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
Prof. Dr. Ralf Möller

Lübeck, den 6. November 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, 6. November 2019

_____

Zusammenfassung

Evaluierung von temporalen Graphdatenbanken am Beispiel von Neo4j

Durch das steigende Interesse an der Analyse von vernetzten Informationen werden Datenbanken des NoSQL-Ansatzes weiterentwickelt und für diese Analyse genutzt (Angles, 2018). Eine Untergruppe dieses Ansatzes stellen die Graphdatenbanken dar, einer der populärsten Vertreter dieser Gruppe ist Neo4j. In dieser Arbeit wird beschrieben, wie sich die Performanz des Systems Neo4j bei der Bearbeitung von verschiedenen Anfragen verhält. Zu diesem Zweck wurden mehrere Anfragen mit semantisch gleichen Vergleichsanfragen an Neo4j gestellt und die Differenz der Bearbeitungszeiten wurde analysiert. Jede Anfrage wurde einer oder mehreren Kategorien zugeordnet, um so eine Korrelation zwischen Kategorie und Bearbeitungszeit einer Anfrage zu erkennen.
Für eine genauere Einordnung der Performanz von Neo4j wurden alle aufgestellten Anfragen ebenfalls auf dem Datenbankmanagementsystem OrientDB ausgeführt (OrientDB-Ltd., 2019). Die Analyse zeigt, dass Neo4j die vorgestellten Anfragen mit der eigenen Anfragesprache Cypher durchschnittlich ca.1,9 mal schneller bearbeitete als OrientDB mit SQL. Neo4j kann für eine erhöhte Performanz in Java bedient werden, die Anfragen wurden in Java ca. 207 mal schneller als in OrientDB bearbeitet. Es konnte keine allgemeine Korrelation zwischen Kategorie einer Anfrage und ihrer Bearbeitungszeit in Neo4j festgestellt werden. Für das Speichern des Datensatzes benötigte Neo4j ca. 2,4 mal mehr Speicherplatz als OrientDB. Die Nutzungsvielfalt und gute Performanz wurden als Stärke und die schlechte Speicherverwaltung als Schwäche von Neo4j hervorgehoben.

Evaluation of temporal graph databases using the example of Neo4j

Due to the increasing interest in analyzing networked information, databases of the NoSQL approach are further developed and used for this analysis (Angles, 2018). A subset of this approach is represented by the graph databases, one of the most popular members of this group is Neo4j. This work describes how the performance of the Neo4j system behaves when handling various requests. For this purpose, several queries with semantically equal comparison queries were processed on the system and the difference of processing times was analyzed. Each query has been assigned to one or more categories to detect a correlation between the category and the processing time of a query.
In order to rank the performance of Neo4j, all queries were also executed on the databasemanagementsystem OrientDB (OrientDB-Ltd., 2019). The analysis shows that Neo4j processed the requested queries with its own query language Cypher on average about 1.9 times faster than OrientDB with SQL. Neo4j can be used for increased performance in Java, the queries were processed in Java about 207 times faster than in OrientDB. There was no general correlation between the category of a query and its processing time in Neo4j. Neo4j needed about 2.4 times more memory than the OrientDB to save the dataset. The versatility and good performance were highlighted as a strength and the poor memory management as a weakness of Neo4j.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

SQL	**S**tructured **Q**uery **L**anguage
NoSQL	**N**ot **o**nly **SQL**
GDB	**G**raph **D**ata**b**ase
API	**A**pplication **P**rogramming **I**nterface
DBMS	**D**ata**b**ase**m**anagement**s**ystem
UDF	**U**ser **D**efined **F**unctions
APOC	**A**wesome **P**rocedures **o**n **C**ypher
ACID	**A**tomicity **C**onsistency **I**solation **D**urability
CAP	**C**onsistency **A**vailibility **P**artition Tolerance
LFU	**L**east **F**requently **U**sed
ASB	**A**bstrakter **S**yntax **B**aum
GUI	**G**rafisches **U**ser **I**nterface
HTTP	**H**yper**t**ext **T**ransfer **P**rotocol
LFU	l**east**f**requently** **u**sed
JVM	**J**ava**v**irtual **m**achine

Kapitel 1

Einführung

Für die Verwaltung von Daten in einer Datenbank ist ein Datenmodell nötig, welches das allgemeine Verhalten der Datenbank definiert. Dieses Modell ist durch folgende Eigenschaften definiert: Ein Liste von verwendeten Datenstrukturtypen, ein Liste von Operatoren, die auf die Daten angewendet werden können und Regeln zur Vollständigkeit der Datenbank (Codd, 1981). Zwei dieser Datenmodelle sind der relationale Ansatz und die NoSQL-Bewegung. Der relationale Ansatz besitzt allgemein eine hohe Effizienz und die NoSQL-Bewegung ist für spezifische Anwendungen entworfen, dadurch zählen diese beiden zu den verbreitetsten Datenmodellen (Vicknair u. a., 2010).
Die meisten Datenbanken werden nach dem relationalen Ansatz verwaltet und mittels Structured Query Language (SQL) bearbeitet. Dieses Datenmodell wurde über viele Jahre optimiert und gilt für viele unabhängige Daten als performanteste Implementierung (Miller, 2013). Der Zugriff auf Eigenschaften von vorhandenen Entitäten kann durch die genutzte Datenstruktur mit einer hohen Performanz ausgeführt werden. Als Datenstruktur wird eine Tabelle verwendet, wobei eine Reihe ein Objekt und die Spalten die dazugehörigen Attribute bilden (Tatarinov u. a., 2002). Die Menge der Operationen, mit denen die Daten verändert oder angefragt werden, ist an den verwendeten SQL-Standard bzw. eine Implementierung des Standards orientiert. Die Regeln zur Vollständigkeit der Datenbank hängen ebenfalls mit dem verwendeten SQL-Standard zusammen.
Als eine Alternative zu dem relationalen Datenmodell gibt es die NoSQL-Bewegung, welche erstmals 1998 erwähnt wurde (Strozzi, 1998). 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 mit der Verwendung eines Graphen als Datenstruktur (Miller, 2013). Die sogenannten Graphdatenbanken (GDBs) werden unter anderem zum Darstellen von Beziehungen von Personen innerhalb eines sozialen Netzwerkes wie Facebook verwendet (Han u. a., 2011). In der Modellierung eines solchen Netzwerkes bestehen viele Relationen zwischen den Personen und jede Person besitzt viele Eigenschaften wie Name oder Geburtsdatum. Das relationale Datenmodell kann bei großen Netzwerken durch eine hohe Anzahl von notwendigen Berechnungen wie zum Beispiel Verbund nicht performant verwendet werden, GDBs besitzen für solche Netzwerke passendere Berechnungen, wie eine effiziente Traversierung über den Graphen und eigenen sich daher besser bei Modellierung eines Netwerkes (Miller, 2013). Für GDBs gibt es keine standardisierte Anfragesprache und die Menge an möglichen Operatoren, sowie die Regeln zur Vollständigkeit sind verschieden. Eine der populärsten Graphdatenbanken ist die erwähnte Datenbank Neo4j (Francis u. a., 2018).

In dem folgenden Kapitel dieser Arbeit wird eine Definition für die Datenstruktur Graph dargelegt und die Verwendung dieser Datenstruktur durch Neo4j wird beschrieben. Es wird ein Überblick von den zur Verfügung stehenden Werkzeugen und der grundlegenden Architektur geschaffen. Anschließend werden die einzelnen Bestandteile der Architektur, die zusammen das Datenbankmanagementsystem bilden, genauer beschrieben. Abschließend wird auf die Modi eingegangen, in denen Neo4j bedient werden kann.
Das Kapitel 3 beschreibt den verwendeten Datensatz und stellt eine Kategorisierung für Anfragen vor. Aufbauend auf diesen Kategorien werden drei Testläufe vorgestellt, welche aus mehreren Anfragen bestehen. Es werden verschiedene Aspekte bei diesen Anfragen betrachtet und es werden Hypothesen über das Verhalten der Anfragen aufgestellt.
Für Kapitel 4 werden alle Anfrage aus Kapitel 3 ausgeführt und die Ergebnisse werden tabellarisch aufgeführt. Mit Hilfe der Ergebnisse folgt eine Überprüfung der aus Kapitel 3 aufgestellten Hypothesen. Anschließend wird ein praktische Anwendungsszenario für Neo4j beschrieben. Aufbauend auf den erfassten Erkenntnissen wird unter Berücksichtigung der Limitierungen eine Bewertung über Neo4j getroffen.

Kapitel 2

Grundlagen zu Neo4j

2.1 Graph als Datenstruktur

Ein Graph ist eine abstrakte Datenstruktur, welche aus der Mathematik stammt (Vicknair u. a., 2010). Graphen werden als ein geordnetes Triple (V(G), E(G), ψ_G) aufgefasst, für GBDs sind V(G) und E(G) endliche Mengen. V(G) ist eine nicht leere Menge von Knoten, auch Punkte genannt. E(G) ist eine Menge von Kanten. Die Funktion ψ weist jeder Kante ein Tupel aus Knoten zu, so stellt $\psi_G(e) = (v_1 v_2)$ eine Verbindung der Knoten v_1 und v_2 durch die Kante e dar. Wenn ψ ein geordnetes Tupel von Knoten verwendet und die Reihenfolge somit relevant ist, besitzt die Kante eine Richtung und der Graph wird als gerichtet bezeichnet, bei einem ungeordneten Paar wird von einem ungerichteten Graph gesprochen. Wenn die Kanten Gewichte oder Kosten zur Traversierung besitzen, wird der Graph als gewichtet bezeichnet und ohne Gewichte als ungewichtet (Bondy und Murty, 1976).

2.2 Temporale Datenbanken

Eine temporale Datenbank besitzt die Fähigkeit, alle Daten in Abhängigkeit von einer zeitlichen Einheit zu speichern (Campos, Mozzino und Vaisman, 2016). Jede Relation und Entität kann ein Datum besitzen, zu welchem diese gültig ist. Zu jedem Zeitpunkt, der nicht diesem Datum entspricht, ist es nicht garantiert, dass der betrachtete Eintrag in der Datenbank gültig ist. Beispiel 1: Wenn eine Person zu einem genannten Zeitpunkt bei einer Firma angestellt ist, kann zu einem Zeitpunkt, der nach dem genannten Zeitpunkt folgt, keine garantierte Aussage darüber getroffen werden, ob die Person noch bei der Firma angestellt ist.
Die Objekte und Relationen können Attribute von einem zeitlichen Datentypen besitzen, welches eine bestimmte Eigenschaft beschreibt und nichts über die Gültigkeit aussagt (Khurana, 2012).

2.3 Überblick zu Neo4j

Neo4j ist eine in Java implementierte temporale Graphdatenbank (Vukotic u. a., 2015). Als grundlegende Datenstruktur wird ein gerichteter und gewichteter Graph verwendet. Knoten stellen die Entitäten dar und Kanten stellen die Relationen zwischen den Entitäten dar. Attribute werden als zusätzliche Informationen in den Knoten oder Katen gespeichert, wie *Name* bei einem Knoten vom Typ *Person*. Knoten und

Kanten können mit Bezeichnern versehen werden, um so leichter in Anfragen verwendet werden zu können. Die zur Verfügung stehenden Operationen von Neo4j sind entweder durch die jeweilige Programmiersprache oder durch die Anfragesprache Cypher definiert, wobei Cypher eine standardisierten Syntax mit mehreren vordefinierten Funktionen besitzt. Es wird das Einbetten weiterer Bibliotheken unterstützt, welche zusätzliche Funktionen zur Verfügung stellen. Durch diese Funktionen ist es unter anderem möglich Daten aus verschiedenen Formaten wie JSON, CSV oder XML in die Datenbank zu laden oder Daten aus einer anderen Web-API (Application programming interface) zu nutzen. Neo4j lässt sich im eingebetteten Modus oder im Server-Modus nutzen. Der eingebettete Modus dient der direkten Nutzung durch die Java Core API von Neo4j. Der Server-Modus ermöglicht die parallele Nutzung auf mehreren Systemen.

2.4 Neo4j als Datenbankmanagementsystem

Ein Datenbankmanagementsystem (DBMS) ist für die Verarbeitung von Anfragen verantwortlich und kann in folgende Teilsysteme unterteilt werden(Angles, 2012):

1. Schnittstellen für den Nutzer
2. Anfragesprache
3. Anfrage-Optimierer
4. Speicherverwaltung
5. Transaktionseinheit
6. Database Engine
7. Operationsmöglichkeiten zum Wiederherstellen von Daten

Die Teilsysteme werden durch die in der Abbildung 2.1 dargestellten Architektur realisiert.

1. Durch die Schnittstellen ist es dem Nutzer möglich mit den System zu agieren und Daten zu manipulieren. Als Schnittstellen für den Nutzer stehen die Webanwendung Neo4j Browser und Desktopanwendung Neo4j Desktop zur Verfügung. Zusätzlich besteht die Möglichkeit, eine von den Neo4j-Treibern unterstützte Programmiersprache oder Java zu verwenden und direkt die Neo4j APIs zu nutzen.
2. Die Anfragesprache beschreibt die Sprache in welcher der Nutzer seine Anfragen an das System stellt. Für Neo4j Browser und Neo4j Desktop ist Cypher die Anfragesprache.
3. Der Optimierer beschleunigt durch Ausführung von mehreren Teilschritten das Bearbeiten von Anfragen. Als Optimierer für Anfragen in Cypher wird der Cypher Query Optimizer verwendet(Neo4j-Inc., 2015b).
4. Die Speicherverwaltung dient dem Ablegen der Daten im physischen Speicher. In Neoj werden die Record Files hierfür verwendet.

5. Die Transaktionseinheit stellt sicher, dass keine Fehler während dem Ausführen von Transaktionen geschehen.

6. Die Database Engine bildet das Gesamtsystem der einzelnen Komponenten und führt die Anfragen aus.

7. Die Operationsmöglichkeiten zum Wiederherstellen und manipulieren von Daten werden mit dem Transaktions-Log realisiert.

Optional kann zur Performanzsteigerung ein Cache verwaltet werden. Dieser Cache hält einen Teil der Datenbank in dem Hauptspeicher und ermöglicht einen schneller Zugriff auf diese Teile der Daten. Die vorgestellten Gegenstände werden in den folgenden Abschnitten erläutert. In der Neo4j Enterprise Version ist es möglich, das DBMS auf mehrere Systeme in einem Netzwerk mittels hoher Verfügbarkeits Funktion zu verteilen (Vukotic u. a., 2015).

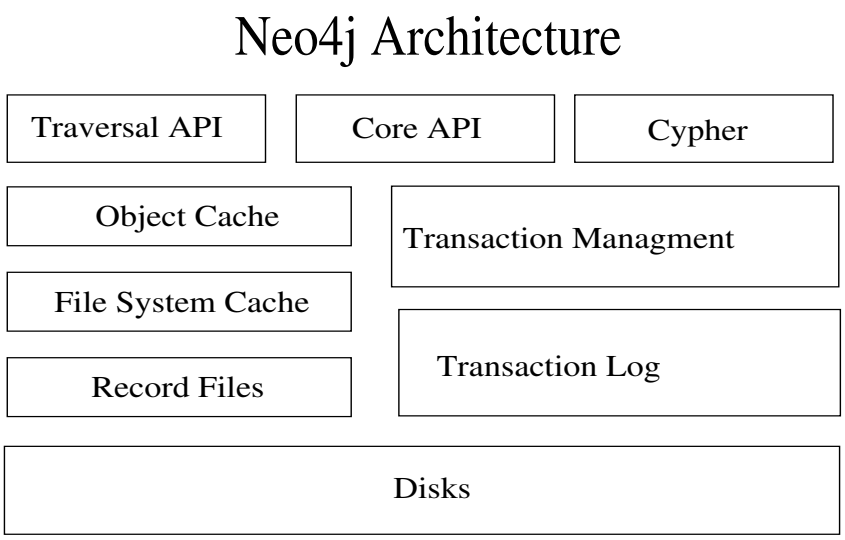

ABBILDUNG 2.1: aus https://dzone.com/articles/graph-databases-for-beginners-native-vs-non-native(26.08.19)

2.4.1 Cypher und APIs für Neo4j

Im Folgenden werden die Möglichkeiten zum Programmieren mit Neo4j beschrieben. Da es über die Neo4j Treiber möglich ist, Neo4j in mehreren Programmiersprachen zu nutzen und die Anzahl der unterstützten Sprachen stetig steigt, werden im folgenden nur die Kernfunktionen in Java beschrieben und verwendet.

Cypher

Im Gegensatz zu den relationalen Datenbanken gibt es bei Graphdatenbanken keine standardisierte Anfragesprache, welche in den meisten Graphdatenbanken Verwendung findet (Han u. a., 2011). In Neo4j besteht seit dem Jahr 2000 die Möglichkeit,

die deklarative Anfragesprache Cypher zu verwendenden (Francis u. a., 2018). Cypher wird von Neo4j Inc. entwickelt und wurde ursprünglich ausschließlich für die Neo4j Datenbank verwendet. Für die APIs von Neo4j sind Kenntnisse in der Programmiersprache Java bzw. einer durch die Neo4j-Treiber unterstützten Programmiersprachen notwendig. Cypher bildet eine Möglichkeit ohne diese Kenntnisse die Datenbank anzusteuern (Vukotic u. a., 2015). In Cypher wird ein Muster durch den Nutzer angegeben und alle Objekte, die dieses Muster erfüllen, werden zurückgegeben. Die wichtigsten Prädikate sind:
Create: Erzeugt ein Objekt in der Datenbank.

```
CREATE (p:Person)
```

Es ist möglich, mehrere Knoten mit den dazugehörigen Relationen zu erzeugen. Indizes für die Objekte oder Attribute von Objekten können ebenfalls über Create erzeugt werden.
Delete: Wie in SQL wird ein oder mehrere Objekte aus der Datenbank entfernt.

```
DELETE (p:Person{name: 'Peter'})
```

Where: Wie in SQL werden Objekte anhand von Attributen gefiltert.
Match: Spezifiziert das Muster in dem Neo4j sucht.

```
MATCH (p1:Person)-[:Friends*2]->(p2:Person)
WHERE p1.name= 'Peter'
RETURN p2.name
```

Es werden alle Personen-Knoten durchsucht, die mit einer Friends-Relation mit dem Personen-Knoten von Peter verbunden sind. p1 stellt einen Bezeichner für den Knoten vom Typ Person dar, [:Friends] ist eine Relation vom Typ Friends und durch "->" wird angegeben, dass es sich um eine ausgehende Kante vom Knoten p1 handelt. Mit der rekursiven Schreibweise [:Friends*2] wird ausgedrückt, dass es sich um die Tiefe 2 handelt, das heißt es werden Freunde von Freunden gesucht.
Return: Es wird angegeben, welche Objekte bzw. welche Attribute der Objekte, die das Muster erfüllen zurückgegeben werden sollen.
With: Dadurch lassen sich in einer Anfrage Objekte manipulieren bevor sie zu einer weiteren Anfrage gegeben werden.

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

Limit: Beschränkt die Anzahl, welche durch das Return-Ausdruck zurückgegeben wird

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

Hier werden nur die Namen von 10 Personen zurückgegeben
SUM/COUNT/AVG: Wie in SQL wird die Summe, die Anzahl oder der Durchschnitt von einer gegebenen Menge gebildet. Solch eine Funktionen ist wird Aggregationsfunktion bezeichnet.

Durch die gegebenen Prädikate wird beeinflusst, welche Daten mittels des Musters gesucht und zurückgegeben werden. Es besteht 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 dieser Evaluation wird sowohl Cypher als auch die APIs für einen direkten Vergleich verwendet. Cypher wird in Neo4j ausschließlich auf einem Prozessorkern ausgeführt. Durch das Einbinden von User Defined Functions (UDF) für Cypher ist das Ausführen auf mehreren Prozessorkernen in einigen Fällen möglich. Diese UDFs können von den Nutzer erstellt und zur Verfügung gestellt werden und durch Bibliotheken wie die Awesome Procedures on Cypher (APOC) verbreitet werden (Neo4j-Inc., 2019a).

Java Core API

Als nahe Schnittstelle zu den Kernfunktionen von Neo4j bietet die Java Core API die meiste Kontrolle über das System und besitzt bei korrekter Verwendung eine bessere Performanz als Cypher (Vukotic u. a., 2015). Zur Verwendung dieser imperativen API sind weitreichende Programmierkenntnisse und Wissen über die Neo4j-Bibliotheken, sowie ein genaues Verständnis über die Daten in dem Graph erforderlich. Wenn diese Kenntnisse gegeben sind, ist die API flexible verwendbar und der Nutzer hat einen hohen Einfluss darauf, wie die Anfragen bearbeitet werden sollen und kann eine optimale Berechnungsstrategie angeben (Vukotic u. a., 2015). Der Nutzer gibt jede Transaktion explizit an, so ist es dem System möglich die Transaktionen zu parallelisieren und nicht ausgelastete Prozessorkerne zu nutzen. Gegeben sei folgende Transaktion:

```
1  try ( Transaction tx = graphDb.beginTx() ){
2          Node Peter = graphDb.getNodeById(Peter_ID);
3          Set<Node> friends = new HashSet<Node>();
4          for (Relationship R : Peter.getRelationships(FRIEND)) {
5            Node friend = R.getOtherNode(userJohn);
6            friends.add(friend);
7          }
8          for (Node friend : friends) {
9            logger.info("Gefundener Freund: "
10            + friend.getProperty("name"));
11         }
12 }
```

Die gesamte Transaktion wird für eine Ausnahme sichere Programmierung mit einem try-Ausdruck umschlossen. In Zeile 2 wird die Variable *Peter* vom Typ Node erstellt und mit dem Knoten mit der ID *Peter_ID* initialisiert. In Zeile 3 wird ein Set vom Typ Node erstellt, welches die Ergebnismenge darstellt. Durch die Zeilen 4-7 werden alle Knoten, die über die Relation FRIEND mit dem Knoten Peter verbunden sind, zur Ergebnismenge hinzugefügt. In den Zeilen 8-11 wird über die Ergebnismenge iteriert und alle gefundenen Nachbarn werden zurückgegeben.

Traversal API

Die deklarative Travesal API dient zum spezifizieren von Traversierungen im Graph, sie ist näher an den Kernfunktionen von Neo4j als Cypher und weiter entfernt als die Core API. Die Traversal API erlaubt einen Zugriff auf Neo4j, welcher abstrakter als die Core API ist. Der Nutzer muss keine genaues Verständnis von den Daten im Graphen haben. Es wird eine Beschreibung zur Traversierung definiert und diese wird auf einen Graphen anwenden. Gegeben sei folgende Traversierung:

```
1 private Traverser getFriends(final Node person)
2 {
3         TraversalDescription td = graphDb.traversalDescription()
4                 .depthFirst()
5                 .relationships( RelTypes.friend, OUTGOING )
6                 .evaluator( Evaluators.toDepth(2) );
7         return td.traverse( person );
8 }
```

Diese Traversierung stellt in Java eine eigene Funktion dar, welche einen Ausgangsknoten als Eingabeparameter erwartet. In Zeile 3 wird eine neue Beschreibung zur Traversierung erstellt. Mit Zeile 4 gibt wird die Tiefensuche als Suchalgorithmus für die Traversierung spezifiziert. Durch Zeile 5 wird angegeben, dass nur über ausgehende, friend-Relationen traversiert wird. In Zeile 6 wird die maximale Traversierungstiefe auf zwei limitiert. Der Ausdruck in Zeile 7 wendet die erstellte Beschreibung zur Traversierung auf den übergebenen Ausgangsknoten an und gibt das Ergebnis zurück.
Beim Traversieren kann der Nutzer zwischen drei grundsätzlichen Vorgehen wählen: Breitensuche, Tiefensuche und bidirektionale Traversierung, diese haben abhängig von der Struktur des Graphen und der gestellten Anfrage eine unterschiedliche Laufzeit (Vukotic u. a., 2015). In Abbildung 2.2 werden durch die Pfeile die Reihenfolgen des jeweiligen Suchalgorithmus dargestellt.
Breadth-First-Search (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-Search (Tiefen-Suche): Zunächst wird ein Knoten der Tiefe 1 gewählt, ausgehend von diesem Knoten wird ein nächsttieferer Knoten, der noch nicht betrachtet wurden, gewählt. Dies wird so lange wiederholt bis keine neuen nächsttieferen Knoten hinzugefügt werden können, danach so lange rückwärts gegangen bis ein Knoten zu erreichen ist, der noch nicht betrachtet wurde.
Bidirektionale Traversierung: Es werden zwei Knoten gewählt und von jedem diesen Knoten wird eine Traversierung mit Tiefen- oder Breitensuche gestartet, die beiden Traversierungen müssen nicht über den gleichen Relationstypen verlaufen oder im selben Schrittintervall vollzogen werden. Bei dem Treffen der beiden Traversierungs-Pfade muss ein Verhalten definiert werden .

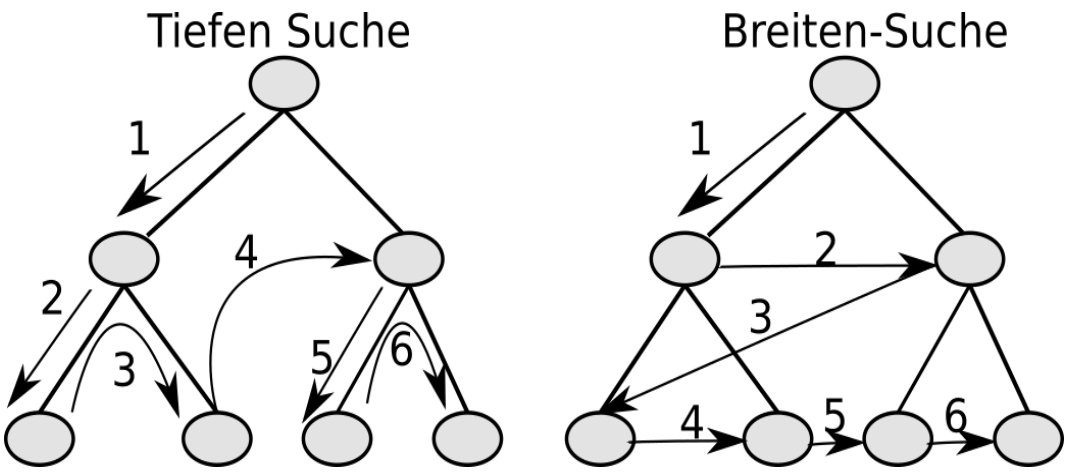

ABBILDUNG 2.2

Wenn der Nutzer mit der Struktur des Graph vertraut ist, kann die ausgewählte Methode einen erheblichen Performanzunterschied hervorbringen (Vukotic u. a., 2015).

2.4.2 Anfragebearbeitung und Planoptimierung

Nach Angaben von Neo4j werden Anfragen in Cypher, nach folgendem Muster bearbeitet (Neo4j-Inc., 2015b):

1. Umwandeln der Eingabe in einen abstrakten Syntax Baum (ASB)
2. Optimieren des ASB
3. Erstellen eines Anfragegraphen aus dem ASB
4. Erstellen eines logischen Plans
5. Optimieren des logischen Plan
6. Erstellen eines Ausführungsplan aus dem logischen Plan
7. Ausführen der Anfrage mit Hilfe des Ausführungsplans

Die Schritte 2-5 werden vom Cypher Query Optimizer übernommen.

1. Die Eingabe wird auf semantische Fehler bezüglich der Datentypen und auf allgemeine syntaktische Fehler überprüft. Wenn keine Fehler erkannt wurden, wird die Eingabe in einen ASB umgewandelt, welcher die Semantik der Eingabe darstellt.

2. Die Optimierung des ASB beinhaltet folgende Schritte:

(i) Alle Bedingungen, die sich in einem **Match**-Prädikat befinden, werden in das **Where**-Prädikat verschoben

(ii) Semantisch-äquivalente **Where**-Prädikate werden zusammengefasst

(iii) Ersetze alle Synonyme wie: **RETURN * => RETURN x as x, y as y**

(iv) Fasse Konstanten zusammen wie: **3 + 3 => 6**

(v) Setze bei anonymen Knoten einen Namen ein wie: **MATCH () => MATCH (n:Person)**

(vi) Ersetze das Gleichheitszeichen durch ein 'IN' wie: **MATCH (n) WHERE id(n) = 12 => MATCH n WHERE id(n) IN [12]**

3. Durch das Erstellen eines Anfragegraphen wird ein abstraktere Darstellung für die Anfrage erzeugt.

4. Aus dem Anfragegraphen wird ein logischer Plane für jede Anfrage erzeugt. Dieser Plan ist ein Baum mit maximal zwei Kindern, welche die verwendeten Operatoren darstellen. Dies gleicht einem logischen Plan für relationale Datenbanken. Aus dem logischen Plan wird der geschätzte Bearbeitungsaufwand für eine Anfrage gelesen. Der Aufwand wird aus den benötigten Eingabe-/Ausgabe-Operatoren auf den Speicher oder Indizies und den durchzuführenden Traversierungen ermittelt. Bei jedem Durchgang werden mehrere Pläne für eine Anfrage erzeugt, der Optimierer wählt aus diesen Plänen mit einem gierigen Suchalgorithmus einen Plan aus, welcher nahe am optimalste Plan ist, aber garantiert nicht der langsamste Plan ist.

5. Nachdem die Pläne erstellt wurden und einer dieser Pläne ausgewählt wurde, wird der ausgewählte Plan nochmals optimiert. Alle Komponenten werden so weit wie möglich vereinfacht und Redundanzen werden zusammengeführt. Jede Art von Verschachtelung, wie ein Match-Prädikat in einer Where Bedingung wird aufgelöst. Die Arbeit des Optimierers ist mit dem 5. Schritt abgeschlossen

6. Der logische Plan gibt vor, wie die Anfrage semantisch bearbeitet wird, es wird nicht spezifiziert mit welchen konkreten Operatoren die Anfrage ausgeführt werden soll. Der Ausführungsplan baut auf dem logischen Plan auf und gibt für die benötigten Operatoren eine hierarchische Anordnung von physischen Implementierungen vor. Dieser Plan wird durch die Database Engine erstellt.

7. Die Operatoren in dem Ausführungsplan werden während der Laufzeit in der vorgegebenen Reihenfolge ausgeführt.

2.4.3 Speicherverwaltung in Neo4j

Im Gegensatz zu relationalen Datenbanken wird der Speicher in Neo4j in sogenannten Record Files unterteilt (Angles, 2012). Jede einzelne Datei speichert einen Teil des Graphen wie Knoten, Kanten, Attribute ab. Die Objekte besitzen eine feste Größe, dies erlaubt einen Zugriff in konstanter Zeit (Robinson, Webber und Eifrem, 2013). Wenn ein 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 besitzt ein Knoten die Größe 15 Byte und werden im Knotenspeicher gespeichert (Neo4j-Inc., 2015c). Die Objekte werden nach (Robinson, Webber und Eifrem, 2013) wie folgt verwaltet:

Verwaltung der Knoten im Speicher

Das erste Byte eines Eintrages kennzeichnet, ob der Knoten verwendet wird bzw. genutzt 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 Attribut des Knoten. Die nächsten 5 Bytes verweisen auf ein Label, welches gegebenenfalls verwendet wurde und sich im Label-Store befindet. Das letzte Byte ist für bestimmte Flags und für zukünftige Verwendungszwecke.

Verwaltung der Kanten im Speicher

Die Kanten werden in dem Relationsspeicher gespeichert. Jeder Eintrag besitzt die IDs zu den zugehörigen Knoten, einen Zeiger zu dem Relationstypen, welcher in dem Relationstypspeicher gespeichert ist, einen Zeiger zu den vorherigen und nächsten Relationen der beiden zugehörigen Knoten und ein Flag, das angibt ob die betrachtete Relation die erste in einer Liste von zugehörigen Relationen im Graphen ist.

Verwaltung der Attribute im Speicher

Neo4j wird als Attributs Graphdatenbank bezeichnet, dort kann jeder Knoten und jede Kante ein oder mehrere Attribute besitzen. Diese Attribute befinden sich im Attributsspeicher. Jeder Eintrag zu einem Attribut ist in 1-4 Blöcke und einer ID zu einem nächstfolgenden Attribut unterteilt. Jeder der Blöcke beinhaltet Informationen über einen Typen, welcher ein String, ein Array oder ein Java-Standard Typ sein kann und einen Zeiger auf ein Index zu dem Namen des Attributs.

Indizies

Ein Index ist eine redundante Kopie von Daten in einer Datenbank, dadurch kann auf ein Element direkt zugegriffen werden, ohne dass über die gesamte Datenstruktur iteriert werden muss. Dieses Vorgehen beschleunigt die Suche nach Daten in der Datenbank. Neo4j besitzt die Indexfreie Nachbarschafts Eigenschaft, dadurch sind die Knoten nicht über einen Index sondern direkt über Relationen verbunden. Bei einer GDB ohne diese Eigenschaft werden die Knoten durch einen globalen Index verbunden und eine weitere Datenstruktur wie ein Binärbaum oder eine Hash-Tabelle wird für das Traversieren verwendet (Robinson, Webber und Eifrem, 2013).
Neo4j nutzt Indizes ausschließlich für Attribute und erlaubt das Erstellen eines Index zu einem oder mehreren Attributen. Sobald ein Index zu einem angegebenen Attribute erstellt wurde, wird dieser immer aktuell gehalten (Neo4j-Inc., 2015a).

Caching im Speicher

Neoj4 nutzt den LRU-K page-affined cache, wodurch die Datenbank aufgeteilt wird und eine feste Anzahl von Teilen der Datenbank im Hauptspeicher gehalten wird. Die Auswahl der Teile, die im Hauptspeicher gehalten werden, geschieht nach dem least frequently used (LFU) Vorgehen. Bei diesem Vorgehen wird ein selten genutzter Teil aus dem Speicher entfernt und einer häufig genutzter Teil wird im Speicher gehalten (Robinson, Webber und Eifrem, 2013).

2.4.4 CAP und ACID unter Neo4j

Das CAP-Theorem charakterisiert das Verhalten einer Datenbank anhand von folgenden drei Eigenschaften: Konsistenz (Consistency), Verfügbarkeit (Availability), Partitionstoleranz (Partition Tolerance) (Simon, 2000). Konsistenz beschreibt die Eigenschaft, dass die Daten bei einer Parallelisierung 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 Systems nicht verändert, wenn das System in mehrere kleinere Teilstücke, auch Partitionen genannt, zerteilt wird. Alle Partitionen müssen das gleiche Verhalten wie das gesamte System aufweisen (Simon, 2000). Neo4j erfüllt die Bedingung der Verfügbarkeit und der Partitionstoleranz (Vukotic u. a., 2015) und wird als "AP-Database" bezeichnet.
Die ACID Eigenschaft setzt sich aus vier Bedinungen zusammen, die das Verhalten der Transaktionen einer Datenbank beschreiben (Haerder und Reuter, 1983). Atomar (Atomicity) beschreibt, dass jede Transaktion einzeln betrachtet wird und entweder komplett fehlschlagen oder erfolgreich sein kann. Durch Konsistenz (Consistency) kann jede Transaktion nur valide Daten verwenden und den validen Zustand einer Datenbank nicht in einen nicht-validen Zustand überführen. Isolation (Isolation) erwartet, dass jede Transaktion unabhängig von einer parallellaufenden Transaktion abläuft und keine dieser Transaktionen beeinflusst. Haltbarkeit (Durability) ist gegeben, wenn sich der Effekt einer Transaktion auf den Speicher ausübt und auch bei einem Absturz des Systems bestehen bleibt (Haerder und Reuter, 1983).
Neo4j erfüllt die ACID Eigenschaft, wenn es auf einem einzelnen System ausgeführt wird (Holzschuher und Peinl, 2013). Das atomische und haltbare Verhalten wird durch den write-ahead log versichert. Bei diesem Mechanismus werden alle Operationen einer Transaktionen nach dem Beenden der Transaktion in einer Log-Datei festgehalten, bevor diese den Speicher beeinflussen, so kann auch bei einem Absturz des Systems die Log-Datei genutzt werden um ein vorherige Transaktion zu wiederholen. Diese Log-Datei wird auch für die hohe Verfügbarkeits Funktion genutzt, welche es erlaubt, die Datenbank in einem Netzwerk auf mehrere Systeme zu verwenden, dennoch ist es nicht mehr möglich, ein ACID Verhalten bei Verwendung der hohen Verfügbarkeits Funktion zu gewährleisten, da es keine absolute Garantie für ein atomisches und konsistente Verhalten gibt (Vukotic u. a., 2015). Um die Isolation und Konsistenz zu gewährleisten, werden Verklemmungen, bei der sich Transaktionen gegenseitig blockieren, verhindert. Neo4j erstellt hierfür vor einer Transaktion Schreib- und Lesesperren und verhindern, dass eine weitere Transaktion auf Daten zugreift bzw. diese überschreibt, die zeitgleich von dieser Transaktion verwendet werden. (Raj, 2015).

2.5 Modi zur Bedienung von Neo4j

Neben der Nutzung von Neo4j mit den Anwendungen Neo4j Browser und Neo4j Desktop, lässt sich Neo4j mit der Programmiersprache Java im eingebetteten Modus und Server-Modus nutzen. Diese Modi beeinflussen wie die Bibliotheken von Neo4j, welche für die Bearbeitung der Anfragen benötigt werden, aufgerufen und

verwendet werden. Dafür wird das Verhalten der Java virtual machine (JVM), welche die Kompatibilität des geschriebenen Java-Codes gewährleistet, angepasst. Der verwendete Speicher ist für beide Modi derselbe.

2.5.1 Der eingebettete Modus

Der eingebettete Modus ermöglicht die direkte Nutzung der Bibliotheken von Neo4j mit einer Programmiersprache, die von der JVM unterstützt wird und für die ein Treiber zur Verfügung steht. Alle Bibliotheken von Neo4j werden durch den GraphDataBaseService von Neo4j verwaltet. Der eingebettete Modus wird für Anwendungen, die auf einem einzelnen System laufen, empfohlen (Raj, 2015). Da sowohl alle Funktionen von Neo4j als auch die Anfragen in der selben JVM agieren. Das System kann in diesem Modus nur von einem Nutzer verwendet werden, welcher dadurch die volle und alleinige Kontrolle über jede Transaktion hat und jede zur Verfügung stehenden Funktion der API nutzen kann. Daraus resultiert, dass dieser Nutzer für ein sicheres Starten und Beenden der Datenbank in seiner Sitzung verantwortlich ist (Robinson, Webber und Eifrem, 2013). Der Modus wird in Abbildung 2.3 dargestellt.

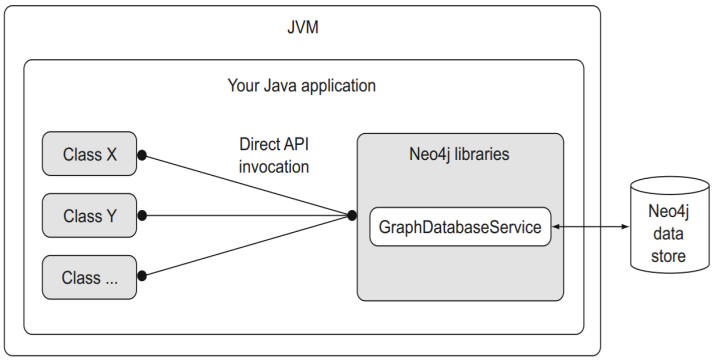

ABBILDUNG 2.3: aus https://livebook.manning.com/book/neo4j-in-action/chapter-10/9 (16.07.19)

2.5.2 Der Server-Modus

Im Server-Modus werden alle Anfragen in einem eigenen Prozess verwaltet und mittels HTTP (Hypertext Transfer Protocol) als JSON-formatiertes Dokument an die REST API übermittelt (Robinson, Webber und Eifrem, 2013). Die REST API läuft in der JVM von Neo4j und verwaltet alle eintreffenden Anfragen der Nutzer und gibt diese an den GraphDatabaseService weiter, welcher die von Neo4j zur Verfügung gestellten Bibliotheken verwaltet.
Der Server-Modus ermöglicht die Verwendung der hohe Verfügbarkeits Funktion, welche das Nutzen der Datenbank auf mehren Systemen erlaubt (Raj, 2015). Da die Anfragen der Nutzer getrennt von der JVM von Neo4j verwaltet werden und so mehrere Nutzer die Möglichkeit besitzen auf die Neo4j Datenbank zuzugreifen. Durch die Übertragungsverzögerung innerhalb der Netzwerks und dem Übertragen durch eine weitere API ist die Verzögerung höher als die des eingebetteten Modus. Dieser Modus wird in Abbildung 2.4 dargestellt.

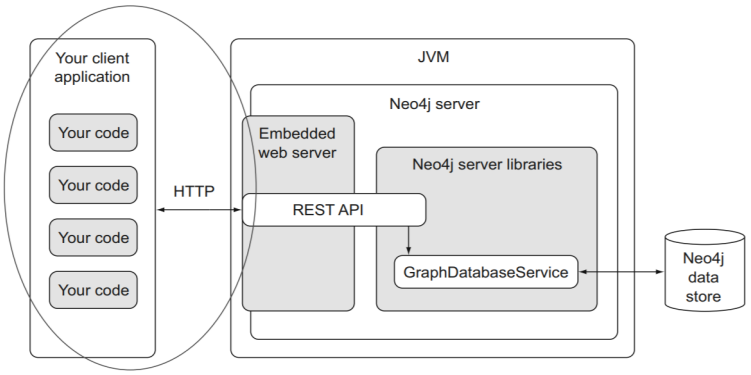

ABBILDUNG 2.4: aus https://livebook.manning.com/book/neo4j-in-action/chapter-10/9 (16.07.19)

Kapitel 3

Testläufe der Evaluation

3.1 Vorraussetzungen

Im Folgenden wird der verwendete Datensatz und Kategorisierung der Anfragen für diese Evaluation beschrieben. Unter Verwendung des Datensatzes werden drei Testläufe mit Grundanfragen und Vergleichsanfragen vorgestellt.

3.1.1 Der verwendete Datensatz

Für die Evaluation wurde ein selbsterstellter Datensatz gewählt, welcher aus einem selbstgeschriebenen Programm erstellt wurde (Folger, 2019). Dies ermöglicht das Stellen von spezifischen Anfragen und das Aufstellen von Hypothesen über das Verhalten des Systems. Die Datenbank läuft im eingebetteten Modus von Neo4j und besteht aus insgesamt 50.004 Entitäten, welche die Objekte der Datenbank bilden. Vier der Entitäten sind vom Typ Aktion und 50000 vom Typ Person. Der Typ Aktion besitzt 3 mit einem Index versehende Attribute mit den Namen *name*, welches aus einem String besteht, *attribute*, welches aus einem Integer besteht und *time*, welches von dem Typen Date ist. Der Typ Person besitzt ebenfalls drei mit einem Index versehende Attribute *name*, *attribute* und *time*. Jeder Wert der einzelnen Attribute tritt genau einmal auf, sodass alle Werte disjunkt sind.
Es existieren insgesamt vier Arten von Relationen. Die Relationen RELATIONSHIP1 und RELATIONSHIP2 bilden jeweils eine Relation von Person zu Aktion und die Relationen RELATIONSHIP3 und RELATIONSHIP4 bilden jeweils eine Relation zwischen Personen. Jede Person besitzt jeweils eine Relation vom Typ RELATIONSHIP1 und RELATIONSHIP2, wobei die Ziel-Aktionen disjunkt sind. Jede Person besitzt 2.500 Relationen vom Typ RELATIONSHIP3 und vom Typ RELATIONSHIP4, die Ziel-Personen wurden zufällig ausgewählt und sind nicht disjunkt. Insgesamt besitzt jede Person 5002 ausgehende Relationen. Da die Relationen als Kanten und die Entitäten als Knoten aufgefasst werden, besitzt der generierte Graph 50.004 Knoten und 250.100.000 Kanten mit einer physischen Gesamtgröße von 102,9 GB. Ein Graph dieser Größe kann zum Beispiel alle Studierenden und Angestellten der Universität Hamburg mit ihren Eintritts- oder Einstellungsdatum und Beziehungen zueinander darstellen.

3.1.2 Anfragen

Für die Evaluation werden den Anfragen folgende Kategorien von Angles zugeordnet (Angles, 2012):

1. Nachbarschaftsanfragen: Es werden alle Knoten, die über eine Anzahl von Kanten erreichbar sind, zusammengefasst. Das k-nächste Nachbarn Problem wird dieser Kategorie zugeordnet (Papadopoulos und Manolopoulos, 2006).

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

3. Mustervergleichsanfragen: Es wird überprüft, ob der Graph ein Sub-Graph enthält, welcher Ähnlichkeiten zu einem gegebenen Muster oder zu anderen Sub-Graphen aufweist. Zu dieser Kategorie gehört das NP-Vollständige Problem des Sub-Graph Isomorphismus (Yannakakis, 1990).

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

Einer Anfrage können mehrere Kategorien zugeordnet werden. Das Evaluation ermittelt, ob in Neo4j eine Korrelation zwischen Kategorie und Bearbeitungszeit einer Anfrage besteht. Hierfür werden 3 Testläufen durchgeführt, in denen eine Menge von Anfragen gestellt und analysiert werden.

3.2 Erster Testlauf

Der erste Testlauf besteht aus einfachen Nachbarschaftsanfragen und Zusammenfassungen, um einen ersten Überblick über die Funktionsweise von Neo4j zu erhalten. Im folgenden Abschnitt werden die Grundanfragen und Vergleichsanfragen des ersten Testlaufes vorgestellt. Alle Anfragen mit der Traversal API verwenden die Breitensuche für die Traversierungen.

3.2.1 Grundanfragen zu ersten Testlauf

Das ist die **Grundanfrage 1.1.1**:

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

Grundanfrage 1.1.1 findet alle Nachbarn, die über eine ausgehende Kante vom Typ RELATIONSHIP3 erreichbar sind, die Start- und Zielknoten werden durch Bedingungen für die Eigenschaft *attribute* gefiltert und das Ergebnis wird durch eine Zusammenfassung dargestellt . Es handelt sich um eine Nachbarschaftsanfrage. Grundanfrage 1.1.2 verwendet die gleichen Bedingungen wie Grundanfrage 1.1.1 und betrachtet alle eingehenden statt ausgehenden Kanten. Grundanfrage 1.1.3 betrachtet

sowohl ausgehende, als auch eingehende Kanten mit Verwendung der gleichen Bedingungen. In den Vergleichsanfragen werden diese drei Grundanfragen semantisch äquivalent in der Java Core API unter Verwendung der Traversal API formuliert und zusätzlich in Cypher mit der Relation RELATIONSHIP2 gestellt. Es entstehen neun Anfragen aus Grundanfrage 1.1.1, dargestellt in Tabelle 3.1.

Anfrage	Name
Grundanfrage 1.1.1	Cypher - RELATIONSHIP3 ausgehend
Grundanfrage 1.1.2	Cypher - RELATIONSHIP3 eingehend
Grundanfrage 1.1.3	Cypher - RELATIONSHIP3 beides
Vergleichsanfrage 1	Cypher - RELATIONSHIP2 ausgehend
Vergleichsanfrage 2	Cypher - RELATIONSHIP2 eingehend
Vergleichsanfrage 3	Cypher - RELATIONSHIP2 beides
Vergleichsanfrage 4	Core API - RELATIONSHIP3 ausgehend
Vergleichsanfrage 5	Core API - RELATIONSHIP3 ausgehend
Vergleichsanfrage 6	Core API - RELATIONSHIP3 ausgehend

TABELLE 3.1: Grundanfrage 1.1 und Vergleichsanfragen

Das ist die **Grundanfrage 1.2**:

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

Diese Anfrage findet die Personen-Knoten mit dem Namen *Person*613. In einer Vergleichsanfrage wird die Überprüfung des Attributes *name* in ein WHERE-Prädikat verschoben. Für eine weitere Vergleichsanfrage wird die Anfrage in der Core API formuliert. Es werden drei Anfragen verglichen, denen semantisch Grundanfrage 1.2 zu Grunde liegt.

Das ist die **Grundanfrage 1.3**:

```
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 Nachbarschaftsanfrage nutzt eine Zusammenfassung und findet die Anzahl der gemeinsamen Nachbarn von *Person*1 und *Person*2. Die erste Vergleichsanfrage findet die gemeinsamen Nachbarn mit der Core und Traversal API. Die zweite Vergleichsanfrage wird in Cypher formuliert und vermeidet den Gebrauch des In-Operators:

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

3.2.2 Hypothesen zum ersten Testlauf

Im Folgenden werden Hypothesen über die Performanz der Grundanfragen des ersten Testlaufes aufgestellt. In der Ergebnissen werden diese Hypothesen überprüft.

Für die **Grundanfragen 1.1.1-1.1.3** werden folgende drei Aspekte betrachtet:

1. Die Performanzunterschiede zwischen den Grundanfragen 1.1.1-1.1.3
2. Die Performanzunterschiede zwischen den Anfragen in Cypher und mit den APIs formuliert
3. Die Performanzunterschiede zwischen den Anfrage mit Traversierung über RELATIONSHIP3 und RELATIONSHIP2.

1: Die Bearbeitungszeit ist am höchsten bei Grundanfrage 1.1.3, da dort die meisten Pfade betrachtet werden. Bei Grundanfrage 1.1.1 ist die benötigte Zeit ist geringsten, da dort die Anzahl der zu betrachtenden Pfade minimal ist.
2: Wie in (Raj, 2015) beschrieben, wird es empfohlen die APIs für eine maximale Performanz zu nutzen, da diese APIs flexibler arbeiten und Cypher als hardwarefernere Sprache in Neo4j aufgefasst wird. Aus diesem Grund wird die Performanz bei den verwendeten Anfragen höher sein, wenn die Java Core API genutzt wird.
3: Da jede Person nur eine Relation vom Typ RELATIONSHIP2 besitzt, aber 2.500 vom Typ RELATIONSHIP3 werden die Anfragen mit RELATIONSHIP2 mindestens 1.000 mal schneller ausgeführt werden. Dies entspricht einer linearen Skalierung des Systems.

Für **Grundanfrage 1.2** werden folgende drei Aspekte betrachtet:

1. Die benötigten Berechnungszeiten der Anfragen
2. Der Performanzunterschied zwischen der Anfrage in Cypher und mit der Core API formuliert
3. Der Vergleich zwischen dem Filtern im Where-Prädikat und im Match-Prädikat

1: Alle 3 Anfragen werden eine sehr geringe Bearbeitungszeit von einigen Millisekunden besitzen, da der Gebrauch von Indizes und die konstante Zugriffszeit ein schnelles Finden von Einträgen erlaubt.
2: Wie bei der Grundanfrage 1.1 wird die Anfrage mit der Java Core API eine bessere Performanz besitzen, insbesondere weil keine Traversal API verwendet wurde.
3: Da der Optimierer, die meisten Bedingungen in das WHERE Prädikat verschiebt sollte nur ein minimaler bis nicht vorhandener Performanzunterschied zwischen den beiden Anfragen bestehen.

Für **Grundanfrage 1.3** wird der Unterschied der Laufzeiten bei semantisch äquivalenten Anfragen betrachtet.
Es wird keine hohe Differenz zwischen den Laufzeiten der in Cypher gestellten Anfragen geben, falls ein Unterschied bestehen sollte, wird dieser auf eine effiziente Implementierung des InOperators zurückgeführt. Die Anfrage in der Core API wird wie in Grundanfrage 1.1 die kürzeste Bearbeitungszeit besitzen.

3.3 Zweiter Testlauf

Der zweite Testlauf verwendet die Nachbarschaftsanfragen bis zu höheren Maximaldistanzen von zwei und drei, wodurch eine höherer Rechenaufwand erzeugt wird. Es wird der Performanzunterschied zwischen der Breitensuche, Tiefensuche und der bidirektionale Traversierung betrachtet. Die Anfragen sind komplexer im Vergleich zum ersten Testlauf und die Skalierbarkeit des Systems wird analysiert. Dadurch wird ein tieferes Verständnis der Funktionsweise von Neo4j erzeugt.

3.3.1 Grundanfragen zum zweiten Testlauf

Das ist die **Grundanfrage 2.1.1**:

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

Diese Nachbarschaftsanfrage findet die Anzahl aller Nachbarn, die über die Relation RELATIONSHIP3 in der Tiefe zwei erreicht werden. Grundanfrage 2.1.2 sucht über die gleiche Relation in der Tiefe drei. In den Vergleichsanfragen werden beide Anfragen in den APIs mit der Breitensuche und Tiefensuche ausgeführt. Es ergeben sich sechs Anfragen.

Anfrage	Name
Grundanfrage 2.1.1	Cypher - Tiefe 2
Grundanfrage 2.1.2	Cypher - Tiefe 3
Vergleichsanfrage 1	Tiefensuche - Tiefe 2
Vergleichsanfrage 2	Tiefensuche - Tiefe 3
Vergleichsanfrage 3	Breitensuche - Tiefe 2
Vergleichsanfrage 4	Breitensuche - Tiefe 3

TABELLE 3.2: Grundanfrage 2.1.1, 2.1.2 und Vergleichsanfragen

Das ist die **Grundanfrage 2.2**:

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

Diese Erreichbarkeits- und Mustervergleichsanfrage findet alle Nachbarn über die Relation RELATIONSHIP3 ausgehend von *Person*1 in der Tiefe zwei, die nicht direkt mit *Person*1 verbunden sind. Als Vergleichsanfragen 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))
```

Das ist die **Grundanfrage 2.3**:

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

Diese Erreichbarkeitsanfrage gibt die Länge des kürzesten Pfades zwischen *Person*1 und *Person*42 über die Relation RELATIONSHIP4 an, die maximale Länge ist auf drei limitiert. Der Standartalgorithmus shortestPath von Cypher wird genutzt. Als Vergleichsanfrage wird diese Anfrage erneut mit der Core API ausgeführt und als folgende Alternative ohne den Standartalgorithmus in Cypher formuliert:

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

Das ist Grundanfrage **Grundanfragen 2.4.1**:

```
MATCH (a)-[RELATIONSHIP4]->(b) RETURN COUNT(DISTINCT(b))
```

Die Anfrage traversiert über den gesamten Graphen, bis jeder Knoten einmal betrachtet wurde. In Cypher kann nur mit Tiefensuche traversiert werden und es ist nicht möglich einen anderer Suchalgorithmus anzugeben. Die Vergleichsanfrage traversiert in der Core und Traversal API mit Tiefensuche über den Graphen. Für weitere Grundanfragen, wird über die Graphen durch Breitensuche und durch bidirektionale Traversierung mit Breiten - und Tiefensuche traversiert. Es ergeben sich die in Tabelle 3.3 aufgelisteten Anfragen.

Anfrage	Name
Grundanfrage 2.4.1	Cypher - Einseitig -Tiefensuche
Vergleichsanfrage 1	Core API - Einseitig - Tiefensuche
Grundanfrage 2.4.2	Core API - Einseitig Breitensuche
Grundanfrage 2.4.4	Core API - Bidirektional - Tiefensuche
Grundanfrage 2.4.3	Core API - Bidirektional - Breitensuche

TABELLE 3.3: Grundanfragen 2.4.1 bis 2.4.4

3.3.2 Hypothesen zum zweiten Testlauf

Es werden Hypothesen zu den komplexen Grundanfrage aus dem zweiten Testlauf vorgestellt. Diese Hypothesen werden ebenfalls in Kapitel 4 überprüft.

Für die **Grundanfragen 2.1.1 und 2.1.2** werden die nachfolgenden drei Aspekte betrachtet:

1. Ein Performanzunterschied zwischen der Traversierung in den Tiefen zwei und drei

2. Die Performanzunterschiede zwischen der Breitensuche und Tiefensuche

3. Der relative Anteil an erreichten Personen im gesamten Graphen

1: Die Bearbeitungszeit der Grundanfrage 2.1.2 wird mindestens 1000 mal so hoch sein wie die Zeit der Grundanfrage 2.1.1, da pro Knoten in der Ergebnismenge weitere 2500 Knoten betrachtet werden müssen. Bei den Vergleichsanfragen mit der Breitensuche und Tiefensuche besteht die Annahme, dass die Suche in einer erhöhten Tiefe einen minimal höheren Rechenaufwand bedeutet, da beide Algorithmen eine Komplexität von O(V+E) besitzen.
2: Bei der Erhöhung von Tiefe zwei zu Tiefe drei steigt die Tiefe um eins an, aber die Breite erhöht sich pro Element in der Tiefe zwei um 2500. Der Graph besitzt ausgehend von dem Knoten der *Person*1 eine relativ hohe Breite im Vergleich zu der Tiefe, dadurch wird die Anfrage mit der Tiefensuche schneller ausgeführt werden.
3: Durch die Tiefe zwei 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 zwei werden mit hoher Wahrscheinlichkeit die meisten Personen bei der Traversierung zur Tiefe zwei erreicht.

Für **Grundanfrage 2.2** werden folgende zwei Aspekte betrachtet:

1. Ein Performanzunterschied bei semantischer Äquivalenz

2. Die allgemeine Performanz von dem Ausdruck *WHERE NOT*.

1: Wie in Grundanfrage 1.2 wird ein kleiner bis nicht vorhandener Unterschied auftreten, da der Optimierer einen der beiden Ausdrücke in den anderen überführen wird.
2: Da eine weitere Anfrage im WHERE-Prädikat gestellt wird und beide Ergebnismengen verglichen werden, wird die Ausführungszeit sehr hoch sein. Durch die Möglichkeit der Parallelisierung wird die Anfrage in der Core-API um ein vielfaches schneller beantwortet werden.

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

1. Die Berechnungszeit für den kürzesten Weg und die Länge von dieses Weges

2. Die Ausführungszeit in Cypher ohne Verwenden des Algorithmus

1: Der Pfad wird die Länge eins oder zwei besitzen, da die meisten Personen über RELATIONSHIP3/ RELATIONSHIP4 mit der Tiefe zwei erreicht werden können. Die Ergebnisse werden mit den Ausführungszeiten von Grundanfrage 2.1.1 zusammenhängen und es wird keine höhere Ausführungszeit als in Grundanfrage 2.1.1 erwartet.
2: Ausgehend von der Annahme, dass der Algorithmus lange optimiert wurde, wird die Anfrage mit dem Algorithmus eine schnellere Ausführungszeit aufweisen als die Anfrage ohne gegebenen Algorithmus. Die alternative Formulierung ist ein naiver Ansatz mit vielen aufwendigen Berechnungen, welcher eine große Ergebnismenge berechnet, die zu einem Ergebnis minimiert wird.

Für die **Grundanfragen 2.4.1-2.4.4** werden nachfolgenden Aspekte betrachtet:

1. Der Unterschied zwischen der bidirektionalen Suche und einseitiger Suche mit dem selben Algorithmus
2. Die relativen Berechnungszeit der 4 Traversierungs-Methoden in der Core API zueinander

1: Die bidirektionale Suche wird schneller ausgeführt, da potenziell durch zwei parallele Pfade weniger Berechnungsschritte zum Suchen eines Pfades, der durch den gesamten Graphen verläuft, benötigt werden.
2: Da über den gesamten Graphen ohne frühzeitigen Abbruch traversiert wird, besitzen die beiden Suchalgorithmen die gleiche Komplexität. Beide Algorithmen werden dementsprechend bei einer einseitigen Ausführung die gleiche Ausführungszeit besitzen. Bei der bidirektionalen Traversierung wird die Grundanfrage 2.4.4 mit der Tiefensuche schneller ausgeführt werden und insgesamt die schnellste Traversierung darstellen. Grundanfrage 2.4.3 wird die zweitschnellste Anfrage darstellen und danach die beiden einseitigen Traversierungen.

3.4 Dritter Testlauf

In dem dritten Testlauf werden alle Grundanfragen außer die Grundanfragen 2.4.3 und 2.4.4 auf dem DBMS OrientDB ausgeführt (OrientDB-Ltd., 2019). Da OrientDB keine bidirektionale Traversierung zur Verfügung stellt, konnten die Grundanfragen 2.4.3 und 2.4.4 nicht ausgeführt werden. Die benötigten Bearbeitungszeiten für die Anfragen auf OrientDB dienen als Referenz für eine bessere Performanzeinschätzung von Neo4j. In OrientDB wurden alle Anfragen in SQL formuliert und evaluiert. Für diesen Testlauf werden keine Hypothesen aufgestellt, da die Performanzunterschiede zwischen den beiden Systemen von viele Faktoren wie beispielsweise interner Implementierungen abhängig sind.

3.4.1 OrientDB

OrientDB ist eine noSQL, Multi-Modell-Datenbank von dem Entwickler OrientDB Ltd. Die Multi-Modell Eigenschaft beschreibt die Fähigkeit, Daten in mehreren Strukturen wie Dokumente, Graph, Schlüssel/Wert-Modell oder als Objekte zu modellieren (OrientDB-Ltd., 2019). Die Daten werden mittels API für eine unterstützte Sprache oder mit den Anfragensprachen Gremlin oder SQL manipuliert. Eine direkte Eingabe-Maske für Gremlin und SQL wird mit dem OrientDB Studio für den Browser zur Verfügung gestellt. OrientDB ist in der Community- und Enterprise-Version verfügbar. Die Enterprise-Version verfügt über zusätzliche Werkzeuge, die das Überwachen der Anfragen ermöglicht und erleichtert. Da die Enterprise-Version über keine Werkzeuge verfügt, die nicht auf der Community-Version laufen, die Einfluss auf die Performanz haben, wurde für diese Evaluation die Community Version 3.0.19 von OrientDB gewählt.

3.4.2 Unterschiede von OrientDB und Neo4j

Der primäre Unterschied zwischen Neo4j und OrientDB stellt die Verwaltung der Daten dar. Neo4j ist eine reine Graphdatenbank und OrientDB eine Multi-Modell-Datenbank. Dadurch ist es OrientDB möglich Daten auf mehr Arten in der Datenbank zu verwalten. Neo4j verwaltet alle Daten in einem reinem Graph. Dadurch ist die Varianz an möglichen Modellierungen der Daten in Neo4j eingeschränkter (Fernandes und Bernardino, 2018). Die beiden Systeme nutzen verschiedene Anfragesprachen und unterstützen unterschiedliche Programmiersprachen. Durch Java erstellt, besitzt der selbe Datensatz in OrientDB eine physische Gesamtgröße von 43,3 GB und ist ca. 58% kleiner als das Äquivalent in Neo4j.

Kapitel 4

Ergebnisse

4.1 Versuchsaufbau

Für die Ausführung der Anfragen wurde die Dell Precision Workstation T5500 verwendet. Das System besitzt 16 Prozessorkerne vom Typ Intel Xeon X5500 mit je 2,66 GHz und 8 MB Level3 Cache. Insgesamt stehen 72 GB DDR3 RAM mit 1333 MHz zur Verfügung und das System läuft auf Ubuntu 18.04. Alle Datensätze werden auf einer SEAGATE ST3250318AS Festplatte mit insgesamt 250 GB Kapazität, 8 MB Cache und 7200 Umdrehungen pro Minute gespeichert.
Jede Anfrage des ersten Testlaufes wurde 50 mal ausgeführt und die Anfragen des zweiten Testlaufes wurden 25 mal ausgeführt. In OrientDB wurden alle Anfragen 25 mal wiederholt. Als repräsentativer Wert für die Bearbeitungszeit einer Anfrage wurde das Arithmetische Mittel aus allen Wiederholungen einer Anfrage gewählt. Bei der Berechnung wurde die Zeit für die allererste Ausführung einer Anfrage nicht berücksichtigt, da durch das Caching eine unverhältnismäßig hohe Bearbeitungszeit beim ersten Ausführen beobachtet wurde. In OrientDB ist die erste Bearbeitungen der Anfrage im Durschnitt ca. 43% langsamer, als die Wiederholungen der Anfragen. In Neo4j ist die erste Bearbeitung mit Cypher ca. 40 % langsamer und mit den APIs ca. 17% langsamer.

4.2 Auswertung

Im folgenden Abschnitt werden Bearbeitungszeiten der Testläufe in Millisekunden tabellarisch präsentiert, nach jeweils 1000 Millisekunden wird ein "." zur Übersichtlichkeit eingefügt. Die Cypher-Code jeder Anfrage wird am Anfang jeder Analyse aufgeführt. Aufbauend auf den Zeiten werden die Hypothesen aus Kapitel 3 überprüft.

4.2.1 Ergebnisse des ersten Testlaufes

Grundanfrage 1.1:

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

Die Ergebnisse für Grundanfrage 1.1.1 werden in Tabelle 4.1 gezeigt. Durch die Bedingungen im Where-Prädikat entstehen im Vergleich zu den darauffolgenden Anfragen relativ hohe Bearbeitungszeiten. Die benötigte Bearbeitungszeit ist in allen drei Szenarien am längsten, wenn die Relationen in beide Richtungen betrachtet werden. Dies ist auf die hohe Anzahl der vorhandenen Relationen zurückzuführen. Bei der Anfrage in Cypher und der Core API für die RELATIONSHIP3 ist die Bearbeitungszeit bei der Suche für beide Richtungen länger als die Suchen für ausgehende und eingehende Kanten addiert. Für die Anfragen mit RELATIONSHIP3 ist die Zeit am geringsten, wenn die ausgehenden Kanten betrachtet werden.
Die Anfrage für RELATIONSHIP3 wird mit der Core API unter Verwendung der Traversal API in allen Fällen fast doppelt so schnell berechnet wie in Cypher. Dieses Verhalten wird bei den weiteren Grundanfragen ebenfalls erwartet, da die Core API sehr nah an den Kernfunktionen von Neo4j arbeitet.
Die Anfragen werden in Cypher für RELATIONSHIP2 etwas mehr als 1000 mal schneller als für REALTIONSHIP3 bearbeitet. Dies deutet auf eine gute Skalierbarkeit des Systems hin, da die Berechnungen beinahe linear zu der Anzahl der zu betrachtenden Relationen ansteigen.

Anfrage	RELATIONSHIP3	Core API	RELATIONSHIP2
Ausgehend	212.620	82.057	160
Eingehend	215.383	123.963	64
Beides	496-704	222.150	165

TABELLE 4.1: Ergebnisse der Grundanfrage 1.1.1

Grundanfrage 1.2:

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

Die beobachteten Zeiten für Grundanfrage 1.2 in Tabelle 4.2 sind im Vergleich zu den Zeiten der anderen Grundanfragen sehr gering. Durch die geringe Anzahl von 50004 Knoten,dem Verwenden von Indizes und der konstanten Zugriffszeit entstehen die geringen Zeiten.
Entgegen der Hypothese entsteht ein hoher, relativer Unterschied, wenn sich die Bedingung im Where-Prädikat statt im Match-Prädikat befindet. Die beobachtete Bearbeitungszeit bei der Anfrage mit der Bedingung in dem Where-Prädikat ist drei mal schneller, als bei der Anfrage mit der Bedingung im Match-Prädikat. Da der Cypher Query Optimizer alle Bedingungen des Match-Prädikates in ein Where-Prädikat verschiebt, benötigt dieser Optimierungsschritt selber eine Berechnungszeit. Für ein optimalen Gebrauch von Neo4j sollten dementsprechend alle Bedingungen direkt in einem Where-Prädikat formuliert werden, dies wird auch von Neo4j Inc. empfohlen (Neo4j-Inc., 2015b).
In dieser Anfrage wird die Traversal API nicht genutzt, sondern nur die Core API. Eine Kernfunktion namens $findNodes()$ übernimmt die Suche nach einem Knoten mit angegebenen Eigenschaften, ohne dass der Nutzer das Vorgehen spezifiziert.

Ohne Where	Mit Where	Core API
1,69	0,54	0,29

TABELLE 4.2: Ergebnisse der Grundanfrage 1.2

Grundanfrage 1.3:

```
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))
```

Bei Grundanfrage 1.3 wird die semantisch äquivalente Anfrage ca. 1,57 mal schneller bearbeitet. Dies ist auf die effiziente Implementierung des IN-Operators zurückzuführen. Wie bei Grundanfrage 1.1.1 ist die Bearbeitung mit der Core API schneller als in Cypher. Die Ergebnisse zu dieser Anfrage befinden sich in Tabelle 4.3.

Grundanfrage	Äquivalent	Core API
13,36	8,66	3,2

TABELLE 4.3: Ergebnisse der Grundanfrage 1.3

Die normierte Verteilungen aller Bearbeitungszeiten des ersten Testlaufes werden in Abbildung 4.1 dargestellt.

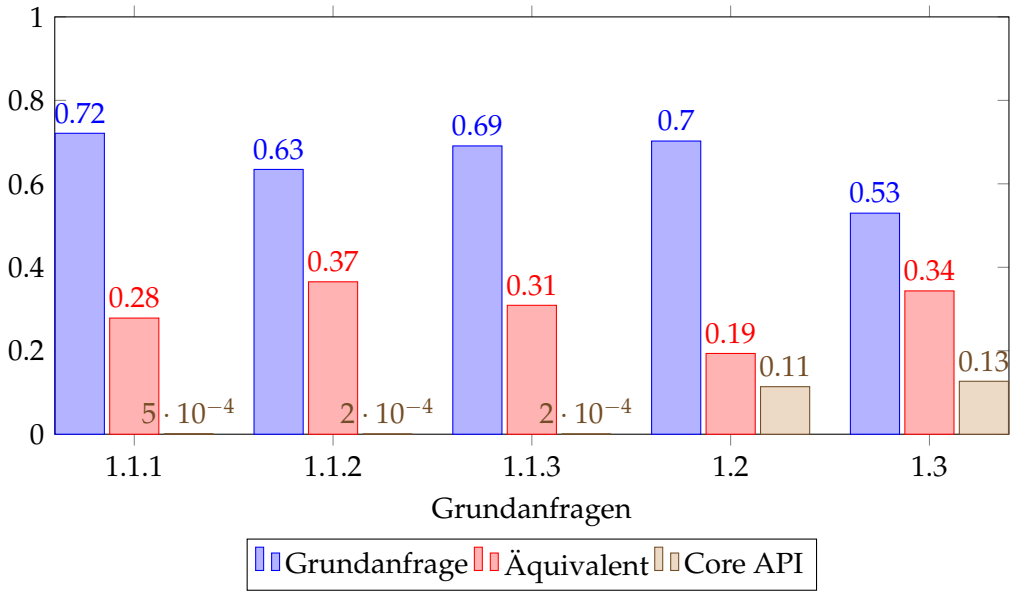

ABBILDUNG 4.1: Übersicht der Anfragen aus dem ersten Testlauf

4.2.2 Ergebnisse des zweiten Testlaufes

Grundanfrage 2.1.1-2.1.4:

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

In der Grundanfrage 2.1.1 werden bei der Traversierung in der Tiefe zwei ausgehend von Person1 99 % aller Knoten des Graphen erreicht. Bei der Tiefe drei werden 100 % der Knoten erreicht. Wie Tabelle 4.4 zeigt, wird mit Cypher die Traversierung in die Tiefe zwei ca. 64 mal schneller ausgeführt als die Traversierung in die Tiefe drei. In der Core API benötigt die Traversierung in die Tiefe drei mit der Breitensuche ca. 18 mal mehr Zeit als in die Tiefe zwei. Mit Verwendung der Tiefensuche benötigt die tiefere Traversierung ca. 1,2 mal mehr Zeit. Da die Anzahl der zu betrachtenden Knoten um den Faktor 2500 ansteigt, besteht in allen Fällen kein linearer Zusammenhang zwischen der Bearbeitungszeit und der Anzahl der Knoten, die betrachtet werden müssen.
Es wird bestätigt, dass die Tiefensuche schneller als die Breitensuche ausgeführt wird, die Tiefensuche in der Tiefe zwei wird ca.18,4 mal schneller als die Breitensuche ausgeführt. In der Tiefe drei wird die Tiefensuche ca. 277 mal schneller als die Breitensuche ausgeführt. Dies wird in Kapitel drei durch das Verhältnis der Breite zur Tiefe begründet.

Anfrage	Cypher	Breitensuche	Tiefensuche
Tiefe 2	3501	1545	84
Tiefe 3	223.556	28.318	102

TABELLE 4.4: Ergebnisse der Grundanfragen 2.1.1 und 2.1.2

Grundanfrage 2.2:

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

Wie Tabelle 4.5 zeigt, besteht ein geringer Unterschied zwischen Grundanfrage 2.2 und der äquivalenten Anfrage in Cypher. Beide Anfragen sind um ein vielfaches langsamer als die Anfrage, die die Core API verwendet. In der Core API Anfrage werden zwei Ergebnismengen vom Java-typ Set gebildet und durch den Aufruf der Funktion removeAll() werden die Elemente der einen Mengen aus der anderen Menge entfernt. Dies entspricht einer Realisierung von *Wherenot* in Java und wird schneller ausgeführt als der Ausdruck in Cypher. Alle drei Anfragen besitzen im Vergleich zu vorangegangen Anfragen eine extrem hohe Bearbeitungszeit.

Grundanfrage 2.2	Äquivalent	Core API
4.740.646	4.753.414	1.609

TABELLE 4.5: Ergebnisse der Grundanfrage 2.2

Grundanfrage 2.3:

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

Der kürzeste Pfade zwischen Person1 und Person2, welcher in Grundanfrage 2.3 untersucht wird, besitzt die Länge zwei. Die Grundanfrage 2.1.1 zeigt, dass bei einer Pfadlänge von zwei die meisten Knoten ausgehend von Person1 erreicht werden. Wie Tabelle 4.6 zeigt, ist die benötigte Bearbeitungszeit der Anfrage in Cypher unter Verwendung des gegebenen Algorithmus geringer als alle anderen Grundanfragen außer Grundanfrage 1.2. Erstmalig wird die Anfrage in Cypher schneller als die Formulierung in der Core API bearbeitet. Die Anfrage wird in Cypher 1,37 mal schneller beantwortet und der absolute Unterschied zwischen den Bearbeitungszeiten beträgt 3,1 ms. Durch die geringe absolute Differenz ist es nicht garantiert, dass der kürzeste Pfad immer am schnellste mit Cypher berechnet wird.
Die äquivalente Formulierung in Cypher besitzt die höchste aller beobachtetet Bearbeitungszeiten der gestellten Anfragen in Neo4j mit ca. 2,3 Stunden. Diese Formulierung ist eine naiver Ansatz ohne Verwendung des Algorithmus und besitzt viele Berechnungsschritten, die durch den Algorithmus entfallen. Zudem stoppt der Limit-Ausdruck nicht vorzeitig die Berechnung, sondern filtert die Ergebnismenge erst nach der Berechnung, dieses Verhalten ist mit der konservativen Stop-Strategie aus SQL zu vergleichen(Carey und Kossmann, 1997).

Grundanfrage 2.3	Äquivalent	Core API
8,4	8.370.298	11,5

TABELLE 4.6: Ergebnisse der Grundanfrage 2.3

Grundanfrage 2.4.1-2.4.4:

```
MATCH (a)-[RELATIONSHIP4]->(b) RETURN COUNT(DISTINCT(b))
```

Die Ergebnisse der Traversierungen des gesamten Graphen, wie in Grundanfrage 2.4.1 werden in Tabelle 4.7 dargestellt. In der Core API mit Verwendung der Traversal API ist die bidirektionale Traversierung mit Breitensuche ca. 1,15 mal schneller als der gleiche Algorithmus bei einer einseitige Traversierung. Bei der Tiefensuche ist die bidirektionale Suche ca. 1,04 mal schneller. In beiden Fällen sind die einseitigen Traversierungen langsamer als die bidirektionale Alternativen.
Trotz einer semantischen Äquivalenz von Grundanfrage 2.4.1 mit der Grundanfrage 2.1.2, bei der alle Knoten erreicht werden, ist eine höhere Bearbeitungszeit zu beobachten.

Anfrage	Zeit
Grundanfrage 2.4.1	284.594
Vergleichsanfrage 1	31.020
Grundanfrage 2.4.2	31.050
Grundanfrage 2.4.3	26.982
Grundanfrage 2.4.4	29.558

TABELLE 4.7: Ergebnisse der Grundanfragen 2.4.1-2.4.4

4.2.3 Ergebnisse des dritten Testlaufes

Für die folgenden Aussagen werden die Werte aus der Tabelle 4.8 betrachtet. Diese Tabelle stellt die Bearbeitungszeiten der Grundanfragen, ihrer Äquivalente in der Core API und in der OrientDB dar.
Die Grundanfragen 1.1.1-1.1.3 werden in Cypher durchschnittlich ca. 12 mal schneller ausgeführt als in der OrientDB, die relativ langen Bearbeitungszeiten bei beiden Systemen entstehen durch die Filter der Ergebnisse nach den Bedingungen. In allen Fällen benötigt die Grundanfrage 1.1.3, welche die eingehenden und ausgehenden Kanten betrachtet, die längste Zeit zur Bearbeitung. Dies zeigt, dass OrientDB und Neo4j durch die Filterung nach einer Bedingung im und der Anzahl der zu betrachtenden Kanten in ihrer Laufzeit beeinflusst werden.
Grundanfrage 1.2, welche solange über den Graphen traversiert bis ein gegebener Knoten gefunden ist, weist in OrientDB eine schnellere Bearbeitungszeit als Neo4j mit der Core API oder Cypher auf und wird ca. 3,7 mal schneller bearbeitet als die in Cypher gestellte Anfrage. Dies lässt sich mit einer effizienteren Implementierung der Indizies erklären. Die Anfrage wird in allen Fällen in weniger als einer Millisekunde beantwortet und bildet die am schnellsten bearbeitete Anfrage.
Grundanfrage 1.3, welche eine einfache Anfrage zum Finden von gemeinsamen Nachbarn darstellt, besitzt in Cypher und der Core API eine geringe Bearbeitungszeit von einigen Millisekunden. In OrientDB befindet sich Grundanfrage 1.3 mit einer benötigten Zeit von 83.955 ms in einer anderen Größenordnung und deutet darauf hin, dass der Vergleich von zwei Mengen miteinander in Neo4j effizienter implementiert ist. Dies ist auch in Grundanfrage 2.2 zu beobachten, für diese Anfrage muss bis in die Tiefe zwei traversiert werden und zwei Mengen müssen miteinander verglichen werden. Die Anfrage besitzt eine hohe Bearbeitungszeit von mehreren Stunden, obwohl das Traversieren in der Tiefe zwei, wie bei Grundanfrage 2.1.1 mit 72.092 ms zu beobachten, eine relativ geringe Zeit benötigt, der Vergleich der beiden Mengen bildet den rechenaufwendigeren Schritt.
Grundanfrage 2.1.1 wird mit Neo4j in Cypher ca. 20,6 mal schneller bearbeitet als in der OrientDB. Bei Grundanfrage 2.1.2 bildet OrientDB das schnellere System und bearbeitet die Anfrage ca. 2,1 mal schneller als Neo4j. Dies zeigt eine bessere Skalierung von OrientDB im Vergleich zu Neo4j mit Cypher beim traversieren des Graphen, mit Verwendung der Core API ist Neo4j fast um ein tausendfaches schneller als OrientDB. In OrientDB wird die Grundanfrage 1.1.2 ca.1,5 mal langsamer ausgeführt als Grundanfrage 1.1.1, mit Cypher wird diese Anfrage in Neo4j ca. 60,9 mal langsamer ausgeführt und mit der Core API ca. 1,2 mal langsamer. Die Grundanfrage 2.1.2 ist mit der Grundanfrage 1.2 die einzige Grundanfrage, die in OrientDB schneller bearbeitet wird, als in Neo4j mit Cypher.

Grundanfrage 2.2 konnte in dieser Evaluation nicht vollständig analysiert werden, da die erste Ausführung der Anfrage ohne Caching ungefähr 70 Stunden benötigt hat. Weitere Wiederholungen mit Caching waren nicht möglich, aber würden bei der durchschnittlichen Verkürzung der Bearbeitungszeit durch das Caching von ca. 43% ungefähr 40 Stunden benötigen .
Wie Neo4j benötigt OrientDB mit 37,6 ms eine kurze Zeit von einigen Millisekunden zur Bearbeitung von Grundanfrage 2.3, dies lässt sich durch eine effiziente Implementierung des shortestpath-Algorithmus erklären. Das Finden des kürzesten Pfades ist ein grundlegendes Problem bei Graphen und der Algorithmus wurde bereits in der OrientDB Version 1.7.8 vom 13. August 2014 veröffentlicht (OrientDB-Ltd, 2014). Durch die frühe Zurverfügungstellung ist es wahrscheinlich, dass der Algorithmus mit einer effizienten Implementierung veröffentlicht wurde oder über die Jahre optimiert wurde.
Für die Grundanfrage 2.4.2, welche mit Breitensuche über den gesamte Graphen traversiert, ist nur ein Vergleich zwischen OrientDB und der Core API von Neo4j möglich, da der Such-Algorithmus in Cypher nicht explizit angegeben werden kann und immer die Tiefensuche verwendet wird. Durch die gleiche Komplexität von O(V+E) bei der Tiefen- und Breitensuche, benötigen die Grundanfragen 2.4.1 und 2.4.2 in OrientDB eine gleich lange Bearbeitungszeit. In Cypher wird die Grundanfrage 2.4.1 ca. 2,7 mal schneller bearbeitet als die gleiche Anfrage in OrientDB. Mit Verwendung der Core API wird die einseitige Tiefen- und Breitensuche durchschnittlich ca. 24.8 mal schneller beantwortet als in OrientDB.
Unter der Berücksichtigung aller beobachteten Zeiten außer von Grundanfrage 2.2 werden die Anfragen mit der Core API durchschnittlich ca. 207 mal schneller bearbeitet als in OrientDB und mit Cypher ca. 1,9 mal schneller als in OrientDB.

Anfrage	Cypher	OrientDB	Core API
Grundanfrage 1.1.1	215.383	3.495.459	64
Grundanfrage 1.1.2	212.620	1.674.758	160
Grundanfrage 1.1.3	496.704	6.350.806	165
Grundanfrage 1.2	0,54	0,17	0,29
Grundanfrage 1.3	8,66	83.955	3,2
Grundanfrage 2.1.1	3.501	72.092	84
Grundanfrage 2.1.2	234.132	110.064	102
Grundanfrage 2.2	4.740.646	$\approx$ 40h	1.609
Grundanfrage 2.3	8,4	37,6	11,5
Grundanfrage 2.4.1	284.594	768.830	31.050
Grundanfrage 2.4.2	-	774.040	31.020

TABELLE 4.8: Ergebnisse der Grundanfragen auf Neo4j und OrientDB

4.3 Anwendungsszenario

Neoj4 Inc. stellt zahlreiche Nutzungen von Graphen in verschiedenen Themenkomplexen vor (Neo4j-Inc., 2019b). Für jeden Themenkomplex werden mehrere Beispielgraphen von Entwicklern, welche nicht für Neo4j arbeiten, zur Verfügung gestellt.

Das folgende Anwendungsszenario stellt den Zweck einer Graphdatenbank für Kinofilme dar.
In diesem Beispiel ist der Nutzer ein Betreiber eines Kinos, welches aktuelle Kinofilme ausstrahlt und spezielle Veranstaltungen anbietet. Bei diesen Veranstaltungen werden mehrere Filme von einer Kategorie oder einem bestimmten Schauspieler oder Regisseur gezeigt. Diese Veranstaltungen werden mehrere Wochen vor dem Veranstaltungstermin verplant und finden über mehrere Tage statt. Die Planung besitzt einen routinierten Ablauf, sodass viele Schritte effizient bearbeitet werden. Der zeitaufwendigste Arbeitsschritt, stellt das Finden von Filmen für das Veranstaltungsprogramm dar.
Aktuell stellt der Kinobetreiber eine Liste von Filmen für die Veranstaltung manuell zusammen, hierfür werden Filme ausgewählt, die der Betreiber selber kennt und es werden verschiedene Quellen aus dem Internet für eine Auswahl verwendet. Die Liste besitzt die Namen der Filme, zusätzliche Informationen wie die Länge eines Filmes sind durch die Vielzahl von Quellen nicht immer vorhanden. Fehlende Informationen müssen für eine optimale Planung des Kinoprogramms erneut herausgesucht werden und einige Filme müssen nach dem initialen Aufstellen der Liste aussortiert werden.
Für eine effizienteres Erstellen einer List von geeigneten Kinofilmen kann Neo4j verwendet werden. Unabhängig von dem Betriebssystem steht die kostenfreie Anwendung *Neo4j Desktop* zur Verfügung. Durch diese Anwendung für den Desktop ist die Bedienung von Neo4j über ein grafischen User Interface (GUI) möglich. Zusätzlich wird die Erweiterung *APOC* benötigt, welche per Mausklick in *Neo4j Dekstop* eingebunden werden kann. Mit dieser Erweiterung ist es möglich Daten mit verschiedenen Dateiformaten wie CSV oder JSON in die lokale Neo4j Datenbank zu importieren. Für das Thema Kino stehen im Internet zahlreiche, große Datensätze mit Filmen kostenfrei zur Verfügung (Kaggle, 2019). Nach dem Importieren eines solchen Datensatzes, kann der Nutzer durch die Eingabemaske eine Anfrage stellen, um so eine übersichtliche vollständige Liste von Filme zu erhalten. Folgende Anfrage erstellt eine Liste von 20 Filmen mit dem Thema Verbrechen:

```
MATCH(m:Film)
WHERE m.Genre = 'Verbrechen' or m.Plot contains 'Verbrechen'
RETURN m.Name as Name, m.Regie as Regie,
        m.Länge as Länge, m.Plot as Plot
LIMIT 20
```

Bei einem hohen Erfolg der Veranstaltung können ausgehend von den gezeigten Filmen Schauspieler aus diesen Filmen gefunden werden:

```
MATCH (s:Schauspieler)-[SpielteIn]->(m:Film)
WHERE m.Genre = 'Verbrechen' or m.Plot contains 'Verbrechen'
RETURN s.Name as Name, count(m) as Anzahl_der_Filme,
ORDER by Anzahl_der_Filme DESC
LIMIT 10
```

Es werden zehn Schauspieler gefunden, die in den meisten Filmen zum Thema Verbrechen mitgespielt haben. Diese Schauspieler sind potenzielle Kandidaten für eine

Veranstaltung, bei der nur Filme von diesen Schauspielern gezeigt werden. Durch Bedingungen bezüglich eines Attributes wie der Bewertung eines Filmen, kann die Auswahl weiter eingegrenzt werden.

```
MATCH (s:Schauspieler)-[SpielteIn]->(f:Film)
WHERE s.name = 'Christian Bale' AND f.Bewertung >= 5
RETURN f.Name as Name, f.Regie as Regie,
        f.Länge as Länge, f.Plot as Plot
LIMIT 20
```

Neo4j übernimmt die Auswertung eines Datensatzes, der in einem CSV oder JSON Format dargestellt wird und von dem Nutzer nicht schnell ausgewertet werden kann. Durch die effiziente Implementierung von Neo4j wird in kurzer Zeit eine vollständige Liste mit Filmen erstellt. Der Arbeitsschritt zum Erstellen einer Liste mit geeigneten Filmen wird damit beschleunigt und effizienter ausgeführt.

4.4 Limitierungen und zukünftige Arbeit

Die durchgeführte Evaluation nutzt mit OrientDB eine einzige Referenzdatenbank. Dies ist keine reine GDB und unterstützt eine erweiterte Verwaltung der Daten. Für eine näherer Einordnung der Performanz von Neoj4 ist ein Vergleich mit einer reinen GDB wie der Sparksee Database (Technologies, 2019) sinnvoll. Diese Datenbank konnte in der Evaluation auf Grund von Kommunikationsprobleme mit Sparsity technologies nicht verwendet werden. Durch einen Vergleich mit der Sparksee Database könnte die Performanz von zahlreichen Graphalgorithmen wie Zentralitäts-Algorithmen, zur Erkennung von Gruppen mit gemeinsamen Eigenschaften, überprüft werden. Dies ist mit OrientDB nicht möglich, da keine vergleichbare Implementierung der von Neo4j unterstützen Graphalgorithmen im vollen Ausmaß in OrientDB gegeben ist.
Ein Vergleich zwischen Neo4j und einer relationalen Datenbank, wie dargestellt in (Vicknair u. a., 2010), ist für eine weitere Beschreibung der Vor- und Nachteile einer GDB sinnvoll. Dieser Vergleich wurde aus zeitlichen Gründen nicht vollzogen. Die Referenz zur OrientDB hat einen Vergleich zu einer anderen noSQL-Datenbank dargestellt und so eine erste Einschätzung zu der Performanz von Neo4j ermöglicht.
Ein weitere zu betrachtender Aspekt, ist das Ausführen einer Neo4j Datenbank im Server-Modus auf mehreren Geräten verteilt. Durch diesen Aspekt können die ACID und CAP Eigenschaften untersucht werden und die Grenzen einer verteilten Neo4j-Datenbank getestet werden. Zudem kann ein Vergleich zwischen der Datenbank im eingebetteten Modus und dem Server-Modus durchgeführt werden. So können Vor- und Nachteile und Anwendungsszenarien spezifiziert werden.
Ein kleinerer nicht betrachteter Aspekt ist die temporale Eigenschaft von Neo4j. Der vorgestellte Datensatz besitzt ein Attribut vom Typen Date und stellt so eine temporale Datenbank dar, es fehlt ein Testlauf mit Anfragen zu diesem Attribute. Eine allgemeine Betrachtung des temporalen Verhalten der Datenbank ist für eine genauere Einordnung von Neo4j im Vergleich zu anderen temporalen Graphdatenbanken sinnvoll.
Die Anfragesprache GraphQL wird als standardisierte Anfragesprache für GDBs angesehen (Proponents, 2019). In der Zukunft sollte GraphQL auch in Neo4j getestet

werden, dies ist mit einer Erweiterung für Neo4j möglich und erfordert die Verwendung einer weiteren API. In dieser Evaluation wurde GraphQL nicht verwendet.

4.5 Fazit

Neo4j unterstützt mit den Java-Standardtypen und zeitlichen Typen wie Time, LocalTime oder DateTime eine Vielzahl von Datentypen, welche für die Modellierungen von Daten notwendig sind (Neo4j-Inc., 2019c). Durch die Zurverfügungstellung der Anwendungen Neo4j Dekstop und Neo4j Browser und einer dazu gegebenen Anfragesprache, werden die benötigten Kenntnisse zur Bedienung des System minimiert. Für eine spezifischeren Gebrauch der Datenbank, werden mehrere Programmiersprachen unterstützt. Durch diese Schnittstellen ist es mögliche, das System auf viele Weisen zu nutzen und eine große Zielgruppe kann das System für ihre Zwecke verwenden.
Der ersten beiden Testläufe zeigen, dass dem Gebrauch der APIs in fast allen Fällen zu einer besseren Performanz führt. Mit Verwendung der Traversal API besitzt der Nutzer eine flexible Möglichkeit die Performanz seiner Anfragen zu beeinflussen. So ist es möglich zwischen Breiten- und Tiefensuche zu wählen. Die am schnellsten bearbeiteten Anfragen besitzen keine gemeinsame Kategorie von den vorgestellten Kategorien aus Absatz 3.1.2. Die am langsamsten bearbeiteten Anfragen besitzen ebenfalls keine gemeinsamen Kategorien. Grundanfrage 2.2, welche die einzige Mustervergleichsanfrage ist, besitzt eine besonders hohe Bearbeitungszeit. Dementsprechend lässt sich nur für Mustervergleichsanfragen ein Bezug von Anfragen-Kategorie und Bearbeitungszeit in Neo4j erkennen, die anderen Kategorien weisen keinen solchen Bezug auf.
Wie die Tabellen 4.2 und 4.6 zeigen, können semantisch gleiche Anfragen in Cypher Unterschiede in ihrer Performanz aufzeigen. Dieses Verhalten ist in den APIs ebenfalls zu beobachten. Das System lässt sich durch eine hohe Anzahl von Schnittstellen leicht benutzen, aber das System effektiv zu nutzen und eine Anfrage effizient ausführen zu lassen, ist nicht trivial und erfordert technologische Kenntnisse. Durch Hinweise von Neo4j Inc. werden einige Möglichkeiten für das effiziente Ausführen gegeben (Neo4j-Inc., 2015b). Diese Hinweise befassen sich unter anderem mit dem Umformulieren der Anfragen, um die benötigten Arbeitsschritte des Optimierers zu minimieren. Wie in den Ergebnissen zur Grundanfrage 1.2 erläutert, deutet dies auf einen langsamen Optimierer hin.
Im Vergleich zu der Muli-model Datenbank OrientDB unter Verwendung von SQL besitzt Neo4j mit der Core API und Cypher ein bessere Performanz. Der verwendete Datensatz ist unter Neo4j ca. 2.4 mal größer als bei OrientDB. Allgemein besitzt Neo4j so eine bessere Performanz, aber die Speicherverwaltung unter Neo4j ist schlechter. Die Einschätzung der Performanz im globalen Kontext verglichen mit mehreren DBMS, wie in (Jouili und Vansteenberghe, 2013) ausgeführt, kann Rahmen dieses Evaluation nicht gegeben werden.

Danksagung

An dieser Stelle möchte ich all jenen danken, die durch ihre fachliche und persönliche Unterstützung zum Gelingen dieser Bachelorarbeit beigetragen haben.

Ebenso gilt mein Dank meinen Freunden und Bekannten für das Korrekturlesen. Zuletzt möchte ich noch all denjenigen danken, die in der Zeit der Erstellung dieser Arbeit für mich da waren, insbesondere meinen Freunden.

Schließlich danke ich meinen Freunden während der Studienzeit für drei sehr schöne Jahre in Lübeck.

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.

— (2018). „The Property Graph Database Model.“ In: *AMW*.

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

Campos, Alexander, Jorge Mozzino und Alejandro Vaisman (2016). „Towards temporal graph databases“. In: *arXiv preprint arXiv:1604.08568*.

Carey, Michael J und Donald Kossmann (1997). „On saying “enough already!” in sql“. In: *ACM SIGMOD Record* 26.2, S. 219–230.

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

Fernandes, Diogo und Jorge Bernardino (2018). „Graph Databases Comparison: AllegroGraph, ArangoDB, InfiniteGraph, Neo4J, and OrientDB.“ In: *DATA*, S. 373–380.

Folger, Ove (2019). *Code für Neo4j*. [Online; gesehen 02.10.19]. URL: https://github.com/McHektor123/BA/tree/master/src.

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.

Jouili, Salim und Valentin Vansteenberghe (2013). „An empirical comparison of graph databases“. In: *2013 International Conference on Social Computing*. IEEE, S. 708–715.

Kaggle (2019). *Kaggle: Your Home for Data Science*. [Online; gesehen 16.09.18]. URL: https://www.kaggle.com/.

Khurana, Udayan (2012). „An introduction to temporal graph data management“. In: *Computer Science Department, University of Maryland MA thesis. Online at https://www. cs. umd. edu/sites/default/files/scholarly_papers/Khurana_SchPaper_1. pdf*.

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.

Neo4j-Inc. (2015a). *Indexes*. [Online; gesehen 20.08.19]. URL: https://neo4j.com/docs/cypher-manual/current/schema/index/.

Neo4j-Inc. (2015b). *Introducing the new Cypher Query Optimizer*. [Online; gesehen 11.06.19]. URL: https://neo4j.com/blog/introducing-new-cypher-query-optimizer/.

— (2015c). *Understanding Neo4j's data on disk*. [Online; gesehen 13.06.19]. URL: https://neo4j.com/developer/kb/understanding-data-on-disk/.

— (2019a). *Neo4j APOC Library*. [Online; gesehen 18.06.19]. URL: https://neo4j.com/developer/neo4j-apoc/.

— (2019b). *Neo4j GraphGists*. [Online; gesehen 10.09.18]. URL: https://neo4j.com/graphgists/.

— (2019c). *Values and types*. [Online; gesehen 28.08.18]. URL: https://neo4j.com/docs/cypher-manual/current/syntax/values/.

OrientDB-Ltd (2014). *Released OrientDB 1.7.8 + new non-blocking backup*. [Online; gesehen 30.08.19]. URL: https://orientdb.com/released-orientdb-1-7-8/.

OrientDB-Ltd. (2019). *Overview of OrientDB*. [Online; gesehen 5.09.19]. URL: http://orientdb.com/docs/3.0.x/misc/Overview.html.

Papadopoulos, Apostolos N und Yannis Manolopoulos (2006). *Nearest Neighbor Search:: A Database Perspective*. Springer Science & Business Media.

Proponents, ISO Graph Query Language (2019). *GQL Standard*. [Online; gesehen 29.09.19]. URL: https://www.gqlstandards.org/.

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)*.

Strozzi, Carlo (1998). *NoSQL A Relational Database Management System*. [Online, gesehen 5.10.19]. URL: http://www.strozzi.it/cgi-bin/CSA/tw7/I/en_US/nosql/Home%20Page.

Tatarinov, Igor u. a. (2002). „Storing and querying ordered XML using a relational database system". In: *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*. ACM, S. 204–215.

Technologies, Sparsity (2019). *Sparsity Technologies: high-performance graph databases*. [Online; gesehen 10.07.18]. URL: http://www.sparsity-technologies.com/.

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.

Yannakakis, Mihalis (1990). „Graph-theoretic methods in database theory". In: *Proceedings of the ninth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*. ACM, S. 230–242.