Großer Beleg
(INF-D-950)

Entwicklung eines Frontends zur Durchführung von Nutzerstudien im Umfeld von „Why“-Graphanfragen

bearbeitet von

Christoph Scheidig

geboren am 7. Juli 1992 in Dresden

Matrikelnummer: 3759930
Studiengang: Diplom Informatik (2010)

Technische Universität Dresden

Fakultät Informatik
Institut für Systemarchitektur
Lehrstuhl Datenbanken

Betreuer: Elena Vasilyeva, M. Sc.
Dr.-Ing. Maik Thiele
Hochschullehrer: Prof. Dr.-Ing. Wolfgang Lehner

Erklärung

Ich erkläre, dass ich die vorliegende Arbeit selbständig, unter Angabe aller Zitate und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe.

Dresden, den 11.01.16

Inhaltsverzeichnis

1 Abstract

Graphdatenbanken kommen in verschiedenen Anwendungen zum Einsatz, in denen das relativ starre Relationenschema weniger gut geeignet ist – bei komplexen, flexibel änderbaren Strukturen und Abhängigkeiten, bei denen die Knoten und Kanten oft untereinander stark variierende Attribute haben. Die dabei entstehende Flexibilität kann sich aber dahingehend nachteilig auswirken, dass der Nutzer mit der Kardinalität der Ergebnisse nicht zufrieden ist und Fragen stellen könnte wie „why so few?“, „why so many?“ und „why empty?“, was eine Änderung der Anfrage notwendig macht.
In meinem Großen Beleg habe ich ein browserbasiertes Frontend erstellt, mit dem ein Nutzer Anfragen an einer Graphdatenbank definieren und ausführen, Ergebnisse anzeigen und die Anfrage anschließend gemäß des von ihm gewählten Problems verändern lassen kann. Das Backend einschließlich der Datenbank und jener Komponenten, die die Anfragen umschreiben sollen, existiert bereits und wird von meiner Anwendung genutzt. Insbesondere habe ich die theoretischen Hintergründe und meine Implementierung beschrieben, sowie bin abschließend auf ähnliche Problematiken aus dem Bereich der relationalen Datenbanken eingegangen.

2 Einleitung und Motivation

Um stark vernetzte Informationen und strukturierte Zusammenhänge zwischen Entitäten darzustellen eignen sich Graphdatenbanken besser als zum Beispiel das klassische relationale Modell.
Die Vorteile graphenorientierter Datenbanken wie Flexibilität und Ausdrucksstärke von Anfragen zeigen ihre Kehrseite in unerwarteten Strukturen der Anfrageergebnisse, wie leeren Ergebnismengen, zu vielen oder zu wenigen Ergebnissen oder dass bestimmte erwartete Graphen in den Ergebnissen fehlen. Verursacht werden kann dies durch Über- oder Unterspezifikation von Anfragen, Fehlern in ihnen oder Fehlen der gesuchten Daten – insbesondere mit Hinblick darauf, dass der Nutzer nur eingeschränkte Kenntnis von den Daten und dem Schema hat und deshalb mit deren Komplexität schnell überfordert sein kann. Um derartige Probleme zu lösen und bessere Ergebnisse zu erhalten, ist es unabdingbar, die Originalanfrage systematisch umzuschreiben. Zu dessen Umsetzung hat der Lehrstuhl Datenbanken schon Anwendungen entwickelt, die in Kombination mit der Ausführung der Abfrage an einer Datenbank eine Umschreibung von Anfragen gemäß eines vorgegebenen Problems durchführen können. Diese sind jedoch nur über die Kommandozeile bedienbar, was in jedem Fall sehr unintuitiv ist und heißt, dass die Anfrage-Erstellung oder -Änderung und das Auslesen alternativer Anfragen manuell erfolgen muss, indem

der Nutzer in Dateien schreibt bzw. aus Dateien liest, auf die der Server während der Ausführung zugreift. Systematische Nutzerstudien durchzuführen ist damit nicht möglich. Man sieht (innerhalb der Anwendung) nicht, wie viele Schritte zum Erreichen des Ziels notwendig waren, wie sich die Kardinalitäten verändert haben oder wie viel Zeit die Abfolge aller Vorgänge in Anspruch nimmt. Zudem müsste der Nutzer eine Anfrage manuell erstellen, indem er ihren Code selber schreibt, was nicht bloß heißt, dass der Nutzer deren Syntax genau kennen muss, sondern auch aufwändig und fehleranfällig ist.

Angesichts dieser Herausforderungen wäre es wünschenswert, ein Frontend zur Durchführung dieser Aktivitäten nutzen zu können. Es soll dem Nutzer ermöglichen, alle Aktivitäten auf einer einheitlichen Benutzeroberfläche durchzuführen, ohne sich mit lästigen Details der Serverbefehle befassen zu müssen und zudem noch Zugriff auf Informationen zu bekommen, die sonst nicht verfügbar wären. Insbesondere soll es auch möglich sein, über diese Oberfläche eine Anfrage graphisch zu erstellen und nach Bedarf zu ändern.

3 Grundlagen

Ausgegangen wird von der Verwendung eines **Attributgraphen**, also eines Graphen, der zusätzlich zu den Knoten und Kanten einen Attributraum definiert hat, der sich aus jeweils einem Attributraum für Knoten und Kanten zusammensetzt sowie zwei Funktionen, die die Knoten bzw. Kanten auf den ihnen zugehörigen Attributraum abbilden.

Formal definiert ist er durch $G = (V, E, u, f, g, A_V, A_E)$ mit $u: E^2 \rightarrow V$, $f : V \rightarrow A_V$, $g : E \rightarrow A_E$, wobei der Attributraum A die Vereinigung aus den Attributräumen A_V für die Knoten und A_E für die Kanten darstellt [VTBL15].

Ein **zusammenhängender Graph** ist ein nichtleerer Graph, bei dem sich für alle Knotenpaare jeweils ein Pfad finden lässt, der sie verbindet.

Die nach der Verarbeitung einer Anfrage möglicherweise auftretenden Probleme sind

- „**why empty?**“ bei leerer Lösungsmenge
- „**why so few?**“, bei zu wenig gefundenen Ergebnissen
- „**why so many?**“, bei zu vielen gefundenen Ergebnissen
- **„why not?“**, wenn bestimmte erwartete Ergebnisse fehlen

Bei der Betrachtung dieser Probleme kann man grundsätzlich von 2 Fällen ausgehen: „Why empty?“ und „Why so many?“. „Why so few?“ entspricht einem „Why so many?“ mit inversem Ziel (Anzahl der Ergebnisse steigern), während „Why not?“ einem mit Constraints behafteten „Why empty?“ entspricht [VTBL15]. „Why not?“-Anfragen werden aber in dieser Arbeit nicht betrachtet

und somit beschränkt diese sich nur auf „Why empty?“, „Why so few?“ und „Why so many?“.

4 Lösungsansätze

Grundsätzlich gibt es zwei Varianten, diese Probleme zu lösen. Der **teilgraphbasierte** Ansatz untersucht den Daten- und den Anfragegraph und berechnet Differenzen dazwischen, um darin möglicherweise die Ursache für unerwünschte Ergebnisse zu finden. Der gefundene Teilgraph muss maximal sein und ggf. auch Bedingungen bezüglich der Kardinalität der Ergebnisse erfüllen. Zum anderen besteht die Möglichkeit, die Anfrage **umzuformen**, um eine Anfrage zu erhalten, bei deren Ausführung das gegebene Problem nicht mehr auftritt. Wenn mehrere Kandidaten zur Verfügung stehen, wird derjenige gewählt, der mit möglichst wenigen Änderungen aus der Originalanfrage entstanden ist [VTBL15].

4.1 Operationen bei Vergleich und Änderung der Graphen

Zur Änderung des Anfragegraphen und zum Vergleich zweier Graphen stehen für das teilgraph- und das umschreibungsbasierte Verfahren gleichermaßen eine Menge von Operationen zur Verfügung, die man zunächst einmal in einfache und komplexe Operationen unterteilen kann. Zu den **einfachen** Operationen gehören verschiedene, paarweise entgegengesetzte Operationen, die zur Lockerung bzw. der Verstärkung der Anfrage dienen – je nachdem, welches Problem vorliegt. Änderungen können sich auf die Topologie sowie auf die Semantik beziehen. Auf **topologischer** Ebene kann man Knoten, Kanten, Richtungen, Quellen oder Ziele von Kanten löschen oder einfügen. Änderungen der **Semantik** entstehen durch Einfügen oder Löschen von Operatoren, Prädikaten, Konstanten und Typen. Manche Operationen wie Konstantenlöschung sind nicht atomar, das heißt sie können nicht einzeln auftreten und ziehen die Durchführung weiterer einfacher Operationen mit sich – in dem Fall die Löschung von Operatoren und Prädikaten. **Komplexe** Operationen setzen sich aus mehreren einfachen Operationen zusammen, die innerhalb eines Schritts durchgeführt werden und werden in 3 Typen unterteilt: knotenorientierte Operationen wie Knotenausschluss, Knotenspaltung oder Intervallerweiterung von Knotenprädikaten; kantenorientierte Operationen sind z.B. Kanten- oder Pfadspaltung, Intervallerweiterung von Prädikaten oder Änderung zugehöriger Quell- oder Zielknoten. Bei den teilgraphbasierten Operatoren, die also sowohl Knoten als auch Kanten umfassen, sind Erweiterung (Hinzufügen neuer Knoten und Kanten), Verdichtung und Lockerung des Teilgraphen zu nennen, also Hinzufügen neuer oder Entfernen vorhandener Kanten bei Beibehaltung der Knotenmenge [VTBL15].

4.2 Vorgehen bei „Why empty?"

Wenn die Ergebnismenge leer ist, stellt sich die Frage „Why empty?“. Beantwortet werden kann sie durch Differenzgraphen, die existierende und fehlende Teile von Anfragegraphen entdecken. Dafür berechnet man maximale gemeinsame zusammenhängende Teilgraphen von Daten- und Anfragegraph und die Differenz zwischen ihnen und dem jeweiligen Anfragegraphen. Maximale gemeinsame zusammenhängende Teilgraphen kann man mit speziellen Algorithmen aus der linearen Algebra berechnen, deren genaue Vorgehensweise grundsätzlich davon abhängt, wie die Graphen abgespeichert sind. Der Lehrstuhl Datenbanken verwendet dafür seinen eigenen Algorithmus **GraphMCS**, der speziell für Attributgraphen aus dem Algorithmus von McGregor heraus entwickelt wurde, der zusammenhängende Teilgraphen mittels Backtracking und Tiefensuche findet. Die darauffolgende Berechnung der Differenzgraphen speichert zuerst die Knoten und Abbildungen zwischen den Kanten der beiden Graphen ab und kann davon ausgehend Differenzgraphen berechnen, da sie durch jenen Vorgang nicht erfasst wurden. Danach werden weitere Überprüfungen bezüglich des Graphen angestellt. Zum Beispiel müssen ggf. Kanten, die sich zusammen mit ihren beiden adjazenten Knoten im Datengraphen befinden, aus dem Differenzgraphen entfernt werden. Am Abschluss wird die **Graphbearbeitungsdistanz** zwischen Daten- und Anfragegraph berechnet; sie ist die Anzahl der Schritte, um aus dem Datengraphen den maximalen gemeinsamen Teilgraphen von Daten- und Anfragegraph zu erhalten [VTBL15].

4.3 Vorgehen bei „Why so many?"

Bei dem anderen Extrem, den „Why so many?“-Anfragen als Folge zu vieler Ergebnisse, soll im Datengraphen nach Knoten und Kanten gesucht werden, die bei der Anfrageverarbeitung weggelassen werden können. Dabei werden Teilgraphen gemäß der erwarteten Größe der Ergebnismenge gesucht und Differenzgraphen berechnet. Zuerst ermittelt man dazu einen **kardinalitätsbeschränkten maximalen gemeinsamen zusammenhängenden Teilgraphen** von Daten- und Anfragegraph, d.h. einen gemeinsamen zusammenhängenden Teilgraphen von Daten- und Anfragegraph, der von maximal C_{maxExp} zusammenhängenden Teilgraphen von Daten- und Anfragegraph Teilgraph ist, wobei C_{maxExp} die maximale erwünschte Kardinalität ist. Um diese Teilgraphen zu finden, wurde der Algorithmus **BoundedMCS** entwickelt. Er sucht alle Kanten, fügt sie auf Grundlage ihrer Enden unter Nutzung von Tiefensuche zusammen und entfernt Teile, die zum Überschreiten der gegebenen Kardinalitätsschranke führen. Diese Suche wird von allen Kanten

als Startpunkt aus durchgeführt. Anschließend werden Differenzgraphen zwischen dem kardinalitätsbeschränkten maximalen gemeinsamen zusammenhängenden Teilgraphen und dem Anfragegraphen berechnet. Diese sind verantwortlich dafür, dass kein Ergebnis mit der passenden Kardinalität geliefert wurde. Grundlage bei der Berechnung sind ebenfalls die Basisoperationen aus Kapitel 4.1. Optimiert wurden BoundedMCS und GraphMCS durch die Suche nach optimalen Möglichkeiten bei der Wahl des Startknotens, die sich nach ihrem Ein- und Ausgangsgrad oder der Kardinalität von Knoten und Kanten richtet. Um auch in schwach zusammenhängenden Graphen innerhalb eines Durchlaufs alle Teilgraphen und diese auch in maximaler Größe zu finden, werden alles abdeckende Spannbäume eingeführt und temporär abgespeichert. Sie ermöglichen es bei der Suche, nach den ausgehenden auch eingehende Kanten zu traversieren. Experimentell wurde nachgewiesen, dass die Nutzung des Spannbaums ohne mehrere Startpunkte die Verarbeitungszeit reduziert und die Zahl der gefundenen gemeinsamen Teilgraphen ansteigen lässt [VTBL15].

4.4 Eine "Why empty?"-Verarbeitungskomponente

Beispielhaft möchte ich eine Anwendung zur Verarbeitung von "Why empty?"-Anfragen vorstellen. Sie kommuniziert mit der Datenbank und besteht aus dem Anfragenmanager, einer Kandidatenwahlkomponente, einer Anfragenlockerungskomponente und dem Kardinalitätsabschätzungsmodul [VTML15].

Die **Anfragenlockerungskomponente** empfängt eine Anfrage, die es zu lockern gilt, von der Datenbank, generiert mehrere gelockerte Anfragekandidaten und gibt sie anschließend an die Kandidatenwahlkomponente. Das könnten theoretisch ((Anzahl der Knoten + Anzahl der Kanten) * Anzahl der Lockerungsoperationen) Anfragen sein, wird aber dadurch optimiert, dass nur die Knoten und Kanten gewählt werden, bei denen die Erfolgswahrscheinlichkeit mittels Heuristiken als höher eingeschätzt wird. Diese Heuristiken sind zum einen *minimale Kardinalität von Knoten und Kanten* in den Daten (mindestens ein Knoten und eine Kante, bei gleicher Kardinalität auch mehrere) gemäß einer Schätzung des Kardinalitätsabschätzungsmoduls sowie *maximale Auswirkung und minimale Knotenkardinalität* (Wahl der Knoten nach erster Heuristik und Wahl der Kante danach, wie sich die Lockerung einer Kante auf ihre direkten Nachbarkanten auswirkt – je stärker die Auswirkungen, desto besser; bei Gleichheit kommen mehrere Knoten bzw. Kanten zur Lockerung infrage) [VTML15].

Die **Kandidatenwahlkomponente** erhält diese Kandidaten in Form einer geordneten Liste und dazu Vergleichsoperatoren. Anhand dieser bewertet sie die Qualität der vorgeschlagenen Anfragen und speichert sie in der daraus resultierenden Reihenfolge ab. Dazu werden sie zuerst hierarchisch

nach durchschnittlicher Pfadkardinalität (der Anzahl der Pfade im Datengraph, die mit dem Anfragepfad übereinstimmen) und wenn diese bei mehreren Kandidaten übereinstimmen, nach Bearbeitungsdistanz (also der Ähnlichkeit zwischen den Graphen, die durch die Anzahl der Bearbeitungsoperationen bei Transformation ausgedrückt wird) verglichen [VTML15].

Der **Anfragenmanager** wählt den besten der ihm vorgeschlagenen Kandidaten und schickt Anfragen an das Kardinalitätsabschätzungsmodul. Die als die beste beurteilte Anfrage wird an die Datenbank gesendet und ausgeführt. Wenn sie immer noch ein leeres Ergebnis liefert, leitet sie der Anfragenmanager an die Anfragenlockerungskomponente weiter, um sie iterativ weiter verändern zu lassen. Um ein möglichst optimales Ergebnis zu finden, wird bei der Lockerung die A*-Suche verwendet [VTML15].

Das **Kardinalitätsabschätzungsmodul** speichert, schätzt und berechnet Kardinalitäten von Anfragen, die es von dem Anfragenmanager erhält. Dazu werden für den Datengraphen u.a. die Anzahl der Knoten und Kanten und für die einzelnen Anfragen die Zahl der Knoten, Kanten und Prädikate gespeichert. Es werden anfrageabhängige Statistiken (spezielle Datenstrukturen) angelegt, in denen die Originalanfrage, Kardinalitäten für Knoten und Kanten und mit dazu abgelegte Pfadausdrücke abgespeichert werden [VTML15].

5 Die Implementierung

Implementiert habe ich die Anwendung auf einem Ubuntu Linux-Computer, der auf einer VMware Virtuellen Maschine läuft. Der Datenbankserver war bereits dort installiert; ich habe das Frontend und die zugehörige Middleware darauf entwickelt und getestet.

5.1 Schnellstart

1. auf dem Server mit Nutzer osboxes.org, Passwort osboxes.org anmelden
2. Kommandozeilen-Befehl node /home/osboxes/Documents/Beleg/server.js eingeben
3. im Browser **http://localhost:8080/** aufrufen

5.2 Der Datenbank-Server

Der Datenbankserver liegt im Homeverzeichnis des Nutzers osboxes, im Unterverzeichnis p-store/p-store/build, die aufzurufende Anwendung ist p-store-server. Sie kann direkt von der Kommandozeile gestartet werden oder – wie in unserem Fall – von einer Anwendung aus und läuft dann über die Kommandozeile des Servers, auf dem die Anwendung läuft. Gestartet wird sie, indem

der Nutzer in dem Verzeichnis den Befehl „./p-store-server“ aufruft.

Es gibt recht viele Unterordner, von denen der Nutzer viele nicht selbst nutzt, sondern auf die nur die Anwendung zugreift. Deren Inhalt werde ich hier nicht aufführen, da ich ihn zum Teil selbst nicht kenne. Der Unterordner „data" enthält für jedes Datenbankschema einen weiteren Unterordner, in dem die Knoten und die Kanten in jeweils einer csv-Datei (vertices.csv und edges.csv) abgelegt sind. [Schemaname].pstore enthält u.a. Befehle zum Erstellen der Tabellen und Spalten und dem Laden der Daten.

Im Verzeichnis „queries“ liegen Textdateien, die die Abfragen enthalten. Zu jeder Anfrage gibt es – nachdem sie ausgeführt wurde – außerdem jeweils 3 csv-Dateien, die während der Anfrageverarbeitung geschrieben werden und u.a. Informationen zu dessen Ablauf und den Ergebnissen enthalten.

OutputFile.query enthält Anfragen, die als Alternative zu der zuletzt ausgeführten vorgeschlagen worden. In *Resultfile.query* werden die Ergebnisse der zuletzt ausgeführten Anfrage geschrieben.

5.3 Die Datenbasis

Die vorliegende Anwendung nutzt das Datenbankschema ldbc_sf1. Die Kanten enthalten die Attribute type, joindate, creationdate, classyear und workfrom, die Knoten verfügen über name, url, creationdate, locationip, browserused, content, length, title, type, firstname, lastname, gender, birthday, imagefile, language und email. Diese sind natürlich nicht auf allen Datensätzen definiert, da diese verschiedene Typen haben und entsprechend über unterschiedliche Merkmale verfügen.

Insgesamt gibt es 100845 Knoten, davon – nach ihrem Typ (Attribut type) betrachtet – 100 Tags, 37980 Kommentare, 51643 Kommentare, 3084 Personen, 8022 Foren und 16 Tagklassen. Tags haben zum Beispiel außer dem Identifikator nur einen Namen und eine URL. Vorname, Nachname und Geschlecht sind nur für Personen definiert. Content und length existieren

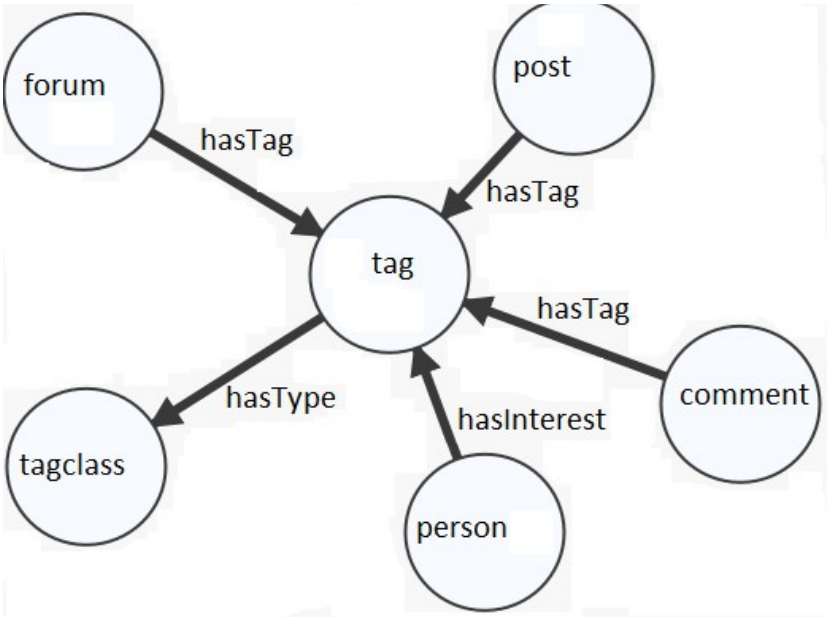

Abbildung 1: Schema der Datenbasis

nur für Posts und Kommentare. Die 102601 Kanten haben die Typen hasTag, hasType und hasInterest. Kanten können

- von Forum, Post oder Comment zu Tag (Typ hasTag)
- von Person zu Tag (Typ hasInterest)
- von Tag zu Tagklasse (Typ hasType)

verlaufen.

Ich habe dies schematisch in einem Graphen dargestellt (Abb. 1).

5.4 Die Syntax der Anfragen

am Beispiel erklärt:

type: RELAX_QUERY attrQueryMCS{	Der Anfang ist in jeder Anfrage gleich:
workspace: "ldbc_sf1" sourceVertexColumn : "source" targetVertexColumn : "target" edgeTypeColumn : "type" edgeTable : "edges" startVertexStrategy : 4 relaxStrategy : 9 fullSearch : 2 spanningTree : 1 distanceCalculator :1 isInteractive : 0 queryType : 1 cardinality : 100 idVertexColumn : "id"	Dazu gehören außer dem Typ und dem einleitenden Schlüsselwort attrQueryMCS auch die Attribute workspace, sourceVertexColumn, targetVertexColumn, edgeTypeColumn, edgeTable, startVertexStrategy, relaxStrategy, fullSearch, spanningTree und distanceCalculator. IsInteractive kann die Werte 0 oder 1 haben. Bei 0 wird die Anfrage automatisch umgeschrieben, bei 1 liefert sie nur Kandidaten. Der queryType gibt das vorliegende Problem an: 0 steht für „why empty“, 1 für „why so few“ und „why so many“. Cardinality gibt die gewünschte Kardinalität an und entfällt, wenn der queryType gleich 0 ist. Am Anfang ist auch kein queryType vorhanden, da der Nutzer noch kein Problem gewählt hat.

vertices{
id:0
outEdges:1
desc{
id : 0
attrName : "name"
	Knoten beginnen jeweils mit dem Schlüsselwort vertices. Sie haben einen fortlaufend gezählten, natürlichzahligen Identifikator (beginnend mit 0).
	Die Attribute outEdges und inEdges geben jeweils

attrPredicate : "="
attrValue : "Knoten"
attrColumn : "name"
table: "vertices"
}
desc{
id : 1
attrName : "type"
attrPredicate : "="
attrValue : "knoten"
attrColumn : "type"
table: "vertices"
}
}
 	
vertices{
id:1
inEdges:1
desc{
id : 0
attrName : "name"
attrPredicate : "="
attrValue : "Knoten2"
attrColumn : "name"
table: "vertices"
}
}
 	
attrStep{
sourceVertex:0
targetVertex:1
step{
pos:1
	den Identifikator einer aus- bzw. eingehenden Kante an. Sie geben, anders als es der Name assoziiert, jeweils genau eine Kante an und können, wenn es mehrere ein- bzw. ausgehende Kanten gibt, entsprechend mehrfach vorkommen.
	
	Desc ist das Schlüsselwort für die Attribute der Knoten. Sie werden ebenfalls durch die id durchnummeriert. AttrName gibt den Namen des Attributs an und ist immer mit attrColumn identisch. AttrValue gibt den Wert des Attributs an. Table verweist immer auf die Tabelle „vertices“.
	
	AttrStep ist das Schlüsselwort für die Kanten. SourceVertex und targetVertex geben den Identifikator für Start- und Zielknoten an.

edge_predicate:"type=hasTag"
lower: "1"
upper: "1"
direction : FORWARD
}
edgedesc{
id : 0
attrName : "classyear"
attrPredicate : "="
attrValue : "2015"
attrColumn : "classyear"
table: "edges"
}
}
}
	Step leitet den Bereich ein, der in jeder Kante genau einmal vorkommt. Er enthält zuerst pos, die natürliche Zahl, die jede Kante eindeutig identifiziert. Edge_predicate existiert nur, wenn der Kante ein Wert für das Attributs type zugewiesen wurde, den ich hier beispielhaft „hasTag“ genannt habe. Lower: “1“, upper: “1“ und direction: FORWARD werden jeder Kante zugewiesen.
	Darüber hinaus werden ggf. weitere Attribute von Kanten durch „edgedesc“ definiert, was die selbe Syntax wie „desc“ für die Knoten hat und entsprechend auf die Tabelle edges verweist.

5.5 Verwendete Technologien

Die Anwendung ist in HTML und JavaScript geschrieben und kann daher über einen gewöhnlichen Browser aufgerufen werden. Alle serverseitigen Aktivitäten erfolgen über Node.js, Diagramme und Graphen werden mit d3.js dargestellt und teilweise auch erzeugt.

Node.js ist eine JavaScript-basierte Plattform, die zum Betrieb von Netzwerkanwendungen und insbes. der Umsetzung lokaler Webserver dient. Mittels JavaScript wird die ereignisgesteuerte Architektur eines Webservers definiert. Node.js ist frei im Internet erhältlich und wird lokal installiert. Eine Ausführungsumgebung kompiliert den JavaScript-Code in ausführbaren Maschinencode.

D3.js („data-driven documents“) ist eine JavaScript-Bibliothek zur DOM-basierten, dynamischen, interaktiven Visualisierung von Daten im Webbrowser durch verschiedene Arten von Diagrammen, Graphen etc.. Dabei wird mittels JavaScript-Code meist eine SVG-Datei erzeugt (z.B. *var svg = d3.select(selector).append("svg")*), in das Dokument eingebunden und deren Elemente über die DOM-Knoten angesprochen und manipuliert. Zudem können Elemente auf der Grundlage extra geladener Daten erzeugt werden.

Socket.io ist eine Erweiterung, die asynchrone Echtzeitkommunikation zwischen Client und Server über einen Socket ermöglichen soll. Serverseitig wird es als Paket installiert, clientseitig wird eine

JavaScript-Datei eingebunden. Clientseitig muss man sich mit *var socket = io.connect()* mit dem Socket verbinden. In der JavaScript-Datei, die den Server definiert, wird am Anfang durch *var io = require('socket.io').listen(server)* der Socket eingerichtet, wobei der Parameter server sich auf die Definition des Servers bezieht (siehe Kapitel 5.7). Weiterhin gibt es einen Abschnitt, in dem die Aktionen des Sockets definiert werden: *io.sockets.on('connection', function(socket){...}*. Der funktionale Code des Servers wird im Rumpf dieser Funktion definiert, auf dem Client kann er an jeder Stelle einer JavaScript-Funktion definiert werden. Wird über den Websocket eine Nachricht verschickt, dann geschieht das – egal, ob vom Client oder vom Server aus – über *socket.emit('name', {Daten})*, wobei an der Stelle von „name“ ein Name steht, der diese Nachricht bezeichnet. Die Daten werden in JSON-Schreibweise notiert. Der Event-Handler, der auf den Empfang einer Nachricht reagieren soll, ist *socket.on('name', function(data){}*, wobei der Name der ist, unter dem die Nachricht verschickt wurde und im Rumpf mit *data.Feldname* auf den Wert der jeweiligen Felder zugegriffen wird.

5.6 Der Client

Meine Anwendung hat nach dem Start die Auswahlmöglichkeit, eine gänzlich neue Anfrage zu erzeugen oder eine Auswahl vordefinierter Templates zur Anfrageerstellung zu nutzen. In beiden Fällen wird über ein Inline-Frame der Grapheditor in Form der Dateien graph-creator.html, template1.html bis template5.html eingebunden. Diese sind inhaltlich identisch, bis auf den Wert der Variable st, die die Nummer des verwendeten Templates angibt. Diese wird in creaph-creator.js benutzt, das die eigentliche Funktionalität des Grapheditors enthält. In Abhängigkeit von st wird definiert, welche Knoten und Kanten der Editor am Anfang enthält. Wenn der Nutzer sich für die Templatenutzung entscheidet, bekommt er zuerst die erste Vorlage angezeigt und kann sich mit einem einfachen Klick für eine andere entscheiden, woraufhin einfach der Inhalt des inneren Frames ausgetauscht wird. Welche das sind, wird in Kapitel 5.6.2 vorgestellt.

Nach dem Abschicken der Anfrage werden deren Ergebnisse graphisch angezeigt. Sollte der Nutzer damit zufrieden sein, kann er den Vorgang beenden und danach eine neue Anfrage erstellen. Alternativ kann zwischen den verschiedenen möglichen Problemen gewählt werden – why empty, why so few und why so many, wobei bei Why so few und Why so many die erwartete Mindest- bzw. Höchstkardinalität angegeben werden muss. Danach wird die Anfrage erneut abgeschickt, woraufhin der Nutzer zwischen vorgeschlagenen Alternativanfragen auswählen kann, um im nächsten Schritt deren Ergebnisse angezeigt zu bekommen. Hier besteht die Möglichkeit zur Rückkehr zu den vorgeschlagenen Alternativen (um eine andere auszuwählen), zum manuellen

Ändern der Anfrage und zum Beenden.

Während der ganzen Vorgänge läuft oben auf der Seite eine Uhr, die die Zeit seit dem Aufruf der Startseite (in Sekunden) zählt. Ihr liegt eine JavaScript-Funktion zugrunde, die die Sekunden hochzählt und jeweils die Anzeige (in Stunden, Minuten und Sekunden) aktualisiert. Um die Zählung seitenübergreifend weiterlaufen zu lassen (und somit bei Aufruf der nächsten Seite nicht wieder von neuem beginnen zu lassen, was dem Zweck des Ganzen widersprechen würde), wird der aktuelle Sekundenwert immer als lokaler Cookie sessionStorage.seconds abgelegt und beim Laden einer Seite neu aufgerufen. Dafür ist auf jeder Seite die Funktion *window.onload = function(){new count(sessionStorage.seconds);}*. Auf der Startseite beginnt die Zählung von vorn, der Aufruf ist deshalb *count(0)*. Die mehrfach, also von jeder Seite verwendete Funktion count und die anderen, von ihr aufgerufenen Funktionen sind in *timer.js* ausgelagert. Ein Großteil dessen habe ich http://de.selfhtml.org entnommen und für diesen Zweck angepasst (u.a. Zählung der Sekunden auf- statt abwärts, Entfernen der Überprüfung, ob die Zählung bei 0 angekommen ist und die Funktion, die daraufhin aufgerufen wird). Am Ende eines Verarbeitungsvorgangs wird die gesamte Bearbeitungszeit ausgegeben und die Zählung der Sekunden auf 0 gesetzt.

5.6.1 Der Grapheditor

Der Grapheditor funktioniert mittels d3.js, wobei ich mich bei einer Vorlage aus dem Internet bedient habe [EDIT13]. Mit Strg und Mausklick kann man einen Knoten erzeugen und wenn man einen Knoten anfasst, Strg drückt und dabei den Cursor zu einem anderen Knoten zieht, wird dazwischen eine gerichtete Kante erzeugt. Durch einfaches Anklicken wird ein Knoten oder eine Kante ausgewählt. Durch Drücken der „Entfernen“-Taste wird er bzw. sie entfernt. Die Knoten und Kanten werden in jeweils einem Array aus JSON-Objekten abgespeichert, z.B.
nodes = [{name: "Peter", id: 0, x: xLoc, y: yLoc},{type: "Person", id: 1, x: xLoc, y: yLoc + 200}];
edges = [{source: nodes[1], target: nodes[0], type: "friend", id:1}];

Was passiert, wenn ein Knoten oder eine Kante ausgewählt ist, definiert die Funktion GraphCreator.prototype.replaceSelectNode bzw. GraphCreator.prototype.replaceSelectEdge: im unteren Bereich des Editors wird ein Drop-down-Menü und ein Eingabefeld eingeblendet und durch die nachfolgend definierten Funktionen removeSelectFromEdge bzw. removeSelectFromNode wird es wieder ausgeblendet, wenn der Knoten bzw. die Kante nicht mehr markiert ist. Es befindet sich jeweils in einem Bereich namens v_properties (für Knoten) bzw. e_properties (für Kanten), dessen Sichtbarkeit mit Javascript verändert wird. Das Menü enthält alle Attribute, die man für Knoten (name, url, creationdate, locationip, browserused, content, length,

title, type, firstname, lastname, gender, birthday, imagefile, language und email) bzw. Kanten (type, joindate, creationdate, classyear und workfrom) definieren kann. Der entsprechende Wert steht in dem Eingabefeld dahinter und kann darin auch geändert werden. Bei Auswahl eines Knotens bzw. einer Kante wird für jedes mögliche Attribut überprüft, ob der gewählte Knoten/ die Kante über dieses Attribut verfügt. Falls ja, wird dem Wert des entsprechenden Eingabefeldes der Attributwert zugewiesen. Vorher werden alle Eingabefelder gelöscht, damit in den Feldern, denen kein neuer Wert zugewiesen wird (wenn er für den aktuell ausgewählten Knoten bzw. die ausgewählte Kante nicht definiert ist), kein veralteter (also zu dem vorher ausgewählten Knoten/ Kante gehöriger) Wert stehen bleibt. Um den Code etwas zu verkürzen, nutze ich eine lokale Variable x, da ich so für jedes Attribut nur einmal „thisGraph.state.selectedEdge...“ schreiben muss. Soweit wie möglich wird der Datentyp des Eingabefeldes an den Wertebereich des Attributs angepasst – Geschlecht als Auswahlliste, Datum, Zahl oder E-Mail als HTML5-Datentyp, der nur einen entsprechenden Wertebereich zulässt. (Dessen Umsetzung leider stark vom Browser abhängig ist).

Beim Navigieren in der Liste mit den Attributen wird die Funktion C*hange* mit dem Parameter type, der die Werte edge und node haben kann, aufgerufen. Sie blendet alle Elemente der Klasse input aus und zeigt nur das Eingabefeld an, was den selben Namen wie das ausgewählte Element aus der Auswahlliste hat und setzt den Cursor darauf. Damit wird sichergestellt, dass immer genau ein Attribut-Eingabefeld angezeigt wird und dies genau das ist, was zu dem Element aus der Auswahlliste gehört. Die Attribute werden mit bei den entsprechenden Knoten bzw. Kanten gespeichert und darüber angesprochen, z.B. thisGraph.state.selectedEdge.(Name der Eigenschaft). In einem Text-Eingabefeld außerhalb des Frames wird die Anfrage angezeigt und bei jeder Änderung im Graphen aktualisiert. Dies wird über einen Event-Handler umgesetzt, der allen Eingabefeldern zugewiesen wird und bei jeder Änderung deren Inhalts eine Update-Funktion aufruft, die den entsprechenden Attributwert ändert, d.h. den Attributwerten des ausgewählten Knotens/ der ausgewählten Kante den aktuellen Inhalt der Eingabefelder zuweist und zudem die Funktion *composequery*() aufruft. Diese nimmt als Parameter den aktuellen Graphen und baut davon ausgehend die Anfrage zusammen. Dazu wird durch die Kanten iteriert, bei ihren einzelnen Attributen überprüft, ob ihnen ein Wert zugewiesen wurde und davon abhängig die Variable query zusammengesetzt. Ähnlich ist es anschließend bei den Knoten, nur dass dort am Anfang zusätzlich durch die Kanten iteriert wird, um zwecks der Attribute inEdges und outEdges zu prüfen, welche Kanten an dem Knoten anliegen. Abschließend wird der Beginn und das Ende der Anfrage vervollständigt (fehlende Klammern etc. ergänzt) und diese Zeichenkette als Parameter an die Funktion Query() auf der Seite übergeben, in der der Editor als Frame eingebunden ist, also *top.Query(query)*, die diesen Wert in das Eingabefeld einträgt. Beim Absenden des Formulars

werden im Kopf der Anfrage weitere Attribute ergänzt, die unabhängig von ihrem eigentlichen Inhalt sind (z. B. workspace, sourceVertexColumn und targetVertexColumn).
Da das Eingabefeld frei editierbar ist, wäre es auch möglich, die Anfrage manuell zu schreiben. Wenn man sich für die Nutzung eines Anfrage-Templates entscheidet, entfällt die Möglichkeit der manuellen Erstellung, da dann der Sinn der Template-Nutzung nicht mehr gegeben wäre. Deshalb ist das Eingabeformular in dem Fall unsichtbar (*style="display:none"*), wird aber dennoch benötigt, weil es zu dem Formular gehört, das die Anfrage enthält und am Ende abgeschickt wird.
Oben links im Grapheditor befindet sich ein Link „Help", mit dem sich ein kleiner Hilfetext dynamisch ein- und ausblenden lassen kann.
In der linken unteren Ecke befindet sich ein Button „delete graph". Dieser ruft die Funktion graph.prototype.deleteGraph() auf. Diese löscht die vorhandenen Knoten und Kanten und aktualisiert die Anzeige des Graphen. Außerdem werden die Bereiche e_properties und v_properties ausgeblendet, indem ihre Eigenschaft „display" auf „none" geändert wird.
Die Funktion updateGraph() ist größtenteils so geblieben, wie sie gegeben war. Sie aktualisiert die graphische Darstellung bei jeder Änderung im Graphen, z.B. bei Änderung oder Löschen eines Knotens oder einer Kante oder beim simplen Ändern der Position eines Elements durch drag & drop. Ich habe ihr einen Aufruf von composequery() hinzugefügt, damit die Anfrage auch beim Einfügen oder Löschen von Knoten oder Kanten (ohne, dass Attribute geändert werden) aktuell gehalten wird.
Desweiteren werden die in einen String umgewandelten Variablen nodes und edges als lokaler Cookie sessionStorage.nodes und sessionStorage.edges abgelegt. Sie können dort ausgelesen werden, wenn man die Anfrage später ändern will. Wenn man den Grapheditor aufruft, prüft er dafür, ob derartige Cookies existieren. Wenn ja, wird dieser String in ein Array aus JSON-Objekten zurückkonvertiert und den ansonsten leeren Variablen nodes bzw. edges zugewiesen. Die dabei berechneten Variablen eidct und idct sind Zähler für die Knoten und Kanten. Sie geben an, welchen Identifikator der nächste neu eingefügte Knoten bzw. die nächste neue Kante haben soll (der um 1 inkrementierte Identifikator des letzten Knotens bzw. der letzten Kante im Array). Beim Aufruf der Startseite werden die Cookies gelöscht, da man sie nur aufruft, wenn man eine neue Anfrage erstellen will und der Editor dann natürlich nicht mehr den Graphen der vorherigen Anfrage anzeigen soll.

5.6.2 Die Templates

Beispielhaft wurden, ausgehend von der Datenbasis (siehe Kapitel 5.3), einige Anfragevorlagen

entwickelt, um dem Nutzer bei der Erstellung mancher Anfragen behilflich zu sein.

Sie haben alle jeweils 3 Knoten und 2 dazwischen verlaufende Kanten. Bei allen Knoten und Kanten wurde das Attribut "type" mit einem Wert versehen, weil es das einzige ist, das an allen Knoten und Kanten definiert ist und außerdem nur sehr wenig Werte annehmen kann.

Die erste Vorlage enthält einen Post, einen ihm zugeordneten Tag und eine Person, die sich für diesen Tag interessiert.

Im zweiten Template wird ein Forum mit dem ihm zugeordneten Tag und dieses Tag wiederum mit einer Tagklasse/ einem Typen von sich verbunden.

Das dritte Template ist ähnlich dem ersten, mit dem Unterschied, dass hier anstatt dem Post ein Kommentar vorliegt.

Viertens ist ein Forum, das ein Tag hat, für das sich eine Person interessiert.

Letztens stellt ein Forum mit einem Tag und einem Post dar, der dem selben Tag zugeordnet ist.

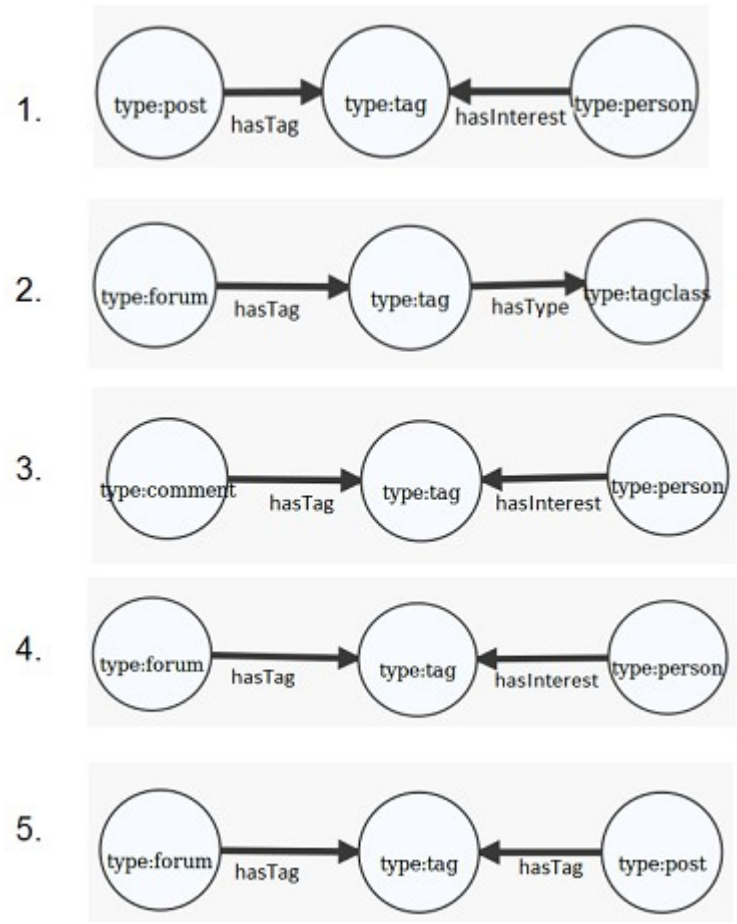

Abbildung 2: Erstellte Anfrage-Vorlagen

5.6.3 Die ersten Ergebnisse

Wenn man eine Anfrage abschickt, wird sie mit der Socket-Funktion *socket.emit('newquery', ...)* an den Server übergeben. Auf der daraufhin aufgerufenen Seite *results1.html* sollen die ersten Ergebnisse angezeigt werden. Zuerst erhält man nur den Hinweis, dass man warten soll, während im Hintergrund die Anfrage verarbeitet wird. Alle 10 Sekunden wird mithilfe des Sockets eine Anfrage „requestdata“ an den Server gestellt. Um die 10 Sekunden zu zählen, nutze ich die eingebundene Datei *request_data.js*, die eine Modifikation des Zählers für die Uhrzeit (*timer.js*) darstellt und auch die dort definierten Funktionen nutzt. Der Unterschied ist, dass die Zeit alle 10 Sekunden wieder auf Null zurückgesetzt wird, um somit wieder von vorne anzufangen, und dass

dann über den Socket eine Anfrage *socket.emit('requestdata', {type: 'results1'});* an den Server geschickt wird, wobei der Typ *results1* dem Server sagt, dass es sich um die Ergebnisse der ersten (also vom Nutzer erstellten) Anfrage handelt. Als Reaktion darauf sendet der Server die gefundenen Ergebnisse und alternative Anfragen wieder zurück. Diese ruft man, nachdem man sich mit dem Socket verbinden hat, über *socket.on('givedata', function(data){...});* ab. Da der Server die Dateien, in der die Ergebnisse und Alternativanfragen abgelegt werden (resultFile.query und outputFile.query), vor Beginn der Anfrageausführung löscht (damit man am Inhalt der Dateien erkennt, ob eine Anfrage bereits fertig verarbeitet wurde oder nicht – schließlich werden sie erst am Ende des Verarbeitungsvorgangs vom Datenbankserver geschrieben), wird man am Anfang zwei leere Strings erhalten. Wenn die Anfrage ausgeführt wurde, sind jene Zeichenketten nicht mehr leer und die Ergebnisse werden textuell auf dem Bildschirm ausgegeben, indem mittels jQuery der Hinweis „Bitte warten Sie...“ durch die Ergebnisse ersetzt wird. Außerdem wird auch ein *storecardinality*-Request an der Server geschickt, dem die aus der 1. Ergebniszeile extrahierte Ergebniskardinalität übergeben wird, damit am Ende ein Überblick über die Kardinalitäten gegeben werden kann (näheres im Kapitel 5.7). Um zu verhindern, dass nach weiteren 10 Sekunden eine neue „requestdata“-Nachricht an den Server geschickt wird, wird die Variable requestdata auf den Wert „0“ gesetzt. Am Anfang wurde sie auf den Wert 1 initialisiert. In der sekündlich aufgerufenen Funktion *calculate* wird geprüft, ob requestdata=1 gilt; nur dann werden die Sekunden weitergezählt bzw. request-Anfragen an den Server geschickt. Desweiteren wird – indem das Attribut display von none auf inline geändert wird – ein Formular eingeblendet, bei dem der Nutzer das vorliegende Problem auswählen kann – why so few, why so many und why empty. Bei der Wahl von why so few oder why so many wird ein zusätzliches Eingabefeld angezeigt, in dem die gewünschte Kardinalität angegeben werden muss. Beim Abschicken wird die Funktion *checkForm*() aufgerufen, die prüft, ob ein Problem ausgewählt wurde und für den Fall, dass es why so few oder why so many ist, auch, ob eine Kardinalität angegeben wurde. Wenn nicht, wird eine entsprechende Fehlermeldung in einem alert-Fenster ausgegeben und durch *return false* das Abschicken verhindert. Ansonsten wird ähnlich wie zuvor die Anfrage der ausgewählte Problemtyp und ggf. die Kardinalität mittels Socket-Aufruf *socket.emit('changetype', {type: ..., card: ...})* an den Server übergeben. Zudem besteht natürlich für den Fall, dass man mit den Ergebnissen zufrieden ist, die Möglichkeit, den Bearbeitungsvorgang zu beenden.

Wenn man später nochmal zu den ersten Ergebnissen zurückkehrt und die Seite somit mit einem GET-Request aufruft, wird *resultqueries_old.html* übergeben. Sie unterscheidet sich dadurch, dass der 10sekündlich stattfindende Aufruf von „requestdata“ entfällt und stattdessen beim Laden einmal „requestresults1“ über den Socket an den Server gesendet wird und das Formular zur Wahl des

Problems sofort sichtbar ist. Infolgedessen werden gleich beim Laden die Ergebnisse der letzten Anfrage angezeigt, die der Nutzer erstellt hat – und nicht die der zuletzt ausgeführten Anfrage.

5.6.4 Vorgeschlagene Anfragen

Bei Auswahl eines Problems wird als nächstes *resultqueries.html* aufgerufen. Das Prinzip ist das selbe wie im vorherigen Schritt. Zuerst wird wieder der Hinweis angezeigt, dass die Anfrage im Hintergrund läuft. Auf die selbe Weise werden alle 10 Sekunden die Ergebnisse abgefragt, wobei bei der *requestdata*-Funktion der Typ *resultqueries* übergeben wird. Nachdem welche gefunden wurden, werden sie angezeigt. Dazu müssen die einzelnen Anfragen erst einmal aus der übergebenen Zeichenkette heraus ermittelt werden. Dabei wird ausgenutzt, dass alle Anfragen mit „type:“ beginnen – eine Zeichenkette, die innerhalb einer Anfrage nicht auftreten dürfte. Deshalb wird der String an allen Vorkommen von „type:“ aufgetrennt (*results.split('type:')*). Das dabei entstehende Array *parts* enthält, ab der Indexposition 1, alle Anfragen. Die Indexposition 0 ist praktisch leer, da sie das enthält, was vor dem ersten Vorkommen der Zeichensequenz „type:“ in outputFile.query steht, und das ist nichts außer ggf. ein Leerzeichen und ein Zeilenumbruch (eben weil alle Anfragen mit „type:“ anfangen). Um die korrekte Syntax wiederherzustellen, wird allen Elementen das durch die Anwendung der split-Funktion verloren gegangene „type:“ wieder angefügt. Anschließend wird durch parts iteriert und eine zweispaltige Tabelle ohne sichtbare Ränder angezeigt, deren Zeilen jeweils aus einem Radio-Button (*<input type=“radio“>*), also einem Auswahlfeld bestehen, das mit den anderen Radio-Buttons (falls mehrere Anfragen zur Auswahl stehen) eine Gruppe bildet, aus der maximal ein Feld ausgewählt werden kann, und daneben der textuellen Anzeige der Anfrage. Der dem Radiobutton zugewiesene Wert ist die Nummer der Anfrage, die mit 1 beginnend fortlaufend gezählt wird, um beim Absenden identifizieren zu können, welche Anfrage ausgewählt wurde. Beim Absenden des Formulars wird die Funktion *complete()* aufgerufen, die überprüft, ob eine Anfrage ausgewählt wurde. Falls nicht, wird wie gewohnt eine Fehlermeldung ausgegeben und der Versand des Formulars abgebrochen. Ansonsten wird jetzt die gewählte Anfrage der Variable query zugewiesen, indem auf das vorhin erzeugte Array parts zugegriffen und das Element mit dem Index ausgewählt wird, der dem Wert des Radio-Buttons entspricht (z.B. der Radiobutton bei der ersten Anfrage hat den Wert 1, infolgedessen wird der Variable query der Wert des Arrayelements mit der Indexposition 1 zugewiesen). Danach wird Verbindung mit dem Socket hergestellt und unter dem Namen „choosealternative“ die gewählte Anfrage übergeben. Als Ziel des Formulars wird die nächste Seite *resultgraphs.html* aufgerufen. Zudem gibt es einen Button, der so wie jener zum Abschicken erst

eingeblendet wird, wenn die Ergebnisse geladen sind und die Aufschrift „change query“ trägt, bei dessen Anklicken der Nutzer auf *newquery.html* weitergeleitet wird, in der der Grapheditor als Inline-Frame eingebunden wird. Er enthält die zuletzt vom Nutzer erstellte Anfrage, der sie dann ändern kann.
Bei einem GET-Request verhält es sich mit *resultqueries_old.html* im Prinzip genauso wie bei den ersten Ergebnissen. Durch die Socket-Nachricht *requestqueries* sofort beim Laden werden die alternativen Anfragen angezeigt, die der Nutzer beim vorherigen Aufruf dieser Seite, also nachdem er das Problem gewählt hatte, gesehen hat.

5.6.5 Ergebnisse

Resultgraphs.html, das nach der Wahl der Alternativanfrage aufgerufen wird, funktioniert im wesentlichen genauso wie results1.html. Die Anfrage wird im Hintergrund ausgeführt und die Ergebnisse, wenn sie verfügbar sind, angezeigt. Anschließend gibt es die Möglichkeit, den Bearbeitungsvorgang zu beenden oder, falls die Ergebnisse nicht zufriedenstellend sind, zum Grapheditor oder zu den alternativen Anfragen zurückzukehren, um die Anfrage manuell zu ändern bzw. einen anderen Vorschlag auszuwählen.
Clientseitig gibt es hier beim GET-Request keinen Unterschied. Über die Socket-Funktion werden die Ergebnisse der zuletzt ausgeführten Anfrage abgerufen. Anders als in den vorherigen Schritten macht es keinen Unterschied, ob die Anfrage eben erst ausgeführt wurde oder schon vorher (und somit die Ergebnisse zum wiederholten Mal angezeigt werden), weil es an dieser Stelle außer durch Zurückspringen keine andere Möglichkeit gibt, weitere Anfragen auszuführen.

5.6.6 Das Ende des Verarbeitungsvorgangs

Wenn der Nutzer den Bearbeitungsvorgang beendet, wird *finish.html* aufgerufen, das die gesamte Bearbeitungszeit und ein Säulendiagramm angezeigt, das den zeitlichen Verlauf der Ergebniskardinalität darstellt. Die dafür benötigten Werte werden aus *data.tsv* ausgelesen und durch d3.js dargestellt. Dafür ist der Bereich „Diagramm“ vorgesehen, die Funktionalität liegt in *diagramm.js* ausgelagert. Der Code wurde [DIAGR12] entnommen. Nachdem am Anfang u.a. Maße und die Achsen definiert werden, definiert *svg* eine Variable, die eine leere SVG-Grafik mit festgelegter Höhe und Breite darstellt, die an den Bereich „Diagramm“ angehängt wird. D3.tsv liest den Inhalt von data.tsv aus und legt number als x- und amount als y-Wert fest. Danach wird das genaue Aussehen des Diagramms beschrieben.

5.6.7 Die Navigationsleiste

Auf der linken Seite befindet sich eine Navigationsleiste, an der der Nutzer sehen kann, welche Bearbeitungsschritte es gibt und in welchem er sich aktuell befindet. Anklicken lässt sich nur ein Teil der Links – nämlich hauptsächlich die, die zu Seiten führen, die man bei sequentieller Vorgehensweise, also ohne „Zurückspringen“ schon einmal besucht hat. So kann man bei der Anzeige der Anfrageergebnisse zwar zum Grapheditor oder zur Startseite zurückgehen, aber nicht zu der Seite, die alternative Anfragen vorschlägt, weil man dafür zuerst das vorliegende Problem ausgewählt haben muss. Um dies kenntlich zu machen, sind die Elemente entsprechend unterschiedlich gekennzeichnet. Die aktuell aufgerufene Seite hat keinen Link zu sich selbst, sondern nur einfachen Text, der nicht weiter gestaltet ist. Die weitere farbliche Gestaltung wird einheitlich über das externe Stylesheet *main.css* definiert. Links werden farblich hervorgehoben; die, die beim Anklicken die entsprechende Seite aufrufen, bekommen die Klasse „links“ zugewiesen und werden in der Stylesheet-Datei dahingehend gesondert behandelt, dass sie beim Berühren auch entsprechend hervorgehoben werden, während beim Berühren der anderen Links gar nichts passieren soll und dafür auch explizit die Stylesheet-Angabe „cursor:default“ zugewiesen wird, was heißt, dass der Mauszeiger beim Berühren des Elements weiterhin sein standardmäßiges Aussehen hat.

5.7 Die Middleware

Die Kommunikation zwischen Client und Datenbank-Server erfolgt durch serverseitige Befehle. Mit Node.js habe ich einen einfachen HTTP-Server aufgesetzt, der über http://localhost:Port aufrufbar ist, wobei ich den Port selbst definieren kann. Dazu muss man Node.js über die Kommandozeile aufrufen und als Parameter den Pfad zu der JavaScript-Datei angeben, die die Definition des Servers enthält. Zusätzlich habe ich das Framework express installiert, mit dem man eine Verzeichnisstruktur auf dem Server erzeugen kann. Das kann so aussehen:

```
var express = require(„express“); var app = express();
app.get('/', function(req, res){res.sendFile('index.html');});
app.get('/other.html', function(req, res){other action;});
```

Damit wurde die gesamte Verzeichnisstruktur des lokalen Servers definiert.

Req steht für „request“ und res für „response“. So wird beim Aufruf des Wurzelverzeichnisses die Datei index.html übergeben und bei Aufruf des Pfades /other.html irgendeine andere Aktion durchgeführt. Es wird zwischen GET- und POST-Requests unterschieden. GET-Requests treten

beim gewöhnlichen Aufruf einer Seite auf, sowie nach dem Absenden eines Formulars mit der HTTP-Methode GET (was in meiner Anwendung nicht genutzt wird); POST-Requests dagegen, wenn ein Formular mit der HTTP-Methode POST abgeschickt wurde. So ist es nötig, für die Dateien, die über beide HTTP-Methoden aufgerufen werden können (*results1.html*, *resultqueries.html* und *resultgraphs.html*), beide Möglichkeiten zu definieren.

Die Definition des Servers sieht so aus: *var server = require('http').createServer(app)*. *Server.listen(8080)* definiert den Port, über den der Server angesprochen wird. Somit ruft man diese Anwendung über **http://localhost:8080/** auf. Außerdem muss mittels *var fs = require(„fs“);* die Nutzung von Dateisystem-Operationen ermöglicht werden.

Die Nutzung von socket.io und express erfordert den separaten Download von Packages, die dann im Unterverzeichnis *node_modules* des jeweiligen Projektes abgelegt werden.

Das Paket socket.io soll asynchrone Echtzeitkommunikation zwischen Client und Server über einen Socket ermöglichen.

Zu Beginn wird ein Kindsprozess erzeugt, in dem der Datenbank-Server läuft, gestartet mit *var child = require(„child_process“); var serv = child.exec(„./p-store-server“)*. Der Parameter cwd steht für „current working directory“ und gibt das Verzeichnis an, in dem der Befehl ausgeführt werden soll; das ist in dem Fall der absolute Pfad zu dem Verzeichnis build, in dem der Server liegt. An diesen werden mittels *child.stdin.write(“command“)* die Befehle geschickt, die auf dem Server ausgeführt werden sollen und, um die korrekte Funktionalität zu gewährleisten, jeweils mit „\n“ abgeschlossen werden. Konkret ist es *child.stdin.write(“start ldbc_sf1\nload ldbc_sf1 vertices:1\nload ldbc_sf1 edges:1\nrun config/runhist\n“)*. Damit wird die Datenbank gestartet, die zugehörigen Daten geladen und eine Konfigurationsdatei ausgeführt. Damit das, was der Kindsprozess ausgibt, in der Konsole sichtbar ist, muss es explizit dort geloggt werden: *console.log(serv.stdout.pipe(process.stdout))*.

Beim Aufruf der Startseite werden die Daten aus data.tsv gelöscht, da diese aus der letzten Abfrage stammen und jetzt nicht mehr benötigt werden. Dazu wird einfach der Kopf neu in die tsv-Datei geschrieben: *fs.writeFileSync('Pfad/data.tsv', “number\tamount“)*, wobei „\t“ einen Tabulator einfügt. Wenn eine neu erstellte Anfrage abgeschickt wird, wird sie mit dem Schlüsselwort „newquery“ an den Socket übergeben und von ihm in einer sich im Verzeichnis *queries* befindlichen Datei *1.txt* abgelegt. Da der Client nach Absenden des Formulars *results1.html (*als POST-Request*)* aufruft, werden nun resultFile.query und outputFile.query, also jene Dateien, die die Ergebnisse der Anfrage und vorgeschlagene alternative Anfragen enthalten, gelöscht, indem mittels fs.writeFileSync('Pfad/Datei', '') jeweils eine leere Zeichenkette in die beiden Dateien geschrieben wird. Die Variable „fs“ steht für Dateisystem-Operationen und wurde am Anfang initialisiert.

Anschließend wird die Anfrage ausgeführt, indem *(„query queries/1.txt“)* in den stdin-Port des Kindsprozesses geschrieben und die Datei *results1.html* an den Client übergeben wird. Wird results1.html als GET-Request aufgerufen, dann wird die Anfrage nicht nochmal ausgeführt, sondern bloß die Datei *results1_old.html* mit den Ergebnissen gesendet.

Dort wählt der Nutzer das Problem aus, das er bei den Ergebnissen sieht und schickt beim Absenden eine Nachricht mit dem Namen „changetype“. Der Server reagiert darauf, indem er die bisherige Anfrage in die Variable q einliest und entsprechend des Problemtyps ändert. Bei „fewmany“, wenn also das Problem „why so few“ oder „why so many“ ist, werden in der Anfrage vor dem Attribut idVertexColumn die zusätzlichen Attribute queryType mit dem Wert 1 und cardinality mit dem unter dem Namen „card“ übergebenen Wert eingefügt. Ansonsten, wenn also das Problem „why empty“ ist, wird der queryType 0 eingefügt. Die Variable query, die die veränderte Anfrage enthält, wird jetzt in *2.txt* abgelegt. Die Originalanfrage in 1.txt wird unverändert gelassen, da der Nutzer später zu diesem Schritt zurückkehren und ein anderes Problem wählen könnte, was es erfordert, dass die eben beschriebenen Parameteränderungen auf die Originalanfrage angewendet werden. Der Client ruft beim Abschicken des Formulars *resultqueries.html* auf. Bevor der Server diese Datei übergibt, führt er fast die selben Kommandos aus wie für das Anzeigen der ersten Ergebnisse – Löschen der beiden Dateien und Ausführen der Anfrage aus *2.txt*.

Wenn der Nutzer eine vorgeschlagene Anfragealternative auswählt und das Formular abschickt, erhält der Server eine Nachricht mit dem Namen „choosealternative“. Mit ihr wird in der Variable query die vom Nutzer ausgewählte Anfrage übergeben und vom Server in die Datei *2.txt* geschrieben. Der Aufruf von *resultgraphs.html* führt wieder zum dritten Mal zum selben serverseitige Ablauf, mit Löschen von outputFile.query und resultFile.query, der Ausführung der Anfrage auf dem Datenbankserver und der Übergabe von resultgraphs.html an den Client.

Bei einem GET-Request wird jeweils nur eine Datei übergeben und serverseitig nichts weiter ausgeführt, da es in der Situation keine Anfrage gibt, die verarbeitet werden müsste.

Wenn der Server über den Socket eine „requestdata“-Nachricht erhält, dann liest er den Inhalt von outputFile.query in die Variable query und resultFile.query in die Variable result ein. Die Zuweisungen result=result+““ bzw. query=query+““ dienen dazu, dass query und result zu einem gültigen String werden, den man z.B. auch mittels Gleichheitsoperator mit einem anderen String vergleichen kann (ansonsten würden sämtliche Vergleiche „false“ liefern). Danach werden query und result mit einer „givedata“ benannten Socket-Nachricht an den Client versandt. Wenn dabei der Typ *results1* bzw. *resultqueries* mit übergeben wird, speichert der Client die Ergebnisse bzw. alternativen Anfragen in *results1.txt* bzw. *resultqueries.txt* ab, um sie später abrufen zu können,

wenn der Nutzer „zurückspringt“ und in outputFile.query und resultFile.query inzwischen andere Ergebnisse bzw. Abfragen liegen (z.B. die Ergebnisse der vorgeschlagenen Anfrage und ggf. die bei dessen Ausführung generierten Alternativen). Wenn der Client bei der Verarbeitung von „givedata“ feststellt, dass die Anfrage nun fertig ausgeführt wurde (erkennbar dadurch, dass query und result nicht mehr leer sind), sendet er eine Nachricht „storecardinality“ an den Server, dem er in der Variable *anz* die Ergebniskardinalität liefert. Der legt diese in *data.tsv* ab, das im Prinzip einer zweispaltigen Tabelle entspricht – number als Identifikator in Form einer fortlaufenden, mit 1 beginnenden natürlichen Zahl und amount als die Kardinalität der Anfrage. Der bisherige Inhalt wird in die Variable *old* eingelesen. Wenn old an jedem Zeilenumbruch aufgesplittet wird, erhält man das Array, das aus den einzelnen Zeilen von data.tsv besteht. Die Anzahl der Zeilen ist der Identifikator für die neu eingefügte Zeile (*parts.length*): Wenn data.tsv nur eine Zeile enthält (den Kopf), dann erhält der erste Eintrag die Nummer 1, bei zwei Zeilen (Kopf und ein Eintrag) die Nummer 2 usw. Zusätzlich zu dem bisherigen Inhalt werden nach einem Zeilenumbruch nun der Identifikator, ein Tabulator und die übergebene Kardinalität in data.tsv geschrieben: *fs.writeFileSync(Pfad/data.tsv, old+“\n“+parts.length+“\t“+data.anz)*. Dort können sie später abgerufen werden, um einen Überblick darüber zu erhalten, wie sich die Kardinalitäten im Laufe der Verarbeitung verändert haben.

Wenn der Server über den Socket eine „requestresults1“-Nachricht vom Client bekommt, übergibt er über „givedata“ die in results1.txt abgespeicherten Ergebnisse der zuletzt vom Nutzer erstellten Nachricht. Bei einer „requestqueries“-Anfrage geschieht das genauso mit den vorgeschlagenen Alternativen aus resultqueries.txt.

6 Durchführung eigener Nutzerstudien

Um die vorliegende Anwendung zu testen und ihren Nutzen zu ermitteln, wurden mehrere Nutzerstudien mit verschiedenen Anfragen durchgeführt. Orientierung boten dafür am Anfang die von mir erstellten Templates (siehe Kapitel 5.6.2).

Zu Beginn habe ich mir in Bezug auf die erste Vorlage die Posts, ihre Tags und die Personen anzeigen lassen, die sich für diesen Tag interessieren und die den Browser Firefox benutzten (siehe Abb. 3). Es gab 1190 Ergebnisse, und ich wollte sie auf 50 begrenzen. Die vorgeschlagene Alternative unterschied sich nur geringfügig von der nach der Wahl des Problems und Angabe der Kardinalität übergebenen Anfrage (die über die Attribute queryType und cardinality

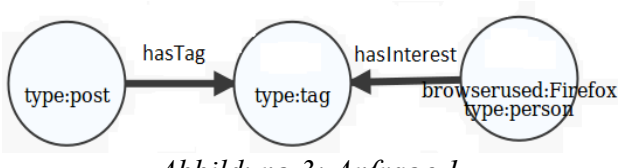

Abbildung 3: Anfrage 1

verfügt), indem mit Ausnahme von type: RELAX_QUERY und workspace alle Attribute am Anfang der Anfrage verschwanden und teilweise etwas verändert in die Mitte eingefügt oder an ihr Ende angehängt wurden. Zudem wurde die Reihenfolge der Knoten und Kanten vertauscht, sodass die Kanten vor den Knoten kommen und der queryType nicht mehr als Zahl, sondern als WHY_SO_MANY definiert wird (dies trifft übrigens auf alle vom System generierten Anfragen zu, ohne dass es jedes Mal explizit erwähnt wird). Mehr Alternativen standen nicht zur Verfügung. Die Anfrage lieferte keine Ergebnisse, was zwar wesentlich näher an der geforderten 50 als an der ursprünglichen 1190 dran ist, aber dennoch kein befriedigendes Ergebnis darstellt.

Die Einschränkung auf weibliche Personen durch manuelle Änderung der Anfrage lieferte schließlich 609 Ergebnisse. Die vorgeschlagene Alternative folgte demselben Prinzip, dass nur die Position der Attribute und Reihenfolgen, aber nichts an den Knoten und Kanten selbst geändert wurde und lieferte ebenfalls keine Ergebnisse.

Zur Einschätzung des Aufwands wurden die Laufzeiten der einzelnen Verarbeitungsschritte gemessen. Nach der recht unkomplizierten Erstellung der Anfrage unter Nutzung des Templates dauerte es etwa 40 Sekunden, bis sie im Hintergrund fertig verarbeitet wurde und die Ergebnisse sichtbar sind. Nach der Wahl des Problems und der Angabe der gewünschten Kardinalität nahm es 50 Sekunden in Anspruch, bis die Alternative ermittelt und angezeigt wurde und später weitere 50 Sekunden für die Verarbeitung dieser Anfrage. Insgesamt sind bis dahin etwa 5 Minuten seit Aufruf der Anwendung vergangen. Nach der Veränderung der Anfrage durch Hinzufügen des Prädikates „Geschlecht“ an dem Knoten der Person wurde letztendlich der ganze Zyklus noch einmal durchlaufen, nahm aber etwas weniger Zeit in Anspruch: 30 Sekunden für die ersten Ergebnisse und jeweils 40 Sekunden für die Alternativenfindung und die Ausführung dieser Abfrage. Nach der Beendigung des Bearbeitungsvorganges an dieser Stelle kam man auf eine Gesamtlaufzeit von etwa 9 Minuten und kann den beschriebenen Verlauf der Kardinalitäten in einem Säulendiagramm sehen. Die 9 Minuten stellen dabei die Gesamtzeit vom ersten Aufruf bis zum Beenden dar und schließt einige Zeit mit ein, in der ich als Nutzer die Ergebnisse und besonders die jeweils vorgeschlagene Alternativanfrage konzentriert betrachte und bewerte, ohne dass das Programm irgendetwas tut. Wenn ein Nutzer aus Ermangelung an echten Alternativen die einzige vorgeschlagene Alternativanfrage wählt, ohne sie sich näher anzuschauen, verkürzt das die Bearbeitungszeit natürlich.

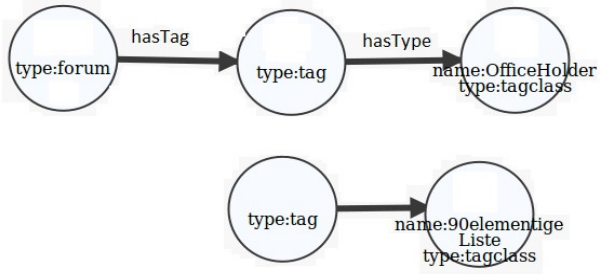

Abbildung 4: Die zweite Anfrage und die vorgeschlagene Alternative

Bei der zweiten Vorlage habe ich dem Knoten mit

dem Typen tagclass ein zusätzliches Attribut *Name* hinzugefügt und ihm den Wert „OfficeHolder“ gegeben. Dies ist eins der 16 Werte, die die gegebenen Datenbasis für das Attribut enthält. Es ist also ein Forum, das ein Tag besitzt, dessen Typ der Tagklasse „OfficeHolder“ entstammt (siehe Abbildung 4). Im ersten Durchlauf erhielt ich 406 Ergebnisse und entschied mich dazu, die Kardinalität auf 100 verringern zu lassen. Als Alternative wurde eine Anfrage vorgeschlagen, bei der der Knoten mit dem Typ Forum und die daran adjazente Kante, die den Knoten mit einem Tag verbindet, eliminiert wurden. Außerdem wurden beim Knoten des Typs tagclass die möglichen Werte des Attributs *Name* auf eine ganze Liste mit 90 Elementen erweitert, von denen viele in der Datenbasis bei keinem Knoten des Typs tagclass vorkommen (siehe Abbildung 4). Die Anfrage lieferte nach ihrer Ausführung exakt 100 Ergebnisse. Bei einem erneuten Durchlauf mit einer gewünschten Kardinalität von 200 Ergebnissen wurde eine ähnliche Anfrage vorgeschlagen, mit einer auf 116 Elementen verlängerten Liste möglicher Attributwerte. Sie lieferte leider unverändert 100 Ergebnisse, sodass ohne dass der Nutzer die Anfrage selbst ändert, das Erreichen des Ziels nicht möglich sein würde.

In einem weiteren Durchlauf wurde das gegenteilige Problem an der selben Anfrage probiert und sollte 700 Ergebnisse erreichen. Die vorgeschlagene Alternative unterschied sich nur durch die bei der ersten Anfrage beschriebenen allgemeinen Merkmale von der ursprünglichen Anfrage (Änderung der Reihenfolge der Attribute, Knoten und Kanten). Der queryType ist WHY_SO_MANY, egal, ob wirklich zu viele oder – wie hier – zu wenige Ergebnisse vorliegen, da in der Definition der Abfrage nicht zwischen den beiden Fällen unterschieden wird. Die Knoten und Kanten an sich wurden jedoch nicht geändert. Es wurden 406 Ergebnisse gefunden. Ohne die Anfrage manuell zu ändern, wäre es nicht möglich, das Ergebnis zu verbessern.

Das Ausführen der ersten Anfrage ging mit 20 Sekunden überraschend schnell, während die Findung der Alternativen – je nach gewählter Zielkardinalität – 70 bis 80 Sekunden dauerte. Zudem dauerte es 60 – 80 Sekunden (nach erwünschter Ergebniskardinalität: 60 sec bei 100, 70 sec bei 700 und 80 sec bei 200 Ergebnissen), bis die vorgeschlagene Alternative verarbeitet und deren Ergebnisse angezeigt wurden.

Dass das Finden von Alternativen länger dauert als die Ausführung für die ersten Ergebnisse liegt daran, dass jetzt, wenn das Problem und ggf. eine Zielkardinalität vorliegt, zusätzliche Operationen durchgeführt werden. Bei der zweiten Abfrage dauert die Ausführung der zweiten Alternative (Ziel: 200) deutlich länger als bei der ersten (Ziel: 100), was sich für mich nur durch die längere Attributliste des Knotens mit dem Typen tagclass erklären lässt. Darüber hinaus ist es schwierig, alle unterschiedlichen Verarbeitungszeiten zu erklären.

Bei allen Messungen ist anzumerken, dass mittels dieses Frontends nur die Zeit gemessen werden

kann, bis die Ergebnisse angezeigt wurden, und nicht die genaue Verarbeitungszeit, da alle 10 Sekunden abgerufen wird, ob die Ergebnisse vorliegen oder nicht (siehe Kapitel 5.6.3). Wenn also beispielsweise von 60 Sekunden die Rede ist, heißt das, dass die Verarbeitungszeit im Hintergrund zwischen 51 und 60 Sekunden gelegen haben muss.

7 Vergleichbares bei relationalen Datenbanken

Auch in relationalen Datenbanken können derartige Probleme auftreten. Die Datenbestände, die sich oft über viele Tabellen erstrecken, sind vielfach sehr groß, sodass es dem Nutzer schwer fällt, Anfragen so präzise zu formulieren (einschließlich Verwendung von Aggregatfunktionen etc.), dass die Ergebnisse seinen Vorstellungen entsprechen. Wenn dem Nutzer, nachdem er die Ergebnisse seiner Anfrage gesehen hat, das Auftreten einer der Probleme (leere Ergebnismenge, zu viele oder zu wenige Ergebnisse) auffällt, kann er die Anfrage manuell modifizieren/ verfeinern, was aufgrund seines beschränkten Wissens über die Daten sehr schwer ist.

Um dem Problem der leeren Antworten zu begegnen, wurden verschiedene Lösungen entwickelt. Ein nicht-interaktives Beispiel ist ein Framework von der Stanford University, das den iterativen Anfrageverfeinerungsprozess ein Stück weit automatisieren soll. Es besteht aus erweiterten Anfragen und Modifikationsoperatoren, mit denen sich solche Anfragen ändern lassen können. Erweiterte Anfragen bestehen aus einer relationalen Anfrage und einem Akzeptanztest, einer Funktion, die alle Antworttupel auf {true, false} abbildet. Akzeptanztests werden aus mehreren primitiven Akzeptanztests zusammengesetzt. Zudem werden Modifikationsoperatoren definiert, die auf eine Anfrage angewendet werden mit dem Ziel, dass die modifizierte Anfrage nun durch den Akzeptanztest angenommen werden kann. Zu jedem Akzeptanztest existiert mindestens ein Modifikationsoperator und umgekehrt. Verallgemeinerung ist ein transitiver Operator, der eine Anfrage schwächt, d.h. die Ergebnismenge einer gegebenen Anfrage für jede Datenbank des selben Schemas vergrößert. Zudem können eigene Verallgemeinerungsregeln definiert werden, z.B. durch Entfernung von Konjunktionen in konjunktiven Anfragen, Verallgemeinerung von Prädikaten und Konstanten (Ersetzen von Prädikaten bzw. Konstanten durch andere) sowie Nachbarschafts-Verallgemeinerung (Ersetzung von Werten durch Nachbarn, z.B. Näherungswerte bzw. Intervalle statt feste Werte). Die Verallgemeinerung soll möglichst minimal sein [CHAUD90].

Einen Kontrast dazu bildet ein Framework eines Forscherteams aus verschiedenen Ländern, das sich interaktiv verhält, d.h. anders als [CHAUD90] schlägt es nicht einfach Alternativanfragen vor, sondern führt den Nutzer durch mehrere Schritte, in denen die Anfrage jeweils etwas geändert wird, bis sie eine nichtleere Antwort liefert. In jedem Schritt werden dem Nutzer Lockerungsvorschläge

zur gegebenen Anfrage gemacht. Vorher wird die Wahrscheinlichkeit berechnet, dass der Vorschlag aus Nutzersicht sinnvoll ist und ihn dem Ziel der Anfrageveränderung näher bringt. Dazu werden die Kosten eines Vorschlags berechnet, die die Effektivität der Lockerung quantifizieren sollen. Dazu wird die Wahrscheinlichkeit, dass der Nutzer den Vorschlag annimmt bzw. ablehnt, der Wert des erwarteten Ergebnisses und weitere Änderungen benötigt, die auf die Anfrage angewendet werden können, wenn sie weiterhin keine Ergebnisse liefert. Umgesetzt wird das entweder durch einen Baum, der alle möglichen Lockerungssequenzen enthält und dann mittels Tiefensuche durchsucht wird oder den FastOpt-Algorithmus, der, anstatt den ganzen Baum zu expandieren, für jede mögliche Lockerung eine obere und eine untere Schranke für die Kosten seiner Kinder berechnet und dabei ermittelt, welche Zweige definitiv nicht zu optimalen Kosten führen und letztendlich schrittweise Teile eines Baumes aufbaut [MMRDPV13].

Umgekehrt gibt es auch in relationalen Datenbanken das Problem mit den zu vielen Ergebnissen, was durch Ranking und Kategorisierung gelöst werden kann. Ranking heißt, dass die Ergebnisse gemäß einer Zielfunktion sortiert werden, wobei Methoden aus dem probabilistischen Information Retrieval zum Einsatz kommen, die sich zur Modellierung von Datenabhängigkeiten und -korrelationen erweitern lassen. Besonders herausfordernd ist hier, dass alle gefundenen Tupel die definierten Bedingungen erfüllen und dass deshalb nicht definierte Attribute betrachtet werden und herausgefunden werden muss, welche Werte hier von Seiten des Nutzers erwünscht wären [SDHW06].

Ein anderer Ansatz besteht darin, die Tupel in Gruppen zu teilen und diese dem Nutzer zur Verfeinerung vorzuschlagen. Dabei kommt die Suche mit Facetten zum Einsatz, um durch die Datenbank zu navigieren. Dabei werden dem Nutzer nacheinander Fragen zu verschiedenen Attributen (die hier die Facetten darstellen) gestellt, wobei die Facetten in Abhängigkeit von den gegebenen Antworten der vorherigen Fragen gewählt werden. In Datenbanken ist es besonders herausfordernd, ohne Metadaten herauszufinden, welche Attribute der Tupel für Facettensuche geeignet sind. Wenn noch keine näheren Informationen bekannt sind, sollte angenommen werden, dass alle Tupel gleichermaßen zur Verfeinerung der Anfrage geeignet sind. Dabei wird ein Entscheidungsbaum zu einem Facettensuchsystem aufgestellt, das es ermöglichen soll, jedes Tupel möglichst schnell eindeutig zu identifizieren (was heißt, dass der Baum eine möglichst geringe durchschnittliche Höhe haben soll). Besser ist sogar, wenn der Nutzer mehrere Attribute vorgeschlagen bekommt und er wählt, zu welchem er den Wert angibt. Ergänzt wird dies durch Modellierungsmöglichkeiten für Ungewissheit (wenn der Nutzer mit „weiß ich nicht“ antwortet), die mit in den Entscheidungsbaum eingebaut wird [RWDNM08].

8 Anhang

8.1 Abbildungsverzeichnis

8.2 Literaturverzeichnis

[VTBL15] E. Vasilyeva, M. Thiele, C. Bornhövd, W. Lehner, Answering “Why Empty?” and “Why So Many?” queries in graph databases, Journal of Computer and System Sciences 82 (2016), Elsevier, 2015
[VTML15] E. Vasilyeva, M. Thiele, A. Mocan, W. Lehner, Relaxation of Subgraph Queries Delivering Empty Results, Proceedings of the 27th International Conference on Scientific and Statistical Database Management, ACM, 2015
[EDIT13] Interactive tool for creating directed graphs using d3.js, http://bl.ocks.org/cjrd/6863459, 2013
[DIAGR12] Bar chart, http://bl.ocks.org/mbostock/3885304, 2012
[CHAUD90] S. Chaudhuri, Generalization and a Framework for Query Modification, Proceedings of the Sixth International Conference on Data Engineering, IEEE Computer Society, 1990
[MMRDPV13] D. Mottin, A. Marascu, S.B. Roy, G. Das, T. Palpanas, Y. Velegrakis, A Probabilistic Optimization Framework for the Empty-Answer Problem, Proceedings of the VLDB Endowment, ACM, 2013
[SDHW06] S. Chaudhuri, G. Das, V. Hristidis, G. Weikum, Probabilistic Information Retrieval Approach for Ranking of Database Query Results, ACM Transactions on Database Systems (TODS) 31, ACM, 2006
[RWDNM08] S. Basu Roy, H. Wang, G. Das, U. Nambiar, M. Mohania, Minimum-effort driven dynamic faceted search in structured databases, Proceedings of the 17th ACM Conference on Information and Knowledge Management, ACM, 2008

8.3 Die Implementierung

Der Server mit den Anwendungen ist Bestandteil dieser Arbeit.