Graphbasierte vs. relationale Datenbanken

Vergleich der Speicherung und Abfrage bei der Analyse von Social Data

ABSTRACT

Die Hausarbeit vergleicht die Abfragesprache *SQL* für relationale Datenbanken mit der Abfragesprache *Cypher* für graphbasierte Datenbanken. Nach einem theoretischen Vergleich der Datenbankunterschiede, folgt ein praktischer Vergleich, bei dem ein Datensatz mit Beiträgen aus Sozialen Netzwerken untersucht wird. Kriterien für den Vergleich sind die Komplexität und Länge der Abfragen und die Dauer der Durchführung.

Schlagworte: SQL, Cypher, relationale Datenbank, graphbasierte Datenbank, Graph Datenbank, Social Media Analyse

AUTOR

Jens Gäbeler

MatrNr. 70453756

j.gaebeler@ostfalia.de

INHALT

EINLEITUNG

Die Bedeutung einer Relation[1] erklärt der Duden folgendermaßen:

a. *(Fachsprache) Beziehung, in der sich [zwei] Dinge, Gegebenheiten, Begriffe vergleichen lassen oder [wechselseitig] bedingen; Verhältnis*
b. *(Mathematik) Beziehung zwischen den Elementen einer Menge*

Eine *relationale Datenbank* ist – verwendet man die mathematische Bedeutung – eine Datenbank in der Beziehungen zwischen Elementen vorkommen und, da dies das einzige beschreibende Attribut dieser Art Datenbank ist, einen wichtigen Teil ausmachen. Die Speicherung von Informationen geschieht allerdings hauptsächlich in Tabellenform und Beziehungen werden durch Fremdschlüssel abgebildet. Ein Fremdschlüssel ist ein Teil des Schemas von Tabellen und verbindet eine Spalte einer Tabelle mit einer Spalte einer anderen Tabelle. Eine konkrete Relation von zwei Objekten wird durch das Zusammenführen (*JOIN*) dieser beiden Tabellen realisiert, also indirekt. Der Begriff *relationalen Datenbank* lässt erwarten, dass die enthaltenen Relationen direkt und unmittelbar zur Verfügung stehen. Durch die nötige Verwendung eines Hilfsmittel, um eine Beziehung darzustellen, ist dies jedoch nicht gegeben.

In dieser Hausarbeit wird daher ein Vergleich zu einem anderen Datenbanktyp, der Beziehungen zwischen Einzelobjekten konkret abspeichert, gezogen. Hier werden Objekte und Relationen als eigenständige Schemabestandteile aufgefasst und verwendet. Objekte werden dabei durch konkrete Relationen verbunden. Die Rede ist von *graphbasierten Datenbanken*. Mithilfe eines Datensatzes mit Beiträgen aus den Sozialen Medien wird analysiert inwieweit sich beide Datenbankarten in ihrer Speicherung und Abfrageverwendung unterscheiden. Da mit Cypher eine neue Sprache erlernt wurde, wurde bei der Erstellung der SQL Abfragen darauf geachtet, keine Funktionen oder sonstige programmatischen Lösungen anzuwenden. Dadurch soll die Vergleichbarkeit der nativen Abfragesprachen erhöht werden.

[1] https://www.duden.de/rechtschreibung/Relation

UNTERSCHIEDE DER BEIDEN DATENBANKTYPEN

Die Unterschiede können leichter erläutert werden, wenn zuvor die Ähnlichkeiten und ihr Ableitung aus dem ER-Modell erläutert werden.

ER-Modell

Die Grundlage bei der Erstellung der beiden Datenbankenarten – der relationalen Datenbank und der Graph Datenbank – ist meist ein ER-Modell (Entity Relationship Modell). Ein ER-Modell kann aufgefasst werden als eine Abbildung eines Teiles der Realwelt, bestehend aus Gegenständen und Beziehungen. Ein Beispiel für ein einfaches ER-Modell mit den Gegenständen (Entitäten) *Studenten* und *Vorlesung* und der Beziehung (Relation) *besuchen* ist in Abbildung 1 zu sehen.

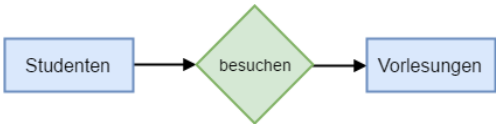


Abbildung 1: Einfaches ER-Modell

Speicherung

Bei relationalen Datenbanken werden aus dem Modell nun Gegenstände in Tabellen umgewandelt. Relationen werden teilweise diesen Tabellen hinzugefügt oder in separate Tabellen, sogenannte Indexe oder Relationstabellen, gespeichert.

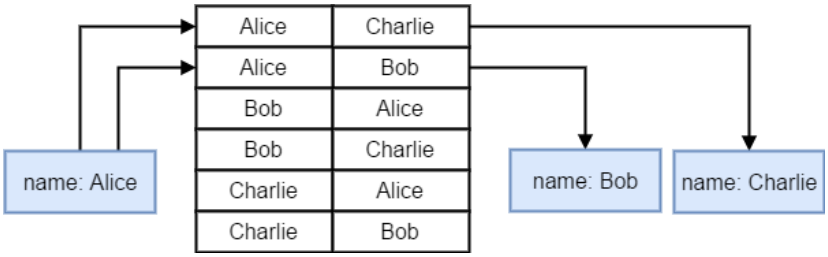


Abbildung 2: Durch globale Indexe realisierte Relationen

Die algorithmische Komplexität der Abfrage über einen Index (wie in Abbildung 2 gezeigt) ist, je nach Implementierung $O(\log n)$ groß, n sei dabei die Anzahl der Zeilen in der Indextabelle. Die gleiche Abfrage in einer Datenbank ohne Indexe aber mit direkten Relationen, wie in Abbildung 3 zu sehen, besitzt eine Komplexität von $O(1)$. Der Durchlauf durch m Schritte „kostet" dann $O(m \cdot \log n)$ für die Indexvariante im Gegensatz zu $O(m)$ mit Relationen.[2]

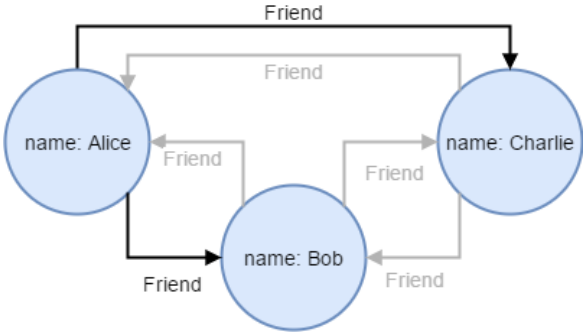


Abbildung 3: Speicherung der Relationen ohne Verwendung eines Indexes

[2] Robinson et al. (2015, S. 149ff.)

Am Beispiel der Graph Datenbank Neo4j werden der Informationen in unterschiedlichen Speichern (engl. *store files*) abgelegt. Es gibt separate Speicherplätze und physikalische Dateien für Knoten, Relationen, Labels und Eigenschaften (z.B. *neostore.nodestore.db* für die Speicherung der Knoten).[3] Alle Einträge eines Stores besitzen die gleiche Länge, wodurch die ID des Eintrags nicht abgespeichert werden muss, sondern aus der Position des ersten Bytes im Store ersichtlich ist. Die Stores werden daher auch als *fixed-size record stores* bezeichnet (dt.: Speicher mit Einträgen festgelegter Größe) und sind wesentlich performanter (mit $O(1)$ i.G.z. zu $O(\log n)$ bei einer Suche).

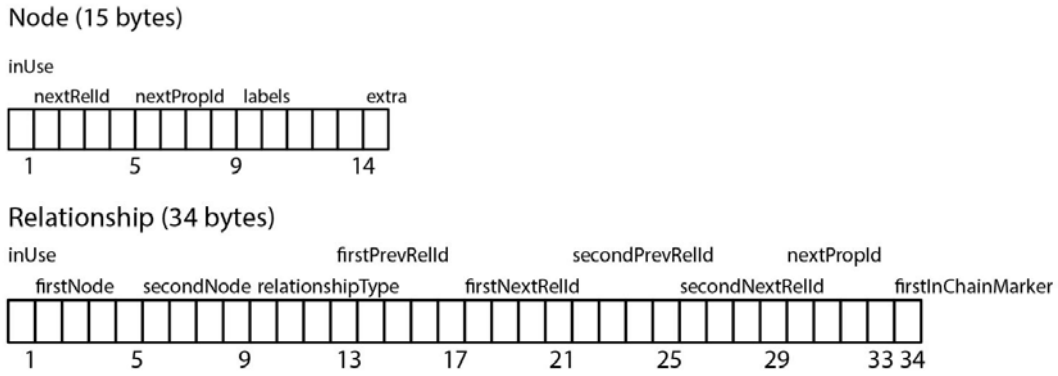


Abbildung 4: Struktur für Einträge in den Knoten- und Relationenspeicher[4]

In Neo4j der Version 2.2 besitzen Knoteneinträge eine Länge von 15 Byte (vgl. Abbildung 4). Somit beginnt der Eintrag der ID 20 bei Byte 300. In einem Knoten werden sein Aktivstatus und die IDs zur nächsten Relation, zur nächsten Eigenschaft und zu Labels abgespeichert.

Unterschieden werden die Eintragsarten in *singly linked lists* und *doubly linked lists*. In Abbildung 5 sind Beispiele für beide Arten zu erkennen. *Singly linked lists* sind Listen, die singulär verlinken, also lediglich auf die nächste ID verweisen. Knoteneinträge (*Node records*) sind ein Beispiel dafür, so verlinkt *Node1* singulär zur Eigenschaft *name*, die als Key-Value Paar im Property Store abgespeichert wird. Das zweite Key-Value Paar wird nicht direkt vom Knoten verlinkt, sondern vom ersten Key-Value Paar.

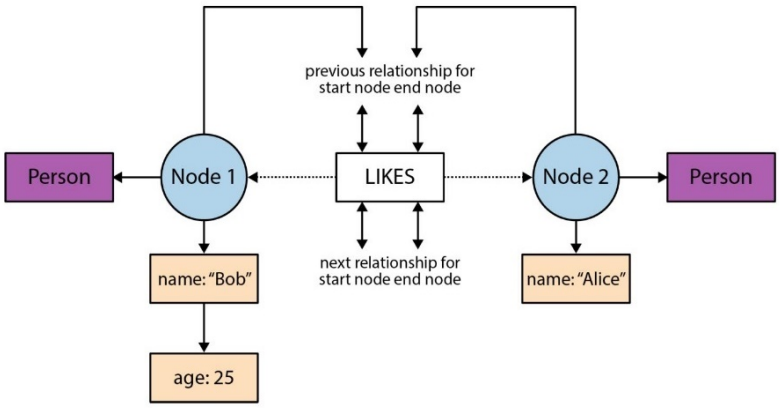


Abbildung 5: Physikalische Speicherung eines Graphs in Neo4j[5]

Doubly linked lists verknüpfen zwei gleichartige IDs, wie in der Abbildung beispielhaft dargestellt. Hier verknüpft *Likes* die Knoten *Node1* und *Node2* als Start- bzw. Endknoten. Zusätzlich wird auf den vorigen und nächsten Eintrag der jeweiligen Knoten verwiesen.

[3] Robinson et al. (2015, S. 153)

[4] Robinson et al. (2015, S. 153)

[5] Robinson et al. (2015, S. 155)

Abfragen

Die für relationale Datenbanken verwendete Abfragesprache Structured Query Language, kurz SQL, ist standardisiert und für unterschiedliche relationale Datenbank in ähnlichem Maße implementiert. Für die Graph Datenbank Neo4j wird vom Hersteller die Abfragesprache Cypher empfohlen, mit dem Hinweis, dass diese den Konzepten von SQL nachempfunden wurde und durch zusätzliche Funktionalitäten zu einfachen und präzisen Anfragen führen soll. [6]

Eine einfache SQL Abfrage um Daten aus einer Datenbank auszulesen sieht folgendermaßen aus: [7]

```
SELECT *
FROM Studenten
WHERE Semester < 10
```

Die gleiche Abfrage in einem Cypher Statement kann wie folgt aussehen: [8]

```
MATCH (s:STUDENTEN)
WHERE s.Semester < 10
RETURN s
```

Wenn Informationen aus verschiedene Quellen verknüpft werden müssen, um eine Ergebnis für eine Anfrage zu erhalten, werden in SQL Statements *JOINS* verwendet. Im Gegensatz erfolgt die Abfrage nach einer solchen Beziehung in Cypher durch eine eigene Schreibweise.[9]

SQL-Abfrage nach dem Namen aller Personen die im „IT-Department“ arbeiten:

```
SELECT name
FROM Person
LEFT JOIN Person_Department ON Person.Id = Person_Department.PersonId
LEFT JOIN Department ON Department.Id = Person_Department.DepartmentId
WHERE Department.name = 'IT Department'
```

Die entsprechende Abfrage mit Cypher für eine Neo4j Datenbank:

```
MATCH (p:Person)<-[:EMPLOYEE]-(d:Department)
WHERE d.name = "IT Department"
RETURN p.name
```

[6] neo4j (2017a)

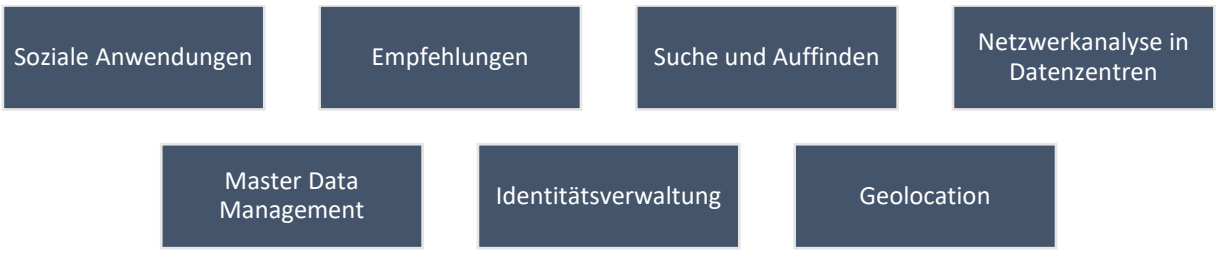
[7] Goldstein (2005)

[8] neo4j (2017b, S. 1)

[9] neo4j (2017a)

ANWENDUNGSFÄLLE FÜR GRAPH DATENBANKEN

Der naheliegendste Anwendungsfall ist die Verwendung von Graph Datenbanken in Applikationen mit sozialen Komponenten. Zudem gibt es eine Reihe weiterer Anwendungen für die Graph Datenbanken verwendet werden können. In seinem Vortrag [10] beschreibt Neo4J Mitarbeiter Ryan Boyd die Anwendungsfälle seiner Kunden:



Die Herausforderungen dieser Anwendungen für relationale Datenbanken und die Empfehlung der Verwendung einer Graph Datenbank lassen sich nach van Bruggen in drei Kategorien teilen.

Komplexe Abfragen

Komplexe Abfragen treten auf, wenn Informationen aus vielen Quellen (in rel. Datenbanken Tabellen) miteinander durch *JOINS* oder kartesisches Produkt verknüpft werden müssen um zum erwünschten Ergebnis zu gelangen. In Graph Datenbanken werden keine Join-Operationen verwendet, jeder Knoten wird durch eine Beziehung zum nächsten Knoten ohne Index (siehe Kapitel: Unterschiede der beiden Datenbanktypen > Speicherung) gespeichert. Diese Beziehung kann somit als die konkret gespeicherte Abbildung eines *JOINS* zwischen diesen beiden Knoten verstanden werden.

In Graph Datenbanken können diese komplexen Abfragen durch eine Art Musterabgleich (engl. Pattern Matching) gestellt werden. Es entsteht eine hohe Performanz, da die Abfragen nicht von der Größe des Datensatzes sind sondern von der Größe der Ergebnismenge abhängen. [11]

Abfragen auf Live Daten

In relationalen Datenbanken werden diese Art von Abfragen vorberechnet und vorformatiert. Liveabfragen laufen somit auf Duplikaten und Ent-Normalisierungen. In Graph Datenbanken können diese Redundanzen entfernt werden und in Echtzeit Abfragen auf dem Originaldatensatz gemacht werden.[12]

Pfadabfragen

In Pfadabfragen wird ebenfalls die Verbindung zwischen den Knoten analysiert. Dies geschieht um beispielsweise den kürzesten Weg zwischen diesen zu finden. Dies ist in relationalen Datenbanken nur möglich wenn die Struktur zwischen jeglichen Knoten in die Abfrage mitgegeben wird. In Graph Datenbanken ist dies nicht nötig, hier wird lediglich der Start- und der Endknoten definiert. Die Berechnung, wie der Pfad aussieht, wird von der Datenbank durchgeführt. [13]

[10] Boyd (09-2015, ab 2:30 min)

[11] van Bruggen (2014, S. 37f.)

[12] van Bruggen (2014, S. 39)

[13] van Bruggen (2014, S. 39)

Anwendungsfälle anderer Datenbanken

Es gibt allerdings auch Anwendungsfälle, die weniger gut in Graph Datenbanken ausgedrückt werden können. Abfragen auf große Datensätze, die wenige Joins benötigen oder viele Berechnungen (wie Count, Sum, Avg) durchführen haben in Graph Datenbanken eine schlechtere Performanz als beispielsweise relationale Datenbanken. Abfragen die nicht nach lokalen Informationen in einem Graph suchen, sondern den Graph selbst untersuchen sind ebenfalls nicht gut für die Verwendung von Graph Datenbanken geeignet. Hierfür gibt es separate, spezialisierte Werkzeuge. Eine weitere Anwendung, die durch Key-Value Stores oder Document Stores effizienter bearbeitet werden kann, sind einfache Abfragen, die mit Lese- oder Schreibanfragen Listensammlungen erstellen.

DATENGRUNDLAGE SOCIAL MEDIA

Als Datengrundlage wird ein Datensatz verwendet, der innerhalb des Forschungsprojekts *TAXOPublish* an der *Hochschule der Medien* mithilfe des Microsoft Tools *Social Engagement* und eines Datenflusses dieser Daten in eine lokale SQL Datenbank entstanden ist. Durch die Verwendung des Tools von Microsoft konnten innerhalb von 2 Jahren ein Datensatz zum Thema Elektrowerkzeuge in der Größenordnung von ~200.000 Beiträgen aus Sozialen Netzwerken, Blogs und Foren gesammelt werden.

Die Beiträge wurden als JSON versendet und als Einzeiler in eine Tabelle abgelegt. Die Tabelle besteht somit aus 200.000 Zeilen. Es wurden zu jedem Beitrag eine Vielzahl an Metadaten abgespeichert. Dazu gehören unter anderem der Profilname des Autors, der Zeitpunkt der Erstellung, die Url zum Beitrag und Autor, teilweise die Geodaten und der Beitrag als Text. Der Text wurde aufbereitet und von Whitespace und HTML-Elementen entfernt, aus dem Text wurden zusätzlich die genannten Tags der Form #hashtag und angesprochene Personen der Form @profile herausgezogen.

Die Daten ist somit sehr flach abgespeichert, wodurch eine Umwandlung in eine CSV Datei auf einfache Weise erfolgen konnte. Hier eine Auflistung aller 28 Spalten:

```
[postid], [externalpostid], [contenttype], [posttype], [posturi], [publicationdate],
[externalprofileid], [profilename], [displayname], [profileicon], [location],
[locationlatitude], [locationlongitude], [locationquadkey], [sourceparam],
[postlanguage], [normalscore], [provider], [providerscore], [sentimentvalue],
[refpost_profilename], [refpost_profileid], [refpost_profileicon],
[refpost_profiledisplayname], [refpost_externalid], [cleanpostcontent],
[contained_tags] , [contained_profiles]
```

Optimierung der Datengrundlage

Die Datengrundlage wurde zur besseren Verarbeitung auf ~166.000 Beiträge verkleinert. Es sind durch die Verkleinerung lediglich die Beiträge aus den Sozialen Netzwerken Twitter, Facebook und Instagram übernommen worden, da Blogbeiträge oft Längen von mehr als 4000 Zeichen aufwiesen und somit nicht in vollem Maße durch SQL Funktionen verarbeitet werden konnten (z.B. WhiteSpace- und HTML Entfernung). ~45.000 Beiträge nennen dabei mindestens ein anderes Profil, ~100.000 Beiträge nennen mindestens einen Hashtag.

Die Erstellung der CSV Datei erfolgte mithilfe des Exporttasks in ein Flatfile, welches Unicode codiert und Tabulator-getrennt abgespeichert wurde. Im Anschluss wurde, um spätere Probleme zu vermeiden, alle doppelten Anführungsstriche (“) mithilfe der Suchen/Ersetzen Funktion eines Texteditors gelöscht.

Speicherung in der jeweiligen DB

Als Grundlage der beiden zu erstellenden Datenbanken wurde das folgende ER-Modell erstellt:

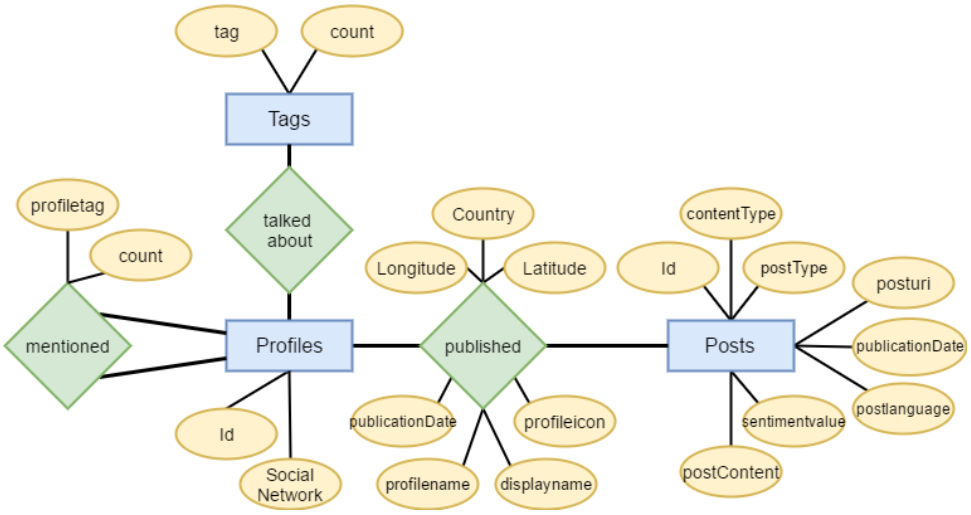


Abbildung 6: ER-Modell der zu importierenden Daten

Für das weitere Vorgehen wird, so gut wie es möglich ist, dieses Modell als Vorlage verwendet.

Relationale DB

Die Speicherung in der relationalen DB erfolgte im Forschungsprojekt nicht-normalisiert. Jeder soziale Beitrag wurde in einer eigenen Zeile mit jeglichen Metadaten, auch des Profils abgespeichert. Für eine bessere Vergleichbarkeit der relationalen Datenbank und der Graph Datenbank wurde für die Hausarbeit die Form der Einzeltabelle aufgebrochen.

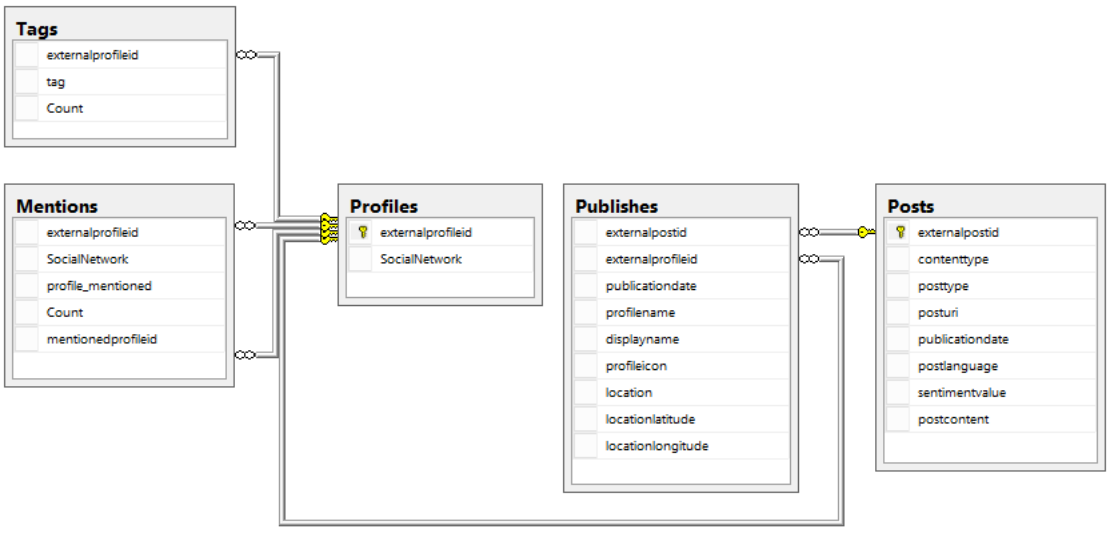


Abbildung 7: Tabellen der relationalen Datenbank

Es entstanden dabei die fünf Tabellen: *Posts*, *Profiles*, *Publishes*, *Tags* & *Mentions*. Die Relation *talks_about* und die Entität *Tags* wurden in einer Tabelle zusammengefasst. Zudem wurden die Tabellen *Profiles* und *Posts* mit Primärschlüsseln versehen, um den Fremdschlüsseln der Relationstabellen *Publishes*, *Tags* und *Mentions* Verbindung zu den Entitätstabellen zu gewähren. In der SQL Datenbank eingefügt sieht das Konstrukt wie in Abbildung 7 dargestellt aus. Das Schema der Datenbank gibt das Entity-Relationship Modell auf sehr ähnliche Weise wieder.

Graph Datenbank

In der Graph Datenbank kann das ER-Modell noch genauer nach der Vorlage erstellt werden, da in Graph Datenbanken jeweils ein Konstrukt für Entitäten und Relationen zur Verfügung steht – in relationalen Datenbanken existiert lediglich das Konstrukt *Tabelle*. Somit entstanden die drei Entitäten *Profile*, *Post* und *Tag* und die drei Relationen *Published*, *Talks_about* und *Mentioned*.

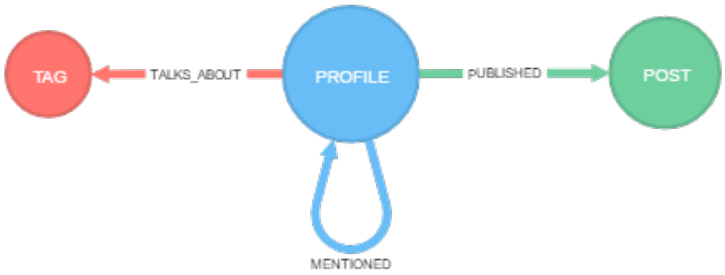


Abbildung 8: Entitäten und Relationen in der Graph Datenbank

In Abbildung 8 ist das Schema der erstellten Graph Datenbank dargestellt. Die Eigenschaften der Entitäten und Relationen gleichen den Eigenschaften des ER-Modells (nicht in der Abbildung zu sehen). Wie in NoSQL-Datenbanken üblich werden allerdings keine Eigenschaften gespeichert, die keinen Inhalt haben, i.G.z. relationalen Datenbanken, deren leere Eigenschaften mit NULL aufgefüllt werden.

Des Weiteren wurden die Indexe *Posts(PostId), Posts(ProfileMentioned), Profile(ProfileId)* für eine bessere Importperformance und der Unique Constraint *Tag(Tag)* für die Prüfung auf Einzigartigkeit erstellt.

Speicherplatzbedarf

Der benötigte Speicherplatz der Graph Datenbank in Neo4J beträgt mit insgesamt 2,86 GB ungefähr doppelt so viel wie der Speicherplatz, der in der relationalen Datenbank SQL Server mit 1,4 GB für den Datensatz benötigt wird.

HERANGEHENSWEISE

Für den Vergleich der beiden Datenbanken wird mithilfe des Social Data Datensatzes Anfragen an beide Systeme gestellt und deren Performance gemessen. Die Fragestellungen sind durch intensive Tests mit beiden Datenbanken und einschlägiger Literatur (Bücher über Neo4J & Neo4J Gists[14]) entstanden.

Testumgebung

Beide Datenbanken wurden auf dem gleichen Computer aufgesetzt. Es handelt sich dabei um einen virtuellen Server, der in einem VMware ESXI Hypervisor eingerichtet wurde. Die Spezifikationen des virtuellen Servers sind:

Hardware:

- Prozessor: virtueller 4-Core Intel® Xeon® CPU E5-2640 @2,50 GHz
- Arbeitsspeicher: 8 GB
- Festplatte: 64 GB virtuelle Festplatte (HDD), Thick-Provision Lazy-Zeroed
- Grafikkarte ohne 3D Unterstützung mit 8 MB Videoarbeitsspeicher

Betriebssystem:

- Microsoft Windows Server 2012 R2 Datacenter (64-bit)

Software:

- Microsoft SQL Server 2014 (64-bit)
- Neo4J 3.3.1

Modifizierte Einstellungen in Neo4J Datenbank:

- dbms.memory.heap.initial_size=2G (statt 512m)
- dbms.memory.heap.max_size=4G (statt 1G)

Methode

Es wurden verschiedene Testfälle erarbeitet, die zu vergleichen sind. Für jede Fragestellungen wurden eine oder mehrere Testabfragen, jeweils für rel.DB und graph.DB, erstellt. Ausgeführt wurden die Abfragen auf (Teil-)Daten der Datengrundlage.

In Neo4J wird die zeitl. Länge der Abfrage im Ergebnis ausgegeben, dies ist im SQL Server Management Studio nicht der Fall. Aus diesem Grund bestand jede Abfrage aus zwei zusätzlichen currentTimeStamp-Abfragen um die Differenz zwischen Beginn und Ende jeden Tests zu ermitteln.

Es wären gerne Implementierung von Graphspezifischen Abfragen in SQL erstellt worden, allerdings hat sich dies als zu aufwendiger Plan herausgestellt. Somit mussten Algorithmen wie PageRank, Connectness und allShortestPaths anderen, eher relationalen Abfragen weichen.

Die Werte wurden mithilfe eines Tabellenkalkulationsprogrammes (Microsoft Excel) gegenübergestellt.

[14] https://neo4j.com/graphgists/

Fragestellungen

Folgende Fragestellungen wurden in Hinblick auf Komplexität der Abfrage und Dauer der Durchführung in beiden Datenbanken gestellt:

Wie funktioniert der CSV Import?

Wie unterscheiden sich einfache Abfragen?

Wie unterscheiden sich kombinierte Abfragen?

Wie unterscheiden sich Graph-spezifische Abfragen?

ERGEBNISSE

Im folgenden Abschnitt werden die Ergebnisse der Abfragenvergleiche gegenübergestellt. Da die SQL- und Cypher Abfragen unter Umständen Längen von einer Seite Text oder mehr aufweisen können, sind in den Ergebnissen lediglich Auszüge davon zu lesen. Die kompletten Abfragen befinden sich im Anhang.

Imports von Daten aus einer CSV-Datei

Einerseits wurde hierbei überprüft wie schnell der Import einer CSV-Datei in der relationalen Datenbank SQL Server und der Graph Datenbank Neo4J. Andererseits wurde überprüft, ob die Menge der zu importierenden Daten eine Auswirkung auf den Import hat. Dazu wurde der Datensatz für jeden Fall beschnitten und in 10 Ausfertigungen verwendet. Jeder Teildatensatz bestand aus den ersten 200 bis 102400 Zeilen $(100 * 2^n \mid n = 1, \dots, 11)$ des Originaldatensatzes.

Unterschiede der Abfrage

SQL Server und Neo4J realisieren den Import auf unterschiedliche Art und Weise. Die SQL Abfrage ist zweigeteilt, sie beinhaltet einerseits die Erstellung der Tabelle und die Deklaration der Spaltentypen und andererseits den Import der CSV-Datei in diese Tabelle. Die Cypher Abfrage benötigt NoSQL-typisch keine Tabellenerstellung. Allerdings müssen die „Spalten“ der CSV-Datei den Eigenschaften der anzulegenden Entität mit Datentyp zugewiesen werden. Die Komplexität der beiden Abfragen ist daher sehr ähnlich.

In SQL:

```
CREATE TABLE [dbo].MENTIONS_IMPORT(
       [postid] [int] PRIMARY KEY,
       [sourceparam] [nvarchar](255) NULL,
       /* ... */
       );

BULK INSERT [dbo].MENTIONS_IMPORT
FROM '...\MSE_Mentions_102400.txt'
WITH (FIELDTERMINATOR = '\t', ROWTERMINATOR = '\n',
       FIRSTROW = 2, DATAFILETYPE = 'widechar')
```

In Cypher:

```
LOAD CSV WITH HEADERS FROM 'file:///MSE_Mentions_102400.txt' AS line
FIELDTERMINATOR '\t'
CREATE (a:POST
              {PostID: toString(line.externalpostid),
               SocialNetwork: toString(line.sourceparam),
               // ...
              })
```

Geschwindigkeit unterschiedlicher Importmengen

Die Erstellung der benötigten SQL Tabelle dauert beim ersten Mal 20 Millisekunden und wird für den Vergleich zu jedem CSV Import addiert. Es wurden jeweils drei Abfrage durchgeführt, in Tabelle 1 ist der Durchschnitt dieser Abfragen aufgelistet. Um auf eine eventuelle Linearität zu überprüfen, wurde berechnet wie viele Anfragen jeweils pro Nanosekunde durchgeführt werden.

Tabelle 1: Durchschnittliche Dauer der Abfragen

n	Dauer SQL	Anfragen pro ns	Dauer Cypher	Anfragen pro ns
200	0,030	151,667	0,133	666,667
400	0,042	105,833	0,176	440,000
800	0,052	65,000	0,223	278,750
1600	0,077	47,917	0,291	181,875
3200	0,141	44,063	0,399	124,688
6400	0,244	38,177	0,667	104,271
12800	0,447	34,896	1,209	94,427
25600	0,813	31,771	2,490	97,266
51200	1,589	31,042	4,882	95,352
102400	3,383	33,040	10,931	106,751

In beiden Abfragen scheint ein Overhead zu bestehen, der sich in den ersten Abfragen (bis n = 3200) bemerkbar macht. Bei größeren Abfragen steigt die Dauer linear an. Dabei ist der Import in den SQL Server mit 33 Zeilen pro Nanosekunde ungefähr dreimal schneller als der Import von 100 Zeilen pro Nanosekunde in Neo4J.

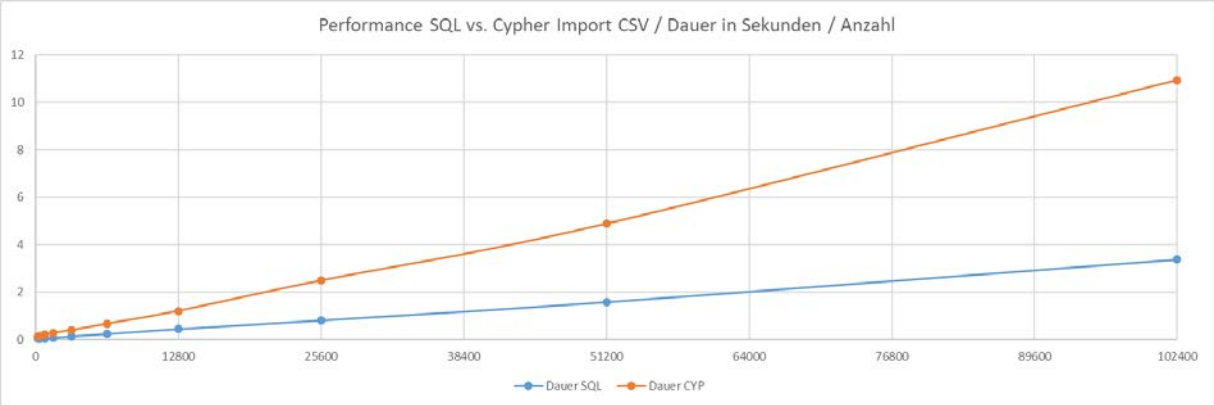


Abbildung 9: Performanz von SQL und Cypher beim CSV Import

Import des Datensatzes

Der Import der Daten ist für jede Datenbank optimiert worden und unterscheidet sich daher im Ablauf.

Für den Import in die SQL Datenbank wurde die CSV-Datei temporär komplett importiert, da diese als Tabelle im SQL Server sehr einfach weiterverarbeitet werden kann. Im Anschluss wurden die Daten aus der temporären Tabelle in die Entitäts- und Relationstabellen kopiert. Daraufhin wurden die Primär- und Fremdschlüssel erzeugt, um danach die Relationentabelle *Mentioned* mit den berechneten Werten zur Entitätstabelle *Profiles* zu befüllen.

Der Import in die Neo4J Datenbank verwendet die CSV Datei mehrmals, um die jeweils benötigten Daten zu importieren. Im ersten und zweiten Schritt werden die *Posts* und *Profile* importiert. Daraufhin erfolgt in vier Schritten die Erstellung der Relation *Published*. Im Anschluss werden die Daten für die Entität *Tags* importiert und die Relation *Talks_About* berechnet und mit Informationen befüllt. Danach werden alle möglichen Namen aus der Relation *Published* der Entität *Profile* als Liste hinzugefügt. Es folgt die zeitintensive Berechnung der Relation *Mentioned*.

Tabelle 2: Ablauf und Dauer des Datensatzimports

SQL	Dauer		Cypher	Dauer
Einfacher CSV Import	5,5 s		Posts importieren	23,1 s
In Tabellen verschieben	68,1 s		Profile importieren	16 s
Primär- und Fremdschlüssel	6,3 s		Publishes berechnen	165,2 s
Mentions berechnen	110,4 s		Tags importieren	79 s
			Talks_About berechnen	176,5 s
			Namen kopieren	2 s
			Mentions berechnen	26100 s
Gesamtdauer	**0,05 h**		**Gesamtdauer**	**7,38 h**

Der tabellarisch aufgebaute Inhalt der CSV Datei kann, wie im ersten Test schon beobachtet, schneller in die Datenbank importiert werden. Die Werte wurden hierbei nur einmal getestet, begründet auf der langen Kalkulation bei der Erstellung der Relation *Mentioned* mit Cypher. Die Verarbeitung der initial im Tabellenformat vorhandenen Informationen scheint performanter im SQL Server verarbeitet werden zu können als in Neo4J. Der Import dauerte in dieser Form mit Neo4J 140-mal länger. Wird die Berechnung der Mentions-Relation ignoriert, ist die SQL Datenbank mit 1,3 Minuten um den Faktor 4 schneller als die Neo4J Datenbank, welche 7,7 Minuten benötigt.

Einfache Abfragen

Im folgenden Abschnitt werden grundlegende SQL Abfragen durchgeführt, um zu überprüfen inwieweit sich die Abfragenkomplexität und die Dauer zwischen SQL Server und Neo4J unterscheiden. Die komplette Gegenüberstellung der erhobenen Daten befindet sich in Anhang C – Vergleichstabellen.

Zählen (Count)

Anzahl der Posts insgesamt

SQL Abfrage:
```
SELECT COUNT(*) FROM Posts
```

Cypher Abfrage:
```
MATCH (a:POST) RETURN COUNT(a)
```

Die Komplexität der Abfrage ist sehr ähnlich. Ein Unterschied besteht in der Geschwindigkeit der beiden Abfragen. In SQL dauert die Abfrage 21 ms, mit Cypher lediglich 1 ms. Da die Daten in Neo4J in *fixed-size record stores* gespeichert werden, muss lediglich die Größe der DB-Datei für Beiträge ermittelt werden. Dies ist um den Faktor 20 schneller als das Zählen der einzelnen Einträge.

Anzahl der Posts, die als Bild veröffentlicht wurden

SQL Abfrage:
```
SELECT COUNT WHERE contenttype = 'IMAGE'
```

Cypher Abfrage:
```
MATCH (a:POST) WHERE a.ContentType = 'IMAGE' RETURN COUNT(a)
```

In dieser Abfrage müssen i.G.z. ersten auch in Neo4J die Daten direkt ermittelt. In SQL wird das Ergebnis nach 39 ms gezeigt, mit Cypher dauert die Abfrage durchschnittlich 110 ms und somit fast 3x so lang. Die Komplexität beider Abfragen ist ähnlich.

Anzahl der Posts, die das Wort ‚Saw' enthalten

SQL Abfrage:
```
SELECT COUNT(*) FROM Posts WHERE postcontent like '%saw%'
```

Cypher Abfrage (a):
```
MATCH (a:POST)
WHERE toLower(a.PostContent) contains 'saw'
RETURN count(a)
```

Cypher Abfrage (b):
```
MATCH (a:POST)
WHERE a.PostContent contains 'saw'
OR a.PostContent contains 'Saw'
OR a.PostContent contains 'SAW'
RETURN count(a)
```

Die Abfrage, ob eine Person über Sägen geschrieben hat, erfordert einen Stringvergleich. In SQL funktioniert dies mit dem *LIKE* Operator und unterschiedet nicht in Groß- und Kleinschreibung. In Cypher wird stattdessen der Operator *CONTAINS* verwendet und Groß- von Kleinschreibung unterschieden. Der Vergleich des Strings kann somit durch Tests aller Varianten oder durch die Funktion *TOLOWER()* auf den Beitrag. Die Komplexität und Länge (bei b) ist dadurch leicht erhöht zur Abfrage in SQL. Die Geschwindigkeiten sind auf ähnlichem Niveau bei knapp unter einer Sekunde. Die schnellste Abfrage wurde mit Cypher und *TOLOWER()* durchgeführt.

Anzeigen (SELECT TOP)

Die ersten 1.000 Posts anzeigen

SQL Abfrage:
```
SELECT TOP 1000 * FROM Posts
```

Cypher Abfrage:
```
MATCH (a:POST) RETURN a.PostContent LIMIT 1000
```

Die ersten 10.000 Posts anzeigen

SQL Abfrage:
```
SELECT TOP 10000 * FROM Posts
```

Cypher Abfrage:
```
MATCH (a:POST) RETURN a.PostContent LIMIT 10000
```

Die ersten 100.000 Posts anzeigen

SQL Abfrage:
```
SELECT TOP 100000 * FROM Posts
```

Cypher Abfrage:
```
MATCH (a:POST) RETURN a.PostContent LIMIT 100000
```

Die Komplexität und Länge der beiden Anfragen ist ähnlich. Bei der Durchführung ist Neo4J 30-40% schneller, die Ermittlung der 100.000 Beiträge dauert mit SQL 2,1 Sekunden, mit Neo4J 1,5 Sekunden.

Dabei ist anzumerken, dass die Werte die Zeiten für die Berechnung ist. Die Darstellung der Informationen erfolgt im SQL Server Management Studio fast direkt. Die Darstellung der Informationen im Neo4J-Browser kann bei 100.000 Beiträgen mehrere Minuten in Anspruch nehmen.

Beide Anfragen verlangsamen sich mit erhöhtem Datensatz, SQL ermöglicht bei 1.000 Beiträgen eine Geschwindigkeit von ca. 100 pro Nanosekunde und verringert sich bei 10.000 und 100.000 Beiträgen auf 27 Beiträge/ns bzw. 21 Beiträge/ns. In Cypher ist diese Veränderung ähnlich, die Anzahl an

Beiträgen die pro Nanosekunde ermittelt werden können sind für 1.000, 10.000 und 10.000 Beiträge 57, 17 und 16.

Aggregatfunktionen (Avg, Sum, Max)

Zeige den durchschnittlichen Sentiment für jeden ContentType an

SQL Abfrage:
```
SELECT contenttype, avg(sentimentvalue) as AvgSentiment
FROM Posts GROUP BY contenttype ORDER BY AvgSentiment DESC
```

Cypher Abfrage:
```
MATCH (n:POST)
RETURN n.ContentType, avg(n.Sentiment) as AverageSentiment ORDER
BY AverageSentiment DESC
```

Die Komplexität ist erneut sehr ähnlich. Lediglich auf die Gruppierung kann in Cypher verzichtet werden, da diese automatisch erzeugt wird. Die SQL Abfrage wird in 58 Millisekunden durchgeführt und ist damit um den Faktor 4 schneller als die Cypher Abfrage, welche durchschnittlich 232 Millisekunden für die Durchführung benötigt.

Zeige die Summe der 100 meistgenannten Tags an

SQL Abfrage:
```
SELECT SUM([count]) as Summe
FROM (SELECT TOP 100 profile_mentioned, [count]
FROM Mentions ORDER BY [count] DESC) as a
```

Cypher Abfrage:
```
MATCH ()-[x:MENTIONED]->()
WITH x.ProfileTag as Profile, x.Count as Amount
ORDER BY Amount DESC LIMIT 100 RETURN SUM(Amount)
```

Die Geschwindigkeit der Kombination Zählen und Summieren wird im SQL Server 20-fach schneller als in Neo4J durchgeführt. Die SQL Abfrage ist nach 10 ms erfolgreich beendet, die Cypher Abfrage erst nach 236 ms. Die Abfragen selbst sind sich sehr ähnlich.

Zeige die Zeichenanzahl des längsten Beitrags an

SQL Abfrage:
```
SELECT max(len(postcontent)) as PostLength FROM Posts
```

Cypher Abfrage:
```
MATCH (n:POST) RETURN max(size(n.PostContent))
```

Die Kombination der Aggregatfunktion MAX und LEN ist in SQL ebenfalls schneller. Mit 38 ms ist die Geschwindigkeit um den Faktor 8 höher als die benötigten 328 ms in Cypher. Die Komplexität ist dagegen sehr ähnlich.

Alle Aggregatfunktionen können, wie in der Einführung zum Thema schon beschrieben und durch diese Tests bestätigt, mit SQL schneller zu Ende geführt als mit Cypher.

Verwendung eines Indexes

Im Folgenden wird die Auswirkung eines Indexes in beiden Datenbanken überprüft. Die bereits durchgeführte Abfrage nach dem Substring „saw“ in den Beitragstexten wird nach der Erstellung eines Indexes für diese Spalte/Eigenschaft wiederholt und verglichen.

Anlegen der Indexe

SQL Abfrage:
```
CREATE INDEX postcontentindex on Posts (externalpostid)
INCLUDE (postcontent);
```

Cypher Abfrage:
```
CREATE INDEX ON :POST(PostContent)
```

Die Abfrage in Cypher ist kürzer und leichter verständlich als die SQL Abfrage.

Abfragen

SQL Abfrage: SELECT COUNT(*) FROM Posts WHERE postcontent like '%saw%'

Cypher Abfrage (a):
```
MATCH (a:POST)
WHERE toLower(a.PostContent) contains 'saw'
RETURN count(a)
```

Cypher Abfrage (b):
```
MATCH (a:POST)
WHERE a.PostContent contains 'saw'
OR a.PostContent contains 'Saw'
OR a.PostContent contains 'SAW'
RETURN count(a)
```

Die Dauer der Abfrage im SQL Server hat sich durch den Index um 15% verkürzt.

Die *TOLOWER()* verwendende Abfrage hat keine Geschwindigkeitssteigerungen erfahren. In der Neo4j Cypher Refcard (Version 3.3) ist dazu vermerkt, das *TOLOWER()* den Index nicht verwendet. Dies kann durch den Test bestätigt werden. Die zweite Cypherabfrage kann durch den Index in 363 ms statt 900 ms durchgeführt werden, dies bedeutet eine Verkürzung der Dauer von ungefähr 60%. Die Cypher Abfrage ist damit auch mehr als doppelt so schnell wie die SQL Abfrage.

Kombinierte Abfrage

Der nächste Block beschäftigt sich zusätzlich zum Vergleich der Abfragezwischen SQL und Cypher damit, inwieweit eine kombinierte Abfrage zweigeteilt oder in einem Durchgang mit höherer Performanz durchlaufen werden kann.

Suche die 10 Person mit den meisten Posts

SQL Abfrage:
```
SELECT TOP 10 profilename, count(*) as Anzahl
FROM Publishes GROUP BY profilename ORDER BY Anzahl DESC
```

Cypher Abfrage:
```
MATCH ()-[pub:PUBLISHED]->(post)
RETURN pub.ProfileName, count(post) as PostsPublished
ORDER BY PostsPublished DESC LIMIT 10
```

Die Abfrage unterscheidet sich lediglich durch die Art, wie in Cypher eine Relation abgefragt wird. Die Abfrage in SQL benötigt 75 ms, mit Cypher dauert es 279 ms bis zum Ergebnis.

Suche die 10 Personen, die am häufigsten genannt wurden

SQL Abfrage:
```
SELECT TOP 10 profile_mentioned, sum([Count]) as Mentioned
FROM Mentions WHERE mentionedprofileid IS NOT NULL
AND mentionedprofileid != externalprofileid
GROUP BY profile_mentioned ORDER BY Mentioned DESC
```

Cypher Abfrage:
```
MATCH (q:PROFILE)-[x:MENTIONED]->(p:PROFILE)
WHERE q <> p
RETURN p.Names[0] as Name, sum(x.Count) as TimesMentioned
ORDER BY TimesMentioned DESC LIMIT 10
```

Bei der Abfrage nach den am häufigsten genannten Profilen ist ebenfalls SQL schneller. Mit 12 ms um den Faktor 9 schneller als die gleiche Abfrage in Cypher.

Suche die 10 Personen mit dem größten Mentioned/Published Ratio (ohne eigene Mentions)

SQL Abfrage:

```
SELECT profilename, hasPublished, isMentioned,
CAST(isMentioned AS float)/CAST(hasPublished as float) as Ratio
FROM (SELECT externalprofileid, profilename,
             count(*) as hasPublished
      FROM Publishes
      GROUP BY externalprofileid, profilename) as pubs
      JOIN
    (SELECT mentionedprofileid, sum([Count]) as isMentioned
      FROM Mentions WHERE mentionedprofileid IS NOT NULL
      AND mentionedprofileid != externalprofileid
      GROUP BY mentionedprofileid) as mens
    ON pubs.externalprofileid = mens.mentionedprofileid
ORDER BY Ratio DESC
```

Cypher Abfrage:

```
MATCH (pr0:PROFILE)-[men:MENTIONED]->(pr1:PROFILE)-
[pub:PUBLISHED]->()
WHERE pr0 <> pr1
WITH pr1.Names[0] as Name, count(pub) as hasPublished,
sum(men.Count) as isMentioned
RETURN Name, hasPublished, isMentioned,
toFloat(isMentioned)/toFloat(hasPublished) as Ratio
ORDER BY Ratio DESC LIMIT 10
```

Die kombinierte Abfrage mit der zusätzlichen Berechnung des Verhältnisses zwischen Menge der Erwähnungen und Menge der eigenen Veröffentlichungen wird mit SQL in der gleichen Dauer beendet wie die Summe der beiden Einzelabfragen. In Cypher ist dies nicht der Fall, die kombinierte Abfrage benötigt durchschnittlich 5,5 Sekunden und ist damit ungefähr 14x langsamer als die Ausführung der Summe beider Einzelabfragen.

Graph Abfragen

Im Folgenden werden Abfragen auf den Graph direkt gemacht und durch den Neo4J-Browser auch visuell aufbereitet. Diese Funktionalität ist im SSMS nicht vorhanden, dennoch werden hier die Zeiten und die Komplexität der Anfragen verglichen.

Konversationen finden

SQL Abfrage:

```
SELECT TOP 25 m1.externalprofileid,
       m1.[Count] as TimesMentioned_12,
       m2.externalprofileid,
       m2.[Count] as TimesMentioned_23,
       m3.externalprofileid
FROM Mentions m1
JOIN Mentions m2
ON m1.mentionedprofileid = m2.externalprofileid

AND m1.externalprofileid != m2.externalprofileid

JOIN Mentions m3
ON m2.mentionedprofileid = m3.externalprofileid
AND m2.externalprofileid != m3.externalprofileid
AND m1.externalprofileid = m3.externalprofileid

GROUP BY m1.externalprofileid,
```

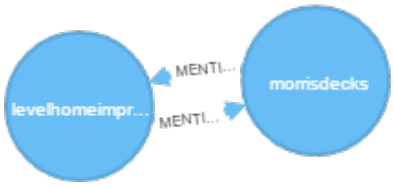


Abbildung 10: Konversation mit zwei sich gegenseitig erwähnenden Profilen

```
        m2.externalprofileid,
        m3.externalprofileid,
        m1.[Count], m2.[Count]
```

Profil1	TimesMentioned_12	Profil2	TimesMentioned_21	Profil1
2219989103	2	1096354783	2	2219989103
2219989103	2	2195093399	7	2219989103
3027653237	1	2206631478	4	3027653237
3156403058	8	2203658604	1	3156403058
3212137617	1	8376527	1	3212137617
1501066719	1	8376527	1	1501066719
659111219	1	1125058208	1	659111219
19722970	5	1034209018	2	19722970
329818995	1	927520689	2	329818995
2219989103	2	927520689	8	2219989103
4036976573	1	3156403058	1	4036976573
41779872	1	784674711	1	41779872
329818995	1	8047359	1	329818995
19722970	5	1034209018	1	19722970

Abbildung 11: Konversationen der SQL Abfrage in Tabellenform

Cypher Abfrage:

```
MATCH p = (u1:PROFILE)-[:MENTIONED]->(u2:PROFILE)-[:MENTIONED]->(u1:PROFILE)
WHERE u1<>u2
      AND NOT (u1:PROFILE)-[:MENTIONED]->(u1:PROFILE)
      AND NOT (u2:PROFILE)-[:MENTIONED]->(u2:PROFILE)
RETURN p LIMIT 25
```

Die Abfrage in Cypher ist durch die Verwendung des Pfades p im Vergleich zur SQL Abfrage sehr verkürzt und erlaubt aufgrund der geringeren Duplikationen weniger Fehler bei der Bearbeitung. Auch die Ausführungszeit ist in diesem Beispiel sehr unterschiedlich. Die SQL Abfrage benötigt 92 Sekunden bis zum Ende der Durchführung, die Cypher Abfrage ist bereits nach 10 Millisekunden beendet und somit 9200-fach schneller.

Profile, die sich selbst im Beitrag nennen

SQL Abfrage:

```
SELECT TOP 25 profile_mentioned, [Count]
FROM Mentions
WHERE externalprofileid = mentionedprofileid
ORDER BY [Count] DESC
```

Cypher Abfrage:

```
MATCH p = (u:PROFILE)-[x:MENTIONED]->(u:PROFILE)
RETURN p
ORDER BY x.Count DESC LIMIT 25
```

Abbildung 12: Profil mit Selbsterwähnung

Die Cypher Abfrage verwendet, ähnlich zum vorigen Beispiel, einen Pfad um dem Muster zu entsprechen. Die SQL Abfrage kann sehr vereinfacht werden, da alle benötigten Informationen in der Tabelle Mentions ohne *JOINS* abgefragt werden können. Dadurch dauert die Abfrage in SQL lediglich 6 ms im Gegensatz zur Cypher Abfrage, die 142 Millisekunden benötigt.

Profile im Umfeld eines Profiles

Als Startprofil wird das Profil mit der ID *588451086* ausgewählt, der Autor hat eine übersichtlich Anzahl an 23 Beiträgen auf Twitter erstellt und dabei 88 Personen erwähnt. Es werden alle Profile gesucht, die ausgehend vom Initial Profil erwähnt werden.

Anzeige aller Profile innerhalb einer Distanz von 2

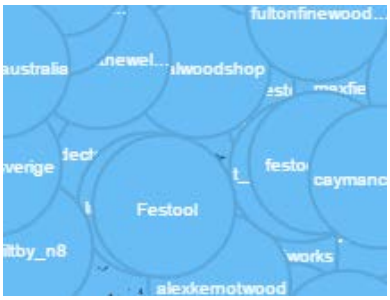


Abbildung 13: Profile in naher Distanz

SQL Abfrage:

```
SELECT DISTINCT mentionedprofileid as Profil, Distance
FROM (SELECT mentionedprofileid, 1 as Distance
       FROM Mentions
       WHERE externalprofileid = '588451086') as t1
     UNION
     (SELECT m2.mentionedprofileid, 2 as Distance
FROM Mentions m1 JOIN Mentions m2
ON m1.mentionedprofileid = m2.externalprofileid
WHERE m1.externalprofileid = '588451086')
```

Cypher Abfrage:

```
MATCH p = (x:PROFILE {ProfileID: '588451086'})-[:MENTIONED*0..2]->(y:PROFILE) RETURN
DISTINCT y, length(p)
```

Anzeige aller Profile innerhalb einer Distanz von 5

SQL Abfrage:

Zu betrachten im Anhang.

Cypher Abfrage:

```
MATCH p = (x:PROFILE {ProfileID: '588451086'})-[:MENTIONED*0..5]->(y:PROFILE) RETURN
DISTINCT y, length(p)
```

Die SQL Abfrage ist mit ~ 40 Zeilen zu lang für die Darstellung auf dieser Seite. Für jede Erhöhung der Distanz wird in der Abfrage ein zusätzliches *SELECT* Statement mit *UNION* eingebunden. Das SELECT Statement besteht dabei jeweils aus einem zusätzlichen JOIN. Somit besteht das fünfte zu vereinende SELECT Statement aus einem fünffachen *JOIN* der Tabelle *Mentions*.

Im Gegensatz dazu ist der Unterschied der Abfrage in Cypher lediglich die zu durchsuchende Reichweite der Relation *Mentioned* durch 0-5 zu ersetzen.

Die Geschwindigkeit der vielfach verjointen SQL Abfrage ist trotz der Länge mit 136 Millisekunden für die 2er Distanz und 16,5 Sekunden für die 5er Distanz doppelt so schnell wir die Cypher Abfragen mit 350 Millisekunden und 32,5 Sekunden.

Anzeige aller Profile von denen das vorgegebene Profil in maximaler Distanz von 2 ist.

Im Gegensatz zur vorigen Abfrage sind nun alle Profile der Startpunkt und es wird gesucht, bei welchen dieser Profile in einer vorgegeben Entfernung das gewählte Profil erwähnt wird.

SQL Abfrage:

```
SELECT DISTINCT externalprofileid as Profil, Distance
FROM
       (SELECT externalprofileid, 1 as Distance
        FROM Mentions
        WHERE mentionedprofileid = '588451086') as t1
       UNION
       (SELECT m1.externalprofileid, 2 as Distance
        FROM Mentions m1
        JOIN Mentions m2 ON m1.mentionedprofileid = m2.externalprofileid
        WHERE m2.mentionedprofileid = '588451086')
```

Cypher Abfrage:

```
MATCH p = (x:PROFILE {ProfileID: '588451086'})<-[:MENTIONED*0..2]-(y:PROFILE) RETURN
DISTINCT y, length(p)
```

Anzeige aller Profile von denen das vorgegebene Profil in maximaler Distanz von 5 ist

SQL Abfrage:

Zu betrachten im Anhang

Cypher Abfrage:

```
MATCH p = (x:PROFILE {ProfileID: '588451086'})<-[:MENTIONED*0..5]-(y:PROFILE) RETURN
DISTINCT y, length(p)
```

Die SQL Abfrage für die 5er Distanz ist ebenfalls sehr lang und kann im Anhang betrachtet werden. Der Unterschied zur Abfrage in die entgegen gesetzte Richtung ist die Austauschung der *externalprofileid* und der *mentionedprofileid* in allen Vorkommen. Die Änderung der Cypher Abfrage besteht lediglich aus der Veränderung der Richtung des Pfeils in der Pfadabfrage.

Die Performanz der SQL Abfrage ist ebenfalls größer als die der Cypher Abfrage. Die Profile in Distanz 2 können bei der SQL Abfrage in 162 Millisekunden ermittelt werden, die Profile in Distanz 5 in 17,8 Sekunden. Die Abfrage in Cypher kann die Profile der Distanz 2 in 400 Millisekunden und die Profile der Distanz 5 in ca. 50 Sekunden ermitteln. Trotz der hohen Verschachtelung durch eine hohe Anzahl an JOINS ist die Geschwindigkeit mit SQL mehr als doppelt so hoch.

FAZIT

Das Ergebnis der Analysen zeigt bezüglich der Performanz, dass die gewählten Abfragen mit dem SQL *Server* in vielen Fällen in weniger Zeit durchgeführt werden können. Der Grund für die Geschwindigkeit sind allerdings die Abfragen selbst gewesen. So sind hauptsächliche Abfragen verglichen worden, deren Erstellung in beiden Datenbanken möglich ist. Gerne wären weitere graphspezifische Beispiele, wie die Berechnung des *PageRanks*, der *Betweenness* und *allShortestPath* in die Hausarbeit aufgenommen worden. In hier nicht weiter beschriebenen Tests liefen diese in der Graphdatenbank sehr performant ab, die zugehörigen relationale Abfragen zu Erstellen hätte allerdings den Umfang dieser Hausarbeit weiter erhöht, teilweise sind die relationalen Entsprechungen der Abfragen nicht möglich. Außerdem sollte berücksichtigt werden, dass der Autor bereits mehrere Jahre SQL Erfahrungen besitzt und die Abfragesprache *Cypher* lediglich im Rahmen dieser Arbeit verwendet wurde.

Im Gegensatz zur Geschwindigkeit ist die Erstellung der Abfragen – in dieser Arbeit in *Cypher* – bei graphbasierten Datenbanken weniger komplex. Die Abfragen sind durch weniger Wiederholungen und die daraus resultierende Übersichtlichkeit wartungsfreundlicher und auch sehr gut für Einsteiger geeignet.

Diese Arbeit ist ein weiteres Beispiel dafür, dass es nicht möglich ist, die beste Datenbank für alle Anwendungsfälle zu finden. Jeder Datenbanktyp, *relational* oder *graphbasiert*, aber auch *Dokument* oder *Key-Value* ist für bestimmte Anwendungsfälle die geeignetste. Diese sollte dafür bei jeder Anforderungsanalyse ermittelt werden, anstatt sich auf eine Datenbank für alles zu verlassen.

ANHANG A – SQL BEFEHLE

Import einer CSV Datei

```
CREATE TABLE [dbo].Temporary_Flat_Import (
	[postid] [int] PRIMARY KEY,
	[externalpostid] [nvarchar](max) NULL,
	[contenttype] [nvarchar](30) NULL,
	[posttype] [nvarchar](30) NULL,
	[posturi] [nvarchar](500) NULL,
	[publicationdate] [datetime2](7) NULL,
	[externalprofileid] [nvarchar](255) NULL,
	[profilename] [nvarchar](100) NULL,
	[displayname] [nvarchar](100) NULL,
	[profileicon] [nvarchar](255) NULL,
	[location] [nvarchar](100) NULL,
	[locationlatitude] [nvarchar](100) NULL,
	[locationlongitude] [nvarchar](100) NULL,
	[locationquadkey] [nvarchar](255) NULL,
	[sourceparam] [nvarchar](100) NULL,
	[postlanguage] [nvarchar](100) NULL,
	[normalscore] [int] NULL,
	[provider] [nvarchar](30) NULL,
	[providerscore] [float] NULL,
	[sentimentvalue] [float] NULL,
	[refpost_profilename] [nvarchar](100) NULL,
	[refpost_profileid] [int] NULL,
	[refpost_profileicon] [nvarchar](255) NULL,
	[refpost_profiledisplayname] [nvarchar](100) NULL,
	[refpost_externalid] [nvarchar](max) NULL,
	[cleanpostcontent] [nvarchar](max) NULL,
	[contained_tags] [nvarchar](2047) NULL,
	[contained_profiles] [nvarchar](2047) NULL);

BULK INSERT [dbo].Temporary_Flat_Import
FROM '...\MSE_Mentions_102400.txt'
WITH (FIELDTERMINATOR = '\t', ROWTERMINATOR = '\n', FIRSTROW = 2, DATAFILETYPE =
'widechar')
```

Import des Datensatzes

Ausgehend vom bereits bestehenden CSV Import (Siehe Anhang A – SQL Befehle: Import einer CSV Datei)

```
SELECT DISTINCT [externalpostid], [contenttype], [posttype], [posturi],
        [publicationdate], [postlanguage], [sentimentvalue],
        [cleanpostcontent] as postcontent
       INTO [TAXOTest].[dbo].[Posts]
       FROM [TAXOTest].[dbo].[Temporary_Flat_Import]

SELECT DISTINCT [externalprofileid], [sourceparam] as SocialNetwork
       INTO [TAXOTest].[dbo].[Profiles]
       FROM [TAXOTest].[dbo].[Temporary_Flat_Import]

SELECT [externalpostid], [externalprofileid], [publicationdate],
        [profilename], [displayname], [profileicon],
        [location], [locationlatitude], [locationlongitude]
       INTO [TAXOTest].[dbo].[Publishes]
       FROM [TAXOTest].[dbo].[Temporary_Flat_Import]

SELECT [externalprofileid], [tag], count(tag) as [Count]
       INTO [TAXOTest].[dbo].[Tags]
       FROM (SELECT A.[externalprofileid],
                    Split.a.value('.', 'VARCHAR(500)') AS tag
              FROM (SELECT [externalprofileid],
                     CAST ('<M>' + REPLACE([contained_tags],
                      ' ', '</M><M>') + '</M>' AS XML) AS String
                     FROM [TAXOTest].[dbo].[Temporary_Flat_Import]
                     WHERE contained_tags != '') AS A
              CROSS APPLY String.nodes ('/M') AS Split(a)) as x
GROUP BY externalprofileid, tag
ORDER BY [COUNT] DESC

SELECT [externalprofileid], [profile_mentioned], [Count]
       INTO [TAXOTest].[dbo].[temp_Mentions]
       FROM (SELECT externalprofileid,
                     profile as profile_mentioned,
count(profile) as [Count]
              FROM (SELECT A.[externalprofileid],
                            Split.a.value('.', 'VARCHAR(500)') AS profile
                     FROM (SELECT [externalprofileid],
                            CAST ('<M>' + REPLACE([contained_profiles],
                            ' ', '</M><M>') + '</M>' AS XML) AS String
                            FROM [TAXOTest].[dbo].[Temporary_Flat_Import]
                            WHERE contained_profiles != ''
 AND contained_profiles != '@' ) AS A
                     CROSS APPLY String.nodes ('/M') AS Split(a)) as x
       GROUP BY externalprofileid, profile) as y
       WHERE LEFT(profile_mentioned, 1) = '@' AND LEN(profile_mentioned) > 1
       ORDER BY profile_mentioned
```

Anzeige aller Profile innerhalb einer Distanz von 5

```
SELECT DISTINCT mentionedprofileid as Profil, Distance
FROM
       (SELECT mentionedprofileid, 1 as Distance
        FROM Mentions
        WHERE externalprofileid = '588451086') as t1
       UNION
       (SELECT m2.mentionedprofileid, 2 as Distance
        FROM Mentions m1 JOIN Mentions m2
        ON m1.mentionedprofileid = m2.externalprofileid
        WHERE m1.externalprofileid = '588451086')
       UNION
       (SELECT m3.mentionedprofileid, 3 as Distance
        FROM Mentions m1
        JOIN Mentions m2
        ON m1.mentionedprofileid = m2.externalprofileid
        JOIN Mentions m3
        ON m2.mentionedprofileid = m3.externalprofileid
        WHERE m1.externalprofileid = '588451086')
       UNION
       (SELECT m4.mentionedprofileid, 4 as Distance
        FROM Mentions m1
        JOIN Mentions m2
        ON m1.mentionedprofileid = m2.externalprofileid
        JOIN Mentions m3
        ON m2.mentionedprofileid = m3.externalprofileid
        JOIN Mentions m4
        ON m3.mentionedprofileid = m4.externalprofileid
        WHERE m1.externalprofileid = '588451086')
       UNION
       (SELECT m5.mentionedprofileid, 5 as Distance
        FROM Mentions m1
        JOIN Mentions m2
        ON m1.mentionedprofileid = m2.externalprofileid
        JOIN Mentions m3
        ON m2.mentionedprofileid = m3.externalprofileid
        JOIN Mentions m4
        ON m3.mentionedprofileid = m4.externalprofileid
        JOIN Mentions m5
        ON m4.mentionedprofileid = m5.externalprofileid
        WHERE m1.externalprofileid = '588451086')
```

Anzeige aller Profile von denen das vorgegebene Profil in maximaler Distanz von 5 ist

```
SELECT DISTINCT externalprofileid as Profil, Distance
FROM
       (SELECT externalprofileid, 1 as Distance
        FROM Mentions
        WHERE mentionedprofileid = '588451086') as t1
       UNION
       (SELECT m1.externalprofileid, 2 as Distance
        FROM Mentions m1
        JOIN Mentions m2 ON m1.mentionedprofileid = m2.externalprofileid
        WHERE m2.mentionedprofileid = '588451086')
       UNION
       (SELECT m1.externalprofileid, 3 as Distance
        FROM Mentions m1
        JOIN Mentions m2 ON m1.mentionedprofileid = m2.externalprofileid
        JOIN Mentions m3 ON m2.mentionedprofileid = m3.externalprofileid
        WHERE m3.mentionedprofileid = '588451086')
       UNION
       (SELECT m1.externalprofileid, 4 as Distance
        FROM Mentions m1
        JOIN Mentions m2 ON m1.mentionedprofileid = m2.externalprofileid
        JOIN Mentions m3 ON m2.mentionedprofileid = m3.externalprofileid
        JOIN Mentions m4 ON m3.mentionedprofileid = m4.externalprofileid
        WHERE m4.mentionedprofileid = '588451086')
       UNION
       (SELECT m1.externalprofileid, 5 as Distance
        FROM Mentions m1
        JOIN Mentions m2 ON m1.mentionedprofileid = m2.externalprofileid
        JOIN Mentions m3 ON m2.mentionedprofileid = m3.externalprofileid
        JOIN Mentions m4 ON m3.mentionedprofileid = m4.externalprofileid
        JOIN Mentions m5 ON m4.mentionedprofileid = m5.externalprofileid
        WHERE m5.mentionedprofileid = '588451086')
```

ANHANG B – CYPHER BEFEHLE

Import des Datensatzes

```
// Q01-IMPORT Posts
USING PERIODIC COMMIT 1000
LOAD CSV WITH HEADERS FROM 'file:///MSE_Mentions_Full.txt' AS line
FIELDTERMINATOR '\t'
CREATE (a:POST
       { PostId: toString(line.externalpostid),
         ContentType: toString(line.contenttype),
         PostType: toString(line.posttype),
         PostUri: toString(line.posturi),
         PublicationDate: toString(line.publicationdate),
         Language: toString(line.postlanguage),
         PostContent: toString(line.cleanpostcontent),
         Sentiment: toFloat(line.sentimentvalue),
         TagsMentioned: split(line.contained_tags, " "),
         ProfilesMentioned: split(replace(line.contained_profiles,"@",""),"  ")
       })
```

```
// Q02-IMPORT Profiles
USING PERIODIC COMMIT 1000
LOAD CSV WITH HEADERS FROM 'file:///MSE_Mentions_Copy.txt' AS line
FIELDTERMINATOR '\t'
MERGE (b:PROFILE
       { ProfileID: toString(line.externalprofileid),
         SocialNetwork: toString(line.sourceparam) })
```

```
// Q03-CREATE Published 1/4
USING PERIODIC COMMIT 1000
LOAD CSV WITH HEADERS FROM 'file:///MSE_Mentions_Copy.txt' AS line
FIELDTERMINATOR '\t'
MATCH (a:POST { PostId: toString(line.externalpostid)})
MATCH (b:PROFILE {   ProfileID: toString(line.externalprofileid),
                     SocialNetwork: toString(line.sourceparam) })
MERGE (b)-[c:PUBLISHED {    PublicationDate: toString(line.publicationdate),
                            ProfileIcon: toString(line.profileicon)}]->(a)
```

```
// Q04-CREATE Published 2/4
USING PERIODIC COMMIT 1000
LOAD CSV WITH HEADERS FROM 'file:///MSE_Mentions_Copy.txt' AS line
FIELDTERMINATOR '\t'
WITH line
WHERE line.location IS NOT NULL
MATCH (a:POST { PostId: toString(line.externalpostid)})
MATCH (b:PROFILE {  ProfileID: toString(line.externalprofileid),
                    SocialNetwork: toString(line.sourceparam) })
MATCH (b)-[c:PUBLISHED {   PublicationDate: toString(line.publicationdate),
                           ProfileIcon: toString(line.profileicon)}]->(a)
SET c.Country = toString(line.location)
SET c.Latitude = toString(line.locationlatitude)
SET c.Longitude = toString(line.locationlongitude)
```

```
// Q05-CREATE Published 3/4
USING PERIODIC COMMIT 1000
LOAD CSV WITH HEADERS FROM 'file:///MSE_Mentions_Copy.txt' AS line
FIELDTERMINATOR '\t'
WITH line
WHERE line.displayname IS NULL
MATCH (a:POST { PostId: toString(line.externalpostid)})
MATCH (b:PROFILE {  ProfileID: toString(line.externalprofileid),
                    SocialNetwork: toString(line.sourceparam) })
MATCH (b)-[c:PUBLISHED {   PublicationDate: toString(line.publicationdate),
                           ProfileIcon: toString(line.profileicon)}]->(a)
SET c.ProfileName = toString(line.profilename)
```

```
// Q06-CREATE Published 4/4
USING PERIODIC COMMIT 1000
LOAD CSV WITH HEADERS FROM 'file:///MSE_Mentions_Copy.txt' AS line
FIELDTERMINATOR '\t'
WITH line
WHERE line.displayname <> ""
MATCH (a:POST { PostId: toString(line.externalpostid)})
MATCH (b:PROFILE {  ProfileID: toString(line.externalprofileid),
                    SocialNetwork: toString(line.sourceparam) })
MATCH (b)-[c:PUBLISHED {   PublicationDate: toString(line.publicationdate),
                           ProfileIcon: toString(line.profileicon)}]->(a)
SET c.DisplayName = toString(line.displayname)
SET c.ProfileName = toString(reverse(split(line.profilename,"@"))[0])
```

```
// Q07-IMPORT Tags
USING PERIODIC COMMIT 1000
LOAD CSV WITH HEADERS FROM 'file:///MSE_Mentions_Copy.txt' AS line
FIELDTERMINATOR '\t'
FOREACH (value IN split(line.contained_tags, " ") | CREATE (t:TAG {Tag: value}))
```

```
// Q08-CREATE Talks_About
USING PERIODIC COMMIT 1000
LOAD CSV WITH HEADERS FROM 'file:///MSE_Mentions_Copy.txt' AS line
FIELDTERMINATOR '\t'
WITH line.externalprofileid as externalprofileid, line.sourceparam as sourceparam,
split(line.contained_tags, " ") as tags

MATCH (b:PROFILE
       { ProfileID: toString(externalprofileid),
         SocialNetwork: toString(sourceparam) })

MATCH (t:TAG) WHERE t.Tag in tags

FOREACH (value IN tags |
       MERGE (b)-[x:TALKS_ABOUT]->(t)
       ON CREATE SET x.Count = 1
       ON MATCH SET x.Count = x.Count + 1)
```

```
// Q09-COLLECT Names to Profile (2s)
MATCH (a:PROFILE)-[b:PUBLISHED]->()
WITH a, COLLECT(DISTINCT b.DisplayName) as dispnams,
  COLLECT(DISTINCT b.ProfileName) as profnams
SET a.Names = dispnams + profnams
```

```
// Q10-CREATE Mentioned (26100s)
MATCH (a:PROFILE)-[:PUBLISHED]-(b:POST), (c:PROFILE)
WHERE b.ProfilesMentioned IS NOT NULL
WITH a, c, FILTER(n in b.ProfilesMentioned WHERE n in c.Names)[0] as match
WHERE match <> []
MERGE (a)-[f:MENTIONED {ProfileTag: match}]->(c)
ON CREATE SET f.Count = 1
ON MATCH SET f.Count = f.Count + 1
```

ANHANG C – VERGLEICHSTABELLEN

Tabelle 3: Dauer des CSV Imports mit SQL

n	Create	T1	T2	T3	Dauer SQL	1/ns
200	0,020	0,010	0,014	0,007	0,030	151,667
400	0,020	0,020	0,027	0,020	0,042	105,833
800	0,020	0,033	0,033	0,030	0,052	65,000
1600	0,020	0,060	0,050	0,060	0,077	47,917
3200	0,020	0,137	0,116	0,110	0,141	44,063
6400	0,020	0,280	0,206	0,187	0,244	38,177
12800	0,020	0,423	0,437	0,420	0,447	34,896
25600	0,020	0,773	0,797	0,810	0,813	31,771
51200	0,020	1,694	1,477	1,537	1,589	31,042
102400	0,020	3,763	3,220	3,107	3,383	33,040

Tabelle 4: Dauer des CSV Imports mit Cypher

n	T1	T2	T3	Dauer CYP	1/ns
200	0,139	0,130	0,131	0,133	666,667
400	0,204	0,160	0,164	0,176	440,000
800	0,219	0,231	0,219	0,223	278,750
1600	0,289	0,289	0,295	0,291	181,875
3200	0,412	0,378	0,407	0,399	124,688
6400	0,628	0,635	0,739	0,667	104,271
12800	1,228	1,143	1,255	1,209	94,427
25600	2,508	2,510	2,452	2,490	97,266
51200	4,919	4,874	4,853	4,882	95,352
102400	12,532	10,244	10,018	10,931	106,751

Tabelle 5: Dauer für das Zählen

	SQL in s				Cypher in s			
	T1	T2	T3	Ø Dauer	T1	T2	T3	Ø Dauer
Zähle alle Posts	0,016	0,024	0,023	**0,021**	0,002	0,001	0,001	**0,001**
Zähle IMAGE Posts	0,040	0,037	0,040	**0,039**	0,114	0,108	0,107	**0,110**
Zähle 'saw' Posts	0,953	0,920	1,050	**0,974**	0,907	0,753	0,745	**0,802**
					0,880	0,983	0,838	**0,900**

Tabelle 6: Dauer für das Anzeigen

	SQL in s				Cypher in s			
	T1	T2	T3	Ø Dauer	T1	T2	T3	Ø Dauer
Zeige 1.000 Beiträge	0,103	0,106	0,110	**0,106**	0,075	0,040	0,055	**0,057**
Zeige 10.000 Beiträge	0,257	0,293	0,257	**0,269**	0,171	0,178	0,165	**0,171**
Zeige 100.000 Beiträge	2,086	2,160	2,124	**2,123**	1,582	1,605	1,495	**1,561**

Tabelle 7: Dauer für Aggregatfunktionen

	SQL in s				Cypher in s			
	T1	T2	T3	Ø Dauer	T1	T2	T3	Ø Dauer
Ø Sentiments / Type	0,067	0,036	0,070	**0,058**	0,251	0,225	0,221	**0,232**
Summe top Mentions	0,013	0,007	0,010	**0,010**	0,253	0,228	0,227	**0,236**
Längste Beitragslänge	0,037	0,040	0,036	**0,038**	0,362	0,320	0,303	**0,328**

Tabelle 8: Unterschiede der Dauer mit Index

	SQL in s				Cypher in s			
	T1	T2	T3	Ø Dauer	T1	T2	T3	Ø Dauer
Zähle 'saw' Posts (Wdh.)	0,953	0,920	1,050	**0,974**	0,907	0,753	0,745	**0,802**
					0,880	0,983	0,838	**0,900**
Erstelle Index	1,850				0,169			
Zähle 'saw' Posts	0,923	0,740	0,790	**0,818**	0,896	0,702	0,719	**0,772**
					0,382	0,348	0,359	**0,363**

Tabelle 9: Kombinierte Abfragen

	SQL in s				Cypher in s			
	T1	T2	T3	Dauer	T1	T2	T3	Dauer
Top10 Posts	0,066	0,073	0,087	0,075	0,291	0,27	0,277	0,279
Top10 Mentions	0,013	0,01	0,013	0,012	0,113	0,109	0,109	0,110
Top10 Ratio	0,083	0,097	0,083	0,088	5,451	5,69	5,52	5,554

Tabelle 10: Graphspezifische Abfragen

	SQL in s				Cypher in s			
	T1	T2	T3	Ø Dauer	T1	T2	T3	Ø Dauer
Top25 Konversations	91,530	93,584	90,540	**91,885**	0,011	0,01	0,01	**0,010**
Top25 Self Mentioner	0,003	0,007	0,007	**0,006**	0,152	0,136	0,139	**0,142**
Nachbarn d<=2	0,136	0,134	0,137	**0,136**	0,36	0,36	0,32	**0,347**
Nachbarn d<=5	19,603	14,740	14,650	**16,331**	29,773	34,465	33,406	**32,548**
istNachbar d<=2	0,177	0,156	0,154	**0,162**	0,370	0,380	0,450	**0,400**
istNachbar d<=5	17,596	17,893	17,807	**17,765**	54,580	50,717	45,621	**50,306**

LITERATURVERZEICHNIS

Boyd, Ryan. 2015. *RDBMS to Graphs. Harnessing the Power of the Graph.*

Goldstein, Jackie. 2005. Writing SQL Queries: Let's Start with the Basics. https://technet.microsoft.com/en-us/library/bb264565(v=sql.90).aspx. Zugegriffen: 14.1.18.

neo4j. 2017a. From Relational to Neo4j. https://neo4j.com/developer/graph-db-vs-rdbms/.

neo4j. 2017b. Neo4j Cypher Refcard 3.3. https://neo4j.com/docs/cypher-refcard/current/. Zugegriffen: 2.12.17.

Robinson, Ian, James Webber, und Emil Eifrem. 2015. *Graph databases. New opportunities for connected data.* Beijing: O'Reilly.

van Bruggen, Rik. 2014. *Learning Neo4j. Run blazingly fast queries on complex graph datasets with the power of the Neo4j graph database*. Birmingham, UK: Packt Pub.

ABBILDUNGSVERZEICHNIS

TABELLENVERZEICHNIS