Zürcher Hochschule
für Angewandte Wissenschaften

School of
Engineering

InIT Institut für angewandte
Informationstechnologie

Bachelorarbeit (Informatik)

Interaktive Konstruktion von Datenbankabfragen

Autoren	Nicolas Kaiser Philippe Schläpfer
Hauptbetreuung	Dr. Mark Cieliebak Dr. Kurt Stockinger
Nebenbetreuung	Jan Deriu

Datum	04.06.2019

Zürcher Hochschule
für Angewandte Wissenschaften

School of
Engineering

Erklärung betreffend das selbständige Verfassen einer Bachelorarbeit an der School of Engineering

Mit der Abgabe dieser Bachelorarbeit versichert der/die Studierende, dass er/sie die Arbeit selbständig und ohne fremde Hilfe verfasst hat. (Bei Gruppenarbeiten gelten die Leistungen der übrigen Gruppenmitglieder nicht als fremde Hilfe.)

Der/die unterzeichnende Studierende erklärt, dass alle zitierten Quellen (auch Internetseiten) im Text oder Anhang korrekt nachgewiesen sind, d.h. dass die Bachelorarbeit keine Plagiate enthält, also keine Teile, die teilweise oder vollständig aus einem fremden Text oder einer fremden Arbeit unter Vorgabe der eigenen Urheberschaft bzw. ohne Quellenangabe übernommen worden sind.

Bei Verfehlungen aller Art treten die Paragraphen 39 und 40 (Unredlichkeit und Verfahren bei Unredlichkeit) der ZHAW Prüfungsordnung sowie die Bestimmungen der Disziplinarmassnahmen der Hochschulordnung in Kraft.

Ort, Datum:

……

Unterschriften:

……...

……

……

Das Original dieses Formulars ist bei der ZHAW-Version aller abgegebenen Bachelorarbeiten zu Beginn der Dokumentation nach dem Titelblatt mit Original-Unterschriften und -Datum (keine Kopie) einzufügen.

Zusammenfassung

Datenbanken enthalten sehr viele strukturierte Daten, die für Personen aus verschiedensten Aufgabengebieten interessant sein können. Der Zugang zu diesen Daten ist für Personen ohne Kenntnisse in einer Datenbanksprache allerdings stark begrenzt. Um diesen Zugang auch Laien bieten zu können, wäre eine Google-ähnliche Applikation wünschenswert, in der ein Benutzer eine natürlichsprachliche Frage, welche sich mit Hilfe von Informationen einer Datenbank beantworten lässt, eingibt und anschliessend die gewünschte Antwort erhält. Diese Applikation benötigt jedoch eine grosse Menge an Trainingsdaten, um mittels Machine Learning lernen und sich verbessern zu können.

Um diese Daten generieren zu können, soll der bestehende Prototyp aus der gleichnamigen Projektarbeit erweitert werden, so dass anhand einer natürlichsprachlichen Frage nicht nur die korrekte Antwort aus der Datenbank geliefert wird, sondern auch die einzelnen Teilschritte des Benutzers in einer geeigneten Form aufgezeichnet werden.

Diese Arbeit beschäftigt sich mit dem Thema *"Interaktive Konstruktion von Datenbankabfragen"*. Das bestehende Konzept wurde teilweise übernommen, so dass sich ein Benutzer seine natürlichsprachliche Frage, welche einer komplexeren Datenbankabfrage entsprechen würde, anhand einer geeigneten Kombination von atomaren Operationen beantworten kann. Der Benutzer stellt sich so die komplette Abfrage zusammen, ohne auch nur eine Zeile SQL-Code schreiben zu müssen und erhält am Schluss das gewünschte Resultat. Der Prototyp wurde angepasst und erweitert, so dass die Einzelschritte des Benutzers in einer geeigneten Form aufgezeichnet werden und die Applikation künftig als Annotationstool genutzt werden kann, um Trainingsdaten für zukünftiges Machine Learning zu generieren. Zusätzlich wurden Funktionen implementiert, um die generierten Logfiles analysieren zu können. Das Ergebnis dieser Arbeit stellt eine Webapplikation dar, welche die geforderten Funktionen erfüllt. Weiterentwicklungen in Richtung Benutzeroberfläche könnten die Benutzerfreundlichkeit noch weiter verbessern.

Abstract

Databases contain a lot of structured data, which can be interesting for people from different domains. However, access to this data is very limited for people without knowledge of a database query language. In order to be able to offer this access to laypersons, a Google-like application would be desirable in which a user enters a natural language question, which can be answered with the information of an underlying database, and then receives the desired answer. Such an application requires a large amount of training data in order to learn and improve through machine learning.

To be able to generate this data, the existing prototype is to be extended. Given a natural language question, not only the correct answer is delivered from the database, but also the individual steps of the user are recorded in an appropriate form.

This thesis deals with the topic *"Interactive construction of database queries"*. The already existing concept from a previous student project was partially adopted, so that a user can answer a natural language question, which would correspond to a more complex database query, with a suitable combination of atomic operations. The user puts together the complete query without having to write a single line of SQL code and finally gets the desired result. The prototype was adapted and extended so that the individual steps of the user are recorded in a suitable form. Additionally, the application can be used as an annotation tool to generate training data for machine learning algorithms. On top of that, functions were implemented to analyze the generated log files. The result of this work is a web application that fulfils the required features. Future developments regarding the user interface could further improve the usability.

Vorwort

Die Bereiche Datenbanken und Software-Entwicklung haben unser Interesse schon nach den ersten Vorlesungen im Studium geweckt. Später im Studium kamen wir mit den Themen Datenanalyse und potenziellen Anwendungen von Machine-Learning-Algorithmen in Kontakt, was uns ebenfalls sehr interessierte. Von da an war uns beiden klar, dass wir eine Bachelorarbeit in diesem Gebiet wählen werden. Als wir dann diese Arbeit ausgeschrieben gesehen haben, mussten wir nicht lange überlegen und haben uns sofort eintragen lassen.

Was uns zusätzlich motivierte diese Arbeit zu verfassen, ist die Tatsache, dass dieses Problem immer noch ein ungelöstes Problem der Informatik ist. Dadurch hatten wir die Möglichkeit, in ein uns völlig neues Themengebiet und ein topaktuelles Forschungsthema einzusteigen. Die Idee einer Applikation, mit der jedermann anhand einer natürlichsprachlichen Frage Informationen aus einer Datenbank beziehen kann, hat uns fasziniert und vorangetrieben.

Nicht selten standen wir vor Problemen, welche uns viel abverlangt haben, bis wir schlussendlich eine zufriedenstellende Lösung präsentieren konnten.

An dieser Stelle wollen wir uns bei Herrn Prof. Dr. Mark Cieliebak, Herrn Prof. Dr. Kurt Stockinger und Herrn Jan Deriu, für all die interessanten Besprechungen, die sehr nützlichen Inputs und die Geduld, die sie uns entgegenbrachten, bedanken. Ohne dem Know-How unserer Betreuer wäre diese Arbeit so nicht möglich gewesen.

Inhaltsverzeichnis

1. Einleitung

1.1. Motivation

„Datenbanken enthalten sehr viele strukturierte Daten, die für Personen aus den verschiedensten Aufgabengebieten interessant sein können. Für den Zugang zu diesen Daten sind allerdings Kenntnisse in einer Abfragesprache für Datenbanken (SQL, SPARQL, etc.) notwendig. Dies hat zur Folge, dass Laien nur indirekt auf dieses Wissen zugreifen können, indem sie einen Datenbankexperten beauftragen, eine spezifische Abfrage zu übersetzen und die gewonnenen Resultate dem Auftraggeber zurückzusenden. Es besteht eine grosse Lücke zwischen einem kleinen Kreis von Datenbankexperten, welche auf die Daten zugreifen können, und der erheblich grösseren Menge von Laien, für welche ein Zugriff auf diese Daten von Nutzen sein könnte. Würde ein Programm existieren, in welchem man eine konkrete Frage eingeben kann und anschliessend das gewünschte Resultat aus der Datenbank erhält, wäre ein riesiger Schritt getan, diese Lücke zu schliessen oder zumindest zu verkleinern. Solch eine Applikation soll "intelligent" sein, was wiederum nur möglich ist, wenn sie aus einer genügend grossen Anzahl bereits getätigter Abfragen lernen kann. Den Grundstein, nämlich die für das Lernen benötigten Daten zu generieren, könnte mit einer Applikation gelegt werden, welche Datenbankexperten verschiedene Abfragen durchführen lässt und deren Vorgehensweise dokumentiert." [1]

1.2. Aufgabenstellung

In dieser Arbeit soll eine Applikation erstellt werden, mit der eine Frage mit Hilfe von atomaren Operationen auf einer relationalen Datenbank beantwortet werden kann, ohne dass dabei Code in einer Abfragesprache geschrieben werden muss. Der Benutzer soll sich dabei die benötigten atomaren Operationen selbst zusammenstellen und ausführen können. Die ausgeführten Operationen sollen dabei in einer geeigneten Form in einem Logfile gespeichert werden. Diese Logfiles sollen künftig als annotierte Daten für Machine Learning verwendet werden können, um das Beantworten der Fragen zu automatisieren.
Das Ergebnis dieser Arbeit soll von Personen benutzt werden können, welche schon einen gewissen Grad an Erfahrung im Datenbankbereich mitbringen, sprich die verschiedenen Operationen von SQL kennen und wissen, für was man sie einsetzt.
Als Grundlage dieser Arbeit dient der Prototyp, welcher in der Projektarbeit *"Interaktive Konstruktion von Datenbankabfragen"* [2] erstellt wurde. Die wichtigsten Operationen sind bereits implementiert, allerdings ist das Tool stark an MySQL gebunden. Dies soll geändert werden, damit in Zukunft eine beliebige relationale Datenbank dem Tool zur Verfügung gestellt werden kann. Die bestehende Benutzeroberfläche soll erweitert werden, so dass die Bedienung möglichst einfach und simpel ist.

1.3. Gliederung des Dokuments

Zu Beginn des Dokuments wird dem Leser einen Überblick über das Fachgebiet von Question-Answering-Systemen verschafft. Dazu werden die verschiedenen Ansätze solcher Systeme beschrieben und erläutert, mit welchen Arten von Daten sie sich beschäftigen.

Im nächsten Kapitel wird das Konzept dieser Arbeit vorgestellt. Es wird die Grundidee der Applikation genauer beschrieben und wie die Struktur des Loggings aussehen soll. Zusätzlich wird der Leser in die eigens entwickelte Sprache, die *Semantic Query Language*, eingeführt.

Anschliessend wird beschrieben, wie das Konzept in eine Webapplikation umgesetzt wurde. Dabei wird mit einem Fallbeispiel begonnen, so dass der Leser einen Überblick über die gesamte Applikation erhält. Nach dem Fallbeispiel werden die verschiedenen Funktionalitäten des Tools genauer beschrieben.

Im darauffolgenden Kapitel wird die Analyse von gut 100 mit dem Tool beantworteten Beispielfragen erläutert. Aus dieser Analyse werden erste Erkenntnisse gezogen, vor allem im Hinblick auf Machine-Learning-Algorithmen, aber auch, um Anhaltspunkte für zukünftige Arbeiten zu identifizieren.

Im letzten Kapitel wird auf Grundlage der Auswertung im vorherigen Kapitel diskutiert, was an dieser Applikation verbessert werden könnte und Punkte angesprochen, was im Hinblick auf das Machine Learning beachtet werden muss.

2. Question-Answering-Systeme

2.1. Überblick

Die Disziplin des Question Answering befasst sich damit, Systeme zu entwickeln, die Fragen beantworten können, welche von einem Benutzer in natürlicher Sprache an das System gestellt werden. Dabei existieren viele verschiedene Ansätze, um dieses Problem zu lösen. Diese Ansätze können grob unterteilt werden in regelbasierte Methoden, Methoden, welche Machine-Learning-Algorithmen einsetzen und hybride Methoden, welche beide Ansätze zu ihrem Vorteil nutzen wollen.

Ein Nachteil von regelbasierten Systemen ist, dass sie nur einen kleinen Teil von gestellten Fragen erfolgreich erfassen können, denn die natürliche Sprache ist sehr vielfältig. Fragen, welche aus demselben Informationsbedürfnis entstehen, können in unterschiedlichen Arten formuliert werden.
Ein Nachteil von Ansätzen mit Machine-Learning-Algorithmen ist, dass es schwierig ist, umfassende Trainingsdatensets für diese Art von Problemen zusammenzustellen.

Weiter können die darunterliegenden Daten, von welchen die Antworten auf die Fragen extrahiert werden sollen, in folgende Kategorien unterteilt werden [3]:

- Unstrukturierte Daten (z.B. Dokumente, das World Wide Web)
- Strukturierte Daten (z.B. relationale Datenbanken)
- Knowledge Bases (eine Datenbank, die Wissen abspeichert, nicht nur reine Daten)

Es wurden schon viele Versuche unternommen, Lösungen für Question-Answering-Systeme zu implementieren. Da es bei dieser Arbeit um ein Question-Answering-System für Datenbanken, also vorwiegend strukturierte Daten, geht, konzentrieren sich die folgenden Abschnitte auf Herangehensweisen auf ebensolchen strukturierten Daten.

SODA
SODA (Search Over DAta Warehouse) [4] ist ein System, das mit Hilfe von regelbasierten Methoden aus einer natürlichsprachlichen Frage eine funktionierende SQL-Abfrage generiert. Dazu nimmt es sich einen sogenannten Metadaten-Graphen zur Hilfe. Dieser besteht zum einen aus drei verschiedenen Schemata. Das erste Schema ist das konzeptuelle Schema. Dieses Schema „dient zur Kommunikation mit dem Business und unterscheidet Entitäten wie z.B. Parteien oder Transaktionen“[4]. Das zweite Schema erweitert das konzeptuelle Schema, indem die Entitäten weiter aufgeteilt werden (z.B. Parteien in Organisationen und Individuen). Das dritte Schema,

das physische, beinhaltet Informationen über die Datenbank, wie beispielsweise Datenbankindizes. Zum anderen besteht der Metadaten-Graph aus einer für ein spezifisches Data Warehouse erstellten Domänenontologie. Mit Hilfe dieser Ontologie können beispielsweise Kunden klassifiziert werden. Um Synonyme zu erkennen, wird DBPedia verwendet.
Der Benutzer kann eine Suche in natürlicher Sprache eingeben. Das System erstellt mehrere mögliche SQL-Queries und rangiert diese. Diese Rangliste wird dann dem Benutzer angezeigt, so dass dieser ähnlich wie bei einer Suchmaschine für das Web das für ihn beste Resultat auswählen kann.

DBPal

DBPal [5] benutzt im Unterschied zu SODA einen Deep-Learning Ansatz, mit dem die Frage des Benutzers zu einer SQL-Abfrage übersetzt wird. Die Autoren erhoffen sich davon, dass damit die Vielfalt der natürlichen Sprache besser erfasst werden kann und somit das gesamte System robuster wird.
Ein weiterer Teil von DBPal ist eine grafische Benutzeroberfläche, bei der einem Benutzer laufend Vorschläge gemacht werden, während er eine Frage am eingeben ist. Abbildung 2.1 zeigt ein Beispiel von dieser Auto-Complete-Funktionalität.

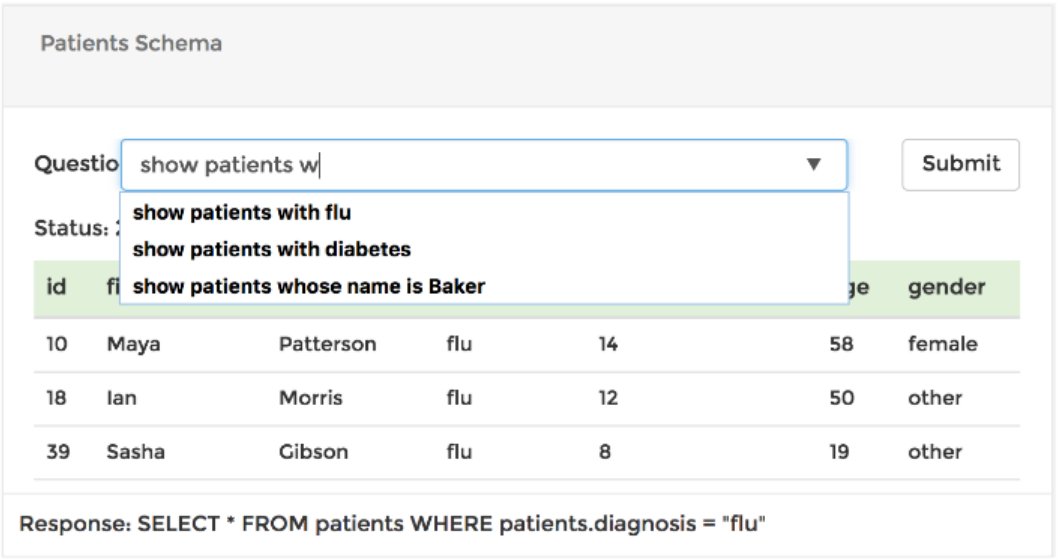

Abbildung 2.1.: Eine Beispielfrage im GUI von DBPal

3. Konzept

3.1. Grundidee

Zielpublikum des Tools sind Benutzer, welche bereits über Kenntnisse in einer Datenbanksprache verfügen, bzw. mindestens wissen, welche Operationen eine Datenbank zur Verfügung stellt und wofür sie gebraucht werden. Die Grundidee der Applikation besteht dann darin, dass der Benutzer eine natürlichsprachliche Frage eingibt und sich anschliessend mit den ihm zur Verfügung stehenden atomaren Operationen eine Abfrage zusammenstellt. Hat er zu Beginn keine Informationen über das Datenbankschema, kann er sich die Datenbank grafisch als ungerichteter Graph in der Weboberfläche ansehen und sich einen Überblick verschaffen. Jedes Mal, wenn der Benutzer eine neue Operation ausgewählt hat, wird er anschliessend in einem Pop-up-Menü gefragt, welche Teile seiner Frage für die gerade gewählte Operation entscheidend waren, sprich welche Wörter er mit der Operation assoziiert. Er führt dabei so lange Operationen aus, bis er sich seine Frage beantwortet hat und das Schlussresultat vor sich sieht. War die Abfrage erfolgreich und der Benutzer konnte sich seine Frage beantworten, werden jegliche Schritte des Benutzers, so wie die einzelnen Wortassoziationen in einem Logfile gespeichert. Die Idee hinter den Logfiles ist dabei, dass in Zukunft diese Files als Trainingsdaten für einen Machine-Learning-Algorithmus verwendet werden können. Es wird dabei darauf abgezielt, dass der Algorithmus anhand der Assoziationen von den einzelnen Wörtern der Frage zu den Operationen gewisse Muster erkennt und somit ähnliche Fragen anschliessend automatisch beantwortet werden können. Als Vereinfachung wurde für die gesamte Arbeit davon ausgegangen, dass mit dem Tool Datenbanken durchsucht werden, die Tabellen mit sprechenden Namen verwalten. Dies ermöglicht neben der Suche durch die Basisdaten der Datenbank auch die Suche durch Metadaten wie die Tabellen- und Spaltennamen.

3.2. Baumstruktur (Zustand der Applikation)

Die grundlegende Datenstruktur der gesamten Applikation bildet eine Baumstruktur. In dieser Baumstruktur ist der gesamte Zustand der Applikation während dem Beantworten einer Frage gespeichert. Der Baum ist so aufgebaut, dass sich die Ebenen immer zwischen einem Operationsknoten und einem Datenknoten abwechseln. Die Operationsknoten stellen die eigentliche Operation, welche der Benutzer gewählt hat, dar. Sie erzeugen als Output einen Datenknoten, welcher die resultierenden Daten der Operation enthält und die nächste Ebene im Baum bildet. Dieser Datenknoten kann dann wiederum als Input für die nächste Operation genutzt werden.

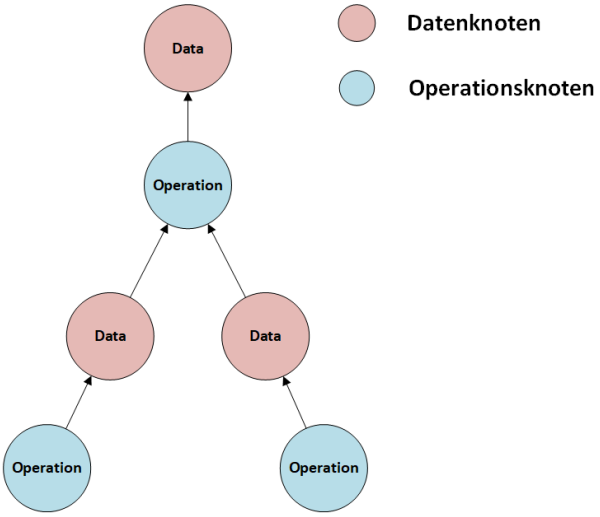

Abbildung 3.1.: Beispiel der Baumstruktur der Applikation

Abbildung 3.1 zeigt ein sehr allgemein gehaltenes Beispiel für diese Baumstruktur. Zu beachten ist, dass der Baum im Gegensatz zu einem in der Informatik üblichen Baum bei den Blättern beginnt und in der Wurzel mündet. In diesem Beispiel könnte der letzte Datenknoten die Antwort auf die gestellte Frage sein.

Es ist möglich, gewisse Teilbäume während dem Beantworten der Frage zu löschen, falls der Benutzer beispielsweise merkt, dass in einem vorherigen Schritt eine falsche Operation ausgeführt wurde. Das Löschen ist allerdings nur möglich, wenn kein im zu löschenden Teilbaum enthaltener Datenknoten einem Operationsknoten als Input gedient hat. Diese Überprüfung muss bei der Umsetzung zwingend implementiert werden, denn sonst könnte es passieren, dass die aufgezeichneten Bäume unvollständig sind und somit nicht mehr für das Machine-Learning verwendet werden können.

3.3. Datenbankunabhängigkeit

Eine Vorgabe war, dass das Tool möglichst unabhängig von einer spezifischen Datenbanktechnologie aufgebaut werden soll. Um dies zu erreichen, wurde entschieden, dass die verschiedenen Operationen eigens implementiert werden und nicht mehr direkt in der Datenbank ausgeführt werden sollen. Zusätzlich wurde für die verschiedenen Operationen eine Sprache (Kapitel 3.4) definiert, die unabhängig von einer Datenbanktechnologie ist und so für die gängigsten Datenbanktechnologien funktioniert. Genannt wird sie *Semantic Query Language*, oder auch *SemQL*. Der Vorteil dabei ist, dass bei einem Wechsel der Datenbanktechnologie nur die Anbindung zur Datenbank angepasst werden muss. Auch das Logging ist dank dieser definierten Sprache unabhängig von der gewählten Technologie.

Die Anbindung an verschiedene Datenbanktechnologien zu implementieren, hätte den Rahmen dieser Arbeit gesprengt, weswegen sich für diese Arbeit auf die Anbindung an relationale Datenbanken, genauer MySQL-Datenbanken, konzentriert wurde.

3.4. Semantic Query Language (SemQL)

Damit die Sprache unabhängig von den Datenbanktechnologien ist, musste ein gemeinsamer Nenner zwischen den verschiedenen Technologien gefunden werden. Etwas, das fast alle gängigen Technologien gemeinsam haben ist die Tatsache, dass sie verschiedene Typen von Entitäten verwalten, welche Beziehungen zueinander haben. In SQL sind das zum Beispiel Tabellen, welche über Primär- und Fremdschlüssel miteinander verbunden sind.
Die Grundlage von SemQL bildet die relationale Bag-Algebra und erweitert deren Operationen. Ein Bag ist dabei „wie ein Set, allerdings können Elemente mehrfach darin vorkommen“ [6].

3.4.1. Operationen

In SemQL sind folgende atomare Operationen definiert:

Get Data

Die *GET-DATA*-Operation holt eine Menge von Daten aus der Datenbank und gibt diese zurück. Sie ist typischerweise die erste Operation einer Query. Diese Operation ist in der relationalen Bag-Algebra nicht definiert.

Merge

Die *MERGE*-Operation verbindet zwei Datenknoten. Es muss spezifiziert werden, anhand welcher Attribute die Datenknoten verbunden werden sollen. Sie entspricht dem Equi-Join zweier Tabellen aus der relationalen Bag-Algebra.

Filter

Die *FILTER*-Operation filtert einen Datenknoten nach einer Bedingung. Diese Bedingung wird auf einem Attribut angegeben und verwendet eine der Operationen $\{<, \leq, \geq, >, =, \neq$, BETWEEN$\}$. Sie entspricht der Selektion aus der relationalen Bag-Algebra.

Extract Values

Die *EXTRACT-VALUES*-Operation extrahiert aus einer Tabelle mit Attributen eine Spalte und gibt deren Werte als Liste aus. Diese Operation ist in der relationalen Bag-Algebra nicht definiert. Sie kann jedoch am ehesten mit der Projektion verglichen werden, mit dem Unterschied, dass Attribute bei dieser Operation entfernt werden, so dass nur noch Werte in einer Liste vorhanden sind. Die Idee hinter dieser Operation ist, dass bei einer Antwort auf eine Frage die Attribute nicht mehr von Interesse sind, sondern nur noch die Feldinhalte.

Duplikatentfernung

Die *DISTINCT*-Operation entfernt Duplikate aus einer Liste von Werten. Diese Operation entspricht der *DISTINCT*-Operation der relationalen Bag-Algebra. In SemQL ist diese Operation nur auf extrahierten Werten definiert.

Aggregatsoperationen

Als Aggregatsoperationen stehen die folgenden Operationen zur Verfügung:

- *MIN*: Selektiert den kleinsten Wert aus einer Liste von numerischen Werten
- *MAX*: Selektiert den grössten Wert aus einer Liste von numerischen Werten
- *SUM*: Berechnet die Summe aus einer Liste von numerischen Werten
- *AVERAGE*: Berechnet den Durchschnitt aus einer Liste von numerischen Werten
- *COUNT*: Gibt die Anzahl Einträge zurück, die in einem Datenknoten vorhanden sind

Diese Aggregatsoperationen sind in der relationalen Bag-Algebra nicht definiert.

Mengenoperationen

Als Mengenoperationen stehen die *UNION*-, *INTERSECT*- und *DIFFERENCE*-Operationen zur Verfügung. Diese drei Operationen entsprechen den Mengenoperationen der relationalen Bag-Algebra.

3.5. Loggingbaum

Wie in Kapitel 3.1 beschrieben sollen die Operationen, welche ein Benutzer ausführt, um eine Frage zu beantworten, in einer geeigneten Struktur gespeichert werden. Durch die Tatsache, dass der ganze Zustand der Applikation bereits in einer Baumstruktur gespeichert ist (Kapitel 3.2), stellte sich heraus, dass Teile dieser Baumstruktur direkt übernommen werden können, um die Baumstruktur für das Logfile aufzubauen. Die Loggingbaumstruktur soll dabei möglichst klein sein und nur gerade die nötigen Informationen enthalten, um die Abfrage reproduzieren zu können. Für die Loggingbaumstruktur wurden deshalb nur die Operationsknoten übernommen. Die Datenknoten wurden nicht übernommen, da sie als Resultate der vorherigen Operation entstehen und somit nicht zwingend nötig gespeichert werden müssen. Die Operationsknoten werden dabei in SemQL in das Logfile eingetragen.

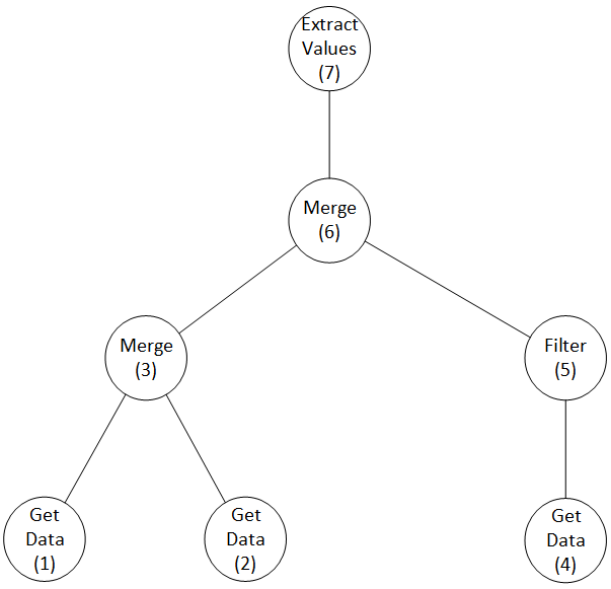

Abbildung 3.2.: Beispiel eines binären Loggingbaums

In Abbildung 3.2 wird ein Beispiel eines binären Loggingbaums dargestellt. In Klammern sind jeweils die IDs der Knoten angegeben. Die IDs sagen gleichzeitig auch aus, in welcher Reihenfolge die Operationen ausgeführt wurden, da sie inkrementell generiert werden. Im Unterschied zu einem klassischen Binärbaum fängt ein Loggingbaum an den Blättern an. Auf das Beispiel in Abbildung 3.2 bezogen heisst das, dass zuerst die beiden *GET-DATA*-Operationen mit den IDs *1* und *2* ausgeführt werden müssen, bevor die *MERGE*-Operation mit der ID *3* durchgeführt werden kann. Zusätzlich zum eigentlichen Loggingbaum wird im Logfile die beantwortete Frage gespeichert sowie ein Flag, das aussagt, ob der Benutzer seine Frage beantworten konnte oder nicht.

3.6. Datenbankschema (Movie Database)

Während der Entwicklung des Tools wurde immer die gleiche Datenbank verwendet. Es wurde dabei eine Datenbank ausgewählt (Abbildung 3.3), welche Informationen über Schauspieler, Regisseure, Filme, Oscars etc. enthält. Sie hat eine optimale Grösse für eine Testdatenbank, da sie über genügend viele Tabellen verfügt, um sinnvolle Operationen ausführen zu können, allerdings auch nicht zu gross ist, um durch zu grosse Datenmengen Probleme zu verursachen. Die wichtigsten Tabellen, die in dieser Arbeit oft benutzt wurden, werden kurz vorgestellt, so dass der Leser ein wenig mehr Hintergrundwissen über die Datenbank erlangt.

movie

Die Tabelle *movie* enthält Informationen über Filme, beispielsweise den Titel, das Budget, den Umsatz, die Beliebtheit, etc.

person

Die Tabelle *person* enthält Informationen über Personen. Dies können entweder Schauspieler, Regisseure oder sonstige Personen sein, welche in einem Film mitgewirkt haben. Über eine Person ist der Name, das Geschlecht, Geburtsort und -datum, sowie Todesort und -datum (falls die Person bereits verstorben ist) bekannt.

cast

Die Tabelle *cast* ist eine Beziehungstabelle zwischen *person* und *movie*. In dieser Tabelle wird die Beziehung "*hat* ***mitgespielt*** *in*" dargestellt, sprich nur die Schauspieler sind erfasst. Zusätzlich wird gespeichert, wie der Charakter im Film heisst.

crew

Die Tabelle *crew* ist ebenfalls eine Beziehungstabelle zwischen *person* und *movie*. In dieser Tabelle wird die Beziehung "*hat* ***mitgearbeitet*** *in*" dargestellt, das heisst alle Mitarbeiter (Regisseure, Drehbuchautoren, Produzenten, etc.) sind erfasst. Zusätzlich wird gespeichert, welche Funktion die Person während der Produktion hatte und in welcher Abteilung sie gearbeitet hat.

Das vollständige Datenbankschema wird auf der nächsten Seite in Abbildung 3.3 gezeigt.

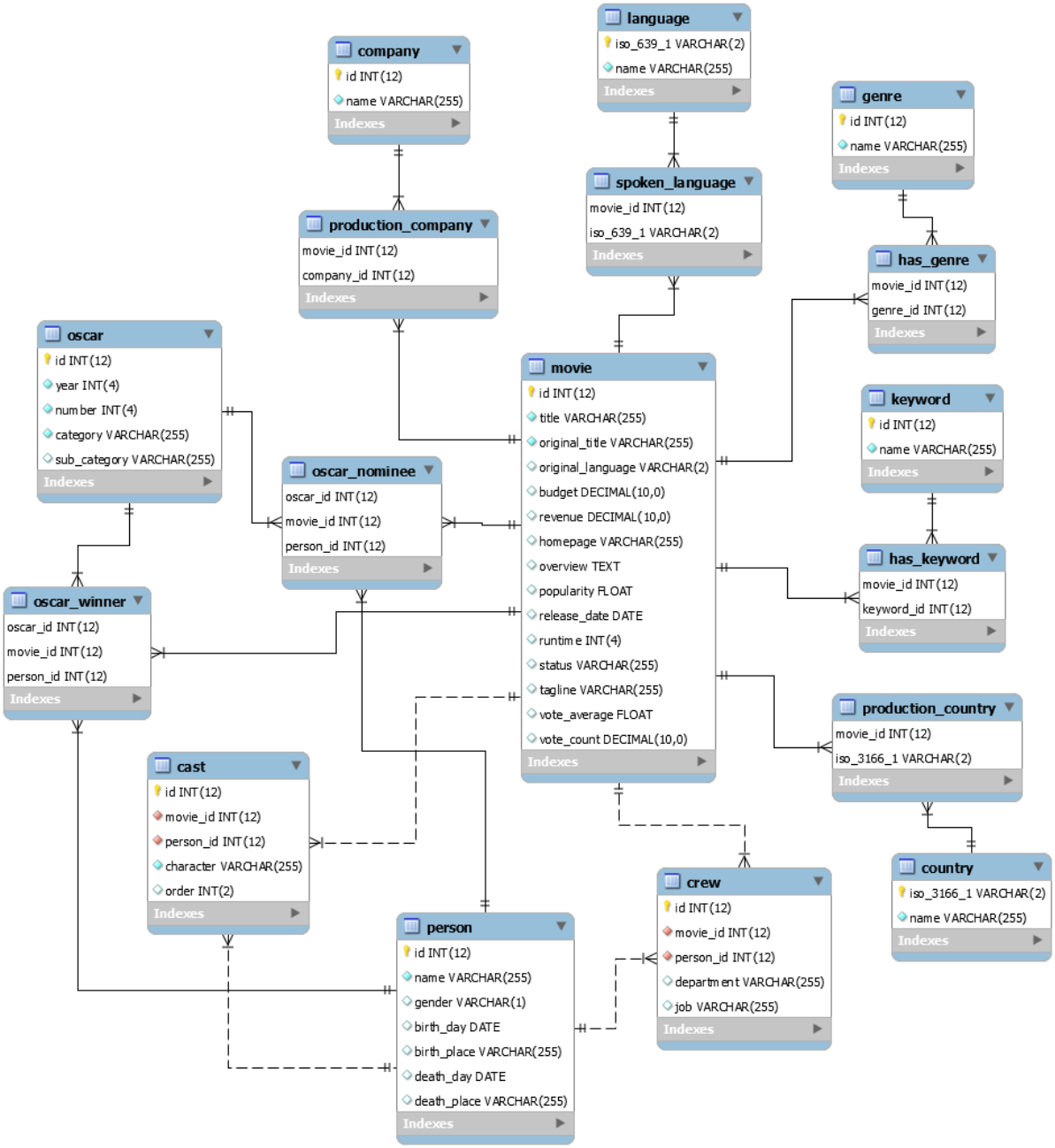

Abbildung 3.3.: Datenbankschema der Movie Database

4. Umsetzung

4.1. Fallbeispiel

In diesem Kapitel wird anhand eines Beispiels aufgezeigt, wie die Beantwortung einer Frage ablaufen könnte. Die Beispielfrage lautet folgendermassen:
In which movies did Brad Pitt play, where the budget was greater than 100000000?

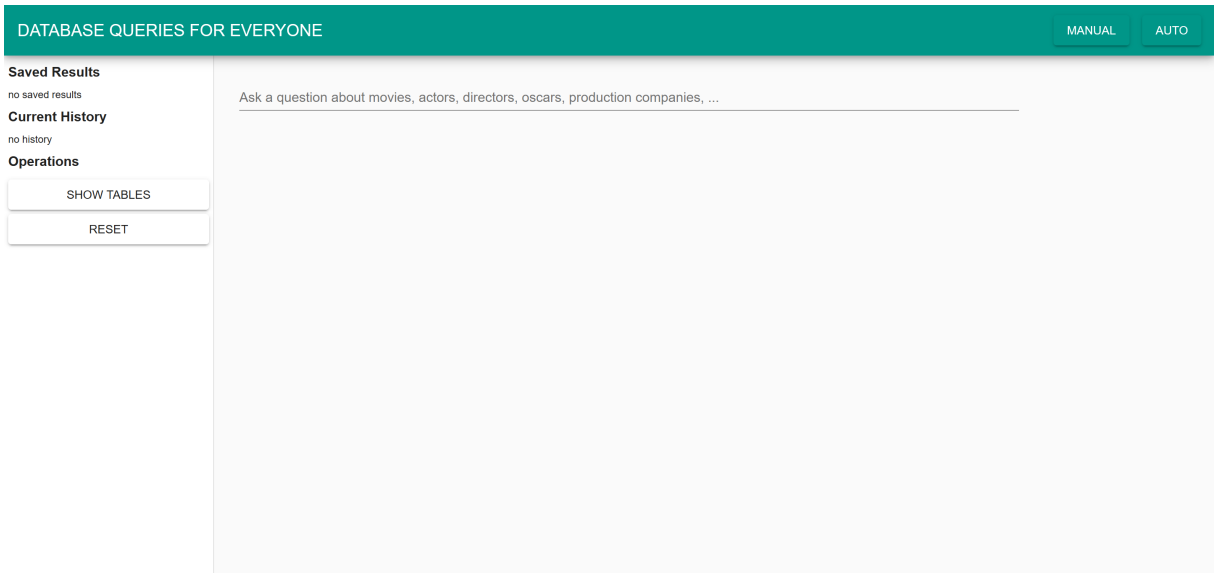

Abbildung 4.1.: Screenshot der Applikation

In Abbildung 4.1 ist ein Screenshot der Applikation ersichtlich. Oben befindet sich in grün die Navigationsbar, welche das Umschalten von manuellem Beantworten auf automatisches Ausführen bereits erstellter Logfiles ermöglicht. Auf der linken Seite befindet sich die Sidebar, in der die verschiedenen Operationen zur Verfügung gestellt werden, sowie die Suchhistorie ersichtlich ist. Der restliche Platz wird verwendet für die Eingabe der gewünschten Frage, sowie das Darstellen der Resultate.

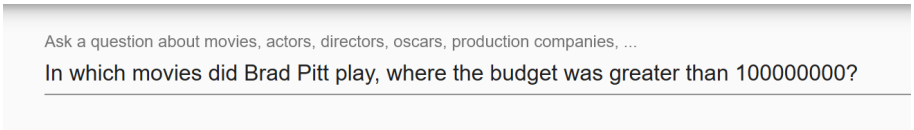

Abbildung 4.2.: Screenshot des Eingabefelds

In Abbildung 4.2 ist das Eingabefeld dargestellt, in welches der Benutzer seine Frage eingeben kann. Wurde die Frage eingegeben und mit Enter bestätigt, wird die Frage unterhalb des Eingabefeldes tokenisiert (Satzzeichen werden entfernt und nach Leerzeichen aufgesplittet) dargestellt. Der Benutzer kann anschliessend einzelne Begriffe auswählen, um mit der Abfrage zu beginnen (siehe Abbildung 4.3).

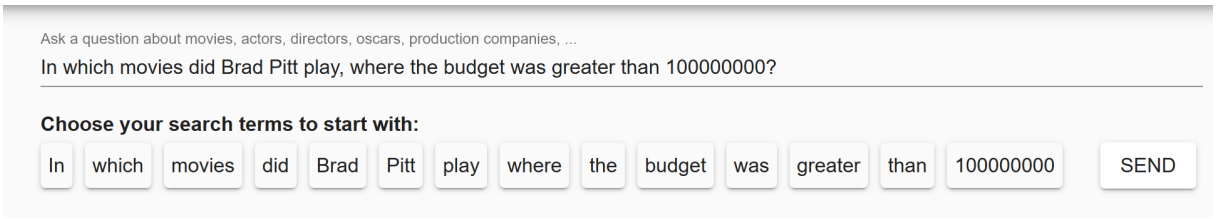

Abbildung 4.3.: Screenshot der Tokenisierung

Ein möglicher Einstieg in die Abfrage sind die Begriffe *Brad* und *Pitt*. Werden die beiden Begriffe ausgewählt und mit *SEND* bestätigt, wird der invertierte Index (siehe Kapitel 4.2) durchsucht und die Suchresultate wie in Abbildung 4.4 dargestellt. In diesem Fall wurden zwei Treffer gefunden, einmal in Tabelle *person* in der Spalte *name* und einmal in der Tabelle *cast* in der Spalte *character*. Interessant für die aktuelle Frage ist dabei der Eintrag in der Tabelle *person*.

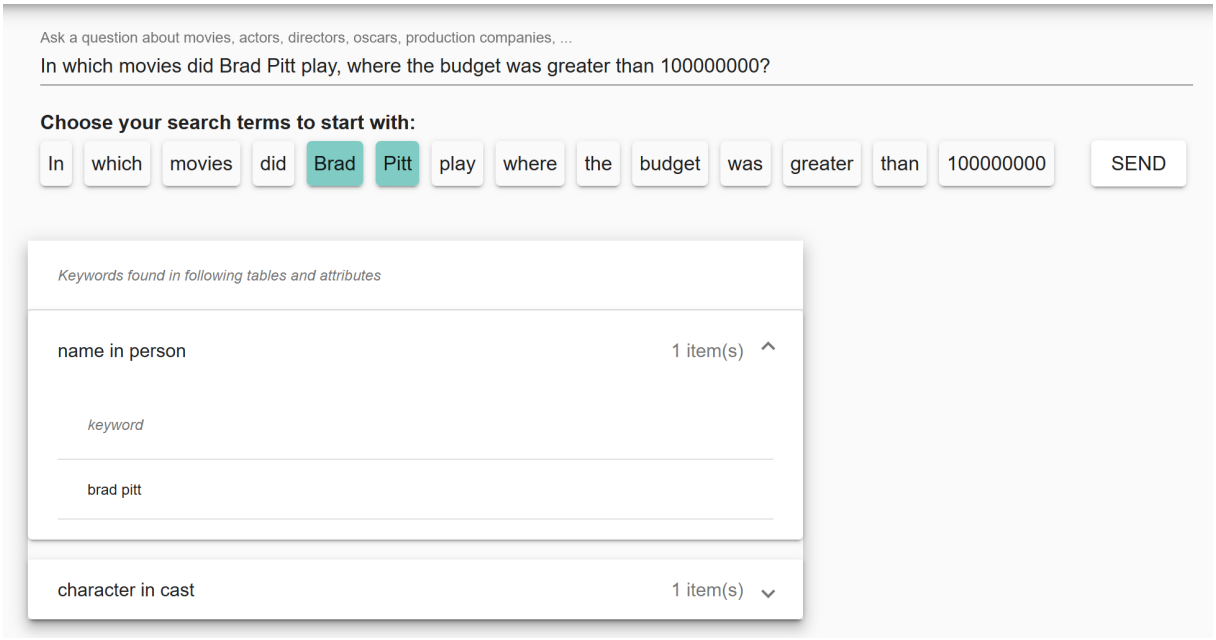

Abbildung 4.4.: Screenshot Suchergebnisse des Invertierten Index

Wählt der Benutzer nun den Eintrag in der Tabelle *person*, wird die Datenbank durchsucht, der komplette Eintrag von *brad pitt* in der Tabelle *person* zurückgegeben und wie in Abbildung 4.5 dargestellt.

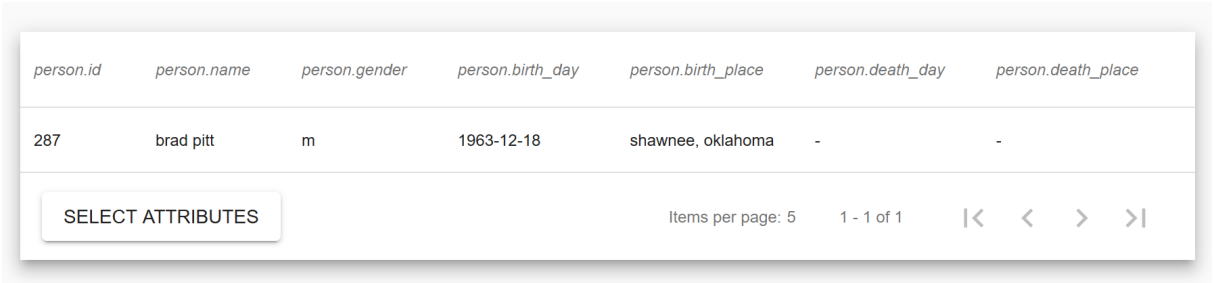

Abbildung 4.5.: Screenshot des Eintrags von Brad Pitt

Dieses Resultat kann nun zwischengespeichert werden, um es zu einem späteren Zeitpunkt wiederzuverwenden. Dies wird mit einem Klick auf den *SAVE*-Button in der Sidebar bewerkstelligt. Anschliessend ist das gespeicherte Resultat in der Sidebar unter den *Saved Results* ersichtlich (siehe Abbildung 4.6) und kann jederzeit mit einem Klick auf den Eintrag wiederhergestellt oder mit einem Klick auf den Abfalleimer gelöscht werden.

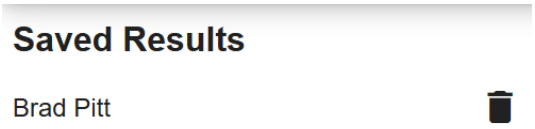

Abbildung 4.6.: Screenshot der gespeicherten Resultate

Nachdem das vorherige Zwischenresultat erfolgreich gespeichert wurde, kann nun mit einem weiteren Teil der Abfrage weitergefahren werden. Sinnvoll wäre zu diesem Zeitpunkt, alle Filme aus der Datenbank zu holen, um sie dann anschliessend mit dem gespeicherten *brad pitt* zu verbinden. Um alle Filme zu erhalten, wird dieses Mal der Token *movies* gewählt und bestätigt. Das Resultat aller Filme ist in Abbildung 4.7 ersichtlich. Es wurden 4803 Filme gefunden.

movie.id	movie.title	movie.original_title	movie.original_language	movie.budget	movie.revenue	movie.homepage	movie.overview	m
5	four rooms	four rooms	en	4000000	4300000		it's ted the bellhop's first night o…	22
11	star wars	star wars	en	11000000	775398007	http://www.starwars.com/films/s…	princess leia is captured and h…	12
12	finding nemo	finding nemo	en	94000000	940335536	http://movies.disney.com/findin…	nemo, an adventurous young cl…	85
13	forrest gump	forrest gump	en	55000000	677945399		a man with a low iq has accom…	13
14	american beauty	american beauty	en	15000000	356296601	http://www.dreamworks.com/ab/	lester burnham, a depressed s…	80

SELECT ATTRIBUTES Items per page: 5 1 - 5 of 4803

Abbildung 4.7.: Screenshot aller gefundenen Filme

Um nun mit den bereits erstellten Zwischenresultaten (*brad pitt* und *movies*) weiterarbeiten zu können, muss in der Sidebar aus der Liste der Operationen (Kapitel 4.6) die gewünschte Operation ausgewählt werden. Um die beiden Zwischenresultate miteinander zu verbinden, wird die *MERGE*-Funktion (Kapitel 4.6.2) benötigt.

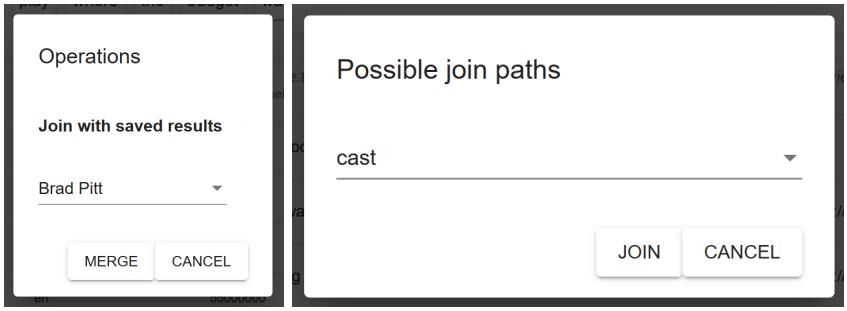

Abbildung 4.8.: Screenshot der Pop-up-Menüs der MERGE-Funktion

In Abbildung 4.8 werden die beiden Pop-up-Menüs dargestellt, welche für die *MERGE*-Funktion benötigt werden. Als Erstes muss ausgewählt werden, mit welchem der gespeicherten Zwischenresultate das aktuelle Zwischenresultat verbunden werden soll. Anschliessend wird in der Datenbank geschaut, über welche Tabellen die beiden Zwischenresultate verbunden werden können. Falls mehrere Möglichkeiten gefunden werden, muss der Benutzer im zweiten Pop-up-Menü auswählen, über welchen Pfad er die beiden Zwischenresultate gerne verbinden möchte. In diesem Beispiel wird das aktuelle Zwischenresultat (*movies*) mit dem gespeicherten Resultat von *brad pitt* über den Pfad *cast* verbunden, da die Filme gesucht werden, in denen er mitgespielt hat.

Abbildung 4.9.: Screenshot der Token-Assoziation der *MERGE*-Operation

Sobald der Benutzer die *MERGE*-Funktion ausgeführt hat, wird er mit einem weiteren Pop-up-Menü gefragt, welche Token aus seiner Frage er mit der ausgeführten Operation assoziiert. In diesem Fall könnten es die Token *did* und *play* sein (siehe Abbildung 4.9). Da die beiden Zwischenresultate, welche verbunden wurden, bereits die Token *movies*, *brad* und *pitt* enthielten, sind diese beim Erscheinen des Pop-up-Menüs bereits ausgewählt. Diese Assoziationen werden im Hintergrund abgespeichert und sind Bestandteil der Logfiles, welche zukünftig für das Machine Learning verwendet werden.

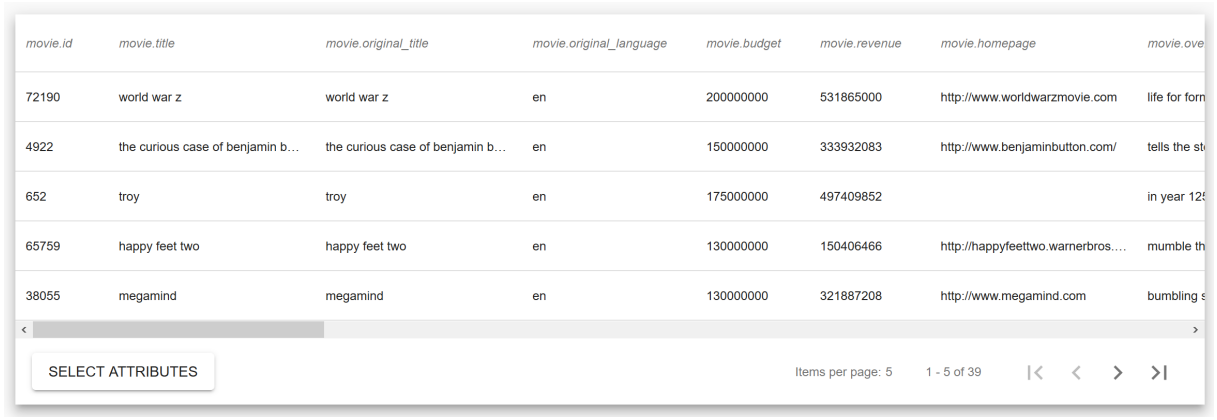

Abbildung 4.10.: Screenshot des Resultats der *MERGE*-Funktion

In Abbildung 4.10 wird das Resultat der *MERGE*-Funktion dargestellt. Die Tabelle hat nun nur noch 39 Einträge, da alle Filme, in denen Brad Pitt nicht mitgespielt hat, rausgefiltert wurden. Allerdings sind in diesem Beispiel nur die Filme interessant, deren Budget grösser als 100'000'000 Dollar war. Aus diesem Grund muss als nächste Operation die *FILTER*-Funktion (Kapitel 4.6.3) ausgewählt werden.

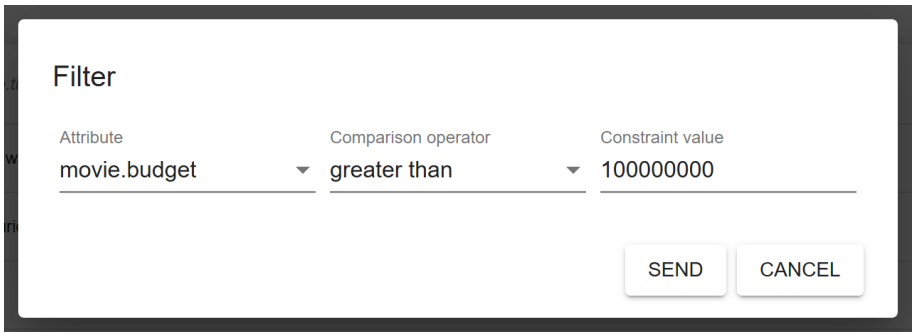

Abbildung 4.11.: Screenshot es Pop-up-Menüs der *FILTER*-Funktion

Mit einem Klick auf den *FILTER*-Button öffnet sich das in Abbildung 4.11 dargestellte Pop-up-Menü. Als Erstes muss der Benutzer das Attribut auswählen, das ihn interessiert, bzw. anhand welchem er filtern möchte. Anschliessend muss der Vergleichsoperator und der eigentliche Wert gewählt oder eingetragen werden. In diesem Beispiel wird das Attribut *movie.budget*, der Vergleichsoperator *greater than* und der Wert *100000000* ausgewählt.

Abbildung 4.12.: Screenshot der Token-Assoziation der *FILTER*-Operation

Nach dem Ausführen der *FILTER*-Operation erscheint wieder das in Abbildung 4.12 abgebildete Pop-up-Menü. Der Benutzer wählt wieder die Token aus, mit denen er die vorherige Operation assoziiert. In diesem Beispiel könnten das die Begriffe *where*, *the*, *budget*, *was*, *greater*, *than* und *100000000* sein.

Das Resultat der *FILTER*-Funktion ist in Abbildung 4.13 abgebildet. Aus den ursprünglich 39 Filmen von Brad Pitt sind nun noch sieben übrig geblieben, welche das Kriterium (Budget über 100'000'000 Dollar) erfüllen.

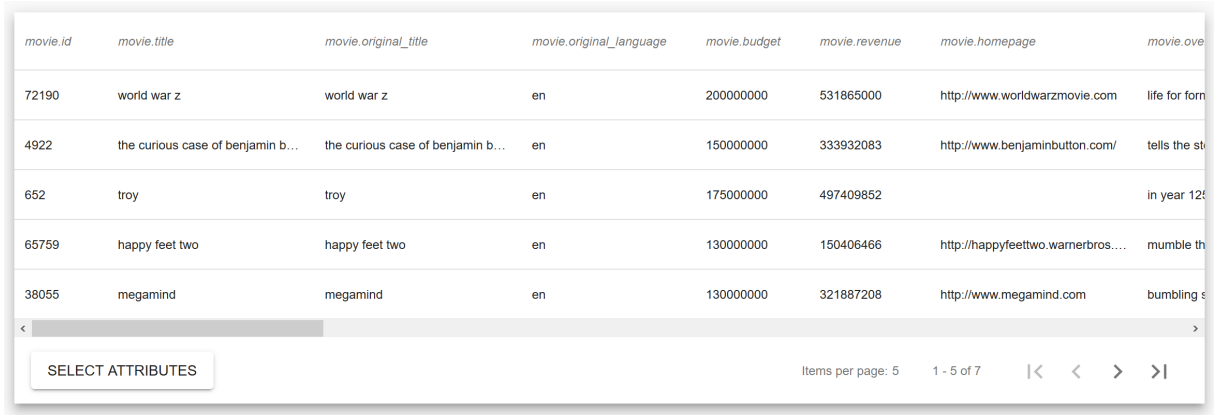

Abbildung 4.13.: Screenshot des Resultats der *FILTER*-Funktion

Da für die Beantwortung der gestellten Frage nur die Filmtitel interessant sind und die restlichen Werte der Tabelle nicht benötigt werden, wird als nächste Operation die *EXTRACT-VALUES*-Funktion (Kapitel 4.6.4) aufgerufen. Abbildung 4.14 zeigt das Pop-up-Menü, in welchem der Benutzer das Attribut auswählen kann, welches er extrahieren möchte. In diesem Beispiel wird das Attribut *movie.title* ausgewählt.

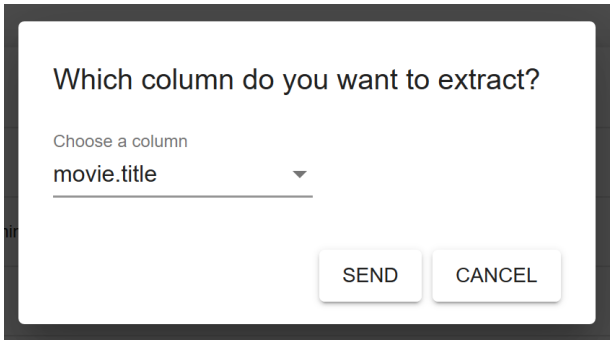

Abbildung 4.14.: Screenshot des Pop-up-Menüs der *EXTRACT-VALUES*-Funktion

Nach dem Ausführen der *EXTRACT-VALUES*-Funktion erscheint wieder das Pop-up-Menü für die Token-Assoziation (Abbildung 4.15). Die letzten beiden noch verbleibenden Token (*in* und *which*) werden ausgewählt.

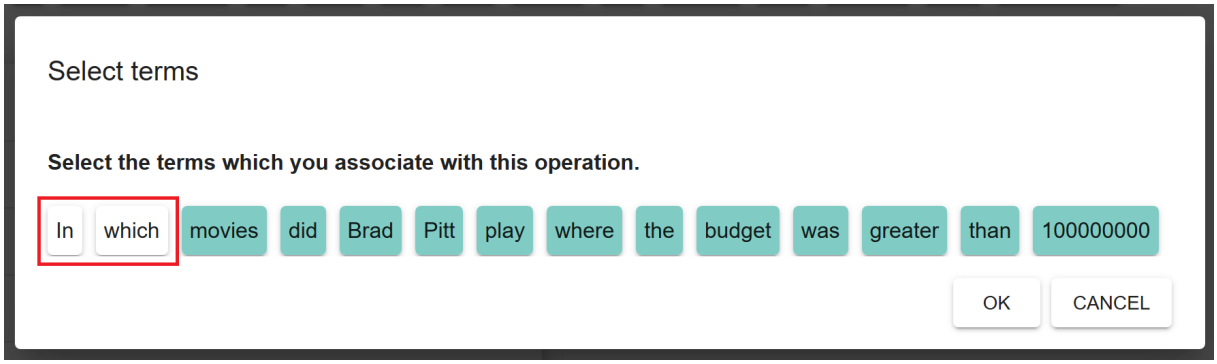

Abbildung 4.15.: Screenshot der Token-Assoziation der *EXTRACT-VALUES*-Funktion

Das Schlussresultat wird in Abbildung 4.16 gezeigt. Es enthält nur noch die sieben Filmtitel und somit die Antwort auf die gestellte Frage.

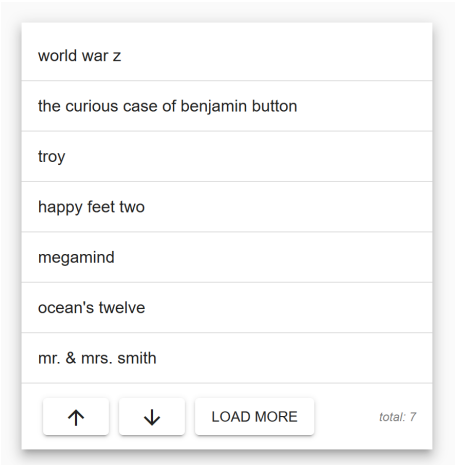

Abbildung 4.16.: Screenshot des Resultats der *EXTRACT-VALUES*-Funktion

Der Benutzer ist nun fertig mit seiner Abfrage und kann mit dem *FINISH*-Button die Abfrage beenden, bzw. gleich eine neue starten. Möchte er seine beendete Abfrage analysieren, kann er in der Navigationsbar auf *AUTO* klicken. Der Benutzer gelangt zur nächsten Seite und kann im Auswahlfeld (Abbildung 4.17) das Logfile auswählen, welches er analysieren möchte.

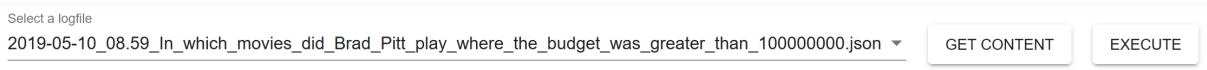

Abbildung 4.17.: Screenshot des Auswahlfeldes

Mit einem Klick auf den *GET-CONTENT*-Button wird dem Benutzer der Inhalt des JSON-Logfiles (Abbildung 4.18) und die grafische Darstellung des Logging-Baums (Abbildung 4.19) angezeigt.

Content of logfile:


```
{
    "query": "In which movies did Brad Pitt play, where the budget was greater than 100000000?",
    "successful": true,
    "operationNodes": {
        "1": {
            "arguments": {
                "table": "person"
            },
            "inputNodes": null,
            "operation": "GET DATA",
            "tokenIndices": [
                4,
                5
            ]
        },
        "2": {
            "arguments": {
                "attribute": "person.name",
                "comparisonOperator": "=",
                "value": "brad pitt"
            },
            "inputNodes": [
                1
            ],
            "operation": "FILTER",
            "tokenIndices": [
                4,
                5
            ]
        },
        "5": {
```


Abbildung 4.18.: Screenshot des JSON-Logfiles

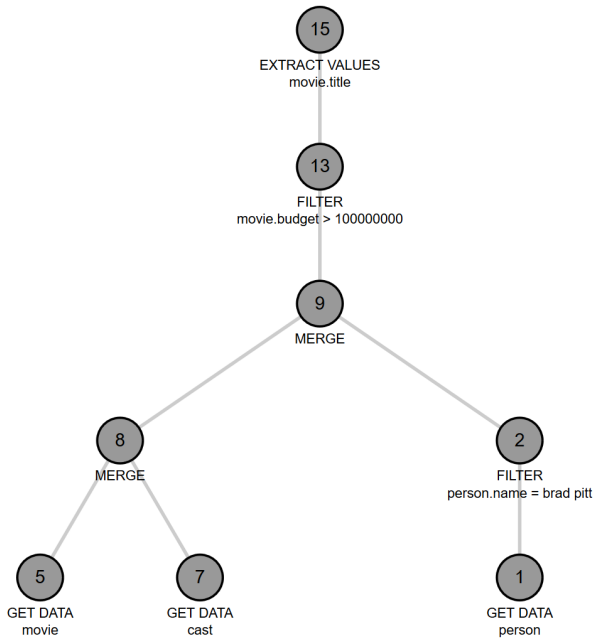

Abbildung 4.19.: Screenshot des Logging-Baums

Mit einem Klick auf den *EXECUTE*-Button wird anhand der Einträge im Logfile die Abfrage reproduziert, automatisch ausgeführt und dem Benutzer das Schlussresultat angezeigt (Abbildung 4.20).

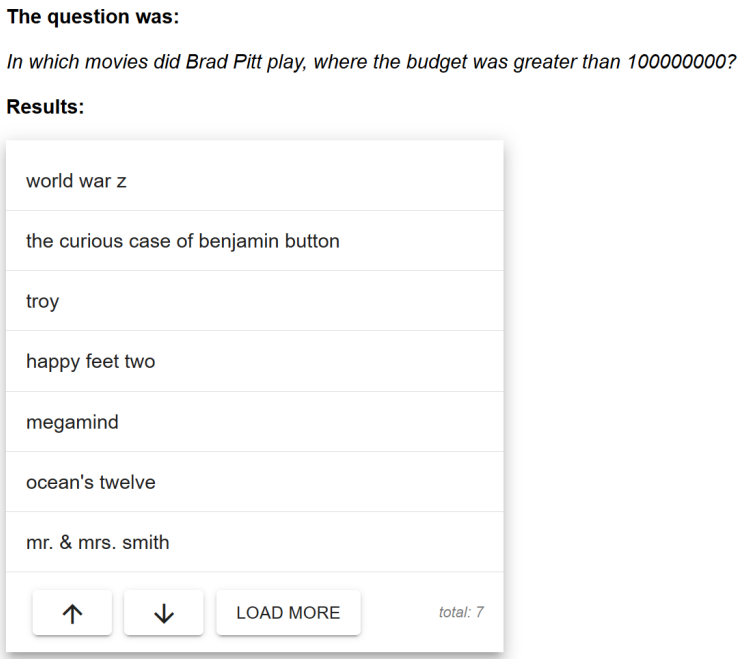

Abbildung 4.20.: Screenshot des Schlussresults

4.2. Invertierter Index

4.2.1. Erstellen des invertierten Index

„Damit der Benutzer einen geeigneten Einstiegspunkt in die Datenbank erhält, muss er die Datenbank nach Begriffen durchsuchen können. Damit die Suche über eine Datenbank nicht zu lange dauert, wurde ein invertierter Index von der Datenbank erstellt. Dieser ermöglicht eine Art Volltextsuche über die kompletten Basisdaten sowie die Metadaten (Attributnamen und Tabellen der Datenbank). “ [7]

Der invertierte Index besteht dabei aus drei Tabellen:

Tabellennamen

Die Namen aller Tabellen des Datenbankschemas werden hier gespeichert. Diese Tabelle des invertierten Index sagt aus, welche Datenbanktabellen der Applikation zur Verfügung stehen.

Beispiel:

table
movie
person
...

Attributnamen

Zu allen Tabellennamen werden die zugehörigen Attributnamen gespeichert. Diese Tabelle sagt aus, welche Tabelle welche Attribute enthält.

Beispiel:

attribute	**table**
title	movie
revenue	movie
name	person
...	...

Attributwerte

Für jeden Wert, der in irgendeiner Datenbanktabelle gespeichert ist, wird festgehalten, in welcher Tabelle in welchem Attribut er zu finden ist. Die Werte wurden unverändert, das heisst ohne Stemming, in die Index-Tabelle eingetragen.

Beispiel:

value	**attribute**	**table**
inception	title	movie
100000000	revenue	movie
brad pitt	name	person
...	...	...

„Der Aufbau des invertierten Index wird in MySQL durchgeführt. Der Grund dafür ist, dass das Aktualisieren des Index mit MySQL einfacher vollzogen werden kann. In SQL können dafür Trigger implementiert werden, welche ausgeführt werden, sobald etwas an der Datenbank geändert wird. “ [7] Die Aktualisierung des invertierten Index war kein Teil der Arbeit und wurde dementsprechend noch nicht umgesetzt. Die verschiedenen Tabellen des invertierten Index werden in Stored Procedures von MySQL erstellt.

4.2.2. Verwendung in der Applikation

Veranschaulicht wird dies anhand der Frage aus Kapitel 4.1:
In which movies did Brad Pitt play, where the budget was greater than 100000000?

Wählt der Benutzer beispielsweise die Token {*brad*, *pitt*, *movies*}, wird der invertierte Index wie folgt durchsucht:

Die *Attributwerte*-Tabelle wird zuerst mit der Konkatenation der drei Suchbegriffe durchsucht. Falls für {*brad, pitt, movies*} kein Eintrag gefunden wird, wird anschliessend die Länge der Kombination auf zwei verringert (ergibt die Kombinationen {*brad, pitt*}, {*brad, movies*} und {*pitt, movies*}). Die Reihenfolge der Begriffe wird jedoch beibehalten. In diesem Beispiel würde bei der Kombination {*brad, pitt*} ein Eintrag in der Tabelle *person* im Attribut *name* gefunden werden, worauf die Länge der Kombinationen nicht noch weiter verringert wird. Als Nächstes wird die *Tabellennamen*- und die *Attributnamen*-Tabelle durchsucht. Diese beiden Tabellen werden direkt

mit Kombinationslängen von eins, das heisst mit den einzelnen Token durchsucht. Die Tabellen werden dabei von unveränderten Token und singularisierten Token durchsucht, beispielsweise von *movies* und *movie*. Durch den Suchbegriff *movie* werden in diesem Beispiel Resultate wie *movie* als Tabellenname oder *movie_id* als Attributname in der Tabelle *cast* gefunden. Als Treffer gilt, wenn der Suchbegriff ein Substring des Tabelleneintrags darstellt.

Die Suche durch den invertierten Index wird nie in einem Logfile festgehalten, da dies eine reine Hilfestellung für den Benutzer ist, wenn er eine Frage manuell beantwortet. Wenn der Benutzer in der Applikation einen Treffer aus der *Tabellennamen*- oder der *Attributnamen*-Tabelle auswählt, wird eine *GET-DATA*-Operation (Kapitel 4.6.1) ausgelöst. Wählt der Benutzer allerdings einen Eintrag aus der *Attributwerte*-Tabelle, werden im Hintergrund gleich zwei Operationen ausgeführt, nämlich die *GET-DATA*-Operation und die *FILTER*-Operation (Kapitel 4.6.3), worauf der spezifische Eintrag direkt herausgefiltert wird.

4.3. Datenbankschema

„Für verschiedene Funktionen, welche das Tool zur Verfügung stellt, müssen Informationen über das Schema der Datenbank vorhanden sein. Dazu wird im Backend ein ungerichteter Graph aufgebaut. Die Informationen, welche für den Graphen benötigt werden, werden aus Tabellen vom Metadaten-Schema INFORMATION_SCHEMA gelesen, welches in jeder MySQL-Datenbank automatisch von MySQL aufgebaut und verwaltet wird [8]. Der Graph kann für eine beliebige MySQL-Datenbank aufgebaut werden, damit das Tool unabhängig von einer spezifischen Datenbank bleibt. Die Graphstruktur wird mithilfe der Python-Library networkx [9] verwaltet.“ [7]
Im Graph wird für jede Tabelle ein Knoten erstellt. Bei jedem Knoten werden die Attribute der Tabelle abgespeichert. Zusätzlich wird abgespeichert, auf welche Attribute die Primär- und Fremdschlüssel einer Tabelle (eines Knotens im Graph) zeigen. Die Kanten werden anhand der Primär-/Fremdschlüssel-Beziehungen aufgebaut. Anhand dieser Kanten werden beispielsweise mögliche Verbindungspfade für die *MERGE*-Funktion (Kapitel 4.6.2) gesucht und dem Benutzer vorgeschlagen.

4.4. Datenbankgraph

Die Applikation stellt eine Funktion zur Verfügung, mit der das Schema der aktuell verwendeten Datenbank als Graphstruktur angezeigt werden kann. Dieser Graph kann jederzeit durch den Benutzer eingeblendet werden. Dies soll es dem Benutzer ermöglichen, sich in kurzer Zeit einen Überblick über die Datenbank zu verschaffen, indem er herauslesen kann, welche Tabellen mit welchen anderen verbunden sind und wie die Attribute einer Tabelle heissen. In Abbildung 4.21 wird gezeigt, wie Tabellen im Graph markiert werden, welche sich im aktuellen Zwischenresultat befinden. Ebenso werden alle Verbindungspfade rot markiert. Als Beispiel wird die Frage aus Kapitel 4.1 übernommen:
”In which movies did Brad Pitt play, where the budget was greater than 100000000?”

Es wird angenommen, der Benutzer möchte sich den Datenbankgraphen zu dem Zeitpunkt ansehen, nachdem *Brad Pitt* mit allen Filmen via der Tabelle *cast* verbunden wurde. Da sich nun im Zwischenresultat die Tabellen *person*, *cast* und *movie* befinden, werden sie im Graphen rot markiert. Dies ermöglicht dem Benutzer, sich besser zu orientieren und die nächsten Operationen zu planen.

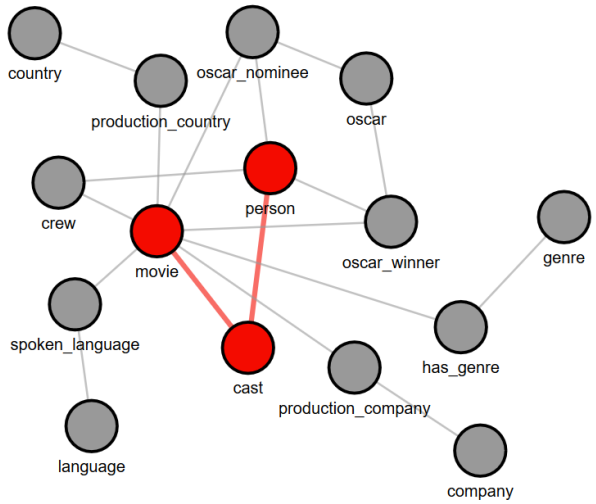

Abbildung 4.21.: Der Datenbankgraph während der Beantwortung der Frage *"In which movies did Brad Pitt play, where the budget was greater than 100000000?"*

Der Graph wird automatisch ausgerichtet, so dass sich die Knoten nicht überlappen. Dem Benutzer wird zusätzlich die Möglichkeit geboten, die einzelnen Knoten nach Belieben von Hand zu verschieben.

In Abbildung 4.22 sieht man, wie die Details einer Tabelle angezeigt werden können. Dazu muss der Benutzer lediglich mit dem Mauszeiger über einen Knoten fahren. Die Details beinhalten den Tabellennamen sowie alle zugehörigen Attribute dieser Tabelle.

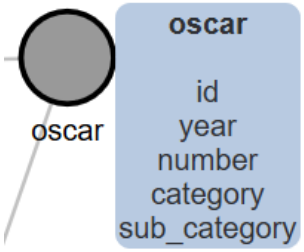

Abbildung 4.22.: Detailansicht der Tabelle *oscar*

4.5. Knotentypen

Da nicht mit allen Knoten gleich umgegangen werden kann, respektive nicht alle Knoten die selben Attribute enthalten, wurden für die Umsetzung verschiedene Knotentypen definiert. Insgesamt wird in der Applikation zwischen acht verschiedene Knotentypen unterschieden. Die Vererbungshierarchie dieser Knotentypen ist in Abbildung 4.23 dargestellt.

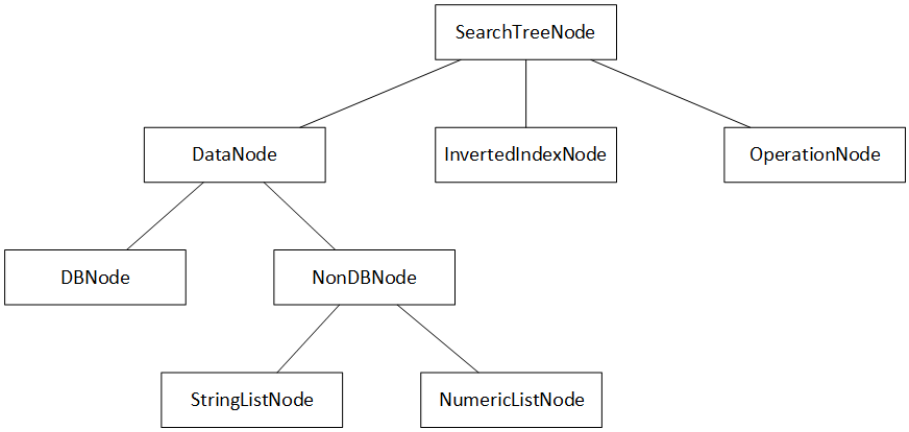

Abbildung 4.23.: Die Knotenhierarchie, wie sie in der Applikation verwendet wird.

Der Basisknotentyp ist der *SearchTreeNode*. Jeder dieser Knoten wird in der Baumstruktur der Applikation abgespeichert, weswegen dieser Basistyp grundlegende Eigenschaften wie z.B. die ID eines Knotens bereitstellt.

4.5.1. Datenknoten

Die in Abschnitt 3.4 beschriebenen SemQL-Operationen können eine Vielzahl von verschiedenen Outputs generieren. Um mit all diesen unterschiedlichen Outputs besser umgehen zu können, wurden verschiedene Knotentypen für Datenknoten definiert. Ein *DataNode* ist dabei eine Art Behälter für Daten, welche von einer Operation erzeugt werden.

4.5.1.1. DBNode

Der *DBNode* ist ein Datenknoten, der Informationen über die Datenbank abgespeichert hat. Zu diesen Informationen gehören die Tabellen, von denen er Datensätze beinhaltet sowie die dazugehörigen Attribute. Der Vorteil davon ist, dass mit Hilfe dieser Unterscheidung Operationen getätigt werden können, welche genau diese Informationen über den Kontext der darunterliegenden Datenbank benutzen. Dazu gehört zum Beispiel die *MERGE*-Operation (Kapitel 4.6.2).

4.5.1.2. NonDBNode

Der Typ *NonDBNode* hat im Gegensatz zum *DBNode* keine Informationen über die Datenbank mehr gespeichert. Dadurch sind alle Operationen, welche Informationen über die Datenbank benötigen, nicht auf diesen Knotentypen anwendbar. Das bedeutet, ein *NonDBNode* beinhaltet auch keine Attribute mehr, er ist lediglich noch eine Liste von Werten. Um eine Abfrage abschliessen zu können, muss das aktuelle Zwischenresultat aus einem *NonDBNode* bestehen. Dies wurde so definiert, da eine Antwort auf eine Frage immer aus den Werten genau einer Tabellenspalte besteht. Ein Beispiel, wie ein Knoten vom Typ *NonDBNode* erstellt werden kann, ist die *EXTRACT-VALUES*-Operation (Kapitel 4.6.4).

Als Subklassen des *NonDBNodes* existieren folgende zwei:

NumericListNode

Der Typ *NumericListNode* beinhaltet eine Liste von n numerischen Werten.

StringListNode

Der Typ *StringListNode* beinhaltet eine Liste von n alphanumerischen Werten.

Die Unterscheidung zwischen *NumericListNode* und *StringListNode* wird gemacht, da es Operationen gibt, welche nur auf numerischen Werten Sinn machen, beziehungsweise auf alphanumerischen Werten nicht definiert sind, wie beispielsweise die *AVERAGE*-Funktion (Kapitel 4.6.6.4), welche den Durchschnitt berechnet.

4.5.2. InvertedIndexNode

Der *InvertedIndexNode* ist ein spezieller Typ von Knoten. Er wird nur benötigt, wenn der Benutzer einen neuen Teilbaum erstellen möchte, sprich wenn die Datenbank mit Hilfe des invertierten Index durchsucht wird. Speziell ist ausserdem, dass die Operation, die zu diesem Knoten geführt hat, nicht geloggt wird. Der Grund dafür ist, dass diese Information für den Machine-Learning-Algorithmus nicht von Relevanz ist. Im Gegensatz zu einem Menschen, der den gewünschten Eintrag in einer Datenbank zuerst finden muss, kann ein Machine-Learning-Algorithmus direkt darauf trainiert werden, anhand von den zur Verfügung stehenden Tokens den richtigen Datensatz aus der Datenbank zu selektieren.

4.5.3. OperationNode

Ein *OperationNode* wird erstellt, sobald der Benutzer eine Operation ausführt. Der *OperationNode* hat als Input eine Liste von Referenzen auf Datenknoten. Als Output hat er eine Referenz auf den Output-Datenknoten. Für das Logging werden schlussendlich nur die Knoten vom Typ OperationNode genutzt. Die genaue Funktionsweise des Loggings ist in Kapitel 4.9 beschrieben.

4.6. Implementierte atomare Operationen

4.6.1. Get Data

Mit der *GET-DATA*-Operation wird ein Datensatz aus der Datenbank geholt. Es ist die einzige Operation, die direkt von der Datenbank abhängig ist, sprich auf der Datenbank ausgeführt wird. Als Argument nimmt die Operation einen Tabellennamen an. Als Resultat ensteht ein DBNode, welcher alle Einträge der gewünschten Tabelle enthält. Die Funktion der Operation wird anhand der Beispielfrage aus Kapitel 4.1 genauer erklärt:

In which movies did Brad Pitt play, where the budget was greater than 100000000?

Zu Beginn der Abfrage könnte der Benutzer an allen Filmen der Datenbank interessiert sein, um sie anschliessend weiter filtern zu können. Aus diesem Grund wählt der Benutzer die *GET-DATA*-Operation mit *movie* als Argument. Die resultierende Tabelle könnte dabei wie folgt aussehen:

movie.id	movie.title	movie.budget	movie.revenue
1	inception	75'000'000	500'000'000
2	titanic	100'000'000	800'000'000
3	the godfather	50'000'000	300'000'000
...	...	...	...

4.6.2. Merge

Mit der *MERGE*-Operation können zwei DBNodes miteinander verbunden werden. Dabei ist es nicht zwingend nötig, dass die in den Zwischenresultaten vorhandenen Tabellen eine direkte Verbindung haben. Da das Datenbankschema im Hintergrund als Graph aufgebaut ist, wird mit Hilfe des Algorithmus von Dijkstra der kürzeste Weg im Graph gesucht und dem Benutzer vorgeschlagen. Sollte es mehrere verschiedene Pfade der gleichen Länge geben, muss der Benutzer manuell auswählen, welcher Pfad für ihn interessant ist. Als Input nimmt die Operation folgende Argumente an:

- DBNode: das aktuelle Zwischenresultat
- DBNode: das zu verbindende Zwischenresultat
- Pfad: der Verbindungspfad (Liste von Tabellennamen)

Als Resultat der *MERGE*-Operation entsteht wieder ein DBNode. Im Grunde ähnelt die *MERGE*-Operation dem *join* von SQL. Um die Operation zu veranschaulichen, wird die Frage aus Kapitel 4.1 übernommen:

*In which movies **did** Brad Pitt **play**, where the budget was greater than 100000000?*

Es wird angenommen, dass folgende Tabelle als aktuelles Zwischenresultat vorliegt:

person.id	person.name
4711	brad pitt

Zudem sei in den gespeicherten Resultaten folgendes Zwischenresultat gespeichert:

movie.id	movie.title
1	ocean's eleven
2	spiderman
3	star wars

Nun sollen die beiden Tabellen durch die *MERGE*-Operation miteinander verbunden werden. Als Argumente wird neben den beiden Tabellen folgender Pfad mitgegeben: [cast]
Der Pfad über die Tabelle *cast* ist für dieses Beispiel interessant, da gefordert wird, dass Brad Pitt im Film mitgespielt hat und nicht etwa Regisseur oder Ähnliches war.

person.id	person.name	cast.pid	cast.character	cast.id	cast.mid	movie.id	movie.title
4711	brad pitt	4711	robert "rusty" ryan	123	1	1	ocean's eleven

Im Resultat ist nun ersichtlich, dass die beiden Tabellen miteinander über die Zwischentabelle *cast* verbunden wurden. Die Filme, in denen Brad Pitt nicht mitgespielt hat, sind bei der Verbindung herausgefiltert worden.

4.6.3. Filter

Mit der *FILTER*-Operation können Einträge im Zwischenresultat nach einer gewünschten Bedingung gefiltert werden. Als Input nimmt die Operation folgende Argumente an:

- DBNode: das aktuelle Zwischenresultat
- Attribut: das Attribut, welches gefiltert werden soll
- Vergleichsoperator: ein Element aus $\{<, \leq, \geq, >, =, \neq, \text{BETWEEN}\}$
- Wert: der gewünschte Wert für die Bedingung (bei *BETWEEN* zwei Werte)

Als Resultat der *FILTER*-Operation entsteht ebenfalls wieder ein DBNode. Um die Operation an einem Beispiel zu veranschaulichen, wird die Frage aus Kapitel 4.1 übernommen:

In which movies did Brad Pitt play, ***where the budget was greater than 100000000?***

Es wird angenommen, dass das aktuelle Zwischenresultat wie folgt aussieht:

movie.id	**movie.title**	**movie.budget**
1	ocean's eleven	80'000'000
2	ocean's twelve	90'000'000
3	12 years a slave	110'000'000

Nun wird die *FILTER*-Operation aufgerufen mit dem aktuellen Zwischenresultat als DBNode und den folgenden weiteren Parametern als Input:
{Attribut: movie.budget, Vergleichsoperator: >, Wert: 100'000'000}

movie.id	**movie.title**	**movie.budget**
3	12 years a slave	110'000'000

Im Resultat der *FILTER*-Operation ist nun nur noch *12 years a slave* vorhanden, da dieser Film als einziger ein Budget über 100'000'000 Dollar hat.

4.6.4. Extract Values

Die *EXTRACT-VALUES*-Operation wird benötigt, um die Werte eines gewünschten Attributs aus einer Tabelle zu extrahieren. Dies wird oft als letzte Operation einer Abfrage genutzt, da in den meisten Fällen als Schlussresultat nur die Einträge einer Tabellenspalte relevant sind. Die Operation nimmt als Input das aktuelle Zwischenresultat, welches ein DBNode sein muss, zusammen mit dem zu extrahierenden Attribut an. Als Resultat der *EXTRACT-VALUES*-Operation

entsteht entweder ein NumericListNode, oder ein StringListNode, je nach dem ob die extrahierten Werte numerisch oder alphanumerisch sind.

Um die Funktion der *EXTRACT-VALUES*-Operation zu veranschaulichen, wird die Frage aus Kapitel 4.1 übernommen:

***In which** movies did Brad Pitt play, where the budget was greater than 100000000?*

Es wird angenommen, dass folgende Tabelle das aktuelle Zwischenresultat darstellt.

person.id	person.name	cast.pid	cast.character	cast.id	cast.mid	movie.id	movie.title
4711	brad pitt	4711	robert "rusty" ryan	123	1	1	ocean's eleven
4711	brad pitt	4711	robert "rusty" ryan	123	2	2	ocean's twelve
4711	brad pitt	4711	ben rickert	154	3	3	the big short

Da zur Beantwortung der Frage in diesem Beispiel nur die Filmtitel benötigt werden, wird die *EXTRACT-VALUES*-Operation mit dem aktuellen Zwischenresultat als DBNode und *movie.title* als Attribut aufgerufen.

ocean's eleven
ocean's twelve
the big short

Im Resultat sind nun nur noch wie gewünscht die Filmtitel enthalten. Da Filmtitel alphanumerisch sind, wird das Ergebnis in einem StringListNode gespeichert.

4.6.5. Duplikatentfernung

Mit der *REMOVE-DUPLICATES*-Operation können Duplikate aus einer Liste von Strings oder Zahlen entfernt werden. Als Input akzeptiert die Operation dementsprechend nur einen StringListNode oder NumericListNode. Die Funktion der *REMOVE-DUPLICATES*-Operation, sowie man sie einsetzt, wird an einem Beispiel erläutert:

Die Beispielfrage lautet wie folgt:
*In which **different** movies did Christian Bale play?*

Es wird angenommen, dass die Abfrage zusammengestellt wurde und mit der *EXTRACT-VALUES*-Operation (Kapitel 4.6.4) die Filmtitel extrahiert wurden. Da Christian Bale beispielsweise in *batman begins* die beiden Charaktere *bruce wayne* und *batman* gespielt hat, könnte das Zwischenresultat wie folgt aussehen:

batman begins
batman begins
the big short

Da nach den verschiedenen Filmen gefragt wurde, wird die *REMOVE-DUPLICATES*-Operation aufgerufen mit der Filmtitelliste als Input. Die Duplikate werden entfernt und folgende Tabelle

wird erneut in einem StringListNode gespeichert.

batman begins
the big short

4.6.6. Aggregatsoperationen

4.6.6.1. Min

Mit der *MIN*-Operation wird der kleinste Wert aus einer Liste von numerischen Werten ausgegeben. Die Operation nimmt als Input daher einen NumericListNode an und erzeugt als Output erneut einen NumericListNode.

4.6.6.2. Max

Mit der *MAX*-Operation wird der grösste Wert aus einer Liste von numerischen Werten ausgegeben. Die Operation nimmt als Input daher einen NumericListNode an und erzeugt als Output erneut einen NumericListNode.

4.6.6.3. Summe

Mit der *SUM*-Operation wird die Summe der Werte aus einer Liste von numerischen Werten ausgegeben. Die Operation nimmt als Input daher einen NumericListNode an und erzeugt als Output erneut einen NumericListNode.

4.6.6.4. Durchschnitt

Mit der *AVERAGE*-Operation wird der Durchschnitt der Werte aus einer Liste von numerischen Werten ausgegeben. Die Operation nimmt als Input daher einen NumericListNode an und erzeugt als Output erneut einen NumericListNode.

4.6.6.5. Count

Mit der *COUNT*-Operation wird die Anzahl der Einträge einer Liste oder Tabelle ausgegeben. Die Operation nimmt als Input daher einen DBNode, StringListNode oder NumericListNode an und erzeugt als Output erneut einen NumericListNode.

Um dies zu veranschaulichen wird folgende Frage als Beispiel verwendet:

In how many movies did Brad Pitt play?

Es wird angenommen, dass die folgende Tabelle das aktuelle Zwischenresultat darstellt:

person.id	person.name	cast.pid	cast.character	cast.id	cast.mid	movie.id	movie.title
4711	brad pitt	4711	robert "rusty" ryan	123	1	1	ocean's eleven
4711	brad pitt	4711	robert "rusty" ryan	123	2	2	ocean's twelve
4711	brad pitt	4711	ben rickert	154	3	3	the big short

Nun wird die *COUNT*-Operation aufgerufen. Da die Input-Tabelle aus drei Zeilen besteht, sieht das Resultat wie folgt aus:

3

4.6.7. Mengenoperationen

Grundsätzlich werden in der Applikation die folgenden drei Mengenoperationen angeboten:

- Vereinigungsmenge
- Schnittmenge
- Differenz

Es wird allerdings unterschieden, ob der Input aus zwei DBNodes, oder aus zwei NonDBNodes besteht.

4.6.7.1. Mengenoperationen mit DBNodes

Vereinigungsmenge

Die *UNION*-Operation vereinigt zwei Zwischenresultate zu einem. Hier werden zwei DBNodes miteinander vereinigt. Wichtig ist dabei, dass die beiden DBNodes die exakt gleichen Attribute enthalten. Ist dies nicht der Fall, kann die Operation nicht durchgeführt werden und der Benutzer erhält eine Fehlermeldung.

Als Beispiel wird folgende Frage gewählt: *In which movies did Brad Pitt* ***or*** *George Clooney play?*

Es wird angenommen, dass das aktuelle Zwischenresultat wie folgt aussieht:

person.id	person.name	cast.pid	cast.character	cast.id	cast.mid	movie.id	movie.title
4711	brad pitt	4711	samuel bass	137	2	2	12 years a slave
4711	brad pitt	4711	ben rickert	154	3	3	the big short

Zudem sieht das gespeicherte Zwischenresultat, welches man mit dem Aktuellen vereinigen möchte, so aus:

person.id	person.name	cast.pid	cast.character	cast.id	cast.mid	movie.id	movie.title
7382	george clooney	7382	matt kowalsky	123	1	1	gravity
7382	george clooney	7382	jack	287	4	4	the american

Nun wird die *UNION*-Operation ausgeführt und man erhält die Vereinigungsmenge der beiden Zwischenresultate:

person.id	person.name	cast.pid	cast.character	cast.id	cast.mid	movie.id	movie.title
4711	brad pitt	4711	samuel bass	137	2	2	12 years a slave
4711	brad pitt	4711	ben rickert	154	3	3	the big short
7382	george clooney	7382	matt kowalsky	123	1	1	gravity
7382	george clooney	7382	jack	287	4	4	the american

Schnittmenge

Die *INTERSECT*-Operation bildet die Schnittmenge aus zwei Zwischenresultaten. Wichtig ist dabei, dass die beiden DBNodes die exakt gleichen Attribute enthalten. Ist dies nicht der Fall, kann die Operation nicht durchgeführt werden und der Benutzer erhält eine Fehlermeldung. Zusätzlich zu den beiden DBNodes wird ein Attribut mitgegeben, welches verglichen werden soll. Dieses Attribut wird vom Benutzer ausgewählt.

Die Funktion der *INTERSECT*-Operation wird anhand der folgenden Beispielfrage verdeutlicht: *In which movies did Brad Pitt and George Clooney play* ***together****?*

Es wird angenommen, dass das aktuelle Zwischenresultat folgendermassen aussieht:

person.id	person.name	cast.pid	cast.character	cast.id	cast.mid	movie.id	movie.title
4711	brad pitt	4711	robert "rusty" ryan	123	1	1	ocean's eleven
4711	brad pitt	4711	robert "rusty" ryan	123	2	2	ocean's twelve
4711	brad pitt	4711	ben rickert	154	3	3	the big short

Zudem befindet sich in den gespeicherten Resultaten folgender DBNode:

person.id	person.name	cast.pid	cast.character	cast.id	cast.mid	movie.id	movie.title
7382	george clooney	7382	danny ocean	137	1	1	ocean's eleven
7382	george clooney	7382	danny ocean	137	2	2	ocean's twelve
7382	george clooney	7382	jack	287	4	4	the american

Nun wird die *INTERSECT*-Operation aufgerufen mit den beiden Tabellen und dem Attribut *movie.id* als Input. Das Attribut *movie.id* wird gewählt, da dies der eindeutige Key der Tabelle *movie* ist und die Filme der beiden Schauspieler miteinander verglichen werden sollen. Wird eine Film-ID in beiden Zwischenresultaten gefunden, wird die Zeile aus dem ersten Zwischenresultat in das Ergebnis übernommen. Die Schnittmenge als Resultat der Operation sieht wie folgt aus:

person.id	person.name	cast.pid	cast.character	cast.id	cast.mid	movie.id	movie.title
4711	brad pitt	4711	robert "rusty" ryan	123	1	1	ocean's eleven
4711	brad pitt	4711	robert "rusty" ryan	123	2	2	ocean's twelve

Differenz

Die *DIFFERENCE*-Operation bildet die Differenz zweier Zwischenresultate. Wichtig ist dabei, dass die beiden DBNodes die exakt gleichen Attribute enthalten. Ist dies nicht der Fall, kann die Operation nicht durchgeführt werden und der Benutzer erhält eine Fehlermeldung. Zusätzlich zu den beiden DBNodes wird ein Attribut mitgegeben, welches verglichen werden soll. Dieses Attribut wird vom Benutzer ausgewählt.

Als Beispiel wird folgende Frage gewählt:
In which movies did Brad Pitt play, but George Clooney didn't?

Es wird angenommen, das aktuelle Zwischenresultat sehe folgendermassen aus:

person.id	person.name	cast.pid	cast.character	cast.id	cast.mid	movie.id	movie.title
4711	brad pitt	4711	robert "rusty" ryan	123	1	1	ocean's eleven
4711	brad pitt	4711	robert "rusty" ryan	123	2	2	ocean's twelve
4711	brad pitt	4711	ben rickert	154	3	3	the big short

Zudem befindet sich in den gespeicherten Resultaten folgender DBNode:

person.id	person.name	cast.pid	cast.character	cast.id	cast.mid	movie.id	movie.title
7382	george clooney	7382	danny ocean	137	1	1	ocean's eleven
7382	george clooney	7382	danny ocean	137	2	2	ocean's twelve
7382	george clooney	7382	jack	287	4	4	the american

Nun wird die *DIFFERENCE*-Operation aufgerufen mit den beiden Tabellen und dem Attribut *movie.id* als Input. Das Attribut *movie.id* wird gewählt, da dies der eindeutige Key der Tabelle *movie* ist und die Filme der beiden Schauspieler miteinander verglichen werden sollen. Wird eine Film-ID in der Tabelle des zweiten DBNodes ebenfalls in der Tabelle des ersten DBNodes gefunden, wird sie aus dem Ergebnis gestrichen. Die Differenz als Resultat der Operation sieht wie folgt aus:

person.id	person.name	cast.pid	cast.character	cast.id	cast.mid	movie.id	movie.title
4711	brad pitt	4711	ben rickert	154	3	3	the big short

4.6.7.2. Mengenoperationen mit NonDBNodes

Vereinigungsmenge

Die *UNION*-Operation bildet wie die *UNION*-Operation aus Kapitel 4.6.7.1 die Vereinigungsmenge aus zwei Zwischenresultaten. Der Unterschied ist hierbei, dass die Operation als Input nicht zwei DBNodes, sondern zwei StringListNodes, respektive zwei NumericListNodes annimmt. Diese Operation wird benötigt, wenn der Benutzer bereits die Werte eines Attributs extrahiert und somit keinen DBNode als Zwischenresultat mehr gespeichert hat.

Als Beispiel wird folgende Frage gewählt:
In which movies did Brad Pitt ***or*** *George Clooney play?*

Es wird angenommen, dass das aktuelle Zwischenresultat die Filme von Brad Pitt darstellt und wie folgt aussieht:

12 years a slave
the big short

Zudem sieht das gespeicherte Zwischenresultat, welches man mit dem Aktuellen vereinigen möchte (Filme von George Clooney), so aus:

gravity
the american

Nun wird die *UNION*-Operation ausgeführt und man erhält die Vereinigungsmenge der beiden Zwischenresultate:

12 years a slave
the big short
gravity
the american

Schnittmenge

Die *INTERSECT*-Operation bildet wie die *INTERSECT*-Operation aus Kapitel 4.6.7.1 die Schnittmenge aus zwei Zwischenresultaten. Der Unterschied ist hierbei, dass die Operation als Input nicht zwei DBNodes, sondern zwei StringListNodes, respektive zwei NumericListNodes annimmt. Diese Operation wird benötigt, wenn der Benutzer bereits die Werte eines Attributs extrahiert und somit kein DBNode als Zwischenresultat mehr gespeichert hat. Ein weiterer Unterschied ist, dass kein Attribut mehr angegeben werden muss, welches verglichen werden soll, da die NonDBNodes nur noch Listen und keine Tabellen mehr als Daten enthalten.

Die Funktion der *INTERSECT*-Operation wird anhand der folgenden Beispielfrage verdeutlicht: *In which movies did Brad Pitt and George Clooney play* ***together****?*

Es wird angenommen, dass das aktuelle Zwischenresultat nur noch die Filmtitel von Brad Pitt's Filmen enthält und folgendermassen aussieht:

ocean's eleven
ocean's twelve
the big short

Zudem befindet sich in den gespeicherten Resultaten ein StringListNode, der die Filmtitel von George Clooney's Filmen enthält:

ocean's eleven
ocean's twelve
the american

Nun wird die *INTERSECT*-Operation aufgerufen mit den beiden StringListNodes als Input. Als Resultat wird die Schnittmenge ausgegeben:

ocean's eleven
ocean's twelve

Differenz

Die *DIFFERENCE*-Operation bildet wie die *DIFFERENCE*-Operation aus Kapitel 4.6.7.1 die Differenz aus zwei Zwischenresultaten. Der Unterschied ist hierbei, dass die Operation als Input nicht zwei DBNodes, sondern zwei StringListNodes, respektive zwei NumericListNodes annimmt. Diese Operation wird benötigt, wenn der Benutzer bereits die Werte eines Attributs extrahiert und somit kein DBNode als Zwischenresultat mehr gespeichert hat. Ein weiterer Unterschied ist, dass kein Attribut mehr angegeben werden muss, welches verglichen werden soll, da die NonDB-Nodes nur noch Listen und keine Tabellen mehr als Daten enthalten.

Als Beispiel wird folgende Frage gewählt:
In which movies did Brad Pitt play, but George Clooney didn't?

Es wird angenommen, das aktuelle Zwischenresultat sehe folgendermassen aus und enthalte die Filmtitel von Brad Pitt's Filmen:

ocean's eleven
ocean's twelve
the big short

Zudem befindet sich in den gespeicherten Resultaten ein StringListNode, der die Filmtitel von George Clooney's Filmen enthält:

ocean's eleven
ocean's twelve
the american

Nun wird die *DIFFERENCE*-Operation aufgerufen mit den beiden StringListNodes als Input. Als Resultat wird die Differenz ausgegeben:

the big short

4.7. Usability Operationen

Zusätzlich zu den atomaren Operationen, welche geloggt werden, wurden auch Operationen implementiert, welche dem Benutzer die Beantwortung einer Frage vereinfachen sollen. Diese Operationen ändern nichts an den Daten, welche sich im Zwischenresultat befinden, weswegen diese Operationen nicht geloggt werden. Da diese Operationen nichts an den Daten ändern, wird auch kein neuer Knoten im Baum eingefügt.

4.7.1. Attributselektion

Bei grossen Datenbanken, welche Tabellen mit vielen Attributen beinhalten, kann ein Zwischenresultat sehr schnell unübersichtlich werden. Um dieses Problem zu lösen, wurde eine Usability-Operation implementiert, mit der der Benutzer bestimmte Attribute auswählen kann, so dass nur noch diese ausgewählten Attribute angezeigt werden. Der Vorteil dieser Operation ist, dass weiterhin z.B. *MERGE*-Operationen durchgeführt werden können, auch wenn allfällige Schlüsselattribute gar nicht mehr im Resultat angezeigt werden. Im Hintergrund werden weiterhin alle Attribute abgespeichert. Falls in einem späteren Schritt zuvor ausgeblendete Attribute wieder angezeigt werden sollen, kann das mit der gleichen Operation gemacht werden. In Abbildung 4.24 wird das Pop-up-Menü für die Attributselektion gezeigt. Um dem Benutzer die Auswahl zu vereinfachen, kann er alle Attribute abwählen oder - falls alle Attribute abgewählt sind - wieder alle Attribute auswählen. Anschliessend kann mit einem Klick auf den Knopf *SEND* die Auswahl bestätigt werden.

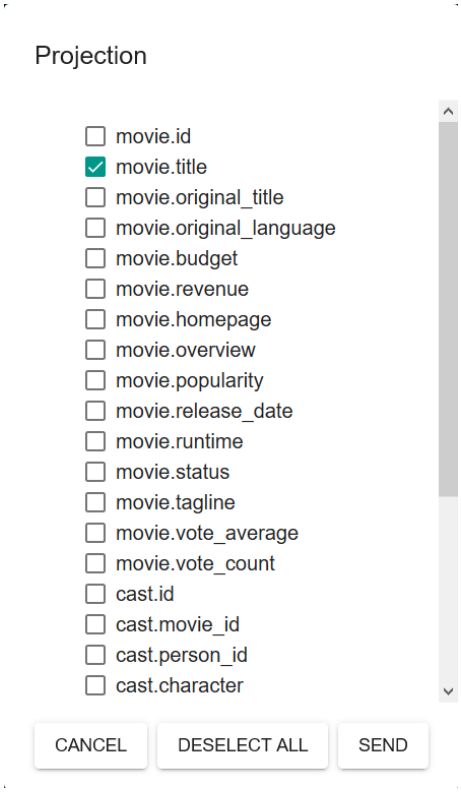

Abbildung 4.24.: Screenshot des Pop-up-Menüs für die Attributselektion.

Die Abbildungen 4.25 und 4.26 zeigen ein Beispiel für Zwischenresultate vor und nach der Attributselektion. Ausgewählt wurden die beiden Attribute *movie.title* und *person.name*, als Beispielfrage wurde *"In which movies did George Clooney play?"* verwendet.

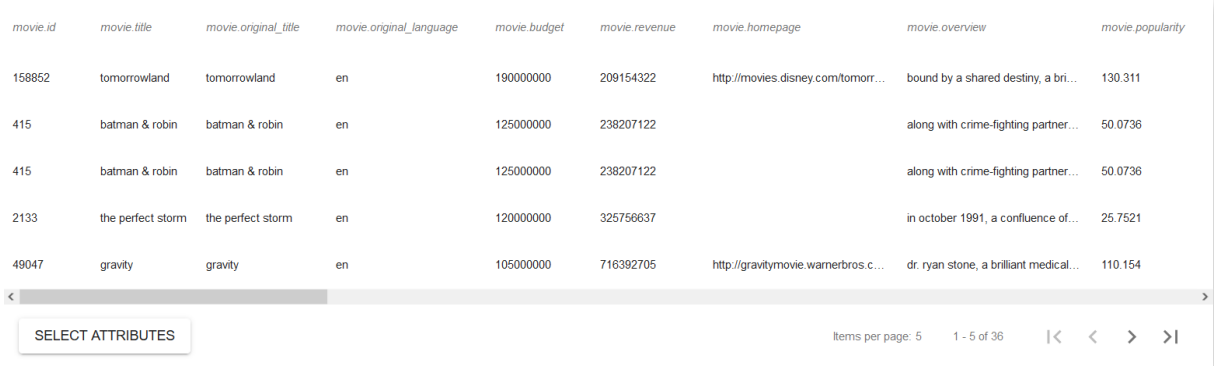

Abbildung 4.25.: Screenshot des Zwischenergebnisses vor der Attributselektion.

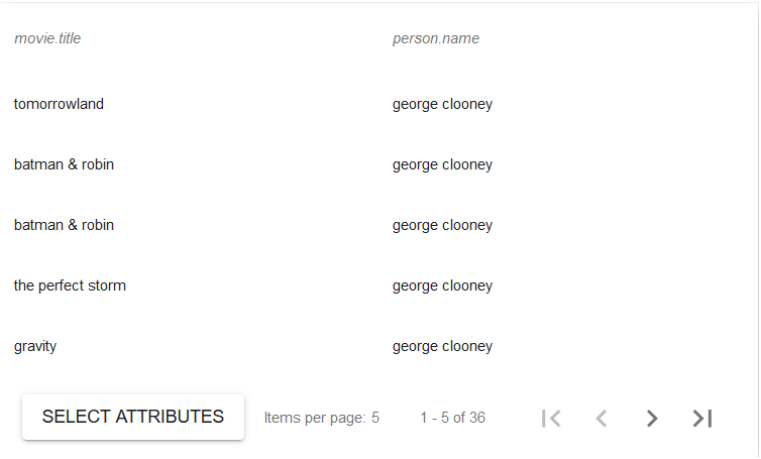

Abbildung 4.26.: Screenshot des Zwischenergebnisses nach der Attributselektion

4.7.2. Sortierung

Eine weitere Operation, welche die Daten eines Zwischenresultats nicht verändert, ist die Sortierung anhand eines Attributes. Der Benutzer kann bei einem gegebenen Zwischenresultat auf ein Attribut klicken, damit die Daten aufsteigend anhand dieses Attributes sortiert werden. Mit einem weiteren Klick werden die Daten absteigend sortiert. Nach einem dritten Klick wird wieder die ursprüngliche Sortierung der Daten angezeigt.

4.8. Baum in Applikation

Die Baumstruktur, wie sie in Kapitel 3.2 beschrieben wird, ist in der Applikation während dem Beantworten einer Frage gespeichert. Mit jeder Operation wird sowohl ein neuer Operations-, als auch ein Datenknoten an den Baum angehängt. Der Benutzer hat jederzeit Zugriff auf diese Baumstruktur. Zusätzlich zu den im Fallbeispiel (Kapitel 4.1) erwähnten Zwischenresultaten wird in der Sidebar im Abschnitt *Current History* auch der aktuelle Strang des Suchbaums abgebildet.

Zur Veranschaulichung wird wieder die Frage *"In which movies did Brad Pitt play, where the budget was greater than 100000000?"* aus dem Fallbeispiel verwendet.

Angenommen, der Benutzer sucht zuerst den Eintrag von *Brad Pitt* in der Datenbank. In Abbildung 4.27 wird links dargestellt, wie der aktuelle Strang in der Applikation dargestellt wird. Zusätzlich sind rechts die zwei Operationen dieses Suchstrangs im Loggingbaum rot markiert. Die IDs der *Current History* $\{0, 3\}$ unterscheiden sich zu denen des Loggingbaums $\{1, 2\}$, da in der *Current History* nur die IDs der Datenknoten ersichtlich sind, im Loggingbaum hingegen nur die der Operationsknoten. Man sieht in Abbildung 4.27 auch, dass die Abfrage über den invertierten Index im Loggingbaum nicht gespeichert wird, sondern direkt mit der *GET-DATA*-Operation gestartet wird.

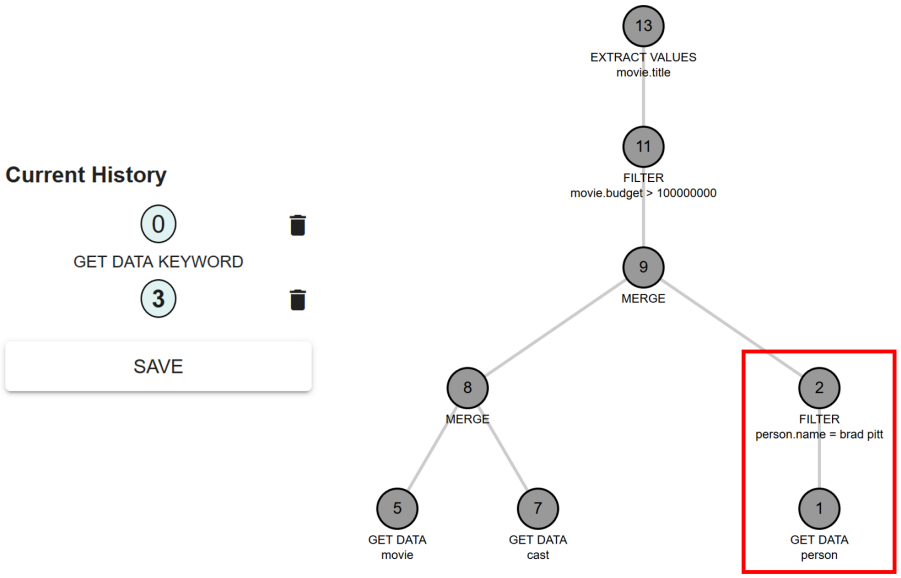

Abbildung 4.27.: Aktueller Strang für das Zwischenresultat *Brad Pitt*

Angenommen, der Benutzer hat in der Zwischenzeit alle Filme der Datenbank in einem Zwischenresultat gespeichert, kann er diese nun mit dem Zwischenresultat *Brad Pitt* verbinden. Bevor der Benutzer die *MERGE*-Operation (Operation-ID 9 im Loggingbaum) ausführt, sind die beiden Teilbäume (*Brad Pitt* und *Filme*) unabhängig voneinander. Die *MERGE*-Operation stellt dann die Verbindung zwischen diesen beiden Zwischenergebnissen her.

Anschliessend wird mit der Bedingung $movie.budget > 100000000$ gefiltert und die Filmtitel als Liste extrahiert. In Abbildung 4.28 wird links die *Current History* nach diesen Operationen dargestellt und rechts die Operationen im Loggingbaum rot markiert. Im Gegensatz zum Loggingbaum wird der Strang in der Applikation von oben nach unten dargestellt, weil dies intuitiver ist für die Benutzer.

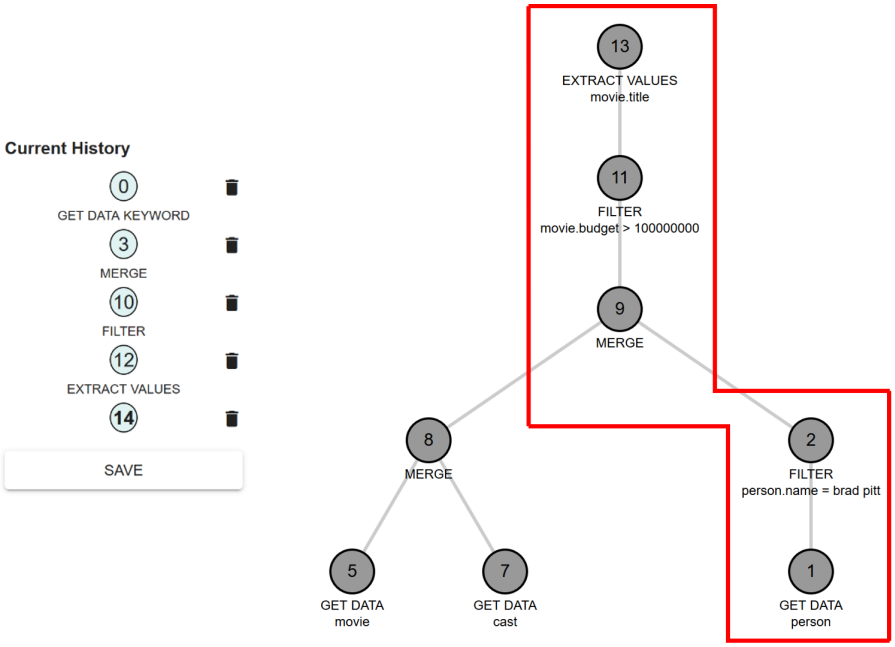

Abbildung 4.28.: Aktueller Strang für das Endresultat

4.9. Logging

Sobald ein Benutzer mit dem Beantworten seiner Frage fertig ist, kann er mit einem Klick auf den *FINISH*-Button die Abfrage abschliessen. Es wird ein Pop-up-Menü angezeigt, in dem der Benutzer gefragt wird, ob er seine gestellte Frage beantworten konnte oder nicht. Diese Unterscheidung ist wichtig, weil später für das Machine Learning nicht beantwortete Fragen anders gehandhabt werden müssen als Fragen, die beantwortet werden konnten. Abbildung 4.29 zeigt dieses Pop-up-Menü.

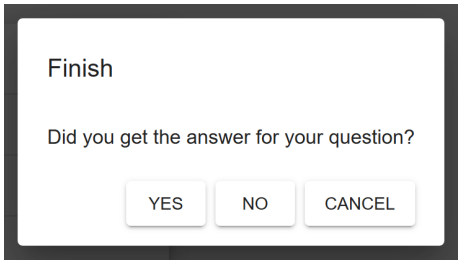

Abbildung 4.29.: Screenshot des *FINISH*-Pop-up-Menüs

Sobald der Benutzer auf *YES* oder *NO* klickt, werden alle Operationsknoten der in Kapitel 4.8 beschriebenen Baumstruktur verwendet, um den Loggingbaum zu generieren. Dieser wird anschliessend ans Backend gesendet, worauf die einzelnen Operationsknoten in SemQL ins Logfile eingetragen werden.

4.9.1. Zusammengefasste Operationen

Um dem Benutzer die Bedienung zu erleichtern, wurden zusammengefasste Operationen entwickelt, so dass der Benutzer mit wenigen Klicks mehrere atomare Operationen zusammengefasst ausführen kann. Für den Loggingbaum werden jedoch wieder die atomaren Operationen benötigt, da für das Machine Learning die Baumstruktur möglichst simpel bleiben soll. Aus diesem Grund werden die zusammengefassten Operationen für den Loggingbaum wieder in atomare Operationen aufgesplittet.

Eine solche zusammengefasste Operation ist zum Beispiel der *Merge* über eine oder mehrere Zwischentabellen. Angenommen, der Benutzer will die Tabelle *person* mit der Tabelle *movie* verbinden. Dazu wählt er den Pfad über die Tabelle *cast* aus. Abbildung 4.30 stellt diesen Pfad dar.

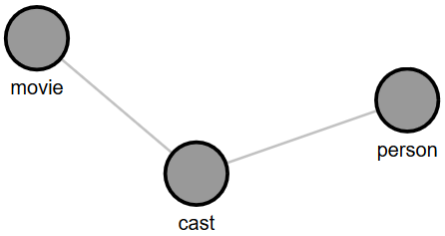

Abbildung 4.30.: Der Merge-Pfad

Im Hintergrund werden bei der Ausführung dieser Operation zwei *MERGE*-Operationen und eine *GET-DATA*-Operation ausgeführt. Als Erstes wird mit der *GET-DATA*-Operation die komplette Tabelle *cast* aus der Datenbank geladen. Anschliessend wird mit der ersten *MERGE*-Operation die Tabelle *movie* mit dem eben erstellten Zwischenresultat *cast* verbunden. Zu guter Letzt wird mit der zweiten *MERGE*-Operation das neu erstellte Zwischenresultat mit der Tabelle *person* verbunden. Der Benutzer sieht nichts von diesen beiden *Merges* und der *GET-DATA*-Operation. Ihm wird in der aktuellen Historie lediglich ein *Merge* angezeigt. Im Hintergrund werden jedoch die einzelnen atomaren Operationen gespeichert, da die SemQL nur einen *Merge* über direkt benachbarte Tabellen zulässt.

4.9.2. Logfile

Im Logfile einer Abfrage werden folgende Informationen gespeichert:

- beantwortete Frage
- Flag, ob die Beantwortung der Frage erfolgreich war
- Loggingbaum, um die Abfrage reproduzieren zu können

Wie bereits erwähnt, wird der Loggingbaum in SemQL festgehalten. Die Semantic Query Language wird dabei eingesetzt, um einen Operationsknoten genau zu beschreiben. Neben dem Operationsnamen wird festgehalten, welcher Knoten als Input dient, welche Argumente der Operation mitgegeben werden und zuletzt, mit welchem Teil der Frage die Operation vom Benutzer assoziiert wird.

Dies wird anhand folgender Beispielfrage verdeutlicht:
In which movies did Matt Damon play?

Für dieses Beispiel wird angenommen, dass der Benutzer bereits alle Personen aus der Datenbank geholt hat. Dieser *GET-DATA*-Operationsknoten hat die ID 1. Der darauf folgende *FILTER*-Operationsknoten wird nun folgend in SemQL dargestellt:

```
{
  "arguments": {
    "attribute": "person.name",
    "comparisonOperator": "=",
    "value": "matt damon"
  },
  "inputNodes": [1],
  "operation": "FILTER",
  "tokenIndices": [4, 5]
}
```

Da dieser Operationsknoten eine *FILTER*-Operation darstellt, wurde bei *operation* der Wert *FILTER* eingetragen. Als Inputknoten wurde die ID des *GET-DATA*-Knoten eingetragen, da dessen Daten gefiltert werden. Die ID wird in einer Liste (erkennbar an den eckigen Klammern) gespeichert, da es Operationen gibt, welche mehrere Inputknoten akzeptieren. Als Argumente wurden das Attribut *person.name*, der Vergleichsoperator = und der Wert *matt damon* gewählt, da in diesem Beispiel nach der Person Matt Damon gefiltert werden soll. Als Letztes werden die Token Indices 4 und 5 eingetragen, da für diese Operation die Begriffe *Matt* (an Stelle 4 des Satzes) und *Damon* (an Stelle 5 des Satzes) entscheidend sind.

Ein Beispiel für ein komplettes Logfile befindet sich im Anhang A.2.

4.9.3. Grafische Darstellung des Loggingbaums

Neben dem Logfile wird zusätzlich eine grafische Darstellung des Loggingbaums als Bild gespeichert. Dieses Bild kann, wie im Fallbeispiel (Kapitel 4.1) beschrieben, vom Benutzer eingesehen werden. Diese grafische Darstellung hilft, eine bestehende Abfrage schnell verstehen zu können.

In Abbildung 4.31 wird ein Beispiel eines solchen grafischen Loggingbaums dargestellt.

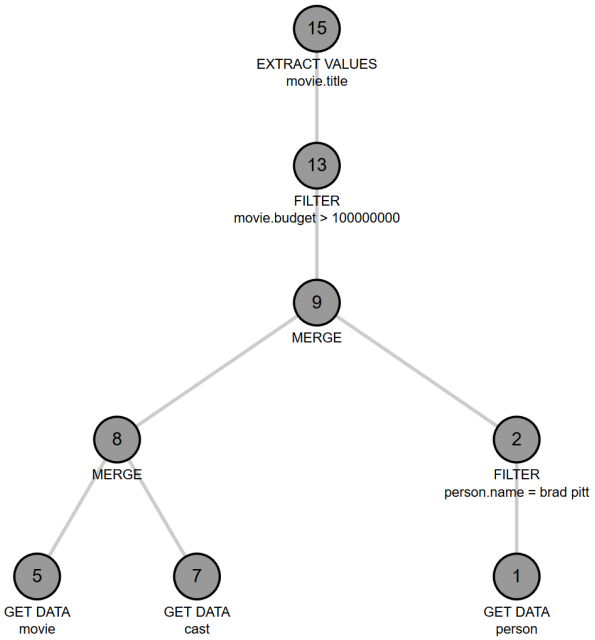

Abbildung 4.31.: Grafische Darstellung eines Loggingbaums

4.10. Automatische Ausführung von Queries

Wie in Kapitel 4.1 bereits erwähnt, lassen sich bereits erstellte Logfiles wieder automatisch abspielen. Somit lassen sich Resultate schnell und ohne erheblichen Aufwand reproduzieren. Das Logfile wird mit einem eigens implementierten Parser geparst, anschliessend werden die einzelnen Operationen ausgeführt und das Schlussresultat an das Frontend gesendet.

Der aktuelle Parser nimmt aktuell keinerlei Query-Optimierungen vor, weshalb einige Operationen nicht sehr performant sind. Wenn grosse Datenbanktabellen geladen werden müssen, dauert eine Query aktuell noch zu lange. In Zukunft wäre in dieser Hinsicht sicherlich noch Verbesserungspotenzial vorhanden.

4.11. Datentypen

Es wurde entschieden, einfachheitshalber nicht jeden einzelnen Datentyp von MySQL zu unterstützen. Beschränkt wurde sich auf die gängigsten Datentypen, allerdings dürfte trotzdem ein Grossteil der Datenbanken bereits mit dem Prototypen verarbeitet werden können. „Es werden folgende Datentypen unterstützt:

Numerische Datentypen

tiny, short, long, int24, integer, longlong, year, decimal, float, double, newdecimal [10]

Alphanumerische Datentypen

char, varchar, text [11]

Datumsformate [12] werden aktuell nicht unterstützt, bzw. werden als String behandelt.“ [7]

4.12. Verwendete Technologien

Da zu Beginn der Arbeit bereits ein Prototyp zur Verfügung stand, wurde mit den selben Technologien weitergearbeitet. Folgende Technologien wurden eingesetzt:

4.12.1. Backend

- Python 3.7
- Flask 1.0.2
- MySQL 5.7

Das Backend wurde in Python umgesetzt, „weil damit schnell neue Funktionen ohne unnötigen Overhead implementiert werden können und sehr einfach mit Stringmanipulationen umgegangen werden kann. Ein weiterer Vorteil ist, dass sich mit dem Python-Lightweight-Framework Flask [13] in kurzer Zeit und mit relativ wenig Code eine funktionierende REST-API aufbauen lässt. Diese Eigenschaft wurde als wichtig für dieses Projekt erachtet, da das Backend nicht allzu komplex ist und die eigentliche Logik der Applikation im Frontend steckt.“ [14]

4.12.2. Frontend

- Angular 7
- Angular Material 7
- D3.js 5.9

Das Frontend wurde mit Hilfe des Typescript-Frameworks Angular [15] geschrieben. Angular Material [16] wurde verwendet, um die ganze Applikation einheitlich und ansprechend zu gestalten. Für die Graph-Darstellungen wurde die Javascript-Library D3.js verwendet [17].

5. Ergebnisse

5.1. Analyse von gestellten Fragen

Damit der Stand der Applikation besser bestimmt werden kann und erste Erkenntnisse für zukünftige Machine-Learning-Algorithmen gezogen werden können, wurden Beispielfragen von Benutzern ausgewertet.

5.1.1. Erhebung der Daten

Die Fragen wurden mit Hilfe von Amazon Mechanical Turk [18] erhoben. Die Erhebung wurde in zwei Teile aufgeteilt. Als Grundlage diente auch hier wieder die Movie-Datenbank.
Die Beispielfragen wurden vom Nebenbetreuer dieser Arbeit, Jan Deriu, erhoben. In den nachfolgenden Abschnitten wird erklärt, wie die Erhebung der Fragen durchgeführt wurde.

Teil 1
Der erste Teil der Fragen wurde so erhoben, dass den Benutzern aus im Voraus ausgewählten Begriffen, welche in der Datenbank vorkommen, sechs Begriffe vorgeschlagen wurden, zu denen Sie eine Frage stellen mussten. Diese sechs Begriffe wurden zufällig aus der Menge von ausgewählten Begriffen zusammengestellt. Dabei ging es nicht darum, dass die Benutzer eine Frage stellen, welche alle sechs Wörter abdeckt. Die Wörter wurden zur Inspiration angezeigt. Die Wörter, welche für diese Menge ausgewählt wurden, sind Tabellennamen, Attributnamen oder auch Felder von der Movie-Datenbank.
Ein Beispiel von 6 Wörtern: *Italy, actor, genre, Disney, oscar, nominee, Wolverine*. Aus diesen Wörtern stellte ein Benutzer die Frage: *Who produced the movie Wolverine?* Mit diesem Verfahren wurden 90 Fragen erhoben.

Teil 2
Der zweite Teil der Erhebung war dem ersten Teil sehr ähnlich. Der einzige Unteschied war, dass die Wörter nicht mehr aus einer im Voraus zusammengestellten Menge ausgewählt wurden, sondern zufällig irgendwelche Wörter aus der Datenbank genommen wurden. Im zweiten Teil wurden 176 Fragen erhoben. Insgesamt kamen durch diese beiden Erhebungen 266 Fragen zusammen, welche anschliessend für die Auswertung genutzt werden konnten.

5.1.2. Filtern der Fragen

Nachdem genügend Fragen erhoben worden sind, wurden sie darauf überprüft, ob sie mit dem Tool überhaupt beantwortet werden können. Es wurden demzufolge zwei Kategorien erstellt: *beantwortbare Fragen* und *nicht beantwortbare Fragen*. Die beantwortbaren Fragen wurden anschliessend mit dem Tool beantwortet, so dass die erstellten Logfiles ausgewertet werden konnten. Die nicht beantwortbaren Fragen wurden darauf untersucht, wieso sie nicht beantwortet werden konnten.

Von allen 266 Fragen konnten 90 verwendet werden. Es konnten also nur 33.8% der gestellten Fragen beantwortet werden. Bei den anderen 176 stellte sich heraus, dass sie in eine von fünf Kategorien eingeteilt werden können:

Kategorie	**Beschreibung**
Daten nicht in DB	Die Frage konnte nicht beantwortet werden, da die benötigten Daten nicht in der Datenbank vorhanden sind.
Operation nicht unterstützt	Die Operation, welche benötigt würde, um die Frage zu beantworten, wird vom Tool derzeit nicht unterstützt.
Sinnlose Frage	Die Frage wurde so gestellt, dass nicht herausgelesen werden konnte, was genau gemeint war, oder die Frage an sich hat keinen Sinn ergeben (non-sense).
Falsche Tokenisierung	Die Tokenisierung, welche die Applikation vornimmt, hat das zu suchende Wort so verändert, dass es in der Datenbank nicht mehr gefunden werden konnte.
Duplikat	Die Frage wurde mehr als ein Mal gestellt.

Tabelle 5.1.: Beschreibung der Kategorien von nicht beantwortbaren Fragen

Die nicht beantwortbaren Fragen teilen sich in den Kategorien folgendermassen auf:

Kategorie	**Anzahl Fragen**
Daten nicht in DB	73 (41.47%)
Operation nicht unterstützt	64 (36.36%)
Sinnlose Frage	37 (21.02%)
Falsche Tokenisierung	1 (0.57%)
Duplikat	1 (0.57%)

Tabelle 5.2.: Verteilung der nicht beantwortbaren Fragen in die Kategorien

Man kann sehen, dass der grösste Teil der Fragen nicht beantwortet werden kann, weil die Daten dafür nicht in der Datenbank vorhanden sind. Auf den ersten Blick scheint das etwas widersprüchlich, da nur Begriffe vorgeschlagen wurden, die auch in der Datenbank vorhanden sind. Die Benutzer sind jedoch nicht davon abgehalten worden, Fragen zu stellen, bei denen es um Filme geht, die nicht in der Datenbank vorhanden sind. Es kommt also sehr darauf an, was für eine Datenbank an das Tool angehängt ist.

Bei der Kategorie *Operation nicht unterstützt* kann mehr herausgeholt werden. In Zukunft könnte das Tool um weitere Operationen ergänzt werden, damit mehr Fragen beantwortet und somit mehr Trainingsdaten generiert werden können.

5.1.3. Auswertung der beantwortbaren Fragen

Einige von den nicht beantwortbaren Fragen wären mit einer kleinen Anpassung sehr spannende, beantwortbare Fragen. Zusätzlich zu den 90 beantwortbaren Fragen wurden so ein Dutzend Fragen von den nicht beantwortbaren verwendet und ein wenig abgeändert. Somit wurden schlussendlich für 102 Fragen die Loggingbäume erstellt und anschliessend ausgewertet.

Noch eine wichtige Bemerkung, die beachtet werden muss:
Die Beantwortung der Fragen wurde durch die beiden Autoren dieser Arbeit durchgeführt. Das heisst, es kann durchaus sein, dass diese zwei Autoren durch das wiederholte, manuelle Testen der Applikation voreingenommen waren, was das Verwenden von bestimmten Operationen betrifft. Viele Fragen können über verschiedene Wege beantwortet werden, wodurch andere Testuser vielleicht andere Operationen häufiger genutzt hätten.

5.1.3.1. Anzahl Operationen in den Queries

Die durchschnittliche Anzahl von benötigten Operationen, um eine Frage zu beantworten liegt bei 6.95. Der Median liegt bei genau 7.0. Abbildung 5.1 zeigt die Verteilung der benötigten Schritte. Man erkennt gut, dass die Queries mit 3, 7 und 8 Operationen den grössten Teil ausmachen. Das ist darauf zurückzuführen, dass viele Operationen sehr ähnliche Strukturen aufweisen. In Abschnitt 5.1.3.3 wird noch genauer darauf eingegangen.
Alle Queries bis auf eine benötigten zwischen 3 und 13 Operationen, um sie zu beantworten. Diejenige mit 17 Operationen besteht aus so vielen Operationen, weil sie Zwischenergebnisse beinhaltet, welche über Tabellen gemerged werden, die im Graph weit voneinander entfernt sind. Für interessierte Leser ist der Loggingbaum dieser Query im Anhang A.3 abgebildet.

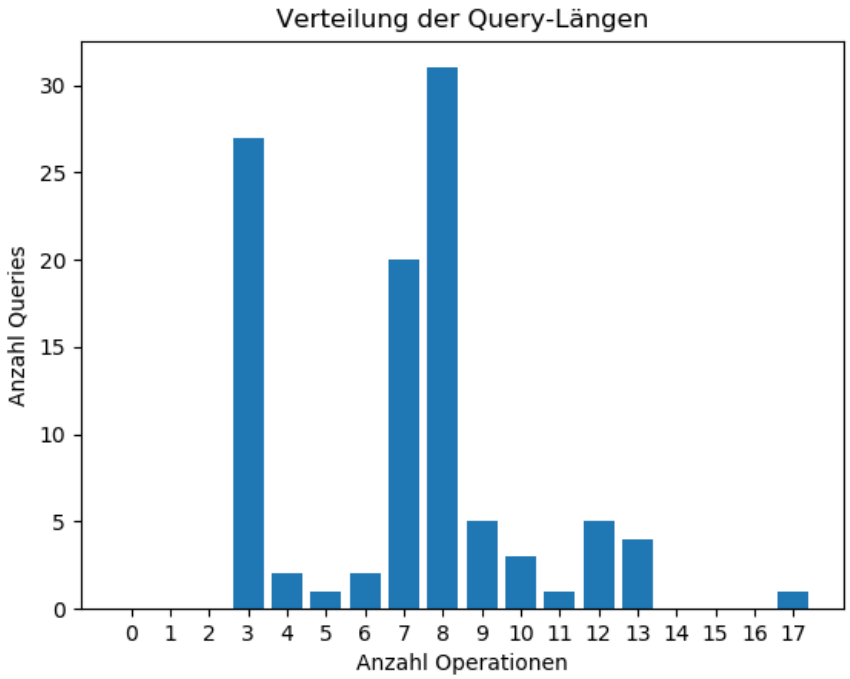

Abbildung 5.1.: Die Verteilung der Anzahl benötigten Operationen

5.1.3.2. Verwendete Operationen

Abbildung 5.2 zeigt, welche Operationen in absoluten Zahlen am meisten genutzt wurden. Die Tatsache, dass die Anzahl Verwendungen einiger Operationen grösser ist als die totale Anzahl gestellter Fragen, rührt daher, dass einige Operationen mehrmals in der gleichen Frage verwendet werden.
Es ist naheliegend, dass *Get Data* so oft genutzt wird, da diese Operation zu Beginn jeder Abfrage mindestens ein Mal verwendet werden muss, um überhaupt an Daten zu kommen. Auffällig ist jedoch, dass die Mengenoperationen wie z.B. *Union* oder *Intersect* sehr selten verwendet worden sind. Auch bemerkenswert ist, dass von den insgesamt 13 Operationen, welche das Tool zur Verfügung stellt, nur deren neun mindestens ein Mal verwendet wurden. Vier Operationen wurde also gar nie verwendet. Dies vier Operationen sind *MIN*, *MAX*, *AVG* und *DIFFERENCE*.

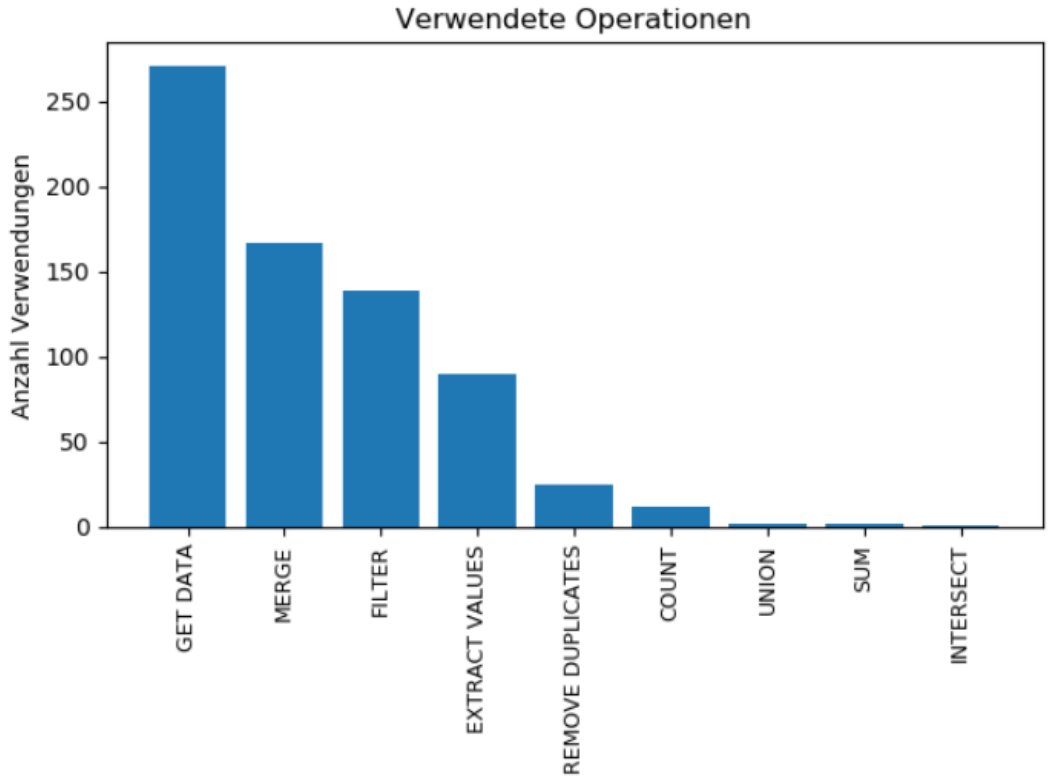

Abbildung 5.2.: Die Anzahl benötigter Operationen in absoluten Zahlen

Die nachfolgende Tabelle sagt aus, in wie vielen Queries eine Operation verwendet wurde. Die Zahlen sind im Gegensatz zu Abbildung 5.2 in relativen Werten angegeben.

Operation	**In wie vielen Queries verwendet**
Filter	100%
Get Data	100%
Extract Values	88.24%
Merge	70.59%
Remove Duplicates	24.51%
Count	11.76%
Sum	1.96%
Union	0.98%
Intersect	0.98%

Filter und *Get Data* wurden in jeder Operation mindestens ein Mal verwendet. *Extract Values* wurde in fast 9 von 10 Fragen genutzt. Auffallend, im Vergleich mit der Abbildung 5.2 ist, dass die *Merge*-Operation in absoluten Zahlen am zweithäufigsten verwendet wird, relativ gesehen aber nur auf Platz vier steht. Dies kommt daher, dass oft entweder gar kein *Merge* notwendig ist, oder dann gleich mehrere. Die *Merge*-Operation wird dann nicht benötigt, wenn die gewünschten Daten in der ersten Menge von Daten, welche von der Datenbank geholt werden, bereits vorhanden sind. Das ist dann oft der Fall, wenn nur drei Operationen verwendet werden müssen (vgl. Abbildung 5.1).

5.1.3.3. Oft verwendete Strukturen in Loggingbäumen

In diesem Kapitel wird die Tatsache untersucht, dass die Strukturen mit den Längen 3, 7 und 8 den grössten Teil von allen Queries ausmachen. Es wird sich erhofft, dass Strukturen in den Loggingbäumen gefunden werden, welche oft vorkommen. Das wäre eine vielversprechende Grundlage für das Machine Learning. Dazu wurden zuerst einige Stichproben von den 102 generierten Loggingbäumen gemacht und analysiert, welche Strukturen sich lohnen könnten, weiter zu untersuchen. In den nachfolgenden Abschnitten werden zwei spezifische Strukturen genauer erläutert.

Merge-Merge-Filter-Struktur (MMF)

Die erste Struktur, die untersucht wird, wird *Merge-Merge-Filter-Struktur* oder abgekürzt *MMF* genannt. Sie heisst so, weil sie aus zwei *Merge*-Operationen einer Filter-Operation und drei Get-Data Operationen besteht. Abbildung 5.3 zeigt ein Beispiel einer solchen Struktur in einem Loggingbaum (rot umrahmt).

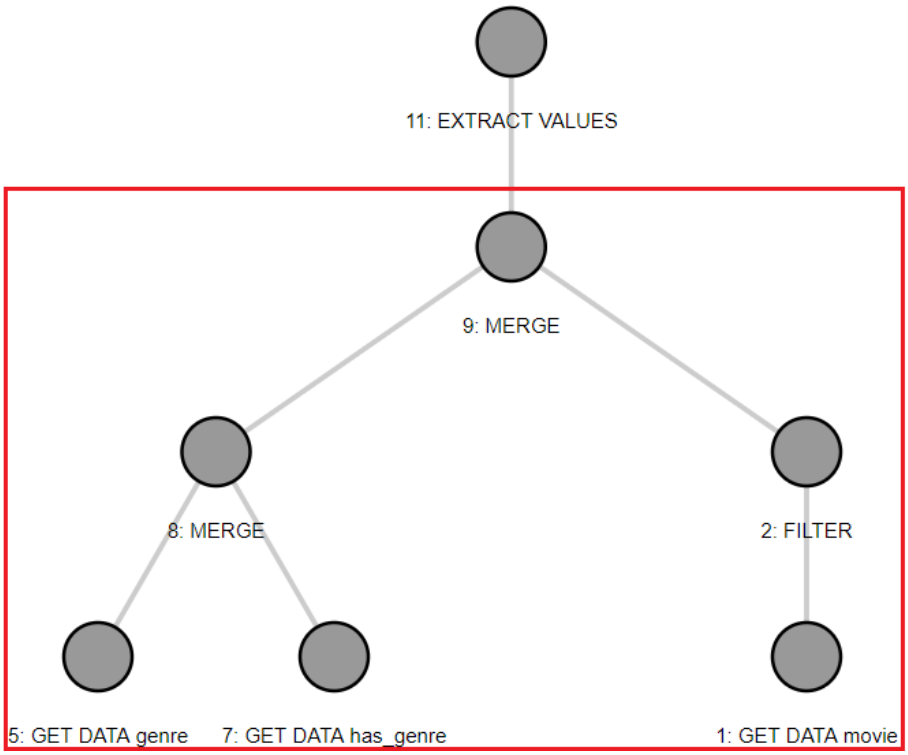

Abbildung 5.3.: Ein Loggingbaum mit der *Merge-Merge-Filter-Struktur* rot umrahmt.

Die MMF-Struktur wird nur anhand ihrer Operationen in dieser Reihenfolge definiert. Dabei kommt es nicht darauf an, welche Tabellen mit diesen Operationen bearbeitet werden. Bei der Auswertung wurde nicht darauf geachtet, an welchem Platz eines Loggingbaumes diese Struktur vorkommt. Das heisst, es wurde nicht unterschieden, ob sie wie in Abbildung 5.3 zu Beginn der Struktur oder erst später im Loggingbaum vorkommt.
Diese Struktur kam in 44 von den 102 generierten Loggingbäumen vor, was 43% aller Queries entspricht. Das ist eine gute erste Voraussetzung für einen Machine Learning Algorithmus, der diese Struktur lernen kann.

No-Merge-Struktur (NM)
Eine weitere Struktur, die oft vorgekommen ist, ist die *No-Merge*-Struktur, genannt *NM*. Diese Struktur zeichnet sich dadurch aus, dass sie keine *Merge*-Operation beinhaltet. In Abbildung 5.4 ist ein Beispiel einer solchen Struktur abgebildet. Diese Struktur ist sehr einfach. Im Unterschied zur *MMF*-Struktur kann mit dieser Struktur eine Frage komplett beantwortet werden, obwohl sie um einiges weniger komplex ist.

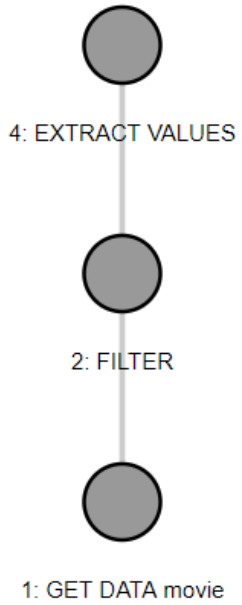

Abbildung 5.4.: Ein Beispiel einer *No-Merge*-Struktur

Ein Beispiel für eine Frage, welche mit dieser Struktur beantwortet werden kann, ist folgende: *What was the release date for Pulp Fiction?*. Die Applikation findet im invertierten Index *Pulp Fiction*. Nachdem dieser Eintrag ausgewählt wurde, muss nur noch anhand des Attributs *release_date* extrahiert werden.

Bei der Auswertung dieser Struktur wurde untersucht, wie viele Fragen mit dieser Struktur direkt beantwortet werden konnten. Es wurde als nicht überprüft, in wie vielen Loggingbäumen diese Struktur vorkommt, sondern wie viele Bäume als Ganzes dieser Struktur entsprechen. Auch hier wurde nur darauf geachtet, dass die Operationen die gleichen sind. Es wurde nicht überprüft, ob die verwendeten Tabellen dieselben sind. Von den 102 beantworteten Fragen konnten 27, also 26.5%, mit dieser Struktur beantwortet werden.

5.2. Verwendung als Annotationstool

Das Tool konnte so weit umgesetzt werden, dass es als Annotationstool für SemQL-Queries verwendet werden kann. In der Projektarbeit *"Interaktive Konstruktion von Datenbankabfragen"* [2] wurde mit Hilfe von Usertests festgestellt, dass das Tool (vor allem für Laien) ohne eine Einführung relativ schwierig zu bedienen ist. Es wird geschätzt, dass sich daran nicht viel geändert hat. Folglich müssen Benutzer, welche Fragen annotieren wollen, vor dem ersten Verwenden zuerst noch instruiert werden.
Während der Beantwortung der Auswertungsfragen durch die Autoren wurde festgestellt, dass ein geübter Benutzer eine Frage im Durchschnitt in drei Minuten beantworten kann. Pro Stunde können demnach 20 Queries annotiert werden.

6. Diskussion und Ausblick

6.1. Format für Machine Learning

Damit ein Machine Learning Algorithmus nicht zu viele Trainingsbeispiele verwenden muss, um nur schon das JSON-Format zu lernen, müssten die Loggingbäume in einem anderen Format abgespeichert werden. Eine Anforderung an dieses Format wäre beispielsweise, dass es kompakter ist als das JSON-Format. Aufgrund der Baumstruktur bietet es sich an, die Operationen in verschachtelten Aufrufen so kompakt als möglich zu notieren.
Angenommen, es würde nach dem Film *Inception* gefiltert werden, könnte die SemQL-Query aus Abbildung 5.4 in die folgende, kompaktere Form gebracht werden:

```
extract_values(filter(get_data(table='movie'), movie.title='Inception'), attribute='movie.title');
```

Alle JSON-Logfiles könnten so automatisiert in dieses Format gebracht werden.

6.2. Weitere atomare Operationen

Aufgrund der gewählten, flexiblen Architektur ist es einfach, in Zukunft weitere atomare Operationen zu implementieren. Wie in der Auswertung in Kapitel 5.1 erwähnt, konnten erst gut ein Drittel aller gestellten Fragen beantwortet werden. Es müssen definitiv noch mehr Operationen in SemQL definiert und im Tool implementiert werden, damit das Tool in der Praxis eingesetzt werden kann. Zu viele Operationen sollten allerdings auch nicht definiert werden, da der Lernprozess für den Machine-Learning-Algorithmus mit jeder zusätzlichen Operation schwieriger wird. Daraus folgt, dass mit jeder neuen Operation auch mehr Trainingsdaten generiert werden müssen. Es sollte ein guter Kompromiss gefunden werden, so dass genügend Fragen beantwortet werden können, gleichzeitig aber der Lernprozess für den Algorithmus nicht zu schwierig wird.

6.3. Benutzung mit weiteren Datenbanken

Da während der ganzen Entwicklung nur die Movie Database eingesetzt wurde, wurde die Applikation nach der Entwicklung noch mit zwei weiteren Datenbanken getestet, um herauszufinden, wie das Tool mit neuen Datenbanken zu handhaben ist. Dazu wurden zwei weitere Datenbanken, welche beide von MySQL zur Verfügung gestellt werden, an das Tool angehängt.

Employees-Datenbank
Die Employees-Datenbank [19] repräsentiert den Datenbestand von Mitarbeitern eines Unternehmens. Diese Datenbank enthält sechs Tabellen und ist insgesamt 160 Megabyte gross. Die grösste Tabelle enthält etwas über 2.8 Millionen Einträge. Aufgrund der grossen Tabellen ist sie gut geeignet, um das Tool auf das Handling mit grossen Datenmengen zu testen.

Sakila-Datenbank
Die Sakila-Datenbank [20] repräsentiert eine DVD-Ausleihe. Die Datenbank enthält 16 Tabellen, von denen die grösste gut 16'000 Einträge beinhaltet. Verglichen mit der MovieDB hat diese Datenbank also kleinere Tabellen. Dafür beinhaltet diese Datenbank zusätzlich zu den Tabellen noch sieben Views.

6.3.1. Aufgetretene Probleme

In diesem Abschnitt wird auf Probleme eingegangen, welche während dem Testen mit den beiden oben genannten Datenbanken aufgetreten sind.

6.3.1.1. Lange Ladezeiten bei grossen Tabellen

Wenn mit dem Tool grosse Datenmengen bearbeitet werden, kann es vorkommen, dass der Benutzer lange auf das Ende einer Operationsausführung warten muss. Die Ladezeit bei einer Tabelle mit 300'000 Einträgen beträgt ca. 20 Sekunden. Das liegt daran, dass die Applikation immer alle Daten auf ein Mal lädt, anstatt eine Art "lazy execution" auszuführen, welche nur die momentan benötigten Daten lädt.

6.3.1.2. Datenbankgraph

Bei der Benutzung des Datenbankgraphs sind die folgenden zwei Probleme aufgetaucht.

Views im Datenbankgraph
Falls die Datenbank Views enthält, werden diese auch im Datenbankgraph angezeigt. Das muss nicht unbedingt ein Problem sein. Die Frage, die sich jedoch stellt ist, ob es es Sinn macht, Views in dieser Applikation zu berücksichtigen. Views bringen keine zusätzlichen Daten, welche nicht schon in der Datenbank abgespeichert sind. Sie sind auch nicht mit anderen Tabellen oder Views verbunden, wie man in Abbildung 6.1 sehen kann. Es könnte jedoch trotzdem von Vorteil sein, dass der Benutzer diese Daten abfragen kann. Zur Zeit ist es so, dass Views zwar im Datenbankgraph abgebildet werden, aber der Benutzer die Daten nicht abfragen kann, was inkonsistent ist. In Abbildung 6.1 ist beispielsweise der Knoten *sales_by_store* eine View.

Automatische Ausrichtung des Graphs bei vielen Tabellen
Wenn eine Datenbank mit vielen Tabellen oder Views verwendet wird, dann sind nicht mehr alle Tabellen komplett sichtbar. Es werden also gewisse Tabellen teilweise abgeschnitten oder gar nicht mehr angezeigt. Diese Problem ist UI-bezogen und könnte beispielsweise mit einer Zoom-Funktion behoben werden. In Abbildung 6.1 ist zu sehen, wie die Tabellen in einem DB-Graph abgeschnitten werden, sobald zu viele Tabellen in der Datenbank vorhanden sind.

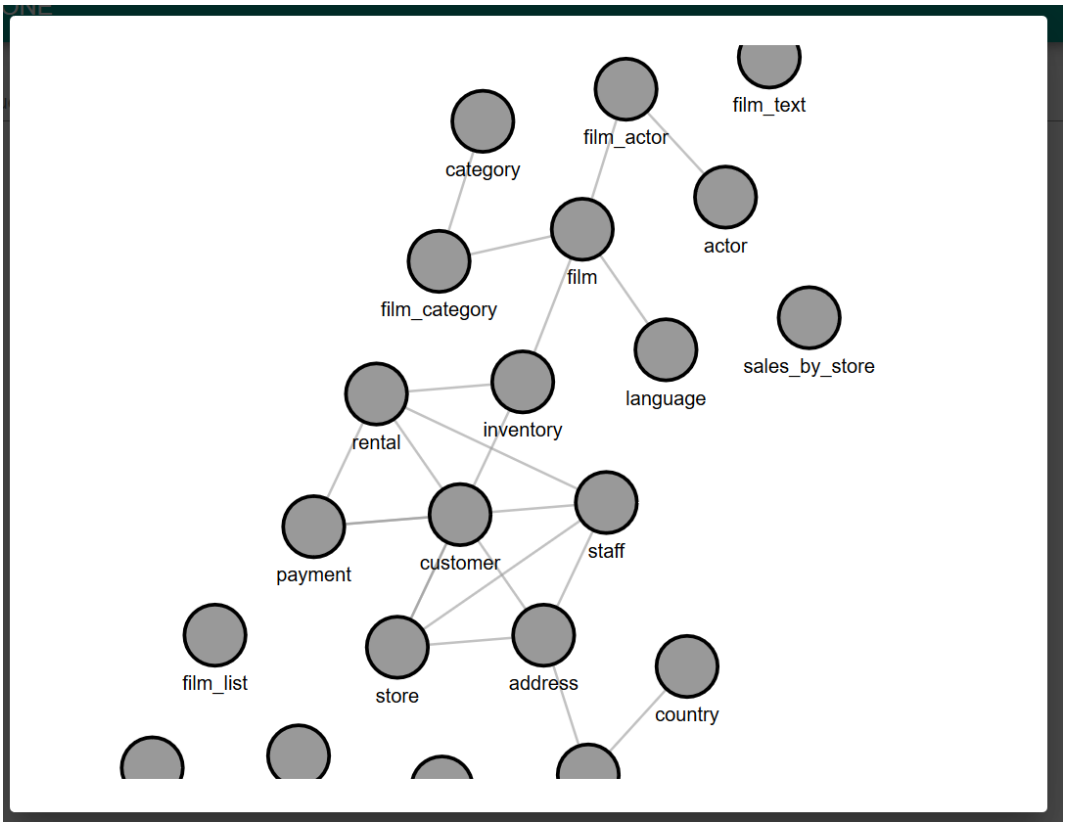

Abbildung 6.1.: Ein Screenshot des Datenbankgraphs mit Views und abgeschnittenen Tabellenknoten

6.3.1.3. Extract Values auf mehrere Attribute

Die *Extract-Values*-Operation ist so definiert, dass nur genau ein Attribut einer Tabelle ausgewählt werden kann. Das kann zum Problem werden, wenn für das Endergebnis mehrere Spalten benötigt werden.
Zur Veranschaulichung dient folgende Beispielfrage: *What are the names of the customers?*. Die Attribute der zugehörigen Tabelle der Sakila-Datenbank sehen folgendermassen aus:

- customer_id
- last_name
- first_name
- email
- ...

Damit die obengenannte Frage beantwortet werden kann, müssen die beiden Attribute *last_name* sowie *first_name* gleichzeitig extrahiert werden können. Zur Zeit ist das noch nicht möglich, aufgrund der Tatsache, dass SemQL nicht so definiert ist. Das müsste in Zukunft noch geändert werden, da ansonsten einige Fragen nicht beantwortet werden können. Während der Entwicklung des Tools ist diese Problematik nie aufgetreten, da in der MovieDB die Tabelle *Person* ein einziges Attribut für den Vor- und Nachnamen enthält.

7. Verzeichnisse

Literaturverzeichnis

[1] Nicolas Kaiser und Philippe Schläpfer. Interaktive Konstruktion von Datenbankabfragen. Projektarbeit, ZHAW Zürcher Hochschule für angewandte Wissenschaften, Dezember 2018. S.9.

[2] Nicolas Kaiser und Philippe Schläpfer. Interaktive Konstruktion von Datenbankabfragen. Projektarbeit, ZHAW Zürcher Hochschule für angewandte Wissenschaften, Dezember 2018.

[3] Marco Antonio Calijorne Soares and Fernando Silva Parreiras. A literature review on question answering techniques, paradigms and systems. *Journal of King Saud University - Computer and Information Sciences*, 2018.

[4] Lukas Blunschi, Claudio Jossen, Donald Kossmann, Magdalini Mori, and Kurt Stockinger. SODA: generating SQL for business users. *CoRR*, abs/1207.0134, 2012.

[5] Prasetya Utama, Nathaniel Weir, Fuat Basik, Carsten Binnig, Ugur Çetintemel, Benjamin Hättasch, Amir Ilkhechi, Shekar Ramaswamy, and Arif Usta. An end-to-end neural natural language interface for databases. *CoRR*, abs/1804.00401, 2018.

[6] Stanford University Infolab. Relational Algebra [Online]. `http://infolab.stanford.edu/~ullman/fcdb/aut07/slides/ra.pdf`. S. 29. [Zugriff am 28.05.2019].

[7] Nicolas Kaiser und Philippe Schläpfer. Interaktive Konstruktion von Datenbankabfragen. Projektarbeit, ZHAW Zürcher Hochschule für angewandte Wissenschaften, Dezember 2018. S.15-18.

[8] MySQL INFORMATION_SCHEMA Tables. `http://dev.mysql.com/doc/refman/8.0/en/information-schema.html`. [Zugriff am 31.05.2019].

[9] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. Exploring network structure, dynamics, and function using networkx. In Gaël Varoquaux, Travis Vaught, and Jarrod Millman, editors, *Proceedings of the 7th Python in Science Conference*, pages 11 – 15, Pasadena, CA USA, 2008.

[10] MySQL - Numerische Datentypen [Online]. `https://dev.mysql.com/doc/refman/8.0/en/numeric-type-overview.html`. [Zugriff am 28.05.2019].

[11] MySQL - Alphanumerische Datentypen [Online]. `https://dev.mysql.com/doc/refman/8.0/en/string-type-overview.html`. [Zugriff am 28.05.2019].

[12] MySQL - Datums- und Zeittypen [Online]. https://dev.mysql.com/doc/refman/8.0/en/date-and-time-type-overview.html. [Zugriff am 28.05.2019].

[13] Flask [Online]. http://flask.pocoo.org/. [Zugriff am 28.05.2019].

[14] Nicolas Kaiser und Philippe Schläpfer. Interaktive Konstruktion von Datenbankabfragen. Projektarbeit, ZHAW Zürcher Hochschule für angewandte Wissenschaften, Dezember 2018. S.26.

[15] Angular [Online]. https://angular.io/. [Zugriff am 28.05.2019].

[16] Angular Material [Online]. https://material.angular.io/. [Zugriff am 28.05.2019].

[17] D3.js [Online]. https://github.com/d3/d3. [Zugriff am 28.05.2019].

[18] MySQL INFORMATION_SCHEMA Tables. http://www.mturk.com/l. [Zugriff am 31.05.2019].

[19] MySQL Employees Sample Database [Online]. http://dev.mysql.com/doc/employee/en/. [Zugriff am 29.05.2019].

[20] MySQL Sakila Sample Database [Online]. http://dev.mysql.com/doc/sakila/en/. [Zugriff am 29.05.2019].

Abbildungsverzeichnis

Tabellenverzeichnis

A. Anhang

A.1. Architektur

„Die grundlegende Architektur dieser Applikation ist eine REST-Architektur, wie sie in Abbildung A.1 skizziert ist. Die Applikation ist aufgeteilt in ein Frontend und ein Backend, welche über eine REST-Schnittstelle miteinander kommunizieren. Über die REST-Schnittstelle, welche vom Backend zur Verfügung gestellt wird, werden JSON-Objekte hin und her gesendet. Typischerweise enthalten die vom Frontend gesendeten JSON-Objekte Informationen zu einer Abfrage, während die JSON-Objekte des Backends die dazugehörigen Resultate enthalten.

Das Frontend besteht aus einer Angular-Applikation, welche in einem Webbrowser dargestellt wird. Das Backend besteht aus einem Python-Webserver, der auf die zugrundeliegende MySQL-Datenbank zugreift. “ [7]

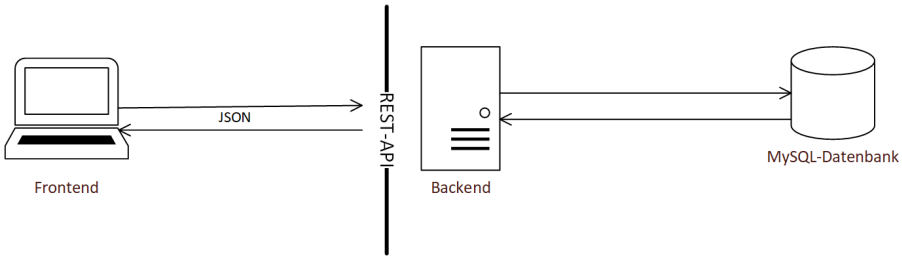

Abbildung A.1.: REST-Architektur

A.2. Beispiel Logfile

```
{
  "query": "In which movies did Brad Pitt play, where the budget
           was greater than 100000000?",
  "successful": true,
  "operationNodes": {
    "1": {
      "arguments": {
        "table": "person"
      },
      "inputNodes": null,
      "operation": "GET DATA",
      "tokenIndices": [4, 5]
    },
```

```
    "2": {
      "arguments": {
        "attribute": "person.name",
        "comparisonOperator": "=",
        "value": "brad pitt"
      },
      "inputNodes": [1],
      "operation": "FILTER",
      "tokenIndices": [4, 5]
    },
    "5": {
      "arguments": {
        "table": "movie"
      },
      "inputNodes": null,
      "operation": "GET DATA",
      "tokenIndices": [2]
    },
    "7": {
      "arguments": {
        "table": "cast"
      },
      "inputNodes": null,
      "operation": "GET DATA",
      "tokenIndices": [2, 3, 4, 5, 6]
    },
    "8": {
      "arguments": {
        "mergeAttribute1": "movie.id",
        "mergeAttribute2": "cast.movie_id"
      },
      "inputNodes": [5, 7],
      "operation": "MERGE",
      "tokenIndices": [2, 3, 4, 5, 6]
    },
    "9": {
      "arguments": {
        "mergeAttribute1": "cast.person_id",
        "mergeAttribute2": "person.id"
      },
      "inputNodes": [8,2],
      "operation": "MERGE",
```

```
      "tokenIndices": [2, 3, 4, 5, 6]
    },
    "11": {
      "arguments": {
        "attribute": "movie.budget",
        "comparisonOperator": ">",
        "value": "100000000"
      },
      "inputNodes": [9],
      "operation": "FILTER",
      "tokenIndices": [2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]
    },
    "13": {
      "arguments": {
        "attribute": "movie.title"
      },
      "inputNodes": [
        11
      ],
      "operation": "EXTRACT VALUES",
      "tokenIndices": [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11,
                       12, 13]
    }
  }
}
```

A.3. Längste Query der Auswertung

Die Frage für diesen Loggingbaum mit 17 Operationen lautet: *"In what country were Hammer Film Productions' series of horror movies filmed?"*

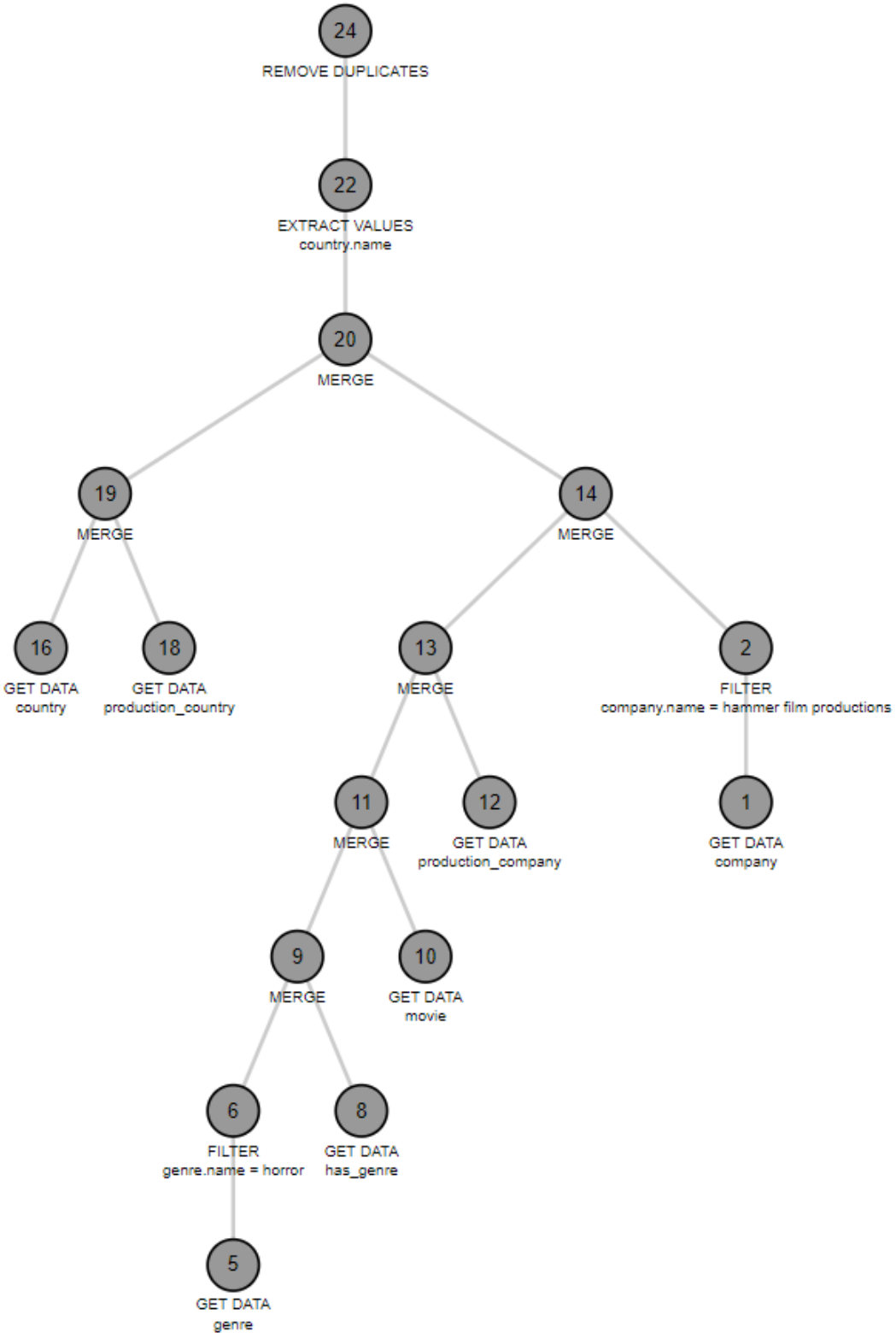

Abbildung A.2.: Der Loggingbaum der längsten Query während der Auswertung

A.4. Github-Dokumentation / Installationsanleitung

Getting started

Prerequisites

- `git` should be installed
- `python 3.7` should be installed
- `npm` should be installed
- `angular 7` or newer should be installed
  (this can be done using `npm install -g @angular/cli` in cmd)
- `mysql server` should be installed
- `mysql connector for python 3.7` should be installed

Install following python packages:

```
pip install flask
pip install flask_cors
pip install mysql-connector
pip install networkx
pip install inflection
pip install nltk
pip install numpy
pip install matplotlib
```

Setup

Clone the project to your filesystem.
The project consists of three folders: `documents`, `ba_backend` and `ba-frontend`.

- `documents` consists of the necessary MySQL-scripts to create the database.
- `ba_backend` consists of the Python API for the RESTful Web Service.
- `ba-frontend` consists of the Angular Single-Page-Application.
- `log_files` is the place where logging files, generated by the application, are stored

git clone

```
git clone https://github.engineering.zhaw.ch/PABA-kaisenic-schlaphi/BA_Code.git
```

Setup the database

Run the following sql-scripts in the `documents`-folder. ${USER} should be replaced with your mysql user. Enter your password for each step.

```
# create the database 'moviedata'
mysql -u ${USER} -p < movie_db_dump.sql

# create the inverted index of 'moviedata'
mysql -u ${USER} -p < reversed_index_complete_dump.sql
```

Run the RESTful Web Service / Backend

```
# change directory to the backend folder
cd ba_backend

# run file main.py
python main.py
```

The server listens now on `http://localhost:5000`.

Run the Angular Single-Page-Application / Frontend
Run using npm

```
# change directory to the frontend folder
cd ba-frontend

# You might want to install the modules first
npm install

# run frontend using npm
npm start
```

The single page application should now run on `http://localhost:4200`.

Team

Stakeholder

- Mark Cieliebak
- Kurt Stockinger
- Jan Milan Deriu

Development Team

- Nicolas Kaiser
- Philippe Schläpfer

A.5. REST-API

Die komplette Dokumentation der REST-API ist unter folgendem Link abrufbar:
`https://github.engineering.zhaw.ch/PABA-kaisenic-schlaphi/BA_Code/blob/master/REST_API.md`

A.6. Offizielle Aufgabenstellung

Titel:	DB4Dummies: Interaktive Konstruktion von Datenbank-Abfragen
Hauptbetreuer:	Mark Cieliebak
Nebenbetreuer:	Kurt Stockinger
Fachgebiete:	Datenanalyse (DA), Datenbanken (DB), Software (SOW)
Zuordnung:	Institut für angewandte Informationstechnologie (InIT)
Studiengang:	Informatik (IT)

Beschreibung:
Datenbanken enthalten sehr viel wertvolles, strukturiertes Wissen. Jedoch können nur Experten die SQL, SPARQL oder sonstige Abfragesprachen beherrschen direkt auf dieses Wissen zugreifen. Dies führt dazu, dass viele normalsterbliche Menschen keinen oder nur indirekten Zugriff auf diese Daten haben (entweder über einen Datenbank-Experten oder über eine Webapp).
In dieser Arbeit wollen wir ein System bauen, welches Datenbank-Operationen in kleine atomare Operationen aufteilt und über ein GUI dem User zur Verfügung stellt. Atomare Operatoren sind z.B.: Abruf einer Tabelle, Filtern von Resultaten, mathematische Operationen (Summe, Produkt, Maximum), Boolesche Operatoren, etc.
Der User soll in der Lage sein, Schritt-für-Schritt mit Hilfe dieser Operationen durch die Daten mittels GUI zu navigieren. Dadurch kann er unabhängig von Datenbankexperten Zugriff auf das Wissen erhalten. Aufgrund der Daten soll der Computer mittelfristig in der Lage sein, die Operationen für den User automatisch zu generieren.
Im HS 2018 fand eine Projektarbeit statt, in der ein Grundsystem von DB4Dummies implementiert wurde.

Aufgabenstellung:
Das Ziel dieser Arbeit ist, das existierende Grundsystem zu erweitern. Potentielle Erweiterungen sind z.B.

- Integration von neuen atomaren Operationen
- Anwendung auf neue Datenbanken
- Logging-Funktionalitäten um das Userverhalten zu verfolgen und Trainingsdaten für Machine Learning zu generieren
- Usability-Experimente mit Nicht-Experten
- Aufbau von Machine-Learning-Methoden, mit denen der Computer lernt, Fragen zu beantworten