UNIVERSITÄT ZU LÜBECK
INSTITUT FÜR INFORMATIONSSYSTEME

Evaluierung von temporalen Graphdatenbanken am Beispiel von Neo4J

Bachelorarbeit

im Rahmen des Studiengangs
Medieninformatik
der Universität zu Lübeck

vorgelegt von
Ove Folger

ausgegeben und betreut von
Felix Kühr und Tanya Braun

Lübeck, den 18. Juli 2019

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Luebeck, 18. Juli 2019

_____

UNIVERSITY NAME

Zusammenfassung

Faculty Name
Department or School Name

Bachelor of Science

Evaluierung von temporalen Graphdatenbanken am Beispiel von Neo4J

von Ove Folger

The Thesis Abstract is written here (and usually kept to just this page). The page is kept centered vertically so can expand into the blank space above the title too...

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

For/Dedicated to/To my…

Kapitel 1

Einführungs

Um Daten in einer Datenbank abzulegen ist ein Datenmodell nötig, welches das Allgemeine Verhalten der Datenbank definiert. Dieses ist durch folgende Eigenschaften definiert ist: ein Liste von Datenstruktur Typen, ein Liste von Operatoren die auf die Daten angewandt werden können und eine Liste der Regeln zur Vollständigkeit der Datenbank (Codd, 1981). Zwei dieser Datenmodelle sind der Relationale-Ansatz und die NoSQL-Bewegung. Die meisten Datenbanken werden heutzutage nach dem relationalen Datenmodell verwaltet und mittels Structured Query Language (SQL) bearbeitet (Miller, 2013). Dieses Schema wurde über viele Jahre lang optimiert und gilt für viele Daten als performanteste Implementierung (Miller, 2013). Die verwendete Datenstruktur bildet eine Tabelle, eine Reihe bildet ein Objekt und die Spalten bilden die dazugehörigen Attribute (Miller, 2013). Die Möglichkeiten der Operationen, mit denen die Daten verändert oder angefragt werden,sind durch den sql-Standard bzw. die Implementierung des Standards limitiert. Die Regeln zur Vollständigkeit der Datenbank sind ebenfalls im Standard festgehalten.
Als eine Alternative zu diesem relationalen Datenmodell gibt es die NoSQL-Bewegung, welche erstmal 1998 erwähnt wurde (Strauch, Sites und Kriha, 2011). Diese Bewegung versuchte zunächst den Gebrauch von SQL als Anfragesprache zu vermeiden und brachte in den folgenden Jahren weitere Datenmodelle hervor. Eines dieser Modelle ist das Darstellen von Daten in der Datenstruktur eines Graphen (Miller, 2013). Die sogenannten Graphdatenbanken(GDB) werden besonders in dem Darstellen von Netzwerken verwendet (Han u. a., 2011), da das relationalen Datenmodell bei großen Datenmengen und vielen komplex verbunden Informationen durch eine hohe Anzahl von notwendigen Joins nicht performant verwendet werden kann (Miller, 2013). Für GDBs gibt es keine standardisierte Anfragesprache und die Menge an möglichen Operatoren ist sehr variable.
Eine der populärsten Graphdatenbank ist das open-source Projekt Neo4j(Francis u. a., 2018). In dieser Arbeit werden die Eigenschaften von Neo4j als Datenbank managment System beschrieben und die verwendete Architektur wird dargestellt.Aufbauend auf diesen Angaben werden Annahmen über das Verhalten des System getroffen und mittels eines selbst erstellten Datensatzes werden diese überprüft. Aufbauend auf dem analysierten Verhalten wird eine abschließende Bewertung zur dem Databank managment System Neo4j gegeben.

Kapitel 2

Grundlagen

2.1 Graph

Ein Graph ist eine abstrakte Datenstruktur, welche für die GDBs verwendet werden (Vicknair u. a., 2010). Ein Graph besteht aus einer endlichen nicht leeren Summe von Knoten, auch Vertex oder Punkt genannt und Kanten. Die Kanten des Graphen bildet die Verbindung zwischen zwei Knoten http://wwwmayr.in.tum.de/lehre/2008WS/ea/EA-7.pdf(7.06.19), wenn die Kanten aus einem geordneten Paar bestehen und somit eine Richtung besitzen wird der Graph als gerichtet bezeichnet, bei einem ungeordneten Paar wird von einem ungerichteten Graph gesprochen(Bondy und Murty, 1976). Wenn die Kanten Attribute, wie zum Beispiel Kosten besitzen wird der Graph als gewichtet bezeichnet und ohne Attribute als ungewichtet http://wwwmayr.in.tum.de/lehre/2008WS/ea/EA-7.pdf(7.06.19).

2.2 Neo4j

Neo4j ist eine Graphdatenbank, welche in Java implementiert wurde (Vukotic u. a., 2015). Als Grundlegende Datenstruktur wird ein gerichteter und gewichteter Graph verwendet.Knoten stellen die Entitäten,beispielsweise eine Person oder ein Produkt da und Kanten stellen die Relationen zwischen den Entitäten da, beispielsweise “isAngestellt” und können optional ein Gewicht und eine Richtung besitzen. Attribute können als zusätzliche Informationen in den Knoten gespeichert werden wie zum Beispiel: Name bei einem Knoten “Person”. Knoten können mit Bezeichnern versehen werden, um so leichter in Anfragen verwendet werden zu können. Die Operationen sind entweder durch durch die Anfragesprache Cypher,welche eine standardisierten Syntax mit mehreren vordefinierten Funktionen besitzt, limitiert oder durch die jeweilige verwendete Programmiersprache.Es wird das Einbetten weiterer Bibliotheken, welche weitere Funktionen oder Algorithmen stellen, unterstützt. Durch weitere Funktionen wie zum Beispiel die Bibliothek “APOC” [1] ist es möglich, Daten aus verschiedenen Formaten wie JSON,XSL oder XML in die Datenbank zu laden oder Daten aus anderen Web-APIs zu nutzen. Neo4j lässt sich in einem eingebetteten Modus oder einem Server Modus nutzen. Der eingebettete Modus dient der direkten Nutzung durch die Java Core API(Application programming interface) von Neo4j. Der Server Modus ermöglicht eine getrennte Ausführung von dem Code und dem bestehenden Neo4j System.

[1] https://neo4j-contrib.github.io/neo4j-apoc-procedures/ (17.05.19)

2.3 Neo4j als Datenbank management system

Ein Datenbank Managment System(DBMS), ist für die Verarbeitung von Anfragen verantwortlich und kann mit folgenden Teilen beschrieben werden: Eine Schnittstelle mit dem Nutzer, eine Anfragesprache, ein Anfrage-Optimierer, eine Speicherverwaltung,eine transaction engine, eine database engine und operation features, zum Wiederherstellen von Daten (Angles, 2012). Als Schnittstelle mit dem Nutzer stehen die Anwendungen "Neo4j Browser" unter Linux und "Neo4j Desktop" unter Windows zur Verfügung, zusätzlich besteht die Möglichkeit eine von den Neo4j Treibern unterstützte Programmiersprache oder Java zu verwenden und direkt die Neo4j APIs zu nutzen. Die Anfragesprache für Neo4j Browser und Neo4j Desktop ist "Cypher". Zum Optimieren von Anfragepläne wird der "Cypher Query Optimizer"verwendet. Die Speicherverwaltung wird durch die Record Files und die transaction unit wird durch das Transaction Management dargestellt. Die database engine bildet das Gesamtsystem der einzelnen Komponenten. Die operation features zum Wiederherstellen von Daten werden in dem Transaktions-Log realisiert. Optional kann zu eventuellen Performanz Steigerung ein Cache verwaltet werden. In der Neo4j Enterprise Version ist es möglich, das DBMS auf mehrere Systeme in einem Netzwerk mittels High Availability Funktion auszulagern (Vukotic u. a., 2015). Zusammengefasst wird die Architektur in 2.1

2.3.1 Cypher und APIs für Neo4j

Im Folgenden werden die Möglichkeiten zum Interagieren mit Neo4j beschrieben. Da es über die Neo4j-Treiber möglich ist, Neo4j in mehreren Programmiersprachen zu nutzen und die genaue Anzahl der Sprachen stetig steigt, werden im folgenden nur die Kernfunktionen in Java beschrieben und verwendet. Im späteren Absatz zu Neo4j im Servermodus wird zusätzlich das Bolt-protocol und die REST HTTP API beschrieben, welche als weitere Schnittstelle dienen.

Cypher

Im Gegensatz zu den relationalen Datenbanken gibt es bei Graphdatenbanken keine standardisierte Anfragesprache, welche bei den meisten Graphdatenbanken Verwendung findet (Han u. a., 2011). In Neo4j besteht seit 2000 die Möglichkeit die deklarative Anfragesprache "Cypher" zu verwendenden (Francis u. a., 2018). Cypher wird von Neo4, Inc. entwickelt und wurde ursprünglich ausschließlich für die Neo4j Datenbank verwendet. Seit Oktober 2015 findet Cypher als "openCypher" Gebrauch in anderen Datenbanken (Francis u. a., 2018). Da für die Traversal API und die Java Core API Kenntnisse in Java bzw. einer durch die Neo4j-Treiber unterstützten Programmiersprachen benötigt werden, bildet Cypher eine Möglichkeit ohne diese Kenntnisse mit der Datenbank zu interagieren(Vukotic u. a., 2015). Die Syntax ist an SQL und Gremlin orientiert. In Cypher wird ein Muster von dem Nutzer angegeben und alle Objekte, die dieses Muster erfüllen, können zurückgegeben werden. Die wichtigsten Prädikate sind:
Where: Dies hat die gleiche Funktion wie in SQL und spezifiziert Objekte durch einen Ausdruck.

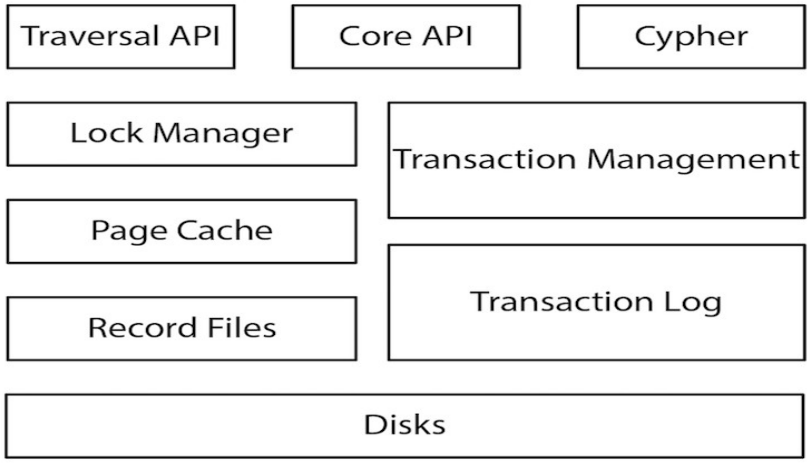

ABBILDUNG 2.1: Allgemeine Architektur von Neo4j.

Match: Dadurch wird das Muster spezifiziert indem Neo4j suchen soll. Beispielweise:

```
MATCH (p1:Person)-[:Friends]->(p2:Person)
WHERE p1.name= 'Peter'
RETURN p.name
```

, hier werden alle Personen-Knoten durchsucht, die mit einer "Friends"-Relation mit dem Personen-Knoten von "Peter" verbunden sind. p1 stellt ein Label für den Knoten vom Typ "Person" da, [:Friends] ist eine Relation vom Typ "Friends" und durch "->" wird angegeben, dass es sich um eine ausgehende Kante vom Knoten p1 handelt. Mit der Schreibweise [:Friends*2] wird ausgedrückt, dass es sich um die Tiefe 2 handelt, dies ist äquivalent zu : [:Friends]-()->[:Friends]-.
Return: Es wird angegeben, welche Objekte bzw. welche Attribute der Objekte, die das Muster erfüllen zurückgegeben werden sollen.

Delete: Wie in SQL wird ein Objekt oder mehrere Objekte aus der Datenbank entfernt.
With: Dadurch lassen sich in einer Anfrage Objekte manipulieren bevor sie zu einer weiteren Anfrage gegeben werden. Beispielsweise:

```
MATCH (p:Person{name: 'Peter'})
WITH COUNT(p) as count
RETURN count
```

Create: Erzeugt ein Objekt in der Datenbank Beispielsweise

```
Create (p:Person)
```

, hier wird ein Knoten vom Typ Person erzeugt. Es ist auch möglich mehrere Knoten mit den dazugehörigen Relationen zu erzeugen. Indizes für die Objekte oder Attribute von Objekten können ebenfalls über Create erzeugt werden.
Limit: Beschränkt die Menge, welche durch das Return-Statement zurückgegeben wird Beispielsweise

```
MATCH (p:person) RETURN p.name LIMIT 10
```

, hier werden nur die Namen von 10 Personen zurückgegeben
SUM/COUNT/AVG: Werden wie in SQL verwendet.
Der Nutzer beeinflusst so welche Daten durch das Musters gesucht werden. Der Nutzer besitzt keine Möglichkeit die Art der Berechnung zu beeinflussen. Cypher wird dadurch als Nutzerfreundlichere, aber auch weniger performante Alternative zu den APIs empfohlen (Vukotic u. a., 2015). In den folgenden Experimenten wurde sowohl Cypher als als auch die Travese API und die Core API für einen direkten Vergleich verwendet. Cypher lässt sich ausschließlich single threaded ausführen. Durch das Einbinden von User Defined Functions(UDF) für Cpyher, ist das Ausführen in mehreren threads in einigen Fällen möglich. Diese UDFs werden durch die APOC Bibliothek zur Verfügung gestellt [2].

Java Core API

Als nahe Schnittstelle zu den Kernfunktionen von Neo4j ist dies die Möglichkeit mit der besten Laufzeit (Vukotic u. a., 2015). Zur Verwendung dieser imperativen API sind weitreichende Programmierkenntnisse und Wissen über die Neo4j-Bibliotheken, sowie einer genauen Vorstellung über die Daten in dem Graph erforderlich. Wenn diese Kenntnisse gegeben sind, ist die API flexible verwendbar und der Nutzer hat hohen Einfluss darauf, wie die Anfragen bearbeitet werden sollen und kann so eine optimale Berechnungsstrategie angeben (Vukotic u. a., 2015). Die so produzierten Queries sind in meisten sehr lang. Eine Beispielhafte Berechung wäre:

```
try ( Transaction tx = graphDb.beginTx() )
{
```

[2] https://neo4j.com/developer/neo4j-apoc/(18.06.19)

```
        Node Peter = graphDb.getNodeById(Peter_ID);
        Set<Node> friends = new HashSet<Node>();
        for (Relationship R : Peter.getRelationships(FRIEND)) {
                Node friend = R.getOtherNode(userJohn);
                friends.add(friend);
        }
        for (Node friend : friends) {
          logger.info("Found friend: " + friend.getProperty("name"));
}
```

Bei diesem Beispiel werden alle Freunde von Peter gefunden und zurückgegeben.Der Nutzer gibt jede Transaktion explizit an,so ist es dem System möglich alle zur Verfügung stehenden Kerne zu nutzen.

Traversal API

Diese deklarative API dient zum spezifizieren von Traversierungen im Graph, sie ist schneller als Cypher und langsamer als die Core API. Die Traversal API erlaubt einen high-level Zugriff auf Neo4j, welcher weniger abstrakt als die Core API ist und dennoch Programmierkenntnisse erfordert. Der Nutzer muss keine genaue Vorstellung von den Daten im Graphen haben.Es wird eine Beschreibung angegeben, wie eine Suche genau ausgeübt werden soll und kann diese dann auf einen Knoten anwenden.

```
private Traverser getFriends(final Node person)
{
TraversalDescription td = graphDb.traversalDescription()
.depthFirst()
.relationships( RelTypes.FRIEND, OUTGOING )
.evaluator( Evaluators.atDepth(2) );
return td.traverse( person );
}
```

Zusammengefasst wäre diese Traversierung: Suche alle Verbindungen von dem Anfangsnode über die Relation FRIEND bis zu Tiefe 2. Beim Traversieren kann der der Nutzer zwischen 2 grundsätzlichen Vorgehen wählen: Breadth-first und Depth-first, diese haben abhängig von der Struktur des Graphen eine unterschiedliche Laufzeit (Vukotic u. a., 2015).
Breadth-First(Breiten-Suche) : Zuerst werden alle Knoten mit derselben Distanz betrachtet, danach werden alle Knoten mit der nächst höheren Distanz betrachtet, dies wird solange ausgeübt bis alle Knoten betrachtet wurden.
Depth-First(Tiefen-Suche) : Gegeben sei die maximale Distanz n, so wird zunächst ein Knoten mit der minimalen Distanz m betrachtet und ausgehend von diesem Knoten wird ein Knoten mit der Distanz m+1 ausgewählt. Dies wird solange wiederholt bis die aktuelle Distanz n-1 beträgt. Ausgehend von diesem Knoten werden alle Nachbarn betrachtet, wenn alle Nachbarn betrachtet wurden, wird von Startknoten ein neuer Knoten mit der Distanz m gewählt.

Wenn der Nutzer mit der Struktur der Daten in dem Graph vertraut ist, kann das ausgewählte eine erheblichen Performance Unterschied hervorbringen[16]. Die Traversion kann auch bidirektional aufgeführt werden, dabei werden 2 Anfangsknoten angeben und die Suche wird so lange ausgeführt, bis es zu einer Kollision der beiden Knoten kommt. Wenn eine Kollision zwischen den beiden Knoten erkannt wird, ist damit zum Beispiel eine Verbindung der beiden Knoten bewiesen. Für die folgenden Experimente wurde bei den meisten Anfragen die Traversal API verwendet.

2.3.2 Anfragebearbeitung und Planoptimierung

Nach Angaben von Neo4j[3], werden Anfragen, die mittels Cypher gestellt werden, nach folgendem Muster bearbeitet:

1. Umwandeln der Eingabe in einen Abstrakten Syntax Baum (ASB)
2. Optimieren des ASB
3. Erstellen eines Anfrage Graphen aus dem Baum
4. Erstelle einen logischen Plan
5. Schreibe den logischen Plan neu
6. Erstelle einen Ausführungsplan aus dem logischen Plan
7. Führe die Anfrage mit Hilfe, des Ausführungsplans aus

Die Schritte 2-5 werden vom Anfrage-Optimierer übernommen.

2. Die Optimierung des AST beinhaltet folgende Schritte:

1. Alle Labels, die sich in einem **Match** befinden, werden in das **Where**-Prädikat verschoben
2. Semantisch-Äquivalente **Where**-Prädikate werden zusammengefasst
3. Ersetze alle Synonyme zum Beispiel: **RETURN * => RETURN x as x, y as y**
4. Fasse Konstanten zusammen zum Beispiel: **3 + 3 => 6**
5. Setze bei anonymen Knoten einen Namen ein zum Beispiel: **MATCH () => MATCH (n)**
6. Ersetze das Gleichheitszeichen durch ein 'IN'wie: **MATCH (n) WHERE id(n) = 12 => MATCH n WHERE id(n) IN [12]**

3. Durch das Erstellen eined Anfragegraphen, wird ein abstraktere Darstellung für die Anfrage erzeugt. Diese lässt sich kosteneffizienter Optimieren.

4. Aus dem Anfragegraph werden logische Pläne für jede Anfrage erzeugt. Dieser

[3]https://neo4j.com/blog/introducing-new-cypher-query-optimizer/ (11.06.19)

Plan ist ein Baum mit maximal 2 Kindern, welche die verwendeten Operatoren darstellen. Dies gleicht dem logischen Plan für relationelle Datenbanken. Aus dem logische Plan wird der geschätzte Bearbeitungsaufwand für eine Anfrage gelesen. Der Aufwand wird aus den benötigten I/O-Operatoren auf den Speicher oder Indizies und den durchzuführenden Traversierungen ermittelt. Bei jedem Durchgang werden mehrere Pläne für eine Anfrage erzeugt, der Optimierers wählt aus diesem Plänen mit einem gierigen Suchalgorithmus einen Plan aus, welcher nicht immer der optimalste Plan ist. Der gierige Suchalgorithmus gewährleistest, dass nicht der langsamste Plan gewählt wird, aber es ist nicht garantiert, dass der optimalste Plan gewählt ist.

5. Nachdem die Pläne erstellt wurden und einer dieser Pläne ausgewählt wurde, wird der ausgewählte Plan nochmals optimiert. Das heißt mehrere Komponenten werden vereinfacht oder zusammengeführt und jede Art von Verschachtelung wird aufgelöst.

Nach diesem Schritt ist die Arbeit des Optimierers abgeschlossen und die Database-Engine kann aus diesem optimierten Plan einen ausführbaren Plan erstellen.

2.3.3 Speicherverwaltung in Neo4j

Graphdatenbanken besitzen eine andere Speicherverwaltung als relationelle Datenbanken (Angles, 2012). In Neo4j wird der Speicher in sogenannten Record Files unterteilt, diese Dateien speichern jeweils einen Teil des Graphen ab, wie Knoten, Kanten, Eigenschaften etc. Die Objekte besitzen in den Dateien eine feste Größe, dies erlaubt einen Zugriff in O(1) (Robinson, Webber und Eifrem, 2013). Wenn zum Beispiel der Knoten mit der ID "100"gesucht wird und ein einzelner Knoten X Bytes groß ist, wird der gesuchte Knoten bei Byte 100*X beginnen. Die genaue Größe variiert, je nach betrachteter Neo4j Version, seit Neo4j Version 3.X besitzt ein Knoten die Größe 15 Byte und werden in node-store gespeichert. [4]. Nach (Robinson, Webber und Eifrem, 2013) werden die Objekte wie folgt verwaltet:

Verwaltung der Knoten im Speicher

Ein Knoten im Knotenspeicher wird folgendermaßen dargestellt: Das erste Byte kennzeichnet, ob der Knoten verwendet wird bzw. verwendet werden kann. Die nachfolgenden 4 Bytes kennzeichnen die ID für die erste Relation, die mit dem Knoten verbunden ist. Die darauffolgenden 4 Bytes beschreiben das erste Attribute des Knoten. Die nächsten 5 Bytes verweisen auf ein Label, welches gegebenenfalls verwendet wurde und ich sich im Label-Store befindet. Das Letzte Byte ist für bestimmte Flags und für zukünftige Arbeiten.

Verwaltung der Kanten im Speicher

Die Kanten werden in dem Relationsspeicher gespeichert. Jeder Eintrag besitzt die IDs zu den zugehörigen Knoten, einen Pointer zu dem Relationstypen, welcher in dem Relationstypspeicher gespeichert ist, Pointer zu den vorherigen und nächsten

[4] https://neo4j.com/developer/kb/understanding-data-on-disk/ (13.06.19)

Relationen von den beiden zugehörigen Knoten und ein Flag das angibt, ob die betrachtete Relation die erste ist.

Verwaltung der Eigenschaften im Speicher

Da Neo4j eine property graph databse ist, kann jeder Knoten und jede Kante Eigenschaften besitzen. Die Eigenschaften befinden sich im Eigenschaftenspeicher. Jeder Eintrag zu einer Eigenschaft ist in 1-4 Blöcke und eine ID zu einer folgenden Eigenschaft unterteilt. Jeder dieser Blöcke beinhaltet Informationen über einen Typen, welcher ein standard- Java-Typ, string oder ein Array sein kann und Informationen über einen Pointer auf einen Index, der auf den Namen der Eigenschaft verweist, zeigt.

Indizies

Traversierung im Speicher

Die festgelegte Größe der Knoten und Kanten hat als Ziel die Traversierung zu beschleunigen (Robinson, Webber und Eifrem, 2013). Eine einfache Traversierung wird beschrieben wie folgt:

1. Starten bei einem Eintrag in den Knotenspeicher
2. Lesen des gegebene Byte für die ID zur ersten Relation und betrachten des entsprechenden Eintrags im Relationsspeicher mit dieser ID
3. Ausgehend von dem Eintrag im Relationsspeicher wird der Pointer zum dazugehörigen Knoten aufgerufen.
4. Springen zu der Knoten im Knotenspeicher

Diese Traversierung beschreibt das suchen eines benachbarten Knotens.

Caching im Speicher

LRU-K page-affined cache beschreibt, dass der Speicher aufgeteilt wird und eine feste Anzahl von Teilen wird im Speicher gehalten. Die Auswahl, der Teile die im Speicher gehalten werden geschieht nach der least frequently used(LFU) Methode. Die Methode besagt, dass ein Teil der selten benutzt wird aus dem Speicher entfernt wird und ein häufig genutzter Teil wird in dem Speicher gehalten (Robinson, Webber und Eifrem, 2013).

2.3.4 CAP und ACID unter Neo4j

Das CAP-Theorem charakterisiert das Verhalten einer Datenbank anhand von folgenden 3 Eigenschaften: Consistency(Konsistenz), Availability(Verfügbarkeit), Partitionstoleranz (Simon, 2000). Konsistenz beschreibt die Eigenschaft, dass die Daten in allen Partitionen zur selben Zeit dieselben Werte besitzen und das gleiche Verhalten aufweisen. Verfügbarkeit beschreibt die Möglichkeit zu jeder Zeit eine Anfrage

an das System stellen zu können und auch zu jeder Zeit eine Antwort auf die gestellte Anfragen bekommen zu können. Partitionstoleranz gewährleistet, dass sich das Verhalten des System nicht verändert, wenn mehrere Partitionen von diesem System erstellt werden und alle Partitionen müssen das gleiche Verhalten aufweisen (Simon, 2000). Neo4j erfüllt die Bedingung der Verfügbarkeit und der Partitionstoleranz[16] und wird so als "AP-Database" bezeichnet.
Die ACID Eigenschaft setzt sich aus 4 Eigenschaften zusammen, die das Verhalten der Transaktionen einer Datenbank beschreiben (Haerder und Reuter, 1983). Atomicy(Atomisch) beschreibt, dass jede Transaktion einzeln betrachtet wird und nur fehlschlagen oder erfolgreich sein kann. Durch Consistency(Konsistenz) kann jede Transaktion nur valide Daten verwenden und den validen Zustand einer Datenbank nicht in einen nicht-validen Zustand überführen. Isolation erwartet, dass jede Transaktion unabhängig von einer parallel-laufenden Transaktion abläuft und keine dieser Transaktionen beeinflusst. Durability(Haltbarkeit) ist gegeben, wenn der Effekt einer Transaktion auf den Speicher ausgeübt wurde und auch bei einem Absturz des Systems beibehalten wird (Haerder und Reuter, 1983). Solange Neo4j auf einem einzelnen System läuft und nicht das High Aviability Feature der Enterprise Edition nutzt, ist es ACID konform (Holzschuher und Peinl, 2013). Das atomische und haltbare Verhalten wird durch das sogenannte write-ahead log(wat) versichert. Bei diesem Mechanismus werden alle Operationen einer Transaktionen nach dem Beenden der Transaktion in einem Log-File festgehalten, bevor diese den Speicher beeinflussen, so kann auch bei einem Absturz des System das Log-File genutzt werden um ein vorherige Transaktion zu wiederholen. Dieses Log-File wird auch für die High-Availability genutzt, welche es erlaubt die Datenbank in einem Netzwerk auf mehrere Systeme zu verwenden, dennoch ist es nicht mehr möglich ein ACID Verhalten zu gewährleisten, da keine absolute Garantie für ein atomisches und konsistente Verhalten gibt (Vukotic u. a., 2015). Eine weitere versicherung für das atomische Verhalten bildet das Verhindern von Deadlocks innerhalb der Transaktion. Zur Verhinderung von Deadlocks wird "RWLock" verwendet, was eine Implementierung des Java "ReentrantReadWriteLock" für Neo4j ist. Dieser verwaltet alle Schreibsperren, die während einer Transaktion erstellt werden und versucht potentielle Deadlocks zu erkennen (Raj, 2015).

2.4 Modi in Neo4j

Wenn der Nutzer Neo4j nicht über die Anwendungen "Neo4j Browser/Desktop"nutzen möchte sondern über ein Programmiersprache lässt sich Neo4j in 2 Modi nutzen, den eingebetteten Modus und den Server Modus. Diese Modi geben an wie die Bibliotheken von Neo4j, welche für die Anfragen des Nutzer benötigt werden, aufgerufen und verwendet werden. Dafür wird das Verhalten der Java Viral Machine (JVM), welche die Kompatibilität des geschriebenen Java-Codes gewährleistet, angepasst. Der verwendete Speicher ist für bei Modi der selbe.

2.4.1 Der eingebettete Modus

Der eingebettete Modus erlaubt es dem Nutzer mittels jeder Programmiersprache, die von der JVM unterstützt wird und für die ein Treiber zur Verfügung steht, die Bibliotheken von Neo4j zu nutzen und so Anfragen an den Datenbank zu stellen. Alle Bibliotheken von Neo4j werden von dem GraphDataBaseService von Neo4j verwaltet. Dieser Modus wird für die meisten Anwendungen, die auf einem einzelnen System laufen, empfohlen (Raj, 2015). Dies ist damit begründet, dass das Gesamtsystem im einbetteten Modus schneller läuft als im Server Modus, aber auch nur auf einem System genutzt werden kann, da sowohl alle Neo4j Funktionen als auch die Anfragen in der selben JVM agieren. Der Nutzer hat so volle und alleinige Kontrolle über jede Transaktion und kann jede zur Verfügung stehenden API nutzen. Daraus resultiert, dass er Nutzer für ein sicheres starten und beenden der Datenbank in seinem Anfragecode verantwortlich ist(Robinson, Webber und Eifrem, 2013). Verdeutlicht wird dieses Verhalten in 2.2.

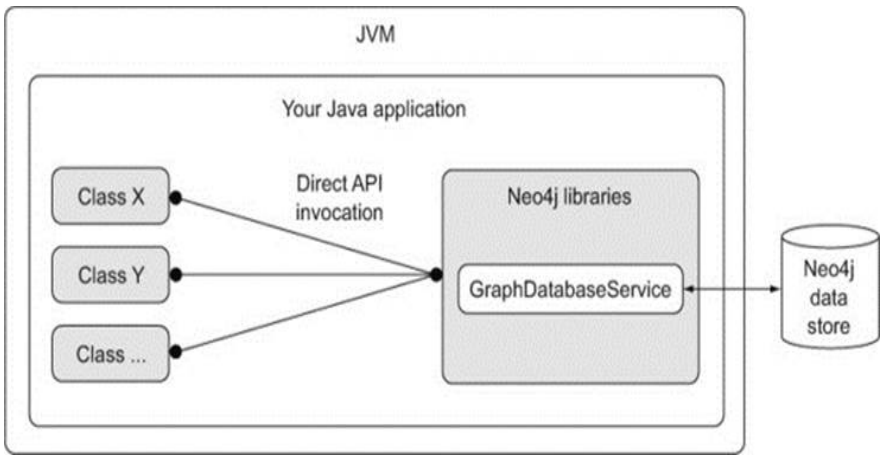

ABBILDUNG 2.2: Allgemeiner eingebetteteter Modus von Neo4j.

2.4.2 Der Server Modus

Im Server Modus werden alle Anfragen von den Nutzern in einem eigenen Prozess verwaltet und mittels HTTP(Hypertext Transfer Protocol) als JSON-Formatiertes Dokument an die sogenannte REST API übermittelt (Robinson, Webber und Eifrem, 2013). Die REST API läuft in der Neo4j JVM und verwaltet alle eintreffenden Anfragen der Nutzer und gibt diese an den GraphDatabaseService weiter, welcher die Bibliotheken verwaltet.

Der Server Modus ist für die Verwendung der High-Availability Fuktionen, welche das Nutzen der Datenbank auf mehren Systemen erlaubt, empfohlen(Raj, 2015). Da die Anfragen des Nutzer getrennt von der JVM der Neo4j Bibliotheken verwaltet werden und so mehrere Nutzer die Möglichkeit besitzen diese Funktionen zu Nutzen. Durch die Übertragungsverzögerung innerhalb der Netzwerks und dem Übertragen durch eine weitere API ist der Geschwindigkeit langsamer als die des eingebetteten Modus. Dies wird in 2.3 verdeutlicht.

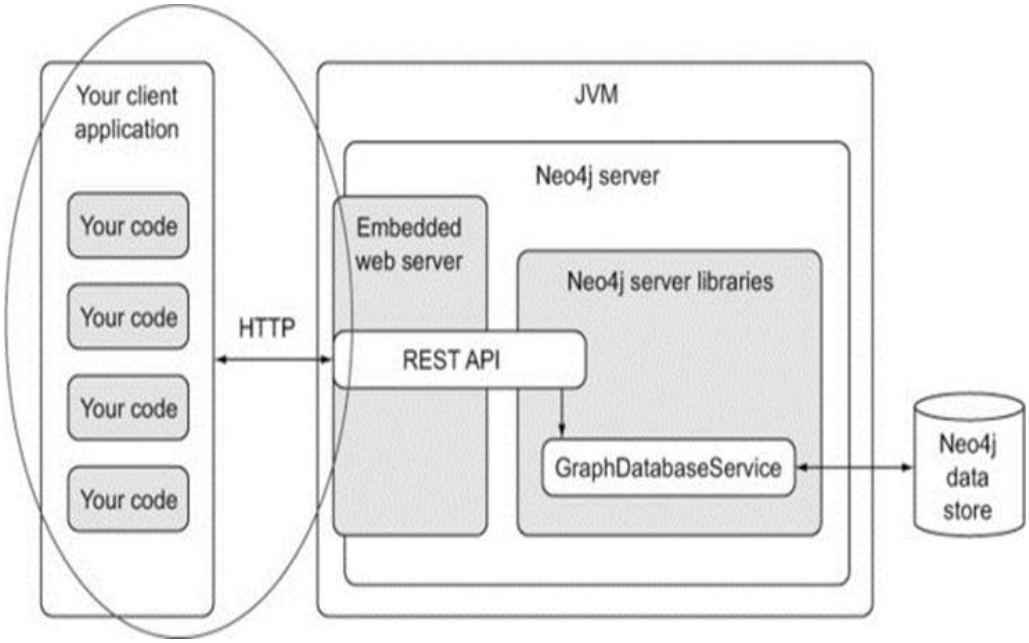

ABBILDUNG 2.3: Allgemeiner Servermodus von Neo4j.

Kapitel 3

Theoretische Grundlagen

3.1 Vorraussetzungen

Im Folgenden werden alle Einstellungen, die für die Evaluation getroffen wurden beschrieben und begründet.

3.1.1 Datensatz

Für das Experiment wurde ein selbsterstellter Datensatz gewählt, dieser wurde aus folgendem Code erstellt [1] unter Verwendung der Parameter [2]. Die Datenbank besteht aus insgesamt 50.004 Entitäten, davon sind 4 vom Typ Action und 50000 vom Typ Person. Der Typ Action besitzt 2 Eigenschaften, mit den Namen 'name', welche einen String erwartet und 'attribute', welche einen Int erwartet. Der Typ Person besitzt ebenfalls 2 Eigenschaften namens 'name' und 'attribute'.
Es existieren insgesamt 4 Arten von Relationen: RELATIONSHIP1 und RELATIONSHIP2 bilden eine Relation von einer Entität des Typen Person zu einer Entität des Typen Action und RELATIONSHIP3 und RELATIONSHIP4 eine Relation von Person zu Person. Jede Person besitzt jeweils eine Relation vom Typen RELATIONSHIP1 und eine vom Typen RELATIONSHIP2, wobei die Ziel-Actions disjunkt sind. Pro Person wurden 2.500 Relationen vom Typen RELATIONSHIP3 und 2.500 Relationen vom Typen RELATIONSHIP4 generiert, da die Ziel-Personen nicht disjunkt sind, kommt es zu einer Variablen Anzahl von zugehörigen Relationen Pro Person. Jede Person besitzt maximal 5002 Relationen. Da die Relationen als Kanten und die Entitäten als Knoten aufgefasst werden, besitzt der generierte Graph 50.004 Knoten und ca. 250.100.000 Kanten mit einer physischen Gesamtgröße von 8.8 GB.

3.1.2 Anfragen

Für die Evaluation werden die Anfragen nach (Angles, 2012) in folgende Kategorien unterteilt: Nachbarschafts-Anfragen, Erreichbarkeits-Anfragen, Muster Passungs Anfragen und Zusammenfassungen.

1. Nachbarschafts-Anfragen: Es wird geprüft, ob ein Knoten direkt über eine Kante mit einem anderen Knoten verbunden ist oder es werden alle Knoten die über eine bestimmte Menge an Kanten erreichbar sind zu einer Menge zusammengefasst, dies ist auch als k-nächste Nachbarn Problem bekannt. Beide

[1] https://github.com/McHektor123/BA/blob/master/Neo4j/InitNeo4j2.java
[2] https://github.com/McHektor123/BA/blob/master/Constants/CONSTANTS.java

Fälle lassen sich auch auf die Verbundenheit von Kanten über Knoten übertragen.

2. Erreichbarkeits-Anfragen: Die Suche nach einem Pfad, welcher 2 Knoten bzw. Kanten verbindet. Wenn ein Pfad besteht ist der Ziel-Knoten von dem Start-Knoten aus erreichbar, bei einem ungerichteten Graphen gilt der umgekehrte Fall ebenfalls. Beim dem Auffinden von mehreren Pfaden, entsteht zudem das Problem des Finden des kürzesten Pfades. Bei einem gewichteten Graphen kann dieses Problem um die Suche des schnellsten/leichtesten Pfades erweitert werden.

3. Muster Passungs Anfragen: Prüfen, ob der Graph ein Sub-Graph enthält, welcher Ähnlichkeiten zu einem gegebenen Muster oder zu anderen Sub-Graphen aufweist. Dies ist als Graph bzw. Sub-Graph Isomorphismus bekannt.

4. Zusammenfassungen: Diese Anfragen fassen eine Ergebnismenge zu einem Wert zusammen, dies ist durch Funktionen wie MAX,AVG oder COUNT ,aber auch dem Lesen von Eigenschaften des Graphen wie die Menge alle Knoten, realisiert.

Eine Anfrage kann dabei mehrere Kategorien beinhalten. In dem folgenden Experimenten werden hauptsächlich Anfragen der Kategorien 1,2 und 4 verwendet. Das gesamte Experiment besteht aus 2 Iterationen, in denen eine Menge von Anfragen gestellt und analysiert werden.

3.2 Erste Iteration

Die erste Iteration besteht primär aus Nachbarschafts-Anfragen und Zusammenfassungen. Im folgenden Abschnitt werden die Grundanfragen der ersten Iteration und die Permutationen aus diesen vorgestellt.

3.2.1 Grundanfragen zur ersten Iteration

Grundanfrage 1.1 wird beschrieben als:

```
MATCH (X:Person)-[:RELATIONSHIP3]-(Y)
WHERE X.attribute>=250 AND Y.attribute>=15
RETURN COUNT(DISTINCT(Y))
```

Grundanfrage 1.1 findet alle Personen, die über RELATIONSHIP3 erreichbar sind, unter der Voraussetzung dass bestimmte Bedingungen für die Eigenschaften 'attribute' erfüllt sind. Es handelt sich um eine Erreichbarkeits-Anfrage. Bei dieser Anfrage wird die Richtung der Relation RELATIONSHIP3 zu ausgehend,eingehend geändert und zusätzlichen werden diese Anfragen semantisch äquivalent in der JAVA Core API unter Verwendung der Traversal API formuliert. Bei den 3 in Cypher formulierten Anfragen wird die Relation RELATIONSHIP3 durch die Relation RELATIONSHIP2 ersetzt. Insgesamt entstehen 9 Anfragen aus Grundanfrage 1.1.
Grundanfrage 1.2 wird beschrieben als:

```
MATCH (p:Person {name:'Person613'}) return p
```

Diese Anfrage findet den Nutzer mit dem Namen "Person613". Sie wird dahingehend verändert, dass die Überprüfung des attributes 'name' in ein WHERE Prädikat verschoben wird und die Anfrage in der Core API formuliert wird. Aus Grundanfrage 1.2 entstehen insgesamt 3 Anfragen.
Grundanfrage 1.3 wird beschrieben als:

```
MATCH (X:Person{name: 'Person1'})-[:Relationship3]->(n1)
WITH COLLECT(n1) as n
MATCH (Y:Person{name: 'Person2'})-[:Relationship3]->(n1)
WHERE n1 in n
RETURN COUNT(DISTINCT(n1))
```

Diese Anfrage ermittelt alle gemeinsamen Nachbarn von Person1 und Person2, so ist es eine Nachbarschafts-Anfrage mit einer genutzten Zusammenfassung. Als Permutationen wird eine semantisch äquivalente Anfrage in den APIs und folgende in Cypher formuliert:

```
MATCH (X:Person{name: 'Person1'})-[:Relationship3]->(n1)
                <-[:Relationship3]-(Y:Person{name: 'Person2'})
RETURN COUNT(DISTINCT(n1))
```

Für alle Anfragen, die mit der Traversal API geschrieben wurden, wird die Breadth-First-Methode für die Traversierungen verwendet.

3.2.2 Erwartungen zur ersten Iteration

Im Folgenden werden Hypothesen über das Verhalten der Grundanfragen zur ersten Iteration aufgestellt und es werden Begründungen ausgestellt. In dem praktischen Teil werden diese Hypothesen überprüft.

Für **Grundanfrage 1.1** entstehen 3 Aspekte, welche betrachtet werden.

1. Das Verhalten beim Ändern der Richtung der Relationen
2. Die Performanzunterschiede zwischen der Cypher und den APIs
3. Die Unterschiede zwischen der Anfrage mit Verwendung von RELATIONSHIP3 und RELATIONSHIP2.

Zum 1. Aspekt: Es besteht die Vermutung, dass die Ausführungszeit am höchsten ist, wenn die Relation in beide Richtung gelten kann und die Zeit ist minimal wenn die Relation als ausgehende Kante beschrieben wird.
Zum 2. Aspekt: Wie in (Raj, 2015) beschrieben, wird es empfohlen die APIs für eine erhöhte Performanz zu nutzen, da diese APIs hardware-nahe arbeiten und Cypher als hardware-ferne Sprache in Neo4j aufgefasst wird. Aus diesem Grund wird erwartet, dass die Performanz bei den verwendeten Anfragen deutlich höher ist, wenn

die Java Core API mit Verwendung der Traversal API genutzt wird.
Zum 3. Aspekt: Da jede Person 1 Relation vom Typ RELATIONSHIP2 besitzt und maximal 2.500 vom Typ RELATIONSHIP3 wird Vermutet, dass die Anfragen mit der Relation RELATIONSHIP2 mindestens um den Faktor 1.000 schneller ausgeführt werden.

Für **Grundanfrage 1.2** werden 2 Aspekte betrachtet .

1. Die allgemeine Performanz und die entstehenden Unterschiede für das Suchen eines Knotens

2. Das Verhalten, wenn die Abfrage, nach dem Namen in das WHERE-Prädikat verschoben wird.

Zum 1. Aspekt: Wie bei der Grundanfrage 1.1 wird vermutet, dass der Java Core API eine höhere Performanz besitzt, insbesondere weil keine Traversal API verwendet wurde. Alle 3 Anfragen werden eine sehr schnelle Ausführungszeit besitzen, da besonders der Gebrauch von Indizes ein schnelles Finden durch Eigenschaften erlaubt.
Zum 2. Aspekt: Da der Optimizer nach Angaben der Neo4j Inc. [3] , die meisten Labels in das WHERE Prädikat verschiebt, sollte nur ein minimaler bis nicht vorhandener Performanzunterschied zwischen den beiden Anfragen bestehen.

Für **Grundanfrage 1.3** besteht ein Aspekt, der untersucht wird.

1. Ein Unterschied in der Laufzeit bei semantisches Äquivalenz

Zum 1. Aspekt: Es wird vermutet, dass es keine signifikante Differenz zwischen den Laufzeiten gibt, falls ein Unterschied bestehen sollte, wird diese auf die Performanz des IN-Operator zurückgeführt.

3.3 Zweite Iteration

Die zweite Iteration verwendet die Nachbarschafts-Anfragen in einer höheren Tiefe vom 2,3, wodurch eine höherer Rechenaufwand simuliert wird. Es wird zusätzlich das Verhalten zwischen der Depth-First-Methode und der Depth-First-Methode betrachtet. Es werden komplexere Anfragen gestellt und es wird die Skalierbarkeit des Systems analysiert.

3.3.1 Grundanfragen zur zweiten Iteration

Grundanfrage 2.1 wird beschrieben als:

```
MATCH t=(p:Person{name :'Person1'})-[:RELATIONSHIP3]->(p1:Person)
                                   -[:RELATIONSHIP3]->(p2)
WHERE NOT (p)-[:RELATIONSHIP3]->(p2)
RETURN COUNT(DISTINCT(p2))
```

[3] https://neo4j.com/blog/introducing-new-cypher-query-optimizer/ (27.06.19)

Diese Anfrage findet alle Personen, welche über die RELATIONSHIP3 mit dem direkten Nachbarn p1 verbunden sind, aber nicht mit Person p verbunden sind. Ein Praktisches Szenario für diese Anfrage, wäre das Finden von Freunden von Freunden, die die Startperson nicht kennt. Diese Anfrage ist eine Erreichbarkeits- und Muster-Passungs Anfrage. Als Permutation wird eine semantisch äquivalente Anfrage in den APIs formuliert und es wird folgende Anfrage in Cypher formuliert:

```
MATCH t=(p:Person{name :'Person1'})-[:RELATIONSHIP3*2]->(p1:Person)
WHERE NOT (p)-[:RELATIONSHIP3]->(p1)
RETURN COUNT(DISTINCT(p1))
```

Grundanfrage 2.2 wird beschrieben als:

```
MATCH (p:Person{name :'Person1'})-[:RELATIONSHIP3*2]->(p1:Person)
RETURN COUNT(DISTINCT(p1))
```

Diese Anfrage findet die Anzahl aller Verbindungen über die Relation RELATIONSHIP3 mit der exakten Tiefe 2. Es handelt sich um eine Nachbarschafts-Anfrage. Diese Grundanfrage wird erneut mit der Tiefe 3 ausgeführt und beide Anfragen werden in den APIs mit der Breath-First-Methode und Depth-First-Methode ausgeführt. Es ergeben sich 6 Anfragen. **Grundanfrage 2.3** wird beschrieben als:

```
MATCH (p:Person{name :'Person1'}),(p1:Person{name :'Person42'}),
                path=shortestPath((p)-[:RELATIONSHIP4*..3]->(p1))
RETURN LENGTH(path)
```

Diese Anfrage gibt die Länge des kürzesten Pfades zwischen die Personen Person1 und Person42 über die Relation RELATIONSHIP4 an, die maximale länge beträgt 3. Diese Anfrage ist eine Erreichbarkeits-Anfrage und nutzt den shortestPath Algorithmus von Cypher. Diese Anfrage wird erneut mit der Core API ausgeführt und als folgende Alternative in Cypher dargestellt:

```
MATCH (p:Person{name :'Person1'}),(p1:Person{name :'Person42'}),
                path=(p)-[:RELATIONSHIP4*..3]->(p1)
RETURN LENGTH(path)
ORDER BY length(path) asc LIMIT 1
```

3.3.2 Erwartungen zur zweiten Iteration

Es werden Hypothesen zu den komplexeren Grundanfrage aus der zweiten Iteration vorgestellt und Ursache für das erwartete Verhalten werden aufgestellt. Diese Hypothesen werden in dem Teil zur praktischen Arbeit überprüft und die Ausführbarkeit von sehr rechenaufwendigen Anfragen wird getestet.

Für **Grundanfrage 2.1** werden folgende Aspekte betrachet:

1. Ein Unterschied in der Laufzeit bei semantisches Äquivalenz
2. Die allgemeine Performanz von dem Ausdruck "WHERE NOT".

Zum 1. Aspekt: Es wird vermutet, dass die Anfrage mit [RELATIONSHIP3*2] eine schneller Ausführungszeit besitzt, da dieser Ausdruck auf den Compiler optimiert ist.
Zum 2. Aspekt: Da eine Weitere Anfrage im WHERE-Prädikat gestellt wird und beide Ergebnismengen verglichen werden, wird vermutet dass die Ausführungszeit sehr hoch ist. Durch die Eigenschaften des Multithreading wird die Anfrage in der Core-API um ein vielfaches schneller beantwortet werden.

Für **Grundanfrage 2.2** werden folgende Aspekte betrachet:

1. Ein Unterschied zwischen der Traversierung in den Tiefen 2 und 3
2. Das Verhalten von Breath-First und Depth-First
3. Der relative Anteil an erreichten Personen im gesamten Graphen

Zum 1. Aspekt: Bei der Anfrage in Cypher wird davon ausgegangen, dass die Zeit für die Traversierung mit der Tiefe 3 in quadratischer Abhängigkeit zu der mit Tiefe 2 steht. Für Breath-First und Depth-First besteht die Annahme, dass ein linearer Unterschied entsteht und eine der beide Methoden signifikant schneller ausgeführt wird.
Zum 2. Aspekt: Da der Graph ausgehend von dem Knoten der Person1 mit der Tiefe 3 eine relativ hohe Breite im Vergleich zu der Tiefe besitzt, wird vermutet dass die Anfrage mit Depth-First signifikant schneller ausgeführt wird.
Zum 3. Aspekt: Durch die Tiefe 2 können unter der Annahme, dass alle Personen über die RELATIONSHIP3 genau 2500 Ziel-Personen besitzen, maximal 6.250.000 Knoten erreicht werden. Da der Graph 50.000 Personen-Knoten besitzt, gibt es eine hohe Redundanz unter den Ziel-Personen. Durch die theoretisch hohe Anzahl von zu erreichenden Knoten bei der Tiefe 2 wird vermutet ,dass alle Personen bei der Traversierung zur Tiefe 2 erreicht werden.

Für **Grundanfrage 2.3** werden folgende Aspekte betrachtet:

1. Die Berechnungszeit für den kürzesten Weg und die Länge von diesem
2. Die Ausführungszeit in Cypher ohne Verwenden des Algorithmus

Zum 1. Aspekt: Durch die Vermutung, dass alle Personen über RELATIONSHIP3/ RELATIONSHIP4 mit der Tiefe 2 erreicht werden können, wird davon ausgegangen dass der Pfad die Länge 1 oder 2 besitzen wird. Die Ergebnisse werden mit den Ausführungszeiten von Grundanfrage 2.2 zusammenhängen und es wird keine höhere Ausführungszeit erwartet.
Zum 2. Aspekt: Ausgehend von der Annahme,dass der Algorithmus für den Compiler optimiert ist, wird der Anfrage mit dem Algorithmus eine signifikant schnellere Ausführungszeit aufweisen als die Anfrage ohne gegebenen Algorithmus.

Kapitel 4

Praktische Arbeit

Literatur

Angles, Renzo (2012). „A comparison of current graph database models“. In: *2012 IEEE 28th International Conference on Data Engineering Workshops*. IEEE, S. 171–177.

Bondy, John Adrian, Uppaluri Siva Ramachandra Murty u. a. (1976). *Graph theory with applications*. Bd. 290. Citeseer.

Codd, Edgar F (1981). „Data models in database management“. In: *ACM Sigmod Record* 11.2, S. 112–114.

Francis, Nadime u. a. (2018). „Cypher: An evolving query language for property graphs“. In: *Proceedings of the 2018 International Conference on Management of Data*. ACM, S. 1433–1445.

Haerder, Theo und Andreas Reuter (1983). „Principles of transaction-oriented database recovery“. In: *ACM computing surveys (CSUR)* 15.4, S. 287–317.

Han, Jing u. a. (2011). „Survey on NoSQL database“. In: *2011 6th international conference on pervasive computing and applications*. IEEE, S. 363–366.

Holzschuher, Florian und René Peinl (2013). „Performance of graph query languages: comparison of cypher, gremlin and native access in Neo4j“. In: *Proceedings of the Joint EDBT/ICDT 2013 Workshops*. ACM, S. 195–204.

Miller, Justin J (2013). „Graph database applications and concepts with Neo4j“. In: *Proceedings of the Southern Association for Information Systems Conference, Atlanta, GA, USA*. Bd. 2324. S 36.

Raj, Sonal (2015). *Neo4j high performance*. Packt Publishing Ltd.

Robinson, Ian, Jim Webber und Emil Eifrem (2013). *Graph databases*. Ö'Reilly Media, Inc."

Simon, Salomé (2000). „Brewer's cap theorem“. In: *CS341 Distributed Information Systems, University of Basel (HS2012)*.

Strauch, Christof, Ultra-Large Scale Sites und Walter Kriha (2011). „NoSQL databases“. In: *Lecture Notes, Stuttgart Media University* 20.

Vicknair, Chad u. a. (2010). „A comparison of a graph database and a relational database: a data provenance perspective“. In: *Proceedings of the 48th annual Southeast regional conference*. ACM, S. 42.

Vukotic, Aleksa u. a. (2015). *Neo4j in action*. Bd. 22. Manning Shelter Island.