeQuery{
2    componentById(id:3){
3       name
4       dependencies1:
5        dependenciesByAncestorId(filter: {depth: {greaterThan: 0}})
             {
6          totalCount
7          nodes{
8          dependencies1: componentByDescendantId{
9              name
10            dependencies2:
11            dependenciesByAncestorId(filter: {depth: {greaterThan:
                  0}}) {
12          totalCount
13            nodes{
14              dependencies2: componentByDescendantId{
15                name
16              }
17            }
18          }
19        }
20
21       }
```

```
22    }
23  }
24  }
```

Listing 5.6: eines Teil des Komponentengraphens

Der Ergebnis sieht man unter Anhang 8.2.

5.3.3 Querying der CRUD-Operation

- Einfügen-Operation

Wir möchten an dem Graph eine Komponente id= 11, Name "Comp Q"und dazugehörige Informationen hinzufügen. Die GraphQL-Anweisung ist wie folgt:

```
1  mutation thirdQuery{
2    createComponent(input:{component:{
3    id:11,
4    name:"Comp Q",license:"MIT",version:"1.0.0",package:"org.q"
5  }}){component{
6      id
7      name
8      license
9      version
10     package
11 }}
12 }
13
14 mutation thirdQuery3{
15   createDependency(input:{dependency:{
16   ancestorId:8,
17   descendantId:11,
18   depth:2
19   linking:"DYNAMIC_LINKED"
20 }}){dependency{
21   ancestorId
22   descendantId
23   depth
24   linking
25
26 }}
27 }
28
29 mutation thirdQuery3{
30   createDependency(input:{dependency:{
31   ancestorId:1,
32   descendantId:11,
33   depth:3
34   linking:"DYNAMIC_LINKED"
35 }}){dependency{
```

```
36    ancestorId
37    descendantId
38    depth
39    linking
40
41  }}
42  }
```

Listing 5.7: Create Operation

- Aktualisierung-Operation

Es sollte der Name der Komponente mit id =11 von "Comp Q"in Comp X aktualisiert werden.

```
1  # Update-Operation: mutate component with id=11. Change the
       components name from #'Comp Q' to 'Comp X'
2    mutation fourQuery{
3      updateComponentById(
4      input:{id:11,componentPatch:{name:"Comp X"}}
5    ){component{
6      id
7      name
8     }
9    }
10 }
```

Listing 5.8: Aktualisierung- Komponentengraphens

- Löschen-Operation

Wir möchten im Graphen eine Komponente id= 11, Name "Comp Q"und dazugehörige Informationen löschen.

```
1
2   mutation MyMutation {
3    deleteDependencyByAncestorIdAndDescendantId(input: {ancestorId
        :8, descendantId: 11}) {
4      componentByDescendantId {
5        id
6        license
7        name
8      }
9    }
10 }
11 mutation MyMutation {
12
13   deleteDependencyByAncestorIdAndDescendantId(input: {ancestorId
        :1, descendantId: 11}) {
14     componentByDescendantId {
15       id
```

```
16        license
17        name
18      }
19    }
20  }
21
22  mutation delete {
23    deleteComponentById (input:{id:11}){
24      component{id name}
25    }
26  }
27
28  //testen original componente Graph
29  query original {
30    allComponents {
31      nodes {
32        id
33        license
34        name
35      }
36    }
37  }
```

Listing 5.9: Löschen-Operation des Komponentengraphens

6 Auswertung

Zu Beginn der Bachelorarbeit wurden mehrere verschiedene funktionale Anforderungen festgelegt, anhand derer das Endergebnis bewertet werden konnte. Danach wurde jede Anforderung analysiert, ob sie erfolgreich umgesetzt wurde und wenn nicht, was die Gründe hinter dem Scheitern waren.

6.1 Relationales Datenbankschema zur Speicherung von Graphen

6.1.1 A1: Relationale Abbildung von Graphen

Diese Voraussetzung ist erfüllt. Durch den Vergleich von drei Konzepten zur Speicherung von Graphen in einer relationalen Datenbank in Kapitel 4 gelingt die Closure Table als unser Konzept zur Speicherung von Graphen in PostgreSQL-Datenbank.

6.1.2 A2: Kompatibilität von Datenbankschema (A1) mit PostGraphile

Diese Voraussetzung ist erfüllt. Das Datenbankschema aus A1 ist kompatibel mit PostGraphile.

PostGraphile hat erfolgreich eine Standard-GraphQL-API-Schicht eingerichtet. Der Endpunkt kann in einem Webbrowser geöffnet werden, sodass man über GraphiQL(Visual Graphql Browser) auf die PostgreSQL-Datenbank zugreifen kann.

6.2 GraphQL-Schnittstelle

6.2.1 A3:CRUD-Operation

Diese Voraussetzung ist erfüllt. PostGraphile fügte dem Schema von PostgreSQL-Datenbank für jede Tabelle automatisch eine CRUD-Mutation hinzu.

6.2.2 A4: Querying des gesamten Komponentengraphens

Diese Voraussetzung ist erfüllt. In Kapitel 5.5.1 konnte schon das Ergebnis gezeigt werden, dass die Anfragen an den gesamten Komponentengraphen abgeschlossen wurden. Dies kann eingeschränkt werden, wenn man die hierarchische Komponente immer wieder nach Abhängigkeiten abfragen müsste. Da GraphQL eigentlich nicht wirklich Rekursion unterstützt, war dies eine bewusste Designentscheidung der GraphQL-Entwickler. Rekursive Abfragen sind eine großartige Möglichkeit, versehentlich eine unbegrenzte Last auf den Servern zu erzeugen.

6.2.3 A5: Querying eines Teils des Komponentengraphens

Diese Voraussetzung ist erfüllt. Die Abbildung aus Kapitel 5.3.2 zeigt schon die Ergebnisse.

6.2.4 A6: Querying von Lizenzinformation

Diese Voraussetzung ist erfüllt. Die Lizenzinformation wurde problemlos durch PostGraphiql zurückgegeben.

7 Zusammenfassung

Die weit verbreitete Wiederverwendung von Software ist immer üblicher geworden, insbesondere durch den weit verbreiteten FLOSS. Die zunehmende Anzahl von Softwarekomponenten aus verschiedenen Quellen hat jedoch zu einer komplexen Komponentenstruktur für Softwareprojekt geführt. Das Lizenzproblem erfordert ebenfalls Aufmerksamkeit. Weit entfernt von den Lehren aus der Lieferkette von Cisco, verklagte FSF Cisco wegen verschiedener Produkte, die unter der Marke Linksys verkauft werden, wegen Verstoßes gegen die Lizenzbestimmungen für urheberrechtlich geschützte Programme von FSF, einschließlich GCC, GNU Binutils und GNU C Library[14]. Schließlich ernannte Cisco einen Direktor, um die Linksys-Produkte entsprechend der Lizenzen für freie Software, sicherzustellen und die nicht bekannt gegebene finanzielle Spende von Cisco an FSF wurde beendet. Vor kurzem gab es einen großen Vorfall, bei dem Oracle 8,8 Milliarden US-Dollar von Google forderte. Hintergrund war, dass das Urheberrecht am GPL-Projekt von OpenJDK Oracle gehörte. Joshua Bloch, der zu dieser Zeit bei Google arbeitete, kopierte 9 Codezeilen direkt von OpenJDK in Googles Android-Projekt. Der Punkt ist, dass das Android-Projekt nicht GPL-kompatibel lizenziert ist, was das Urheberrecht von Oracle verletzte [16]. Dieser Vorfall zeigt uns, dass die Lizenzsicherheit der Software-Lieferkette nicht nur darin besteht, Komponenten einzuführen, sondern dass die kopierten und eingefügten Code-Schnipsel auch Punkte sind, die Aufmerksamkeit erfordern. Die Verwendung von Open Source-Komponenten, die nicht mit der Open Source-Lizenzvereinbarung übereinstimmen, führt zu potenziellen Rechtsstreitigkeiten, von denen die häufigsten Verstöße gegen die GPL-Lizenz (GNU General Public License) sind. Wenn ein ignoranter Benutzer eine nicht konforme Lizenz verwendet, führt dies zu einem Produktrückgang oder das Produkt wird infiziert und gezwungen, den Quellcode seiner eigenen kommerziellen Produkte zu öffnen. Daher ist die Abfrage der Abhängigkeit der Komponente von entscheidender Bedeutung.

Die grundlegende Aufgabe dieser Arbeit bestand darin, eine Lösung für das schwierige Problem der Implementierung der in Kapitel 1 erwähnten Abfragekomponenten zu finden. Für die Abhängigkeiten zwischen den Abfragekomponenten verwendet diese Bachelorarbeit die Closure Table Konzeption aus Kapitel 4,

um eine PostgreSQL-Datenbank zu entwerfen, und die Graphql-Abfragesprache, um die Abfragen durchzuführen. Aus den Ergebnissen (Kapitel 5) geht hervor, dass die Komponentenabhängigkeiten gut abgefragt werden. Aber es gibt einige Unzulänglichkeiten, die ich beobachtet habe. Um die Abhängigkeiten zwischen Komponenten abfragen zu können, kann GraphQL verwendet werden, aber hierarchische Abfragen sind durch GraphQL begrenzt, da GraphQL selbst nicht rekursiv sein soll. Die Anforderungen in Bezug auf diese Bachelorarbeit umfassen alle zuvor beschriebenen und diskutierten Funktionen, die einzeln implementiert wurden. Mögliche Verbesserungen und Ergänzungen in zukünftigen Implementierungsschritten sind unter anderem: das gesamte Datenmodell in einem Datenbank-Schema umzusetzen und aufzeigen wie es mit GraphQL funktionieren kann. Außerdem ist GraphQL großartig, da es uns erlaubt, in einem deklarativen Stil zu arbeiten, indem es ermöglicht, nur die Informationen oder Operationen auszuwählen, die man braucht, aber es ist wie jedes andere Tool auch kein silver bullet[1]. Das Hauptmerkmal von GraphQL besteht darin, dass Sie eine Abfrage senden können, die nur die Informationen angibt, die Sie benötigen und dadurch genau diese zu erhalten. Sie können dies aber auch mit REST erreichen, indem Sie den Namen der Felder, die Sie in der URL verwenden möchten, übergeben (wobei Sie die Parsing und Rückgabe-Logik selbst implementieren).

GraphQL wird einige Aufgaben komplexer machen, die Verwendung von GraphQL in einer einfachen Anwendung (z.B. einer Anwendung, die einige wenige Felder jedes Mal auf die gleiche Weise verwendet) ist nicht empfehlenswert, da dies die Komplexität erhöht. Dies ist aus Wartungssicht nicht gut. Wenn ein Client eine Anfrage sendet, in der viele Felder und Ressourcen abfragt werden, zum Beispiel wie "Gib mir die Abhängigkeit von Informationen über die Versionen, in denen eine Version für alle Componente ". Wir können einfach jede gewünschte Abfrage ausführen lassen. Eine GraphQL-API muss sorgfältig entworfen werden. Es geht nicht nur darum, sie auf eine REST-API oder eine Datenbank zu setzen. Bei komplexen Abfragen ist eine REST-API möglicherweise einfacher zu entwerfen, da REST mehrere Endpunkte für bestimmte Abfragen optimieren kann, um die Daten auf effiziente Weise abzurufen.

Im Rahmen dieser Arbeit wurde gezeigt, wie Komponenten-Graphen in einer relationalen Datenbank (PostgreSQL) abgebildet werden können und als GraphQL-Schnittstelle bereitgestellt werden können. Hierzu wurden unterschiedliche Konzepte zur Abbildung von Graphen in relationalen Datenbanken gegenübergestellt. Desweiteren wurde die Kompatibilität, als auch der Umgang, mit einem Komponentengraphen über eine GraphQL-Schnittstelle dargestellt.

[1] https://en.wikipedia.org/wiki/No_Silver_Bullet

8 Anhang

8.1 Result von fristQuery (5.3.1)


```
{
  "data": {
    "allDependencies": {
      "nodes": [
        {
          "info": {
            "id": 1,
            "name": "Comp A",
            "license": "MIT",
            "dependencies1": {
              "nodes": [
                {
                  "dependencies1": {
                    "id": 2,
                    "name": "Comp B",
                    "license": "Apache",
                    "dependencies2": {
                      "nodes": [
                        {
                          "dependencies2": {
                            "id": 5,
                            "name": "Comp E",
                            "license": "MIT"
                          }
                        },
                        {
                          "dependencies2": {
                            "id": 6,
                            "name": "Comp F",
                            "license": "MIT"
                          }
                        },
                        {
                          "dependencies2": {
                            "id": 9,
                            "name": "Comp I",
```

```
                                "license": "MIT"
                            }
                        }
                    ]
                }
            }
        },
        {
            "dependencies1": {
                "id": 3,
                "name": "Comp C",
                "license": "MIT",
                "dependencies2": {
                    "nodes": [
                        {
                            "dependencies2": {
                                "id": 7,
                                "name": "Comp G",
                                "license": "MIT"
                            }
                        },
                        {
                            "dependencies2": {
                                "id": 8,
                                "name": "Comp H",
                                "license": "GPL"
                            }
                        },
                        {
                            "dependencies2": {
                                "id": 10,
                                "name": "Comp K",
                                "license": "unknown"
                            }
                        }
                    ]
                }
            }
        },
        {
            "dependencies1": {
                "id": 4,
                "name": "Comp D",
                "license": "MIT",
                "dependencies2": {
                    "nodes": []
                }
            }
        },
        {
            "dependencies1": {
```

```
                    "id": 5,
                    "name": "Comp E",
                    "license": "MIT",
                    "dependencies2": {
                      "nodes": [
                        {
                          "dependencies2": {
                            "id": 9,
                            "name": "Comp I",
                            "license": "MIT"
                          }
                        }
                      ]
                    }
                  }
                },
                {
                  "dependencies1": {
                    "id": 6,
                    "name": "Comp F",
                    "license": "MIT",
                    "dependencies2": {
                      "nodes": []
                    }
                  }
                },
                {
                  "dependencies1": {
                    "id": 7,
                    "name": "Comp G",
                    "license": "MIT",
                    "dependencies2": {
                      "nodes": [
                        {
                          "dependencies2": {
                            "id": 10,
                            "name": "Comp K",
                            "license": "unknown"
                          }
                        }
                      ]
                    }
                  }
                },
                {
                  "dependencies1": {
                    "id": 8,
                    "name": "Comp H",
                    "license": "GPL",
                    "dependencies2": {
                      "nodes": []
```

```
                    }
                  }
                },
                {
                  "dependencies1": {
                    "id": 9,
                    "name": "Comp I",
                    "license": "MIT",
                    "dependencies2": {
                      "nodes": []
                    }
                  }
                },
                {
                  "dependencies1": {
                    "id": 10,
                    "name": "Comp K",
                    "license": "unknown",
                    "dependencies2": {
                      "nodes": []
                    }
                  }
                }
              ]
            }
          }
        },
        {
          "info": {
            "id": 2,
            "name": "Comp B",
            "license": "Apache",
            "dependencies1": {
              "nodes": [
                {
                  "dependencies1": {
                    "id": 5,
                    "name": "Comp E",
                    "license": "MIT",
                    "dependencies2": {
                      "nodes": [
                        {
                          "dependencies2": {
                            "id": 9,
                            "name": "Comp I",
                            "license": "MIT"
                          }
                        }
                      ]
                    }
                  }
```

```
                },
                {
                  "dependencies1": {
                    "id": 6,
                    "name": "Comp F",
                    "license": "MIT",
                    "dependencies2": {
                      "nodes": []
                    }
                  }
                },
                {
                  "dependencies1": {
                    "id": 9,
                    "name": "Comp I",
                    "license": "MIT",
                    "dependencies2": {
                      "nodes": []
                    }
                  }
                }
              ]
            }
          }
        },
        {
          "info": {
            "id": 3,
            "name": "Comp C",
            "license": "MIT",
            "dependencies1": {
              "nodes": [
                {
                  "dependencies1": {
                    "id": 7,
                    "name": "Comp G",
                    "license": "MIT",
                    "dependencies2": {
                      "nodes": [
                        {
                          "dependencies2": {
                            "id": 10,
                            "name": "Comp K",
                            "license": "unknown"
                          }
                        }
                      ]
                    }
                  }
                },
                {
```

```
                    "dependencies1": {
                      "id": 8,
                      "name": "Comp H",
                      "license": "GPL",
                      "dependencies2": {
                        "nodes": []
                      }
                    }
                  },
                  {
                    "dependencies1": {
                      "id": 10,
                      "name": "Comp K",
                      "license": "unknown",
                      "dependencies2": {
                        "nodes": []
                      }
                    }
                  }
                ]
              }
            }
          },
          {
            "info": {
              "id": 4,
              "name": "Comp D",
              "license": "MIT",
              "dependencies1": {
                "nodes": []
              }
            }
          },
          {
            "info": {
              "id": 5,
              "name": "Comp E",
              "license": "MIT",
              "dependencies1": {
                "nodes": [
                  {
                    "dependencies1": {
                      "id": 9,
                      "name": "Comp I",
                      "license": "MIT",
                      "dependencies2": {
                        "nodes": []
                      }
                    }
                  }
                ]
```

```
            }
          }
        },
        {
          "info": {
            "id": 6,
            "name": "Comp F",
            "license": "MIT",
            "dependencies1": {
              "nodes": []
            }
          }
        },
        {
          "info": {
            "id": 7,
            "name": "Comp G",
            "license": "MIT",
            "dependencies1": {
              "nodes": [
                {
                  "dependencies1": {
                    "id": 10,
                    "name": "Comp K",
                    "license": "unknown",
                    "dependencies2": {
                      "nodes": []
                    }
                  }
                }
              ]
            }
          }
        },
        {
          "info": {
            "id": 8,
            "name": "Comp H",
            "license": "GPL",
            "dependencies1": {
              "nodes": []
            }
          }
        },
        {
          "info": {
            "id": 9,
            "name": "Comp I",
            "license": "MIT",
            "dependencies1": {
              "nodes": []
```

```
            }
          }
        },
        {
          "info": {
            "id": 10,
            "name": "Comp K",
            "license": "unknown",
            "dependencies1": {
              "nodes": []
            }
          }
        },
        {
          "info": {
            "id": 2,
            "name": "Comp B",
            "license": "Apache",
            "dependencies1": {
              "nodes": [
                {
                  "dependencies1": {
                    "id": 5,
                    "name": "Comp E",
                    "license": "MIT",
                    "dependencies2": {
                      "nodes": [
                        {
                          "dependencies2": {
                            "id": 9,
                            "name": "Comp I",
                            "license": "MIT"
                          }
                        }
                      ]
                    }
                  }
                },
                {
                  "dependencies1": {
                    "id": 6,
                    "name": "Comp F",
                    "license": "MIT",
                    "dependencies2": {
                      "nodes": []
                    }
                  }
                },
                {
                  "dependencies1": {
                    "id": 9,
```

```
                    "name": "Comp I",
                    "license": "MIT",
                    "dependencies2": {
                      "nodes": []
                    }
                  }
                }
              ]
            }
          }
        },
        {
          "info": {
            "id": 5,
            "name": "Comp E",
            "license": "MIT",
            "dependencies1": {
              "nodes": [
                {
                  "dependencies1": {
                    "id": 9,
                    "name": "Comp I",
                    "license": "MIT",
                    "dependencies2": {
                      "nodes": []
                    }
                  }
                }
              ]
            }
          }
        },
        {
          "info": {
            "id": 6,
            "name": "Comp F",
            "license": "MIT",
            "dependencies1": {
              "nodes": []
            }
          }
        },
        {
          "info": {
            "id": 9,
            "name": "Comp I",
            "license": "MIT",
            "dependencies1": {
              "nodes": []
            }
          }
```

```
        },
        {
          "info": {
            "id": 3,
            "name": "Comp C",
            "license": "MIT",
            "dependencies1": {
              "nodes": [
                {
                  "dependencies1": {
                    "id": 7,
                    "name": "Comp G",
                    "license": "MIT",
                    "dependencies2": {
                      "nodes": [
                        {
                          "dependencies2": {
                            "id": 10,
                            "name": "Comp K",
                            "license": "unknown"
                          }
                        }
                      ]
                    }
                  }
                },
                {
                  "dependencies1": {
                    "id": 8,
                    "name": "Comp H",
                    "license": "GPL",
                    "dependencies2": {
                      "nodes": []
                    }
                  }
                },
                {
                  "dependencies1": {
                    "id": 10,
                    "name": "Comp K",
                    "license": "unknown",
                    "dependencies2": {
                      "nodes": []
                    }
                  }
                }
              ]
            }
          }
        },
        {
```

```
            "info": {
              "id": 7,
              "name": "Comp G",
              "license": "MIT",
              "dependencies1": {
                "nodes": [
                  {
                    "dependencies1": {
                      "id": 10,
                      "name": "Comp K",
                      "license": "unknown",
                      "dependencies2": {
                        "nodes": []
                      }
                    }
                  }
                ]
              }
            }
          },
          {
            "info": {
              "id": 8,
              "name": "Comp H",
              "license": "GPL",
              "dependencies1": {
                "nodes": []
              }
            }
          },
          {
            "info": {
              "id": 10,
              "name": "Comp K",
              "license": "unknown",
              "dependencies1": {
                "nodes": []
              }
            }
          },
          {
            "info": {
              "id": 4,
              "name": "Comp D",
              "license": "MIT",
              "dependencies1": {
                "nodes": []
              }
            }
          },
          {
```

```
            "info": {
              "id": 5,
              "name": "Comp E",
              "license": "MIT",
              "dependencies1": {
                "nodes": [
                  {
                    "dependencies1": {
                      "id": 9,
                      "name": "Comp I",
                      "license": "MIT",
                      "dependencies2": {
                        "nodes": []
                      }
                    }
                  }
                ]
              }
            }
          },
          {
            "info": {
              "id": 9,
              "name": "Comp I",
              "license": "MIT",
              "dependencies1": {
                "nodes": []
              }
            }
          },
          {
            "info": {
              "id": 6,
              "name": "Comp F",
              "license": "MIT",
              "dependencies1": {
                "nodes": []
              }
            }
          },
          {
            "info": {
              "id": 7,
              "name": "Comp G",
              "license": "MIT",
              "dependencies1": {
                "nodes": [
                  {
                    "dependencies1": {
                      "id": 10,
                      "name": "Comp K",
```

```
                    "license": "unknown",
                    "dependencies2": {
                      "nodes": []
                    }
                  }
                }
              ]
            }
          }
        },
        {
          "info": {
            "id": 10,
            "name": "Comp K",
            "license": "unknown",
            "dependencies1": {
              "nodes": []
            }
          }
        },
        {
          "info": {
            "id": 8,
            "name": "Comp H",
            "license": "GPL",
            "dependencies1": {
              "nodes": []
            }
          }
        }
      ]
    }
  }
}
```

Listing 8.1: Result von fristQuery (5.3.1)

8.2 Result von secondQuery (5.3.2)

```
{
  "data": {
    "componentById": {
      "name": "Comp C",
      "dependencies1": {
\begin{lstlisting}[caption= Result von fristQuery (5.3.1)]
        "nodes": [
          {
            "dependencies1": {
              "name": "Comp G",
              "dependencies2": {
```

```
                "totalCount": 1,
                "nodes": [
                  {
                    "dependencies2": {
                      "name": "Comp K"
                    }
                  }
                ]
              }
            }
          },
          {
            "dependencies1": {
              "name": "Comp H",
              "dependencies2": {
                "totalCount": 1,
                "nodes": [
                  {
                    "dependencies2": {
                      "name": "Comp X"
                    }
                  }
                ]
              }
            }
          },
          {
            "dependencies1": {
              "name": "Comp K",
              "dependencies2": {
                "totalCount": 0,
                "nodes": []
              }
            }
          }
        ]
      }
    }
  }
```


Listing 8.2: Result von secondQuery (5.3.2)

Literaturverzeichnis

[1] Bill Karwin. *SQL Antipatterns, Avoiding the Pitfalls of Database Programming, The Pragmatic Bookshelf*. 2010.

[2] Andrew McGregor. *Graph Stream Algorithms: A Survey*. 2014. URL: https://dl.acm.org/doi/10.1145/2627692.2627694.

[3] Eve Porcello Alex Banks. *Learning GrapQL- Declarative data fetching for modern web apps*. 2018. Kap. Graph Theory, S. 27–30.

[4] Nikolay Harutyunyan, Andreas Bauer und Dirk Riehle. "Industry Requirements for FLOSS Governance Tools to Facilitate the Use of Open Source Software in Commercial Products". In: *Journal of Systems and Software* (2019), S. 1–44. DOI: 10.1016/j.jss.2019.08.001. URL: https://osr.cs.fau.de/wp-content/uploads/2019/08/jss-2019-harutyunya-bauer-riehle.pdf.

[5] *30 minutes to understand the core concepts of graphql*. URL: https://developpaper.com/30-minutes-to-understand-the-core-concepts-of-graphql/.

[6] *A Brief History of PostgreSQL*. URL: https://www.postgresql.org/docs/8.4/history.html.

[7] *Einfuehrung: Graphentheorie*. URL: https://www.ingenieurkurse.de/unternehmensforschung/graphentheorie/einfuehrung-graphentheorie.html.

[8] *Facebook. GraphQL.* URL: https://graphql.org/learn/best-practices/.

[9] *GraphQL community. How to GraphQL: The Full stack Tutorial for GraphQL*. URL: https://www.howtographql.com/.

[10] *GraphQL: A query language for your API.* URL: www.graphQL.org/.

[11] Bill Karwin. "SQL Antipatterns: Avoiding the Pitfalls of Database Programming (Pragmatic Programmers)". In: Kap. 3, S. 18–33.

[12] *Koenigsberger Bruckenproblem*. URL: https://mathepedia.de/Koenigsberger_Brueckenproblem.html.

[13] Prof. Dr. Klaus Meyer-Wegener. *Vorlesung Implementierung von Datenbanksystemen; 8. Transaktionen*.

[14] *More background about the Cisco case*. URL: https://www.fsf.org/licensing/2008-12-cisco-complaint.

[15] *Multigraphen*. URL: https://lehrerfortbildung-bw.de/u_matnatech/imp/gym/bp2016/fb1/6_m2_aug/2_kopiervorlagen/2_multi/.

[16] *Oracle seeks 8.8 billion for Google's use of Java in Android*. URL: https://www.cio.com/article/3048813/oracle-seeks-93-billion-for-googles-use-of-java-in-android.html.

[17] *Relationale Datenbank*. URL: https://de.wikipedia.org/wiki/Relationale_Datenbank.

[18] *Was sind eigentlich relationale Datenbanken?* URL: https://t3n.de/news/eigentlich-relationale-datenbanken-683688/.

Abbildungsverzeichnis

Listings