Hochschule
Bonn-Rhein-Sieg
University of Applied Sciences

Fachbereich Informatik
Department of Computer Science

Bachelorarbeit

Studiengang Bachelor Computer Science

Werkzeuggestütztes Ermitteln von Domain-Grenzen bei der Zergliederung monolithischer Anwendungen auf der Basis von Beziehungen in der darunterliegenden Datenbank

von

Kai Klemmer

Erstprüfer	Prof. Dr. Sascha Alda
Zweitprüfer	Prof. Dr. Manfred Kaul
Betreuer	Thomas Bayer predic8 GmbH
eingereicht am	08.08.2023

Name: Kai Klemmer
Adresse: Adelheidisplatz 20
53229 Bonn

Erklärung

Ich versichere an Eides statt, die von mir vorgelegte Arbeit selbstständig verfasst zu haben.

Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Arbeiten anderer entnommen sind, habe ich als entnommen kenntlich gemacht. Sämtliche Quellen und Hilfsmittel, die ich für die Arbeit benutzt habe, sind angegeben.

Die Arbeit hat mit gleichem Inhalt bzw. in wesentlichen Teilen noch keiner anderen Prüfungsbehörde vorgelegen.

....
Ort, Datum Unterschrift

Inhaltsverzeichnis

Abbildungsverzeichnis vi

Abkürzungsverzeichnis viii

1 Einleitung 1
- 1.1 Motivation und Problemstellung . . . 1
- 1.2 Zielsetzung . . . 1
- 1.3 Abgrenzung . . . 1
- 1.4 Struktur der Thesis . . . 1

2 Grundlagen 3
- 2.1 Kohäsion und Kopplung . . . 3
- 2.2 Domain-Driven-Design . . . 3
- 2.3 DDD im Kontext: Monolith und Microservices . . . 4
- 2.4 Definition: monolithische Software-Architektur . . . 4
- 2.5 Vor- und Nachteile eines Monolithen . . . 4
- 2.6 Definition: Microservices . . . 5
- 2.7 Abgrenzung: Microservices und SOA . . . 5
- 2.8 Vor- und Nachteile von Microservices . . . 5
- 2.9 Skalierbarkeit von Monolithen und Microservices . . . 6
- 2.10 Beispiel: Versandhaus als monolithische Anwendung . . . 6
- 2.11 Beispiel: Versandhaus als Microservices . . . 7
- 2.12 Vorgehen zum Zerlegen eines Monolithen . . . 8
- 2.13 Datenbanken: Abfragen und Persistierung . . . 9
  - 2.13.1 Abfragen und Tabellenverknüpfungen . . . 9
  - 2.13.2 Persistierung von Daten . . . 10
- 2.14 Fazit Monolithen und Microservices . . . 11

3 Analyse und Konzeption 12
- 3.1 Konzeption des Prototyps (ist-Zustand) . . . 12
  - 3.1.1 Defintion: Graph, Knoten und Kanten . . . 14
  - 3.1.2 vom SQL-Log bis zum Graph . . . 14
  - 3.1.3 Zwischenformat: Datenaufbereitung . . . 14
  - 3.1.4 Auswertung: Datenanalyse . . . 15
    - 3.1.4.1 Verfahren der relativen Häufigkeit . . . 15
- 3.2 Interner Datensatz . . . 17
- 3.3 Open Source CRM Datensatz . . . 20
- 3.4 Einführung der Neuen Verfahren . . . 20
  - 3.4.1 Nachbarschaftsverfahren . . . 21
  - 3.4.2 Join-Kontext . . . 21
    - 3.4.2.1 Max-Join . . . 21
    - 3.4.2.2 Fuzzy Join . . . 22
    - 3.4.2.3 Sub Joins . . . 22
    - 3.4.2.4 Gleiche und ungleiche Join-Ketten . . . 23
    - 3.4.2.5 LCOM: Lack Of Cohesion . . . 24
    - 3.4.2.6 Join-Stärke . . . 24
  - 3.4.3 Ähnlichkeitsverfahren . . . 25
- 3.5 Konzeption: Soll-Zustand . . . 26
  - 3.5.1 User Stories . . . 26
  - 3.5.2 Technologien: Ist-Zustand vs. Soll-Zustand . . . 27

4 Anwendung **30**
4.1 Refactoring Backend: IOUtils . . . 30
4.2 Refactoring Backend: Erweiterung des Datenmodels . . . 31
4.3 Refactoring Backend: DataPrepper . . . 32
4.4 Refactoring Backend: DataAnalyzer . . . 34
4.5 Refactoring Backend: DataVisualizer . . . 35
4.6 Refactoring Frontend . . . 36
4.7 Erweiterung des Backends . . . 37
4.7.1 Implementierung der Häufigkeit von Joins . . . 37
4.7.2 Implementierung Join-Stärke . . . 38
4.7.3 Implementierung des Nachbarschaftsverfahren . . . 38
4.7.4 Implementierung LCOM . . . 39
4.7.5 Implementierung Ähnlichkeitsverfahren . . . 40
4.8 Join Aufbereitung . . . 42
4.9 Erweiterung des Frontends . . . 42
4.10 Bedienung der Anwendung . . . 43

5 Evaluation **44**
5.1 Profil des Datensatzes . . . 44
5.2 Vergleich der Ergebnisse . . . 44
5.2.1 Nachbarschaftsverfahren . . . 44
5.2.2 LCOM . . . 44
5.2.2.1 LCOM ohne Max-Join . . . 45
5.2.3 Join-Kontext: relative Häufigkeit von Joins . . . 45
5.2.3.1 Join-Kontext: Relative Häufigkeit ohne Max-Join . . . 46
5.2.4 Ähnlichkeitsverfahren . . . 46
5.3 Verfeinerung: LCOM . . . 47
5.4 Fazit . . . 47

6 Zusammenfassung **48**
6.1 Offene Punkte . . . 48
6.2 Ausblick . . . 48

7 Literaturverzeichnis **49**

8 Anhang **52**

Abbildungsverzeichnis

2.1 Darstellung des Scale Cubes, nach Martin L. Abbott (2015) 6
2.2 Beispiel für eine monolithische Software-Architektur (Erstelt mit draw.io) . . 7
2.3 Beispiel für einer Microservices Software-Architektur (Erstelt mit draw.io) . 8
2.4 Visualisierung der Ergebnismenge von Join-Operationen (Erstelt mit draw.io) 10
2.5 Beispiel einer mit `@Entity`-annotierten Klasse 10

3.1 Menüleiste des Frontends 12
3.2 Darstellung der Daten als Graph im Frontend 13
3.3 Darstellung des Graphs nach Anheben des Thresholds 13
3.4 Hilfe-Menü des Bash-Skipts 14
3.5 Beispiel für eine Join-Abfrage in SQL 14
3.6 Abbildung der Architektur des Prototyps (Erstellt mit draw.io) 17
3.7 Worst-Case Szenario des Nachbarschaftsverfahren 21
3.8 Darstellung von Sub-Joins 23
3.9 Beispiel: Ungleiche Joins 23
3.10 Darstellung vom Clustering durch das Ähnlichkeitsverfahren 26
3.11 Use Case Diagramm (Erstellt mit PlantUML) 27
3.12 Abbildung der Architektur des Monolith-Killers MK2 (Erstellt mit draw.io) . . 29

4.1 Methode zum Zerteilen des Date-Pfads 30
4.2 Methode zum Erhalten des Datei-Pfads 30
4.3 Ausschnitt der Node-Klasse 31
4.4 Ausschnitt der Join-Klasse 31
4.5 Main-Methode des DP 32
4.6 Beispiel für einen Join aus den Testdaten 32
4.7 Methode zum Erhalten der Abbildung von Alias auf Tabellenname 33
4.8 Methode zum Erhalten der Paare 33
4.9 Methode zum zurück mappen von Alias auf Tabellenname 33
4.10 Ausschnitt der ausgelagerten Funktion nach dem Refactoring 34
4.11 Methode zur Berechnung des Kantengewichts 35
4.12 Methode zum Hinzufügen bzw. Entfernen von Kanten 36
4.13 Step-Methode in VisJS 37
4.14 Methode zum Vergleichen zweier Joins 38
4.15 Methode zum Finden aller benachbarten Kanten 39
4.16 Methode zum Berechnen von LCOM 40
4.17 Teil 1 des Cluster-Algorithmus 40
4.18 Methode zum Berechnen Ähnlichkeit zwischen zwei Knoten 41
4.19 Teil 2 des Cluster-Algorithmus 41
4.20 Methode zur Aufbereitung der Fuzzy-Joins 42
4.21 Clustering der Knoten im Frontend 42
4.22 Vereinfachte Darstellung der Methode zur Wahl eines Verfahrens im DA . . 43
4.23 Vereinfachte Darstellung eines Skripts zur Ausführung des DA 43

8.1 Firmenanwendung vor Anheben des Thresholds 52
8.2 Firmenanwendung nach dem ersten Anheben des Threshold 52
8.3 Firmenanwendung nach inkrementellen Anheben des Threshold 53
8.4 Firmenanwendung nach weiterem Anheben des Threshold 53
8.5 Starkzusammenhängender Kern Firmenanwendung 54
8.6 CRM intial 54
8.7 CRM vor dem Setzen des ersten Schnitts 55

8.8 CRM mit drei identifizierten Domänen 55
8.9 DA Main-Methode vor den Refactoring 56
8.10 Beispiel für Aufteilung der Daten in der Option-API 57
8.11 Zugriff auf das DOM-Element 58

Abkürzungsverzeichnis

DA DataAnalyzer 15, 30, 34, 35, 42, 43

DDD Domain-Driven-Design 3, 4, 8, 11

DDL Data Definition Language 9

DML Domain Manipulation Language 9

DP DataPrepper 14, 15, 18, 20, 21, 30, 32, 34, 42

DV DataVisualizer 35

JP JoinPrepper 42

JPA Jakarta Persistence Layer 10

LCOM Lack of Cohesion 24, 39, 44, 45, 47

ORM Object-Relation-Model 10

SOA Service Oriented Architecture 5

SQL Sequel Language 3, 9, 14, 15, 17, 18, 20, 28, 32, 33

SRP Single Responsibility Pattern 5, 24

VCS Version Control System 9

1 Einleitung

In der Software-Entwicklung folgt ein Paradigma dem anderen. Die Entwicklung der Suchanfragen von „Microservices“ und „Service Oriented Architecture“ zeigen, dass Microservices an Popularität gewonnen haben (Google (2023)). Entwickler tendieren dazu, Microservices, statt eine SOA umzusetzen. Diese Beobachtung wird weiterhin von der "The State of Developer Ecosystem" Umfrage von JetBrains untermauert. Dort gaben im Jahr 2021 88% der Entwickler an, ihr System nach einem Microservices-Ansatz zu entwickeln. Dahingegen gaben 38% der Entwickler an, einen Service orientierten Ansatz bei der Entwicklung ihres Systems zu wählen (JetBrains (2021)).

1.1 Motivation und Problemstellung

Für Unternehmen, welche von einer monolithischen Anwendung weg, hin zu Microservices wollen, kann ein Werkzeug zum Finden von Domain-Grenzen bei der Beratung helfen. Ein Werkzeug welches Domain-Grenzen identifiziert, kann Hinweise darüber liefern, ob eine Anwendung zerlegt, refaktoriert oder neu-implementiert werden sollte. Das Finden von Domain-Grenzen einer monolithischen Anwendung und somit die Zerlegung in Microservices ist keine Trivialität (Dehghani (2018)). Das Verfahren ist sowohl theoretisch in der Planung als auch in der technischen Umsetzung teuer (Dehghani (2018)).

1.2 Zielsetzung

Ein Prototyp zum Unterstützen Finden von Domain-Grenzen existiert bereits und wurde vom Autor dieser Arbeit während des Praxissemesters im Sommer 2022 in Kooperation mit der predic8 GmbH implementiert. Dieser Prototyp schlug einen Schnitt für eine firmeninterne Anwendung vor, welcher von den beteiligten für sinnvoll empfunden wurde.

Das Verfahren, welches verwendet wurde um die Domain-Grenzen zu identifizieren, basierte auf der Auswertung von Join-Abfragen, die beim Zugriff der Anwendung auf die darunterliegende Datenbank generiert wurden. Im Rahmen dieser Bachelorarbeit werden verschiedene Verfahren implementiert. Die daraus entstehenden Ergebnisse sollen anhand der Qualität des jeweils erzeugten Schnitts geprüft werden. Weiterhin wird geprüft, ob die verwendeten Verfahren verbessert werden können.

1.3 Abgrenzung

Diese Arbeit beschäftigt sich ausschließlich mit der Entwicklung eines Werkzeugs, welches bei der Entscheidung, ob und wie eine monolithische Anwendung in Microservices zerlegt werden kann hilft. Die konkreten Schritte welche nötig sind um eine technische Zerlegung vorzunehmen, werden nicht behandelt. Dafür existieren Design-Muster, welche in Kapitel 2 vorgestellt werden.

1.4 Struktur der Thesis

In Kapitel 2 werden die wichtigsten Konzepte vorgestellt, welche für das Verständnis der kommenden Kapitel (3,4,5) benötigt werden.

Daraufhin wird in Kapitel 3 der Prototyp und seine Funktionalitäten vorgestellt. Dieser wird verwendet um neugewonnene Datensätze, welche zum Zeitpunkt des Praxissemesters nicht vorlagen, auszuwerten. Daran sollen die Stärken und Schwächen des aktuell genutzten Verfahrens bzw. Software als Ganzes zu erkennen sein. Dies ist wichtig um dem Leser anschließend die für diese Arbeit neu entwickelten Verfahren näher zu bringen. Letztlich wird eine Anforderungsanalyse durchgeführt, um Rahmenbedingen für die Implementierung der Software zu erörtern.

Im anschließenden Kapitel 4 werden die Implementierungsdetails behandelt. Dazu gehört das Refactoring den Prototypen sowie die Implementierung der Algorithmen neuer Verfahren.

In Kapitel 5 werden die Auswertungen der Verfahren analysiert und diskutiert. Daraus soll hervorgehen, welches Verfahren sich am besten dazu eignet, bei der Beratung einen Monolithen zu zerlegen, helfen kann.

2 Grundlagen

Dieses Kapitel bildet das Fundament der vorliegenden Bachelorarbeit. Die für das Verständnis am wichtigsten Begriffe und Themenbereiche werden eingeführt und definiert.

Zu Beginn werden die Begriffe der Kopplung und Kohäsion erläutert und welche Bedeutung sie für Monolithen und Microservices haben. Anschließend werden die im Zentrum dieser Arbeit stehenden Architektur-Stile definiert und deren Vor- und Nachteile werden beleuchtet. Im Zuge dessen wird das Konzept von Domain-Driven Design angeschnitten und im Kontext der Microservice-Architektur betrachtet. Da bereits Muster zum Zerlegen von Monolithen in Microservices existieren, werden einige dieser in ihren Ansätzen vorgestellt. SQL-Abfragen stehen im Zentrum der in dieser Arbeit entwickelten Verfahren zum Ermitteln von Domain-Grenzen, daher wird das Konzept des Tabellenverbunds (Join) erörtert.

2.1 Kohäsion und Kopplung

In der Informatik bezeichnet Kohäsion den Zusammenhang von Elementen innerhalb eines Moduls, während die Kopplung die Abhängigkeit von Modulen beschreibt (predic8 (2016)). Um den Microservices Architektur-Stil erfolgreich umzusetzen, gilt es, die service-interne Kohäsion zu maximieren und die Kopplung zwischen den Services zu minimieren. Bezogen auf den Microservices-Architekturstil bedeutet dies, dass verschiedenen Services eines Systems sollen lose aneinandergekoppelt sein sollen. Durch die lose Kopplung wird garantiert, dass die Microservices eines Systems, unabhängig voneinander ausgeliefert werden können. Zudem führt die Unabhängigkeit der Services dazu, dass Änderungen nicht Service-übergreifend sind (Newman (2015)). Eine hohe service-interne Kohäsion führt dazu, dass Code, der sich gemeinsam verändert, zusammenbleibt (Newman (2015)).

2.2 Domain-Driven-Design

DDD ist ein Konzept zur Entwicklung von Software-Produkten, bei denen der Fokus auf der Modellierung und Implementierung der Geschäftslogik liegt. Dabei soll das Verhalten der Software, sowie die interne Logik, ein Abbild der realen Geschäftsdomäne sein. Vernon (2016)

Eine Domäne bezeichnet den Bereich bzw. das Problemfeld, in welchem die Software eingesetzt wird. Innerhalb einer Domäne existieren Elemente, welche in enger Zusammenarbeit zwischen den Software-Entwicklern und den Domänenexperten —In der Regel Mitarbeiter, welche sich besonders gut mit dem jeweiligen Feld auskennen— ausfindig gemacht werden, um ein möglichst präzises und praxisnahes Abbild der Geschäftslogik zu ermöglichen. Vernon (2016) Eine Domäne wie beispielsweise ein Versandhaus kann sehr komplex modelliert werden, bedenkt man die Teilelemente, die darin enthalten sein können: Bestellungen, Kunden, Artikel, Artikel -Empfehlungen, Versand und vieles mehr.

Daher existiert im DDD das Konzept des abgegrenzten Kontexts (Bounded Context). Dieser repräsentiert einen klar definierten Bereich bzw. ein Teilmodell innerhalb der Gesamtdomäne der Anwendung. Jeder abgegrenzte Kontext enthält ein eigenes Domänenmodell, das spezifisch für diesen Kontext verwendet wird. In einem abgegrenzten Kontext wird eine ubiquitäre Sprache (Ubiquitous Language) verwendet, dessen Zweck es ist, spezifisches Vokabular und spezielle Begriffe zu verwenden, die für die Domäne und die Akteure innerhalb des Kontextes sinnvoll sind. Die ubiquitäre Sprache hilft, Missverständnisse zu vermeiden und die Kommunikation zwischen den Entwicklern und den Domänenexperten zu verbessern.Vernon (2016)

2.3 DDD im Kontext: Monolith und Microservices

Bei der Umsetzung von Microservices ist ein häufig gewählter Ansatz, dass Microservices um eine Geschäftsdomäne modelliert werden bzw. nach den Geschäftsfunktionalitäten ausgerichtet sein sollen (Newman (2015), Google ([o. D.]), Amazon ([o. D.(a)])). Jeder Abgegrenzte Kontext beschäftigt sich mit einem Teilproblem bzw. einem Teil der gesamten Geschäftsdomäne. Die Teildomänen können mit Hilfe der von DDD zur Verfügung gestellten Methoden erarbeitet werden. Dabei repräsentiert jeder abgegrenzte Kontext einen Microserivce (Microsoft (2022)).

DDD kann auch bei Monolithen eingesetzt werden. Wie bei der Ermittlung von Service- bzw. Domain-Grenzen bei der Microservices-Architektur, kann DDD bei Monolithen genutzt werden um voneinander unabhängige Module zu identifizieren (Microsoft (2022)). Ein Modul dient zur Organisation und Separierung von Funktionalitäten eines Programms (Grzybek (2019)). Weiterhin ermöglichen Module, dass Teams unabhängig voneinander am Monolithen arbeiten können (Newman (2019)). In diesem Fall spricht man auch von einem modularen Monolithen.

2.4 Definition: monolithische Software-Architektur

Man spricht von einer monolithischen Software-Architektur, wenn es sich bei der vorliegenden Software um eine Anwendung handelt, die als eine eigene, in sich geschlossene Einheit implementiert wird und die ausschließlich gemeinsam ausgeliefert werden kann (Richardson (2014), Newman (2019)).

Monolithen gehören zu den gängigsten Architektur-Stilen, welche von Unternehmen genutzt werden, um ihre Dienste zur Verfügung zu stellen(Richardson (2014)). Obwohl der Kern dieser Arbeit das Ermitteln von Domain-Grenzen einspannt, ist es keinesfalls zielführend, Microservices als einzige Lösung bzgl. der Umsetzung eines Systems in Betracht zu ziehen. Sam Newman, ein angesehener, unabhängiger Berater, welcher bereits an vielen Migrationen von Monolithen zu Microservices gearbeitet hat, empfiehlt Start-ups mit einem Monolithen zu starten. Der Grund dafür sei zum einen, dass die zu entwickelnde Domain im Laufe der Zeit vielen Veränderungen unterliegt. Um Microservices erfolgreich umzusetzen, werden stabile Service-Grenzen benötigt (Newman (2015)). Es kann daher vorteilhaft sein eine monolithische Software-Architektur zu verwenden.

2.5 Vor- und Nachteile eines Monolithen

Ein wesentlicher Vorteil der monolithischen Software-Architektur besteht darin, dass Monolithen einfacher zu entwickeln und auszuliefern sind (Newman (2015)). Dies liegt beispielsweise an den simplen Technologien, die für Monolithen verwendet werden. In vielen Fällen wird ein Monolith in Java entwickelt und besteht aus einer einzigen WAR-Datei, welche im Verzeichnis des Webservers hinterlegt wird(Richardson (2014)). Da der Monolith über eine Code-Basis verfügt, verläuft die Kommunikation lediglich über Modulgrenzen hinaus, anstatt über das Netzwerk. Reporting-Abfragen sind leicht umzusetzen, da man eine Datenbank hat, welche alle Daten enthält (Newman (2015)). Wächst ein Monolith, so entsteht über Zeit eine Komplexität, welche das Warten und Erweitern bzw. Abändern erschweren. Auch das Einführen von neuen Technologien kann schwierig sein (Amazon ([o. D.(a)])). Das aus der Definition hervorgehende Merkmal, den Monolithen nur als Ganzes ausliefern zu können, kann ebenfalls als Nachteil angesehen werden. Es kann dazu führen, dass verschiedene Teams, die am Monolithen arbeiten auf andere Teams warten müssen, wenn sie ihre Änderungen in Betrieb bringen wollen (Newman (2015)). Dies kann ebenfalls zu längeren Release-Zyklen führen.

2.6 Definition: Microservices

Aktuell gibt es keine einheitliche Definition für den Begriff „Microservices“ (James Lewis (2014)). Es gibt einige Charakteristiken, welche sich in den Versuchen Microservices zu definieren, wiederfinden. Dazu zählt beispielsweise das Single-Responsibility-Pattern (SRP), welches auf Microservices angewandt wird. Das heißt manchen Definitionen zu Folge, sollen Microservices genau eine Funktionalität umsetzen (Amazon ([o. D.(a)]), Reselman (2022a)). Eine weitere Charakteristik, die häufig in den Definitionen zu Microservices auftaucht ist das voneinander unabhängige Ausliefern der Services (James Lewis (2014),Newman (2015)). Microservices sind voneinander unabhängig auslieferbare Services, welche nach einer Geschäftsdomäne modelliert und lose aneinandergekoppelt sind (Newman (2019)). Intern existiert eine hohe Kohäsion und die verschiedenen Services kommunizieren über ein Netzwerk (Google ([o. D.]), Newman (2019)).

Aus Abschnitt 2.4 geht hervor, dass das Umsetzen von Microservices nicht immer sinnvoll ist. Es ist jedoch nicht von der Hand zu weisen, dass Microservices in vielerlei Hinsicht Vorteile hingegen eines Monolithen haben. In Fällen von Unternehmen, welche ihre organisatorischen Grenzen gut gezogen haben, ermöglichen Microservices, dass Entwickler unabhängiger voneinander Arbeiten können (Newman (2015)).

2.7 Abgrenzung: Microservices und SOA

Sowohl Microservices als auch SOA sind Ausprägungen eines verteilten Architektur-Stils, dennoch unterscheiden sie sich voneinander. Der Hauptunterschied zwischen den beiden Ausprägungen besteht im Umgang mit Daten. Bei einer Microservices-Architektur wird das Teilen einer Datenbank vermieden, Das heißt, es wird eine Duplikation von Daten in Kauf genommen, um die Microservices unabhängig voneinander zu gestalten(Newman (2015), Amazon ([o. D.(b)])). Dieser Ansatz fördert die lose Kopplung zwischen den Services. Bei SOA wiederum wird eine Strategie angestrebt, wodurch das Teilen von Daten unterstützt wird. Dies kann zu hoher Kopplung führen, wodurch die Skalierbarkeit oder die Abänderbarkeit eingeschränkt wird(Amazon ([o. D.(b)])).

2.8 Vor- und Nachteile von Microservices

Das falsche Abgrenzen von Services kann zu serviceübergreifenden Änderungen und stark aneinandergekoppelten Komponenten führen (Newman (2019)). Einer der größten Vorteile von Microservices ist die Fähigkeit, einzelne Komponenten eines Systems voneinander unabhängig auszuliefern (Newman (2019)). Dies ermöglicht schnellere und agile Entwicklungsprozesse, welche von kleinen Teams verwaltet werden können (predic8 (2022), predic8 (2019)). Microservices führen einen hohen Grad an Komplexität ein (Reselman (2022b)). Da es sich um voneinander unabhängige Services handelt, die über das Netzwerk kommunizieren, benötigt man Orchestrierungswerkzeuge wie Kubernetes (Reselman (2022b)). Weitere Komplexität wird durch Service übergreifende Transaktionen eingeführt. Will man Microservices betreiben, so muss darauf geachtet werden, dass die Daten, auf welche von den verschiedenen Services zugegriffen werden, konsistent sind. Zudem müssen Netzwerkfehler in Betracht gezogen werden. Dazu wird beispielsweise das Saga-Pattern benötigt, welches für jede Transaktion eine Umkehrtransaktion ausführen kann (Ibryam (2021), Microsoft ([o. D.(a)]).) Obwohl Microservices einen gewissen Grad an Komplexität einführen, gewinnt man an Flexibilität (Reselman (2022b)). Ein Beispiel für die gewonnene Flexibilität ist, dass man bei Microservices nicht an ein und dieselbe Technologie gebunden ist. Zudem lassen sich Microservices individuell skalieren.

2.9 Skalierbarkeit von Monolithen und Microservices

Der Scale Cube ist ein Model zu Darstellung der Skalierbarkeit von Software. Die x-Achse drückt Skalierbarkeit anhand von Duplizierung der Instanz einer Anwendung aus. Dort befindet sich der Monolith, welcher in Kombination mit einem Load-Balancer verwendet wird (Richardson (2014)). Die y-Achse spiegelt die Skalierbarkeit von einzelnen Komponenten eines Systems wider. Die individuelle Skalierbarkeit von Funktionalitäten eines Systems sind auf die Charakteristik, dass Microservices genau eine Funktionalität umsetzen sollen, zurückzuführen. Demnach werden Microservices der y-Achse entlang skaliert. Auf der z-Achse werden Daten partitioniert. Dies wird hauptsächlich verwendet, um Datenbanken zu skalieren Richardson ([o. D.]).

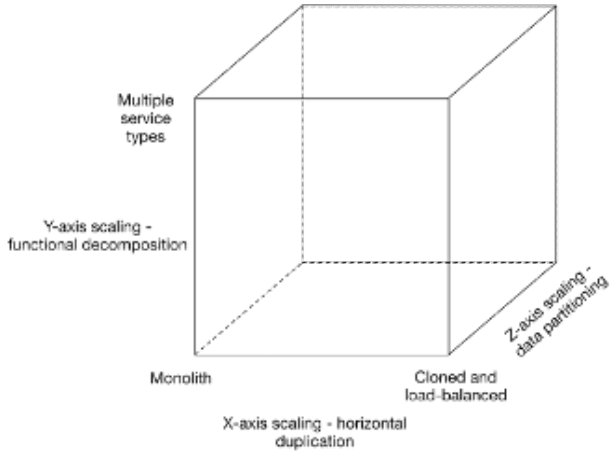

Abbildung 2.1: Darstellung des Scale Cubes, nach Martin L. Abbott (2015)

2.10 Beispiel: Versandhaus als monolithische Anwendung

Ein Beispiel für eine monolithische Software-Architektur kann ein Onlineversandhaus sein. Als Kunde kann man sich dort registrieren bzw. anmelden und den Warenkatalog nach bestimmten Artikeln durchsuchen. Zudem werden auf den einzelnen Produktseiten zu den bestimmten Artikeln, Kaufvorschläge für andere Produkte aufgelistet. Weiterhin kann man als Kunde die dort gelisteten Produkte bestellen. Der Monolith verwaltet also neben der Kunden- und Artikelverwaltung auch den Bestell- sowie den Versandprozess. Alle nötigen Prozesse um eine Bestellung durchzuführen, angefangen bei der Registrierung bzw. der Anmeldung bei wiederkehrenden Kunden, bis hin zum Versand werden von einer Anwendung verwaltet. Die gesamte Applikation ist in Java geschrieben und wird in einer .WAR-Datei gebündelt. Diese ist auf einem Computer hinterlegt, auf welchem ein Webserver wie Tomcat installiert ist (Richardson (2014)). Zu Persistenz zwecken existiert eine Datenbank. In dieser Datenbank werden alle Daten gespeichert, die für die Anwendung von Bedeutung sind. Dazu zählen Kundendaten, Informationen über Artikel, Bestellinformationen und weitere für den Betrieb notwendige Daten. Geht man davon aus, dass das Unternehmen wächst und an Kunden dazugewinnt, muss berücksichtigt werden, dass mehr Rechenleistung benötigt wird um alle Anfragen, die bei dem Webserver eingehen, verarbeitet werden können. In diesem Fall muss das Unternehmen den Monolithen skalieren. Dazu werden weitere Instanzen des Monolithen auf einem oder mehreren Rechnern hochgefahren (Skalierung entlang der x-Achse) (Richardson (2014)). Dabei ist zu beachten, dass das Hochfahren einer weiteren Instanz der Anwendung bedeutet, dass die gesamte Anwendung dupliziert wird, da der Monolith ein einziger Prozess ist (predic8 (2022)). Demnach erhält man weitere Instanzen von Funktionalitäten, die von der

Auslastung nicht betroffen sind. Diese werden dennoch mit skaliert. Der Service, welcher dafür zuständig ist andere Artikel vorzuschlagen, während man auf einer Produktseite ist, profitiert von einer Skalierung. Der Service, der für die Bestellung zuständig ist, benötigt möglicherweise keine Skalierung. Der Grund dafür ist nachvollziehbar, bedenkt man das eigene Kaufverhalten. Man betrachtet und vergleicht viele Artikel, bevor man eine einzige Bestellung aufgibt. Demnach werden viele Produktseiten besucht und viele Vorschläge werden generiert, wohingegen der Bestellvorgang weniger oft durchgeführt wird. Existieren mehrere Instanzen des Monolithen, so wird ein Load Balancer verwendet um die Lasten je nach Bedarf auf die verschiedenen Instanzen zu verteilen (predic8 (2022), Richardson (2014)).

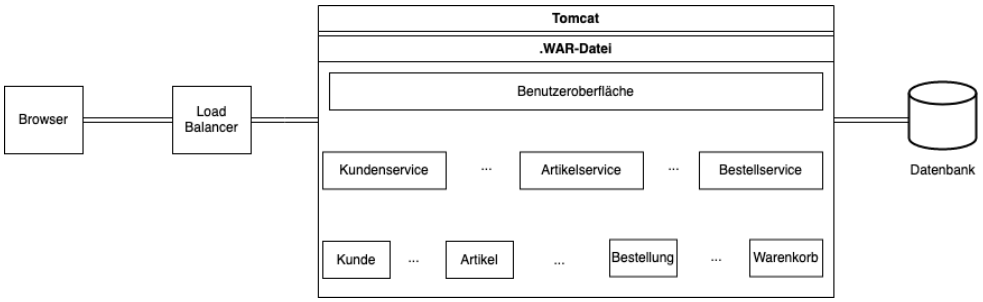

Abbildung 2.2: Beispiel für eine monolithische Software-Architektur (Erstelt mit draw.io)

2.11 Beispiel: Versandhaus als Microservices

Im Kontext dieses Beispiels werden die gleichen Funktionalitäten zur Verfügung gestellt, wie im Beispiel zur monolithischen Anwendung. Allerdings, statt eines Monolithen, kommen Microservices zum Einsatz. Anstatt die gesamte Anwendung in einer WAR-Datei gebündelt auf einem Webserver liegen zu haben, läuft jeder Microservice in einem Docker-Container. Java gehört zu den beliebtesten Programmiersprachen. Unter Programmierern ist Spring Boot eines der beliebtesten Frameworks, um Webanwendungen in Java zu entwickeln (JetBrains (2021) und JetBrains (2022)). Es macht Sinn, dass die Microservices mit diesen Technologien umgesetzt werden. Zudem verfügt jeder Service über seine eigene Datenbank. Damit die Kommunikation der Services untereinander geregelt wird, existiert ein Message-Broker, wie zum Beispiel Apache Kafka. Für das Ausliefern der Services in eine Produktionsumgebung können DevOps-Praktiken genutzt werden. Dazu gehört unter anderem eine CI/CD-Pipeline um den Code automatisiert zu testen, zu bauen und auszuliefern (Microsoft ([o. D.(b)])). Muss der Service, welcher für Produktvorschläge zuständig ist, skaliert werden, ist dies problemlos möglich, ohne andere Komponenten des Systems skalieren zu müssen (Richardson ([o. D.])).

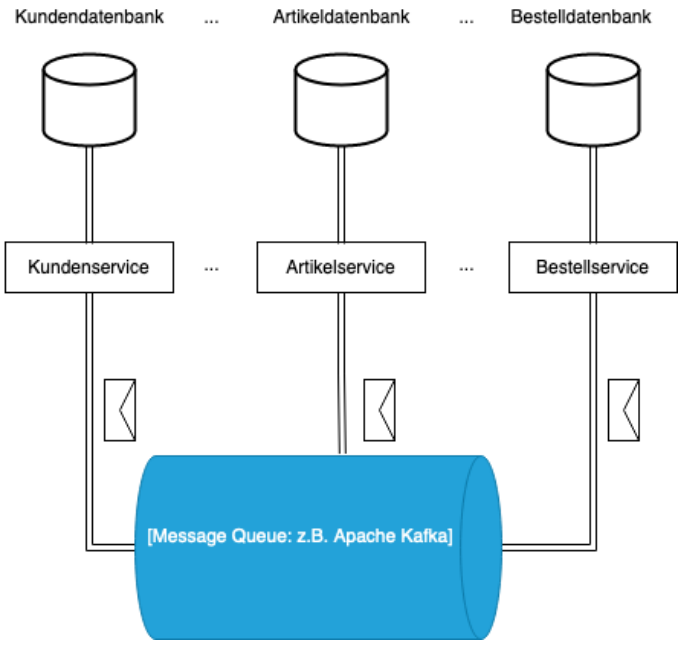

Abbildung 2.3: Beispiel für einer Microservices Software-Architektur (Erstelt mit draw.io)

2.12 Vorgehen zum Zerlegen eines Monolithen

Anhand der letzten beiden Abschnitte wurde gezeigt, wie ein und dieselbe Anwendung aussehen könnte, würde man sie mit unterschiedlichen Architektur-Stilen umsetzen. In diesem Abschnitt sollen Design-Muster vorgestellt werden, welche bei dem Vorgang der Zerlegung eines Monolithen in Microservices Unterstützung bieten. Die erste Art einen Monolithen aufzulösen, ist ein horizontaler Schnitt. Dabei wird jede entstehende Schicht von einem Team betreut. Ist mit einer neuen Anforderung zu rechnen, so müssen Änderungen in allen Schichten vorgenommen werden. Dazu müssen sich die Teams untereinander austauschen (predic8 (2019)). Damit ist das Ziel, voneinander unabhängige Services auszuliefern nicht erfüllt. Ein weiterer Ansatz ist ein vertikaler Schnitt. Dieser zerlegt die Anwendung in fachliche Teildomänen. Dabei wird jede Teildomäne von einem Team betreut (predic8 (2019)).

Ein von Experten häufig empfohlenes Vorgehen basiert auf einer inkrementellen Migration. Dabei werden Bereiche, die bereits zu einem gewissen Grad vom Monolithen entkoppelt sind, ausgelagert, und zu einem Microservice konvertiert. Dies hilft den Entwicklern zum einen, mehr über Microservices zu lernen und zum anderen sorgt die inkrementelle Migration für kleinere Schritte, was die Migration als Ganzes überschaubarer macht (Newman (2019), Dehghani (2018)). In Bezug auf das Versandhaus-Beispiel könnte zuerst die Benutzerauthentifizierung ausgelagert werden (Dehghani (2018)). Im Folgenden sollen Kernfunktionalitäten entkoppelt werden, indem die Abhängigkeit zum Monolithen mit neuen Microservices reduziert wird. Dies führt schließlich dazu, dass der Monolith eine Abhängigkeit zu den Services entwickelt (Dehghani (2018)). Anschließend sollen Teile, mit unklaren Domain-Grenzen dekonstruiert werden. Diese verschwommenen Grenzen verlocken dazu, den gesamten Teil zu einem Service zu konvertieren. Dies führt jedoch zu der gleichen engen Kopplung, allerdings als Microservice verpackt. Es wird empfohlen auf DDD zurückzugreifen, und bessere Grenzen zu identifizieren (Dehghani (2018)). Ein weiterer Ansatz ist das „Branch by Abstraction“-Pattern. Dieses Pattern ermöglicht es den

Code des Monolithen zu verändern, ohne dass Verzweigungen im VCS entstehen. Stattdessen können die Änderungen mit bereits geschriebenem Code koexistieren. Im ersten Schritt wird eine Abstraktion erzeugt, welche die Funktionalität ersetzen soll. Dies könnte beispielsweise ein Interface sein. Im zweiten Schritt soll die Abstraktion genutzt und implementiert werden. Zu diesem Zeitpunkt existieren die alte und neue Implementierung gleichzeitig im Monolithen. Genutzt wird jedoch nur eine. Funktioniert die neue Implementierung ausreichend gut, kann ein Wechsel stattfinden. Im letzten Schritt wird die alte Implementierung entfernt. Selbst mit existierenden Ansätzen und Design-Patterns bleibt die Frage nach dem „Wie genau“ offen, wenn es um das Setzen der Schnitte geht. Diese Frage soll, mit in dieser Arbeit vorgeschlagenen Methoden, beantwortet werden.

2.13 Datenbanken: Abfragen und Persistierung

Da die in dieser Arbeit entworfenen Verfahren darauf beruhen, SQL-Logdateien auszuwerten und Join-Ketten im Zentrum dieser stehen, werden diese definiert und erklärt.

SQL ist eine Datenbanksprache, welche unter anderem verwendet wird um Strukturen in einer Datenbank zu definieren (DDL), und Daten nach gewissen Kriterien zu bearbeiten und abzurufen (DML) (Almir Vuk (2018)). Daten werden in Tabellen gespeichert, welche aus Zeilen und Spalten bestehen. Tabellen können bei einer Abfrage verknüpft werden. Ein Join ist eine Datenbankabfrage, welche zwei oder mehrere Datenbanktabellen verknüpft (mysqltutorial ([o. D.])).

2.13.1 Abfragen und Tabellenverknüpfungen

Das SELECT-Schlüsselwort, gefolgt von einer Auflistung beliebiger, existierender Spaltennamen, dient als Projektion der Daten und blendet alle nicht aufgelisteten Spalten aus. Durch das WHERE-Schlüsselwort, gefolgt von einer Kondition, werden Reihen selektiert, die im Ergebnis vorzufinden sind (Stein ([o. D.])). Eine Join-Abfrage verbindet die zu verknüpfenden Tabellen und wendet darauf die Selektion der Spalten an, ausgedrückt durch die WHERE-Klausel. (mysqltutorial ([o. D.])). Die Schlüsselwörter LEFT, RIGHT, INNER und OUTER beschreiben die Ergebnismenge, die aus der Join-Abfrage resultiert. Ein INNER-Join gibt nur die Datensätze zurück, für die die Verknüpfungsbedingung in beiden Tabellen erfüllt ist. Das bedeutet, dass nur die Datensätze zurückgegeben werden, für die eine Übereinstimmung in beiden Tabellen vorhanden ist (mysqltutorial ([o. D.])). Ein LEFT-(OUTER)-Join gibt alle Datensätze aus der linken Tabelle und die übereinstimmenden Datensätze aus der rechten Tabelle zurück. Wenn es in der rechten Tabelle keine Übereinstimmung gibt, werden NULL-Werte für die rechten Tabellenspalten zurückgegeben (mysqltutorial ([o. D.])). Ein RIGHT-(OUTER)-Join gibt alle Datensätze aus der rechten Tabelle und die übereinstimmenden Datensätze aus der linken Tabelle zurück. Wenn es in der linken Tabelle keine Übereinstimmung gibt, werden NULL-Werte für die linken Tabellenspalten zurückgegeben (mysqltutorial ([o. D.])).

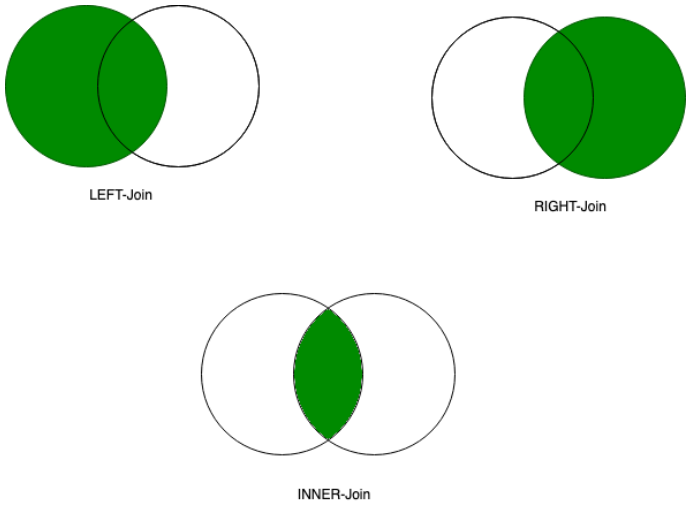

Abbildung 2.4: Visualisierung der Ergebnismenge von Join-Operationen (Erstelt mit draw.io)

2.13.2 Persistierung von Daten

In Abschnitt 2.10 wurde eine monolithische Anwendung anhand eines Beispiels erläutert. Darin wurde das Persistieren von Daten in einer Datenbank erwähnt. Um eine Persistierung von Java Objekten durchführen zu können, wird eine ORM-Schicht benötigt. Eine ORM-Schicht sorgt für die Konvertierung von Java Objekten, sodass diese in einer Datenbank verwaltet werden können. Die ORM-Schicht wird von einem Framework wie beispielsweise Hibernate, implementiert und ist durch die JPA Spezifikation definiert. Aufgrund der verstrickten Historie von Hibernate und JPA werden diese Begriffe oft durcheinander gebracht. Viele Kernideen von Hibernate wurden in der JPA aufgenommen, was möglicherweise ein Grund für die Verwechslung der beiden sein kann (Tyson (2022)). Durch die ORM-Schicht entsteht eine Abbildung der Java-Klassen auf die Tabellen der Datenbank. Dadurch können Java-Entwickler Java-Objekte in einer Datenbank persistieren und bearbeiten, ohne die gewohnte objektorientierte Umgebung zu verlassen. Die Abbildungen von Klassen zu Datenbankobjekten wird durch Annotationen umgesetzt. Die `@Entity`-Annotation sorgt dafür, dass Objekte welche der damit versehenen Klasse entstammen, in der Datenbank persistiert werden können. Dazu wird in der Datenbank eine Tabelle mit dem Namen der Klasse generiert. Die Felder einer mit `@Entity` annotierten Klasse werden als Attribute der Tabelle der Datenbank genutzt (Tyson (2022)).

```
@Entity
public class Customer {
    // ...
    private String name;
    // ...
}
```

Abbildung 2.5: Beispiel einer mit `@Entity`-annotierten Klasse

In Beispiel 2.5 führt die `@Entity`-Annotation zu Erzeugung einer Customer-Tabelle. Das Feld `name` ist eine Spalte in der Customer-Tabelle.

2.14 Fazit Monolithen und Microservices

In diesem Kapitel wurden die wichtigsten Grundlagen für das Verstehen der folgenden Kapitel behandelt. Für Microservices wird eine hohe interne Kohäsion sowie eine niedrige Kopplung zu anderen Services vorgeschlagen. Dies führt unter anderem dazu, dass die Services unabhängig voneinander ausgeliefert werden können. Aufgrund von hoher internen Kopplung können Monolithen schwer abänderbar sein. DDD ist ein wichtiges Werkzeug für das Ermitteln von Domain-Grenzen. Es kann sowohl dafür genutzt werden einen Monolithen zu modularisieren als auch für das Definieren von Grenzen zwischen Microservices. Es existieren Design-Muster, welche sich mit dem Zerlegen von Monolithen beschäftigen. Diese basieren auf dem Ermitteln von lose gekoppelten Elementen eines Monolithen. Frameworks wie Hibernate können dazu genutzt werden, eine Brücke zwischen der Welt der Objekt-orientierten Programmierung und relationalen Datenbanken zu bauen.

Sowohl Monolithen als auch Microservices haben Vor- und Nachteile. Welcher Architektur-Still letztlich umgesetzt wird, sollte von den individuellen Anforderungen an die Anwendung abhängig sein. Während der Anfertigung dieser Ausarbeitung, ersetzte Amazon für das Streaming-Portal „Amazon Prime Video“, die vorhandene Microservices-Architektur durch einen Monolithen. So war Amazon in der Lage, Kosten für den Betrieb der Anwendung um 90% zu reduzieren (Kolny (2023)). Der Fall Amazon macht klar, dass Microservices nicht immer die beste Möglichkeit bieten um eine Anwendung zu realisieren. Falls Microservices sinnvoll umgesetzt werden sollen, ist das Ziehen von Grenzen zwischen den einzelnen Microservices eine der Hauptaufgaben (Microsoft (2022)).

3 Analyse und Konzeption

In diesem Kapitel geht es darum, den im Praxissemester angefertigten Protoyp vorzustellen und die darin implementierten Verfahren auf neuen Daten anzuwenden. Daraus können bereits erste Erkenntnisse für potenzielle Verbesserungen gezogen werden. Weiterhin werden verschiedene Verfahren zum Finden von Domain-Grenzen vorgestellt, dessen Implementierungsdetails in Kapitel 4 besprochen werden.

3.1 Konzeption des Prototyps (ist-Zustand)

Der während des Praxissemesters entwickelte Prototyp ist eine Zusammensetzung aus einem Backend, in welchem die Daten aufbereitet und analysiert werden, und einem Frontend in welchem die Daten als Graph dargestellt werden.

Per Mausklick auf die Dateileiste öffnet sich ein Dialog, welcher das Laden einer Datei ermöglicht. Daraufhin erscheinen die Knoten und Kanten, welche in der Datei hinterlegt sind. Unterhalb der Dateileiste existieren Werkzeuge, um den Umgang mit den Knoten und Kanten bzw. dem Graphen zu vereinfachen. Bei besonders großen Graphen, welche viele Knoten und Kanten enthalten kann das Deaktivieren der Knotennamen oder Kantengewichten eine bessere Übersicht verschaffen. Ein weiteres Werkzeug ermöglicht das Fixieren von Knoten. Die Regler existieren, um den Graphen als Ganzes zu manipulieren. Sie ermöglichen das Verschieben des Graphs entlang der y-Achse und das Zusammenziehen bzw. Ausbreiten der Knoten.

Abbildung 3.1: Menüleiste des Frontends

Nachdem eine Datei über den Dialog ausgewählt wurde, erscheint der Graph, mit den in der Datei definierten Kanten und Knoten.

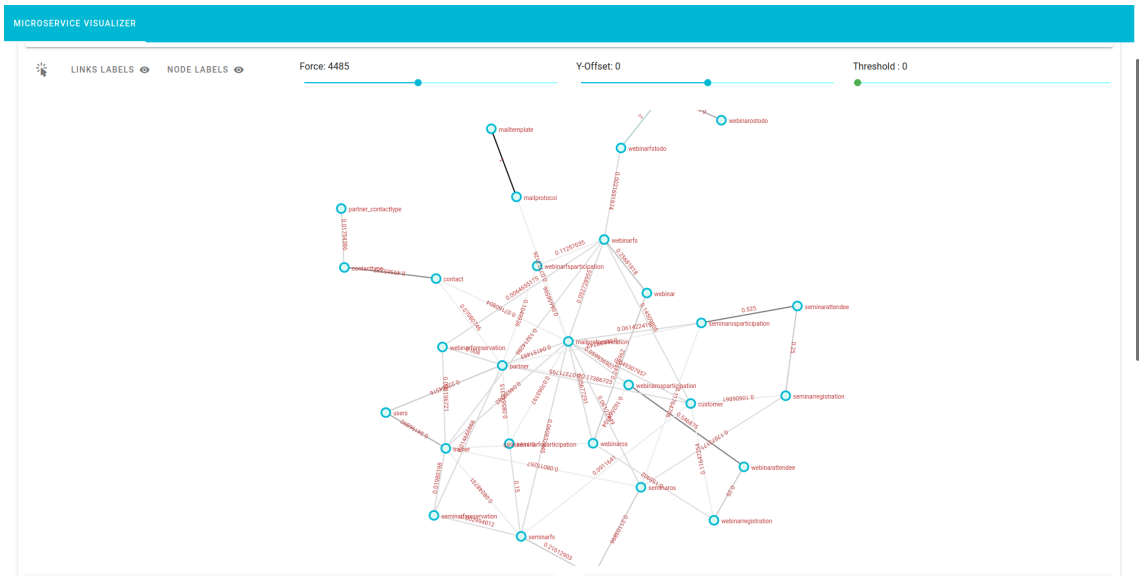

Abbildung 3.2: Darstellung der Daten als Graph im Frontend

Um die Kanten im Graph verschwinden zu lassen und potenzielle Domain-Grenzen zu finden, kann der Threshold-Regler verschoben werden. Nach jeder Verschiebung werden alle Kanten ausgeblendet, deren Kantengewicht geringer ist als der Threshold. Zudem werden alle Kanten, dessen Gewicht höher ist als der Threshold, eingeblendet.

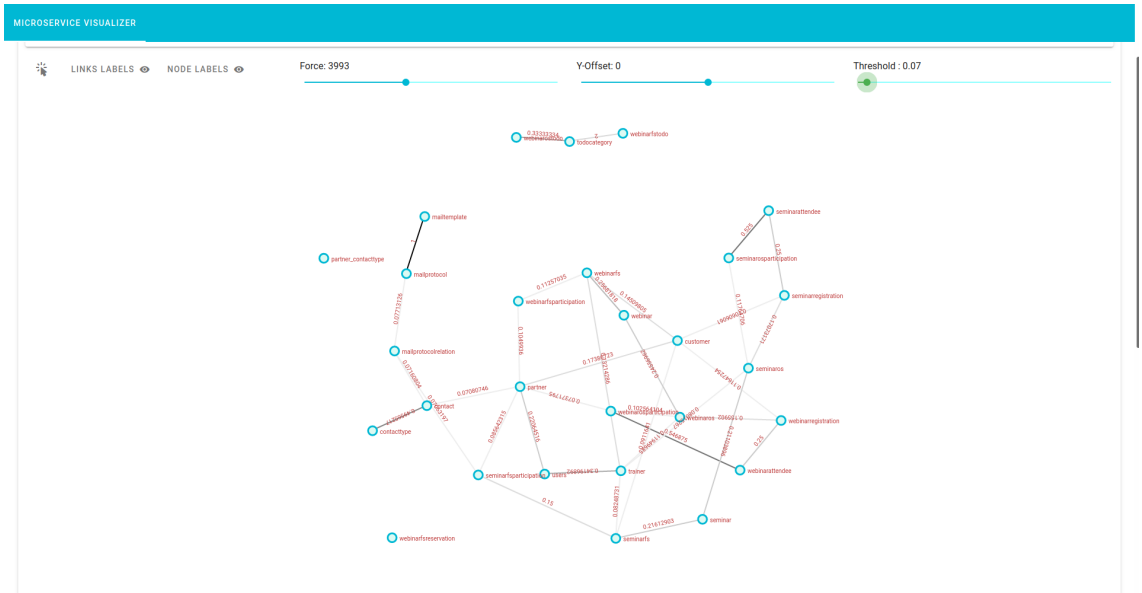

Abbildung 3.3: Darstellung des Graphs nach Anheben des Thresholds

Eine Datei, welche in das Frontend geladen werden kann muss erst erzeugt werden. Dazu wird das Backend der Applikation verwendet. Um die Auswertung zu vereinfachen kann ein Bash-Skript verwendet werden, welches sowohl die Datenaufbereitung als auch

die Datenauswertung vornimmt. Im Hintergrund führt das Skript Aufrufe an die jeweiligen Klassen des Backends durch (`java -cp`).

```
kaiklemmer: /Users/kaiklemmer/programming/monolith-killer $ ./mst.sh -h
This Toolchain is used to prepare and analyze log data for the Microservices Visualizer.
As for the log file, it should contain SQL-statements, seperated by a semicolon(;).
Be aware that this scrpit might not work as intended on other unix-based systems.
This script was written using Ubuntu 21.04. (Hirsute Hippo)
It also is confirmed to be compatible with M1 Mac, MacOS 12 (Monterey)

Syntax: ./mst.sh [-d|h|i]
options:
d     Target directory.
h     Print this Help.
i     Input log file.

kaiklemmer: /Users/kaiklemmer/programming/monolith-killer $
```

Abbildung 3.4: Hilfe-Menü des Bash-Skipts

3.1.1 Defintion: Graph, Knoten und Kanten

Ein Graph (G) welcher im Frontend visualisiert wird besteht aus einer Menge Knoten (N) und Kanten (E).

Ein Knoten $n \in N$ verfügt über einen eindeutigen Namen. Eine Kante $e \in E$ verbindet zwei Knoten $n_1, n_2 \in N$. Zwei Knoten $n_1, n_2 \in N$ sind verbunden oder benachbart, gdw. $\exists e \in E$ mit $e(n_1, n_2)$

Zudem verfügt eine Kante e über ein Kantengewicht w, welches anhand der Metrik bestimmt wird.

3.1.2 vom SQL-Log bis zum Graph

Um einen Graph im Frontend visualisieren zu können wird zunächst eine SQL-Logdatei benötigt, welche Join-Abfragen enthält. Join-Abfragen enthalten die Hauptinformationen, welche für das Verfahren benötigt werden, um einen Schnitt vorzuschlagen. Sie enthalten Informationen darüber, welche Tabellen der Datenbank der Anwendung, während der Nutzung miteinander verbunden werden.

```
select * from A a_0 left outer join B b_1
    on a_0.a_id=b_1.id left outer join C c_2
        on b_1.b_id=c_2.id left outer join D d_3
            on b_1.b_id=d_3.id left outer join E e_4
                on a_0.a_id=e_4.id left outer join F f_5
                    on e_4.e_id=f_5.id where ...
```

Abbildung 3.5: Beispiel für eine Join-Abfrage in SQL

3.1.3 Zwischenformat: Datenaufbereitung

Das Datenaufbereitungswerkzeug– der DataPrepper (DP)– bekommt eine SQL-Logdatei als Eingabeparameter. Die SQL-Befehle innerhalb der Datei sind mit einem beliebigen

Delimiter zu trennen. Der DP geht standardgemäß davon aus, dass die Befehle mit einem Semikolon(;) getrennt werden. Andernfalls kann der Delimiter vom Nutzer spezifiziert werden.

Im ersten Schritt werden alle SQL-Befehle, die das Schlüsselwort „JOIN“ unabhängig von Groß- und Kleinschreibung enthalten, rausgefiltert. Mit Hilfe von regulären Ausdrücken werden die in dem Join enthaltenen Tabellen welche paarweise verknüpft sind extrahiert. Existiert ein Join j über den Tabellen A und B, so ermittelt der DP das Tabellenpaar $p \in P$. p ist ein Tupel, welches aus exakt zwei Tabellen (t_1, t_2) besteht. Das Ergebnis ist die Menge aller Tabellenpaare P, welche in eine Datei geschrieben werden. Bezogen auf das Beispiel 3.5 werden die Paare:
$[\{A, B\}, \{B, C\}, \{B, E\}, \{A, E\}, \{E, F\}]$ extrahiert.

3.1.4 Auswertung: Datenanalyse

Die im ersten Schritt ermittelten Tabellenpaare, welche der DP in eine Datei schreibt, werden als Eingabeparameter für die Datenanalyse des DataAnalyzers (DA) verwendet. Bei der Datenanalyse werden neben den nötigen Berechnungen, auch N und E, die Knoten und Kanten des Graphs identifiziert. Die Berechnungen basieren auf einem Verfahren, dass die relativen Häufigkeiten der aufkommenden Tabellenpaare ermittelt. Dieses Verfahren unterlief während des Praxisprojekts eine Reihe von Veränderungen, begonnen bei der bloßen Zählung der vorkommenden Tabellenpaare, bis hin zu einer selbst erarbeiteten Metrik.

3.1.4.1 Verfahren der relativen Häufigkeit

Das einfachste Verfahren zur Berechnung der Kantengewichte der Kanten E, basiert auf den relativen Häufigkeiten der Tabellenpaare, die vom DP ermittelt wurden. Existiert ein Paar $p \in P$, bestehend aus den Tabellen t_1, t_2, so existiert im Graph eine Kante e welche zwei Knoten n_1, n_2 verknüpft. n_1 und n_2 repräsentieren die Tabellen t_1 und t_2 des Paares p. Damit entspricht das Gewicht der Kante

$$e = \frac{\|P_{direkt}\|}{\|P\|} \tag{3.1}$$

mit

$$P_{direkt} \coloneqq \{\forall p_i, p_j \in P | (t_{1i} = t_{1j} \wedge t_{2i} = t_{2j}) \vee t_{1i} = t_{2j} \wedge t_{2i} = t_{1j}\} \tag{3.2}$$

Es handelt sich um eine Zählung aller identischen Tabellenpaare, die über allen Joins auftreten, in Relation zu allen anderen Tabellenpaaren. Vorteilhaft ist, dass dieser Wert genormt ist und in einem Intervall $0 \leq w \leq 1$ liegt.

Bei der Zählung der Tabellenpaare ist die Reihenfolge der darin auftretenden Tabellen zu vernachlässigen. Seien J und J' zwei Joins über den Tabellen A und B. J verknüpft A mit B und J' verknüpft B mit A. So entstehen bei der Aufbereitung der Daten durch den DP, zwei Paare p_1, p_2 mit $\{A, B\}$ und $\{B, A\}$ Bei der Zählung gelten diese beiden Tabellenpaare als gleich, da die Information, aus welchem Join die Paare kommen, verloren geht. So spielt ausschließlich die Nutzung der Tabellen in der zugrundeliegenden Anwendung eine Rolle.

Würde die Richtung, in welche die Tabellenpaare verknüpft, betrachtet werden, würde dies zu weiteren Kanten im Graph führen. Für die Auswertung ist nicht wichtig, ob $e(n_1, n_2)$

eine besser gewichtete Kante ist als $e(n_2, n_1)$. Wichtig ist, wie stark die Verbindung der kante e zwischen n_1 und n_2 ist, da dies die Stärke des Verbunds der Tabellen t_1 und t_2 im Vergleich zu den restlichen Tabellen darstellt.

Während des Praxissemesters wurde die Beobachtung gemacht, dass die Kantengewichte schnell gegen 0 konvergierten. Eine der ersten Erweiterungen entstand durch das Hinzuzählen von $indirekten$ vorkommen der Tabellen in anderen Paaren. Sei P eine Menge von Paaren.

$$P_{indirekt} \coloneqq \{\forall p_i, p_j \in P | t_{i1} \in p_j \vee t_{i2} \in p_j\} \tag{3.3}$$

Dann kann das Gewicht einer Kante als

$$e = \frac{\|P_{direkt}\| + \|P_{indirekt}\|}{\|P\|} \tag{3.4}$$

definiert werden. So konvergieren die Kantengewichte langsamer gegen 0. Die Betrachtung der direkten und indirekten Vorkommen von Paaren regte neue Ideen an. Beispielsweise das in Relation von P_{direkt} und $P_{indirekt}$. Diese Überlegungen führten zu folgender Metrik für die Berechnung eines Kantengewichts e:

$$e = \begin{cases} 1 - \sqrt{1 - \frac{1}{\|P_{direkt}\| * 2}} & \text{if } \|P_{indirekt}\| = 0 \\ \left(1 - \sqrt{1 - \frac{1}{\|P_{direkt}\| * 2} - \frac{1}{\|P_{indirekt}\|}}\right)^2 & \text{if } \|P_{indirekt}\| > 0 \end{cases}$$

Die zugrundeliegende Idee ist, dass wenn der Nenner „groß“ ist, Brüche gegen 0 konvergieren. Das Abziehen der Brüche von 1 sorgt dafür, dass die Differenz in diesem Fall nah an 1 ist, was für einen starken Zusammenhalt der Knoten steht, welche über diese Kante verbunden sind. Dies führt zu einem positiven Effekt, sollten die Tabellen, welche von den Knoten repräsentiert werden, oft gemeinsam vorkommen. Weiterhin verstärkt wird dieser Effekt durch die indirekten Paare. Dies spricht für die generelle Wichtigkeit einer Tabelle, da ein genügend großes $P_{indirekt}$ darauf schließen lässt, dass die betroffenen Tabellen auch anderweitig viel genutzt werden. Das Ergebnis wird quadriert für den Fall, dass ein Paar öfters indirekt als direkt vorkommt. Dies würde andernfalls zu einem negativen Kantengewicht führen. Das Ziehen der Wurzel erfolgt, um den Effekt der Quadrierung zu relativieren.

Ein Problem mit diesem Verfahren ist die Abhängigkeit zu dem zugrundeliegenden Datensatz. Je ausgereifter der Datensatz ist, desto besser spiegelt sich der Gebrauch der Anwendung in einer Produktionsumgebung wider. Im Umkehrschluss bedeutet dies, dass eine Logdatei, welche nicht den wahren Anwendungsfall reflektiert zu falschen Schlussfolgerungen führen kann. Demnach ist der vorgeschlagene Schnitt der Anwendung von den gesammelten Logdateien abhängig, sowie von dem Zeitraum, über welchen die Daten gesammelt wurden.

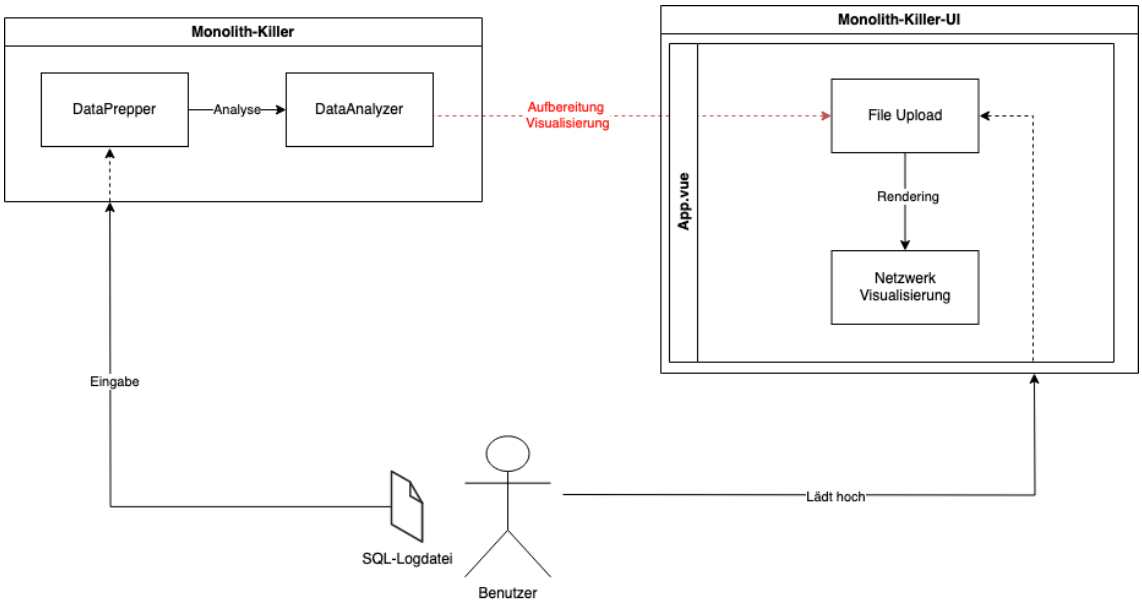

Abbildung 3.6: Abbildung der Architektur des Prototyps (Erstellt mit draw.io)

3.2 Interner Datensatz

Der erste Datensatz wurde aus einer firmeninternen Anwendung gewonnen. Bei dieser handelt es sich um das Kursverwaltungssystem, welches intern genutzt wird um Trainer, Kunden, Partnerfirmen und Kurse zu organisieren und den Arbeitsablauf zu koordinieren. Die Verwaltung nutzt das System unter anderem, um die Abbuchungen für die durchgeführten Kurse einzusehen. Zudem können über die Anwendung E-Mails verschickt werden, um Trainer an anstehende Kurse zu erinnern. Die Vorbereitung für die Durchführung eines Kurses enthält standardisierte Schritte. Die Verwaltung hat daher die Möglichkeit eine To-Do-Liste, welche für jeden Kurs existiert, abzuarbeiten. Weiterhin werden Kunden (Firmen) und deren Ansprechpartner in der Software verwaltet. Der E-Mailverkehr mit den Firmen kann eingesehen werden. Kursanmeldungen und Teilnehmer werden ebenfalls von der Software verwaltet. Nach erfolgreicher Teilnahme an einem Kurs (online oder offline), werden Teilnehmer zu einem Kundenportal eingeladen, wo sie ihre Teilnahme anhand eines Zertifikats nachweisen können. Dieses Zertifikat kann heruntergeladen werden. Die Kursteilnahme sowie der Trainer können bewertet werden.

Um den Datensatz anzureichern wurden weitere SQL-Logdateien, aus anderen Firmenanwendungen gesammelt. Die Anwendung, welche von Interessenten genutzt wird, um sich für Kurse anzumelden, greift auf dieselbe Datenbank zu wie das Verwaltungswerkzeug. Durch das Verknüpfen der Datensätze können die Verbindungen zwischen den Kurstabellen und den Tabellen, welche für die Registrierung zuständig sind, hervorgehoben werden. Weiterhin entsteht durch das Zusammenhängen der Logdateien eine Art „künstlicher“ Monolith. Das Hinzufügen von Logdateien aus dem Kundenportal macht keinen Sinn, da für dieses eine eigene Datenbank existiert. Dies würde unumgänglich dazu führen, dass in der Visualisierung zwei disjunkte Anwendungen angezeigt werden. Damit wird bezweckt, dass die Funktionalität der Software zum Finden von Domain-Grenzen in einem kontrollierten Umfeld stattfindet, da bereits Wissen über die einzelnen Services existiert. Daher kann entschieden werden ob das Werkzeug Domain-Grenzen findet.

Der Datensatz entstand über einem Zeitraum von 7 Tagen und reflektiert den wahren Gebrauch der Anwendung. Mit einer Größe von 117MB handelt es sich um den reichhaltigsten Datensatz, welcher bisher ausgewertet wurde. Der Datensatz setzt sich zusammen aus 55924 SQL-Abfragen. Aus diesen gehen 32 individuelle Join-Ketten vor. Die zuvor, während des Praxissemesters verwendeten Testdaten wurden lokal generiert, wodurch sich der Nutzen der Anwendung nicht sachgemäß widerspiegelte. Daher weist der neue Datensatz Eigenarten auf, mit denen vorher nicht gerechnet wurde. Zwischen der Zeit des Praxissemesters und der Auswertung der neugewonnen Daten ist viel Zeit vergangen. Die Firmenanwendung hat sich durch den Austausch des im Backend genutzten Frameworks verändert. Die neugewonnenen Daten waren anders formatiert als der zuvor verwendete Datensatz, was bei der Datenaufbereitung im DP zu Fehlern führte. Daher mussten die Methoden zur Aufbereitung der Daten an die neuen Daten angepasst werden.

Einige Teile des Datensatzes waren unbrauchbar aufgrund von unvollständigem Abfragen. Beispielsweise enthielten einige Abfragen Tabellen, dessen Namen unvollständig dargestellt wurden. Weiterhin enthielten einige Abfragen, fehlerhafte Tabellennamen, welche Buchstaben durch einen Unterstrich(_) ersetzten. Für diese und alle künftigen Auswertungen des Datensatzes wurden die Fehler manuell korrigiert. Dies war möglich, weil das Datenmodell der Firmenanwendung etwa 30 Tabellen umspannt und die Namen eindeutig sind. Ein Beispiel für solch eine Korrektur ist die Tabelle `partner`. In einigen Abfragen fand man sie als `p_rtner` wieder. Es stellt sich die Frage, wie mit größeren Datenmodellen, dessen Tabellen nicht leicht einprägsam sind, umzugehen ist. Tabellen mit einem _ pauschal zu entfernen macht keinen Sinn, da —abhängig von der genutzten Technologie– das Namensschema automatisch generiert wird.

Eine Eigenart des neuen Datensatzes ist, dass einige Tabellen der Datenbank nicht in der Visualisierungskomponente dargestellt werden. Dabei handelt es sich keineswegs um einen Fehler. Eine Überprüfung der Testdaten macht deutlich, dass einige Tabellen nicht vom Logging erfasst wurden. Dies ist kein technischer Fehler, sondern ein in Abschnitt 3.1.4.1 beschriebenes Problem. Wenn das Logging Teile der Anwendung während der Zeit in der Daten gesammelt werden nicht erfasst, können diese nicht in die Entscheidung für einen vorgeschlagenen Schnitt einfließen. Erklären könnte man dies durch die Sommerpause, in denen weniger Kurse stattfinden. Demnach wird die Anwendung weniger verwendet, da Mitarbeiter der Verwaltung beurlaubt sind. Daher stellt sich die Frage, wie mit solchen Situationen umzugehen ist. Einerseits kann argumentiert werden, dass Bereiche der Anwendung, welche selten genutzt werden, wenig Bezug zum System als Ganzes aufweisen und daher nicht elementar für den Betrieb sind. Erklären lassen würde sich dies durch das Implementieren von Features, welche in der Planungsphase für sinnvoll empfunden wurden und im Betrieb der Anwendung nicht so häufig genutzt werden wie anfänglich vermutet. Andererseits könnten Funktionalitäten und damit im entfernteren Sinne Tabellen der Datenbank, die keinen Bezug zum Rest des Systems haben, aber dennoch benötigt werden in einen eigenen Microservice konvertiert werden und einen eigenen abgegrenzten Kontext bilden. Wiederum muss bedacht werden, dass dadurch Schnittstellen entstehen. In solch einer Situation macht es Sinn einen Experten hinzuzuziehen. Dieser weiß über existierende Tabellen in der Datenbank Bescheid und kann auf fehlende Tabellen in der Visualisierung hinweisen.

Weiterhin ging aus den Daten hervor, dass Joins einer gewissen Länge das Ergebnis der Visualisierung beeinflussen. Ein Beispiel sind Reporting-Abfragen. Diese tauchen in Fällen auf, in denen Daten aus verschiedenen Bereichen der Anwendung aggregiert werden, um eine Übersicht zu erstellen. Ein Beispiel für eine Reporting-Abfrage ist die Ermittlung aller Kurse welche künftig stattfinden. Bei Abfragen dieser Art, sind nicht nur Informationen

über den Kurs von Interesse. Es werden auch Informationen über die Teilnehmer benötigt. Handelt es sich um einen Firmenkurs oder sind Privatpersonen beteiligt. Weiterhin wichtig ist, ob es sich bei einem Kurs um ein Seminar handelt welches Vorort durchgeführt wird, oder ob es ein Webinar online durchgeführt wird. Im Falle eines Seminars spielt der Status des Raums eine Rolle, in dem der Kurs stattfindet.

Werden diese Reporting-Abfragen häufig durchgeführt so erweckt es den Anschein, dass die darin verknüpften Tabellen und damit die Domänen, die sie repräsentieren können, stark aneinandergekoppelt sind. Dies kann übersehen werden wenn man sich nicht mit der zu zerlegenden Anwendung auskennt. Um dem vorzubeugen wurde eine Filterfunktion eingeführt. Diese Ermittelt die durchschnittliche Länge der Joins und filtert alle Joins raus die länger sind. Wie lang diese Abfragen werden können, hängt von der Domäne und dem Umfang der Anwendung ab. Im Fall der Testdaten erfasste das Werkzeug 29 individuelle Tabellen sowie einige Join-Ketten mit über 30 Verknüpfungen. Dabei wurden Tabellen mehrfach untereinander verknüpft. Bei den Reporting-Abfragen handelte es sich um Abfragen, welche durch Einblicke in das Mailprotokoll verursacht wurden. Im Mailprotokoll kommen viele verschiedene Teilbereiche der Anwendung zusammen, was die langen Abfragen erklärt.

Anhang 8.1 bis 8.5 zeigen den Zerfall der Anwendung. Initial sind die Tabellen, welche für Events innerhalb eines Kurses genutzt werden, vom Rest der Anwendung abgetrennt. An dieser Stelle ist es fraglich, ob es sich um einen ersten Hinweis auf einen Microservice handelt. Events werden in Veranstaltungen genutzt, um der Verwaltung einen Überblick über die Historie eines Kurses zu verschaffen. Events beinhalten unter anderem Informationen darüber, ob ein Teilnehmer bereits über die technischen Voraussetzungen informiert wurde. Die Tabellen `event` und `event_type` lassen vermuten, dass es verschiedene Ausprägungen von Events gibt. Eine mögliche Erklärung für diesen Sachverhalt beruht darauf, dass die Firmenanwendung so programmiert ist, dass die Events, welche zu einer Veranstaltung gehören erst dann aus der Datenbank geladen werden, wenn der Benutzer mit ihnen interagiert. Im Vorfeld werden die Events und Ihre Arten bereits aus der Datenbank geladen, sodass sie im Frontend, ohne Verzögerung angezeigt werden können. Alternativ kann es sich um einen schlichten Fehler der Programmierung handeln, bei dem vergessen wurde, beim Laden der Veranstaltungsseite, die Kursspezifischen Events aus der Datenbak zu laden. In den nächsten Schritten wird deutlich, dass sich die Verwaltung der To-Do-Listen für die Kurse lösen. Dies kann dadurch erklärt werden, dass die To-Do-Listen für Kurse einen standardisierten Ablauf darstellen. Dieser ist statischer Natur und demnach werden selten neue To-Do-Einträge erzeugt. Eventuell wird das Feature zum Abhaken der To-Do-Einträge in den Kursen von den Trainern oder der Verwaltung vernachlässigt oder vergessen. Dies würde erklären, weshalb sich die To-Do-spezifischen-Tabellen von den Veranstaltungen abspalten. Nach genügend vielen Schritten bleibt nur der „Kern“ der Anwendung im Graph verbunden. Drumherum sind die losgelösten Tabellen, die keinerlei Verbindung zu anderen Tabellen aufzeigen. Die Event-bezogenen Tabellen lösen sich mit inkrementierten Threshold auf. Dies heißt keineswegs, dass die Tabellen nicht zusammengehören, sondern, dass sie weniger oft in den Joins vorkamen als andere Tabellen und demnach relativ weniger vertreten sind. Bei einem richtigen Monolithen würde man erwarten, dass sich Cluster bilden. Das bedeutet, es gibt viele, vom Kern gelöste kleine Gruppen (lose Kopplung), die intern hochwertige Kantengewichte haben (hohe, service-interne Kohäsion), sodass diese bei dem inkrementellen Vorgehen nicht entfernt werden.

3.3 Open Source CRM Datensatz

Der Zweite Datensatz basiert auf einem Open Source Customer Relationship Management (CRM) Werkzeug von „vtiger“. Bei dem Werkzeug handelt es sich um eine Software, welche von Unternehmen für die Verwaltung und den Verkauf von Produkten genutzt werden kann. Zudem unterstützt es das Verwalten von Kontakten und Lieferanten, das Erstellen von Reports, Versenden von E-Mails und viele weitere Funktionalitäten, die sich Rund um das Verkaufen und Vermarkten von Produkten orientieren. Extrahiert wurden die Daten mit der Logging-Funktionalität, die von MySQL zur Verfügung gestellt wird.

Der aufbereitete Datensatz ist 2.3MB groß und enthält 1142 Joins. Davon sind 134 individuelle Join-Ketten. Zudem besteht die Datenbank aus weitaus mehr als 100 verschiedenen Tabellen. Die vergleichsweise kleine Größe des Datensatzes lässt sich dadurch erklären, dass die Daten lokal generiert wurden. Das CRM wurde $nicht$ in Produktion betrieben. Die Daten spiegeln demnach nicht den wahren Gebrauch der Anwendung wider. Dennoch ist anzumerken, dass die Beziehungen der Tabellen trotz des geringen Datenbestands dargestellt werden können.

Wie bei der Firmenanwendung musste die Methode zur Aufbereitung der Daten im DP angepasst werden, da die generierten SQL-Abfragen nicht dem Format entsprachen, welches vorher verwendet wurde. Die Logdatei musste außerdem per Hand angepasst werden, da einige Abfragen Unregelmäßigkeiten aufwiesen, welche das Resultat beeinflussen könnten. Einige Abfragen enthielten ein Alias für die Tabellen einer Join-Kette. Andere Abfragen wiederum enthielten Klammerpaare, wodurch der DP diese Tabellen nicht ordnungsgemäß erfassen konnte.

Beim Einladen der aufbereiteten Daten ins Frontend wurde anhand des Graphen die monolithische Struktur des CRMs ersichtlich (8.6). Anders als bei dem Graph der firmeninternen Anwendung, wo die Kerntabellen deutlich zu erkennen und sehr vermascht waren, bilden sich in diesem Fall eine Insel von Tabellen, mit einer Tabelle in der Mitte, die die anderen zusammenhält. Anhang 8.7 und 8.8 stellen den Zerfall der Anwendung dar. Anhang 8.7 zeigt die Anwendung, kurz bevor die markierte Kante verschwindet, was ein potenzieller Hinweis auf einen Microservice liefert. Bei einem Graphen dieser Größe fällt das Erkennen von verschwindenden Kanten schwer. Statt die Kanten verschwinden zu lassen, bestünde ein alternativer Ansatz darin, die Kanten in einer anderen Farbe zu kennzeichnen. Durch die Unvertrautheit mit der Anwendung sowie dem darunter liegenden Datenmodell, und der Tatsache, dass die Testdaten lokal generiert wurden, indem eine Person das CRM genutzt hat, was nicht dem eigentlichen Gebrauch der Anwendung darstellt, lässt sich die Qualität des vorgeschlagenen Schnitts in Bezug auf die gebildeten Domänen schwer beurteilen. hauptsächlich konnte aus den neugewonnenen Daten, Wissen über die Aufbereitung der SQL-Logdateien gewonnen werden. Weiterhin wurde ersichtlich, dass ein geringer Datenbestand, nicht zwangsläufig dazu führt, dass der Prototyp keinen Schnitt finden kann.

3.4 Einführung der Neuen Verfahren

In Abschnitt 3.1.4.1 wurde die bisher genutzte Metrik und wie sie zustande kam erörtert. In den folgenden Abschnitten werden die für diese Ausarbeitung neu entworfenen Verfahren beschrieben. Das erste Verfahren bewegt sich weiterhin im Rahmen der Paare. Es nutzt die direkten Nachbarknoten, um für mehr Kontext bei der Bestimmung der Kantengewichte zu sorgen. Anschließend wird das Konzept des Join-Kontexts eingeführt. Dieser soll dabei helfen, die Tabellen nicht nur im Kontext des eigenen Verbunds, sondern im Rahmen der gesamten Join-Kette zu betrachten. Der Join-Kontext hilft unter anderem bei der Ermittlung von Join-Ketten, die durch Reporting-Abfragen entstanden sind.

3.4.1 Nachbarschaftsverfahren

Um mehr Kontext in die Berechnung der Kantengewichte einfließen zu lassen, lohnt es sich benachbarte Kantengewichte zu betrachten. Die Kanten e und $e' \in E$ sind benachbarte Kanten $\longleftrightarrow e(n_1, n_2)$ und $e'(n'_1, n'_2)$ mit $n'_i = n_i, i \in \{1, 2\}$ existiert. Die Menge $E'_e \subseteq E$ beschreibt alle benachbarten Kanten einer Kante e. Die Formel um diesen Sachverhalt auszudrücken ist:

$$e = \frac{\|P_{direkt}\| + \|P_{indirekt}\|}{\|E'_e\|} \tag{3.5}$$

Dadurch entsteht eine Gewichtung der Kante e im Kontext der beiden Knoten, die sie verbindet und den benachbarten Verbindungen, welche an die Knoten anknüpfen. Ein Nachteil an diesem Verfahren entsteht durch Fälle, in denen es einen Knoten gibt, der im Zentrum des Graphen steht und verschiedene Teile der Anwendung zusammenhält. Durch einen großen Nenner verlieren die einzelnen Kanten an Bedeutung.

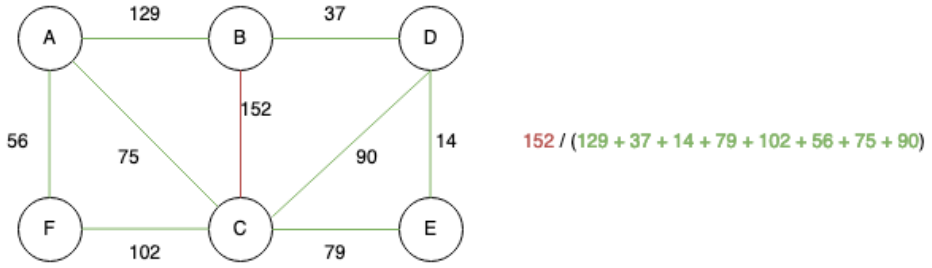

Abbildung 3.7: Worst-Case Szenario des Nachbarschaftsverfahren

3.4.2 Join-Kontext

Beim Verfahren der relativen Häufigkeiten wurden ausschließlich Paare für die Berechnung des Schnitts in Erwägung gezogen. Um Kontext in die Bewertung mit einfließen zu lassen wurde zuerst das Vorkommen der Tabellen eines Paares in anderen Paaren berücksichtigt. Abschließend wurde das Nachbarschaftsverfahren vorgestellt, welches zudem noch die benachbarten Kantengewichte berücksichtigt. Der Erhalt des Join-Kontextes sorgt dafür, dass die betrachteten Paare als Teil eines Ganzen angesehen werden.

J ist ein Join über den Tabellen $A, B, C, D \longleftrightarrow A, B, C, D$ bilden Paare $p \in P$ und $P \in J$ Die Länge eines Joins J ist definiert als $len_J = \|P\|$

3.4.2.1 Max-Join

Join-Kontext basierte Verfahren, können erweitert werden, indem man den maximalen-Join ermittelt und entfernt. Die Begründung für das Entfernen der maximalen Join-Kette wurde im Abschnitt 3.2 behandelt. Sei V eine Menge von Joins, die mit Hilfe des DP extrahiert wurden. Ein Join $j \in V$ wird als maximaler-Join oder Max-Join bezeichnet $\longleftrightarrow \forall J_i \in V \exists J$ mit $Len_J > LenJ_i$. Da es in einem System mehr als eine Reporting-Abfrage geben kann, macht es Sinn die durchschnittliche Join-Länge zu ermitteln. Join-Abfragen die länger als der Durchschnitt sind werden ebenfalls entfernt.

3.4.2.2 Fuzzy Join

Der Fuzzy Join oder Min-Join ist ein Join, der die potenziellen Grenzen zwischen zwei Domänen verschwimmen lässt. Ein Beispiel dafür ist ein Join, der Tabellen mit einer User-Tabelle verknüpft. Während des Praxissemesters gab es in der firmeninternen Anwendung Joins zwischen einer Benutzer- und einer Trainer-Tabelle. Diese waren stark miteinander verknüpft. Dadurch kann der Eindruck entstehen, dass diese Tabellen nicht zu trennen sind und zu einer Domäne aggregiert werden.

Die Ermittlung und Entfernung des Fuzzy -Joins ist keine Trivialität. Aus dem Praxissemester geht hervor, dass es sich bei Fuzzy Joins nicht immer um einen Join handelt, der nur genau zwei Tabellen– also das Minimum an benötigten Tabellen– verknüpft. Ein Ansatz zur automatisierten Entfernung des Fuzzy Joins könnte darin bestehen, Joins J mit $Len_J = 1$ und einer darin vorkommenden Tabelle, welche das Wort `user` enthält, auszusortieren. Abhängig von der Anwendung können Benutzer authentifizierende Funktionen leicht zu isolieren sein (Dehghani (2018)). Im Vergleich zum maximalen Join, wo die Anzahl an Tabellen in einer Join-Kette einen Hinweis auf das Vorliegen einer Reporting-Abfrage liefert, ist die Länge bei einem Fuzzy-Join kein ausreichender Hinweis. Der Fuzzy Join kann als Teil, in einer längeren Join-Kette enthalten ein. Eine weitere Möglichkeit zur Entfernung kann mit Hilfe eines Datenbankexperten durchgeführt werden. Dieser kann die Tabellen, welche einen möglichen Fuzzy-Join enthalten benennen.

3.4.2.3 Sub Joins

Sub-Join-Ketten sind Join-Ketten, die in anderen Joins vollkommen enthalten sind. Seien J und J' Joins. J ist ein Join über eine Menge von Tabellen A. Analog dazu ist J' ein Join über die Tabellenmenge A'. J' ist ein Sub-Join von $J \longleftrightarrow A' \subset A$

Da unter Betrachtung des Join-Kontexts nicht mehr die Paare direkt und indirekt gezählt werden, kann die Häufigkeit des Sub-Joins genutzt werden, um das Vorkommen eines Joins weiter hervorzuheben. Das heißt, wenn J' komplett in J enthalten ist, kann das Vorkommen von J um das Vorkommen von J' amplifiziert werden. Der Grund dafür beruht auf der Annahme, dass die Tabellen, die im Komplement der Schnittmenge von A und A' enthalten sind, basierend auf dem Sub-Join an Bedeutung verlieren. Dadurch wird das Verfahren möglicherweise grob granularer, sodass die Tabellen des Joins J stärker zusammenhängend sind. Ein Beispiel dafür sind zwei Joins J und J' mit J = $[\{A, B\}, \{A, C\}, \{B, C\}, \{B, D\}, \{C, E\}]$ und $J' = [\{A, B\}, \{B, C\}, \{B, D\}]$. J' ist als Teilmenge, paarweise in J enthalten. Das Kompliment der Schnittmenge $A\,([\{A, C\}, \{C, E\}])$ könnte im Fall, dass die Sub-Joins nicht betrachtet werden, an Relevanz verlieren.

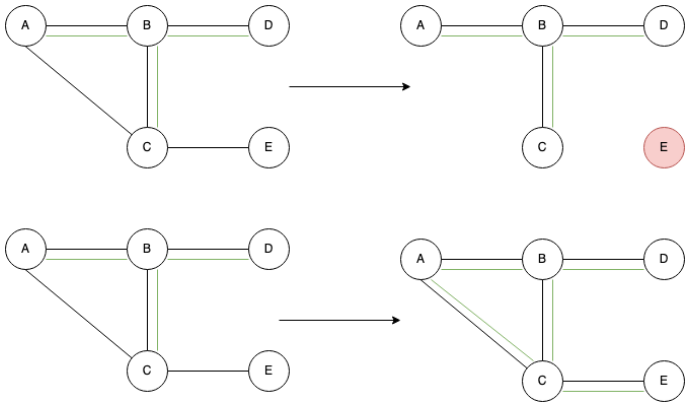

Abbildung 3.8: Darstellung von Sub-Joins

3.4.2.4 Gleiche und ungleiche Join-Ketten

Unter Betracht des Join-Kontexts, ist es wichtig Regeln festzulegen, um zu entscheiden, wann zwei Join-Ketten als gleich gelten. Zuvor, im Rahmen der Zählung des Verfahrens der relativen Häufigkeiten, wurde kein Unterschied gemacht zwischen einer Verknüpfung von einer Tabelle A mit B und einer Verknüpfung von B mit A. Die erste Eigenschaft, auf die zwei Joins geprüft werden müssen, um festzustellen, ob sie gleich sind, ist die Länge des Joins. Seien J und J' zwei Join-Ketten über den Tabellen A und B. Daraus folgt $Len_J = Len'_J = 1$.

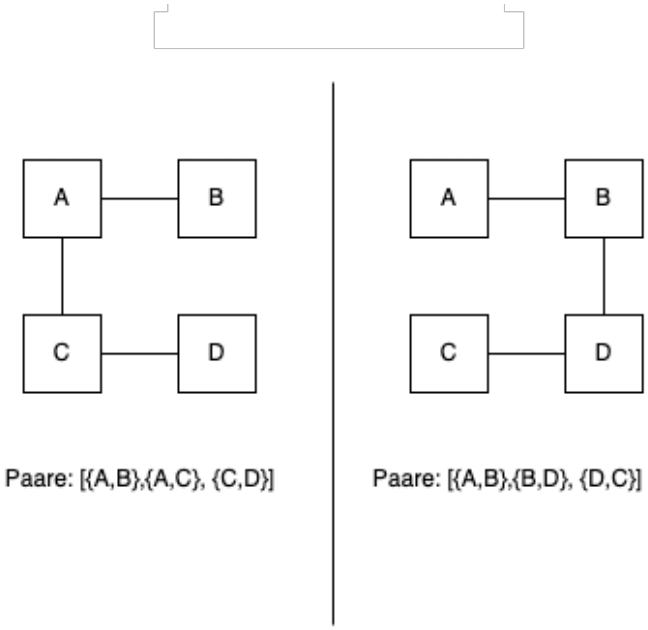

Abbildung 3.9: Beispiel: Ungleiche Joins

3.4.2.5 LCOM: Lack Of Cohesion

Joins spiegeln den Gebrauch von Funktionaliäten der Anwendung wieder. Die Datenbank ist ein Abbild der Beziehungen der Klassen des Monolithen. Daher macht es Sinn die LCOM-Metrik zu betrachten. Lack of Cohesion ist eine Metrik, welche das Konzept der Kohäsion auf Klassen in einer Code-Basis anwendet (aivosto ([o. D.])). Für den Zweck dieser Arbeit wird LCOM wie von Chidamber und Kemerer definiert betrachtet (Kleine (2013),aivosto ([o. D.])). Die Metrik wird benutzt, um die Qualität von objekt-orientierten Code zu messen. LCOM basiert auf der Differenz von paarweise nicht-kohäsiven und paarweise kohäsiven Methoden einer Klasse. Dabei entstehen zwei Mengen, P und Q. P ist die Menge der Methoden, die kein gemeinsames Attribut einer Klasse teilen. Q ist die Menge der Methoden-Paare einer Klasse, welche mindestens ein gemeinsams Attribut haben.

$$LCOM = \begin{cases} \|P\| - \|Q\| & \text{if } \|P\| - \|Q\| > 0 \\ \\ 0 & \text{sonst} \end{cases}$$

Der vorliegende Sachverhalt unterscheidet sich von dem eigentlichen Gebrauch der Metrik. LCOM ist darauf ausgelegt die Kohäsion von Klassen anhand der sich darin befindenden Methoden und Attributen zu berechnen. Es müssen Annahmen getroffen werden um die Übersetzung der Metrik zu gewehrleisten. LCOM soll dazu genutzt werden, die Kohäsion der Anwendung als Ganzes anhand der Joins zu bestimmen. Dazu sollen P und Q für alle Kombinationen der Joins berechnet werden. Aus der Summe dieser Berechnungen geht hervor, wie stark ein Join am Rest der Anwendung hängt. Die Joins repräsentieren die Methoden und die darin vorkommenden Tabellenpaare spiegeln die Attribute wider. So ergeben sich die Mengen P und Q als die Mengen der Joins, die keine Tabellenpaare gemein haben und die Menge der Joins, die sich Tabellenpaare teilen. Ist die Differenz von P und Q „groß“, so bedeutet dies, ein Join teilt sich wenige Tabellenpaare mit andere Joins. Demnach ist es eine Überlegung wert, diesen Teil der Anwendung abzukapseln, um das SRP umzusetzen. Ist die Differenz der beiden Mengen „klein“, teilt sich dieser Join viele Tabellenpaare mit anderen Joins. Daraus kann geschlossen werden, dass Tabellen die in einem Join vorkommen, stark an die Anwendung gekoppelt sind.

Daraus folgt, LCOM für einen Join $j \in J$ ergibt sich aus:

$$\sum LCOM(j, j_i), \forall j_i \in J \tag{3.6}$$

Der Fall $\|P\| - \|Q\| < 0$ wird zugelassen, da dies einen negativen Einfluss auf die Summe hat, was insgesamt zu einem niedrigeren Wert führt.

3.4.2.6 Join-Stärke

Die bisher vorgestellten Verfahren sind auf die eine oder andere Art von der Häufigkeit, mit welcher die Join-Ketten oder Tabellenpaare auftreten, abhängig. Reduziert man den Datensatz auf einzigartige Join-Ketten, so lässt sich die Anwendung anhand der Join-Stärke zerlegen. Dabei soll ersichtlich werden, welche Tabellen von den individuell auftretenden Joins verwendet werden. Demnach ist dieses Verfahren nicht auf die Häufigkeit der Joins angewiesen, sondern auf das Ausführen einer Funktion im Monolithen, welche einen Join

auslöst. Es kann zu Problemen führen, wenn während dem Zeitraum, in welchem Daten gesammelt werden, Joins, die Teil der Anwendung sind nicht ausgelöst werden. Für dieses Verfahren muss das Datenaufbereitungswerkzeug angepasst werden. Die Bewertung der Kanten spielt bei diesem Verfahren keine Rolle, da jede Kante jedes Joins genau einmal abgebildet wird. Existieren zwei Joins J über den Tabellen A und B, sowie J' über den Tabellen A, B und C, so können zwei Kanten $e(n_1, n_2)$ und $e'(n'_1, n'_2)$ existieren, mit $n_1 = n'_1$ und $n_2 = n'_2$

3.4.3 Ähnlichkeitsverfahren

Da die zu zerlegende Anwendung im Frontend als Graph visualisiert wird, macht es Sinn, sich mit Methoden aus der Graphentheorie zu beschäftigen. Im Vergleich zu vorherigen Verfahren, welche von der Häufigkeit der vorkommenden Joins abhängig sind, beruht dieses Verfahren auf der Ähnlichkeit der Knoten, die im resultierenden Graph eingezeichnet werden. Da jeder Knoten im Graph eine Tabelle widerspiegelt, werden mit diesem Verfahren Tabellen gruppiert, die ähnliche Nachbarn haben. Dies geschieht, indem für jeden Knoten im Graph, die Ähnlichkeit zu allen anderen Knoten berechnet wird. Die Ähnlichkeit wird anhand der gemeinsamen Nachbarn bestimmt. In Abschnitt 3.1.1 wurde die Nachbarschaft von Knoten definiert. Dieses Verfahren ist das erste unter den vorgestellten, welches keine Kanten entfernt, sondern die Knoten, basierend auf ihrer Ähnlichkeit zu einem Cluster gruppiert.

Das Ähnlichkeitsverfahren benötigt, genau wie das Verfahren der Join-Stärke, jeden Join nur einmal. Die Ähnlichkeitsfunktion wird definiert als:

$$S : \mathbb{N} \times \mathbb{N} \to \mathbb{R} \tag{3.7}$$

Die Funktion bildet zwei Knoten auf eine Gleitkommazahl ab, welche der Ähnlichkeit dieser beiden Knoten entspricht. Um die Ähnlichkeit zu berechnen werden Mengen verwendet. Seien A, B zwei Mengen, die die Nachbarn von zwei Knoten n_1 und n_2 beinhalten. Die Ähnlichkeit von n_1 und n_2 ist definiert als

$$\frac{|A \cap B|}{|A \cup B|} \tag{3.8}$$

Man spricht auch vom Jaccard-Index der Mengen A und B (Rees (2021)). Es handelt sich bei der Ähnlichkeit von zwei Knoten um das Verhältnis von selben Nachbarn zu allen gemeinsamen Nachbarn. Dieses Verfahren kann mit anderen Verfahren verknüpft werden um ein „hybrid“-Verfahren zu bilden. Verwendet man zusätzlich ein Verfahren welches auf der Häufigkeit von Tabellenpaaren basiert, welche Teilweise den Gebrauch der Anwendung wiederspielgen, könnte man abschätzen, wieviel Kommunikation es zwischen den Services geben wird.

Der Schwellwert, welcher entscheidet ab wann Knoten ähnlich genug sind, um gemeinsam in einem Cluster zu landen, kann variabel gestaltet sein. Zu Beginn kann als Schwellwert für die Ähnlichkeit der zu clusternden Knoten, die durchschnittliche Ähnlichkeit gewählt werden. Anschließend kann dieses Verfahren mehr oder weniger selektiv werden, indem eine Standardabweichung vom Durchschnittswert auf den Schwellwert addiert oder subtrahiert wird.

Eine alternative Möglichkeit um das Ähnlichkeitsverfahren umzusetzen ist, den Overlap-Koeffizenten zu nutzen um die Ähnlichkeit der Knoten zu bestimmen. Dieser unterscheidet sich vom Jaccard-Koeffizienten im Nenner. Anstatt den Schnitt der gemeinsamen Nachbarn in Relation zu allen Nachbarn zu betrachten, wird der Schnitt in Relation zu dem

Minimum der beiden Mengen gesetzt.

$$\frac{|A \cap B|}{\min A, B} \tag{3.9}$$

Dies führt dazu, dass die Ähnlichkeit zweier Knoten davon abhängt, ob die kleinere, der beiden Nachbarmengen im Schnitt enthalten ist.

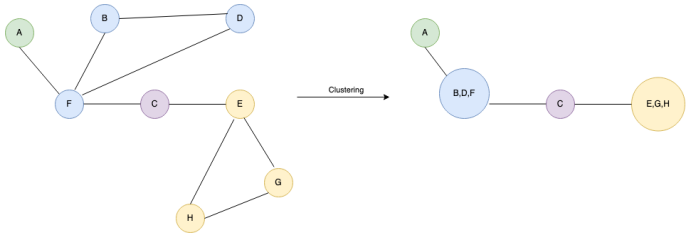

Abbildung 3.10: Darstellung vom Clustering durch das Ähnlichkeitsverfahren

3.5 Konzeption: Soll-Zustand

Bevor eine mögliche Implementierung für ein Werkzeug zum Ermitteln von Domain-Grenzen vorgeschlagen werden kann, müssen Rahmenbedingungen in Form eines Konzepts festgehalten werden. Anhand dieser Konzeption wird gemessen, ob das Werkzeug, welches im Rahmen dieser vorgelegten Arbeit, entwickelt wurde, die gesetzten Ziele erreicht.

3.5.1 User Stories

In diesem Abschnitt werden die User-Stories gesammelt, welche als Grundlage für die Implementierung dienen.

- Als Berater möchte ich das Werkzeug über ein Interface steuern können, damit die Bedienung einfacher ist.
- Als Berater möchte ich die dargestellten Daten gruppieren um so das Ergebnis ablesen zu können.
- Als Berater möchte ich Informationen über die Beziehungen der Geschäftsobjekte haben, um mir ein Bild von der Domäne zu machen.
- Als Berater möchte ich das Ergebnis inkrementell erzeugen können, um das Ergebnis anhand der Historie nachvollziehen zu können.
- Als Datenbankexperte möchte ich in der Lage Abfragen aus der Auswertung ausschließen, um eine bessere Auswertung zu erhalten.
- Als Datenbankexperte möchte ich Abfragen, welche ein Rauschen in der Auswertung erzeugen, aus der Auswertung entfernen.
- Als Kunde möchte ich das Ergebnis visuell sehen können, um das Ergebnis nicht aus einer Datei ablesen zu müssen.
- Als Kunde möchte ich die Visualisierung anpassen können, um eine bessere Übersicht zu haben.

Aus den beschriebenen User-Stories, lassen sich drei Akteure identifizieren.

- Den Berater, welcher schnell Informationen über die Domäne haben will.

- Den Datenbankexperten, welcher sich Kontrolle über die in das Auswertungswerkzeug eingegebenen Daten wünscht.
- Der Kunde, welcher sich eine intuitive Oberfläche wünscht, die ihm das Ergebnis anzeigt.

Die grundlegenden Bedingungen der Software wurde in einem Meeting erarbeitet. Ideen zu neuen Verfahren wurden aufgrund von gewonnenen Erkenntnissen entwickelt oder kamen durch Brainstorming in Meetings zu stande.

Die Kunden sind Firmen, welche Interesse daran haben ihren Monolithen in Microservices aufzuteilen. Berater sind Teil des Teams, welches für die Beratung zur Aufteilung des Monolithen hinzugezogen werden. Datenbankexperten, welche bei der Migration zu Microservices helfen, können die einzugebenen Daten vor der Auswertung manipulieren. Die Datenbankexperten kennen sich mit der Datenbank aus und wissen ob es Reporting-Anfragen gibt und können sie entfernen.

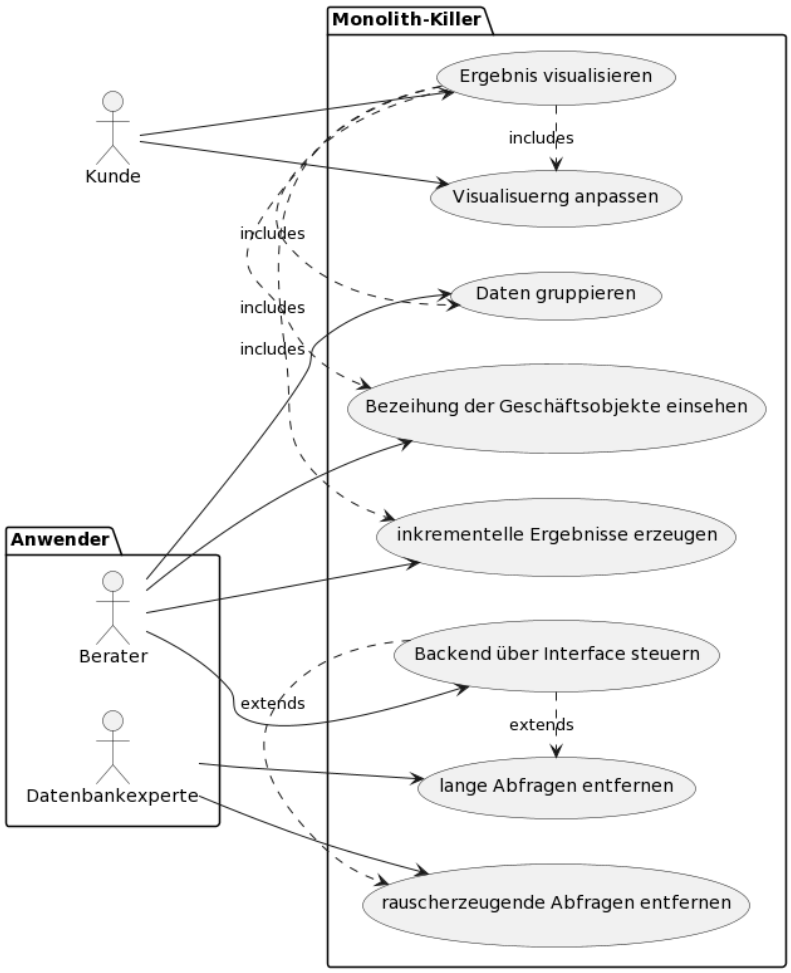

Abbildung 3.11: Use Case Diagramm (Erstellt mit PlantUML)

3.5.2 Technologien: Ist-Zustand vs. Soll-Zustand

Das Frontend des Prototyps soll mit neuen und aktualisierten Technologien umgesetzt werden. Die im Prototyp verwendete Skriptsprache ist JavaScript. In der Neuimplementierung des Frontends soll TypeScript verwendet werden. Dadurch soll Typsicherheit bezüg-

lich der in das Frontend geladenen Daten eingeführt werden. Zudem wird ein einheitliches Datenformat zwischen Front- und Backend eingeführt, was den Entwicklungsprozess erleichtert. Da das im Prototyp genutzte Framework nicht mehr weiterentwickelt wird, wird es ersetzt.

Statt über einen Regler, welcher den Threshold steuert, ab wann eine Kante aus dem Graph entfernt wird, sollte das neue Frontend dies über Buttons umsetzen. Per Button-Click soll die Kante mit dem geringsten Gewicht ausgeblendet werden. Die ausgeblendeten Kanten können auch wieder eingeblendet werden, durch das Clicken auf einen anderen Button. Zudem soll es zwei Buttons geben, die alle Kanten aus- oder einblenden können. Die Dicke einer Kante e zwischen zwei Knoten n_1 und n_2 soll dem Kantengewicht entsprechen. Das heißt, hat eine Kante ein verhältnismäßig hohes Kantengewicht, so soll dies durch eine dick gezeichnete Kante deutlich werden. Letztlich soll es einen Button geben, über den man per Click die Knoten des Netzwerks nach der Cluster-ID gruppieren kann. Das Cluster wird durch ein von einem normalen Knoten zu unterscheidendem Muster dargestellt. Der Name des Knotens, der ein Cluster abbildet, besteht aus allen Knoten-IDs, die sich im Cluster befinden. Per Click auf ein Cluster, löst sich dieses zu den sich darin befindenden Knoten auf.

Das Backend soll refaktoriert werden, so dass es künftig mit neuen Verfahren erweitert werden kann. Zudem soll es möglich sein, das gewählte Verfahren mit einer Variable zu steuern. Es soll möglich sein, zwischen einem Häufigkeitsverfahren und einem Verfahren, dass auf dem Join-Kontext beruht zu wählen. Die Verfahren sollen möglichst das gleiche Datenformat beibehalten. Natürlich ist dabei zu beachten, dass bei Häufigkeitsverfahren der Join-Kontext nicht enthalten ist und die Daten dementsprechend leichte Veränderungen aufweisen. Das Datenaufbereitungswerkzeug erhält als Eingabe eine SQL-Logdatei und erzeugt als Ausgabe, entweder eine Liste von Paaren, falls der Join-Kontext zu ignorieren ist, oder eine Liste mit Joins. Ein Join ist ein selbstgeschriebener Datentyp, welcher über eine Liste von Paaren verfügt. Ein Paar besteht aus zwei `String`-Variablen, die die Tabellennamen zwei verknüpfter Tabellen aus dem Join enthalten. Neben einer Liste von Paaren enthält ein Join eine Zählvariable, welche die Häufigkeit, mit welcher der Join im SQL-Log vorkommt, festhält. Eine Menge von Joins bilden einen Graph. Aus diesem werden im letzten Schritt der Datenaufbereitung die Knoten und Kanten für die Visualisierung gewonnen. Knoten und Kanten sind ebenfalls selbstgeschriebene Datentypen, welche sich dem vorgegebenen Format des Frontend-Frameworks zur Graph Visualisierung anpassen.

Für die Datei Ein- und Ausgabe wird eine wiederverwendbare Klasse erstellt. Um die aufbereiteten Joins vor der Analyse statistisch auswerten zu können und um lange Joinketten oder Fuzzy-Joins zu entfernen, wird eine weitere Klasse erstellt, welche als Zwischenschritt dient. Diese arbeitet mit den aufbereiteten Join-Ketten.

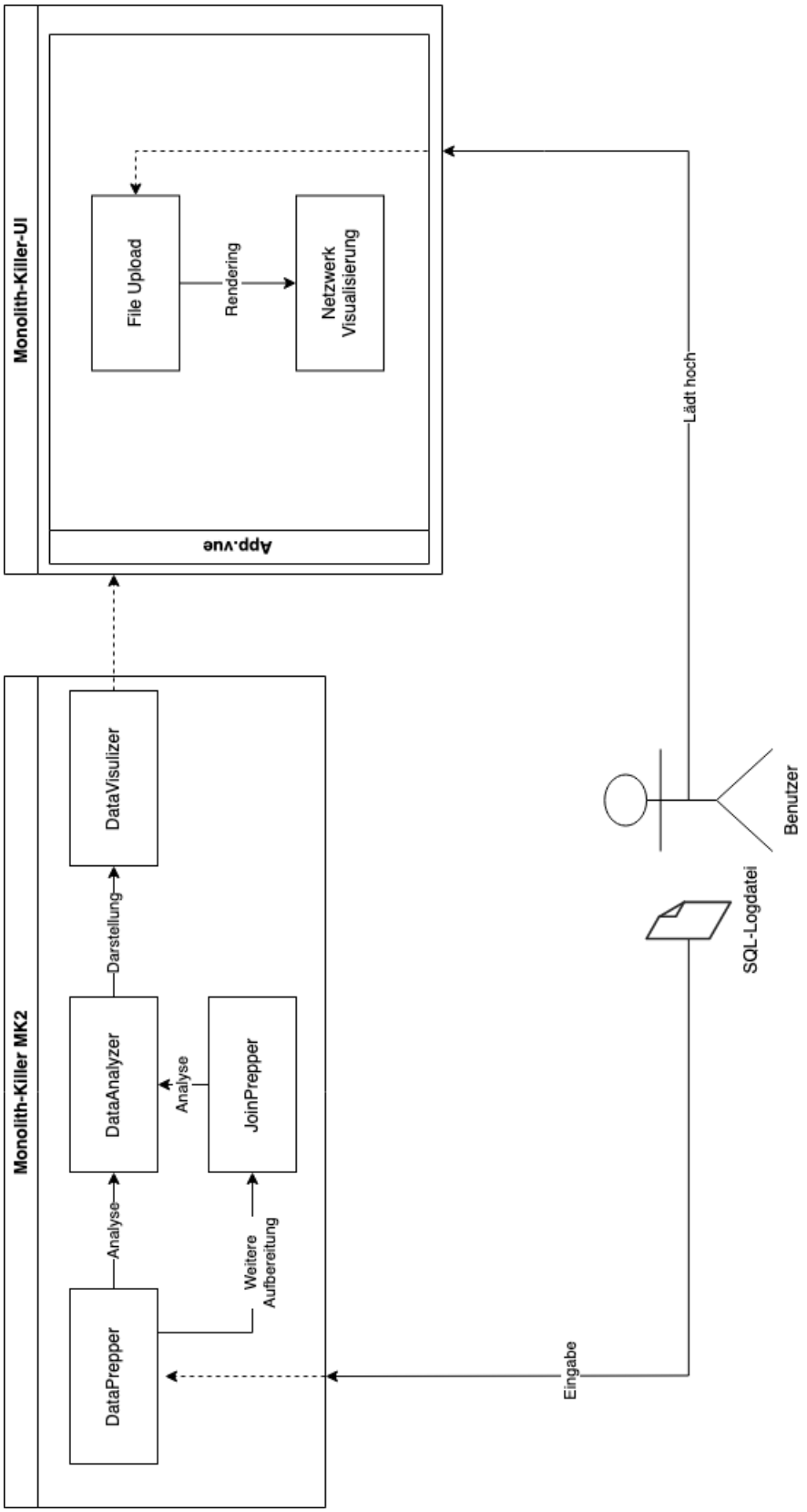

Abbildung 3.12: Abbildung der Architektur des Monolith-Killers MK2 (Erstellt mit draw.io)

4 Anwendung

In diesem Kapitel werden die Implementierungsdetails behandelt. Dies beinhaltet unter anderem die Algorithmen der Verfahren, die in Kapitel 3 vorgestellt wurden und die Umsetzung der Visualiserungskomponente.

Während des Praxisprojekts lag der Fokus auf der Konstruktion eines funktionierenden Prototyps. Das Prinzip des Test-Driven-Development (TDD) wurde während der Entwicklung angewandt. Dabei wurde die Wiederverwendbarkeit von Funktionalitäten vernachläßigt. Ganz besonders davon betroffen waren die Funktionen, welche den Datei-In- und Output regeln. Diese wurden einmal für die Aufbereitungskomponente (DataPrepper, DP) angefertigt und schließlich in das Auswertungswerkzeug (DataAnalyzer, DA) kopiert.

4.1 Refactoring Backend: IOUtils

Als ersten Schritt des Refactorings wurden alle Methoden, die für die Verarbeitung des Ein- und Ausgabepfads verantwortlich sind, in eine eigene Datei (`IOUtils.kt`) ausgelagert. Das Schreiben in eine Datei ist in vier Schritte aufgeteilt. Im ersten Schritt wird der Eingabepfad erfasst und in eine Liste aufgeteilt. Ein Datei-Pfad besteht aus einer Zeichenkette, welche die Namen der darin vorkommenden Verzeichnisse mit einem Schrägstrich (`/`) trennt. Der Schrägstrich kann als Separator genutzt werden, um die verschiedenen Komponenten des Pfads zu identifizieren.

```
fun getInputPathAsArray(inputDirectoryFromArgs: String): List<String>
    = inputDirectoryFromArgs.split("/")
```

Abbildung 4.1: Methode zum Zerteilen des Date-Pfads

Im zweiten Schritt wird die Liste dazu verwendet einen `String` zu generieren, der den Datei-Pfad ohne den Dateinamen beinhaltet. Dies ist wichtig für den dritten Schritt, in dem der Datei-Pfad um einen neuen Dateinamen erweitert wird. Dieser neue Pfad wird genutzt, um die aus dem Verfahren resultierende Datei auf die Festplatte des System zu schreiben.

```
fun getInputFileDirectory(inputDirectoryFromArgs: String): String =
    (0..getInputPathAsArray(inputDirectoryFromArgs).size - 2).fold("")
        { acc, i -> "$acc${getInputPathAsArray(inputDirectoryFromArgs)[i]}/" }
```

Abbildung 4.2: Methode zum Erhalten des Datei-Pfads

Die Methode zum Schreiben einer neuen Datei wurde angepasst, sodass alle Datei erzeugenden Schritte des Gesamtprozesses diese verwenden können. Die Notwendigkeit entstammt der Tatsache, dass das Werkzeug weiterhin das alte Verfahren, welches auf Paaren basiert unterstützen muss. Zeitgleich, muss das Beschreiben von Dateien mit den neu entworfenen Datenstrukturen gewährleistet sein.

4.2 Refactoring Backend: Erweiterung des Datenmodels

Um den Anforderungen gerecht zu werden und zusätzliche Funktionalitäten einzubinden, wurde das bestehende Datenmodell erweitert. In Kotlin können `data classes` verwendet werden, wenn die Hauptaufgabe der Klasse das Halten von Daten ist (Kotlin ([o. D.])).

Das Datenmodell bestand ursprünglich aus drei Daten-Klassen (`Node`,`Link`,`Elements`). Diese spiegelten den Graphen wider, welcher im Frontend visualisiert wurde. Die Klasse `Elements` ist ein Aggregat, welches sowohl eine Liste von Nodes als auch Links enthält. `Elements` wird nach Abschluss der Auswertung in eine Datei geschrieben, welche vom Frontend als Eingabe genutzt wird, um die sich darin befindenden Knoten und Kanten zu visualisieren. Die Klassen `Node` und `Link` enthalten Attribute, welche vom Framework des Frontends vorgegeben wurden. Zu diesen Attributen gehörten beispielsweise `name: String` und `id: String` Der Name wird für die Anzeige des Knoten-namens benötigt. Die ID wird für die eindeutige Identifikation des Knotens benötigt. Da die Namen der Knoten bereits die Tabellennamen enthielten, entschied man sich für die ID ebenfalls den Tabellennamen zu verwenden. So wurden Knoten mit zwei identischen `String`-Werten befüllt. VisJS, das Framework, welches im Zuge des Refactorings das alte Framework, vue-network-3d ersetzten wird, benötigt ein Attribut das `label` heißt. Demnach musste das Attribut `name` umbenannt werden. Um den Konstruktoraufruf besser zu gestalten, wurden `JSON`-Annotationen genutzt.

```
data class Node(val id: String, var clusterId: Int? = 0) {
    val label: String @JsonGetter get() = id
}
```

Abbildung 4.3: Ausschnitt der Node-Klasse

Die `JSON`-Annotation `@JsonGetter` ermöglicht die Befüllung des Feldes `label` mit dem Inhalt des Attributs `id`, ohne den Wert ein zweites Mal im Konstruktor angeben zu müssen. Zudem hat die `Node`-Klasse eine ID für die Zugehörigkeit eines Clusters erhalten. Dies ist für das Ähnlichkeitsverfahren von Bedeutung.

In der Klasse, welche die Kanten des Graphs repräsentieren, wurden Umbenennungen vorgenommen, um sich dem Frontend-Framework anzupassen. Zudem wurde ein Attribut `color: String` eingeführt, für das Verfahren der Join-Stärke. Anhand der für jeden Join individuell generierten Farben, kann der genaue Verlauf des Joins beobachtet werden.

Da mit der Implementierung der neuen Verfahren der Join-Kontext berücksichtigt wird, wurde eine weitere Klasse `Join` eingeführt. Diese Klasse dient als Verkapslung der Paare eines Joins, welche in den alten Verfahren, unabhängig voneinander in die resultierende Datei geschrieben wurden.

```
data class Join(var pairs: List<Pair<String, String>>, var count: Int = 1,
                var color: String? = "#1976d2")
```

Abbildung 4.4: Ausschnitt der Join-Klasse

Die `count`-Variable wird verwendet, um das Vorkommen des Joins zu speichern. Ein Join wird mit `count = 1` initialisiert, da das Instanziieren seine Existenz impliziert.

Eine letzte neue Klasse `Graph` wurde zum Datenmodell hinzugefügt, welche als Aggregat der Joins dient.

4.3 Refactoring Backend: DataPrepper

Der DP ist das Aufbereitungswerkzeug, welches aus den SQL-Abfragen die benötigten Daten zur Auswertung extrahiert. Die Main-Methode des DP führt eine Reihe von Methoden aus, die ineinander gekapselt sind. Anschließend wird über jedes, ermittelte SQL-Statement iteriert, um die Paare zu identifizieren.

```
fun main(args: Array<String>) {
    writePairSet(
        provideDirectoryAndFileName(args),
        getLowerCaseSQLStatements(getJoinContainingStatementsOnly(
            getStatements(args[0]))).flatMap {
            mapBackAliases(getPairs(it), getTableNameAliasMap(it))
        })
}
```

Abbildung 4.5: Main-Methode des DP

Der DP unterlief mehreren Veränderungen als ursprünglich geplant. Zu Begin sollte der DP dahingehend angepasst werden, neben Paaren, auch Joins aufbereiten zu können. Diese Anpassung war leicht umzusetzen. Da Paare in einer Liste gespeichert wurden (`List<Pair<...>>`) und ein Join ebenfalls eine Liste von Paaren ist, musste die Main-Methode des DP angepasst werden. Dazu, wurde die `mapBackAliases`-Methode, welche eine Liste von Paaren für das jeweilige SQL-Statement zurückliefert, mit einem Konstruktoraufruf, der Join-Klasse umhüllt.

Damit war die Arbeit jedoch noch nicht getan. Es wurden neue Datensätze eingeführt, die ein anderes SQL-Format verwenden. Um das Problem weiter beschreiben zu können, lohnt es sich die Methoden zu betrachten, die für das Identifizieren der Paare genutzt werden. In den Testdaten des Praxisprojekts waren Joins so formatiert, dass auf den Tabellennamen ein Alias folgte.

```
select * from Users users_0 left outer join Trainer trainer_1
    on users_0.users_0_id=trainer_1.user_id left outer join ...
```

Abbildung 4.6: Beispiel für einen Join aus den Testdaten

Von da an, sollte eine `Map`-Datenstruktur verwendet werden, um die Alias auf die Tabellennamen abzubilden. Zum Extrahieren dieser Abbildung von Alias auf den Tabellennamen wurden reguläre Ausdrücke verwendet.

```
fun getTableNameAliasMap(sqlQuery: String): Map<String, String> {
    return "(?<=from\\s|join\\s)(\\w+\\s\\w+)".toRegex().findAll(sqlQuery)
        .associate{ match ->
            match.value.split(" ").toTypedArray().let { it[1] to it[0] } }
}
```

Abbildung 4.7: Methode zum Erhalten der Abbildung von Alias auf Tabellenname

Bezogen auf 4.6 werden damit `Users users_0` und `Trainer trainer_1` ermittelt. Da der Tabellenname und das dazugehörige Alias mit einem Leerzeichen getrennt ist, wird dieses als Separator genutzt.

Um die Paare zu identifizieren, die über einen Join miteinander Verbunden werden, wurde ebenfalls ein regulärer Ausdruck verwendet. Die `on`-Klausel eines Joins beschreibt welche Tabellen über welches Attribut verknüpft werden.

```
fun getPairs(sqlQuery: String): List<Pair<String, String>> {
    return "(?<=\\son\\s|=)(\\w+)".toRegex().findAll(sqlQuery).toList()
        .map { it.value }.chunked(2).flatMap { it.zipWithNext() }
}
```

Abbildung 4.8: Methode zum Erhalten der Paare

Die Paare, bestehend aus den Alias, werden schließlich mit der `mapBackAliases`-Methode zurück auf die Tabellenpaare gemappt.

```
fun mapBackAliases(aliases: List<Pair<String, String>>,
                   aliasMap: Map<String, String>):
        List<Pair<String, String>> =
    aliases.map { Pair(mapBackAlias(it.first, aliasMap),
        mapBackAlias(it.second, aliasMap)) }
```

Abbildung 4.9: Methode zum zurück mappen von Alias auf Tabellenname

Die Ermittlung der Tabellenpaare musste angepasst werden, um die neuen Datensätze verarbeiten zu können. Der Hauptunterschied der neuen Daten bestand in der genutzten SQL-Syntax. Der Open-Source Datensatz wies Unregelmäßigkeiten bei den Join-Verknüpfungen auf. In einigen Join-Operationen wurden Aliasnamen verwendet, in anderen wurden Klammern verwendet. Weiterhin gab es Join-Abfragen, die sich auf das Datenbanksystem selbst bezogen. Diese hatten keinen Bezug zu der eigentlich Anwendung. Um mit dem Datensatz besser arbeiten zu können, wurde dieser manuell angepasst und vereinheitlicht. Daraus resultierte eine Methode, welche den Schritt die Aliasnamen auf Tabellennamen abzubilden, überspringen kann.

4.4 Refactoring Backend: DataAnalyzer

Der DataAnalyzer ist für die Berechnung der Metriken der Verfahren zuständig. Dieser unterlief die meisten Veränderungen aufgrund der neu implementierten Verfahren. Zuvor musste das Refactoring des DA vorgenommen werden. Anhang 8.9 zeigt die `main`-Methode des DA. Das Refactoring wurde in drei Stufen unterteilt.

- Extrahieren des Schreibvorgangs.
- Vereinfachen des Quellcodes.
- Erweitern der Funktionalitäten des DA.

Das Refactoring des Schreibvorgangs wurde bereits in Abschnitt 4.1 besprochen. Wie im DP, wird die Methode, welche die Ergebnisse erzeugt, von der `writeToFile`-Methode umhüllt. Die `main`-Methode wurde aufgelöst und ersetzt durch `calculateRelativeHaeufigkeit`. Diese gibt eine Map (`HashMap<Pair<String, String>, Double>`) zurück.

Die Ermittlung der direkten und indirekten Paare wurde vereinfacht. Eine neue Methode `getDirectAndIndirectOccurence()` wurde eingeführt, in welche der Prozess ausgelagert wurde. Innerhalb der Methode eine `forEach`- anstelle einer `while`-Schleife verwendet. Dadurch besteht keine Notwendigkeit mehr in jeder Iteration ein Element aus der Tuple-Liste zu entfernen. Weiterhin wird nun keine Kopie der Liste mehr benötigt.

```
fun getDirectAndIndirectOccurence(/*...*/): Tuple<Double, Double> {
    val directOccurence = tuples.filter { directCompareTuple ->
        directComparePairs(
            currentTuple,
            directCompareTuple
        )
    }
    val indirectOccurence = (tuples - directOccurence.toSet()).filter
    { indirectCompareTuple ->
        indirectComparePairs(
            currentTuple,
            indirectCompareTuple
        )
    }
}
```

Abbildung 4.10: Ausschnitt der ausgelagerten Funktion nach dem Refactoring

Mit dem `distinct()`-Aufruf auf `tuples` wird erreicht, dass jedes existierende Paar durchlaufen wird. Dieses Refactoring führt zudem dazu, dass keine Listenmanipulationen mehr stattfinden.

Die `directComparePairs`-Methode überprüft, ob die beiden verglichenen Paare dieselben Komponenten beinhalten und vernachlässigt dabei die Reihenfolge.
Bei der `indirectComparePairs`-Methode wird überprüft, ob mindestens eines der beiden Elemente eines Paares in dem anderen Paar vorkommt.

Nachdem die direkten und indirekten Paare ermittelt wurden, kann die Berechnung der Metrik beginnen. Zu jedem individuellen Paar existiert ein Eintrag in `dict`. Dieses bildet

ein Paar von Strings (Tabellennamen) auf eine Gleitkommazahl (Kantengewicht) ab. An der Berechnung für die Metrik wurde nichts verändert.
Sie wird durch die `calculateLinkWeight`-Methode aufgerufen und ist definiert als:

```
fun calculateLinkWeight(direct: Double, indirect: Double): Double
= when (indirect) {
    0.0 -> 1 - sqrt((1 - (1 / direct * 2)))
    else -> 1 - sqrt(((1 - (1 / direct * 2)) - (1 - (1 / indirect))).pow(2))
}
```

Abbildung 4.11: Methode zur Berechnung des Kantengewichts

Die resultierende `HashMap` wird in eine neue Methode (`createElements`) gespeist um die Kanten und Knoten zu generieren.

4.5 Refactoring Backend: DataVisualizer

Für Join-Kontext basierte Verfahren wurde das Erzeugen der Elemente in ein neues Modul ausgelagert. Der DataVisualizer (DV) erzeugt die Kanten und Knoten basierend auf dem Ergebnis vom DA. Da für das Ähnlichkeitsverfahren die Knoten und Kanten des Netzwerks bekannt sein müssen, wird das Clustering ebenfalls vom DV durchgeführt.

Um von den Joins an die Knoten zu gelangen, wird über jeden Join iteriert. Da ein Join aus verknüpften Tabellenpaaren besteht, wird zudem über diese iteriert. Existiert noch kein Knoten mit dem Tabellenname des aktuellen Paares, wird diese in die Liste der Knoten aufgenommen.

Data: J, Join-Liste
Result: N, Knoten-Liste
$N \leftarrow \emptyset$;
forall $j \in J$ **do**
  forall $(p_1, p_2) \in P$ **do**
    if $Node(p_1) \notin N$ **then**
      $N \leftarrow N \cup Node(p_1)$;
    end
    if $Node(p_2) \notin N$ **then**
      $N \leftarrow N \cup Node(p_2)$;
    end
  end
end

Algorithm 1: Knotenerzeugender Algorithmus

Der Algorithmus zum Ermitteln der Kanten funktioniert ähnlich. Befindet sich eine Kante noch nicht in der Liste aller Kanten, wird sie aufgenommen. Der Wert der Kante entspricht der Häufigkeit des Joins aus dem die Paare (Quelle und Ziel) entstammen. Existiert die Kante schon, wird ihr Wert um die Häufigkeit des Joins erhöht. Da es sich bei dem resultierenden Graphen um einen ungerichteten Graphen handelt, spielt es keine Rolle, ob eine Kante von p_1 nach p_2 führt, oder umgekehrt. Dieser Algorithmus dient als Grundlage für alle anderen Verfahren. Um die relative Häufigkeit der Joins zu betrachten, ist dies durch das Teilen des Kantengewichts durch die Menge der Joins möglich. Weiterhin

wird dieser Algorithmus auch für das Berechnen der Join-Stärke genutzt. Der einzige Unterschied besteht darin, dass bei dem Algorithmus zur Berechnung der Join-Stärke, der Schritt, welcher das Kantengewicht einer existierenden Kante erhöht, übersprungen wird.

Data: J, Join-Liste
Result: E, Kanten-Liste
$E \leftarrow \emptyset$;
forall $j \in J$ **do**
  forall $p \in P$ **do**
    ; /* $p = (p_1, p_2)$ */
    if $Edge(p) \notin E$ **then**
      $E \leftarrow E \cup Edge(p)$;
    else
      ; /* Wird bei Berechnung der Join-Stärke übersprungen. */
      $Edge(p) \leftarrow Edge(p) + j.count$;
    end
  end
end

Algorithm 2: Kantenerzeugender Algorithmus

4.6 Refactoring Frontend

Die Slider-Komponente, welche den Threshold anhebt und so Kanten ausblendet, wurde durch eine Reihe von Buttons ersetzt. Umgesetzt wurden die Buttons mit einer Methode `step()`. Als Übergabeparameter erhält sie einen `boolean`, durch welchen ausgedrückt wird, ob Kanten entfernt, oder hinzugefügt werden sollen.

```
step(isForward) {
  const border = isForward
  ? Math.min(...this.links.map((l) => l.name))
  : Math.max(...this.removedLinks.map((l) => l.name));
  this.removeLinks(border, isForward);
}
```

Abbildung 4.12: Methode zum Hinzufügen bzw. Entfernen von Kanten

Als nächstes sollte das Frontend-Framework ausgetauscht werden. Dies hatte mehrere Gründe. Ein Hauptgrund war der fehlende Support für `vue-d3-network`. Der letzte Release fand –zum Stand dieser Arbeit– am 16.02.2019 statt. Demnach gab es Unklarheiten über die Zukunft dieses Frameworks. Weiterhin würde der Beibehalt von `vue-d3-network` dazu führen, auf künftige Version von Vue verzichten zu müssen. Auch die Benutzerfreundlichkeit von `vue-d3-network` ließ zu wünschen übrig. Das Framework unterstützt keine Zoom-Funktion, was bei größeren Netzwerken von Vorteil sein kann. Das Bewegen innerhalb des Viewports wird ebenfalls nicht unterstützt. Demnach wurde der Entschluss gefasst, das Frontend von `vue-d3-network` nach `visJS` zu migrieren. Der Austausch des Frameworks geschieht über das Anpassen der `package.json`-Datei. Die `package.json`-Datei wird zum Verwalten der Abhängigkeiten im Frontend verwendet. Zudem wurde Vue von version `2.6` auf `3.3` aktualisiert. Damit wurde die Vue Option-API abgeschafft und

durch die Composition-API ausgetauscht. In älteren Vue-Versionen wurde Code, welcher logisch zusammengehört getrennt (Vue ([o. D.(a)])). Dies lag an der Option-API, welche vorschrieb die Daten anhand ihrer Art zu separieren. Ein Beispiel dafür waren die Eigenschaften der `vue-d3-network`-Komponente. Wie im Anhang 8.10 deutlich wird, kann es aufgrund der Aufteilung des Codes in größeren Projekten unübersichtlich werden. Mit der Migration von Vue 2 nach Vue 3 wurde ebenfalls Vuetify, ein Vue-Komponenten-Framework, durch Quasar ausgetauscht. Da diese syntaktisch ähnlich sind, gab es bei der Umstellung keine Schwierigkeiten.

Im alten Frontend mussten Knoten, die aus dem Netzwerk entfernt werden sollen, in ein neues Array geschrieben werden, da sie sonst verloren gingen. Mit VisJS kann die `hidden` Eigenschaft einer Kante genutzt werden, um sie auszublenden. Dazu werden alle IDs von Kanten gesucht die auszublenden sind. Anschließend wird eine Update-Funktion, welche vom `DataSet` zur Verfügung gestellt wird, auf den Kanten ausgeführt.

```
const step = (isForward: boolean) => {
  const ids = data.value.edges.getIds({
    filter: function (item) {
      return isForward? ((item.value! <= findSmallestShownEdgeValue())) :
        ((item.value! >= findBiggestHiddenEdgeValue()));}})

  ids.forEach(id => {
    data.value.edges.updateOnly({id: id, hidden: isForward,})})}
```

Abbildung 4.13: Step-Methode in VisJS

Um das Netzwerk anzeigen zu können wurde eine Referenz verwendet. Dadurch kann auf den `div`-Container im Template zugegriffen werden. Die if-Abfrage wird benötigt, da der Zugriff auf die Referenz erst möglich ist, wenn die Komponente den mount-Punkt des Vue-Komponenten-Lifecycles erreicht hat (Vue ([o. D.(b)])). Anhang 8.11 zeigt den Aufbau der Vue-Komponente die das Netzwerk rendert.

4.7 Erweiterung des Backends

Das Backend wurde mit den in Kapitel 3 vorgestellten Verfahren erweitert. Die Algorithmen bzw. die Implementierungen sollen in diesem Abschnitt behandelt werden.

4.7.1 Implementierung der Häufigkeit von Joins

Unter betracht des Join-Kontexts wird eine Methode erstellt, welche zwei Join-Ketten auf Gleichheit prüft. Dazu wurde eine Methode `compareJoins()` eingeführt. Diese überprüft zuerst ob zwei Joins die gleiche Länge haben. Falls nicht können die Joins nicht gleich sein. Haben die Joins die gleiche Länge, so werden sie anhand der Paare verglichen. Die Paare werden alphabetisch sortiert. Der Grund dafür ist, dass die $Richtung$, in welche die Tabellen verknüpft werden, vernachlässigt wird.

```
fun compareJoins(a: List<Pair<String, String>>, b: List<Pair<String, String>>):
        Boolean { if (!compareJoinLength(a, b)) return false

    return (a.sortedWith(compareBy(Pair<String, String>::first)
        .thenBy(Pair<String, String>::second))
        .map(::sortPairAlphabetically).toSet() == b.sortedWith(
        compareBy(Pair<String, String>::first).thenBy(
            Pair<String, String>::second
        )
    ).map(::sortPairAlphabetically).toSet())
}
```

Abbildung 4.14: Methode zum Vergleichen zweier Joins

Die Methode, welche die Häufigkeit für einen Join j berechnet, iteriert über die Liste aller Joins. In jeder Iteration, wird die Liste nach Joins gefiltert die gleich j sind. Die Gleichheit wird durch die Methode in Abbildung 4.14 bestimmt. Die `count`-Variable von j wird um die Anzahl an gefilterten Listeneinträgen erhöht.

Die Implementierung des Sub-Join-Verfahrens gelingt durch das Erweitern dieser Methode. Statt nur nach Joins zu filtern, welche gleich j sind, wird die `count`-Variable von j zudem noch um die Anzahl der Sub-Joins erhöht.

Data: J, Join-Liste
$J' \leftarrow \emptyset$;
forall $j \in J$ **do**
  $J' \leftarrow J' \cup j' \leftrightarrow j' = j$;
  $j.count \leftarrow j.count + \|J'\|$;
  if *Sub-Join-Verfahren* **then**
    $J' \leftarrow J' \cup j'' \leftrightarrow j'' \sim j$;
    $j.count \leftarrow j.count + \|J''\|$;
  end
end

Algorithm 3: Algorithmus zum Berechnen der Häufigkeit eines Joins

4.7.2 Implementierung Join-Stärke

Das Verfahren der Join-Stärke zeichnet sich durch die Reduzierung der Datenmenge aus. Die Joins, welche aus der Datenaufbereitung resultieren, werden auf die individuellen Ausprägungen reduziert. Implementierungstechnisch geschieht dies durch einen Aufruf der `distinct`-Methode auf der Liste der Joins. Um diese in der Visualisierung unterscheiden zu können wird eine Methode verwendet, die jedem Join eine zufällige Farbe zuweist. Dadurch wird in der Visualisierung ersichtlich, wie oft die Tabellen der Anwendung in individuellen Joins vorkommen.

4.7.3 Implementierung des Nachbarschaftsverfahren

Das Nachbarschaftsverfahren verläuft analog zum Verfahren der relativen Häufigkeit, wie in 4.10 beschrieben. Der Unterschied findet sich in der Berechnung der Kantengewichte wieder. Anfangs wird der Wert im `dict` (Hashmap-Datenstruktur, beschrieben in 4.4), der zu dem Tabellenpaar passt, auf die Summe der direkten und indirekten Verbindungen

gesetzt. Für das Nachbarschaftsverfahren müssen bereits Knoten und Kanten existieren, um die direkte Nachbarschaft zu identifizieren. Anschließend müssen alle benachbarten Kanten ausfindig gemacht werden. Die Summe aller Nachbarkanten wird durch das gewicht der aktuellen Kante geteilt und ergibt somit das neue Kantengewicht.

Data: E, Kanten-Liste
$N \leftarrow \emptyset$;
forall $e \in E$ **do**
  $N \leftarrow N \cup neighboors(e)$;
  $e.value \leftarrow \frac{e.value}{\sum^{N} n.value}$;
end

Algorithm 4: Algorithmus zum Berechnen der Kantengewichte im Nachbarschaftsverfahren

Um alle benachbarten Kanten zu erhalten, wird eine Methode `getAllLinks(nodeName: String, links: List<Link>)` verwendet. Diese filtert die eingegebene Liste nach allen Kanten, die zu oder von dem mitgegebenen Knoten verlaufen. Da eine Kante zwei Knoten verknüpft wird diese Methode zweimal aufgerufen.

```
fun getAllNeighbours(link: Link, links: List<Link>): List<Link> =
    (getAllLinks(link.from, links) + getAllLinks(link.to, links))
        .filter { it != link }
```

Abbildung 4.15: Methode zum Finden aller benachbarten Kanten

4.7.4 Implementierung LCOM

Bei der LCOM-Metrik werden die Kardinalitäten der Mengen P und Q berechnet und deren Differenz wird als Wert für den Join übernommen. Um P und Q zu erhalten, wird über jeden Join iteriert. Dabei wird LCOM für jeden Join in Kombination mit allen anderen Joins berechnet.

Data: J, Join-Liste
forall $j \in J$ **do**
  forall $j' \in J \setminus j$ **do**
    $(P, Q) = LCOM(j, j')$;
    $j.value \leftarrow P - Q$;
  end
end

Algorithm 5: Algorithmus zum Berechnen von LCOM

Für die Berechnung von LCOM werden alle Tabellenpaare von zwei Join-Ketten verglichen. Um Q, die Menge der gemeinsamen Tabellenpaare zu erhalten, werden die verknüpften Tabellen der Joins zuerst mit `set()` in Mengen konvertiert. Die Schnittmenge der beiden ist mit `intersect()` zu erhalten. P ist die Menge der sich unterscheidenden Tabellen. Dafür werden die Mengen zuerst vereint. Nach der Vereinigung wird die Schnittmenge entfernt. Dies resultiert in einer Menge, welche nur die nicht gemeinsamen Tabellen enthält.

```
fun getPAndQ(j: Join, jp: Join): Pair<Int, Int> {
    val q = (j.pairs.toSet().intersect(jp.pairs.toSet())).size
    val p = (j.pairs.toSet().union(jp.pairs.toSet())).size - q
    return p to q
}
```

Abbildung 4.16: Methode zum Berechnen von LCOM

4.7.5 Implementierung Ähnlichkeitsverfahren

Um das Ähnlichkeitsverfahren anwenden zu können, müssen wie bei dem Nachbarschaftsverfahren, die Knoten und Kanten des Graphen identifiziert sein. Der Algorithmus ist in zwei Hälften aufgeteilt. Im ersten Teil berechnet der Algorithmus die Ähnlichkeit von einem Knoten zu allen anderen Knoten.

```
private fun calculateSimilarityBetweenN1andAllOtherNodes(/*...*/): {
    val map = mutableMapOf<String, MutableList<Pair<String, Double>>>()
    var allN2s = mutableListOf<Pair<String, Double>>()
    nodes.forEach { n1 ->
        nodes.forEach { n2 ->
            if (n1 != n2) allN2s.add(
                getJaccardOrOverlap(n1, n2, edges, isJaccard))
        }
        map[n1.id] = allN2s
        allN2s = mutableListOf()
    }
    return map
}
```

Abbildung 4.17: Teil 1 des Cluster-Algorithmus

Um die Ergebnisse festzuhalten wird eine Hashmap (`map`) verwendet. Jeder Eintrag in der Hashmap besteht aus einem Schlüssel, welcher aus dem Knotennamen besteht. Der dazugehörige Wert ist eine Liste bestehend aus Paaren. Ein Paar besteht aus einem Knotennamen und einem Ähnlichkeitswert. Die Berechnung der Ähnlichkeit findet in der `getJaccardOrOverlap`-Methode statt. In der Methode wird ein Filter verwendet um alle Kanten zu erhalten, in denen n_1 als Start oder Ziel vorkommt. Mit einem Aufruf von `flatmap` kann auf die einzelnen Elemente einer Kante (`from` und `to`) zugegriffen werden. Diese werden in einer Menge gespeichert. Aus der Menge werden die Knoten n_1 und n_2 entfernt, sodass nur die Nachbarn von n_1 übrig bleiben. Analog geschieht dies für n_2. Abschließend werden die Mengen wie in 3.4.3 beschrieben, zur Evaluierung der Ähnlichkeit genutzt.

```
fun getJaccardOrOverlap(/*...*/): Pair<String, Double> {
    val n1Set = edges.filter { edge -> (edge.from == n1.id ||
            edge.to == n1.id) }.flatMap {
        setOf(it.from, it.to).minus(n1.id).minus(n2.id)
    }.toSet()
    val n2Set = edges.filter { edge -> (edge.from == n2.id
            || edge.to == n2.id) }.flatMap {
        setOf(it.from, it.to).minus(n2.id).minus(n1.id)
    }.toSet()
    return if (isJaccard) n2.id to (((n1Set.intersect(n2Set)).size.toDouble() /
            (n1Set.union(n2Set)).size.toDouble()))
    else n2.id to (((n1Set.intersect(n2Set)).size.toDouble() /
            (minOf(n1Set.size, n2Set.size).toDouble())))
}
```

Abbildung 4.18: Methode zum Berechnen Ähnlichkeit zwischen zwei Knoten

Der zweite Teil des Algorithmus ist für das Clustern der Knoten zuständig. Um zu entscheiden, wann zwei Knoten n_1 und n_2 ähnlich genug sind, um sie in einem Cluster zu gruppieren, wird die durchschnittliche Ähnlichkeit der Knoten berechnet. Liegt die Ähnlichkeit eines Knotens n_2 unterhalb des Durchschnitts, so kann sich dieser nicht im selben Cluster, wie n_1 befinden.

```
map.forEach { nodeToNodesSim ->
    val mapEntryAverageSim = getMapEntryAverageSim(nodeToNodesSim)
    val tmpClusterId: Int = when {
        nodes.find { node -> node.id == nodeToNodesSim.key }?.clusterId != 0
        -> nodes.find { node -> node.id == nodeToNodesSim.key }?.clusterId!!
        else -> globalClusterId++
    }
    nodes.find { node ->
        node.id == nodeToNodesSim.key }?.clusterId = tmpClusterId
    val tmpClusterNodes = getSimilarNodes(
        nodeToNodesSim,
        useStd,
        mapEntryAverageSim,
        getMapEntryAverageSimStd(nodeToNodesSim,
            mapEntryAverageSim))
    tmpClusterNodes.forEach { simNode ->
        elements.nodes.find { en ->
            en.id == simNode
        }?.clusterId = tmpClusterId
    }
}
```

Abbildung 4.19: Teil 2 des Cluster-Algorithmus

Um die Cluster-Id zu bestimmen wird geprüft, ob sich der Knoten n_1 bereits in einem

Cluster befindet. Wenn nicht, wird die globale Cluster-Id inkrementiert. Alle Knoten die mindestens durchschnittlich ähnlich zu n_1 sind erhalten die gleiche Cluster-Id.

4.8 Join Aufbereitung

Der Join Prepper (JP) ist ein Werkzeug welches zwischen der Datenaufbereitung und der Datenanalyse zum Einsatz kommt. Der DP bringt die Daten in ein Format aus denen Informationen über die Joins gewonnen werden können. Der DA führt die Analyse durch, welche den Joins ihre Wertigkeit, je nach Verfahren gibt. Mit diesem Zwischenschritt sollen die Daten vor der Analyse weiter aufbereitet werden. Er dient dazu dem Benutzer mehr Kontrolle über die auszuwertenden Daten zu geben. Dazu zählt das Ermitteln und Entfernen des Max-Joins sowie Join-Ketten, welche überdurchschnittlich lang sind oder das Angeben eines vermeintlichen Fuzzy Joins. Der Benutzer kann beim Aufruf des JP einen Fuzzy-Join in Form eines Strings übergeben. Dieser String wird zu einem `Pair<String, String>` konvertiert, indem die Eingabe mit einem Delimiter aufgeteilt wird. Der standard Delimiter nach welchem die Eingabe durchsucht wird, ist ein Komma (,). Da es verschiedene Separatoren geben kann, sucht eine Methode nach dafür infrage kommende Symbole.

```
tables.split(delimiter ?: ",").toTypedArray().let
{ it[0].trim() to it[1].trim() }
```

Abbildung 4.20: Methode zur Aufbereitung der Fuzzy-Joins

4.9 Erweiterung des Frontends

Um die Implementierung abzuschließen muss das Frontend erweitert werden. Da alle Knoten über eine Cluster-Id verfügen ist es möglich, diese anhand der Id zu gruppieren. Dazu werden alle verschiedenen Cluster-Ids identifiziert. VisJs ermöglicht das Clustern der Knoten durch eine Funktion `cluster(options?: ClusterOptions)`, welche auf dem Netzwerk Objekt aufgerufen wird. Die `joinCondition` wird auf alle Knoten des Netzwerks angewandt. In dieser wird über die Knoten iteriert und geprüft, ob die Cluster-Id übereinstimmt. An dieser Stelle wird die Typsicherheit von TypeScript ignoriert. Der `Node`-Datentyp, welcher von VisJS implementiert wurde, verfügt aufgrund von unvollständiger Type-Definitions über keine Cluster-Id.

```
// ...
clusterIds.forEach(cid => {
  var clusterOptionsByClusterId = {
    joinCondition: function (childOptions) {
      return childOptions.clusterId == cid;
    },
  };
  network.value?.cluster(clusterOptionsByClusterId);
})
```

Abbildung 4.21: Clustering der Knoten im Frontend

4.10 Bedienung der Anwendung

Die Anwendung liegt in Form einer `JAR`-Datei vor. Das Aufrufen der einzelnen Programme geschieht über den Klassenpfad. Dazu kann der Befehl `java -cp` genutzt werden, welcher nach der Main-Methode der aufgerufenen Klasse im angegebenen Pfad sucht. In jedem Schritt der Auswertung werden Eingabeparameter verlangt. Der DA verlangt beispielsweise nach dem Pfad der Eingabedatei sowie dem gewählten Verfahren.

```
fun getJoinContextCalculationMethod(joins: MutableList<Join>, method: String):
        List<Join> {
    return when (method) {
        // ...
        "lcom" -> calcLCOM2(joins)
        "join-strength" -> calculateDistinctJoinStrength(joins)
        // ...
        else -> calculateJoinOccurrence(joins, "occurrence")
    }
}
```

Abbildung 4.22: Vereinfachte Darstellung der Methode zur Wahl eines Verfahrens im DA

Dieser Aufruf kann umständlich sein, besonders wenn Eingabeparameter verlangt werden. Zudem liegt keine Beschreibung vor, in welcher Reihenfolge die Parameter anzugeben sind. Dafür wurden Bash-Skripte entwickelt, welche die Anwendung vereinfachen. Für jeden Teilprozess der Auswertung existiert ein Skript. Dadurch ist eine Einsicht in die erzeugten Ergebnisse jeden Schritts der Auswertung möglich. Jedes Skript ist mit einer Hilfsfunktion ausgestattet welche die Eingabeparameter erläutert. Die Eingabeparameter werden mit `getops` ermittelt. Bei der Fallunterscheidung werden die verschiedenen Parameter Variablen zugewiesen. Anschließend wird der Java-Befehl ausgeführt, welcher die Auswertung vornimmt.

```
#!/bin/bash
BASIC_CMD="java -cp monolith-killer-2.jar path.to.main.class"
while getopts c:m:f:h flag
do
    case "${flag}" in
        c) context=${OPTARG};;
        m) verfahren=${OPTARG};;
        f) file=${OPTARG};;
        h)
            Help
            exit 1
            ;;
    esac
done

$BASIC_CMD $file $context $verfahren
```

Abbildung 4.23: Vereinfachte Darstellung eines Skripts zur Ausführung des DA

5 Evaluation

In diesem Kapitel werden von den verschiedenen Verfahren erzeugten Ergebnisse verglichen. Zudem werden Ansätze für die potenzielle Verbesserung einiger Verfahren vorgestellt.

5.1 Profil des Datensatzes

Die Informationen wurden durch eine selbstgeschriebene Klasse (`JoinPrepper`) erhalten. Bei dem Datensatz handelt es sich erneut um einen firmeninternen Datensatz, welcher durch lokal generierte Daten angereichert wurde. Der Datensatz umfasst 18780 Join-Statements. Es existieren 38 unterschiedliche Joins. Der längste Join besteht aus 152 Tabellenpaaren. Einige Abfragen sind höchstwahrscheinlich aufgrund eines Fehlers des Schreibvorgangs unterbrochen worden. Dadurch wurden diese nicht mit einem Delimiter versehen. Zum Zweck der Analyse soll sie jedoch beibehalten werden, um festzustellen, ob diese Abfrage Einfluss auf den Schnitt hat. In einer weiteren Analyse wird dieser Join entfernt (Max-Join). Die zugrundeliegende Datenbank der Anwendung umfasst 22 Tabellen. Die durchschnittliche Länge eines Joins beträgt 10.59 Tabellenpaare, mit einer Standardabweichung von 9.41.

5.2 Vergleich der Ergebnisse

In diesem Abschnitt werden alle Ausprägungen der implementierten Verfahren verglichen, anhand der in Abschnitt 5.1 beschriebenen Daten.

5.2.1 Nachbarschaftsverfahren

Das Nachbarschaftsverfahren war das erste neu vorgestellte Verfahren. Der im Abschnitt 3.4.1 erdachte Worst-Case tritt nicht direkt ein. Der Vermutung entsprechend müsste der Graph im initialen Schritt darauf hinweisen, dass die Verbindung zwischen den To-Do-Tabellen und der `Veranstaltung`-Tabelle schwach ist. Die Begründung dafür ist, dass die `Veranstaltung`-Tabelle im Zentrum der Anwendung steht. Demnach müsste der Wert der Kante zwischen `veranstaltungtodo` und `Veranstaltung` gering sein, da die `veranstaltungtodo`-Tabelle sonst nur mit To-Do-bezogenen Tabellen kommuniziert. Die `veranstaltungtodo`-Tabelle beschreibt die To-Do-Items, welche in der Veranstaltung abgespeichert werden. Daher wird sie bei jedem Anlegen einer neuen Veranstaltung benutzt, sowie wenn ein Eintrag abgehakt wird. Die `todocategory`-Tabelle liefert Templates für To-Do-Listeneinträge. Deren vergleichsweise schwache Beziehung lässt sich dadurch erklären, dass To-Do-Templates selten angelegt werden.

Weiteres Voranschreiten macht klar, dass Nachbarschaftsverfahren implementiert wie in 3.4.1 beschrieben, sorgt dafür, dass sich der Kern der Anwendung rauskristallisiert. Die sich im Zentrum befindenden Tabellen sind stark verknüpft und vermascht. Die drumherum befestigten Tabellenzweige haben durch die Relevanz der Häufigkeit einen zu schwachen, internen zusammenhalt.

5.2.2 LCOM

Der Zusammenhalt, der vom Kern der Anwendung abzweigenden Tabellen, ist gering. Dies lässt sich durch die Tatsache erklären, dass bei der Berechnung von LCOM über alle Joins iteriert wird. Demnach fließt die Häufigkeit mit in die Bewertung der Kanten ein. Weiterhin ist Q für die Tabellen in den Abzweigungen des Kerns gering, da sie mit wenigen Tabellen verknüpft sind. Dennoch ist der Zusammenhalt intern oft stärker als die Kanten, die zum Kern der Anwendung führen. Im Gegensatz zum Nachbarschaftsverfahren wird ein Schnitt entlang der To-Do-Tabellen gesetzt. Auch die Session-Tabelle spalten sich

vom Kern ab. Im Grunde unterscheiden sich Sessions und To-dos nicht. In beiden Fällen handelt es sich um Tabellen, die nur untereinander und mit der Veranstaltungstabelle kommunizieren. Die Sessions in einen eigenen Service auszulagern kann zu feingranular sein. Das Abtrennen der Veranstaltungen würde aufgrund der Konnektivität zu vielen Schnittstellen führen.

Ein möglicher Schnitt ist das Aufteilen der Anwendung in einen To-Do-Service und einen Veranstaltungsservice, welcher weiterhin grobe monolithische Züge aufweist.

5.2.2.1 LCOM ohne Max-Join

Durch das Entfernen von längeren Join-Ketten verändert sich der Schnitt. Der erste Schnitt wird erneut bei den To-Do-Tabellen gesetzt. Die Gewichtung der Kanten hat sich im Vergleich zu den Daten, welche längere Join-Ketten enthalten nicht geändert. Die Berechnung von LCOM für zwei Joins basiert auf den Kardinalitäten der Mengen P und Q. Das Entfernen von Reporting-Abfragen hat keinen Einfluss auf die vom Kern der Anwendung abzweigenden To-Do-Tabellen. P und Q bleiben unverändert.

Weiterhin lösen sich die Mail-bezogenen Tabellen vom Kern der Anwendung ab. Reporting-Abfragen werden in der Firmenanwendung unter anderem durch das Mail-Protokoll erzeugt.

Ein möglicher Schnitt ist das Aufteilen der Anwendung in einen To-Do-Service, einen Mail-Service und einen Veranstaltungsservice. Das Entfernen des Max-Joins, sowie das Entfernen von überdurchschnittlich langen Joins führt dazu, das die Anwendung im Kern weniger stark zusammenhängt.

5.2.3 Join-Kontext: relative Häufigkeit von Joins

Das auf Häufigkeiten basierende Verfahren dazu führen, dass Tabellen welche am wenigsten verwendetet werden, zuerst abgetrennt werden, ist eine Beobachtung welche im Abschnitt 3.1.4.1 beschrieben wurde und sich bei der Auswertung der Daten erneut bewahrheitet. Die Verbindung zwischen den Tabellen `veranstaltungparticipation` und `veranstaltung` wurde mit 0.031 gewichtet. Im Vergleich dazu hängt der To-Do-Tabellen-Ast mit einer Gewichtung von 0.14 an der Veranstaltung. Die `veranstaltungparticipation`-Tabelle wird dann benutzt, wenn man eine Kursseite besucht. Auf einer Kursseite befindet sich neben einer Tabelle für To-Do-Einträge eine Tabelle für Teilnehmer. Die Verwaltung kann die Teilnehmer-Tabelle verwenden, um E-Mails zu oder Zertifikate an Teilnehmer zu verschicken. Die Einträge der To-Do-Tabelle werden für gewöhnlich abgehakt. Eine plausible Erklärung ist, dass während der Erfassung der Log-Dateien, die Teilnehmer weniger oft verwaltet wurden.

Das Amplifizieren der Join-Ketten mit Sub-Joins hat nicht den erhofften Effekt. Die Kante welche den Zweig bestehend aus To-Do-Tabellen an die Veranstaltung knüpft, sollte stärker sein und so für einen gröberen Schnitt sorgen. Der Wert der Kante wurde geringer. Dieser Effekt ist auch bei der Session-Tabelle zu betrachten. Eine mögliche Erklärung dafür ist, dass Tabellen die als Äste vom Kern der Anwendung abzweigen, isoliert verwendet werden. Dies wird ist daran zu erkennen, dass diese Äste keine weiteren Verbindungen zum Rest der Anwendung besitzen. Demnach gibt es keine Sub-Joins, welche das Vorkommen amplifizieren können.

5.2.3.1 Join-Kontext: Relative Häufigkeit ohne Max-Join

Die Aussage aus Abschnitt 5.2.2.1, dass durch das Entfernen von längeren Join-Ketten, die Anwendung im Kern weniger fest zusammenhält, bestätigt sich erneut. Wieder trennen sich die Mail-bezogenen Tabellen vom Kern der Anwendung ab. Der interne Zusammenhalt der To-Do-Tabellen bleibt bestehen.

5.2.4 Ähnlichkeitsverfahren

Einer der Hauptunterschiede der beiden Metriken zum Clustern der Daten liegt bei der Granularität. Der Jaccard-Index gruppiert die Tabellen fein granularer als der Overlap-Koeffizient. Dieser sieht vor die Funktionalitäten der Veranstaltung (To-Do-Liste und Sessions) in einen Service auszulagern. Der Grund für das Zusammenlegen ist die Ähnlichkeit der `veranstaltungsession`- und `veranstaltungtodo`-Tabelle. Diese zweigen vom gleichen Knoten des Kerns der Anwendung ab. Für `veranstaltungsession` ergibt sich eine Menge A, an benachbarten Knoten, mit $A = \{\texttt{veranstaltung}\}$. Für `veranstaltungtodo` ergibt sich eine Menge $B = \{\texttt{veranstaltung}, \texttt{todocategory}\}$. Der daraus resultierende ein Jaccard-Index ist:

$$
\begin{aligned}
&\frac{|A \cap B|}{|A \cup B|} \\
=&\frac{|\{\texttt{veranstaltung}\} \cap \{\texttt{veranstaltung}, \texttt{todocategory}\}|}{|\{\texttt{veranstaltung}\} \cup \{\texttt{veranstaltung}, \texttt{todocategory}\}|} \\
=&\frac{|\{\texttt{veranstaltung}\}|}{|\{\texttt{veranstaltung}, \texttt{todocategory}\}|} \\
=&\frac{1}{2} \\
=&0.5
\end{aligned}
\tag{5.1}
$$

Der vom Overlap-Koeffizient vorgeschlagene Schnitt entspricht etwa dem, was aus den vorherigen Verfahren hervorging. Dieser sieht vor einen To-Do-Service anzulegen. Die `veranstaltungsession`-Tabelle wird in den Kern der Anwendung aufgenommen. Der Overlap-Koeffizient mit den Gleichen Mengen A und B entspricht:

$$
\begin{aligned}
&\frac{|A \cap B|}{\min A, B} \\
=&\frac{|\{\texttt{veranstaltung}\} \cap \{\texttt{veranstaltung}, \texttt{todocategory}\}|}{\min \{\texttt{veranstaltung}\}, \{\texttt{veranstaltung}, \texttt{todocategory}\}} \\
=&\frac{|\{\texttt{veranstaltung}\}|}{|\{\texttt{veranstaltung}\}|} \\
=&\frac{1}{1} \\
=&1
\end{aligned}
\tag{5.2}
$$

Dies ist möglicherweise darauf zurückzuführen, dass `veranstaltungtodo` noch weitere Verbindungen zu den Todo-spezifischen Tabellen hat. Deren Ähnlichkeiten werden jeweils mit 1 bewertet. Die Ähnlichkeit zum Kern der Anwendung ist im Gegensatz dazu 0.5. Die Gruppierung ist nicht perfekt, da die `todocategory`-Tabelle außerhalb des Clusters liegt. Dies ist anhand der Berechnug des Overlap-Koefizientens zu erklären. Die `todocategory`-Tabelle verfügt über drei Nachbarn, `veranstaltungtodo`,

todocategory_durchfuehrungen und todocategory_arten. Bei der Berechnung des Overlap-Koeffizientens mit der Nachbartabelle todocategory_durchfuehrungen ergibt sich eine leere Menge, denn:

$$\begin{aligned}
&\frac{|A \cap B|}{\min A, B} \\
=&\frac{|\{\texttt{verabstaltungtodo}, \texttt{todocategory_arten}, \texttt{todocategory_durchfuehrungen}\} \cap \emptyset|}{\min \{\texttt{verabstaltungtodo}, \texttt{todocategory_arten}, \texttt{todocategory_durchfuehrungen}\}, \emptyset} \\
=&\frac{|\emptyset|}{|\emptyset|} \\
=&\frac{0}{0} \\
=&0
\end{aligned} \tag{5.3}$$

Analog gilt dies für todocategory und todocategory_arten. Gleiches gilt für das Auslagern der Mails in einen eigenen Service. Die mailprotocol-Tabelle wurde nicht in das Cluster aufgenommen.

Das Einbeziehen der Standardabweichung bei der Bestimmung des Mindestähnlichkeitsmaßes führt zu einer feineren Aufteilung der Services. Dies trifft sowohl auf die Berechnung der Ähnlichkeit basierend auf dem Jaccard-Index, als auch auf der Berechnung basierend auf dem Overlap-Koeffizienten zu. Das strengere Ähnlichkeitsmaß sieht in beiden Fällen vor, den Kern der Anwendung ebenfalls aufzuspalten. Ohne das Hinzuziehen der Standardabweichung bleibt der Kern der Anwendung erhalten.

5.3 Verfeinerung: LCOM

Die LCOM-Metrik erzeugt Werte die nicht genormt sind. Ein Versuch die Metrik zu normaliseren soll über das Umstellen der Formel geschehen. Das Verhältnis von P und Q soll so umgestellt werden, dass die Anzahl der gemeinsamen Tabellen im Verhältnis zu der Anzahl an Tabellen steht, die nicht geteilt werden. Dazu wird P -Q umgestellt nach $\frac{P}{Q}$. Gibt es mehr gemeinsame Tabellen statt unterschiedliche, so ist Q größer als P. Der Bruch $\frac{P}{Q}$ konvergiert gegen 0. Mit $1 - \frac{P}{Q}$ konvergiert LCOM gegen 1. Damit wird ein großes Q belohnt, indem der Wert näher bei 1 liegt. Weiterhin kann die Metrik verfeinert werden, indem nur die individuellen Joins in Betracht gezogen werden. Dadurch fällt die Häufigkeit der Joins bei der Berechnung weg.

5.4 Fazit

Das Einbeziehen der Häufigkeit bei der Auswertung der Daten ist ein zweischneidiges Schwert. Einerseits wird dadurch der Zusammenhalt von den am häufigsten verwendeten Tabellen verstärkt. Dies kann dazu führen, dass die vom Kern der Anwendung abzweigenden Tabellen unterhalb zu schwach miteinander verknüpft sind und auseinander fallen. Dies resultiert in unverbundenen Knoten, welche um den Kern der Anwendung verteilt sind. Andererseits hilft die Häufigkeit dabei erkennbar zu machen, wie stark die Kommunikation von Tabellen ist. Ein Schnitt zwischen einer starken Kommunikation führt zu neuen Schnittstellen und Netzwerkverkehr. Weiterhin hilft die Häufigkeit dabei viel verwendete Teile einer Anwendung zu identifizieren. Vernachlässigt man die Häufigkeit der Nutzung von Tabellen und lagert einen selten verwendeten Teil des Systems aus, entsteht dadurch ein Microservice, der ebenfalls selten verwendet wird. Demnach entstehen die bestmöglichen Resultate durch die Benutzung mehrerer Verfahren. Durch das Vergleichen der Ergebnisse kann die Entscheidung für den besten Schnitt gefällt werden.

6 Zusammenfassung

Das Ermitteln von Domain-Grenzen einer monolithischen Anwendung ist keine Trivialität. Das (weiter)-entwickelte Werkzeug ist in der Lage für einen Monolithen eine Aufteilung der Domäne zu empfehlen. Weiterhin wird für die Interpretation des erzeugten Schnitts Wissen über die Domäne benötigt. Andernfalls kann es schwerfallen, die Qualität des Schnitts zu bewerten. Die Anwendung ist zum jetzigen Zeitpunkt mit mehreren Verfahren ausgestattet, welche bei dem Finden von Domain-Grenzen Unterstützung bieten. Die predic8 GmbH nutzt das Werkzeug um in wenigen Minuten Informationen über firmeninterne Anwendungen zu gewinnen. In kurzer Zeit ist zu erkennen wie verwachsen die Beziehungen der Geschäftsobjekte eines Monolithen sind. Zudem wird ersichtlich, ob dieser nur eine oder mehrere Fachdomänen abdeckt. Daraus lässt sich ableiten, ob die vorliegende Anwendung refaktoriert, in Microservices zerlegt oder neu-implementiert werden sollte. Im Falle einer Zerlegung lässt sich die grobe Anzahl der Microservices, per Knopfdruck erkennen.

6.1 Offene Punkte

Die Auswirkung des Entfernens des Fuzzy-Joins konnten nicht getestet werden, da die Firmenanwendung überarbeitet wurde und über keine Benutzertabelle mehr verfügt. Aufgrund dessen bleibt es fraglich, welche Auswirkungen das Entfernen des Fuzzy Joins hat.

6.2 Ausblick

Das Erzeugen von Dateien, angefangen bei der Aufbereitung zur Analyse, bis hin zur Visualisierung, kann unübersichtlich werden. Besonders deutlich wird dies bei dem Versuch zwei Auswertungen zu vergleichen. Zudem kann die Bedienung der Software kompliziert sein, da dies über die Konsole geschieht. Die Umwandlung der Anwendung in eine Spring Boot Applikation ist eine Überlegung wert. Die Aufbereitung und Verarbeitung der Daten kann so vor dem Benutzer verbogen werden. Das Hochladen der SQL-Logdatei führt zu einer direkten Auswertung in der Spring Boot Applikation, welche die verschiedenen Verfahren über Endpunkte abrufbar macht.

Eine Möglichkeit um die Anwendung zu verbessern, ohne selbst an neue Daten gelangen zu müssen, bietet sich durch eine Veröffentlichung der Software. Unter einer Open-Source-Lizenz können Freiwillige das Werkzeug nutzen und die geplante Zerlegung selber durchführen. Als ein durch die Open-Source-Community vorangetriebenes Projekt, können Erfahrungen gesammelt werden, um die Anwendung zu verbessern und zu erweitern. Dadurch fällt die Notwendigkeit weg, selbst an Datenbank-Logdateien gelangen zu müssen.

7 Literaturverzeichnis

AIVOSTO, [o. D.]. *Cohesion metrics*. Auch verfügbar unter: https://www.aivosto.com/project/help/pm-oo-cohesion.html#LCOM1. [Online; Stand: 26. Juli 2023].

ALMIR VUK, Peter Geelen, 2018. Einfache Abfragen mit Selektion und Projektion (SQL-DML/DQL). Auch verfügbar unter: https://social.technet.microsoft.com/wiki/contents/articles/34477.sql-server-commands-dml-ddl-dcl-tcl.aspx. [Online; Stand: 28. Juli 2023].

AMAZON, [o. D.(a)]. *Break a Monolithic Application into Microservices with AWS Copilot, Amazon ECS, Docker, and AWS Fargate*. Auch verfügbar unter: https://aws.amazon.com/de/tutorials/break-monolith-app-microservices-ecs-docker-ec2/. [Online; Stand: 25. Juli 2023].

AMAZON, [o. D.(b)]. *Was ist serviceorientierte Architektur?* Auch verfügbar unter: https://aws.amazon.com/de/what-is/service-oriented-architecture/. [Online; Stand: 25. Juli 2023].

DEHGHANI, Zhamak, 2018. *How to break a Monolith into Microservices*. Auch verfügbar unter: https://martinfowler.com/articles/break-monolith-into-microservices.html. [Online; Stand: 25. Juli 2023].

GOOGLE, 2023. *Google Trends*. Auch verfügbar unter: https://trends.google.de/trends/explore?date=all&q=Service%20Oriented%20Architecture,Microservices&hl=de. [Online; Stand: 25. Juli 2023].

GOOGLE, [o. D.]. *Monolithische Anwendung in Mikrodienste refaktorieren*. Auch verfügbar unter: https://cloud.google.com/architecture/microservices-architecture-refactoring-monoliths?hl=de. [Online; Stand: 25. Juli 2023].

GRZYBEK, Kamil, 2019. Modular Monolith: A Primer. Auch verfügbar unter: https://www.kamilgrzybek.com/blog/posts/modular-monolith-primer. [Online; Stand: 25. Juli 2023].

IBRYAM, Bilgin, 2021. Distributed transaction patterns for microservices compared. Auch verfügbar unter: https://developers.redhat.com/articles/2021/09/21/distributed-transaction-patterns-microservices-compared. [Online; Stand: 25. Juli 2023].

JAMES LEWIS, Martin Fowler, 2014. Microservices. Auch verfügbar unter: https://www.martinfowler.com/articles/microservices.html. [Online; Stand: 25. Juli 2023].

JETBRAINS, 2021. *The State of Developer Ecosystem 2021*. Auch verfügbar unter: https://www.jetbrains.com/lp/devecosystem-2021/microservices/. [Online; Stand: 25. Juli 2023].

JETBRAINS, 2022. *The State of Developer Ecosystem 2022*. Auch verfügbar unter: https://www.jetbrains.com/lp/devecosystem-2022/microservices/. [Online; Stand: 25. Juli 2023].

KLEINE, Matthias, 2013. Kopplung und Kohäsion - LCOM-Metriken. Auch verfügbar unter: https://prinzipien-der-softwaretechnik.blogspot.com/2013/02/kopplung-und-kohasion-lcom-metriken.html. [Online; Stand: 25. Juli 2023].

KOLNY, Marcin, 2023. Scaling up the Prime Video audio and video monitoring service and reducing costs by 90 percent. Auch verfügbar unter: https://www.primevideotech.com/video-streaming/scaling-up-the-prime-video-audio-video-monitoring-service-and-reducing-costs-by-90. [Online; Stand: 25. Juli 2023].

KOTLIN, [o. D.]. *Data classes*. Auch verfügbar unter: https://kotlinlang.org/docs/data-classes.html. [Online; Stand: 25. Juli 2023].

MARTIN L. ABBOTT, Michael T. Fisher, 2015. *The Art of Scalability: Scalable Web Architecture, Processes, and Organizations for the Modern Enterprise*. Addison-Wesley Professional. ISBN 978-0-13-403280-1.

MICROSOFT, 2022. *Design a DDD-oriented microservice*. Auch verfügbar unter: https://learn.microsoft.com/en-us/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/ddd-oriented-microservice. [Online; Stand: 25. Juli 2023].

MICROSOFT, [o. D.(a)]. *Saga distributed transactions pattern*. Auch verfügbar unter: https://learn.microsoft.com/en-us/azure/architecture/reference-architectures/saga/saga. [Online; Stand: 25. Juli 2023].

MICROSOFT, [o. D.(b)]. *What is DevOps?* Auch verfügbar unter: https://learn.microsoft.com/en-us/devops/what-is-devops. [Online; Stand: 25. Juli 2023].

MYSQLTUTORIAL, [o. D.]. *MySQL Join*. Auch verfügbar unter: https://www.mysqltutorial.org/mysql-join/. [Online; Stand: 25. Juli 2023].

NEWMAN, Sam, 2015. *Building Microservices*. O'Riley.

NEWMAN, Sam, 2019. *Monolith to Microservices Evolutionary Patterns to Transform your Monolith*. O'Riley.

PREDIC8, 2016. *The State of Developer Ecosystem 2022*. Auch verfügbar unter: https://youtu.be/mTG99l7LVOY. [Online; Stand: 25. Juli 2023].

PREDIC8, 2019. *Microservices schneiden - Schnitt und Architektur*. Auch verfügbar unter: https://youtu.be/1REgyrRowNw. [Online; Stand: 25. Juli 2023].

PREDIC8, 2022. *Warum ein Monolith oft besser ist als Microservices*. Auch verfügbar unter: https://youtu.be/rcLG4ZsNEA0. [Online; Stand: 25. Juli 2023].

REES, Brad, 2021. Similarity in Graphs: Jaccard Versus the Overlap Coefficient. Auch verfügbar unter: https://developer.nvidia.com/blog/similarity-in-graphs-jaccard-versus-the-overlap-coefficient-2/. [Online; Stand: 25. Juli 2023].

RESELMAN, Bob, 2022a. 5 design principles for microservices. Auch verfügbar unter: https://developers.redhat.com/articles/2022/01/11/5-design-principles-microservices#what_is_a_microservices_oriented_application. [Online; Stand: 25. Juli 2023].

RESELMAN, Bob, 2022b. The disadvantages vs. benefits of microservices. Auch verfügbar unter: https://developers.redhat.com/articles/2022/01/25/disadvantages-microservices. [Online; Stand: 25. Juli 2023].

RICHARDSON, Chris, 2014. Microservices: Decomposing Applications for Deployability and Scalability. Auch verfügbar unter: https://www.infoq.com/articles/microservices-intro/. [Online; Stand: 25. Juli 2023].

RICHARDSON, Chris, [o. D.]. *The Scale Cube*. Auch verfügbar unter: https://microservices.io/articles/scalecube.html. [Online; Stand: 25. Juli 2023].

STEIN, H., [o. D.]. *Einfache Abfragen mit Selektion und Projektion (SQL-DML/DQL)*. Auch verfügbar unter: https://oer-informatik.de/sql-dml-selektion-projektion. [Online; Stand: 28. Juli 2023].

TYSON, Matthew, 2022. *What is JPA? Introduction to Java persistence*. Auch verfügbar unter: https://www.infoworld.com/article/3379043/what-is-jpa-introduction-to-the-java-persistence-api.html. [Online; Stand: 28. Juli 2023].

VERNON, Vaughn, 2016. *Domain-Driven Design Distilled*. Addison-Wesley Professional. ISBN 978-0134434421.

VUE, [o. D.(a)]. *Composition API FAQ*. Auch verfügbar unter: https://vuejs.org/guide/extras/composition-api-faq.html#more-flexible-code-organization. [Online; Stand: 25. Juli 2023].

VUE, [o. D.(b)]. *Template Refs*. Auch verfügbar unter: https://vuejs.org/guide/essentials/template-refs.html#accessing-the-refs. [Online; Stand: 25. Juli 2023].

8 Anhang

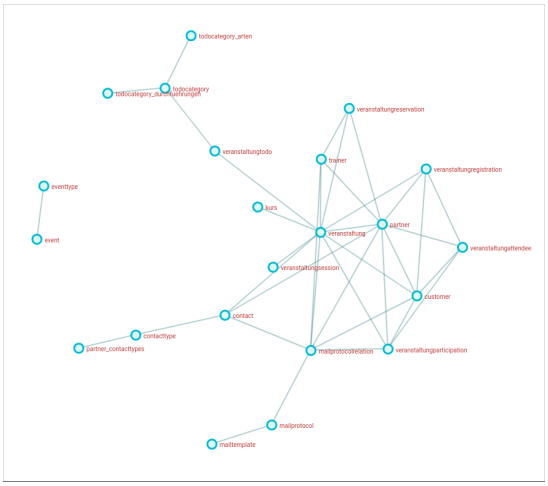

Abbildung 8.1: Firmenanwendung vor Anheben des Thresholds

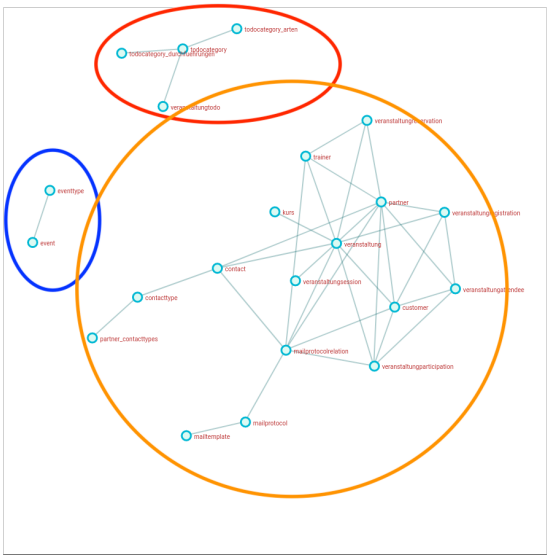

Abbildung 8.2: Firmenanwendung nach dem ersten Anheben des Threshold

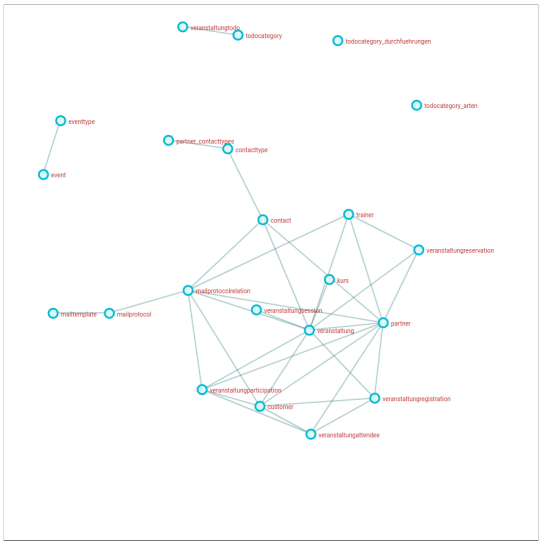

Abbildung 8.3: Firmenanwendung nach inkrementellen Anheben des Threshold

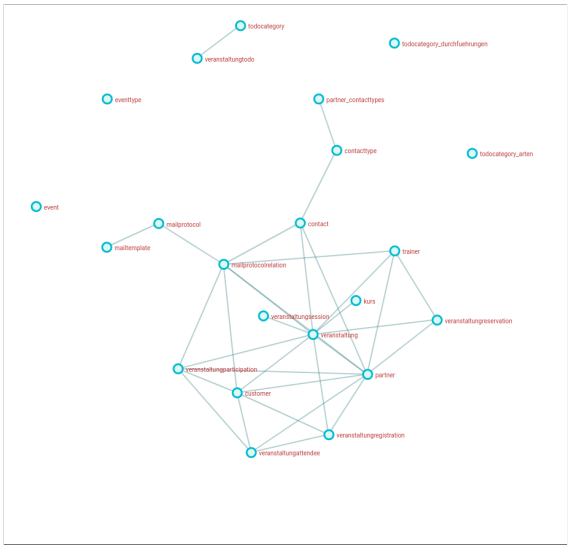

Abbildung 8.4: Firmenanwendung nach weiterem Anheben des Threshold

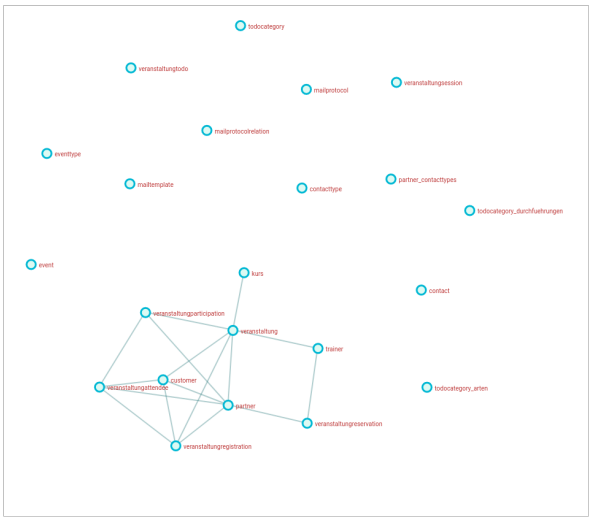

Abbildung 8.5: Starkzusammenhängender Kern Firmenanwendung

Abbildung 8.6: CRM intial

Abbildung 8.7: CRM vor dem Setzen des ersten Schnitts

Abbildung 8.8: CRM mit drei identifizierten Domänen

```kotlin
fun main(args: Array<String>) {

    val tuples: List<Pair<String, String>> = jacksonObjectMapper()
        .registerKotlinModule()
        .readValue(File(args[0]).readText(Charsets.UTF_8))

    // for the algorithm
    val dict = hashMapOf<Pair<String, String>, Float>()
    val nodesAndLinks = Elements(mutableListOf(), mutableListOf())
    val copyOfCopyOfTuples = tuples.toMutableList()
    val copyOfTuples = tuples.toMutableList()
    var tuplesToDirectRemove: MutableList<Pair<String, String>> = mutableListOf()
    var tuplesToIndirectRemove: MutableList<Pair<String, String>> = mutableListOf()

    while (copyOfTuples.size > 0) {
        val currentTuple = copyOfTuples.removeAt(0)

        tuplesToDirectRemove.addAll(copyOfTuples.filter
        { currentDirectCompareTuple ->
            directCompare(currentTuple, currentDirectCompareTuple) })
        copyOfTuples.removeAll(tuplesToDirectRemove)
        copyOfCopyOfTuples.removeAll(tuplesToDirectRemove)

        tuplesToIndirectRemove.addAll(copyOfCopyOfTuples.filter
        { currentIndirectCompareTuple ->
            indirectCompare(currentTuple, currentIndirectCompareTuple) })
        copyOfCopyOfTuples.addAll(tuplesToDirectRemove)

        dict[currentTuple] =
            calculateLinkWeight(tuplesToDirectRemove.size.toFloat() + 1f,
                tuplesToIndirectRemove.size.toFloat())
        tuplesToDirectRemove = mutableListOf()
        tuplesToIndirectRemove = mutableListOf()
    }

    dict.map { listOf(Node(it.key.first, it.key.first),
        Node(it.key.second, it.key.second)) }.flatten().forEach {
        when {
            checkIfNodeIsAlreadyInList(it, nodesAndLinks.nodes)
            -> nodesAndLinks.nodes.add(it)
        }
    }

    dict.map { Link(sid = it.key.first, tid = it.key.second, name = it.value) }
        .forEach { nodesAndLinks.links.add(it) }

    File(getInputFileDirectory(args[0]) + getOutputFileName(args[0])).writeText(
        GsonBuilder().setPrettyPrinting()
            .create().toJson(nodesAndLinks)
    )
}
```

Abbildung 8.9: DA Main-Methode vor den Refactoring

```
<template>
  <!-- ... -->
  <v-col>
    {{ "Force: " + nodeForce }}
    <v-slider v-model="nodeForce" color="primary" min="0" max="10000" />
  </v-col>
  <!-- ... -->
</template>

<script>
export default {
  nodeForce: 5000,
  // ...
  computed: {
    options() {
      return {
        // ...
        force: this.nodeForce,
        // ...
        size: {
          h: "",
          w: "",
        },
      };
    },
  },
  // ...
  mounted() {
    this.options.size.h = this.$el.clientHeight * this.vuePortSize;
    this.options.size.w = this.$el.clientWidth;
  },
}
</script>
```

Abbildung 8.10: Beispiel für Aufteilung der Daten in der Option-API

```
<template>
  <!-- ... -->
<div ref="networkContainer"/>
  <!-- ... -->
</template>

<script setup lang="ts">
  //...
  const network = ref<Network>();
  const networkContainer: Ref<HTMLElement | null> = ref(null);
  //...
  if(networkContainer.value) {
  network.value = new Network(networkContainer.value, data.value,
      networkOptions.value)
  }
  //...
</script>
```

Abbildung 8.11: Zugriff auf das DOM-Element