UNIVERSITÄT
LEIPZIG

Abschlussarbeit zur Erlangung des akademischen Grades
Bachelor of Science (B.Sc)

Thema:
Kontinuierliche Graph-Query Notifikationen auf Basis eines RDBMS

Vorgelegt von:
Maximilian Zimmer

Geboren am:
21.01.1999

in:
Erfurt

eingereicht am:
12. Juli 2021

Universitärer Betreuer:
Christopher Rost

Kontinuierliche Graph-Query Notifikationen auf Basis eines RDBMS

Maximilian Zimmer

Kurzfassung

Die Ergebnismengen mancher Anfragen müssen immer aktuell bleiben, unabhängig davon, wie volatil die zugrunde liegenden Daten sind. Für solche *Continuous Queries* muss bei jeder Änderung an der Datenbank überprüft werden, ob diese reevaluiert werden müssen. In der vorliegenden Bacheloarbeit wird ein Konzept zur Registrierung und Evaluierung von Graphanfragen vorgestellt, die auf ein RDBMS *gemappt* werden. Diese Graphanfragen werden als *Continuous Queries* registriert. Das bedeutet, dass bei Änderungen an der Datenbank ein Prüfmechanismus stattfinden muss, der durch DML-Anweisungen verursachte Ergebnismengenänderungen der Anfragen erkennt. Es wird gezeigt, wie eine Implementierung der Registrierung der Anfragen und eine Erkennung von Ergebnismengenänderungen in Java umgesetzt werden kann. Abschließend wird das implementierte Verfahren bezüglich der benötigten Zeit analysiert. Diese Evaluierung zeigt, dass die zeitliche Performanz der Anfrageregistrierung dieses Verfahrens besonders von der Art der in der **WHERE**-Klausel genutzten, *logischen Operatoren* abhängt. Die zeitliche Performanz der Ergebnismengenänderung hängt hingegen von der Anzahl registrierter Anfragen ab, die ein *Knoten-* beziehungsweise *Kantenlabel* betreffen, und ob die Überprüfung in historischen oder aktuellen Daten durchgeführt werden muss.

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Für Applikationen, deren Daten stark in Beziehung zueinander stehen, ist es attraktiv, diese in einem Graphen zu modellieren. Diese Speicherstruktur legt weniger Wert darauf, die Daten einzeln zu betrachten, sondern die Beziehungen untereinander zu analysieren. Da an solchen Graphen sehr hochfrequent Änderungen vorgenommen werden, ist es essentiell, dass für einen Nutzer aus Anfragen generierte, relevante Daten schnell und effizient aktualisiert werden. Um Informationen von einer solchen Speicherstruktur zu erhalten, kann jedoch nicht SQL als Anfragesprache verwendet werden, wie es bei relationalen Datenbanksystemen üblich ist, sondern eine Graphanfragesprache, wie beispielsweise Cypher für Neo4J Graphdatenbanken oder PGQL für Oracle Datenbanken. Auf Letzterer, PGQL, liegt der Fokus in diesem Projekt. Für Anfragen auf diesen Graphen, die immer auf dem neuesten Stand sein müssen und sich häufig ändern, wie zum Beispiel die aktuellen Freundesbeziehungen eines Nutzers im sozialen Netzwerk oder der aktuelle Preis einer Aktie eines Unternehmens, muss ein effizientes Verfahren implementiert werden, so dass der Nutzer nach der Änderung der relevanten Daten so schnell wie möglich eine Benachrichtigung erhält. Es ist jedoch sehr ineffizient Anfragen bei jeder Änderung an der Datenbank neu zu evaluieren, da die vielen zusätzlichen Zugriffe die Verfügbarkeit einschränken, von vielen Änderungen oftmals nur sehr wenige einen bestimmten Nutzer betreffen und viele, hochfrequente Änderungen am Graphen vorgenommen werden können, wie beispielsweise INSERT, UPDATE oder DELETE Anweisungen. Daher ist es notwendig, dass Anfragen analysiert werden und einer bestimmten Gruppe an Entitäten zugewiesen wird, so dass Überprüfungen nur stattfinden müssen, wenn an von einer Anfrage betroffenen Entitäten etwas geändert wird.

1.2 Problemstellung

Das Problem wird nun anhand eines Beispiels verdeutlicht.

Abbildung 1: Freundschaftsbeziehung

Im Graphen 1 ist die Freundschaftsbeziehung zwischen zwei Personen modelliert. Beide haben einen Vor- und Nachnamen und sind mit der gerichteten Kante *friendOf* verbunden, die anzeigt, dass *Franz Schmidt* ein Freund von *Hans Meier* ist. Ein

Nutzer möchte nun die Vornamen aller Personen erfahren, von denen *Franz Schmidt* ein Freund ist und deren Vorname entweder *Lisa* oder deren Hobby Fußball ist. Eine entsprechende PGQL-Anfrage ist in Listing 1 dargestellt.

Listing 1: PGQL Anfrage

```
SELECT p2.firstname as firstname
FROM MATCH (p1:Person) - [friendOf] -> (p2:Person)
WHERE p1.firstname = 'Franz' AND p1.lastname = 'Schmidt'
AND (p2.firstname = 'Lisa' OR p2.hobby = 'Football')
```

Weiterhin soll er eine Benachrichtigung erhalten, sobald sich an der Ergebnismenge seiner Anfrage etwas ändert. Die Anfrage wird also registriert, indem restriktierende und selektierende Eigenschaften extrahiert werden, und der Nutzer benachrichtigt, sollten Änderungen am Graphen 1 dazu führen, dass sich die Ergebnismenge seiner Anfragen ändert. Es werden anschließend zu unterschiedlichen Zeitpunkten t_i bestimmte Änderungen vorgenommen:
$t_1 \rightarrow$ Einfügen einer Person namens *Lisa Schneider*
$t_2 \rightarrow$ Einfügen einer Freundschaftsbeziehung zwischen *Franz Schmidt* und *Lisa Schneider*
$t_3 \rightarrow$ Hinzufügen des Hobbys *Fußball* zu *Lisa Schneider*
Alle Operationen geschehen nacheinander, beginnend mit t_1. Der nach den Einfügeoperationen entstandene Graph ist in Abbildung 2 gezeigt.

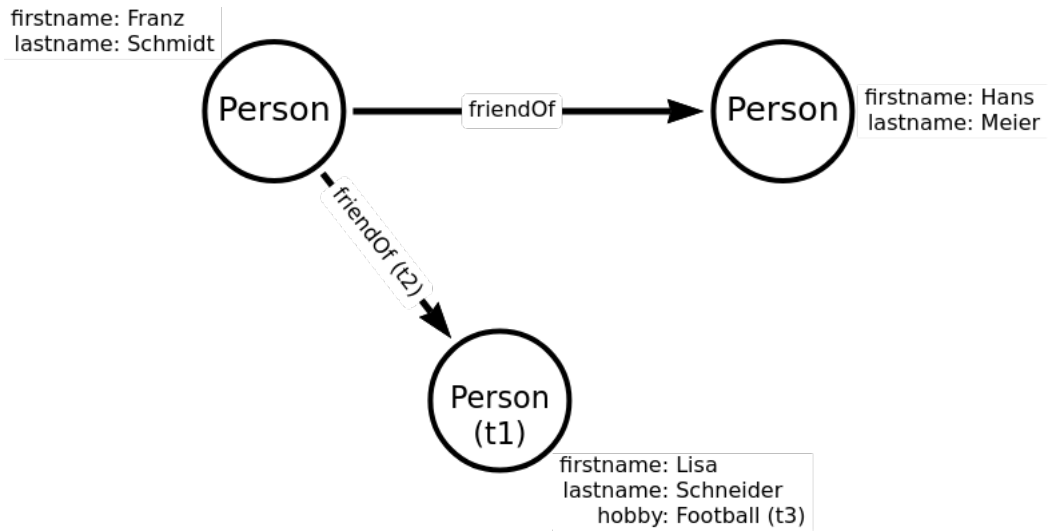

Abbildung 2: Freundschaftsbeziehungen - erweitert

Im Graphen 2 wurden die Zeitstempel der Einfügeoperationen in Klammern hinter die eingefügten Eigenschaften, Knoten oder Kanten geschrieben. Bezüglich der Anfrage,

hat sich nach der ersten Einfügeoperation zum Zeitpunkt t_1 noch nichts geändert, da keine neuen Freunde zu *Franz Schmidt* hinzugefügt worden sind, sondern lediglich ein weiterer Personenknoten zum Graphen hinzugefügt wurde. Zum Zeitpunkt t_2 wird dann eine Benachrichtigung an den Nutzer gesendet, da sich die Ergebnismenge seiner Anfrage änderte. Der Vorname *Lisa* wird in die Ergebnismenge der Anfrage aufgenommen, da der Personenknoten nun die Anfrageeinschränkung erfüllt, nämlich der Zielknoten einer *friendOf*-Kante zu sein, die gleichzeitig einen Personenknoten mit Vornamen *Franz* und Nachnamen *Schmidt* als Quellknoten besitzt. Die Einfügeoperation zum Zeitpunkt t_3 ändert die Ergebnismenge der Anfrage nicht mehr, da der Personenknoten mit Vorname *Lisa* bereits Teil der Ergebnismenge ist und somit das Einfügen des Hobbys *Football* keine Änderung verursacht.

1.3 Zielsetzung

Ziel dieser Arbeit ist es, ein Verfahren zu entwickeln, dass die in Kapitel 1.2 angeschnittenen Probleme effizient löst. Zum einen die Registrierung der Anfragen, bei der relevante SQL-Prädikate herausgefiltert und abgespeichert werden, Trigger für die verschiedenen DML-Anweisungen erstellt werden und Filter entsprechend der selektierten Prädikate angelegt werden müssen. Die Anfragen sind in SQL gestellt, da der Graph in einem relationalen Datenbanksystem gespeichert ist. Um es dem Nutzer zu ermöglichen, die Anfragen in PGQL zu stellen, wurde ein bereits entwickelter Übersetzer zur Verfügung gestellt, dessen Funktionsweise in Kapitel 2.7 erläutert wird. Zum anderen muss ein System entwickelt werden, das abhängig vom gewünschten Benachrichtigungstyp des Nutzers, Notifikationen an diesen sendet. Um dies umzusetzen, muss ein Algorithmus implementiert werden, der Ergebnismengenänderungen bei Änderungen an der Datenbank erkennt. Sowohl die Anfrageregistrierung als auch die Erkennung von Ergebnismengenänderung werden bezüglich der zeitlichen Performanz analysiert. Zunächst wird dafür im Kapitel 2 grundlegende, für diese Arbeit notwendige, Begriffe erklärt. Anschließend wird im Kapitel 3 auf verwandte Arbeiten in diesem Bereich eingegangen, bevor danach im Kapitel 4 auf konkrete Umsetzungen der Lösungen eingegangen wird. Daraufhin wird deren zeitliche Performanz in Kapitel 5 evaluiert.

2 Grundlagen

2.1 Continous Queries vs Select Queries

Eine normale Select Anfrage, wie man sie aus SQL kennt, stellt eine Anfrage an die Datenbank, gibt die Ergebnismenge einmal zurück und terminiert anschließend. Es gibt jedoch Anfragen, bei denen ein Nutzer wissen möchte, sobald sich die Ergebnismenge ändert. Beispielsweise möchte er benachrichtigt werden, sobald sich die Anzahl seiner Freunde auf einer Social Media Plattform ändert. Hier kommen Continous Queries ins Spiel [1]. Dies sind Anfragen, die zeit- oder ortsabhängig sind und kontinuierlich reevaluiert werden müssen. Ein Beispiel für eine ortsabhängige Anfrage wäre die Suche nach der von dem Nutzer aus nächstgelegenen Tankstelle. Sobald sich der Nutzer bewegt, ändert sich die Ergebnismenge dieser Anfrage nach einer bestimmten zurückgelegten Strecke. Zeitabhängige Anfragen sind beispielsweise, wie schon oben erwähnt, die Menge an Freunden eines Nutzers einer Social Media Plattform. Diese Ergebnismenge ist nur zusammen mit einem Zeitpunkt wohldefiniert und kann sich ständig ändern. In diesem Projekt geht es unter anderem um die Registrierung und die Reevaluierung solcher zeitabhängigen Continous Queries.

2.2 Arten von Benachrichtigungen

Vemuri et al. stellen in [2] Continous Query Notifications vor. Die dort definierten Arten der Benachrichtigungen, zusammen mit denen der Dokumentation von Oracle [3], wurden in diesem Absatz übernommen. Grundsätzlich wird zwischen zwei Arten der Benachrichtigung unterschieden, der Object Change Notification (OCN) und der Query Result Change Notification (QRCN). Die Unterschiede werden anhand der in Listing 2 gezeigten Anfrage verdeutlicht.

Listing 2: Beispielanfrage SQL

```
SELECT salary
FROM employees
WHERE deparment_id = 10
```

Object Change Notification
Bei der OCN wird eine Benachrichtigung gesendet, sobald eine Transaktion etwas an einem, in der Anfrage vorkommenden, Objekt ändert. Hierbei spielt es keine Rolle, ob sich die Ergebnismenge der Anfrage ändert. Bezüglich der Anfrage 2 wird also eine Benachrichtigung gesendet, sobald eine Transaktion etwas an der *Employee*-Tabelle ändert, unabhängig davon ob es das Prädikat der Anfrage erfüllt wird (beispielsweise sollte die $department\_id = 5$ sein).

Query Result Change Notification
Bei der QRCN wird erst dann eine Benachrichtigung gesendet, sollte eine Transaktion etwas an der Ergebnismenge einer Anfrage ändern. Bezüglich der Anfrage 2 wird eine Benachrichtigung gesendet, sollten beispielsweise Zeilen in die *Employee*-Tabelle eingefügt oder gelöscht werden, die das Anfrageprädikat ($deparment\_id = 10$) erfüllen oder sollte das Gehalt eines Mitarbeiters aktualisiert werden, der im Department mit der $deparment\_id = 10$ angestellt ist.

2.3 Property Graph Model

Die nachfolgenden Erläuterung zu dem Modell sind aus der Arbeit[4] und der Webseite [5] entnommen. Ein Property Graph besteht aus Knoten mit einem oder mehreren Labels, im Beispielgraphen 3 des sozialen Netzwerks also *Person*, *Post*, *Comment*, und Kanten zwischen den Knoten, die die Beziehungen untereinander modellieren und ebenfalls mit einem oder mehreren Labels versehen sind. Beispielsweise existiert zwischen dem Knoten *Comment* und *Person* eine Kante mit dem Label *hasCreator*, die beschreibt, wer der Verfasser des Kommentars ist, und zwischen den Knoten *Person* und *Post* eine Kante *likes*, die aussagt, dass diese Person diesen Post geliket hat. Sowohl Knoten als auch Kanten können eine beliebige Menge (auch die leere Menge) an Schlüssel-Wert-Paaren enthalten, die ihre Eigenschaften beschreiben. Diese können variabel hinzugefügt oder entfernt werden und sind schemafrei, was bedeutet, dass ein Knoten oder eine Kante unterschiedliche beliebige Eigenschaften haben kann, anders als in einem relationalen Modell, bei dem das Schema der Eigenschaften eindeutig durch die Spalten der zugehörigen Tabelle vorgegeben ist. Im Beispiel wurde das verdeutlicht, indem ein Personenknoten das Attribut *hobby* hat, der andere *age* und der letzte nur die Eigenschaften *firstname* und *lastname* besitzt. Auch Kanten können eine beliebige Anzahl schemafreier Eigenschaften haben. Im Beispielgraphen 3 haben die Kanten jedoch keine Attribute, sondern lediglich ein Label.

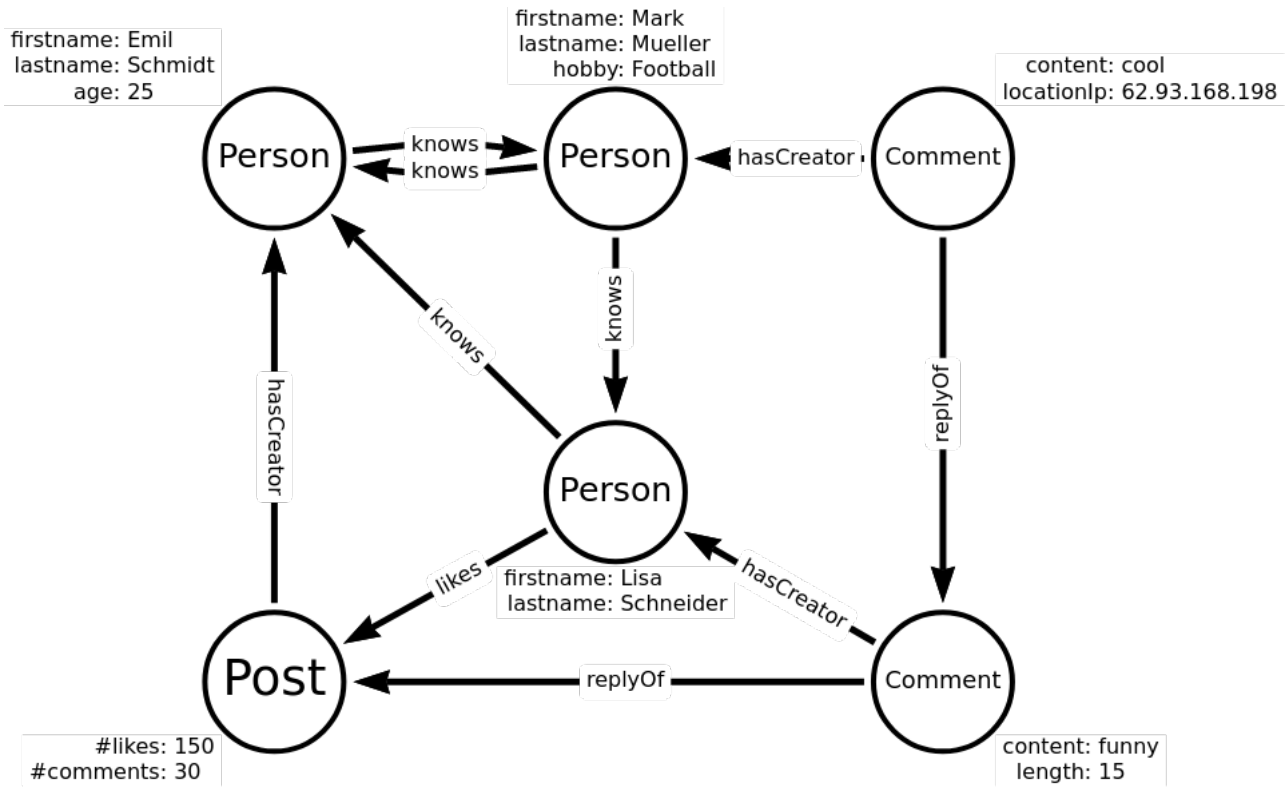

Abbildung 3: Graph - soziales Netzwerk

2.4 Temporal Property Graph Model

In diesem Absatz genutzte Informationen zum Temporal Property Graph Model (TPGM) stammen aus der Publikation [6]. Werden die Knoten, Kanten und Eigenschaften des Beispielgraphen 3 um einen Start- und Endzeitstempel erweitert (*txStart, txEnd*), die beschreiben, wann eine Kante, ein Knoten oder eine Eigenschaft hinzugefügt beziehungsweise gelöscht wurde, so wird vom Temporal Property Graph Model, ein Modell zur Darstellung bitemporaler Graphen, gesprochen, dargestellt in Abbildung 4. Ist der Wert aktuell, so wird der Endzeitstempel auf den größten Zeitstempel gesetzt, den das System erlaubt (standardgemäß *9999-12-31*, in diesem Fall bedingt durch MariaDB *38-01-19*). Somit sind Knoten und Kanten, zusammen mit deren Eigenschaften, nur in einem bestimmten Zeitintervall aktiv, und es können aktuelle, aber auch historische Analysen durchgeführt werden. Im Beispielgraphen 4 wurden die Start- und Endzeitstempel der Eigenschaften aus Übersichtsgründen weggelassen. Auf dem Temporal Property Graph Model können verschiedene Operationen durchgeführt werden, wie beispielsweise Transformation und Gruppierung, die auch auf dem Property Graph Model möglich sind, aber auch zeitbezogene Snapshots und Analysen der Graph Evolution, die nur auf dem TPGM möglich sind. Unter

Transformation wird beispielsweise das Hinzufügen eines neuen Attributs verstanden, das sich aus bereits vorhandenen Attributen ableitet. Zum Beispiel kann auf einem Knoten das Attribut *Dauer* hinzugefügt werden, das sich aus der Differenz von *txStart* und *txEnd* ergibt. Beim Gruppieren werden Knoten, die in einer oder mehreren Eigenschaften gleich sind, zusammengetragen, und es wird eine aggregierte Funktion auf eines ihrer Attribute ausgeführt, wie es auch in SQL üblich ist. Die Snapshot-Operation macht einen Snapshot der Datenbank an einem bestimmten Zeitpunkt oder für einen bestimmten Zeitraum. Hier kann beispielsweise eine Zeit angegeben werden und liegt diese zwischen dem *txStart* und *txEnd* Attribut eines Knotens, so wird dieser in den Snapshot aufgenommen. Die Graphevolution-Operation dient dann dazu, zwei Snapshots zu verschiedenen Zeitpunkten miteinander zu vergleichen und die Unterschiede anzuzeigen. Somit lassen sich die Entwicklungen des Graphen visualisieren und besser nachvollziehen.

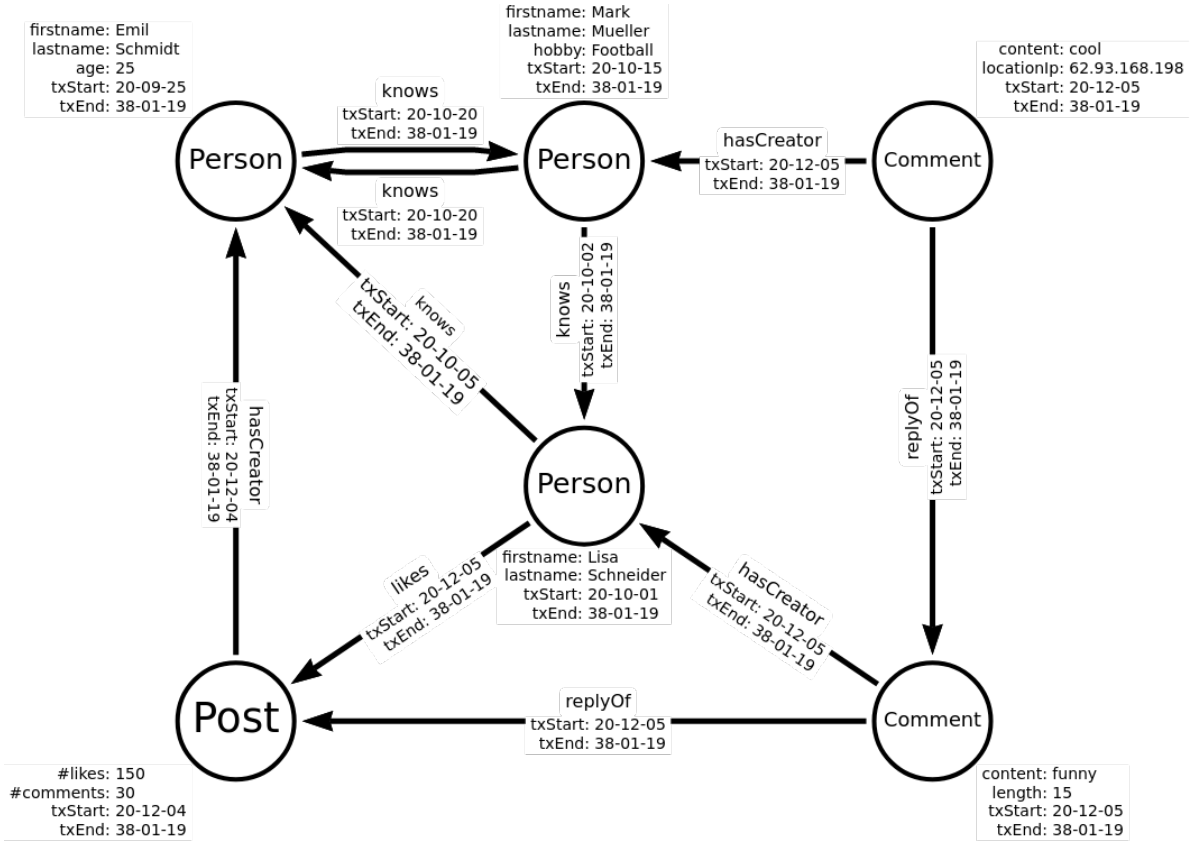

Abbildung 4: Graph - soziales Netzwerk mit Zeitstempeln

2.5 PGQL / TPGQL

Die SQL basierte, für das Property Graph Model ausgelegte Anfragesprache PGQL [7], ermöglicht es, Muster von Knoten und Kanten im Graphen zu suchen und ausgewählte Eigenschaften der Knoten und Kanten tabellenförmig zurückzugeben. Entwickelt wurde PGQL von Oracle und wird von Graphdatenbanken von Oracle unterstützt. Oskar van Rest et al. beschäftigten sich in der Arbeit [8], die diesem Absatz als Quelle dient, mit dieser Anfragesprache. Erklärt wird die Syntax mit Hilfe der im Listing 3 definierten Anfrage.

Listing 3: PGQL-Anfrage

```
SELECT p.lastname AS lastname, c.content AS content
FROM MATCH (c:Comment) - [hasCreator] -> (p:Person)
WHERE p.firstname='Hans' AND c.length > 50
```

Das **SELECT** Statement gibt hier, wie auch in SQL üblich, an, welche Attribute tabellarisch zurückgegeben werden sollen. In der **FROM MATCH** Klausel steht hier eine Cypher-Anfrage, die ein Muster definiert. Hierbei zeigen runde Klammern einen Knoten und eckige Klammern eine Kante an. Innerhalb der Klammern können optional Aliasse angegeben werden, die benötigt werden, sollte eine Eigenschaft dieses Knotens oder dieser Kante zurückgegeben oder in der **WHERE** Klausel genutzt werden. Weiterhin muss auch vor dem `":"` kein Alias stehen, sondern lediglich nach dem `":"` ein Label, um das Muster zu restriktieren. Das Muster kann nach Kanten unabhängig von ihrer Richtung suchen, indem vor und nach der Kante kein Pfeil angegeben wird, beispielsweise `() - [] - ()`. Hier werden alle Kanten getroffen, unabhängig davon, ob sie ein- oder ausgehend sind. Alternativ ist es möglich eine Richtung anzugeben, wie auch im Beispiel 3, in dem vor oder nach der Kante ein Pfeil hinzugefügt wird. Im Bezug auf die PGQL-Anfrage 3 werden also nur Kanten gesucht, die ausgehend von einem Kommentarknoten und eingehend in einen Personenknoten sind. Die **FROM MATCH** Klausel gibt nun also einen Teilgraphen zurück, der zusätzlich mit einer **WHERE** Klausel noch auf bestimmte Eigenschaften der Knoten und Kanten gefiltert werden kann. In unserer Beispielanfrage 3 werden also nur Teilgraphen zurückgegeben, die einen Kommentarknoten mit einer Länge größer als 50 Zeichen besitzen, einen Personenknoten beinhalten, dessen Vorname gleich Hans ist, und die eine Kanten vorweisen, die das Label *hasCreator* hat und die beiden Knoten, mit Richtung zum Personenknoten, miteinander verbindet. PGQL bietet noch weitaus mehr Funktionalitäten, wie beispielsweise das Suchen von Pfaden mit beliebiger Länge, in dem ein Stern an den Kantenpfeil angefügt wird : `(a) - [] ->* (b)`. Diese Anfrage kann kombiniert werden, mit **FROM MATCH SHORTEST**, die den kürzesten Pfad aus allen Pfaden zwischen den beiden Knoten findet. Also gibt die in Listing 4 definierte Anfrage die minimale Anzahl an Personen zurück, über die sich Personen mit dem Vornamen Hans und

Listing 4: PGQL-Anfrage - MATCH SHORTEST

```
SELECT count(k) AS pathLength
FROM MATCH SHORTEST (p1:Person)-[k:knows]->(p2:Person)
WHERE p1.firstname ='Hans' AND p2.firstname = 'Lisa'
```

Lisa kennen.
Weiterhin ist es auch möglich, das Label vollständig wegzulassen. Beispielsweise wird bei folgender Anfrage einfach jeder Knoten zurückgegeben
`SELECT * FROM MATCH (v)`
Ergebnismengen solcher Anfragen ändern sich also bei jeder DML-Anweisung, die etwas an einem Knoten in der Graphdatenbank ändert. Beim Definieren der Labels des Knotens oder der Kante in der PGQL Anfrage kann auch mit logischen Operatoren gearbeitet werden. Somit gibt also `SELECT * FROM MATCH (v:Person|Universität)` sowohl alle Knoten mit Label Person, als auch alle Knoten mit Label Universität zurück, da ein *logisches Oder* zwischen den Labels steht. Standard Graphanfragesprachen, wie beispielsweise PGQL, Cypher oder Gremlin, unterstützen nativ keine temporalen Graphen. Aus diesem Grund wird in einem aktuellen Forschungsprojekt an der Univsersität Leipzig an einer Erweiterung für PGQL, names T-PGQL, gearbeitet. Hier wird PGQL um eine temporale Syntax erweitert, die es ermöglicht, zeitliche Einschränkungen in die Graphanfragen aufzunehmen. Ähnliche Erweiterungen wurden bereits zur *Graph Definition Language*, kurz GDL, hinzugefügt, wodurch Temporal-GDL entstand [9]. Beide Spracherweiterungen können auf Graphen im TPGM angewendet werden. Ein Beispiel für diese temporale Erweiterung ist es, wenn die **WHERE**-Klausel der PGQL-Anfrage 3 um folgendes Prädikat erweitert wird `p.txStart > '2020-10-10'` Nun werden nur noch Personen in die Ergebnismenge aufgenommen, die erst nach dem 10. Oktober 2020 hinzugefügt wurden. In diesem Projekt konnte nur eine kleine Untermenge an Funktionalitäten von TPGQL / PGQL umgesetzt werden. Welche das sind und wieso andere nicht umgesetzt wurden, wird im Kapitel HIER KAPITEL EINFÜGEN genauer erklärt.

2.6 Repräsentation des Graphen im RDMS

In diesem Projekt wird der TPGM Graph nicht in einer Graphdatenbank abgespeichert, sondern in dem relationalen Datenbankmanagementsystem MariaDB modelliert. Um diese Repräsentation umzusetzen, gibt es viele verschiedene Tabellenschemas, wie zum Beispiel das Speichern aller Knoten in einer Tabelle und alle Kanten in einer anderen. Für diese Arbeit wurde ein bereits vorhandenes Schema vorgegeben, das im Folgenden erklärt wird.

Vorgegebenes Tabellenschema
Für jedes Knoten- und Kantenlabel werden zwei system-versionierte (temporale) Tabellen benötigt. Diese temporale Tabellen fügen Start- und Endzeitpunkt eines Datensatzes automatisch ein und setzen bei Löschanweisungen (**DELETE**) den Endzeitpunkt *txEnd* auf den Zeitpunkt der Löschung [10]. Bei Aktualisierungen (**UPDATE**) wird bei dem aktualisierten Datensatz der Endzeitstempel gesetzt, und es wird ein neuer Datensatz mit den aktualisierten Werten eingefügt. Diese Tabellenart erlaubt es, den Verlauf aller Datenänderungen lückenlos zu speichern und einfache zeitbezogene Analysen zu ermöglichen. Die erste der beiden Tabellen ist die Knoten- oder Kantentabelle selbst, in der die ID des Knoten oder der Kante, zusammen mit dem Label und dem Start- und Endzeitpunkt gespeichert ist. Die zweite Tabelle ist die Eigenschaftstabelle, in der die zu dem Knoten oder der Kante zugehörigen Schlüssel-Wert-Paare gespeichert sind. Die Tabelle besteht aus der ElementID, die eine Fremdschlüsselbeziehung zur zugehörigen ID der Knoten-/Kantetabelle hat, dem Eigenschaftsschlüssel, dem Eigenschaftswert, und die zur Eigenschaft gehörigen Start- und Endzeitstempel, wie es im Bild 5 dargestellt ist. Durch diese Art der Speicherung wird die Schemafreiheit der Eigenschaften gewährleistet. Eine Einschränkung ist jedoch der Datentyp für das *pvalue*-Attribut. Dieser ist fix und muss zur Weiterverarbeitung, sollte ein Datentyp wie *Integer* oder *Timestamp* benötigt werden, zunächst konvergiert werden. Die Kantentabellen besitzen neben der ID und dem *Label* noch eine Spalte mit Namen *Source* beziehungsweise *Target*, wie in Abbildung 6 dargestellt. Diese haben eine Fremdschlüsselbeziehung zur ID des entsprechenden Quell- beziehungsweise Zielknotens. In diesem Fall ist das Label des Quell- und Zielknotens gleich, nämlich *Person*. Somit referenzieren beide Fremdschlüssel die gleiche Tabelle.

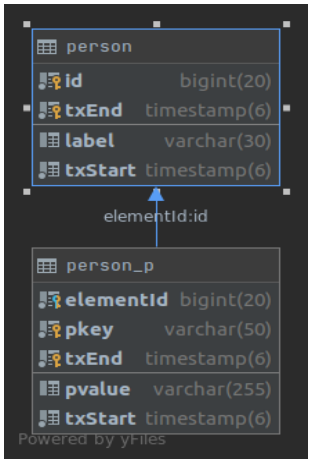

Abbildung 5: Knoten mit Label *Person*

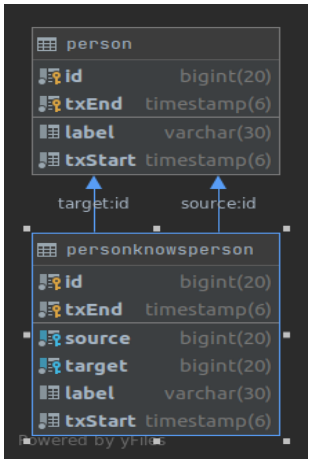

Abbildung 6: Kante mit Label *knows*

Beispiel
Beispielhaft wird nun betrachtet, wie die Personenknoten mit Nachnamen *Schmidt* und *Mueller* und deren Beziehung zueinander im MariaDB modelliert wird. Tabelle 1 stellt die zu dem Label *Person* zugehörige Knotentabelle dar.

Tabelle 1: Knotentabelle - Label *Person*

Person			
id	**label**	**txStart**	**txEnd**
1	Person	20-09-25	38-01-19
2	Person	20-10-15	38-01-19

Es muss noch angemerkt werden, dass der Zeitstempel *38-01-19* der größte von MariaDB unterstütze Zeitstempel ist, und somit für noch aktuelle Daten als *txEnd* gesetzt ist. Tabelle 2 ist die zu der Personentabelle 1 zugehörige Eigenschaftstabelle.

Tabelle 2: Eigenschaftstabelle - Label *Person*

Person_p				
elementId	**pkey**	**pvalue**	**txStart**	**txEnd**
1	firstname	Emil	20-09-25	38-01-19
1	lastname	Schmidt	20-09-25	38-01-19
1	age	25	20-09-25	38-01-19
2	firstname	Mark	20-10-15	38-01-19
2	lastname	Mueller	20-10-15	38-01-19
2	hobby	football	20-10-15	38-01-19

Durch diese Art der Speicherung in der Eigenschaftstabelle können beliebig Attribut hinzugefügt, gelöscht oder geändert werden. Weiterhin ist zu erkennen, ob eine Eigenschaft nachträglich eingefügt wurde oder wie in diesem Fall, zusammen mit dem Knoten, da der *txStart* Zeitstempel sowohl bei den Knoten als auch bei den zugehörigen Eigenschaften übereinstimmt. Ändert nun *Emil Schmidt* seinen Namen zu *Emil Beutlin*, so wird die zweite Zeile in Tabelle 2 nicht gelöscht, sondern der Wert von *txEnd* auf das aktuelle Datum gesetzt. Weiterhin wird eine neue Zeile eingefügt, die nun die Attribute *elementId*=1, *pkey*=lastname und *pvalue*=Beutlin besitzt. Die Tabelle 3 visualisiert diese Änderungen für den Fall, dass das Aktualisieren des Namens am 1. März 2021 stattgefunden hat.

Tabelle 3: Eigenschaftstabelle aktualisiert - Label *Person*

Person_p				
elementId	**pkey**	**pvalue**	**txStart**	**txEnd**
1	firstname	Emil	20-09-25	38-01-19
1	lastname	Schmidt	20-09-25	21-03-01
1	age	25	20-09-25	38-01-19
1	lastname	Beutlin	21-03-01	38-01-19
2	firstname	Mark	20-10-15	38-01-19
2	lastname	Mueller	20-10-15	38-01-19
2	hobby	football	20-10-15	38-01-19

Zur Tabelle 3 muss noch angemerkt werden, dass die Zeile vor der Aktualisierung, deren Endzeitstempel nun auf den 1. März 2021 gesetzt wurde, bei einer Anfrage auf die aktuellen Daten nicht mehr angezeigt wird. Um diese Zeile mit in die Ergebnismenge aufzunehmen, muss in den historischen Daten gesucht werden, beispielsweise indem die **FROM**-Klausel um den Zusatz **FOR SYSTEM_TIME ALL** erweitert wird. Um die Beziehungen der Knoten zu speichern, wird die Kantentabelle 4 benötigt.

Tabelle 4: Kantentabelle - Label *knows*

PersonKnowsPerson					
id	**source**	**target**	**label**	**txStart**	**txEnd**
1	1	2	knows	20-10-15	38-01-19
2	2	1	knows	20-10-15	38-01-19

In dieser werden also nun, wie in Paragraph 2.6 bereits erklärt, die beiden Personenknoten in Relation miteinander gesetzt. Dies geschieht, indem der Primärschlüssel *id* der beiden Personenknoten einmal in der *source*- und einmal in der *target*-Spalte referenziert wird. Die Kante besitzt selbst noch eine eigene ID, die den Primärschlüssel darstellt. Auch die Kantentabelle hat noch eine zugehörige Eigenschaftstabelle, in diesem Fall PersonKnowsPerson_p. Da die Kante jedoch keine Eigenschaften besitzt, ist die Tabelle leer und wird somit weggelassen.

2.7 Von PGQL zu SQL

Da nun klar ist, wie der Graph in dem relationalen Datenbankmanagementsystem repräsentiert wird, steht nun noch offen, wie TPGQL Anfragen darauf funktionieren. Hierzu ist ein Übersetzer notwendig, der wie die Architektur, vorgegeben war. Für jede Eigenschaft, die von einem Knoten benötigt wird, muss die zu dem Knoten gehörige Eigenschaftstabelle *gejoint* werden, und für jede Kante die benötigt wird, die entsprechende Kantentabelle. Durch die vielen Joins der Tabellen werden aus relativ kurzen TPGQL Anfragen deutlich längere SQL Anfragen. Um dies zu verdeutlichen, wird die in Listing 5 definierte PGQL-Anfrage in SQL übersetzt.

Listing 5: PGQL-Anfrage

```
SELECT p.lastname AS lastname, c.content AS content
FROM MATCH (c:Comment) - [hasCreator] -> (p:Person)
WHERE p.firstname='Hans' AND c.length > 50
```

Zunächst ist zu erkennen, dass sowohl vom Personenknoten als auch vom Kommentarknoten Eigenschaften benötigt werden. Somit müssen beide Eigenschaftstabellen, genauso wie die Kantentabelle *gejoint* werden. Die in SQL übersetzte Anfrage ist in Listing 6 gezeigt.

Listing 6: SQL-Übersetzung

```
SELECT p_p1.pvalue as lastname, c_p1.pvalue as content
FROM comment c LEFT JOIN comment_p c_p1 ON c.id = c_p1.
   elementId
AND c_p1.pkey='content'
LEFT JOIN comment_p c_p2 ON c.id = c_p2.elementId
AND c_p2.pkey='length'
JOIN commenthascreatorperson e ON e.source = c.id
JOIN person ON p.id = e.target
LEFT JOIN person_p p_p1 ON p.id = p_p1.elementId
AND p_p1.pkey='lastname'
LEFT JOIN person_p p_p2 ON p.id = p_p2.elementId
AND p_p2.pkey='firstname'
WHERE p_p2.pvalue='Hans' AND c_p2.pvalue > 50
```

Dadurch, dass in der Beispielanfrage 5 von jedem der beiden Knoten zwei Eigenschaften benötigt werden, müssen die zugehörigen Eigenschaftstabellen jeweils zwei mal *gejoint* werden. Eigenschaftstabellen werden mit einem **LEFT JOIN** gejoint, da auch Knoten, die keine Eigenschaft besitzen, zunächst in das Muster mit aufgenommen werden. Der benötigte Eigenschaftsschlüssel wird immer direkt nach dem **LEFT JOIN** selektiert und wird immer als Alias für den zurückgegebenen Wert in der **SELECT**-Klausel gesetzt.

2.8 ActiveMQ

Da es sich um Benachrichtigungen für die oben erwähnten Continous Queries handelt, muss auch ein Message Broker genutzt werden, der diese Benachrichtigungen erstellt, versendet und empfängt. Hier wird ActiveMQ genutzt, der vollständig den Java Message Service 1.1 (JMS) implementiert. Der Broker fungiert in diesem Fall als Server, während alle anderen Anwendungen im Message Service Netzwerk als Klienten agieren. Die in diesem Absatz genutzen Informationen stammen von der Webseite [11]. ActiveMQ kennt zwei Arten von Zielen für die Benachrichtigungen, die Queue und das Topic. Zunächst zur Queue. Eine Queue funktioniert nach dem FIFO Prinzip (first in, first out) und kennt zwei Arten von *Clients*, den *Producer* und den *Consumer*. Der *Producer* erstellt die Benachrichtigung und lädt diese in die Queue, und die *Consumer* ziehen sich die Nachrichten aus der Queue, eine nach der anderen. Hat ein *Consumer* eine Nachricht empfangen, so wird diese aus der Queue gelöscht und sie steht keinem anderen *Consumer* mehr zur Verfügung. Das bedeutet, dass eine Nachricht nur von genau einem *Consumer* gelesen werden kann. Das Topic kennt ebenfalls zwei Arten von *Clients*, den *Publisher* und den *Subscriber*. Hier sendet der *Publisher* eine Nachricht an ein Topic, und jeder *Subscriper* des Topics erhält diese Nachricht.

3 Verwandte Arbeiten

Patent CQN
Im Patent für *Continuous Query Notifications* von Oracle [2] werden Mechanismen zum effizienten Auswerten von *Continuous Queries* für relationale Datenbanken beschrieben. Grundsätzlich wird hier zwischen zwei Phasen unterschieden, der Registrierungsphase und der Transaktions-Evaluations-Phase. Während der Registrierungsphase wird der Anfrage eine einzigeartige *queryId* zugewiesen und die Prädikate der Anfrage extrahiert. Diese Prädikate sind boolsche Ausdrücke in der **WHERE**-Klausel und dienen anschließend per Konjunktion, verbunden mit der Disjunktion, einer abstraken *CHG_COL*-Funktion als Schlüssel einer Map, deren Wert die entsprechende *queryId* der zugehörigen Anfrage ist. Diese abstrake *CHG_COL*-Funktion bekommt als Parameter eine Spalte mitgegeben und liefert *TRUE* zurück, sollte an dieser Spalte eine Änderung vorgenommen worden sein, sonst *FALSE*. Jede Spalte, die in der **SELECT**- oder **WHERE**-Anweisung vorkommt, muss mit der *CHG_COL*-Funktion überprüft werden. Somit kann diese Map als Filter dienen, so dass nicht nach jedem DML/DDL Statement jede Anfrage neu evaluiert werden muss, sondern durch den Filter ein Großteil der Anfragen ausgeschlossen werden können. Diese Filter werden zugehörig zu einer Tabelle angelegt, so dass nur Anfragen, die sich auf die entsprechende Tabelle beziehen, überprüft werden müssen.

NiagaraCQ
Das Ziel des Niagaprojektes [12], entwickelt von Chen et al. im Jahr 2000, war es, ein verteiltes Datenbanksystem zum Anfragen von verteilten XML Datenmengen mittels einer Anfragesprache wie XML-QL zu entwickeln. Es soll möglich sein, eine sehr große Menge an *Continuous Queries* zu registrieren und diese untereinander zu gruppieren, ohne die Gruppen bei jeder Neuregistrierung neu zu evaluieren. Eine Annahme ist es, dass viele der registrierten Anfragen sich untereinander sehr ähneln und gruppiert werden können, um sich Berechnungen zu teilen und durch gemeinsame Ausführungspläne I/O Kosten zu sparen, da nicht jede Anfrage separat ausgeführt werden muss. Mit Ausführungsplänen sind hierbei Gruppenpläne gemeint, die von allen Anfragen in der gleichen Gruppe geteilt werden. Diese Pläne sind von den gemeinsamen Teilen der einzelnen Anfragen abgeleitet. Wird eine neue Anfrage registriert, so wird sie der Gruppe zugewiesen, deren Signatur am ähnlichsten mit der der neuen Anfrage ist. Für NiagaraCQ wurde ein eigenes Verfahren implementiert, wie Signaturen angelegt werden. Diese werden mit den Prädikaten der Anfrage erstellt, indem die Konstanten in den Prädikaten mit Platzhaltern ersetzt werden. Das Niagaraprojekt ist auf das Web ausgelegt, in dem sehr viele Anfragen registriert, dynamisch gelöscht und hinzugefügt werden. Daher werden eher unähnliche Gruppenmitglieder toleriert, als das Gruppen nach jedem Mitglied neu aufgeteilt werden, da diese erhebliche Performanzeinbußen mit sich bringen würden.

Management of Streams and Tables
In der Arbeit *One SQL to Rule Them All* [13] beschreiben Begoli et al. SQL-Erweiterungen, so dass das relationale Modell, zusammen mit einer leicht erweiterten SQL-Syntax, effektiv für die Bearbeitung von *streaming data* wird. Zunächst sollen *time-varying relations* eine gängige Grundlage für SQL werden. Diese *time-varying relations* sind Relationen, die sich über die Zeit verändern. Sie können als *Mapping* verstanden werden, die jedem Punkt in der Zeit eine statische Relation zuweisen. Diese Idee ist kompatibel mit bereits bestehenden temporalen Tabellen, die den Zugriff mit dem **AS OF SYSTEM TIME**-Zusatz, der im SQL:2011 Standard [14] definiert wurde, auf einen beliebigen Punkt in der Zeit ermöglichen, und benötigt keine SQL-Erweiterung. Weiterhin wird ein Konzept vorgestellt, um *event time streaming semantics* in SQL zu ermöglichen. Hierzu sollen die Zeilen der verschiedenen Relation um *event timestamps* erweitert werden und davon abhängig bearbeitet werden, unabhängig von der Ankunftsreihenfolge oder der Bearbeitungszeit. Dies ist notwendig, um Daten mit *watermarks* zu versehen. *Watermarks* sind ein Mechanismus bei der Bearbeitung von Datenströmen, um eine zeitliche Spanne für die Vollständigkeit eines *event streams* zu definieren. Diese zeitliche Spanne ist notwendig, um die Vollständigkeit der Input-Daten für temporale Aggregationen zu gewährleisten. Zuletzt wird eine kleine Menge an *materialization controls* hinzugefügt, um die Flexibilität im Verarbeiten der Breite moderner Streamingaufgaben zu gewährleisten. So soll die *stream materialization* durch das Rendern des Anfrageergebnisses zu einem Stream, der Änderungen an der Ergebnisrelation vornimmt, umgesetzt werden. Dieser Stream ist selbst eine *time-varying relation* auf die *streaming SQL* angewandt werden kann. Weiterhin muss ein *materializiation delay* umgesetzt werden, um die Materialisierung unvollständiger und spekulativer Ergebnisse einer *time-varying relation* zu verzögern. Mit dieser Erweiterung können in SQL kontinuierliche Anfragen gebaut werden, die ihre Ergebnismenge periodisch reevaluieren. Diese sind jedoch bisher nur konzeptionell und in keinem SQL-Standard oder System umgesetzt.

Graphstreaming
In der Veröffentlichung *Practice of Streaming Processing of DynamicGraphs: Concepts, Models, and Systems* [15] wird sich unter verschiedenen Vorraussetzungen mit Algorithmen der Graphstreamevaluation auseinandergesetzt. Der Begriff *Graphstreaming* bezieht sich hier meist auf einen dynamischen Graphen, der beispielsweise durch das Einfügen oder Löschen von Knoten modifiziert wird. Es werden verschiedene Frameworks für *graph stream processing* in Bezug auf Performanz und Architektur miteinander verglichen, darunter beispielsweise *Apache Flink* oder *Aspen*. Parameter bei diesen Vergleichen sind beispielsweise das Einfügen, Löschen oder Aktualisieren von Knoten, die Art der Speicherung (aufgeteilt oder zusammen) und der Speicherort (Main memory, GPU oder disk). Weiterhin wird verglichen, wie viele Informationen

zu einem Knoten beziehungsweise einer Kante gespeichert werden können (Typen, Gewichte, Eigenschaften, Timestamps) und wie diese repräsentiert werden. Abschließend wird der Unterschied zwischen Graphdatenbanken und Graphstreamsystemen im Bezug auf verschiedene Parameter, wie beispielsweise Datenverteilung, die Speicherung historischer Daten oder genutzte APIs beziehungsweise Modelle verdeutlicht. Beim *Graphstreaming* stellen Graphänderungen selbst das Datenmodell dar, das heißt, es wird ein *Stream* von Graphänderungen betrachtet. Auf diesem *stream* können dann Analysen durchgeführt werden, beispielsweise mit Hilfe von *pattern matching*. In dem in dieser Arbeit besprochenen Ansatz werden die Änderungen jedoch zunächst umgesetzt und anschließend wird ein persistenter Graph analysiert.

4 Implementierung

Grundsätzlich lässt sich die Implementierung in drei Kategorien unterteilen. Zunächst die Anfrageregistrierung 4.1, bei der relevante Informationen aus der Anfrage extrahiert, Trigger auf genannten Tabellen angelegt und für Folgeschritte notwendige Tabellen in die Datenbank eingespeist werden. Die Begrifflichkeiten *Graph Change Notification* und *Query Result Change Notification* wurden von der Veröffentlichung [2] übernommen. Die zweite Kategorie ist der Prüfmechanismus zur Erkennung von Ergebnismengenänderungen 4.2, bei der die registrierten Anfragen mit stattgefundenen Änderungen an der Datenbank verglichen und Ergebnismengenänderungen festgestellt werden. Der letzte Schritt ist das Erstellen der Benachrichtigung für den Nutzer, abhängig von dem gewählten Benachrichtigungstyp 4.3.

4.1 Anfrageregistrierung

Bei der Anfrageregistrierung wird der Anfrage zunächst eine einzigartige ID zugewiesen, um diese später identifizieren zu können. Diese wird in einer Anfragentabelle, zusammen mit ihrem Text, dem Nutzer, der sie registrierte, und der Benachrichtungsart abgespeichert. Anschließend muss zwischen den beiden Benachrichtigungstypen differenziert werden.

4.1.1 Graph Change Notification (GCN)

Bei der Registrierung einer Anfrage, mit Benachrichtigungstyp GCN, werden zunächst in der Anfrage vorkommende Tabellen extrahiert. Diese Tabellen werden nun in einer weiteren relationalen Tabelle der *AnfrageID* zugewiesen, in der sie vorkommen. Diese Tabelle dient also als Mapping zwischen dem Tabellennamen und der *AnfrageID*. Da bei diesem Benachrichtigungstyp nur von Interesse ist, ob sich etwas an den in der Anfrage vorkommenden Tabellen ändert und nicht ob sich die Ergebnismenge der Anfrage ändert, werden auf allen Tabellen und ihren zugehörigen Eigenschaftstabellen Trigger angelegt. Ein Trigger ist eine Funktion, die vor oder nach Änderungen an einer Tabelle Tätigkeiten vornimmt, wie beispielsweise das Verhindern oder Erlauben von Änderungen an der Tabelle [16], oder wie in diesem Fall, das Einfügen relevanter Daten in eine Notifikationstabelle. Trigger sind immer an eine DML Anweisung gebunden, das heißt, dass jeweils ein Trigger für UPDATE, für DELETE und für INSERT notwendig ist. Wird also eine Änderung an einer Entität vorgenommen, die in der Anfrage erwähnt wurde, so fügt der Trigger für die Benachrichtigung notwendige Informationen in eine dafür angelegte Tabelle ein. Wird nun beispielsweise die oben genannte Anfrage 6 mit Benachrichtigungstyp GCN registriert, so werden auf den Tabellen *Person*, *Person_p*, *Comment*, *Comment_p*, *CommentHasCreatorPerson* und *CommentHasCreatorPerson_p* jeweils drei

Listing 7: Insert-Trigger

```
INSERT INTO notifications
(queryId, tableName, statement, changeTimestamp)
SELECT queryId, 'Comment', 'INSERT', CURRENT_TIMESTAMP
FROM queryTableMap WHERE tableName = 'Comment'
```

Trigger angelegt, vorausgesetzt, diese wurden nicht bereits durch eine vorherige Anfrageregistrierung angelegt. Diese sorgen nun dafür, dass alle AnfrageIDs, die mit dieser Tabelle in Verbindung stehen, zusammen mit dem Namen der Tabelle, der Art der DML-Anweisung und dem Zeitpunkt der Änderung, in eine Tabelle eingefügt werden, über die dann die Benachrichtigung erstellt wird. In Listing 7 ist der **INSERT**-Trigger für die Kommentartabelle beispielhaft definiert. Somit stehen in der Notifikationstabelle die für die Notifikationserstellung notwendigen Informationen.

4.1.2 Query Result Change Notification (QRCN)

Bei der Anfrageregisterung für den Benachrichtigungstyp QRCN sind mehr Schritte notwendig. Hier wird zunächst unterschieden, ob eine Anfrage eine Kante beinhaltet oder sich nur auf Knoten bezient. Zunächst zu Anfragen die keine Kanten beinhalten.

Knotenregistrierung
Sollte keine Kante beinhaltet sein, so kann sich die Anfrage nur auf einen Knoten und dessen Eigenschaftstabelle beziehen, da aktuell nur ein Muster einer Anfrage unterstützt wird. Es werden zunächst die Aliasse aus der Anfrage extrahiert und dem Tabellennamen, mit Hilfe einer Hashmap, zugewiesen. Hierbei gibt es einen Alias für die Knotentabelle selbst und für jede weitere benötigte Eigenschaft einen weiteren Alias, da jede Eigenschaft mit einem eigenen **LEFT JOIN** auf die Eigenschaftstabelle zu der Anfrage hinzugefügt werden muss. Restriktionen in dieser Anfrage können sich entweder auf Atrribute des Knotens selbst, wie beispielsweise Start- und Endzeitstempel oder das Label, oder auf Eigenschaften des Knotens aus der Eigenschaftstabelle beziehen. Im Folgenden wird für Ersteres der Begriff *Knoteneinschränkungen*, und für Letzteres *Eigenschaftseinschränkungen* verwendet. Nun wird die **WHERE**-Klausel betrachtet und am *OR* aufgeteilt, sollte eines in der Klausel vorliegen. Jeder Teil wird anschließend noch einmal am *AND* aufgeteilt, sollte dies vorkommen, so dass jede Einschränkung für sich betrachtet wird. Bei jeder dieser Einschränkungen werden nun die restriktierenden Prädikate dem Alias und der dazugehörigen Tabelle zugeordnet, so dass unterschieden werden kann, ob es sich um eine *Knoten- oder Eigenschaftseinschränkung* handelt. Abhängig

von dieser Entscheidung werden die Prädikate nun entweder einer zu dem Knoten gehörigen *Knoteneinschränkungstabelle* oder in einer zu der Knoteneigenschaftstabelle zugehörigen *Eigenschaftseinschränkungstabelle* gespeichert und bekommen eine ID zugewiesen. Weiterhin wird eine Filtertabelle angelegt, die zu dem in der Anfrage vorkommenden Knoten gehört. Da jede Einschränkung einzeln betrachtet wurde, werden hier noch notwendige Konjunktionen vorgenommen, so dass die Logik der Restriktion wieder stimmt. Jede Zeile entspricht einer Konjunktion der Anfrage und alle Zeilen zusammen der Disjunktion der einzelnen Konjunktionen. Verdeutlicht wird dies nun anhand eines Beispiels.

Listing 8: Beispiel Knotenanfrage

```
SELECT c_p2.pvalue AS length FROM comment c
LEFT JOIN comment_p c_p1 ON c.id = c_p1.elementId
AND c_p1.pkey = 'content'
LEFT JOIN comment_p c_p2 ON c.id = c_p2.elementId
AND c_p2.pkey = 'length'
WHERE c_p1.pvalue <> 'boring' AND c_p2.pvalue < 100
OR c_p1.pvalue = 'cool'
```

Die in Listing 8 definierte Anfrage selektiert also die Länge aller Kommentare, die entweder eine Inhaltsbeschreibung ungleich 'boring' haben und eine Länge kleiner als 100 Zeichen, oder eine Inhaltsbeschreibung gleich 'cool' vorweisen. Wie beschrieben, wird zunächst eine HashMap erstellt, die Aliasse extrahiert und ihnen Tabellennamen zuweist. Diese hat die Form
c → comment
c_p1 → comment_p
c_p2 → comment_p
Der Alias dient also als Schlüssel und der Tabellenname stellt den Wert dar. Die Informationen, welches Alias welchem Tabellennamen zugewiesen werden soll, lässt sich aus den LEFT JOIN Anweisungen der Anfrage ableiten. Hier ist das Muster immer **LEFT JOIN + *Tabellenname* + *Alias***. Anschließend werden allen Aliassen, deren Werte in der HashMap ein *_p* enthalten und sich somit eindeutig auf eine Eigenschaftstabelle beziehen, der jeweilige Eigenschaftsschlüssel zugewiesen. Somit ensteht das Mapping
c_p1 → content
c_p2 → length
Sind die notwendigen Mappings erstellt, so wird die WHERE Klausel betrachtet. Da diese am *OR* aufgeteilt wird, entstehen zwei Teile. Zunächst der erste Teil. Durch das Muster ***alias* + *.pvalue*** ist zu erkennen, dass die Einschränkungen sich auf die Eigenschaften selbst und nicht auf ihre Zeitattribute beziehen. Da bereits ein Alias → Eigenschaftsschlüssel Mapping erstellt wurde, kann der einschränkende Wert dem Schlüssel zugeordnet werden. In diesem Fall entsteht somit

content <> 'boring'
length < 100
Dies stellt die Einschränkung des ersten Disjunktionsteils dar. Die Analyse des zweiten verläuft analog zum Ersten, und liefert
content = 'cool'
Es wurde oben bereits erwähnt, dass es eine zu einer Eigenschaftstabelle zugehörige Eigenschaftseinschränkungstabelle gibt. In diesem Fall gehört diese zur Tablle Comment_p und heißt CommentPredicates. Die durch die Registrierung entstandenen Änderungen an der *commentPredicates*-Tabelle sind in Tabelle 5 dargestellt.

Tabelle 5: CommentPredicates

CommentPredicates			
id	**pkey**	**pvalue**	**tableName**
1	content, length	`<>'boring', <100`	Comment
2	content	`='cool'`	Comment

Es ist zu erkennen, dass Konjunktionsglieder aus einem Disjunktionsteil kommasepariert in eine Zeile geschrieben werden, um die logische Verbindung zu berücksichtigen. Nun werden diese Prädikate genutzt, um die zu dem Kommentarknoten gehörige Filtertabelle, namens Comment_f, dargestellt in Tabelle 6, zu erstellen.

Tabelle 6: Comment_f

Comment_f				
vertexResId	**propPredicatesId**	**queryId**	**tableName**	**selectItems**
null	1	1	Comment	length
null	2	1	Comment	length

In dieser Tabelle werden also die Einschränkungen mit der AnfrageID in Verbindung gestellt. Die Spalte *vertexResId*, die auf die *Knoteneinschränkung* verweist, ist hier immer *null*, da bisher keine *Knoteneinschränkungen* in der Anfrage vorkamen. Wird die **WHERE**-Klausel um eine Knoteneinschränkung erweitert, so wird diese in die Filtertabelle aufgenommen. Beispielhaft wird die folgende Erweiterung des zweiten Disjunktionsteils betrachtet

```
c_p1.pvalue = 'cool' AND c.txStart > '2020-01-18'
```

Hier wird also noch von dem Kommentar verlangt, dass dieser erst nach dem 18. Januar 2020 hinzugefügt wurde. Da das keine *Eigenschaftseinschränkung* darstellt, sondern sich auf die Knotentabelle selbst bezieht, wird diese Einschränkung in der zu der Kommentartabelle zugehörigen *Knoteneinschränkungstabelle* gespeichert. Die Tabellen 7 und 8 verdeutlichen, wie dieser Eintrag gespeichert wird.

Tabelle 7: CommentRestrictions

CommentRestrictions			
id	**restrictionName**	**restrictionValue**	**tableName**
1	txStart	`> '2020-01-18'`	Comment

Tabelle 8: Comment_f

Comment_f				
vertexResId	**propPredicatesId**	**queryId**	**tableName**	**selectItems**
null	1	1	Comment	length
1	2	1	Comment	length

Die Filtertabelle wird nun also auch um die ID der Knoteneinschränkung ergänzt. Da diese konjunkiert mit der Eigenschaftseinschränkung ist, stehen beide IDs in der gleichen Zeile. Nun besteht jedoch noch die Möglichkeit, dass es Zeiteinschränkungen gibt, die sich nicht auf den Knoten selbst, sondern auf die Eigenschaften des Knotens beziehen. Beispielhaft wird nun die erneute Erweiterung des zweiten Disjunktionsteils betrachtet

```
c_p1.pvalue = 'cool' AND c.txStart > '2020-01-18'
AND c_p1.txStart > '2020-01-20'
```

Hier wird also noch von der Inhaltseigenschaft des Knotens verlangt, dass diese erst zwei Tage später, also am 20. Januar 2020, eingefügt wurde. Diese Änderung hat keine Auswirkungen auf die Filtertabelle, aber auf die Eigenschaftseinschränkungstabelle, dargestellt in Tabelle 9.

Tabelle 9: CommentPredicates

CommentPredicates			
id	**pkey**	**pvalue**	**tableName**
1	content, length	`<>'boring', <100`	Comment
2	content	`='cool' AND txStart > '2020-01-20'`	Comment

Hier wurde also die Einschränkung auf dem Wert um das Zeitattribut erweitert. Der Vorteil des Speicherns der Einschränkungen in einer separierten Tabelle und nicht in der Filtertabelle selbst ist, dass diese wesentlich übersichtlicher bleibt und die Wahrscheinlichkeit, dass sich bei vielen registrierten Anfragen Einschränkungen wiederholen, sehr groß ist. Somit müssen diese nicht mehrmals gespeichert werden, sondern es wird überprüft, ob das *pkey-pvalue*-Paar bereits in der zugehörigen Eigenschaftseinschränkungstablle existiert, und die ID wird zurückgegeben. Analog gilt das ebenfalls für die Knoteneinschränkungstabelle.

Listing 9: INSERT-Trigger

```
INSERT INTO potentialChangesVertex
(vertexId, vertexRestrictionsId, properyPredicatesId,
queryId, tableName, statement, changeTimestamp, selectItems
   )
SELECT NEW.id, vertexRestrictionsId, propPredicatesId,
queryId, vertexTableName, 'INSERT', current_timestamp,
selectItems FROM comment_f
```

Zusätzlich zur Extraktion der Prädikate werden Trigger auf den betroffenen Tabellen angelegt. Diese fügen, bei Änderung an betroffenen Tabellen, die ID des geänderten Knotens zusammen mit der Filtertabelle in eine sogenannte *potentialChangesVertex*-Tabelle ein. Hier werden Änderungen gespeichert, die eventuelle Ergebnismengenänderungen verursacht haben und darauf überprüft werden müssen. Der nach Einfügen auf der Kommentartabelle, durch den INSERT-Trigger, für jede eingefügte Zeile, ausgeführte Programmcode, ist in Listing 9 beispielhaft gezeigt. Hier wird also die betroffene Filtertabelle, zusammen mit der Art der DML-Anweisung, in diesem Fall 'INSERT', der ID des eingefügten Knotens und der Zeitpunkt des Einfügens in die *potenialChangesVertex*-Tabelle eingefügt, um dann weiter auf Ergebnismengenänderungen zu prüfen. Das Stichwort *NEW* gibt hierbei an, dass es sich um eine neu eingefügte Zeile handelt. Bei dem DELETE-Trigger würde dieses durch OLD ersetzt werden. Einen UPDATE-Trigger gibt es auf den Knotentabellen selbst nicht, da auf diesen Tabellen nichts aktualisiert wird. Da hier nur das Label, die ID und der Start- und Endzeitstempel gespeichert sind, ist das nicht notwendig. Aktualisierungen finden lediglich auf den Eigenschaftstabellen statt.

Kantenregistrierung

Die Registrierung für Anfragen mit Kanten läuft größtenteils ähnlich ab. Hier muss angemerkt werden, dass für bei zu registrierenden Anfragen nur eine Kante erlaubt ist. Wieso das der Fall ist, wird am Ende des Kapitels 4.1 erklärt. Wie auch bei den Knoten, werden aus der Anfrage zunächst die Aliasse extrahiert und den Tabellen zugewiesen. Hier wird geschaut, welche Tabelle die Kantentabelle ist und die für die Anfrage notwendige Filtertabelle wird für dieser angelegt. Da die Kante selbst nur eine leere Eigenschaftstabelle besitzt, können sich die Prädikate nur auf die Eigenschaftstabellen des Quell- und Zielknotens beziehen. Zeitattribute können jedoch von der Kantentabelle und den beiden Knotentabellen in die Restriktion mit aufgenommen werden. Das Aufteilen der WHERE Klausel erfolgt wie bei Anfragen, die sich nur auf einen Knoten beziehen. Es wird also wieder jede Einschränkung einzeln betrachtet. Anschließend wird die Einschränkung der jeweiligen Tabelle zugeordnet.

Listing 10: Beispiel Kantenanfrage

```
SELECT c_p1.pvalue AS content, p_p1.pvalue AS lastname
FROM comment c
LEFT JOIN comment_p c_p1 ON c.id = c_p1.elementId
AND c_p1.pkey='content'
LEFT JOIN comment_p c_p2 ON c.id = c_p2.elementId
AND c_p2.pkey='length'
JOIN commenthascreatorperson e ON e.source = c.id
JOIN person p ON p.id = e.target
LEFT JOIN person_p p_p1 ON p.id = p_p1.elementId
AND p_p1.pkey='lastname'
LEFT JOIN person_p p_p2 ON p.id = p_p2.elementId
AND p_p2.pkey='firstname'
WHERE p_p2.pvalue='Hans' AND c_p2.pvalue > 100
AND e.txStart > '2020-01-20'
```

Sollte es sich um eine Eigenschaftstabelle handeln, so werden die Prädikate in die zu der Knotentabelle zugehörige Eigenschaftseinschränkungstabelle geschrieben, andernfalls in die zu dem Knoten oder der Kante zugehörige Einschränkungstabelle. Dies wird nun in einem Beispiel verdeutlicht. Bei der in Listing 10 definierten Anfrage werden also Kommentare gesucht, die von einer Person mit dem Vornamen Hans verfasst wurden, eine Länge von mehr als 100 Zeichen besitzen und nach dem 20. Januar 2020 entstanden sind. Tabellarisch wird dann die Inhaltsbeschreibung und der Nachname des Verfassers zurückgegeben. Die Extraktion der Prädikate erfolgt analog zur Knotenregistrierung, so dass die beiden Eigenschaftseinschränkungstabellen 10 und 11 entstehen.

Tabelle 10: CommentPredicates

CommentPredicates			
id	**pkey**	**pvalue**	**tableName**
1	content, length	`<>'boring', <100`	Comment
2	content	`='cool' AND txStart > '2020-01-20'`	Comment
3	length	`> 100`	HasCreator

Hinweis: Prädikat 1 und 2 sind noch von der registrierten Knotenanfrage 8. Die *CommentHasCreatorPerson*-Tabelle wurde hier mit *HasCreator* abgekürzt.
Weiterhin wird noch die Tabelle *CommentHasCreatorPersonRestrictions* 12 erstellt, die die Einschränkung der Kante auf dem Startzeitstempel beinhaltet.

Tabelle 11: PersonPredicates

PersonPredicates			
id	**pkey**	**pvalue**	**tableName**
1	firstname	='Hans'	CommentHasCreatorPerson

Tabelle 12: CommentHasCreatorPersonRestrictions

CommentHasCreatorPersonRestrictions			
id	**restrictionName**	**restrictionValue**	**tableName**
1	txStart	>'2020-01-20'	CommentHasCreatorPerson

Sind die jeweiligen Einschränkungstabellen erstellt, werden die notwendigen Konjunktionen wieder in die zu der Kante zugehörigen Filtertabelle 13 eingetragen.

Tabelle 13: CommentHasCreatorPerson_f

CommentHasCreatorPerson_f					
persPredId	**comPredId**	**queryId**	**persResId**	**comResId**	**edgeResId**
1	3	2	null	null	1

Die *persResId* und *comResId* sind hier wieder *null*, da keine Einschränkung auf den Knotentabellen selbst, sondern nur auf deren Eigenschaftstabellen vorliegt. Die *queryId* beträgt 2, da es die zweite bisher registrierte Anfrage ist. In der Applikation besitzt diese Tabelle noch jeweils eine Spalte für den Namen des Quell- beziehungsweise Zielknotens, in diesem Fall also *Comment* und *Person*. Diese beiden Spalten wurden hier jedoch aus Platzgründen weggelassen. Die Trigger werden bei Kantenanfragen auf der Kantentabelle angelegt. Diese fügen dann, ähnlich wie die Trigger auf den Knotentabellen, die zugehörige Kantenfiltertabelle, zusammen mit der geänderten *KantenID* und den betroffenen IDs der Quell- und Zielknoten, in eine*potentialChanesEdge*-Tabelle ein, wo die Änderungen dann auf tatsächliche Ergebnismengenänderungen überprüft werden können. Steht nun mehr als eine Kante in der Anfrage, so wird in beiden Kantenfiltertabellen die Restriktionen gespeichert. Diese müssten jedoch dann wieder in Verbindung gebracht werden, und es würde eine dritte Tabelle benötigt werden, die die Konjunktionen der beiden einzelnen Kantenfiltertabellen speichert. Für jede weitere Kante müsste dann eine neue Tabelle erstellt werden. Dies ist sehr speicheraufwändig und wurde aus Zeitgründen in diesem Projekt nicht umgesetzt.

4.2 Prüfmechanismus für Ergebnismengenänderungen

Die beiden *potentialChanges*-Tabellen für Kanten und Knoten werden periodisch von einem Java Thread auf neue Einträge überprüft. Abhängig davon, welche Tabelle Einträge vorweist, wird dann entsprechend gehandelt.
Bei Einträgen in der *potentialChangesVertex*-Tabelle werden sich zunächst die Schlüssel und Werte der restriktierenden Eigenschaften oder die der Knotenattribute aus der jeweiligen Tabelle geholt. Anschließend wird für die KnotenID, an der eine Änderung stattgefunden hat, eine HashMap erstellt, deren Schlüssel die ID ist und dessen Wert eine Liste an bereits für diese ID überprüften PrädikatIDs oder KnoteneinschränkungsIDs darstellt. Dies kostet zwar Speicherkapazität, jedoch müssen Prädikate für einen Knoten nicht mehrmals überprüft werden. Dies bringt vor allem bei häufigen Änderungen am gleichen Knoten Verbesserungen der Performanz. Nun muss unterschieden werden, ob es sich um eine Löschung oder Neueinfügung eines Knotens handelt oder um eine Änderung an den Eigenschaften eines Knotens. Der Prüfmechanismus für Änderung am Knoten selbst, und nicht an der Eigenschaftstabelle, wird mit folgendem Beispiel verdeutlicht. Angenommen es existiert ein Knoten mit der $ID = 1$, und den Eigenschaften, dargestellt in Tabelle 14 .

Tabelle 14: Comment_p

Comment_p				
elementId	**pkey**	**pvalue**	**txStart**	**txEnd**
1	content	nice	2020-01-15	38-01-19
1	length	150	2020-01-15	38-01-19

Nun wird dieser mit Hilfe folgender DELETE-Anweisung gelöscht.

```
DELETE FROM comment WHERE id=1
```

Aufgrund der **CASCADE** Beziehung unter Comment und Comment_p werden also auch alle Einträge mit der *elementId* = 1 aus der Comment_p Tabelle gelöscht. Durch diese Änderung wird ein Eintrag in die *potentialChangesVertexTabelle* 15 eingefügt.

Tabelle 15: potentialChangesVertex

potentialChangesVertex						
verId	**verResId**	**proPredId**	**queryId**	**tabName**	**stat**	**changeTime**
1	null	1	1	Comment	DELETE	2021-04-15
1	1	2	1	Comment	DELETE	2021-04-15

Diese Tabelle stellt also die Filtertabelle von *Comment_f* dar, erweitert um die ID des gelöschten Knotens. Nun werden Anfragen an den gelöschten Knoten gestellt, um

Listing 11: Prüfanfrage nach Knotenlöschung

```
SELECT foo.count(*) FROM (
SELECT elementId, pkey, pvalue FROM comment_p c_p1
FOR SYSTEM_TIME ALL
WHERE c_p1.elementId = 1 AND c_p1.pkey = 'content'
AND c_p1.pvalue <> 'boring' AND c_p1.txEnd BETWEEN
'2021-04-05' + INTERVALL 1000 MICROSECOND
AND '2021-04-05' - INTERVALL 1000 MICROSECOND
UNION SELECT elementId, pkey, pvalue FROM comment_p c_p1
FOR SYSTEM_TIME ALL WHERE c_p1.elementId = 1
AND c_p1.pkey='length' AND c_p1.pvalue < 100
AND c_p1.txEnd BETWEEN '2021-04-15' + INTERVALL 1000
   MICROSECOND
AND '2021-04-15' - INTERVALL 1000 MICROSECOND)
foo
```

zu überprüfen, ob die in der Filtertabelle vermerkten Prädikate von diesem erfüllt werden. Für die erste Zeile der Tabelle 15 würde die Eigenschaftseinschränkung
content, length || <>'boring', < 100
gefunden werden. Mit der in Listing 11 definierten Anfrage wird nun überprüft, ob eine Ergebnismengenänderung stattgefunden hat. Hier muss dazu gesagt werden, dass die Zeitstempel in der Applikation selbst bis auf die tausendstel Sekunde genau sind. Aus Platzgründen wurde der Zeitstempel gekürzt. Das der Zeitstempel in einem Intervall gesucht wird, dient dazu, dass eine kleine Toleranz gegeben wird, sollte der Prozess etwas langsam laufen oder die Datenbank gerade stark ausgelastet sein. Diese Anfrage 11 gibt die Anzahl an Reihen zurück, die von den Prädikaten getroffen wurden. Stimmt die zurückgegebene Anzahl mit der Anzahl an Schlüsseln, in diesem Fall zwei, überein, so kann gesagt werden, dass der gelöschte Kommentar die Anforderungen erfüllt. Ist dies der Fall, so wird die getroffene *PrädikatenID* in den HashMap-Eintrag der *KnotenID* mit aufgenommen und für diesen Knoten muss diese *PrädikatenID*, solange seine Eigenschaften nicht geändert werden, nicht mehr überprüft werden. Der **FOR SYSTEM_TIME ALL**-Zusatz gibt hier an, dass nicht nur aktuelle Reihen zurückgegeben werden sollen, also Reihen deren Endzeitstempel gleich '38-01-19' ist, sondern auch bereits gelöschte Reihen. Das ist notwendig, da der Knoten bereits gelöscht wurde und somit in der aktuellen Sicht auf die Datenbank nicht mehr angezeigt wird. Der zweite Eintrag aus der *potentialChangesVertex*-Tabelle wird nicht mehr überprüft, da bereits ein Disjunktionsteil getroffen hat und somit eine Ergebnismengenänderung feststeht. Es werden also alle Eintrage mit *stat = 'DELETE', queryId=1, verId = 1, changeTimestamp = '2021-04-15'* aus

der *potentialChangesVertex*-Tabelle gelöscht. Da die EigenschaftsID nun in der Liste der betroffenen KantenID steht, wird eine Notfikation in die Notifikationstabelle geschrieben. Angenommen der erste Eintrag hätte nicht getroffen, so würde beim zweiten Eintrag die Knoteneinschränkung und Eigenschaftseinschränkung separat betrachtet werden. Auch für Knoteneinschränkungen wird eine HashMap angelegt, die als Schlüssel ebenfalls die KantenID hat und deren Wert eine Liste der getroffenen KnoteneinschränkungsIDs darstellt. Für die Knoteneinschränkung wird die Knotentabelle selbst angefragt und auch hier werden die Reihen gezählt, ob diese Anzahl an Reihen mit der Anzahl an Schlüsseln übereinstimmt. Ist dies der Fall, so wird die KnoteneinschränkungsID in die Liste der KantenID aufgenommen. Zuletzt wird nun überprüft, ob die EigenschaftseinschränkungsID und die KnoteneinschränkungsID in den zu der KantenID zugehörigen Liste vorkommt. Ist dies der Fall, so wird eine Benachrichtigung in die Notifikationstabelle eingefügt. Liegt keine Änderung an der Knotentabelle selbst vor, sondern an der Eigenschaftstabelle des Knotens, muss anders gehandelt werden. Hier muss bei **INSERT** überprüft werden, ob die neue Eigenschaft Prädikate des Knotens trifft und keine anderen Prädikate des Knotens vorher trafen. Wäre dies der Fall, so würde zwar ein Prädikat mehr treffen, es hätte jedoch keine Ergebnismengenänderung zur Folge, da ein weiterer erfüllter Disjunktionsteil nichts am Ergebnis der WHERE Klausel ändern würde. Anders wäre es jedoch, wenn die eingefügte Eigenschaft ein SELECT-Item darstellt. Sollte das eintreffen, so muss überprüft werden, ob der Knoten ein beliebiges anderes Prädikat erfüllt. Trifft dies ein, so hat eine Ergebnismengenänderung stattgefunden, da ein neuer Wert zurückgegeben wird. Bei **DELETE** ist es ähnlich. Hier muss überprüft werden ob die gelöschte Eigenschaft ein Prädikat erfüllte und ob diese Eigenschaft das einzige Prädikat erfüllte. Sollte es noch andere getroffene Prädikate geben, so wäre nur ein Disjunktionsteil falsch, was jedoch nichts am Ergebnis der WHERE Klausel ändert. Auch hier wäre es anders, wenn die Eigenschaft ein SELECT-Item wäre. Trifft hier ein beliebiges Prädikat, dann liegt eine Ergebnismengenänderung vor, da nun ein Wert nicht mehr zurückgegeben wird. Bei einer **UPDATE** Anweisung muss überprüft werden, ob die alte Eigenschaft ein Prädikat traf und die neue es nicht mehr trifft beziehungsweise, ob die alte Eigenschaft ein Prädikat nicht traf, die neue es jedoch trifft. Es muss also ein Trefferunterschied zwischen alter und neuer Eigenschaft vorliegen, da sich nur dann der Wahrheitswert eines Disjuntionsterms ändert. Weiterhin muss es das einzige Prädikat sein, das trifft, andernfalls würde sich das Ergebnis der WHERE Klausel auch nicht ändern. Auch hier ist es, wie bei den anderen beiden DML-Anweisungen auch, etwas anderes, wenn die Eigenschaft ein SELECT-Item ist. Ist das der Fall, so muss auch hier nur ein beliebiges Prädikat für den Knoten stimmen, so dass eine Ergebnismengenänderung vorliegt. Dieses Prozedere wird nun mit der in Listing 12 definierten DML-Anweisung verdeutlicht Hier wird also die Inhaltsbeschreibung des Kommentars mit der *Id = 1* auf 'boring'

Listing 12: Update-Anweisung

```
UPDATE comment_p SET pvalue='boring'
WHERE elementId=1 AND pkey='content
```

Listing 13: Prüfanfrage alte Eigenschaft

```
SELECT foo.count(*) FROM (
SELECT elementId, pkey, pvalue FROM comment_p
FOR SYSTEM TIME ALL WHERE elementId=1
AND pkey='content' AND pvalue<>'boring' AND
txEnd BETWEEN '2021-04-15' + INTERVALL 1000 MICROSECOND
AND '2021-04-15' - INTERVALL 1000 MICROSECOND
UNION SELECT elementId, pkey, pvalue FROM comment_p
FOR SYSTEM TIME ALL WHERE elementId=1 AND
pkey='length' AND pvalue<100 AND txEnd='38-01-19')
foo
```

gesetzt. Um die alte Eigenschaft abzuspeichern, wurde hierfür die *potentialChangesVertex* -Tabelle um eine weitere Spalte namens *propertyKey* erweitert. In dieser steht nun der Schlüssel der aktualisierten Eigenschaft. Zunächst wird überprüft, ob der Eigenschaftsschlüssel, in diesem Fall 'content', überhaupt in einem Prädikat vorkommt oder Teil der SELECT-Items ist. Ist dies nicht der Fall, so steht bereits fest, dass keine Ergebnismengenänderung stattgefunden hat. Andernfalls müssen nun zwei Anfragen gebaut werden. Die erste Anfrage überprüft, ob die Eigenschaft vor der Aktualisierung ein Prädikat traf, und die zweite, ob die neue Eigenschaft es auch trifft. Es wird hier angenommen, dass der Änderungszeitstempel gleich '2021-04-15' ist. Mit der in Listing 13 definierten Anfrage wird überprüft, ob die alte Eigenschaft des Knotens das Prädikat erfüllt hat. Hier muss darauf geachtet werden, dass die Eigenschaft mit dem aktualiserten Eingenschaftsschlüssel in der Vergangenheit gesucht wird und die nicht aktualisierte Eigenschaft in den aktuellen Daten. Um die beiden Zeilen jedoch zu vereinigen, muss bei beiden der Zusatz **FOR SYSTEM_TIME ALL** angegeben werden. Da die Inhaltsbeschreibung des Kommentars vor der Aktualisierung gleich 'nice' war, treffen beide Anfragen und es werden zwei Zeilen zurückgegeben. Somit ist die Anzahl der Schlüssel gleich der Anzahl der zurückgegebenen Zeilen, was bedeutet, dass das Prädikat zu wahr ausgewertet wird. Die in Listing 14 definierte Anfrage überprüft nun die neue Eigenschaft. Bei der Anfrage für die aktualisierte Eigenschaft wird nun in den aktuellen Daten gesucht. Da der neue Eigenschaftswert gleich 'boring' ist, trifft der erste Teil der Anfrage nicht, und es wird nur eine Zeile zurückgegeben. Da die Anzahl zurückgegebener

Listing 14: Prüfanfrage neue Eigenschaft

```
SELECT foo.count(*) FROM (
SELECT elementId, pkey, pvalue FROM comment_p WHERE
elementId=1 AND pkey='content' AND pvalue<>'boring'
UNION SELECT elementId, pkey, pvalue FROM comment_p
WHERE elementId=1 AND pjey='length' AND pvalue<100)
foo
```

Zeilen nun ungleich der Anzahl an Schlüsseln ist, trifft das Prädikat den aktuellen Knoten nicht mehr. Somit liegt ein Unterschied im Wahrheitswert des Disjunktionsterms, vor und nach der Aktualisierung der Eigenschaft, vor. Nun muss noch überprüft werden, ob das andere Prädikat ebenfalls trifft. Wäre dies der Fall, so würde keine Ergebnismengenänderung vorliegen, da sich am Wahrheitswert der WHERE Klausel nichts geändert hätte. Da die Inhaltsbeschreibung des Kommentars jedoch ungleich 'cool' ist, trifft dieses Prädikat nicht, die EigenschaftseinschränkungsID 1 wird aus der Liste für diese KnotenID gelöscht, und es wird eine Benachrichtigung in die Notifikationstabelle eingefügt. Wäre statt des Schlüssels 'content' der Schlüssel 'length' auf einen Wert aktualisiert worden, der immer noch größer als 100 ist, läge auch eine Ergebnismengenänderung vor. Es hätte sich zwar kein Wahrheitswert eines Prädikates geändert, jedoch trifft das erste Prädikat immer noch und der Eigenschaftsschlüssel 'length' ist ein SELECT-Item, was eine Ergebnismengenänderung nach sich gezogen hätte. Knoten- und Eigenschaftsänderungen können auch Auswirkungen auf Kantenanfragen haben. Dies wurde in diesem Projekt leider nicht umgesetzt. Warum genau wird am Ende dieses Abschnittes erklärt. Der Prüfmechanismus für Kantenanfragen funktioniert ähnlich zu dem für Knotenanfragen. An Kanten können nur Löschungen oder Neueinfügungen stattfinden, da die Kante keine Eigenschaft besitzt, die aktualisiert werden können. Angenommen, es existiert eine Kante mit *ID* = 1, die einen Personenknoten, mit einer Eigenschaft 'firstname' = 'Hans', und einen Kommentarknoten, der eine Länge von mehr als 100 Zeichen vorweist, verbindet. Die KnotenID des Personenknotens soll 1 sein, und die des Kommentarknotens ebenfalls. Nun wird diese Kante mit der DML-Anweisung

```
DELETE FROM commenthascreatorperson WHERE id = 1
```

gelöscht. Durch den Trigger auf der Kantentabelle wird nun ein Eintrag in die *potentialChangesEdge*-Tabelle 16 eingefügt Durch die hohe Spaltenanzahl wurde die Tabelle aus Lesbarkeitsgründen in zwei Teile aufgeteilt, und es wurden Abkürzungen verwendet. **SV** steht für SourceVertex, **TV** für TargetVertex, **E** für Edge, **PId** für PredicatesId und **ResId** für RestrictionId. Nun wird jedes dieser Prädikate separat überprüft und, sollte es treffen, entweder der Liste der KantenID hinzugefügt, falls

Tabelle 16: PotentialChangesEdge

potentialChangesEdge						
SVId	**TVId**	**SVPId**	**TVPId**	**SVName**	**TVName**	**queryId ...**
1	1	3	1	Comment	Person	2 ...

... SVResId	**TVResId**	**ERestId**	**tabName**	**stat**	**EId**	**chaTime**
... null	null	1	comHasCreaPer	DELETE	1	2021-04-15

es sich um die Kantenrestriktion handelt, oder die der Knoten, sollte es sich um eine Knoten- oder Eigenschaftseinschränkung handeln. Die Überprüfung selbst erfolgt wie bei den Knoten. Es werden Anfragen gebaut und zurückgegebene Zeilen gezählt. Sollten diese mit der Anzahl an Schlüsseln übereinstimmen, so liegt ein Treffer vor. Ist am Ende der Überprüfung jede EinschränkungsID in der entsprechenden Liste, so wird eine Benachrichtigung in die Notifikationstabelle eingefügt. Andernfalls werden alle Einträge in der *potentialChangesEdge*-Tabelle mit der *edgeId=1, sourceVertexId = 1, targetVertexId = 1, changeTimestamp = '2021-04-15', statement='DELETE'* gelöscht.

Einschränkungen

Das Umsetzen der Auswirkungen von Knotenlöschungen auf Kantenanfragen gestaltete sich deshalb schwierig, da die **ON DELETE CASCADE** Regel, die anschließend auf die Kantentabelle angewandt wurde, keine Trigger auslöst [17]. Daher hätte die vorgegebene Architektur geändert werden müssen, so dass diese Löschregeln von der Anwendungsapplikation übernommen und somit die Trigger ausgelöst werden. Das Verlagern der Operationen von der Datenbankseite auf die Applikationsseite bringt jedoch erhebliche Performanzeinbußen mit sich. Eine Idee ist es, die Löschregeln zu entfernen und einen Trigger diese ausführen zu lassen, da dieser auch die Trigger der anderen Tabellen startet. Dies ist jedoch daran gescheitert, dass dynamisches SQL in Triggern nicht erlaubt ist. Dieses Problem in Kombination mit der fortschreitenden Zeit hat dazu geführt, dass diese Funktionalität leider nicht mehr umgesetzt werden konnte. Unter anderem an dem Verbot von dynamischen SQL in Triggern ist die Funktionalität der Ergebnismengenänderungserkennung, nach Eigenschaftsänderungen an Knoten, von Kantenanfragen ebenfalls gescheitert. Hier müsste nach Änderungen an der entsprechenden Eigenschaftstabelle ein Trigger überprüfen, in welchen Kantentabellen der Quell- oder Zielknoten gleich der zu der Eigenschaftstabelle zugehörigen Knotentabelle ist. Da diese Anfrage abhängig vom Namen der Eigenschaftstabelle ist, müsste der Trigger hier eine Anfrage dynamisch erstellen. Eine Lösungsidee ist das Erstellen eines Triggers, der nur die Art der DML-Anweisung, die geänderte *KnotenID* und die Eigenschaftstabelle, an der die Änderungen stattfanden, in eine Tabelle einfügt. Diese Daten werden sich anschließend von der Applikation aus der

Tabelle geholt und es wird applikationsseitig überprüft, welche Kanten die zu der Eigenschaftstabelle zugehörige Knotentabelle beinhalten. Ist dies geschehen, so muss entschieden werden, ob diese Knotentabelle Quell- oder Zieltabelle betroffener Kantentabellen ist, um die richtigen Prädikate zu überprüfen. Nun können die Prädikate des entsprechenden Knotens überprüft werden, und es kann entschieden werden, ob die Eigenschaftsänderung Auswirkungen auf den Wahrheitswert der Prädikate hat. Das Problem hierbei ist jedoch, dass der Trigger der Eigenschaftstabelle keinerlei Informationen über betroffene *KantenIDs* hat. Wie auch oben bereits erklärt, findet eine Ergebnismengenänderung nur statt, sollte die Eigenschaft Teil der **SELECT**-Items sein oder sollte sie Konjunktionsteil der einzigen Filterzeile sein, die dazu führt, dass die Kantenanfrage eine spezielle Kante trifft. Sollte sie nicht die einzige Filterzeile sein, so würde sie keine Auswirkung auf den Wahrheitswert der **WHERE**-Klausel haben und zu keiner Ergebnismengenänderung führen, da sie nur ein weiterer wahr ausgewerteter Disjunktionsteil wäre. Es müssen also alle betroffenen *KantenIDs* gesucht werden, die diesen Knoten als Quell- oder Zielknoten vorweisen. Anschließend müssen alle Filterzeilen aus der zu der Kantentabelle zugehörigen Filtertabelle überprüft werden. Sollten nun die Filterzeilen als einziges treffen, die das Prädikat beinhalten, das die aktualisierte Eigenschaft beinhaltet, so kann gesagt werden, dass eine Ergebnismengenänderung vorliegt. Sollte eine beliebige andere Zeile der Filtertabelle treffen, die das Prädikat mit der aktualisierten Eigenschaft nicht beinhaltet, so liegt keine Änderung vor. Dieses Prozedere konnte aufgrund der fortschreitenden Zeit ebenfalls nicht implementiert werden.

4.3 Erstellen der Notifikation

Zunächst wird ein sogenannter *BrokerService* gestartet, auf dem eine *ActiveMQ-ConnectionFactory* angelegt wird. Über diese findet die Kommunikation zwischen zwei Java Threads, dem *NotificationCreator* und dem *NotificationsConsumer*, statt. Anschließend wird beim Start des Programms für jeden Nutzer eine dynamischen Queue angelegt. Dynamisch bedeuet, dass die Queues nicht bereits vorm Start des Programms feststehen müssen, sondern mit einem *createQueues*-Befehl dynamisch angelegt werden. Dies ist sehr nützlich, da nur ein *Broker* und wenige *Clients* existieren. Es kann jedoch bei einer größeren Anzahl an *Brokern* und *Clients* zu Problemen führen, sollte sich ein Nutzer mit dem falschen *Broker* verbinden oder ein Queue-Name falsch geschrieben werden. In diesem Fall werden fälschlicherweise weitere Queues angelegt, was zu Kommunikationsproblemen führt [18]. Sind die Queues angelegt, so liest ein Java Thread, der *NotificationsCreator*, die in die Notifikationstabelle eingefügten Zeilen. Da dort die *AnfrageID* gespeichert ist, kann über diese der Nutzer herausgefunden werden, der die Anfrage registrierte. Da jeder Nutzer eine eigene Queue besitzt, deren Name gleich dem Nutzernamen ist, kann nun die Benachrichtigung über diese Queue an den Nutzer gesendet werden. Dies geschieht, indem eine

TextMessage erstellt wird, deren Inhalt abhängig von der Art der Benachrichtigung ist. Eine Nachricht vom Benachrichtigungstyp GCN hat folgende Syntax

```
Graph Change Notification for the following query: <PGQLText>
Caused by <Statement> on Table <TableName>.
Time of change: <changeTimestamp> for user <Username>
```

Benachrichtigungen vom Typ QRCN haben eine ähnliche Syntax

```
Query Result Change Notification. The resultset of
the following query changed: <PGQLText>
Caused by <statement> on Table <TableName>.
Time of change: <changeTimestamp> for user <Username>
```

Der *NotificationConsumer*-Thread überprüft die Queue des aktuell angemeldeten Nutzers periodisch. Dies geschieht über einen *ContextLookup*. Die dynamischen Queues und deren Inhalt werden in einer JNDI Datei gespeichert [19]. Diese können dann über den *ContextLookup* nachgeschlagen und der Inhalt beispielsweise auf der Konsole ausgegeben werden.

5 Analyse des Verfahrens

5.1 Ergebnismengenänderungserkennung

Der Algorithmus wurde auf einem, von der Universität Leipzig zur Verfügung gestellten, Server getestet. Dieser Server besitzt acht Prozessoren mit jeweils 4 Kernen und einer Taktrate von 3,7 GHz, 16 GB DDR3 RAM und als Betriebssystem Debian GNU/Linux 10. Die genutzte MariaDB-Version ist 10.5.11. Variiert wurde zwischen der Anzahl der registrierten Anfragen, der Frequenz der Änderungen an der Datenbank, der Art der DML-Anweisungen (DELETE/UPDATE/INSERT) und der Art der registrierten Anfragen (Kanten-bzw. Knotenanfragen).

Knotenanfragen
Zunächst wurden 1014 Knotenanfragen mit gleicher Syntax registriert.

Listing 15: Syntax registrierte Knotenanfragen

```
SELECT c_p1.pvalue AS content FROM comment c
LEFT JOIN comment_p c_p1 ON c.id = c_p1.elementId
  AND c_p1.pkey = 'content'
LEFT JOIN comment_p c_p2
  ON c.id = c_p2.elementId AND c_p2.pkey = 'length'
WHERE c_p2.pvalue > #i AND c_p1.pvalue LIKE '#c%
```

Das '#' Symbol signalisiert in der Anfrage 15, dass es sich um einen Parameter handelt. Der Parameter **i** lag im Interval [0,39) und der Parameter **c** nahm einmal jeden Buchstaben im Alphabet an. Dies ergibt 39*26 = 1014 Anfragen. Nach der Registrierung wurden, zunächst mit einer Frequenz von 1 Löschung pro 2000ms, Knoten entfernt. Es wurde zuerst die Anzahl der Anfragen, deren Ergebnismengen von der Änderungsanweisung betroffen waren, in Relation zu der Zeit gesetzt, und anschließend die Anzahl der überprüften Zeilen, die in die *potentialChangesVertex*- bzw. *potentialChangesEdge*-Tabelle eingefügt wurden, in Relation zur Zeit. Um eine gute Vergleichsbasis zu bieten, wurde bei Löschanweisungen immer nach 1000 gelöschten Knoten, beziehungsweise Kanten, abgebrochen. Auf Abbildung 7 ist zu erkennen, dass die Anzahl der getroffenen Anfragen, deren Ergebnismenge sich änderte, Auswirkungen auf die Dauer der Überprüfung hat. Desto mehr getroffene Anfragen, desto länger dauert die Überprüfung. Dies folgt daher, dass für getroffene Anfragen Benachrichtigungen zunächst in Form einer Liste erstellt werden und die anschließend noch in die Notifikationstabelle eingetragen werden müssen. Weiterhin lässt sich aus der Abbildung schließen, dass während dieses Durchlaufs alle gelöschten Kommentare eine Länge von unter 20 oder über 38 hatten. Die Restriktion auf den Anfangsbuchstaben des Inhalts ist hier nur da, um die Anzahl der Anfragen zu erhöhen und einen weiteren Parameter hinzuzufügen. Da jedoch jeder Inhalt

mit einem Buchstaben aus dem Alphabet beginnt, ist die Länge die eigentliche Einschränkung. Auf Abbildung 8 ist zu erkennen, dass immer nur die Anzahl an registrierten Anfragen auch überprüft wird. Dies ist dann nicht der Fall, wenn eine Überprüfung länger dauert, als die Zeitspanne zwischen zwei Änderungen, in diesem Fall länger als zwei Sekunden. Dies würde dazu führen, dass die Filtertabelle zweimal in die *potentialChangesVertex*-Tabelle eingefügt wird und somit die doppelte Anzahl an Reihen überprüft werden muss. Das Evaluationsergebnis nach Vervierfachung der Frequenz ist in den Abbildungen 9 und 10 visualisiert. In der Abbildung 10 ist gut erkennbar, dass die Überprüfung der Änderungen gelegentlich länger benötigt, als die Zeitspanne zwischen zwei Änderungen. Dies führt dazu, dass gegen Ende über 3000 Zeilen in einem Durchlauf überprüft werden müssen. Finden die Änderungen weiter so hochfrequent statt, so wird sich die Anzahl zu überprüfender Zeilen pro Durchlauf weiter erhöhen. Dies führt dazu, dass das Programm immer langsamer wird, bis die Anzahl an Reihen zu groß ist und eine Java Heap Space Exception geworfen wird. Dies dauert bei nur 1014 registrierten Anfragen jedoch sehr lang. Damit lässt sich auch erklären, weshalb die Abbildung 9, die die Anzahl der Treffer in Relation zur Zeit setzt, so unterschiedlich zu der des vorherigen Durchlaufs (Abbildung 7) ist. Wenn davon ausgegangen wird, dass die gelöschten Kommentare hier ähnliche Längen haben, wie die des vorherigen Durchlaufs, also hautpsächlich unter 20 und über 40, dann sind die Anzahl der Treffer nur Vielfache von den Treffern des ersten Durchlaufs. Vielfache deswegen, da, durch die höhere Änderungsfrequenz, nun mehrere Knotenlöschung auf einmal betrachtet und die Treffer zusammengefasst werden. Dementsprechend wurden die Treffer hier mit dem Faktor 2 und 3 multipliziert, da in Abbildung 10 abgebildet ist, dass maximal knapp über 3000 Reihen, also drei Knotenlöschungen auf einmal, überprüft werden. Jetzt wird die Anzahl an Anfragen etwas mehr als verdoppelt, indem das Intervall von **i** bis auf inklusive 95 erweitert wird. Das ergibt 26*96 = 2496 Anfragen. Die Art der Anfragen bleibt gleich und die beiden Löschfrequenzen ebenfalls. Die Messungen für die Frequenz von 1 Löschung pro 2000ms sind in den Diagrammen 11 und 12 dargestellt. Die beiden Abbildungen (11, 12) ähneln sehr denen des Durchlaufs mit 1000 Anfragen und 2000ms Änderungsperiode (7, 8). Auch hier war die Länge der Knoten immer kleiner als 20 oder größer als 70. Der Einfluss der Treffer wird auf Grafik 11 jedoch etwas deutlicher, da hier auf der rechten Seite (bei Trefferanzahl größer als 70) noch differenziert wird, da die Punkte nicht alle die gleiche x-Koordinate haben, wie es in Abbildung 7 der Fall war. Abbildung 12 zeigt hier wieder nur an, dass die Zeitspanne zwischen zwei Änderungen nie größer war, als die Dauer der Überprüfung. Somit wurden immer nur genau 2496 Zeilen überprüft. Die Evaluationsergebnisse nach Vervierfachung der Löschfrequenz sind in den Abbildungen 13 und 14 visualisiert. Auf Abbildung 14 ist zu erkennen, dass nur zu Beginn die 2496 Zeilen überprüft werden. Da jede Überprüfung länger als eine halbe Sekunde dauerte, vervielfacht sich die Menge zu überprüfender Zeilen

mit jeder weiteren Löschung. Bereits nach wenigen Überprüfungen ist die Anzahl der Zeilen auf über eine halbe Millionen gestiegen. Daraus kann geschlossen werden, dass bei über tausend registrierten Anfragen die Änderungsperiode nicht länger weit unter 2 Sekunden sein darf, da sonst die Anzahl der zu überprüfenden Zeilen zu groß wird und das Programm abstürzt. Um auch andere Arten der DML-Anweisungen zu berücksichtigen, wurde nun die Eigenschaftstabelle eines Knotens aktualisiert. Hierzu wurden wieder 2496 Knotenanfragen registriert und ein Knoten ausgewählt, auf dem zunächst die Längeneigenschaft aktualisiert, dann gelöscht und anschließend wieder eingefügt wurde. Ursprünglich hatte der Knoten eine Länge von 3, diese wurde dann auf 80 aktualisiert, daraufhin gelöscht und anschließend wurde die Eigenschaft mit der Länge 3 wieder eingefügt. Die entstandenen Messungen sind in den Diagrammen 15 und 16 gezeigt. Die Werte bei knapp unter 80 Treffern kamen vom Aktualisieren der Längeneigenschaft. Hier waren immer genau 77 Anfragen betroffen. Da die Länge auf 3 gesetzt war, änderte sich an der Ergebnismenge der ersten drei Anfragen nichts. Lediglich die Knotenanfragen, die eine Längenrestriktion von > 3 bis > 79 hatten, nehmen nun den Knoten mit der aktualisierten Längeneigenschaft in ihre Ergebnismenge auf. Bei der anschließenden Löschung waren 80 Knotenanfragen betroffen, da die Längeneigenschaft entfernt wurde und somit alle Knoten, die die Restriktion bisher erfüllten, nicht mehr Teil der Erbebnismenge sind. Nach der Löschung wird die Längeneigenschaft mit einem Wert von 3 wieder eingefügt. Somit erfüllt der Knoten die Einschränkungen der ersten 3 Anfragen wieder und wird in deren Ergebnismenge aufgenommen. Durch Abbildung 15 wird deutlich, dass die Aktualisierungsanweisung am längsten Zeit benötigt. Das hat den Grund, dass für Aktualisierungsanweisungen zwei Anfragen an die Datenbank gestellt werden müssen. Zunächst muss überprüft werden, ob die Eigenschaft vor der Änderung essentiell für die Erfüllung der Restriktionen einer Anfrage war und anschließend, ob die Änderung dieser Eigenschaft dazu führte, dass sich der Wahrheitswert der **WHERE**-Klausel einer Anfrage änderte. Sollte die aktualisierte Eigenschaft jedoch Teil der ***SELECT***-*Items* einer Anfrage sein, deren **WHERE**-Klausel bereits erfüllt ist, so steht sofort fest, dass eine Ergebnismengenänderung vorliegt. Am wenigsten Zeit benötigten die Überprüfungen der Einfügeoperationen. Das hat die Ursache, dass die Eigenschaft nicht in den historischen, sondern in den aktuellen Daten gesucht werden muss. Die Überprüfzeiten von über 1000ms bei der Überprüfung der Einfügeoperationen entstanden vermutlich dadurch, dass die Überprüfung der Aktualisierungsoperation den Beginn der Überprüfung der Einfügeoperation leicht verzögerte.

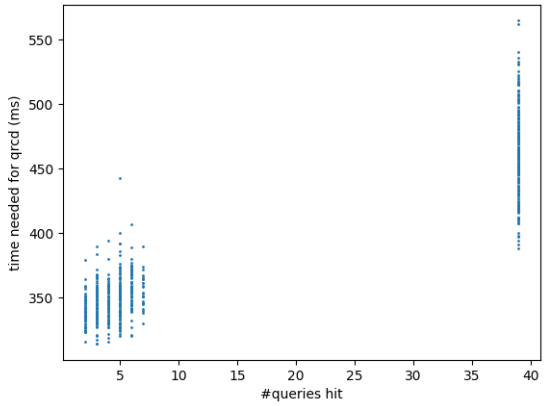

Abbildung 7: #Treffer-Zeit-Relation 1014 Knotenanfragen bei 1 Löschung pro 2000ms

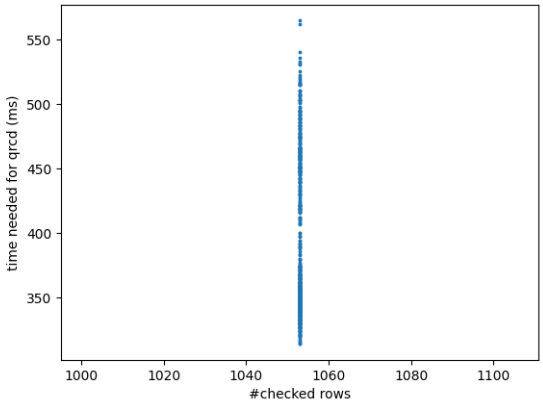

Abbildung 8: #Zeilen-Zeit-Relation 1014 Knotenanfragen bei 1 Löschung pro 2000ms

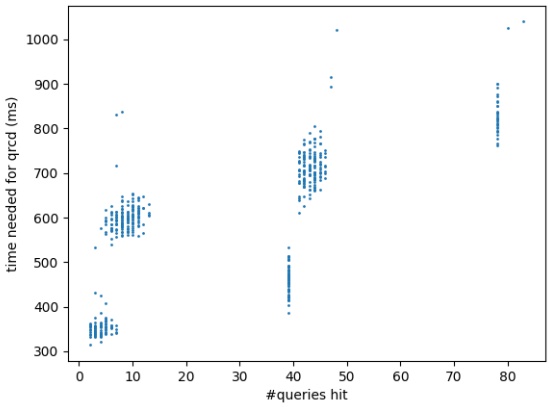

Abbildung 9: #Treffer-Zeit-Relation 1014 Knotenanfragen bei 1 Löschung pro 500ms

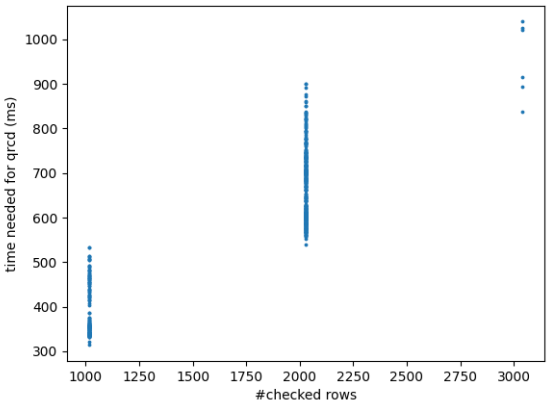

Abbildung 10: #Zeilen-Zeit-Relation 1014 Knotenanfragen bei 1 Löschung pro 500ms

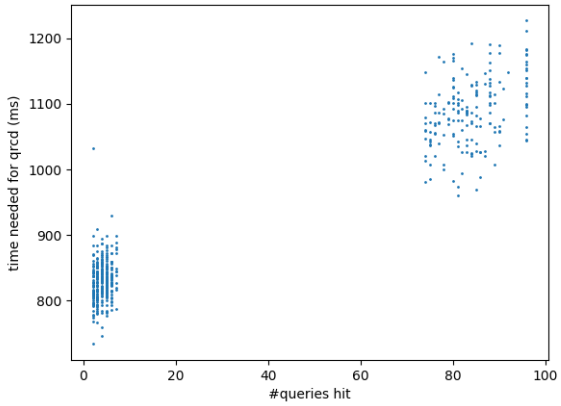

Abbildung 11: 2496 Knotenanfragen. 2000ms Löschfrequenz. #Treffer-Zeit-Relation

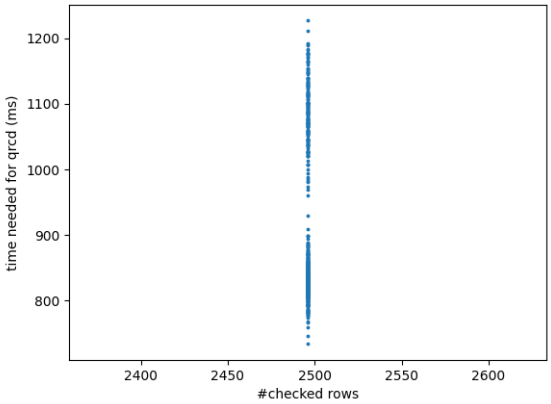

Abbildung 12: 2496 Knoteanfragen. 2000ms Löschfrequenz. #Zeilen-Zeit-Relation

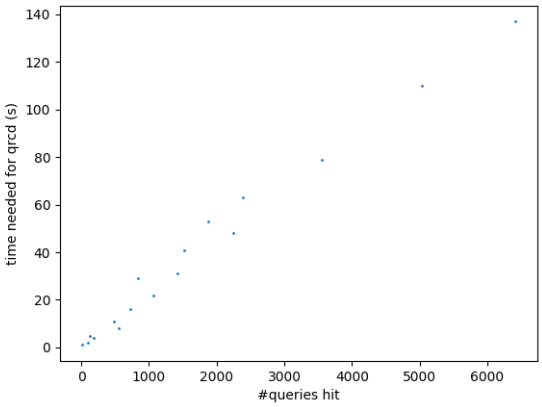

Abbildung 13: 2496 Knotenanfragen. 500ms Löschfrequenz. #Treffer-Zeit-Relation

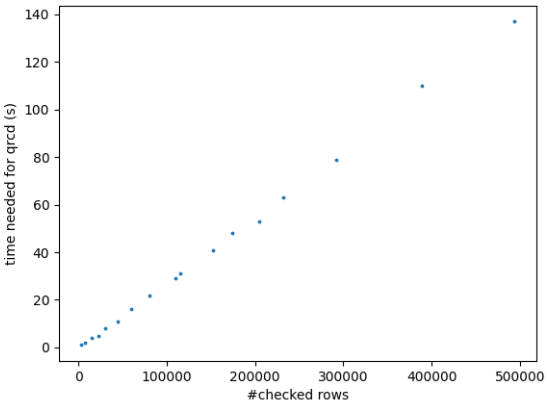

Abbildung 14: 2496 Knotenanfragen. 500ms Löschfrequenz. #Zeilen-Zeit-Relation

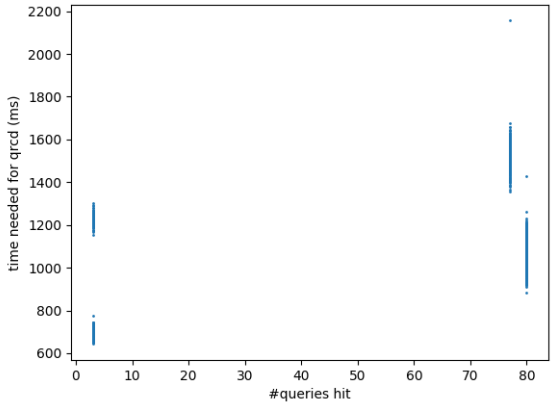

Abbildung 15: 2496 Knotenanfragen. 2000ms Aktualisiert-/Lösch-/Einfügefrequenz. #Treffer-Zeit-Relation

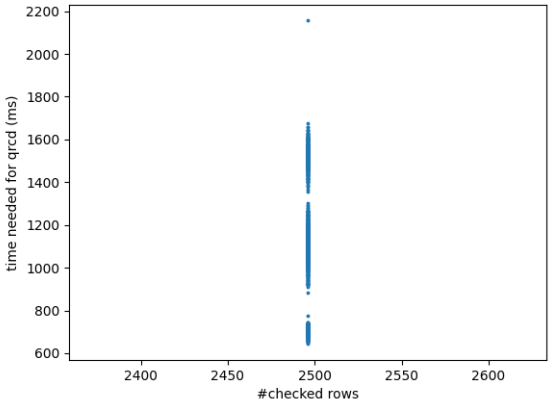

Abbildung 16: 2496 Knotenanfragen. 2000ms Aktualisiert-/Lösch-/Einfügefrequenz. #Zeilen-Zeit-Relation

Kantenanfragen

Um die Performanz bei Kantenanfragen mit der der Knotenanfragen vergleichen zu können, wurden nun 1014, zu den Knotenanfragen ähnliche, Kantenanfragen registriert.

Listing 16: Syntax registrierte Kantenanfragen

```
SELECT c_p1.pvalue AS content FROM comment c
LEFT JOIN comment_p c_p1 ON c.id = c_p1.elementId
  AND c_p1.pkey='content'
LEFT JOIN comment_p c_p2 ON c.id = c_p2.elementId
  AND c_p2.pkey='length'
JOIN commenthascreatorperson e ON e.source = c.id
JOIN person p ON e.target = p.id
LEFT JOIN person_p p_p1 ON p.id = p_p1.elementId
  AND p_p1.pkey='firstname'
WHERE c_p2.pvalue > #i AND p_p1.pvalue LIKE '#c%'
```

Das '#' Symbol signalisiert in der Anfrage 16 wieder einen Parameter. Da es zunächst 1014 Anfragen sind, liegt **i** im Intervall [0,39) und der Parameter **c** nimmt wieder jeden Buchstaben des Alphabets genau einmal an. Nun werden Kanten aus der *commenthascreatorperson*-Tabelle mit einer Frequenz von 1 Löschung pro 2000ms entfernt. Die bei diesem Prozedere entstandenen Messungen wurden in den Abbildungen 17 und 18 visualisiert. Dort ist zu erkennen, dass die allgemein benötigte Zeit für

die Überprüfung der Ergebnismengenänderungen deutlich geringer ist, als die bei 1014 registrierten Knotenanfragen. Das liegt daran, dass hier lediglich die Kanten, aber nicht der Quell- oder Zielknoten der Kante, gelöscht wurden. Somit müssen die Eigenschaften nur in aktuellen Graphen nachgeschlagen werden, ein Zugriff auf historische Daten ist nicht notwendig. Somit verringert sich die Anzahl zu überprüfender Zeilen und der Algorithmus läuft schneller. Allgemein kann also geschlussfolgert werden, dass ein Zugriff auf historische Daten zeitaufwändiger ist, da viel mehr Zeilen überprüft werden müssen. Die nach Vervierfachung der Frequenz entstandenen Messwerte sind in den Grafiken 19 und 20 gezeigt. Auch hier macht sich der zeitliche Unterschied im Vergleich zu den Knotenanfragen (Abbildungen 9 und 10) deutlich bemerkbar. In Abbildung 20 ist die Anzahl zu überprüfender Reihen nur einmal über 3000 gestiegen. Das liegt daran, dass der Algorithmus die Zeilen hier schneller überprüft und seltener der Fall vorliegt, dass die Überprüfzeit länger dauert als die Zeit bis zur nächsten Änderung. Nun wird das Intervall des Längenrestriktionsparameters **i** wieder bis auf inklusive 96 erweitert, so dass 2496 Kantenanfragen registriert werden. Die bei einer Löschperiode von 2000ms enstandenen Messwerte sind in den Abbildungen 21 und 22 gezeigt. Umso mehr Anfragen registriert sind, desto signifikanter ist der Unterschied zwischen Überprüfungen von Knoten- und Kantenanfragen. Während auf der Abbildung 11 die Zeit pro Überprüfung auf bis zu 1200ms stieg, sind hier, auf Grafik 21 alle Überprüfungen auf knapp unter 400ms, also nur ein Drittel der benötigten Überprüfungszeit. Auch hier wurde, durch Abbildung 22 gezeigt, nie länger als 2 Sekunden für eine Überprüfung benötigt. Die Evaluationsergebnisse nach Vervierfachung der Frequenz sind in den Abbildungen 23 und 24 visualisiert. Im Vergleich zur äquivalenten Anzahl an Knotenanfragen und Löschfrequenz (Abbildung 13, 14) weisen diese Kurven einen wesentlich geringeren Anstieg auf. Die Überprüfungszeit dauerte nie länger als 4 Sekunden, während der Algorithmus bei der Überprüfung der Knotenanfragen, auf Grafik 13 und 14, bis zu 140 Sekunden benötigte. Auch der Vergleich der zu überprüfenden Zeilen wird bei steigender Anzahl an Anfragen und höherer Löschfrequenz noch deutlicher. Die Knotengrafik 14 zeigt, dass hier bis zu 500.000 Reihen am Ende der 1000 Löschoperationen überprüft werden mussten, während auf Abbildung 24 die 100.000 Reihen nicht überschritten werden.

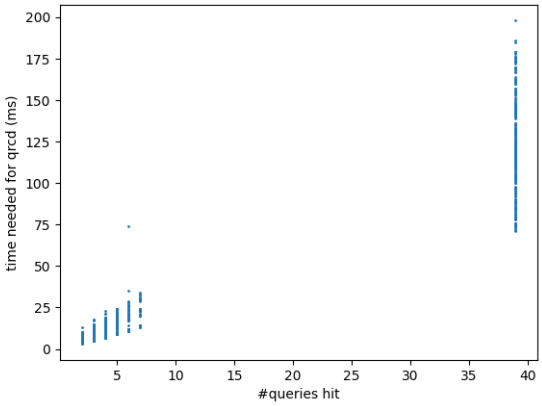

Abbildung 17: 1046 Kantenanfragen. 2000ms Löschfrequenz. #Treffer-Zeit-Relation

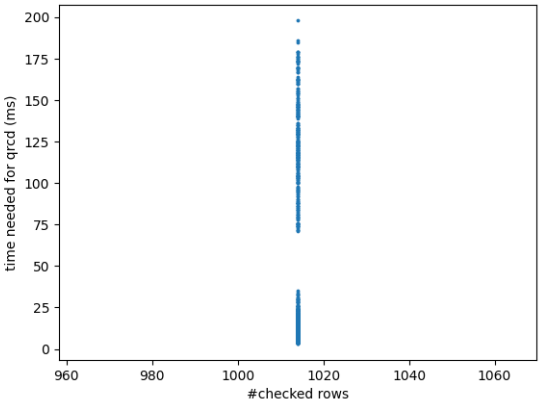

Abbildung 18: 1046 Kantenanfragen. 2000ms Löschfrequenz. #Zeilen-Zeit-Relation

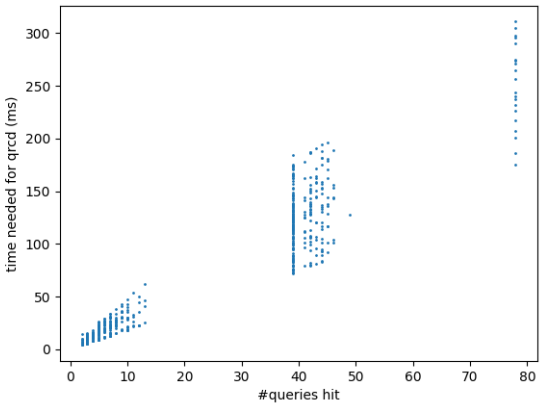

Abbildung 19: 1046 Kantenanfragen. 500ms Löschfrequenz. #Treffer-Zeit-Relation

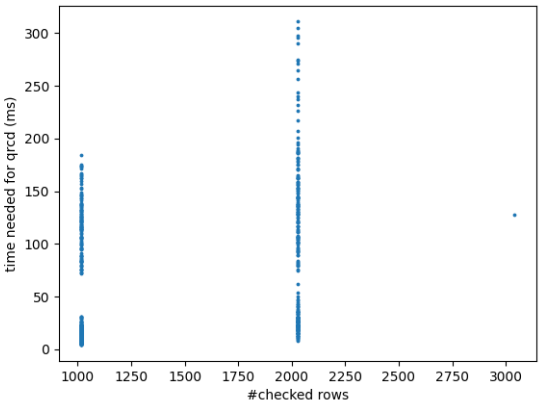

Abbildung 20: 1046 Kantenanfragen. 500ms Löschfrequenz. #Zeilen-Zeit-Relation

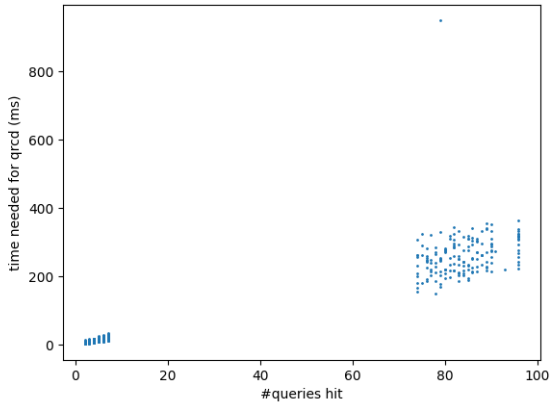

Abbildung 21: 2496 Kantenanfragen. 2000ms Löschfrequenz. #Treffer-Zeit-Relation

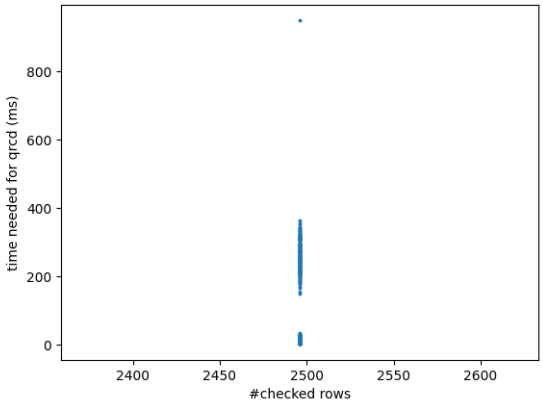

Abbildung 22: 2496 Kantenanfragen. 2000ms Löschfrequenz. #Zeilen-Zeit-Relation

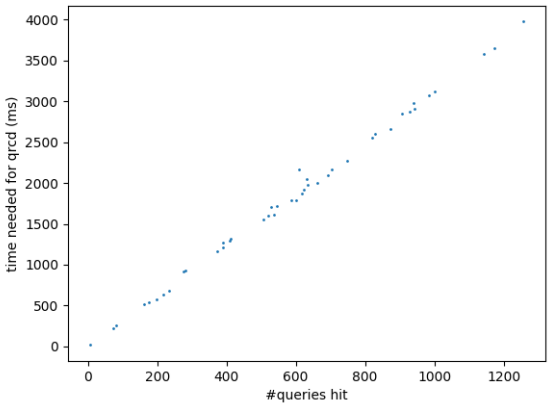

Abbildung 23: 2496 Kantenanfragen. 500ms Löschfrequenz. #Treffer-Zeit-Relation

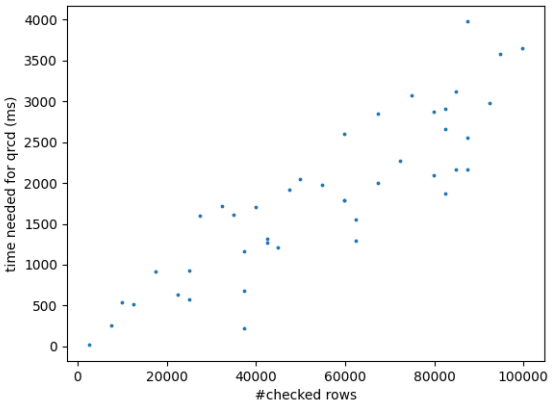

Abbildung 24: 2496 Kantenanfragen. 500ms Löschfrequenz. #Zeilen-Zeit-Relation

5.2 Anfrageregistrierung

Bei den für die Registrierung getesten Anfragen wurde zwischen der Anzahl an Prädikaten, der Anzahl an Joins, der Art der logischen Verbindung (*OR* oder *AND*), sowie der Art der registrierten Anfragen (Kanten- bzw. Knotenanfragen) variiert.

Knotenanfragen
Zunächst wurden 3940 Knotenanfragen registriert, mit einer Syntax wie in Listing 15 beschrieben. Hierzu lag der Längenparameter **i** im Intervall [0,154) und der Paramter **c** nahm wieder jeden Buchstaben des Alphabets genau einmal an. Dementsprechend kamen zwei *Joins* in der Anfrage, beide auf die Eigenschaftstabelle *comment_p*, und zwei Prädikate vor. Auf Grafik 25 ist zu erkennen, dass die Kurve linear steigt. Das bedeutet, dass jede Anfrage in etwa gleich lang für ihre Registrierung benötigt. Im Schnitt benötigte jede Anfrage 13,997 ms Registrierzeit. Anschließend wurde für den Knoten und die beiden Eigenschaften Einschränkungen auf das Startzeitstempelattribut *txStart* hinzugefügt.

Listing 17: Syntax registrierte Knotenanfragen - erweitert

```
SELECT c_p1.pvalue AS content FROM comment c
LEFT JOIN comment_p c_p1 ON c.id = c_p1.elementId
  AND c_p1.pkey = 'content'
LEFT JOIN comment_p c_p2 ON c.id = c_p2.elementId
  AND c_p2.pkey = 'length'
WHERE c.txStart > current_timestamp
  AND c_p1.txStart > current_timestamp
  AND c_p2.txStart > current_timestamp AND c_p2.pvalue > #i
  AND c_p1.pvalue LIKE '#c%
```

In Listing 17 ist verdeutlicht, dass nun drei temporale Prädikate hinzugefügt wurden. Somit besitzt die Anfrage zwei *Joins* und fünf Prädikate. Die auf Grafik 26 abgebildete Kurve ist etwas steiler als die vorherige auf Abbildung 25. Die hier im Durchschnitt benötigte Zeit pro Anfrageregistrierung beträgt 19,127 ms, also circa 5 ms mehr als im vorherigen Durchlauf. Um die genauen Auswirkungen der verschiedenen Parameter auf die Anfrageregistrierung zu analysieren, wird als Basiswert eine Anfrage 4000 Mal registriert.

Listing 18: Registrierte Knotenanfrage vier Prädikate und ein Join

```
SELECT c_p1.pvalue as content
FROM comment c
LEFT JOIN comment_p c_p1
  ON c.id = c_p1.elementId AND c_p1.pkey='content'
WHERE c_p1.pvalue = 'cool'
  AND c_p1.pvalue = 'boring'
  AND c_p1.pvalue = 'interesting'
  AND c_p1.pvalue = 'nice'
```

Die in Listing 18 gezeigte Anfrage wird nun 4000 mal registriert, um wieder einen guten Mittelwert zu erhalten. Ihre Prädikate sind ausschließlich mit Konjunktionen

miteinander verbunden. Im Schnitt hat die Registrierung dieser Anfrage 12,739 ms benötigt. Werden die Konjunktionen der Anfrage 18 nun durch Disjunktionen ersetzt, entstehen die in Abbildung 28 dargestellten Messwerte. Die dort gezeigte Kurve ist wesentlich steiler als die in Abbildung 27. Die im Durchschnitt benötigte Zeit, um diese Anfrage zu registrieren, beträgt 30,032 ms, also mehr als das Doppelte der Zeit die benötigt wird, um die gleiche Anfrage mit Konjunktionen zu registrieren. Die Ursache liegt darin, dass jede Anfrage zunächst am *OR* gespalten und jede Disjunktion für sich betrachtet wird. Das muss geschehen, da jeder Disjunktionsteil in der Filtertabelle eine eigene Reihe darstellt. Bei der Anfrage, deren Prädikate mit der Konjunktion verbunden wurden, muss nur dieser eine Disjunktionsteil betrachtet und anschließend in eine Zeile eingefügt werden. Bei der Anfrage, deren Prädikate mit der Disjunktion verbunden sind, müssen hingegen vier Teile betrachtet und jeder in eine eigene Zeile in die Filtertabelle eingefügt werden. Um die Auswirkung der Anzahl an Parameter zu verdeutlichen wird die Anfrage 18 nun um vier Prädikate erweitert.

Listing 19: Registrierte Knotenanfrage acht Prädikate und ein Join

```
SELECT c_p1.pvalue as content
FROM comment c
LEFT JOIN comment_p c_p1
  ON c.id = c_p1.elementId AND c_p1.pkey='content'
WHERE c_p1.pvalue = 'cool' AND c_p1.pvalue = 'boring'
  AND c_p1.pvalue = 'interesting' AND c_p1.pvalue = 'nice'
  AND c_p1.pvalue = 'funny' AND c_p1.pvalue = 'complex'
  AND c_p1.pvalue = 'short' AND c_p1.pvalue = 'long'
```

Die im Durchschnitt benötigte Zeit beträgt 12,898 ms. Der Unterschied im Vergleich zwischen den Messwerten, dargestellt in Abbildung 27 und 29, ist nur minimal. Nach der Ersetzung der Konjunktionen durch Disjunktionen enstanden die in Abbildung 30 gezeigten Messwerte. Hier ist die durchschnittlich benötigte Zeit bereits bei 53,561 ms. Daraus lässt sich schlussfolgern, dass nicht die Anzahl der Prädikate selbst, sondern die Anzahl an *OR*s in der **WHERE**-Klausel die Performanz der Anfragenregistrierung bestimmt. Um den Parameter der Anzahl an **JOINS** noch zu betrachten, wurde die Anfrage 19 mit den gleichen Prädikaten genommen und um **JOINS** auf jede mögliche Eigenschafts in der *comment_p*-Tabelle erweitert. Die Eigenschaften hatten jedoch keine Auswirkung auf die **WHERE**-Klausel. Es ging nur darum, die Anzahl an **JOINS** zu erweitern. Die Messwerte für die Konjunktion sind in Abbildung 31 visualisiert, und die für die Disjunktion in Abbildung 32. Diese beiden Abbildungen weisen kaum Unterschiede zu den Abbildungen 28 und 29 auf. Die durschnittliche Registrierzeit betrug bei den Disjunktionen 53,561 ms und bei den Konjunktionen 13,684 ms. Damit ist gezeigt, dass bei der Anfrageregistrierung auch die Anzahl an **JOINS** einen sehr kleinen bis keinen Einfluss auf die, für die Anfrageregistrierung

benötigte Zeit hat. Den stärksten Einfluss hat die Art der logischen Verbindungen zwischen den Prädikaten, also *OR* oder *AND*.

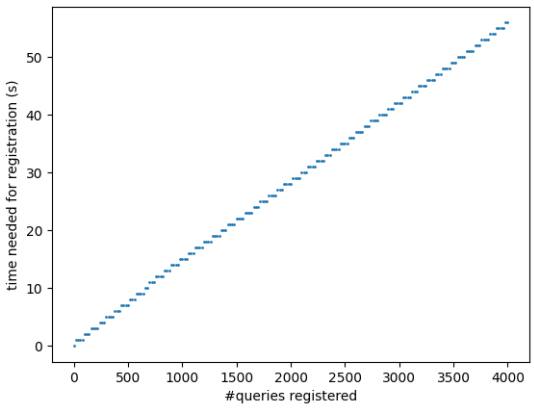

Abbildung 25: 3940 registrierte Knotenanfragen. Zwei Prädikate, zwei Joins.

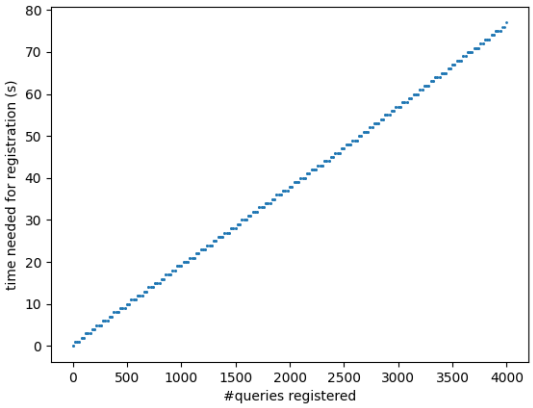

Abbildung 26: 3940 registrierte Knotenanfragen. Fünf Prädikate, zwei Joins.

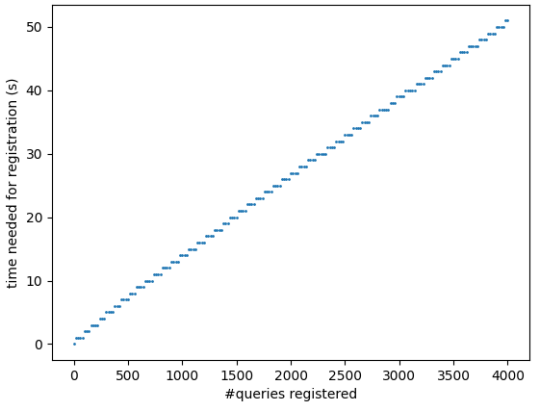

Abbildung 27: 4000 mal registrierte Knotenanfrage. Konjunktionen, vier Prädikate, ein Join.

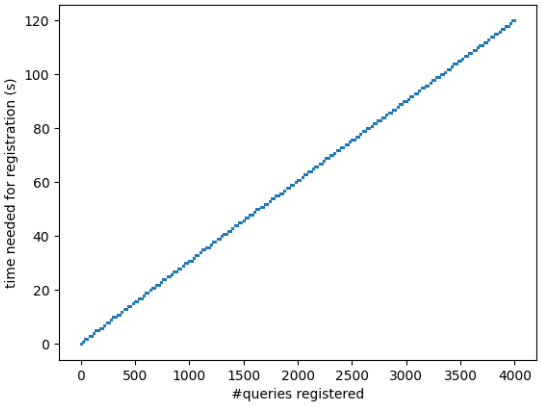

Abbildung 28: 4000 mal registrierte Knotenanfrage. Disjunktionen, vier Prädikate, ein Join.

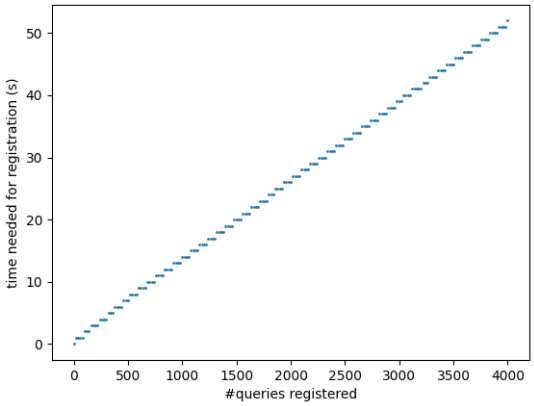

Abbildung 29: 4000 mal registrierte Knotenanfrage. Konjunktionen, acht Prädikate, ein Join.

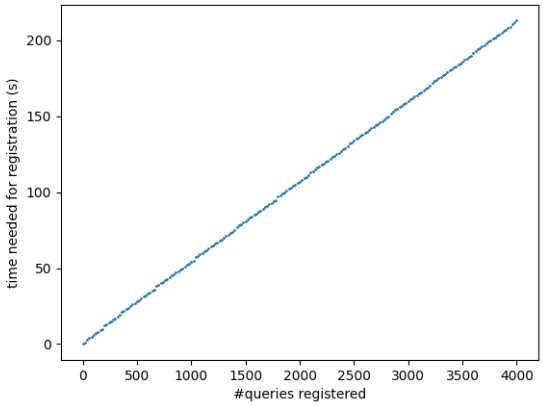

Abbildung 30: 4000 mal registrierte Knotenanfrage. Disjunktionen, acht Prädikate, ein Join.

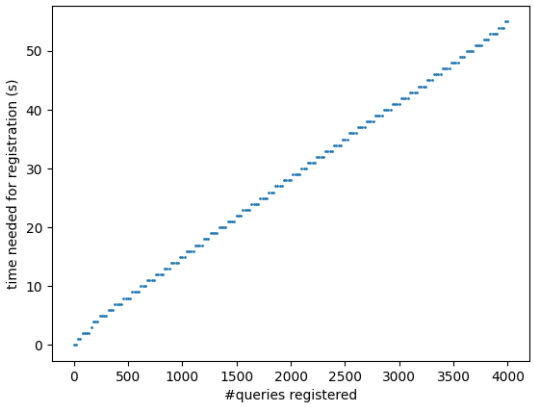

Abbildung 31: 4000 mal registrierte Knotenanfrage. Konjunktionen, acht Prädikate, vier Joins

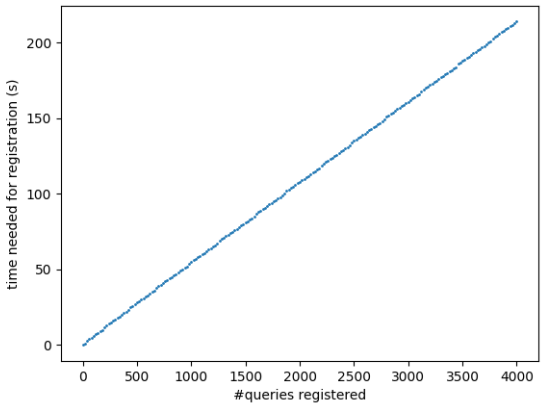

Abbildung 32: 4000 mal registrierte Knotenanfrage. Disjunktionen, acht Prädikate, vier Joins.

Kantenanfragen

Die bei der Registrierung von Kantenanfragen gemessenen Werte bestätigen das Ergebnis. Es wurde zunächst eine geringe Anzahl an *JOINS* genutzt und vier Prädikate in die **WHERE**-Klausel aufgenommen. Die Messungen nach 4000-maligen Registrieren der in Listing 20 gezeigten Anfrage sind in Grafik 33 dargestellt. Werden die Konjunktionen durch Disjunktionen ersetzt, entstehen die in Grafik 34 gezeigten

Listing 20: Registrierte Kantenanfrage vier Prädikate und vier Joins

```
SELECT c_p1.pvalue as content, p_p1.pvalue as firstname
FROM comment c
LEFT JOIN comment_p c_p1 ON c.id = c_p1.elementId
  AND c_p1.pkey='content'
JOIN commenthascreatorperson e ON e.source = c.id
JOIN person p ON p.id = e.target
LEFT JOIN person_p p_p1 ON p_p1.elementId = p.id
WHERE c_p1.pvalue = 'cool' AND p_p1.pvalue = 'Mark'
  AND c_p1.pvalue = 'boring' AND p_p1.pvalue = 'Lisa'
```

Messwerte. Die Abbildungen 33 und 34 ähneln den Abbildungen 27 und 28 sehr stark. Dies ist auch logisch, da die Anzahl an Prädikaten und die Art der logischen Verbindung gleich sind. Nur die Anzahl an **JOINS** unterscheidet sich, deren Auswirkungen auf die Registrierungszeit jedoch schon nach Abbildungen 31 und 32 als sehr gering eingestuft wurde. Um dies noch einmal zu verdeutlichen, wurde die Anfrage 20 um weitere sechs Joins und vier Prädikate erweitert. Es wurde die *comment_p* - und *person_p*-Tabelle jeweils dreimal mehr *gejoint* und jeweils zwei Einschränkungen auf die Eigenschaften *firstname* und *content* hinzugefügt. Die dabei gemessenen Werte für Konjunktion und Disjunktion sind in den Abbildungen 35 beziehungsweise 36 visualisiert. Auch diese beiden Abbildungen sind sehr ähnlich zu den Abbildungen für Knotenanfragenregistrierungen 31 und 32. Die Zeit für die Registrierung der Anfrage mit acht, mittels Konjunktion verbundenen, Prädikaten, unterscheidet sich kaum zur Registrierungszeit der Kantenanfrage mit vier, mittels Konjunktion verbundenen, Prädikaten, dargestellt auf Abbildung 33. Nur die Registrierungszeit der Anfrage, deren Prädikate mittels Disjunktion verbunden waren, hat sich stark erhöht.

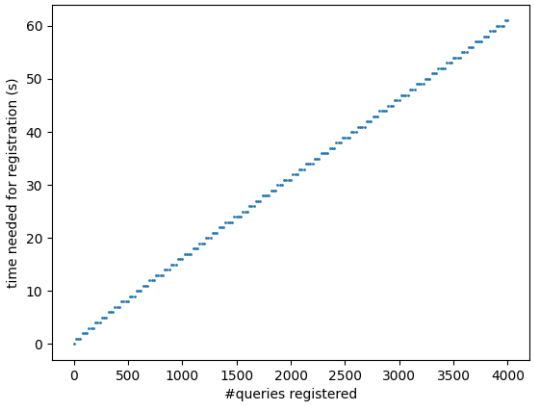

Abbildung 33: 4000 mal registrierte Kantenanfrage. Konjunktionen, vier Prädikate, vier Joins.

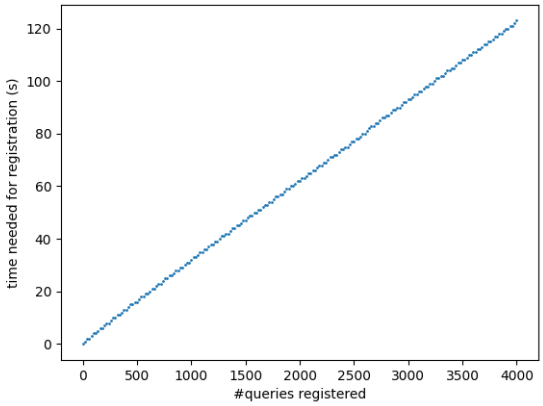

Abbildung 34: 4000 mal registrierte Kantenanfrage. Disjunktionen, vier Prädikate, vier Joins

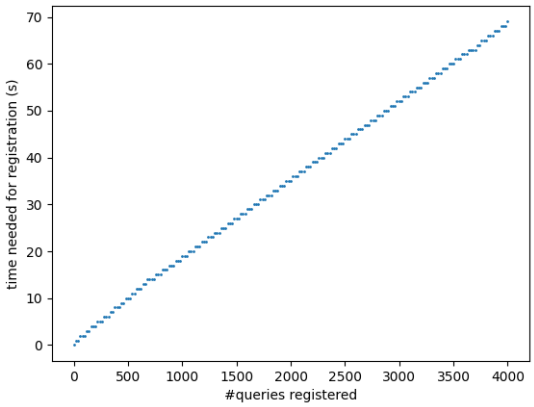

Abbildung 35: 4000 mal registrierte Kantenanfrage. Konjunktionen, acht Prädikate, zehn Joins.

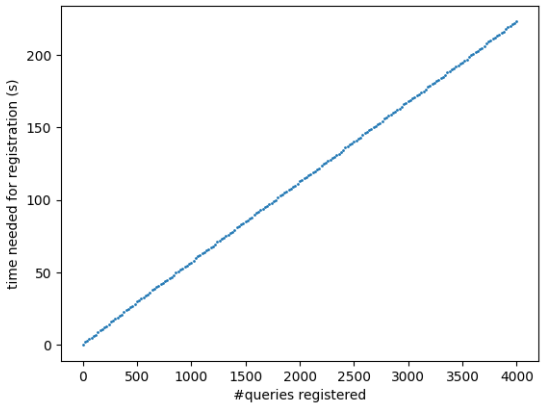

Abbildung 36: 4000 mal registrierte Kantenanfrage. Disjunktionen, acht Prädikate, zehn Joins.

6 Zusammenfassung und Ausblick

Zu Beginn der Arbeit wurde im Kapitel 1 die Problemstellung und Zielsetzung für dieses Projekt geklärt. Nachdem im Anschluss in Kapitel 3 relatierte Veröffentlichungen zu der in dieser Arbeit gegebenen Problemstellung angeschnitten wurden, werden im Kapitel 2 konkrete Begriffe geklärt, die zum Verständnis der Umsetzung der Lösung notwendig sind. Im Kapitel 4.1 wird anschließend geklärt, wie der Registrierungsprozess für *Continuous Queries* aussieht. Es werden Prädikate aus der **WHERE**-Klausel extrahiert, den Tabellen zugewiesen, in der sie gespeichert sind, und anschließend in einer dazugehörigen Filtertabelle abgelegt. Auf in der Anfrage betroffenenen Tabellen werden *Trigger* angelegt, die Zeilen aus betroffenen Filtertabellen selektieren und in eine *potentialChanges*-Tabelle einfügen, die für die Erkennung von Ergebnismengenänderungen Informationen bereitstellt. Im Kapitel 5.2 wurde gezeigt, dass die Zeit für die Registrierung solcher Anfragen maßgeblich von der logischen Verbindung der Prädikate abhängt. Disjunktionen sind wesentlich zeitaufwändiger zu registrieren als Konjunktionen. Dies hat als Ursache, dass jeder Disjunktionsteil einzeln betrachtet werden muss, da er eine eigene Zeile in der Filtertabelle darstellt. Bei der folgenden Ergebnismengenänderungserkennung werden, nach Änderungen an der Datenbank, Anfragen an betroffene Knoten oder Kanten gestellt. Mit diesen Anfragen wird überprüft, ob der Knoten oder die Kante Prädikate erfüllt, die in einer Filtertabelle hinterlegt sind. Ist dies der Fall, so wird eine Benachrichtigung an den Nutzer geschickt, der diese Anfrage registriert hat. Die Zeit, die die Ergebnismengenänderungserkennung benötigt, hängt hauptsächlich von der Anzahl an Reihen ab, die überprüft werden müssen. Umso mehr Anfragen auf einer Tabelle registriert wurden, desto länger dauert der Überprüfungszeitprozess. Ein weiterer Parameter ist die Art der Änderung. Da bei *Delete*- und *Updateanweisungen* historische Daten überprüft werden müssen, um eine Ergebnismengenänderung festzustellen, ist die Datenmenge, aus der selektiert werden muss, größer als bei *Insertanweisungen*, bei denen nur in aktuellen Daten gesucht werden muss. Das hat zur Folge, dass die Überprüfung von *Delete*- und *Updateanweisungen* länger benötigt. Dies erklärt auch, weshalb Kantenanfragen schneller überprüft werden als Knotenanfragen. Die Prädikate der Quell- und Zielknoten werden bei der Löschung einer Kante nicht entfernt, weshalb auch hier nur in aktuellen Daten gesucht werden muss, um eine Ergebnismengenänderung einer Anfrage festzustellen. Im Kapitel 5 wurde gezeigt, dass bei einer großen Zahl an Anfragen und einer hohen Frequenz an Änderungen das aktuelle Verfahren nach kurzer Zeit sehr lange für Überprüfungen benötigt.

Literatur

[1] S. Babu, *Continuous Query*, pp. 492–493. Boston, MA: Springer US, 2009.

[2] S. S. Vemuri, B. Sinha, A. Ganesh, and S. B. Chitti, "Generating continuous query notifications," Oct. 11 2011. US Patent 8,037,040.

[3] "Using continuous query notification (cqn)." `https://docs.oracle.com/en/database/oracle/oracle-database/12.2/adfns/cqn.html#GUID-373BAF72-3E63-42FE-8BEA-8A2AEFBF1C35`. Last accessed on 05-07-2021.

[4] R. Angles, "The property graph database model.," in *AMW*, 2018.

[5] M. Knight, "What is a property graph?," Apr. 4 2021.

[6] C. Rost, A. Thor, and E. Rahm, "Temporal graph analysis using gradoop," in *BTW 2019 – Workshopband* (H. Meyer, N. Ritter, A. Thor, D. Nicklas, A. Heuer, and M. Klettke, eds.), pp. 109–118, Gesellschaft für Informatik, Bonn, 2019.

[7] "Property graph query language." `https://pgql-lang.org/`. Last accessed on 01-07-2021.

[8] O. van Rest, S. Hong, J. Kim, X. Meng, and H. Chafi, "Pgql: a property graph query language," in *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems*, pp. 1–6, 2016.

[9] C. Rost, K. Gomez, M. Täschner, P. Fritzsche, L. Schons, L. Christ, T. Adameit, M. Junghanns, and E. Rahm, "Distributed temporal graph analytics with gradoop," *The VLDB Journal*, pp. 1–27, 2021.

[10] "System-versioned tables." `https://mariadb.com/kb/en/system-versioned-tables/`. Last accessed on 30-06-2021.

[11] "What is apache activemq?." `https://www.openlogic.com/blog/what-apache-activemq`. Last accessed on 20-06-2021.

[12] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang, "Niagaracq: A scalable continuous query system for internet databases," in *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, pp. 379–390, 2000.

[13] E. Begoli, T. Akidau, F. Hueske, J. Hyde, K. Knight, and K. Knowles, "One sql to rule them all-an efficient and syntactically idiomatic approach to management of streams and tables," in *Proceedings of the 2019 International Conference on Management of Data*, pp. 1757–1772, 2019.

[14] K. Kulkarni and J.-E. Michels, "Temporal features in sql: 2011," *ACM Sigmod Record*, vol. 41, no. 3, pp. 34–43, 2012.

[15] M. Besta, M. Fischer, V. Kalavri, M. Kapralov, and T. Hoefler, "Practice of streaming processing of dynamic graphs: Concepts, models, and systems," *arXiv preprint arXiv:1912.12740*, 2019.

[16] Wikipedia, "Datenbanktrigger — wikipedia, die freie enzyklopädie," 2019. [Online; Stand 28. Juni 2021].

[17] "Invoke triggers for foreign key cascade actions." `https://jira.mariadb.org/browse/MDEV-19402`, 2019. Last accessed on 05-06-2021.

[18] J. Reock, "Is activemq's dynamic queue creation working for you?." `https://www.openlogic.com/blog/activemqs-dynamic-queue-creation-working-you`. Last accessed on 20-06-2021.

[19] "Jndi support." `https://activemq.apache.org/jndi-support.html`. Last accessed on 20-06-2021.

Zimmer, Maximilian
Name, Vorname

3707301
Matrikelnummer

Ich versichere, dass ich die Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Darüber hinaus versichere ich, dass die elektronische Version der Bachelorarbeit mit der gedruckten Version übereinstimmt.

Leipzig, 11.07.2021
Ort, Datum


Unterschrift