Zürcher Hochschule
für Angewandte Wissenschaften

School of
Engineering

InIT Institut für angewandte
Informationstechnologie

Projektarbeit (Informatik)

Interaktive Konstruktion von Datenbankabfragen

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

Datum	21.12.2018

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

Mit der Abgabe dieser Projektarbeit 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 Projektarbeit 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 Projektarbeiten 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 den verschiedensten Aufgabengebieten interessant sein können. Leider ist der Zugang zu diesen Daten für Personen ohne Kenntnisse in einer Datenbanksprache 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 braucht jedoch eine grosse Menge an Trainingsdaten, um mittels Machine Learning daraus lernen und sich verbessern zu können.

Um diese Daten generieren zu können, soll ein Konzept erstellt und implementiert werden, welches anhand einer natürlichsprachlichen Frage die korrekte Antwort aus der Datenbank liefert. Zudem sollen Vorbereitungen getroffen werden, dass in einer weiteren Arbeit das Festhalten der vom Benutzer gewählten Operationen möglichst einfach umgesetzt werden kann.

Diese Arbeit beschäftigt sich mit dem Thema "Interaktive Konstruktion von Datenbankabfragen". Es wurde ein Konzept ausgearbeitet, wie 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 komplexe Abfrage zusammen, ohne auch nur eine Zeile SQL-Code schreiben zu müssen und erhält am Schluss das gewünschte Resultat. Dieses Konzept wurde erfolgreich implementiert. Der Prototyp stellt eine Webapplikation dar, welcher die geforderten Funktionen erfüllt. Weiterentwicklungen in Richtung Benutzeroberfläche könnten die Benutzerfreundlichkeit noch verbessern. Zudem müsste das Aufzeichnen der vom Benutzer gewählten Operationen noch fertig umgesetzt werden, um effektiv Trainingsdaten generieren zu können.

Inhaltsverzeichnis

1 Einleitung ... 9
1.1 Motivation ... 9
1.2 Aufgabenstellung ... 9
1.3 Gliederung des Dokuments ... 10
2 Theoretische Grundlagen ... 11
2.1 Jailer ... 11
2.1.1 Vorteile ... 11
2.1.2 Nachteile ... 12
2.1.3 Fazit ... 12
3 Konzept ... 13
3.1 Grundidee ... 13
3.1.1 Beispiel ... 13
3.2 Knoten ... 14
3.3 Atomare Operationen ... 14
3.4 Zwischenresultate ... 14
4 Umsetzung ... 15
4.1 Architektur ... 15
4.2 Invertierter Index ... 15
4.2.1 Erstellen des invertierten Index ... 15
4.2.2 Verwendung in der Applikation ... 16
4.3 Datenbankschema ... 17
4.4 Datenbank-Graph ... 18
4.5 Datentypen ... 18
4.6 Knotentypen ... 19
4.6.1 Beispiel ... 19
4.7 Implementierte atomare Operationen ... 20
4.7.1 Join ... 20
4.7.2 Selektion ... 22
4.7.3 Projektion ... 23
4.7.4 Distinct ... 24
4.7.5 Aggregatsoperationen ... 24
4.8 Suchverlauf ... 25
4.9 Verwendete Technologien ... 26
4.9.1 Backend ... 26
4.9.2 Frontend ... 26

5 Ergebnisse 27
5.1 Applikation 27
5.2 User Tests 28
5.2.1 Erkenntnisse 28
5.2.2 Fazit 29
5.3 Was kann das Tool (noch) nicht? 29
5.3.1 Unlösbare Fragentypen 29
5.3.2 Datumsformate 30
5.3.3 Synonymproblematik 30
6 Diskussion und Ausblick 31
6.1 Weitere atomare Operationen 31
6.2 Logging und Machine Learning 31
6.3 Datenbankenkompatibilität 31
6.4 Usability 31
6.5 DB-Graph erweitern 32
7 Verzeichnisse 33
7.1 Quellenverzeichnis 33
7.2 Abbildungsverzeichnis 34
8 Anhang 35
8.1 Offizielle Aufgabenstellung 35
8.2 Github-Dokumentation / Installationsanleitung 36
8.3 REST-API 37

1 Einleitung

1.1 Motivation

Datenbanken enthalten sehr viele strukturierte Daten, die für Personen aus den verschiedensten Aufgabengebieten interessant sein können. Leider sind für den Zugang zu diesen Daten 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 besorgen, könnte mit einer Applikation gelegt werden, welche Datenbankexperten verschiedene Abfragen durchführen lässt und deren Vorgehensweise dokumentiert.

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 SQL-Code geschrieben werden muss. Der Benutzer soll sich dabei die benötigten atomaren Operationen selbst zusammenstellen und auf der Datenbank ausführen können. 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.

Das Tool soll möglichst unabhängig von der Datenbank sein, so dass in Zukunft eine beliebige MySQL-Datenbank dem Tool zur Verfügung gestellt, bzw. die Datenbank bei Bedarf mit möglichst wenig Aufwand ausgetauscht werden kann. Die Applikation soll eine ansprechende Benutzeroberfläche haben, so dass die Bedienung möglichst einfach und simpel ist.

In einer späteren Version der Applikation soll es möglich sein, die Vorgehensweise eines Benutzers zu speichern. Genauer gesagt soll es möglich sein, die Reihenfolge und die Art der verschiedenen Operationen festzuhalten, welche der Benutzer wählt, um seine Frage zu beantworten. Um dies bewerkstelligen zu können, sollen Mechanismen in die grafische Oberfläche eingebaut werden, die es dem Benutzer ermöglichen, zu erfassen, welche Operation für welchen Teil der Frage entscheidend ist. Es soll also auch darauf geachtet werden, dass dieser Vorgang später in einer geeigneten Form aufgezeichnet werden kann.

1.3 Gliederung des Dokuments

Zu Beginn des Berichts wird ein Tool vorgestellt, welches in eine ähnliche Richtung geht wie die Aufgabenstellung. Vor- und Nachteile dieses Tools werden aufgezeigt, sowie ein kurzes Fazit gezogen. Anschliessend wird das Konzept für diese Arbeit vorgestellt. Darin wird die Grundidee beschrieben mit Hinblick auf das Aufzeichnen der Schritte eines Benutzers. Es wird zudem erklärt, was genau unter einer atomaren Operation verstanden wird.

Im vierten Kapitel wird die Umsetzung, also die eigentliche Implementation der Applikation beschrieben. Am Anfang des Kapitels wird auf die Architektur eingegangen. Anschliessend werden die verschiedenen Funktionalitäten erklärt und wie sie umgesetzt wurden. Ein wesentlicher Bestandteil dieses Kapitels sind die verschiedenen atomaren Operationen, welche in dieser Arbeit implementiert wurden. Am Ende des Kapitels werden die verwendeten Technologien erwähnt.

Das darauffolgende Kapitel zeigt die Ergebnisse der Arbeit auf. Während der Projektarbeit wurden Usability-Tests mit Benutzern durchgeführt, deren Erkenntnisse in diesem Kapitel beschrieben werden. Zudem wird darauf eingegangen, wo die Grenzen des Tools liegen und wie konkrete, derzeit noch ungelöste Probleme, sowie mögliche Lösungsansätze aussehen.

Zum Schluss wird diskutiert, was am Tool erweitert werden könnte und wie mögliche Lösungsvarianten der im vorherigen Kapitel erwähnten Problemstellungen aussehen könnten.

2 Theoretische Grundlagen

Viele Datenbankverwaltungssysteme wie beispielsweise MySQL-Workbench [1] geben dem Benutzer die Möglichkeit, das Datenbankschema genauer zu betrachten und so herauszufinden, wie die Datenbank aufgebaut ist [2]. Alle weiteren Schritte müssen jedoch mittels SQL-Abfragen ausgeführt werden. Solche Tools richten sich an DB-Experten mit der Absicht, deren Arbeit mit Datenbanken zu erleichtern.

2.1 Jailer

Es existiert jedoch ein Tool, welches die Navigation durch Datenbanken mit nur minimalem Einsatz der Abfragesprache SQL erlaubt. Dieses Tool nennt sich Jailer und ist ein Open-Source-Projekt [3]. Jailer bietet die Möglichkeit, eine beliebige Datenbank, die analysiert werden soll, zu importieren. Das Tool ist aufgeteilt in einen Bereich, mit dem das Datenbankschema analysiert werden kann und einen zweiten Teil, mit dem die Daten durchsucht werden können.

2.1.1 Vorteile

Das Datenbankschema wird anhand eines Graphen dargestellt, bei dem die Tabellen über die Fremd-/Primärschlüssel-Beziehungen miteinander verbunden sind. Die Navigation wird durch Klicks auf die einzelnen Graphknoten ermöglicht. Bei Bedarf können zudem die Attribute einer Tabelle angezeigt werden. Im Schema-Browser kann auf eine bestimmte Tabelle fokussiert werden, so dass alle ihre Nachbartabellen rundherum angezeigt werden. So kann sich der Benutzer durch das gesamte Datenbankschema navigieren. Die Darstellung des Schemas ist sehr übersichtlich, selbst wenn viele Tabellen gleichzeitig angezeigt werden. Das Beispiel in Abbildung 2.1 zeigt ein Ausschnitt eines solchen Datenbankschemas wie es in Jailer dargestellt wird.

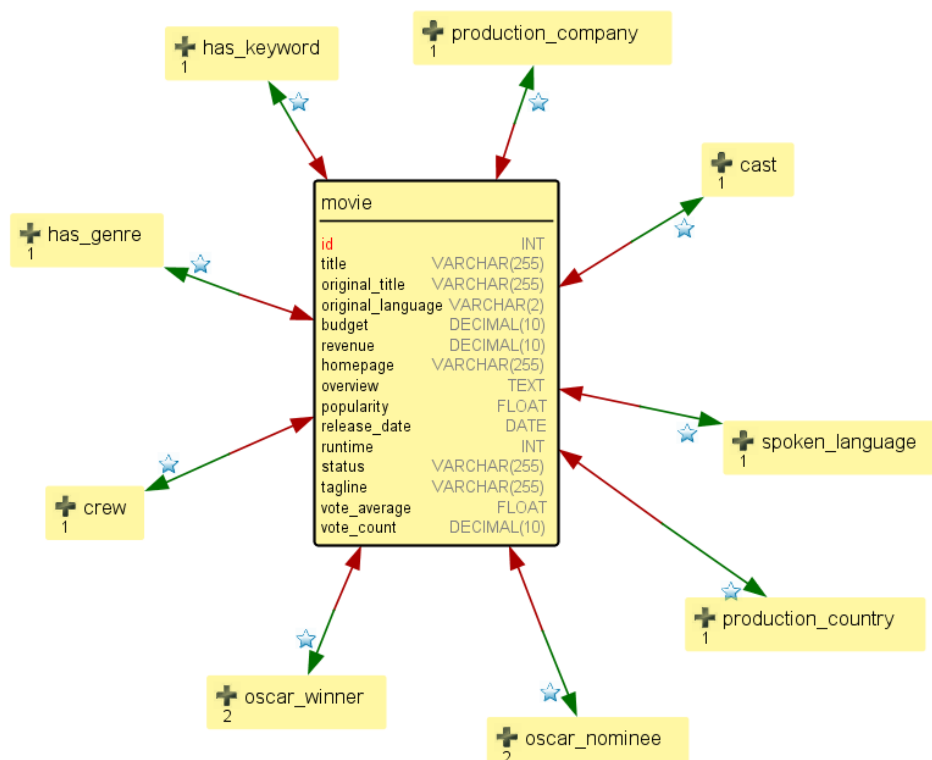

Abbildung 2.1: Schema-Browser mit erweiterten Attributinformationen der Tabelle „movie“

Von einer Tabelle im Schema-Browser aus kann der Daten-Browser geöffnet werden. Der Daten-Browser eignet sich dafür, die Daten, welche sich in einer Tabelle befinden, zu durchsuchen. Mit Hilfe der Filter-Funktion können WHERE-Klauseln für Tabellen gesetzt werden. Die Tabelle wird direkt anhand der WHERE-Klausel gefiltert.
Ausgehend von einer Tabelle kann mit direkten Nachbarn gejoint werden. In der gejointen Tabelle werden die Einträge anhand des Fremdschlüsselattributs gruppiert. Wenn ein bestimmter Eintrag in der Ausgangstabelle ausgewählt wird, werden in der gejointen Tabelle alle mit dem ausgewählten Eintrag markiert, wie in Abbildung 2.2 zu sehen ist. Diese zwei Funktionen sind sehr nützlich, um sich durch die Datenbank zu navigieren und sich einen Überblick über die Datenbank zu verschaffen.

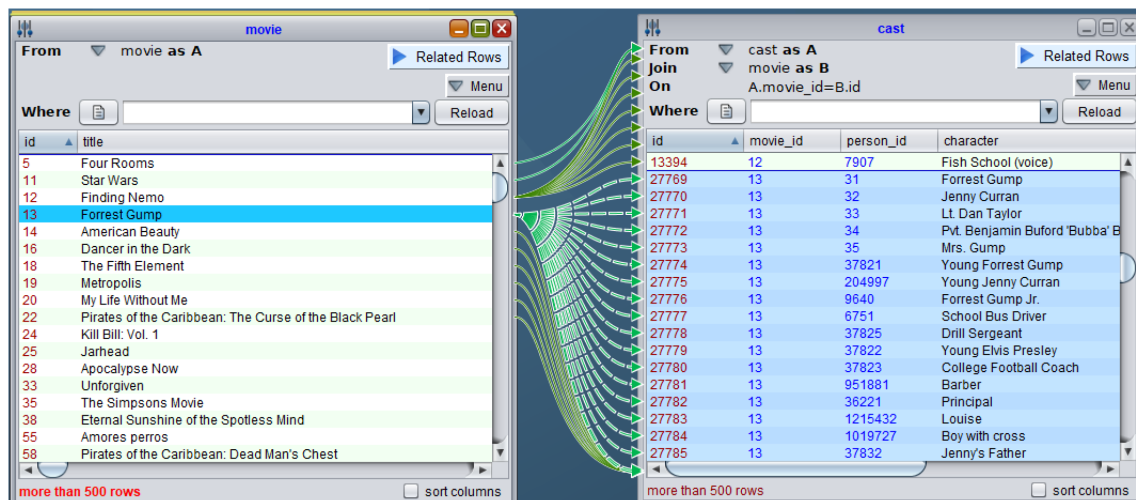

Abbildung 2.2: Daten-Browser mit einem ausgeführten Join auf eine Nachbartabelle

2.1.2 Nachteile

Das Tool bietet keine Möglichkeit, die gesamte Datenbank nach einem bestimmten Begriff oder mehreren Begriffen zu durchsuchen. Der Benutzer muss also anhand des Schema-Browsers zuerst herausfinden, wo die gewünschten Daten gespeichert sind, bzw. sein könnten, ehe er sich dann der Suche widmen kann.
Da das Tool nur Joins mit direkten Nachbartabellen zur Verfügung stellt, kann es unübersichtlich werden, sobald über mehrere Tabellen gejoint werden muss, um das gewünschte Resultat zu erhalten.
Jailer unterstützt nur sehr wenige Funktionen, um Daten weiter zu bearbeiten, nachdem Tabellen miteinander gejoint wurden. Es werden zwar die Anzahl Datensätze angezeigt, die eine Selektion oder ein Join als Resultat zurückgegeben haben, weitere Aggregatsfunktionen werden jedoch nicht zur Verfügung gestellt. Zudem ist es auch nicht möglich, auf bestimmte Attribute zu projizieren.

2.1.3 Fazit

Jailer bietet einige nützliche Funktionen, um sich einen ersten Eindruck von einer Datenbank zu verschaffen. Die Navigation durch die Datenbank ist durch den Graphen sehr benutzerfreundlich und übersichtlich. Einfache Join-Abfragen können zudem ohne viel Aufwand und Datenbankwissen ausgeführt werden. Komplexere Abfragen jedoch, welche aus vielen verschiedenen Operationen bestehen, sind mit dem Tool entweder sehr aufwändig zu erstellen oder erst gar nicht möglich, ohne dass der Benutzer SQL-Kenntnisse besitzt.

3 Konzept

3.1 Grundidee

Die Grundidee der Applikation besteht darin, dass der Benutzer eine Frage in das Tool eingeben und sich anschliessend, durch das Ausführen von verschiedenen Operationen, durch die Datenbankabfrage "durchklicken" kann. Die gewünschte Frage kann, falls nötig, unterteilt werden, so dass der Benutzer Teilresultate erzielen und diese zum Endresultat kombinieren kann. Es wurde entschieden, die verschiedenen Teilresultate im Hintergrund in einer Baumstruktur, genauer gesagt in einem umgekehrten binären Baum, zu speichern, wie in Abbildung 3.1 dargestellt. Der Baum wird gegen unten immer kleiner, bis er letztendlich in der Wurzel mündet, welche die Antwort der kompletten Abfrage beinhaltet und somit das Endresultat darstellt. Eine Abfrage soll an verschiedenen geeigneten Startpunkten begonnen werden können und durch eine geeignete Wahl von Operationen im korrekten Ergebnis resultieren. Ein Startpunkt ist dabei geeignet, wenn es sich um einen Begriff handelt, welcher in der Datenbank enthalten ist, da die erste Operation einer Abfrage immer aus einer Volltextsuche durch die Datenbank besteht. Dies dient dazu, überhaupt mal einen Ausgangspunkt zu erhalten. Solch ein Startpunkt ist gleichzeitig ein Blattknoten in unserer Baumstruktur. So werden für komplexere Fragen zuerst verschiedene Teile einer Frage beantwortet. Diese Teilantworten können gespeichert und dann später miteinander verbunden werden.

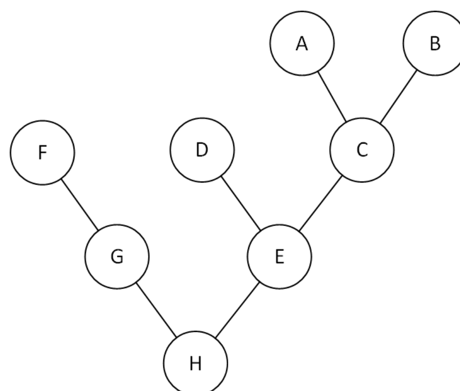

Abbildung 3.1: Beispiel eines Logging-Baums

3.1.1 Beispiel

Die gewünschte Frage sei folgende: *„In how many movies did George Clooney play?"*

Als erstes wird die Frage tokenisiert, sprich jedes Wort der Frage wird zu einem möglichen Startbegriff. Der Benutzer kann nun auswählen, mit welchen Begriffen er starten möchte, um sich durch die komplette Abfrage zu „hangeln". Es spielt dabei keine Rolle, ob er mit den Begriffen *„George Clooney"* oder *„movies"* beginnt, in beiden Fällen gelangt er zum Ziel. Wählt der User jedoch *„did play"* als Startbegriff, wird er nicht zum Ziel gelangen, da dieser Begriff mit hoher Wahrscheinlichkeit nicht in der Datenbank vorkommt, sondern eine Beziehung von Zwischenresultaten darstellt.

In Abbildung 3.1 kann man gut erkennen, wie ein solcher Baum zustande kommt. Die Zwischenresultate A und B werden zusammengefügt, so dass ein neuer Knoten C entsteht. Dieser wird wiederum mit dem Zwischenresultat D verbunden, so dass der Knoten E entsteht.

Durch jede Suche im invertierten Index wird jeweils ein eigener Teilbaum generiert, welcher entweder durch Operationen weiter verfeinert werden kann, oder mit weiteren Teilbäumen zu einem grösseren Baum verbunden werden kann. Das Endresultat der Abfrage würde im oben genannten Beispiel im Knoten H stehen.

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 Daten der Datenbank auch die Suche durch Metadaten wie die Tabellen- und Spaltennamen.

3.2 Knoten

Ein Knoten ist ein Bestandteil des binären Suchbaumes und stellt einen Zustand, bzw. eine Teilantwort einer Frage dar. Der Zustand wird mithilfe einer Tabelle dargestellt. Auf einem Knoten können diverse Operationen durchgeführt werden, um das aktuelle Resultat zu verändern, bzw. zu verfeinern, so dass man sich schrittweise dem gewünschten Endresultat nähern kann. Nach einer ausgeführten Operation wird das neue Resultat in einem neuen Knoten gespeichert und der neue Knoten dem aktuellen Suchbaum angehängt.

Ein Vorteil dieses Modells ist, dass in einem Knoten nicht nur das Ergebnis einer Operation gespeichert werden kann, sondern auch weitere Informationen, welche für das zukünftige Logging wichtig sind. Dazu gehören bestimmte Wörter einer Frage, die der Benutzer mit dieser Operation assoziiert. So können etwas abstraktere Begriffe wie zum Beispiel das Wort *„play“* mit einer bestimmten Operation in Verbindung gebracht werden. Diese Assoziationen sind wichtig, damit das Tool in einem späteren Schritt lernen kann, welche Operationen für welche Formulierungen oder Wörter in Frage kommen.

3.3 Atomare Operationen

Ein relationales Datenbankmanagementsystem (DBMS) bietet sehr viele verschiedene Operationen an, welche auf einer Datenbank ausgeführt werden können. Im Prototypen der Anwendung werden die wichtigsten der dort zur Verfügung stehenden Operationen bereits unterstützt, so dass man einen Eindruck bekommen kann, was für Abfragen mit einem solchen Tool überhaupt durchgeführt werden können, ohne auch nur eine Zeile SQL-Code zu schreiben.
Eine atomare Operation wird von der Applikation als einzelner Schritt angesehen, den ein Benutzer auf Daten der Datenbank ausführen kann. Dabei ist eine atomare Operation des Tools nicht dasselbe wie eine atomare Operation in einem DBMS. Ein solcher Schritt wird - wie oben beschrieben - als Knoten in einem binären Baum gespeichert.
Eine atomare Operation in dieser Applikation ist so definiert, dass sie als Input einen oder zwei Knoten erhält, und als Output einen neuen Knoten zurückgibt, der dann im Baum als Nachfolgeknoten eingefügt wird.

3.4 Zwischenresultate

Der Benutzer hat die Möglichkeit, bereits gefundene Zwischenresultate zu speichern. Diese sind, wie bereits weiter oben erwähnt, in Form eines Knotens im Suchbaum abgelegt. Speichert der Benutzer das Zwischenresultat, wird der komplette aktive Suchbaum persistiert, so dass jederzeit alle vorangegangenen Zustände wiederhergestellt werden können.
Der Benutzer kann sich nach dem Zwischenspeichern mit einem weiteren Teil der Abfrage beschäftigen und das gespeicherte Zwischenresultat für eine zukünftige Operation als Inputparameter verwenden.

4 Umsetzung

4.1 Architektur

Die grundlegende Architektur dieser Applikation ist eine REST-Architektur, wie sie in Abbildung 4.1 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. Die komplette Dokumentation der REST-Schnittstelle befindet sich im Anhang in Kapitel 8.3.

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

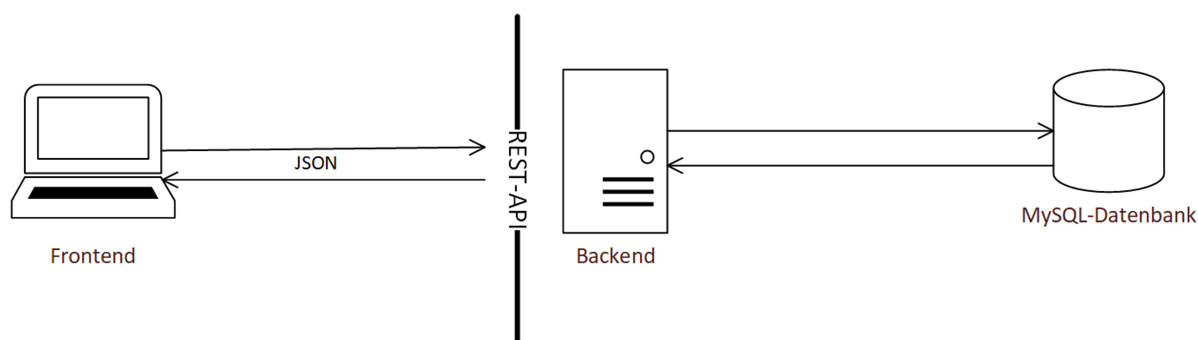

Abbildung 4.1: Grundlegende Architektur der Applikation

4.2 Invertierter Index

4.2.1 Erstellen des invertierten Index

Damit der Benutzer einen Einstiegspunkt in die Datenbank erhält, muss er die Datenbank nach Begriffen durchsuchen können. Damit die Suche über eine Datenbank nicht zu lange dauert, wird ein invertierter Index von der Datenbank erstellt. Dieser ermöglicht eine Art Volltextsuche über die kompletten Daten, Attributnamen und Tabellen der Datenbank.

Der invertierte Index besteht aus den folgenden drei Tabellen:

- **Tabellennamen**
  Die Namen von allen Tabellen der Datenbank werden hier gespeichert. Bei einem Klick auf einen solchen Eintrag, wird dem Benutzer die ausgewählte Tabelle angezeigt.

  Beispiel:

table
person
movie
oscar

- **Attributnamen**
  Zu allen Tabellennamen werden die zugehörigen Attributnamen gespeichert. Bei einem Klick auf einen solchen Eintrag, wird dem Benutzer die gesamte Tabelle angezeigt, welche das gewünschte Attribut enthält.

  Beispiel:

attribute	table
name	person
birthday	person
title	movie

- **Attributwerte**
  Für jeden Wert, der in irgendeiner Tabelle gespeichert ist, wird der eigentliche Wert und wo er zu finden ist (in welcher Tabelle, in welchem Attribut) gespeichert. Bei einem Klick auf einen solchen Eintrag, wird dem Benutzer der spezifische Record in der Tabelle angezeigt. Dazu wird dem erstellten Knoten eine Bedingung (beispielsweise "person.name = `george clooney`") hinzugefügt. Diese Bedingung wird beim Weiterverarbeiten des Zwischenresultates jeweils mitgegeben.

  Beispiel:

value	attribute	table
george clooney	name	person
inception	title	movie
paris	birth_place	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 sogenannte Trigger implementiert werden, welche ausgeführt werden, sobald etwas an einer Datenbank geändert wird. Die Aktualisierung des invertierten Index wurde im Prototypen noch nicht umgesetzt. Die verschiedenen Tabellen des invertierten Index werden in Stored Procedures von MySQL erstellt.

4.2.2 Verwendung in der Applikation

Wenn der Benutzer eine Anfrage an das System stellt, muss er als Erstes Begriffe aus seiner gestellten Frage auswählen. Anhand von diesen Begriffen wird der invertierte Index durchsucht. Dabei werden alle möglichen Kombinationen aus den Begriffen durchprobiert. Dies wird so lange durchgeführt, bis im invertierten Index ein Treffer erzielt wird.

Bei einer Anfrage mit n Wörtern wird zuerst nach allen Kombinationen mit der Länge n gesucht. Falls nichts gefunden wurde, wird nach allen Kombinationen der Länge n-1 gesucht, dann n-2 und so weiter. So lange, bis ein Eintrag im invertierten Index gefunden wurde.

Zusätzlich werden alle Begriffe sowohl in ihrer Plural-, als auch in ihrer Singularform abgefragt. Es kommt oft vor, dass Wörter in einer Frage in der Pluralform geschrieben, in der Datenbank aber in der Singularform abgespeichert sind. Mit dieser Massnahme kann die Ausbeute erhöht werden.

tables
movie

attribute	*table*
budget	movie
movie_id	crew
movie_id	cast

Keywords found in following tables and attributes	
company, name	12 entries
cast, character	1 entries

Abbildung 4.2: Beispiel einer Anfrage an den invertieren Index

Im Index wird nach Substring-Matches gesucht. Es muss also nicht das ganze Wort exakt vorkommen. Es reicht, wenn der Suchbegriff mit einem Teil eines im Index enthaltenen Wortes übereinstimmt. Wurde nach allen möglichen Suchkombinationen immer noch nichts gefunden, muss der Benutzer andere Begriffe der Frage auswählen und die Suche durch den invertierten Index nochmals starten. Abbildung 4.2 zeigt ein Beispiel für die Anfrage *„movies budget"* an den invertierten Index.

Dem Benutzer werden nun die Ergebnisse der Suche durch den invertierten Index dargestellt. Als erstes wurde die Datenbanktabelle *„movie"* gefunden. Als nächstes wurden drei Attribute von verschiedenen Datenbanktabellen gefunden, welche einen Teil der Suchbegriffe enthalten. Zuunterst werden dem Benutzer die Treffer aus den eigentlichen Daten der Datenbank angezeigt. Diese Ergebnisse werden anhand des Tabellennamens und Attributnamens gruppiert. Beispielsweise wurden in Abbildung 4.2 zwölf Firmen gefunden, welche mindestens einen der Suchbegriffe im Firmennamen enthält.

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 Daten, welche dazu benötigt werden, werden aus der Metadaten-Tabelle INFORMATION_SCHEMA gelesen, welche in jeder MySQL-Datenbank automatisch von MySQL aufgebaut und verwaltet wird.
Der Graph kann für eine beliebige MySQL-Datenbank aufgebaut werden, damit das Tool unabhängig von spezifischen Datenbanken bleibt. Die Graphstruktur wird mithilfe der Python-Library networkx verwaltet [4].

Im Graph wird für jede Tabelle einen Knoten erstellt. Bei jedem Knoten wird zusätzlich gespeichert, auf welche Attribute die Primär- und Fremdschlüssel zeigen. Die Kanten werden anhand der Primär-/Fremdschlüssel-Beziehungen aufgebaut. Der Graph ist ungerichtet, weil zwischen zwei Tabellen A und B entweder von Tabelle A nach B oder umgekehrt von Tabelle B nach A gejoint werden kann.

4.4 Datenbank-Graph

Der Graph, welcher die Datenbank abbildet, kann während dem Beantworten einer Frage durch den Benutzer eingeblendet werden. Dies soll 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.

In Abbildung 4.3 wird gezeigt, wie Tabellen im Graph markiert werden, welche sich im aktuellen Zwischenresultat befinden. Ebenso werden alle Pfade, über die gejoint wurde, rot markiert.

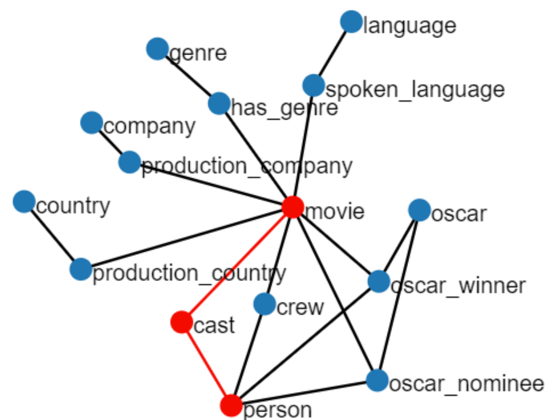

Abbildung 4.3: Der Graph einer Datenbank nach manuellem Verschieben der Knoten

Aktuell wird der Graph automatisch ausgerichtet, die Darstellung ist jedoch noch sehr unübersichtlich. Aufgrund unseres Fokus auf die verschiedenen Datenbankoperationen wurde nur begrenzt Zeit in die Darstellung des Graphen investiert, weshalb die automatische Ausrichtung sicherlich noch nicht zufriedenstellend umgesetzt ist. Dem Benutzer wird jedoch die Möglichkeit geboten, die einzelnen Knoten nach Belieben von Hand zu verschieben.

Die graphische Darstellung des Schemas wurde mit Hilfe der Javascript-Library D3.js implementiert [5].

4.5 Datentypen

In einem ersten Schritt wurde entschieden, einfachheitshalber nicht jeden einzelnen Datentyp von MySQL zu unterstützen. Beschränkt wurde sich auf die gängigsten Datentypen, jedoch 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) [6]
- Alphanumerische Datentypen
  (char, varchar, text) [7]

Datumsformate [8] werden aktuell nicht unterstützt, bzw. werden als String behandelt.

4.6 Knotentypen

Zu jedem Zeitpunkt befinden sich die Zwischenresultate in einem Knoten des binären Baumes. Die verschiedenen Knotentypen werden benötigt, um zu unterscheiden, auf welchen Zwischenresultaten welche Operationen möglich sein sollen.

Es wird zwischen diesen Knotentypen unterschieden:

- **InvertedIndexNode**
  Spezieller Knoten, der als einziger ein Resultat des invertierten Indexes enthalten kann.
- **SQLNode**
  Dieser Knoten enthält die Resultate einer Datenbankabfrage.
- **NumericListNode**
  Dieser Knoten beinhaltet ein Resultat, welches aus nur einer Tabellenspalte besteht. Der Datentyp dieser Spalte muss numerisch sein. Hierbei wird nicht unterschieden, ob der Datentyp eine Gleitkommazahl oder eine ganze Zahl darstellt. Zusätzlich müssen mehrere Zeilen im Resultat enthalten sein.
- **NumericNode**
  Dieser Knoten ist das Pendant zum NumericListNode. Der einzige Unterschied ist hierbei, dass das Resultat nur aus genau einer Zeile besteht.
- **StringListNode**
  Dieser Knoten beinhaltet ein Resultat, welches aus nur einer Tabellenspalte besteht. Der Datentyp dieser Spalte muss alphanumerisch sein. Zusätzlich müssen mehrere Zeilen im Resultat enthalten sein.
- **StringNode**
  Dieser Knoten ist das Pendant zum StringListNode. Der einzige Unterschied ist hierbei, dass das Resultat nur aus genau einer Zeile besteht.

4.6.1 Beispiel

Das aktuelle Zwischenresultat besteht aus folgender Tabelle:

movie.budget
100‘000‘000
120‘000‘000
80‘000‘000

In dieser Tabelle könnten die Budgets von drei verschiedenen Filmen abgespeichert sein. Da die Tabelle aus nur einer Spalte besteht, jedoch mehrere Einträge hat, wird sie in einem NumericListNode gespeichert. Dieser stellt beispielsweise die Funktion AVERAGE zur Verfügung, welche den Durchschnitt der vorhandenen Werte berechnet. Der Knotentyp NumericListNode ist zudem der einzige, welcher AVERAGE anbietet, da es nur Sinn macht, aus einer Liste von numerischen Werten den Durchschnitt zu berechnen. Das Resultat der AVERAGE-Operation, welches aus einem numerischen Wert besteht, wird nun in einem NumericNode gespeichert.

4.7 Implementierte atomare Operationen

In dieser Arbeit wurden die wichtigsten Operationen umgesetzt, so dass ein Grossteil der aus unserer Sicht häufigsten Fragen beantwortet werden kann. Zu einem späteren Zeitpunkt können theoretisch beliebig viele Operationen hinzugefügt werden.

4.7.1 Join

Die Applikation unterscheidet zwischen zwei verschiedenen Arten von Joins. Mit dem einen Join kann ein SQL-Node mit einer direkt umliegenden Tabellen gejoint werden. Mit der anderen Art können zwei beliebige Zwischenresultate miteinander verbunden werden. Abbildung 4.4 zeigt die Definition der Join-Operation in Pseudocode.

```
func join(node1: SQLNode, node2: SQLNode) returns SQLNode {
    newNode = performJoin(node1, node2)
    return newNode
}
```

Abbildung 4.4: Definition der Join-Operation in Pseudocode

4.7.1.1 Join mit umliegenden Tabellen

Bei diesem Typ von Join werden einem Benutzer alle umliegenden Tabellen zum Joinen vorgeschlagen. Als umliegende Tabellen werden Tabellen verstanden, welche eine direkte Verbindung per Fremd-/Primärschlüssel zu den Tabellen haben, welche sich bereits im Zwischenresultat befinden. Diese umliegenden Tabellen können als direkte Nachbarn im Graph des Datenbankschemas verstanden werden.

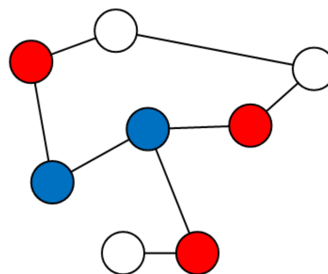

Abbildung 4.5: Ein Zwischenresultat in einem Graph mit den direkten Nachbarn

Im Beispiel von Abbildung 4.5 sind die blauen Knoten (Tabellen) schon im aktuellen Zwischenresultat vorhanden. Die roten Knoten werden als direkte Nachbarn erkannt. Das heisst, dem Benutzer werden in diesem Beispiel alle rot markierten Tabellen vorgeschlagen, um den Join durchzuführen. Dieser Join ist vor allem nützlich, wenn man noch keine genaue Vorahnung hat, welche Daten in welchen Tabellen gespeichert sind, sprich um sich einen Überblick über die Datenbank zu verschaffen.

4.7.1.2 Join mit Zwischenresultaten

Es ist auch möglich, zwei verschiedene SQL-Nodes miteinander zu verbinden. Dabei kommt es nicht darauf an, wie weit auseinander sich die Tabellen befinden. Falls die Tabellen direkte Nachbarn sind, wird auf den Fall „Join mit umliegenden Tabellen" zurückgegriffen. Falls sich jedoch keine der Tabellen direkt nebeneinander befinden, wird mit Hilfe des Algorithmus von Dijkstra der kürzeste Pfad zwischen den Tabellen bestimmt. Es kann vorkommen, dass es mehrere kürzeste Pfade zwischen Tabellen gibt. In diesem Fall werden dem Benutzer alle kürzesten Pfade zur Auswahl angezeigt.
In Abbildung 4.6 ist ein Beispiel dazu gegeben.

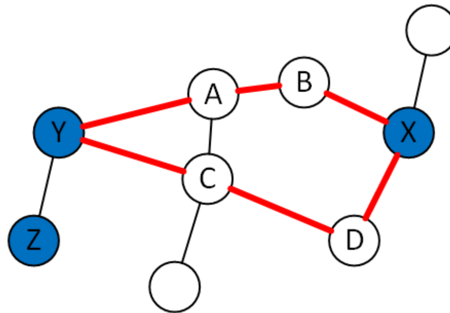

Abbildung 4.6: Zwei Zwischenresultate (blau) mit den kürzesten Pfaden dazwischen (rot)

Aktuell sind zwei SQL-Nodes (in blau) vorgegeben. Der Benutzer hat sich also diese beiden Zwischenresultate mit Hilfe von vorangegangenen Operationen erstellt. Im einen Zwischenresultat befinden sich die Tabellen Y und Z, im anderen die Tabelle X. Der Benutzer möchte nun diese zwei Knoten miteinander verbinden. Die Applikation findet also zwei kürzeste Pfade zwischen diesen Tabellen (X → B → A → Y; X → D → C → Y), beide mit einem Gewicht von 3.
Der Benutzer kann nun entscheiden, über welchen dieser Pfade er joinen will. Die beiden Pfade führen zu einem unterschiedlichen Resultat, schliesslich kommen verschiedene Tabellen darin vor. Anhand der Tabellennamen muss nun entschieden werden, welcher Pfad die Frage von ihm beantworten wird. Bei dieser Operation greift die Vereinfachung, dass von sprechenden Tabellennamen ausgegangen wurde.

Abbildung 4.7: Das Menü für die zwei verschiedenen Join-Möglichkeiten

Abbildung 4.7 zeigt die graphische Oberfläche für die zwei verschiedenen Join-Operationen. In diesem Beispiel existiert bereits ein vom Benutzer erstelltes Zwischenresultat mit dem Namen *„movies"*. Im zweiten Auswahlfeld kann der User von allen direkten Nachbarn eine Tabelle auswählen, falls er den Join mit einer umliegenden Tabelle bevorzugt.

4.7.2 Selektion

Mit der Selektion können die Einträge in der Tabelle des Zwischenresultates weiter eingeschränkt werden. Dem Knoten wird eine weitere Zwangsbedingung angehängt und das Resultat wird nach der Bedingung gefiltert. Die Definition in Pseudocode ist in Abbildung 4.8 dargestellt.

```
func selection(node: SQLNode, constraint: String) returns SQLNode {

    return node.filter(constraint)

}
```

Abbildung 4.8: Definition der Selektionsoperation in Pseudocode

Hierzu ein Beispiel:

movie.budget
100‘000‘000
120‘000‘000
80‘000‘000

```
selection(node, `movie.budget > 100000000`)
```

movie.budget
120‘000‘000

Die bestehende Tabelle könnte in diesem Fall Budgets von verschiedenen Filmen enthalten. Nun ist der Benutzer an Budgets über 100 Mio. interessiert und führt eine Selektion mit der korrekten Zwangsbedingung durch. Somit erhält er eine neue Tabelle, welche nur noch den Wert 120 Mio. enthält.

Die Selektion ist auf folgende Knotentypen anwendbar:
- SQLNode
- NumericListNode
- StringListNode

Selection

Attribute: crew.job
Comparison operator: equals
Constraint value: Director

SEND CANCEL

Abbildung 4.9: Beispiel einer Selektion mit bereits abgefüllten Parametern

In Abbildung 4.9 zeigt das Selektions-Formular im Prototypen. In diesem Beispiel wird die Zwangsbedingung *„crew.job = `director`“* hinzugefügt.

4.7.3 Projektion

Mit einer Projektion können bestimmte Attribute eines SQL-Nodes angezeigt werden. Diese Operation eignet sich, wenn z.B. aufgrund von einigen Joins das Resultat viele Spalten enthält. Bevor das Zwischenergebnis zu unübersichtlich wird, können mit Hilfe der Projektion unwichtige Spalten weggelassen werden. Ein weiterer typischer Anwendungsfall dieser Operation ist es, am Schluss einer Abfrage noch auf ein bestimmtes Attribut zu projizieren, so dass nicht mehr der ganze Record angezeigt wird. Abbildung 4.10 zeigt, wie der Benutzer mit Hilfe der Projektion die für ihn wichtigen Attribute auswählen kann.

Die Projektion ähnelt derjenigen Projektion eines DBMS, jedoch mit einem wichtigen Unterschied. Im Hintergrund werden die Attribute, die nicht ausgewählt wurden nicht gelöscht, sie werden lediglich nicht mehr in der grafischen Oberfläche angezeigt. Das hat den Vorteil, dass weiterhin noch gejoint werden kann, auch wenn die Primär- oder Fremdschlüssel gar nicht mehr angezeigt werden. Dies ist bei einer Abfragesprache nicht der Fall.

Werden bei der Projektion mehrere Spalten ausgewählt, wird das Resultat der Operation weiterhin in einem SQL-Node gespeichert. Wird jedoch auf eine einzelne Spalte projiziert, wird folgende Fallunterscheidung gemacht:

- Datentyp der Tabellenspalte ist numerisch, mehrere Zeilen sind im Resultat
  ➔ Resultat wird in einem NumericListNode gespeichert
- Datentyp der Tabellenspalte ist numerisch, Resultat enthält nur eine Zeile
  ➔ Resultat wird in einem NumericNode gespeichert
- Datentyp der Tabellenspalte ist alphanumerisch, mehrere Zeilen sind im Resultat
  ➔ Resultat wird in einem StringListNode gespeichert
- Datentyp der Tabellenspalte ist alphanumerisch, Resultat enthält nur eine Zeile
  ➔ Resultat wird in einem StringNode gespeichert

Projection

- [ ] person.id
- [x] person.name
- [ ] person.gender
- [x] person.birth_day
- [x] person.birth_place
- [ ] person.death_day
- [ ] person.death_place

SEND CANCEL

Abbildung 4.10: Beispiel einer Projektion auf 3 Attribute

4.7.4 Distinct

Die DISTINCT-Funktion entfernt Duplikate aus der Tabelle des Zwischenresultates. Dies kann beispielsweise nützlich sein, wenn man beantworten möchte, mit welchen verschiedenen Produktionsfirmen George Clooney bereits zusammengearbeitet hat. Da er sicherlich mit verschiedensten Firmen mehrfach zusammengearbeitet hat, muss die DISTINCT-Funktion eingesetzt werden, um die Frage beantworten zu können.

DISTINCT ist auf folgende Knotentypen anwendbar:

- NumericListNode
- StringListNode

4.7.5 Aggregatsoperationen

4.7.5.1 COUNT

Bei der COUNT-Funktion wird die Anzahl Zeilen in der Tabelle des Zwischenresultats gezählt und in einem NumericNode gespeichert. Diese Funktion wird typischerweise als letzte Operation einer Abfrage angewendet, beispielsweise wenn beantwortet werden möchte, in wie vielen Filmen Brad Pitt mitgespielt hat.

COUNT ist auf folgende Knotentypen anwendbar:

- SQLNode
- NumericListNode
- StringListNode

4.7.5.2 AVERAGE

Die AVERAGE-Funktion berechnet aus den Einträgen eines NumericListNodes den Durchschnitt und speichert das Resultat in einem NumericNode.

4.7.5.3 SUM

Die SUM-Funktion berechnet die Summe aus allen Einträgen eines NumericListNodes und speichert das Resultat in einem NumericNode.

4.7.5.4 MIN

Die MIN-Funktion wählt aus den Einträgen eines NumericListNodes das Minimum und speichert das Resultat in einem NumericNode.

4.7.5.5 MAX

Die MAX-Funktion wählt aus den Einträgen eines NumericListNodes das Maximum und speichert das Resultat in einem NumericNode.

4.8 Suchverlauf

Das Tool stellt eine Übersicht über den aktuellen Suchverlauf zur Verfügung. In diesem Suchverlauf werden alle Zwischenresultate angezeigt, die der Benutzer seit der Suche im invertierten Index mit einem bestimmten Knoten erstellt hat. Diese Zwischenresultate stellen einen Teilbaum des gesamten Suchbaumes dar.

Der Benutzer kann beliebige Zwischenschritte im Verlauf auswählen, so dass ihm die zugehörigen Daten für diesen Zwischenschritt angezeigt werden. Falls der Benutzer falsche Operationen ausgeführt hat, dies jedoch erst im Nachhinein bemerkt, kann er zum Knoten zurückspringen, bevor die falsche Operation ausgeführt wurde und dort mit der korrekten Operation weiterfahren. Alle Operationen, welche nach der falschen Operation ausgeführt wurden, werden dann automatisch gelöscht.

Die Namensgebung für die einzelnen Operationen wurde so gewählt, dass die Bezeichnungen kurz bleiben und dennoch genügend Information vorhanden ist, damit der Benutzer erkennen kann, welche Informationen in welchem Zwischenschritt vorhanden sind, wie in Abbildung 4.11 zu sehen ist. Er kann auch nachvollziehen, mit welchen Schritten er auf das aktuelle Zwischenresultat gekommen ist.

Abbildung 4.12 zeigt denselben Verlauf wie Abbildung 4.11, allerdings dargestellt in einem binären Logging-Baum. Dieses Beispiel veranschaulicht, wie bei einem Join die verschiedenen Teilresultate zu einem Zwischenresultat zusammengefügt werden und wie das neue Zwischenresultat mit Hilfe von weiteren Operationen verfeinert wird.

Current History

Index result
george clooney
george clooney join movie
Projection of movie.title and movie.budget
Select movie.budget > 100000000

Abbildung 4.11: Beispiel eines Suchverlaufs

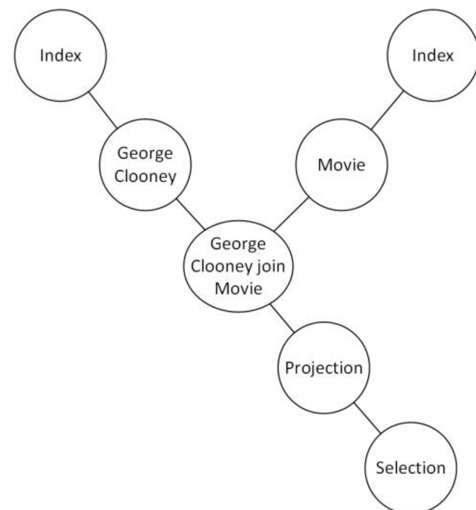

Abbildung 4.12: Der Suchverlauf aus Abbildung 4.11 dargestellt in einem binären Logging-Baum

4.9 Verwendete Technologien

4.9.1 Backend

- Python 3.7
- Flask 1.0.2
- MySQL 5.6

Für das Backend wurde Python verwendet, 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 in kurzer Zeit und mit relativ wenig Code eine funktionierende REST-API aufbauen lässt [9].
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.

Bis anhin unterstützt der Prototyp nur MySQL-Datenbanken.

4.9.2 Frontend

- Angular 6
- Angular Material 7

Das Frontend wurde mit Hilfe des Typescript-Frameworks Angular [10] geschrieben. Die Wahl fiel darauf, weil wir diese Technologie schon in vorherigen Projekten genutzt haben und dadurch keine Zeit für die Einarbeitung verloren ging. Angular Material [11] wurde verwendet, um die ganze Applikation einheitlich und ansprechend zu gestalten.

5 Ergebnisse

5.1 Applikation

Abbildung 5.1 zeigt den fertigen Stand der Applikation. Links befindet sich der Navigationsbereich, rechts davon sieht man die Frage, welche der Benutzer an das System gestellt hat, sowie das Ergebnis der aktuellen Operation.

Zuoberst in der Navigationsleiste werden alle gespeicherten Zwischenresultate angezeigt („Saved Results“). Die Zwischenresultate können vom Benutzer einzeln gelöscht werden. Direkt unter den Zwischenresultaten wird der Suchverlauf des aktuellen Strangs des Suchbaums angezeigt („Current History“). Die Zwischenschritte des Suchverlaufs können angewählt werden, so dass das Ergebnis der zugehörigen Operation angezeigt wird. Mit einem Klick auf den „Save“-Button wird der gesamte Suchverlauf bei den gespeicherten Zwischenresultaten angezeigt. Der aktuelle Suchverlauf wird dann zurückgesetzt. Unterhalb des Suchverlaufs blendet die Applikation die Operationen ein, die auf den aktuellen Knoten angewendet werden können. In Abbildung 5.1 sind das die Operationen „Selektion“, „Count“ und „Distinct“ welche auf dem Zwischenresultat vom Typ „StringListNode“ ausgeführt werden können.
Mit dem Knopf „Finish“ kann die Frage abgeschlossen werden und mit dem Knopf „Reset“ kann der gesamte Inhalt der Applikation zurückgesetzt werden, so dass eine neue Frage gestellt werden kann.

Rechts des Navigationsbereichs befindet sich die Frage, welche der Benutzer an das System gestellt hat. Unterhalb davon werden die Ergebnisse des aktuellen Knotens in Tabellenform angezeigt. Die Tabelle zeigt fünf Einträge pro Seite an, durch die der Benutzer blättern kann.

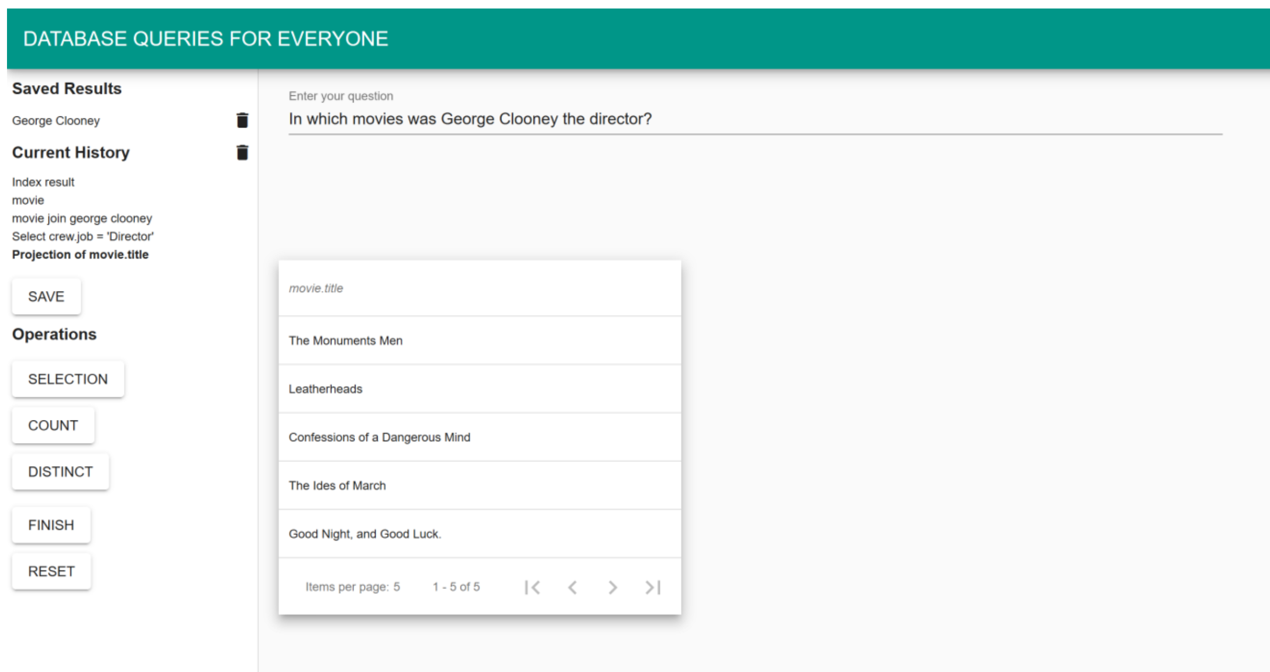

Abbildung 5.1: Screenshot der aktuellen Version unserer Applikation

5.2 User Tests

Um erste Eindrücke zu erhalten, wie das Tool bei potenziellen Benutzern ankommt und ob diese überhaupt mit der Bedienung klarkommen, wurden drei Testpersonen, welche alle über Datenbankwissen verfügen, gebeten, die Applikation zu testen.
Für diesen Benutzertest wurde eine Liste von fünf Fragen erstellt, welche sich mit dem Tool beantworten lassen. Die Komplexität der Fragen, bzw. die Anzahl benötigter Operationen, welche man ausführen muss, um das korrekte Resultat zu erhalten, stieg von Frage zu Frage.
Die drei Personen wurden zu Beginn des Tests nur kurz informiert, für was dieses Tool geeignet ist, allerdings nicht wie man das Tool bedient. Diese Vorgehensweise wurde so gewählt, um einen besseren Eindruck davon zu bekommen, welche Funktionen des Tools ein Benutzer intuitiv benutzt, resp. welche Operationen nicht selbsterklärend sind. Einen Hinweis gab es erst auf Anfrage der Testpersonen, nachdem sie nicht mehr weiterkamen. Von den weiter oben beschriebenen Funktionen waren zum Zeitpunkt der User Tests alle implementiert bis auf die graphische Darstellung des DB-Schemas (DB-Graph).

5.2.1 Erkenntnisse

Folgende Erkenntnisse haben wir aus den User Tests gewonnen:

- Die Funktion „Zwischenspeichern von Ergebnissen“ wurde ohne Erklärung nicht verwendet.
  → Gespeicherte Zwischenergebnisse wurden erst nach Erklärung für Joins verwendet.

- Die meisten Operatoren wurden ohne Erklärung nicht verwendet.
  → Hier war ebenfalls eine Erklärung notwendig.

- Die Tatsache, dass am Ende einer Abfrage eine Projektion durchgeführt werden muss, so dass man nur noch das gewünschte Resultat und nicht noch weitere ungewollte Tabellenspalten im Resultat hat, war nicht immer klar.
  → Nach einer Erklärung wurde das auch eingehalten.

- Das Ausfüllen des Popups für das Logging war unklar. Die Testpersonen wussten nicht, welche Begriffe sie anwählen sollten.
  → Eine Testperson wies darauf hin, dass sie mit dem Datenbankschema als Hilfe deutlich weniger Probleme gehabt hätte, zu erkennen, welche Informationen bereits im Resultat stecken.

- Teilweise war unklar, welche Informationen in welchen Tabellen enthalten sind.
  → Auch hier könnte das Datenbankschema helfen.

- Die Handhabung und Funktionsweise der Operationshistory war ohne Erklärung unklar
  → Eine Testperson wies darauf hin, dass ein „Undo/Redo“-Button wünschenswert wäre.

- Es hat sich gezeigt, dass Nomen als Startbegriffe sich mit Abstand am besten eignen für die Suche im invertierten Index. Dies ergibt durchaus Sinn, da diese Begriffe in der Regel in der Datenbank gespeichert sind und Verben beispielsweise oft Beziehungen zwischen Tabellen darstellen und somit nicht direkt in der Datenbank abgebildet sind.

- Eine Testperson dachte, dass für die Operation „Join mit Zwischenresultaten“ die beiden Zwischenresultate Tabellen beinhalten müssen, welche direkte Nachbarn voneinander sind. Sie wusste nicht genau, wie viele Datenbankwissen für das Tool benötigt wird.
  → Auch dieses Problem kann mit einer Erklärung gelöst werden.

- Die Testpersonen hatten keine Probleme damit, die Vorschläge für Join-Pfade so zuzuordnen, dass das richtige Ergebnis herausgekommen ist.

5.2.2 Fazit

Alle oben genannten Probleme können entweder mit einer Erklärung vor der Verwendung des Tools oder mit der Bereitstellung des Datenbankschemas gelöst oder zumindest minimiert werden. Für einen zukünftigen User Test mit deutlich mehr Testpersonen, in dem bereits Daten für Machine Learning gesammelt würden, gäbe es vor Beginn des Tests eine Einführung in das Tool, in welcher die verschiedenen Operationen anhand einer Beispielfrage vorgestellt werden. Somit wären die meisten Unklarheiten von Anfang an beseitigt und die Testpersonen könnten sich sofort auf die eigentliche Arbeit, das Finden der gesuchten Antworten und das damit verbundene Sammeln der Machine-Learning-Daten, konzentrieren.

5.3 Was kann das Tool (noch) nicht?

5.3.1 Unlösbare Fragentypen

Folgende Typen von Fragen können noch nicht beantwortet werden:

- Mengenoperationen, wie beispielsweise INTERSECT sind derzeit nicht möglich. Eine Beispielfrage, welche nicht beantwortet werden kann, ist:
  *In which movies did **George Clooney and Brad Pitt** play?*

- Zwangsbedingungen können derzeit nur mit einem AND verknüpft werden, sprich es müssen alle Bedingungen zutreffen. Das logische OR ist derzeit nicht möglich.
  Eine Beispielfrage, welche nicht beantwortet werden kann, ist:
  *Which actors were **born in New York or died in New Jersey**?*
  Diese Frage wird in der Praxis wohl eher selten gestellt, jedoch enthält sie zwei Zwangsbedingungen, welche aber nicht zwingend beide erfüllt werden müssen, um im Resultat aufzutauchen.

- Der „GROUP BY“-Operator von SQL ist derzeit auch noch nicht implementiert. Somit kann diese Beispielfrage derzeit nicht beantwortet werden:
  Which actor played in the most movies?
  In diesem Beispiel müssten die Tabelle nach der Personen-ID gruppiert werden, ehe man anschliessend das Maximum daraus gewinnen kann.

- Ja/Nein-Fragen sind auch nicht direkt beantwortbar. Man kann jedoch daraus schliessen, dass ein „nein“ einem leeren und „ja“ einem nichtleeren Resultat entspricht.

5.3.2 Datumsformate

Da Datumsformate in der aktuellen Version unserer Applikation als Strings behandelt werden, werden auch noch keine speziellen Operationen angeboten, mit welchen man ein Datum verarbeiten kann. Es ist jedoch bereits möglich, ein Datumsvergleich zu machen. Dazu muss der Vergleichswert allerdings im exakten Format angegeben werden, ansonsten kann die Operation zu unerwünschten Ergebnissen führen, da zur Zeit ein Stringvergleich auf die beiden zu vergleichenden Daten ausgeführt wird.

5.3.3 Synonymproblematik

Falls kein Begriff einer Frage in der Datenbank vorkommt, hat der Benutzer keinen Einstiegspunkt und die Frage kann in dieser Form nicht beantwortet werden. Das kann vorkommen, wenn der Benutzer Synonyme für Wörter verwendet, deren Bedeutung zwar in der Datenbank abgebildet ist, sie jedoch nicht wortwörtlich vorkommen. Dieser Problematik kann teilweise entgegengewirkt werden, indem sich der Benutzer einen Überblick über die Datenbank verschaffen kann. Mit dem DB-Graph wurden erste Schritte in diese Richtung unternommen. Denn wenn der Benutzer die Bezeichnungen für die in Frage kommenden Tabellen oder Attribute kennt, kann er seine Frage so formulieren, dass sicher ein Einstiegspunkt gefunden wird.

6 Diskussion und Ausblick

6.1 Weitere atomare Operationen

Mit den bei dieser Version unterstützten Operationen können bereits sehr viele Fragen beantwortet werden. Falls in Zukunft zusätzliche Operationen hinzugefügt werden sollen, kann die Applikation aufgrund der flexiblen Architektur beliebig ergänzt werden.

6.2 Logging und Machine Learning

Schlussendlich soll die Applikation auch von Personen benutzt werden können, welche über kein Datenbankenwissen verfügen. Um dies zu ermöglichen, muss das Tool lernen, mit welchen Operationen welche Art von Fragen beantwortet werden können. Dazu muss das Tool alle Schritte, die ein Benutzer macht, in einer geeigneten Form aufzeichnen, damit sie später von einem Machine-Learning-Algorithmus verarbeitet werden können.

6.3 Datenbankenkompatibilität

Die Applikation unterstützt in der aktuellen Version nur MySQL-Datenbanken. Um in Zukunft weniger eingeschränkt zu sein, wäre es sinnvoll, wenn die Datenbankformate von den grössten RDBMS-Anbietern unterstützt werden würden.

Damit das Tool für viele Personen interessant wird, könnte eine Funktion implementiert werden, mit der ein Benutzer eine eigene Datenbank in das Tool importieren kann. Eine weitere Funktion in diesem Zusammenhang könnte sein, dass ein Benutzer eine schon vorhandene Datenbank aus einer Liste auswählen kann, falls er mehr über ein bestimmtes Thema erfahren möchte. So können - sobald das Tool intelligent ist - viele Datenbanken für Benutzer zugänglich gemacht werden, die selber über keine SQL-Kenntnisse verfügen.

6.4 Usability

Damit das Tool ohne jegliche Erklärung problemlos genutzt werden kann, müsste sehr viel Aufwand in die Benutzeroberfläche investiert werden. Es müssten viele Tooltips implementiert werden, so dass die Applikation einem Benutzer bei allfälligen Unklarheiten weiterhelfen kann.
Unserer Meinung nach ist es jedoch wichtiger und sinnvoller, Zeit in die eigentliche Logik der Applikation zu investieren, denn diese Version der Applikation wird zum Sammeln von Daten benutzt, welche anschliessend für das Machine Learning gebraucht werden. Dementsprechend ist die aktuelle Version eher ein Expertentool, das intelligente Endprodukt jedoch ein Tool für jedermann.

6.5 DB-Graph erweitern

Eine weitere Möglichkeit ist, den bestehenden DB-Graphen zu erweitern. Die automatische Ausrichtung müsste definitiv als Erstes verbessert werden. Anschliessend wäre es benutzerfreundlicher, wenn der Graph interaktive Funktionen bieten würde, beispielsweise einen Klick auf einen Knoten, so dass die in der Tabelle enthaltenen Attribute angezeigt werden. Es wurden Mockups erstellt, wie die beiden Funktionen „Joinen mit umliegender Tabelle“ und „Joinen mit gespeicherten Resultaten“ verbessert werden könnten.

In Abbildung 6.1 wird dargestellt, wie ein Join mit einer umliegenden Tabelle aussehen könnte. Man wählt eine Tabelle des Graphen aus, dieser wird farblich markiert (gelb im Mockup) und der prägnanteste Begriff der ausgewählten Tabelle soll eingeblendet werden, zusammen mit ein paar Beispieleinträgen. Dies soll dem Benutzer einen Überblick über die neue Tabelle verschaffen, falls er sich nicht sicher ist, welche Daten in der gewählten Tabelle vorhanden sind.

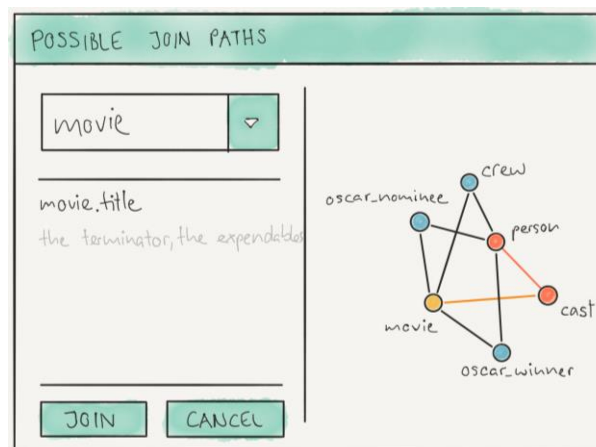

Abbildung 6.1: Mockup zu „Join mit umliegender Tabelle“

In Abbildung 6.2 wird der Join mit gespeicherten Resultaten dargestellt. Hier kann, falls mehrere Joinpfade möglich sind, ein Joinpfad ausgewählt werden und man erhält wieder den farblich markierten Pfad und die prägnantesten Begriffe der im Pfad vorkommenden Tabellen.

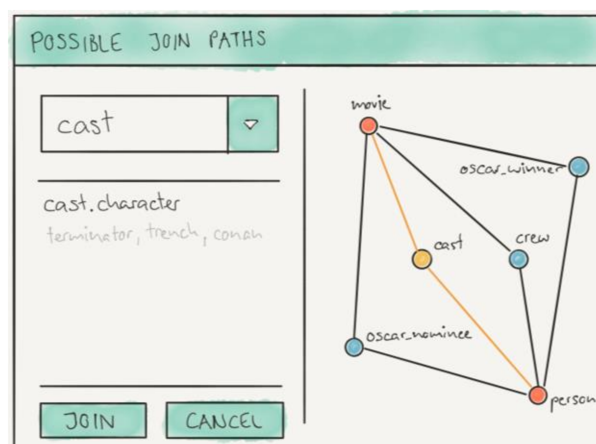

Abbildung 6.2: Mockup zu „Join mit gespeicherten Resultaten“

7 Verzeichnisse

7.1 Quellenverzeichnis

[1] „MySQL Workbench,“ [Online]. Available: https://www.mysql.com/de/products/workbench/. [Zugriff am 15 Dezember 2018].

[2] „MySQL - Reverse Engineering Using a Create Script,“ [Online]. Available: https://dev.mysql.com/doc/workbench/en/wb-reverse-engineer-create-script.html. [Zugriff am 15 Dezember 2018].

[3] „Jailer,“ [Online]. Available: https://github.com/Wisser/Jailer. [Zugriff am 15 Dezember 2018].

[4] „Networkx,“ [Online]. Available: https://networkx.github.io/. [Zugriff am 15 Dezember 2018].

[5] „D3.js,“ [Online]. Available: https://d3js.org/. [Zugriff am 15 Dezember 2018].

[6] „MySQL - Numerische Datentypen,“ [Online]. Available: https://dev.mysql.com/doc/refman/8.0/en/numeric-type-overview.html. [Zugriff am 15 Dezember 2018].

[7] „MySQL - Alphanumerische Datentypen,“ [Online]. Available: https://dev.mysql.com/doc/refman/8.0/en/string-type-overview.html. [Zugriff am 15 Dezember 2018].

[8] „MySQL - Datums- und Zeittypen,“ [Online]. Available: https://dev.mysql.com/doc/refman/8.0/en/date-and-time-type-overview.html. [Zugriff am 15 Dezember 2018].

[9] „Flask,“ [Online]. Available: http://flask.pocoo.org/. [Zugriff am 15 Dezember 2018].

[10] „Angular,“ [Online]. Available: https://angular.io/. [Zugriff am 15 Dezember 2018].

[11] „Angular Material,“ [Online]. Available: https://material.angular.io/. [Zugriff am 15 Dezember 2018].

7.2 Abbildungsverzeichnis

Abbildung 2.1: Schema-Browser mit erweiterten Attributinformationen der Tabelle „movie“ 11
Abbildung 2.2: Daten-Browser mit einem ausgeführten Join auf eine Nachbartabelle 12
Abbildung 3.1: Beispiel eines Logging-Baums 13
Abbildung 4.1: Grundlegende Architektur der Applikation 15
Abbildung 4.2: Beispiel einer Anfrage an den invertieren Index 17
Abbildung 4.3: Der Graph einer Datenbank nach manuellem Verschieben der Knoten 18
Abbildung 4.4: Definition der Join-Operation in Pseudocode 20
Abbildung 4.5: Ein Zwischenresultat in einem Graph mit den direkten Nachbarn 20
Abbildung 4.6: Zwei Zwischenresultate (blau) mit den kürzesten Pfaden dazwischen (rot) 21
Abbildung 4.7: Das Menü für die zwei verschiedenen Join-Möglichkeiten 21
Abbildung 4.8: Definition der Selektionsoperation in Pseudocode 22
Abbildung 4.9: Beispiel einer Selektion mit bereits abgefüllten Parametern 22
Abbildung 4.10: Beispiel einer Projektion auf 3 Attribute 23
Abbildung 4.11: Beispiel eines Suchverlaufs 25
Abbildung 4.12: Der Suchverlauf aus Abbildung 4.11 dargestellt in einem binären Logging-Baum 25
Abbildung 5.1: Screenshot der aktuellen Version unserer Applikation 27
Abbildung 6.1: Mockup zu „Join mit umliegender Tabelle“ 32
Abbildung 6.2: Mockup zu „Join mit gespeicherten Resultaten“ 32

8 Anhang

8.1 Offizielle Aufgabenstellung

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 können 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.

Ausblick:

Diese Arbeit soll die Grundlage für eine nachfolgende Bachelor-Arbeit bilden, in der mittels Machine Learning der Computer selbstständig lernt, aus einer natürlichspachlichen Frage ("Wie viele Filme hat die Ehefrau von Brad Pit gedreht") die passenden atomaren Operationen zu erkennen und zu kombinieren.

Aufgabenstellung:

Das Ziel dieser Arbeit ist die Grundfunktionen des Frameworks zu entwickeln. Konkrete Aufgaben sind:

- Definieren der atomaren Operationen
- Daten für einen Prototypen finden
- Konzeption eines GUIs mit Fokus auf Usability
- Logging-Funktionalitäten um das Userverhalten zu verfolgen
- Usability-Experimente mit Nicht-Experten

8.2 Github-Dokumentation / Installationsanleitung

Getting started

Prerequisites

- **git** should be installed
- **python 3.7** should be installed
- **npm** should be installed
- **angular 6** 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
```

Setup

Clone the project to your filesystem. The project consists of three folders: **documents**, **pa_backend** and **pa-frontend**.

- **documents** consists of the necessary MySQL-scripts to create the database.
- **pa_backend** consists of the Python API for the RESTful Web Service.
- **pa-frontend** consists of the Angular Single-Page-Application.

git clone

```
git clone https://github.engineering.zhaw.ch/PABA-kaisenic-schlaphi/PA_Code.g
it
```

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 pa_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 pa-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`.

8.3 REST-API

Description of the RESTful Web Service JSON API.

Overview

URI	Methode	Beschreibung
`/lookup`	`POST`	Returns the result from the inverted index tables
`/tabledata/preview`	`POST`	Returns a preview of five results from a desired table
`/tabledata`	`POST`	Returns the result from a mysql-query
`/tabledata/join`	`POST`	Returns the result from an sql join
`/tabledata/possiblejoinpaths`	`POST`	Returns the possible join paths between two intermediate results
`/dbgraph`	`POST`	Returns the whole database scheme as a graph

POST - 200 `/lookup`

Request

`search` : the desired search terms (e.g. "cast george clooney")

```
{
  "search": "movie"
}
```

Response

```
{
  "attributes": [
    {
      "attribute": "movie_id",
      "table": "cast"
    }
  ],
  "keywords": [
    {
      "attribute": "title",
      "keyword": "hey arnold! the movie",
      "table": "movie"
    }
  ],
  "tables": [
    {
      "table": "movie"
    }
  ]
}
```

POST - 200 `/tabledata/preview`

Request

`table` : the desired table

```
{
  "table": "person"
}
```

Response

```
{
  "attribute_names": [
    "id",
    "name",
    "gender",
    "birth_day",
    "birth_place"
  ],
  "data": [
    [
      1.0,
      "George Lucas",
      "m",
      "1944-05-14",
      "Modesto, California"
    ]
  ]
}
```

POST - 200 /tabledata

Request

- **tables** : all database-tables, that are used for the join
- **constraints** : all constraints for the mysql-query (e.g. movie.budget < 1000000)


```
{
  "tables": ["person"],
  "constraints": ["person.name = 'arnold schwarzenegger'"]
}
```


Response


```
{
  "attribute_names": [
    "person.id",
    "person.name",
    "person.gender",
    "person.birth_day",
    "person.birth_place",
    "person.death_day",
    "person.death_place"
  ],
  "data": [
    [
      1100.0,
      "Arnold Schwarzenegger",
      "m",
      "1947-07-30",
      "Thal, Steiermark",
      "-",
      "-"
    ]
  ],
  "recommendations": [
    "cast",
    "oscar_nominee",
    "crew",
    "oscar_winner"
  ]
}
```

POST - 200 `/tabledata/join`

Request

- **`tables_in_query`** : all database-tables existing in the current result
- **`table_to_join`** : the table you want to join with
- **`constraints`** : all constraints for the mysql-query (e.g. movie.budget < 1000000)

```
{
  "tables_in_query": ["person", "cast"],
  "table_to_join": "movie",
  "constraints": ["person.name = 'arnold schwarzenegger'"]
}
```

Response

```
{
  "attribute_names": [
    "person.id",
    "person.name",
    "person.gender",
    "person.birth_day",
    "person.birth_place",
    "cast.id",
    "cast.movie_id",
    "cast.person_id",
    "cast.character",
    "movie.id",
    "movie.title",
    "movie.original_title",
    "movie.original_language",
    "movie.budget",
    "movie.revenue"
  ],
  "data": [
    [
      1100.0,
      "Arnold Schwarzenegger",
      "m",
      "1947-07-30",
      "Thal, Steiermark",
      2597.0,
      534.0,
      1100.0,
      "The Terminator",
      534.0,
      "Terminator Salvation",
      "Terminator Salvation",
      "en",
      200000000.0,
      371353001.0,
    ]
  ],
  "recommendations": [
    "production_country",
    "oscar_nominee",
    "crew",
    "oscar_winner",
    "production_company"
  ]
}
```

POST - 200 `/tabledata/possiblejoinpaths`

Request

- **`tables_result1`** : all database-tables existing in the first intermediate result
- **`tables_result2`** : all database-tables existing in the second intermediate result


```
{
  "tables_result1": ["oscar"],
  "tables_result2": ["movie"]
}
```


Response

The response contains also information to display the result as a graph.


```
{
  "paths": [
    [
      "oscar_nominee"
    ]
  ],
  "subgraph": {
    "directed": false,
    "graph": {},
    "links": [
      {
        "source": "oscar_nominee",
        "target": "oscar"
      },
      {
        "source": "oscar_nominee",
        "target": "movie"
      }
    ],
    "multigraph": false,
    "nodes": [
      {
        "attribute_names_of_key": {
          "movie": [
            "movie_id",
            "id"
          ],
          "oscar": [
            "oscar_id",
            "id"
          ]
        },
        "id": "oscar_nominee"
      },
      {
        "attribute_names_of_key": {
          "oscar_nominee": [
            "id",
            "movie_id"
          ]        },
        "id": "movie"
      },
      {
        "attribute_names_of_key": {
          "oscar_nominee": [
            "id",
            "oscar_id"
          ]
        },
        "id": "oscar"
      }
    ]
  }
}
```

POST - 200 `/dbgraph`

Request

There is no JSON required.

Response

```
{
  "directed": false,
  "graph": {},
  "links": [
    {
      "source": "cast",
      "target": "person"
    },
    {
      "source": "cast",
      "target": "movie"
    }
  ],
  "multigraph": false,
  "nodes": [
    {
      "attribute_names_of_key": {
        "movie": [
          "movie_id",
          "id"
        ],
        "person": [
          "person_id",
          "id"
        ]
      },
      "id": "cast"
    },
    {
      "attribute_names_of_key": {
        "cast": [
          "id",
          "person_id"
      },
      "id": "person"
    },
    {
      "attribute_names_of_key": {
        "cast": [
          "id",
          "movie_id"
        ],
      },
      "id": "movie"
    },
  ]
}
```