Fakultät für Elektrotechnik und Informatik
Institut für Praktische Informatik
Fachgebiet Datenbanken und Informationssysteme

Evaluation eines Graph-Datenbanksystems

Bachelorarbeit
im Studiengang Informatik

Nicolas Tempelmeier
Matrikelnummer: 2841570

Prüfer: Prof. Dr. Udo Lipeck
Zweitprüfer: Dr. Hans Hermann Brüggemann
Betreuer: Prof. Dr. Udo Lipeck

7. August 2014

Inhaltsverzeichnis

Kapitel 1

Einleitung

In diesem Kapitel wird das Thema und die Motivation für diese Arbeit vorgestellt. Danach wird der Aufbau der Arbeit erläutert.

1.1 Motivation

In der Informatik werden für unterschiedliche Anwendungsfälle unterschiedliche Datenstrukturen verwendet, da diese individuelle Vor- und Nachteile mit sich bringen. Insbesondere Operationen können unterschiedliche Kosten verursachen. Möchte man zum Beispiel den Vorgänger eines Elementes in einer Liste ermitteln, verursacht dies in einer einfach verketteten Liste lineare Kosten in Abhängigkeit der Größe der Liste, in einer doppelt verketteten Liste jedoch nur konstante. Graphdatenbanken gehören zur Gruppe der NoSQL-Datenbanken und folgen diesem Gedanken.

NoSQL steht für „not only SQL“. Eine NoSQL-Datenbank kann also SQL implementieren, muss es aber nicht. Es existieren vier Kerngruppen von NoSQL-Systemen: Wide Colum Stores, Document Stores, Key/Value-Datenbanken und Graphdatenbanken. Außerhalb dieser Kerngruppen gibt es allerdings noch weitere nicht relationale Systeme [EFH+11]. Eine offizielle Definition für NoSQL-Datenbanken existiert nicht, unter [NOR] findet sich jedoch eine inoffizielle. Unter anderem beinhaltet diese Schema-Freiheit, hohe Skalierbarkeit, ein nicht relationales Datenmodell und eine Open-Source-Lizenz.

Mit der Entwicklung des Web 2.0 entstand der Bedarf, große Mengen an unstrukturierten Daten zu speichern. In der Praxis führte dies in Verbindung mit relationalen Datenbanken zu Problemen. Dies wiederum führte zu einer gesteigerten Popularität von NoSQL-Systemen [EFH+11].

Während viele NoSQL-Datenbanken möglichst schnell auf sehr großen Datenmengen

operieren sollen, werden Graphdatenbanken vor allem dafür benutzt hochgradig verknüpfte Daten zu speichern, bei denen die Informationen vor allem in der Beziehung der Daten untereinander liegen. Diese Art von Information kann auf einem relationalen Datenbanksystem kostenaufwändige Joins verursachen. Auf diesen Informationen sollen Graphdatenbanken eine effiziente Ausführung von Anfragen ermöglichen.

In dieser Arbeit soll das Verhalten der Graphdatenbank Neo4j evaluiert werden. Dazu werden direkte Vergleiche zu einer relationalen Datenbank gezogen und die Funktionalität von Neo4j beschrieben. Dafür werden bestehende relationale Datenbanken in Graphdatenbanken transformiert und Queries aus bestehenden Anwendungen zu Queries an Neo4j übersetzt.

1.2 Über den Aufbau dieser Arbeit

In Kapitel 2 wird die Graphdatenbank Neo4j vorgestellt. Dazu werden das Property-graph-Modell erläutert und technische Grundlagen von Neo4j beschrieben. Des Weiteren wird die Neo4j-eigene Anfragesprache „Cypher“ vorgestellt.

In Kapitel 3 wird ein Algorithmus erläutert, der relationale Datenbanken in Graphdatenbanken transformiert. Dazu werden zunächst die einzelnen Relationen klassifiziert und anschließend deren Einträge in Graphelemente umgewandelt.

In Kapitel 4 wird die Implementierung dieser Transformation vorgestellt. Zunächst wird der Entwurf einer Anwendung beschrieben, die den Algorithmus umsetzt. Danach wird auf wichtige Mechanismen der Implementierung eingegangen. Zuletzt wird die Bedienung anhand eines Beispiels erläutert.

In Kapitel 5 findet die eigentliche Evaluation von Neo4j statt. Dazu werden die Ausdrucksfähigkeit von Cypher, die Kosten typischer Graph-Operationen und das zeitliche Verhalten konkreter Queries diskutiert.

Kapitel 6 bietet eine Zusammenfassung der Ergebnisse sowie einen Ausblick für weitere Fragestellungen.

Kapitel 2

Grundlagen von Neo4j

In diesem Kapitel sollen die Hauptkonzepte von Neo4j vorgestellt werden. Dazu werden zunächst die Idee der Graph-Datenbanken vorgestellt. Danach wird die Modellierung von Daten mit Hilfe sogenannter Propertygraphen erläutert. Des Weiteren wird die Architektur und die Inbetriebnahme von Neo4j behandelt. Zuletzt folgt eine Vorstellung der Neo4j-eigenen Anfragesprache „Cypher“.

2.1 Motivation

Graphdatenbanken gehören zur Gruppe der NoSQL-Datenbanken. Während die meisten Arten der NOSQL-Datenbanken sich hauptsächlich damit beschäftigen, große Mengen an Daten zu speichern, beschäftigen sich Graphdatenbanken hauptsächlich damit, Informationen in der Struktur der Daten, also in der Art, wie die Daten verknüpft sind, zu speichern [EFH+11].

Intuitiv lassen sich viele Sachverhalte als Graph modellieren. Graphdatenbanken bemühen sich, diese Modellierung nahezu eins zu eins in eine Datenbank zu übertragen. Dazu werden die Daten in Graphdatenbanken in Form eines sogenannten Propertygraphen gespeichert. Außerdem sind viele Graphdatenbanken schemalos. Dies erlaubt es Benutzern, unkompliziert auf Änderungen zu reagieren und den Datensatz zu erweitern. Insgesamt besteht also nur ein relativ geringer Modellierungsaufwand. Auf der anderen Seite verspricht man sich für eine bestimmte Art von Anfragen eine bessere Performance als mit relationalen Systemen. In diesen verursachen Joins einen Großteil der Kosten [RWE13]. Den Joins gegenüber stehen in Graphdatenbanken Verbindungen mit Kanten. Das Bestimmen des Zielknotens einer einzelnen Kante verursacht im Idealfall nur konstante Kosten. Ein weiterer Vorteil gegenüber relationalen Systemen ist, dass jeweils nur ein Teil der Daten betrachtet werden muss, da diese direkt verbunden sind und Operationen

auf dem gesamten Datensatz nicht zwangsläufig erfolgen müssen.

Graphdatenbanken werden daher vermehrt in Bereichen des Semantic Web, der Geoinformation, der Bioinformatik und von sozialen Netzwerken eingesetzt.

2.2 Propertygraph-Modell

Graphen werden in der Mathematik über zwei Mengen definiert: Der Menge der Knoten V und der Menge der Kanten $E \subseteq (V \times V)$. Dabei können die Kanten sowohl gerichtet als auch ungerichtet sein. Abbildung 2.1 zeigt einen gerichteten Beispielgraphen. Für diesen Graphen lautet die Knotenmenge $V = \{1, 2, 3\}$ und die zugehörige Kantenmenge $E = \{(1,\ 3), (2,\ 1), (3,\ 2)\}$. Ein Multigraph ist ein Graph, in dem zwei Knoten auch durch mehrere Kanten verbunden werden können. Außerdem sind auch Schleifen erlaubt, Ziel- und Ursprungsknoten einer Kante können somit auch identisch sein.

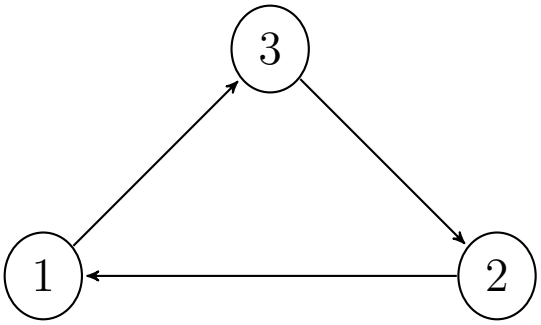

Abbildung 2.1: Gerichteter Beispielgraph

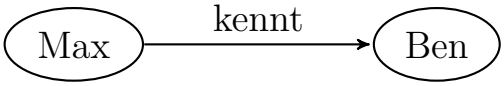

Abbildung 2.2: Darstellung einer einfachen Beziehung

Intuitiv ist diese Struktur gut geeignet, um Beziehungen darzustellen. Abbildung 2.2 zeigt eine einfache derartige Darstellung. Aus dieser Intuition hat sich das Propertygraph-Modell entwickelt [TPG]. Dieses Modell erweitert einen gerichteten Multigraphen so, dass zusätzliche Informationen in den Knoten und Kanten des Graphen abgespeichert werden können. Die Daten werden in Form von Schlüssel/Wert-Paaren für Knoten und Kanten modelliert, den sogenannten Properties. Außerdem verfügen Kanten über ein sogenanntes Label, das den Kantentyp festlegt. Abbildung 2.3 zeigt einen solchen Propertygraphen. Das Propertygraph-Modell von Neo4j erweitert dieses Modell noch, indem es auch Labels für die Knoten vergibt. Ein Knoten kann dabei mehrere Labels besitzen. In diesem Modell werden die Knoten „Nodes“ und die Kanten „Relationships“ genannt. Abbildung 2.4 zeigt einen Propertygraph, wie er in Neo4j benutzt wird.

Abbildung 2.3: Ein Propertygraph

Abbildung 2.4: Ein Neo4j Propertygraph

Nun soll anhand eines Beispiels gezeigt werden, wie ein Propertygraph vernetzte Daten abbilden kann. Dafür wird Schritt für Schritt die Modellierung eines Propertygraphen erläutert. Gegeben sei folgende Situation:

Es existiert eine Gruppe von Freunden: Anna, Julia, Karsten und Mark. Julia ist 20, Karsten 23 und Mark 22 Jahre alt. Das Alter von Anna ist unbekannt. Karsten und Mark spielen zusammen Fußball im Verein FC Altdorf, der seit 1991 existiert. Karsten seit 2006 und Mark seit 2007. Dort haben sie Tim kennen gelernt, der auch in diesem Verein Fußball spielt. Mark mag ihn aber nicht. Anna und Julia spielen Basketball im Verein SV Neustadt, der erst seit 2008 existiert. Sie spielen in diesem Verein seit dessen Gründung. Bei der Modellierung soll auch das Geschlecht der Personen beachtet werden.

Zunächst soll die Situation der vier Freunde abgebildet werden. Jede Person wird durch einen Node dargestellt, daher bekommt jeder dieser Nodes das Label „Person“. Die Properties der Personen-Nodes sind demnach der Name, das Geschlecht und (falls bekannt) das Alter. Da alle untereinander befreundet sind, geht von jedem der vier Nodes eine „befreundet“-Relationship zu den jeweils drei anderen Nodes. Abbildung 2.5 zeigt einen Propertygraphen, der diese Elemente enthält.

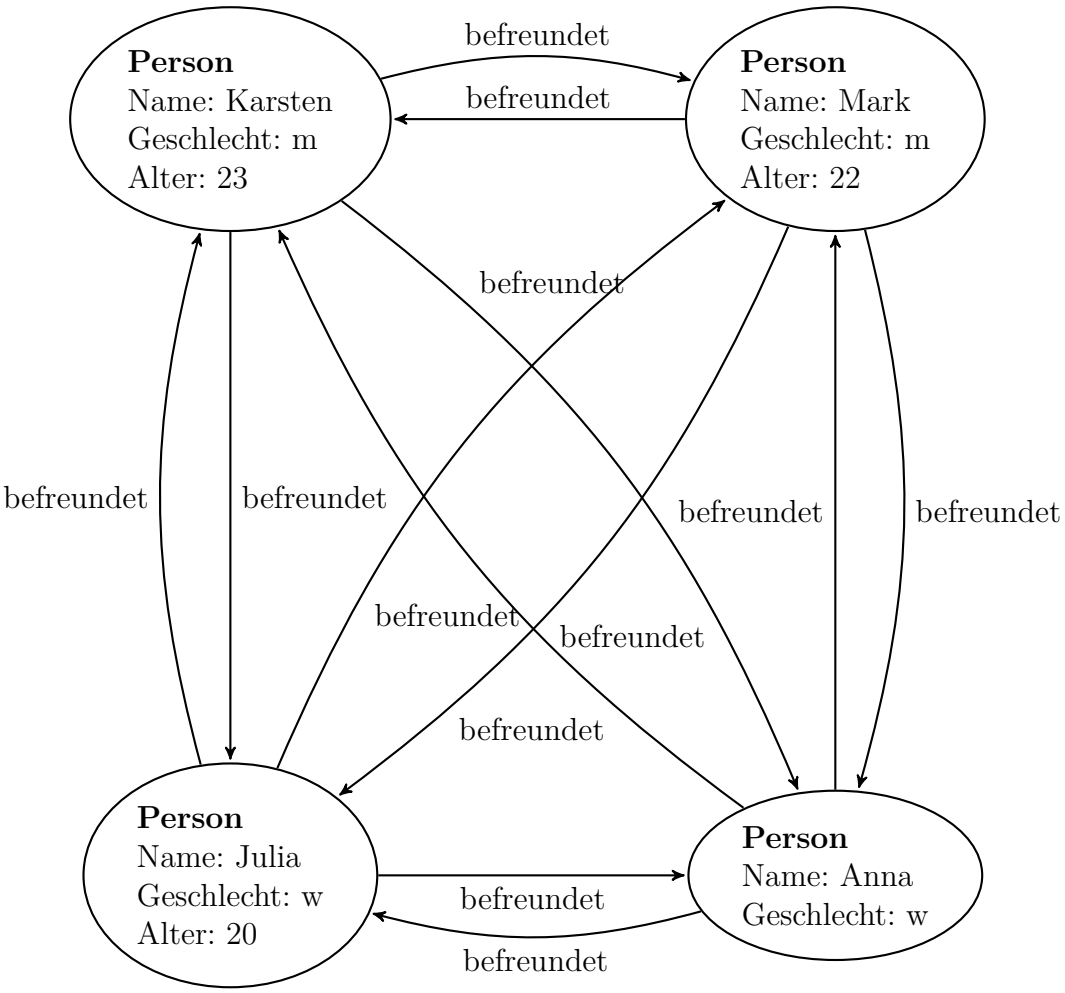

Abbildung 2.5: Die ersten vier Nodes und ihrer Relationships

Auf eine ähnliche Weise können nun die Nodes ergänzt werden, die die Vereine repräsentieren. Abbildung 2.6 zeigt den entsprechenden Propertygraphen.
Hierbei tragen die eingefärbten Relationships folgende Properties:

spielt_in: seit: 2006
spielt_in: seit: 2007
spielt_in: seit: 2008
spielt_in: seit: 2008

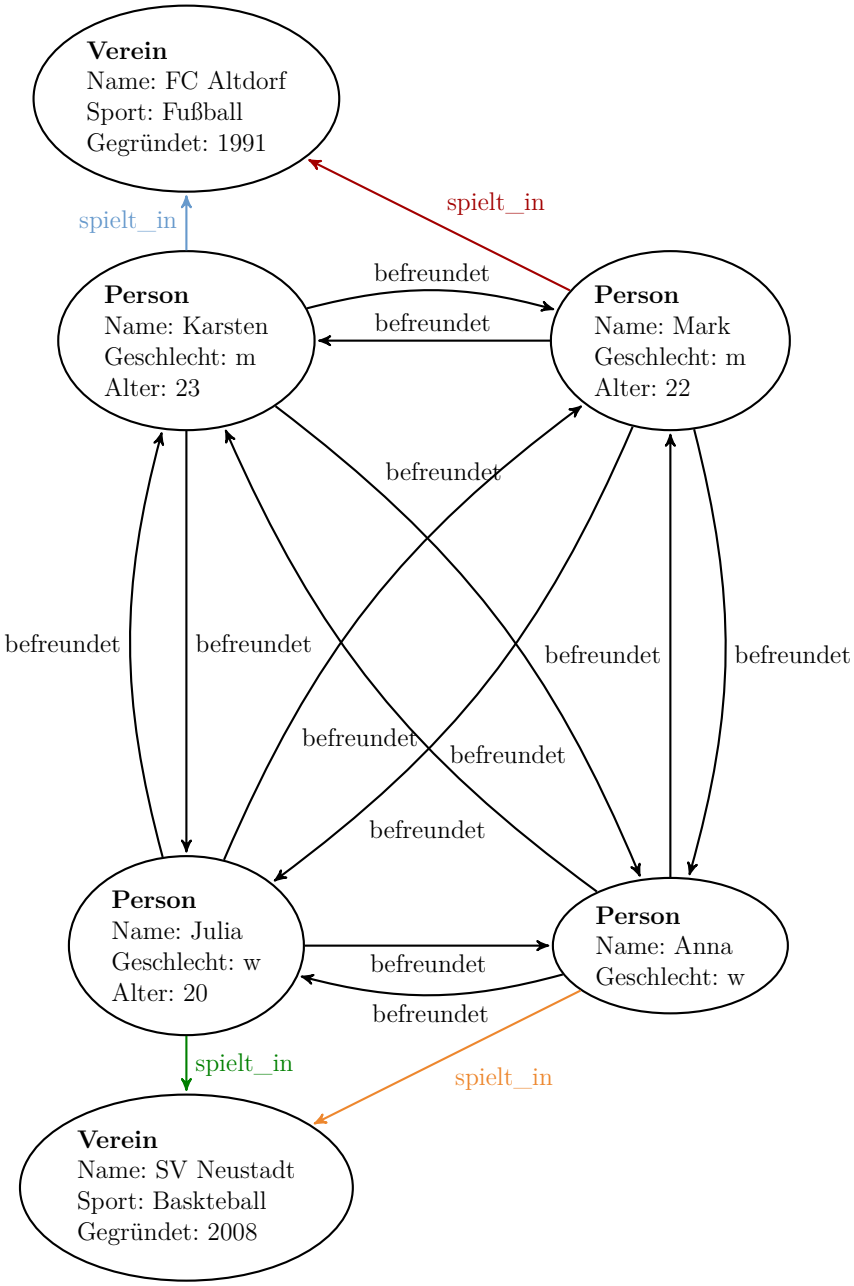

Abbildung 2.6: Propertygraph mit ergänzten Vereinen

Im letzten Schritt muss noch Tim, der Bekannte aus dem Fußballverein, hinzugefügt werden. Da Tim Mark und Karsten kennt und diese beiden Personen Tim kennen, werden vier „kennt“-Relationships hinzugefügt. Zusätzlich wird von Mark zu Tim eine „mag_nicht“-Relationship erstellt. Abbildung 2.7 zeigt den vollständigen Propertygraphen.

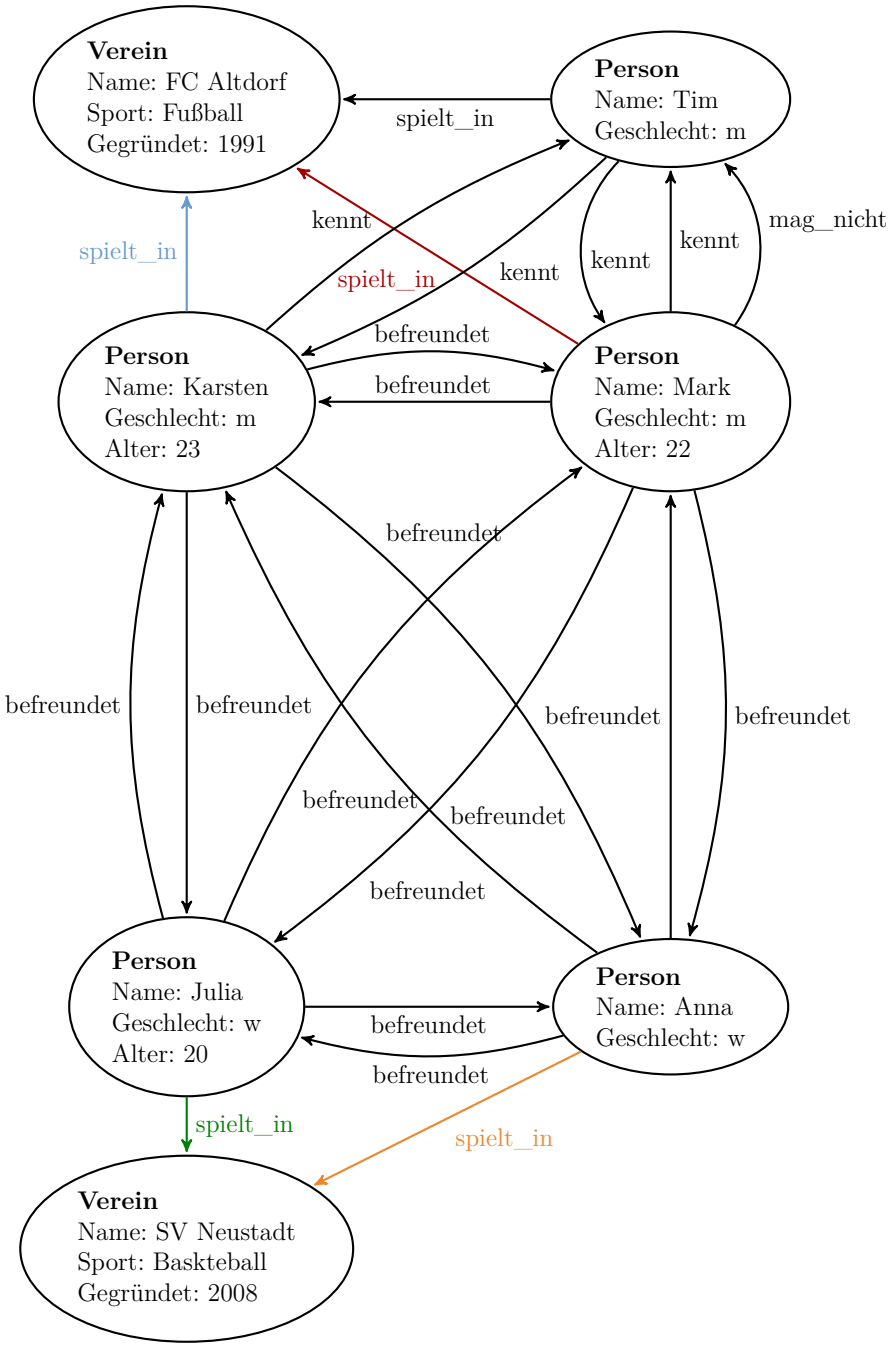

Abbildung 2.7: Der vollständige PropertyGraph
(ausschließlich oben genannter Relationship-Properties)

2.3 Neo4j

In diesem Abschnitt werden technische Grundlagen, Aufbau und Konfiguration von Neo4j erläutert.

2.3.1 Technische Grundlagen

Neo4j kann auf zwei verschiedene Arten betrieben werden: Embedded und als dedizierter Server. Die Embedded-Variante kann nur für Java-Applikationen genutzt werden. Beim Embedded-Betrieb wird Neo4j vollständig in die eigene Applikation eingebettet, so dass Neo4j unter der gleichen Prozess-ID wie die eigene Applikation läuft. In diesem Modus kann Neo4j direkt über die Java-API angesprochen werden, mit deren Hilfe man beliebige Abstraktionsebenen nutzen kann.

Im dedizierten Betrieb ist Neo4j ein eigenständiger Prozess. In diesem Modus muss es über eine andere API angesprochen werden. Neo4j bietet dazu eine REST-API an. „REST“ steht dabei für „Representational State Transfer“ und bezeichnet eine Software-Architektur für Hypermedia-Anwendungen, insbesondere Webanwendungen [Fie00]. Diese beinhaltet unter anderem das Konzept der Zustandslosigkeit, so dass Adressen eindeutig Ressourcen zugeordnet werden. Unter Neo4j wird die REST-API über HTTPS-Anfragen angesprochen. Unter [NCS] werden außerdem noch Interfaces für verschiedene Sprachen angeboten, die jedoch meist auf die REST-API aufbauen.

Der Propertygraph von Neo4j ist schemalos, das heißt es können beliebige Elemente in den Graphen eingefügt werden. Diese Schemalosigkeit kann durch sogenannte „unique-Constraints“ eingeschränkt werden. Existiert so ein Constraint für eine Label/Property-Kombination, darf jeder Wert dieser Property in nur jeweils genau einem Node mit diesem Label vorkommen.

Intern erhalten Nodes und Relationships jeweils eine eindeutige ID. Die IDs sind fortlaufend und beginnen bei Null.

Auch ein Transaktionsmanagement wird von Neo4j angeboten. Dieses steht sowohl unter der Java- als auch unter der REST-API zur Verfügung. Im Gegensatz zu den meisten NoSQL-Datenbanken genügt das Transaktionsmanagement dabei den ACID-Eigenschaften [Neo14]. Diese beinhalten:

- Atomarität: Entweder werden alle Operationen einer Transaktion ausgeführt oder keine.
- Konsistenz (consistency): Nach jeder Transaktion ist die Datenbank in einem konsistenten Zustand, sofern sie vorher in einem konsistenten Zustand war.

- Isolation: Während einer Transaktion können andere Vorgänge nicht auf modifizierte Daten zugreifen.
- Beständigkeit (durability): Ist eine Transaktion abgeschlossen, sind die Daten dauerhaft gespeichert.

Um diese Eigenschaften, insbesondere die Isolation, zu erfüllen verwendet Neo4j write-Locks. Sollte durch einen write-Lock ein Deadlock entstehen, wird ein Rollback für die verursachende Transaktion ausgeführt und es tritt eine Exception auf.

Neo4j ist in zwei Versionen erhältlich: Der kostenlosen Community- und der kostenpflichtigen Enterprise-Edition. In der Enterprise-Edition sind zusätzlich zu den in der Community-Edition enthalten Features noch folgende Komponenten enthalten [N4S]:

- High-Performance Object Cache
- High-Availability-Clustering
- Hot-Backups
- Advanced Monitoring

Der High-Perfomance Object Cache ist eine alternative Cacheimplementierung, die mehrere Konfigurationsmöglichkeiten liefert und dementsprechend effizienter arbeiten kann.

Das High-Availability-Clustering bezeichnet das Betreiben einer verteilten Graphdatenbank. Dafür wird ein Master-Slave-Netz aufgebaut. Dabei besitzt jede Instanz den vollständigen Datensatz. Write Queries, die an eine Slave-Instanz gerichtet werden, werden in zwei Phasen ausgeführt: Zunächst werden sowohl auf der Slave- als auch auf der Master-Instanz Locks gesetzt. Danach werden die Queries auf beiden Instanzen gleichzeitig ausgeführt. Nach Ausführen der Query auf dem Master werden die restlichen Slave-Instanzen vom Master ausgehend synchronisiert. Sollte die Master-Instanz ausfallen, wird eine geeignete Slave-Instanz zum neuen Master bestimmt [Neo14].

Hot-Backups sind Backups, die zur Laufzeit der Datenbank angelegt werden können. Das Benutzen eines Backups, das auf diese Weise angelegt wurde, erfordert jedoch einen Neustart von Neo4j. Mit Hilfe des Advanced Monitorings können zur Laufzeit Kenngrößen der Datenbank angezeigt werden.

2.3.2 Speicherarchitektur

In [Neo14] werden die Daten, die von Neo4j auf der Festplatte gespeichert werden, beschrieben. In [Lin12] wird deren Inhalt erläutert. Dabei handelt es sich hauptsächlich

um drei Dateien: „neostore.nodestore.db“, „neostore.relationsshipstore.db“ und „neostore.propertystore.db“. Jede dieser Dateien ist in weitere atomare Einheiten unterteilt, die sogenannten Records. Die Struktur eines Records des gleichen Typs ist dabei immer identisch.

In der neostore.nodestore.db werden die Nodes, also die Knoten abgespeichert. Die Records dieser Datei enthalten Pointer zur ersten Relationship und einen Pointer zur ersten Property. Die eigentlichen Daten, also die Werte der Properties, werden nicht in den Knoten gespeichert.

In der neostore.relationsshipstore.db werden die Relationships, also die Kanten gespeichert. In den Records der Relationships werden Pointer zu Ziel- und Ursprungsknoten gespeichert. Außerdem finden sich sowohl für den Ursprungs- als auch den Zielknoten Pointer zur jeweils nächsten und vorherigen Relationship. Pro Relationship werden also Pointer zu insgesamt vier weiteren Relationships gespeichert. Dadurch ist jede Relationship Teil von zwei doppelt verketteten Listen: Die erste enthält die Relationships des Ursprungsknotens und die zweite die Relationships der Zielknoten. Zusätzlich wird auch noch der Typ der Relationship und ein Pointer zur ersten Property der Relationship gespeichert.

In der neostore.propertystore.db werden die Properties, also die eigentlichen Daten gespeichert. Dabei wird nicht zwischen Properties für Nodes und für Relationships unterschieden. Es existieren zusätzlich eigene Dateien für Array- und Stringproperties. Dies ist der Größe der String- und Arrayeinträge geschuldet. Die Namen der entsprechenden Dateien lauten neostore.propertystore.db.strings und neostore.propertystore.db.arrays. In dem Record einer Property findet sich der Name und der Wert der Property sowie ein Pointer zur jeweils nächsten Property.

Abbildung 2.8 zeigt die Speicherhierarchie in Neo4j. Eine Besonderheit ist, dass zwei verschiedene Caches existieren. Wird auf einen Node oder eine Relationship zugegriffen, die momentan nicht geladen ist, wird zuerst der Objekt-Cache aufgerufen. In diesem Cache werden die Nodes und die Relationships in einer Form gespeichert, die Graph-Traversierungen, also die Routenfindung innerhalb eines Graphen, erleichtert. Wird kein passender Eintrag im Objekt-Cache gefunden, wird der Dateisystem-Cache aufgerufen. Die Einträge diese Caches haben die gleiche Form wie die Records der Dateien auf der Festplatte. Findet sich auch in diesem Cache kein passender Eintrag, wird zuerst vom Arbeitsspeicher, danach von der Festplatte gelesen. Die Einträge auf der Festplatte haben das oben erläuterte Format.

2.3.3 Indexe

Indexe sind seit Version 1.0 Teil von Neo4j. Mittlerweile stehen in Neo4j stehen drei Arten von Indexen zur Verfügung: Legacy-, Auto- und Schema-Indexe. Dabei sind Auto-Indexe

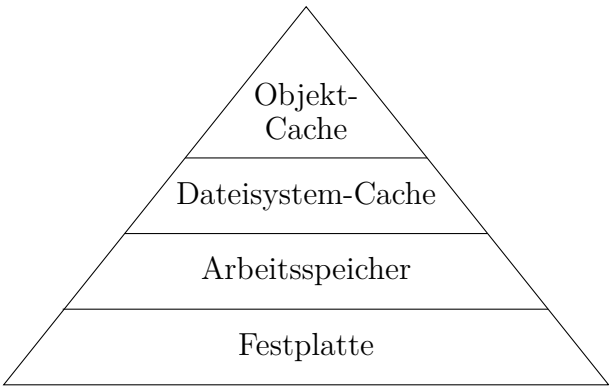

Abbildung 2.8: Speicherhierarchie in Neo4j

ein Spezialfall von Legacy-Indexen.

Legacy- und Auto-Indexe
Legacy-Indexe sind die älteste Art von Indexen in Neo4j. Wie der Name schon nahelegt, wird nicht mehr empfohlen, diese Art von Indexen zu verwenden [Neo14]. Legacy-Indexe können für Nodes oder Relationships angelegt werden. Innerhalb eines Indexes dürfen jedoch ausschließlich Nodes oder ausschließlich Relationships enthalten sein. Für jeweils einen Index muss ein eindeutiger Name vergeben werden. Nodes/Relationships müssen einzeln zu einem bestehenden Legacy-Index hinzugefügt werden. Dabei muss für jedes Element, das dem Index hinzugefügt wird, ein Schlüssel/Wert-Paar angegeben werden. Dieses kann dabei völlig unabhängig vom Node/Relationship gewählt werden. Innerhalb eines Legacy-Indexes können also auch verschiedene Schlüssel existieren.

Auto-Indexe sind automatisiert angelegte Legacy-Indexe. Neo4j stellt jeweils genau einen Auto-Index für Nodes und genau einen für Relationships bereit. In der Datei „neo4-server.properties“ können die Auto-Indexe konfiguriert werden. Für jeden Auto-Index kann dort angegeben werden, nach welchen (einzelnen) Properties indiziert werden soll. Ist ein Auto-Index konfiguriert und wird ein passendes Element dem Graphen hinzugefügt, wird es auch automatisch dem Auto-Index hinzugefügt. Auto-Indexe werden jedoch nicht rückwirkend angelegt. Bestehende Nodes/Relationships müssen dem Auto-Index manuell hinzugefügt werden.

Legacy/Auto-Indexe können mit Hilfe der start-Klausel benutzt werden (siehe 2.4.3 - start) .

Schema-Indexe
Schema-Indexe wurden in der Version 2.0 eingeführt und sind die empfohlene Variante, Indexe anzulegen [Neo14]. Schema-Indexe können ausschließlich für Nodes angelegt werden. Schema-Indexe können nicht benannt werden. Ein einzelner Index kann für Nodes mit einem bestimmten Label nach einer einzelnen Property angelegt werden. Nodes, die dem Graphen hinzugefügt werden, werden automatisch indiziert. Diese Indexe werden

allerdings auch rückwirkend für schon bestehende Nodes angelegt.

Schema-Indexe werden automatisch benutzt, wenn sie vorhanden sind. Sie müssen also nicht in besonderer Weise in den Queries adressiert werden. Zum Anlegen von Schema-Indexen siehe Abschnitt 2.4.5 - create.

2.3.4 Inbetriebnahme

Um Neo4j zu betreiben, sind diese technischen Voraussetzungen notwendig [Neo14]:

- Intel Core i3 CPU oder vergleichbar
- 2 GB Arbeitsspeicher
- 10 GB SATA Festplattenspeicher
- ext4-Dateisystem
- Oracle Java 7 oder OpenJDK 7

Zur Inbetriebnahme der Neo4j Community-Edition als dedizierter Server unter einem Linux-System sind folgende Schritte erforderlich:

1. Download der Community Editon von [N4W].
2. Entpacken der Dateien in ein lokales Verzeichnis.
3. In diesem Verzeichnis den Befehl

   ```
   ./bin/neo4j console
   ```

   ausführen, um Neo4j zu starten.

Die Hauptkonfiguration von Neo4j im dedizierten Betrieb erfolgt über die Dateien „neo4j-server.properties“ und „neo4j.properties“. In der neo4j-server.properties können Einstellungen vorgenommen werden, die den dedizierten Betrieb betreffen. Dies umfasst unter anderem den Port, unter dem Neo4j erreichbar ist, ob das HTTPS-Protokoll genutzt werden soll, und das Verzeichnis, in dem die Daten gespeichert werden. In der neo4j.properties-Datei können Konfigurationen erfolgen, die die Datenbank selber betreffen. Unter anderem können dort das Memory-Mapping, die Version des Cypher-Parsers und Auto-Indexe konfiguriert werden.

Da Neo4j in Java implementiert ist, läuft jede Neo4j-Instanz innerhalb einer Java Virtual Machine (JVM). Damit ist die Performance von Neo4j auch immer von der Konfiguration seiner JVM abhängig. Ein wesentlicher Parameter der JVM ist die Heap-Size. Diese sollte beim Start ausreichend groß gewählt werden. [Neo14] empfiehlt für Datenbanken mit bis zu 10 Millionen Elementen einen Heap von 512MB. Dabei sollte beachtet werden, dass Memory-Mapping zusätzlichen Platz im Heap beansprucht und der Heap je nach Konfiguration des Memory-Mappings zusätzlich vergrößert werden muss.

2.4 Die Anfragesprache Cypher

2.4.1 Einleitung

Cypher ist eine Neo4j-eigene deklarative Anfragesprache [Neo14]. Sie war erstmals in der Neo4j Version 1.4 enthalten, die im Juni 2011 veröffentlicht wurde [NRN]. Mit Hilfe von Cypher können verschiedene Graph-Operationen ausgedrückt werden. Dabei reicht der Funktionsumfang von der einfachen Wiedergabe von Nodes und Relationships über Graph-Traversierungen bis hin zu komplexen Operationen wie zum Beispiel der Bestimmung eines kürzesten Weges.

In dem nachfolgenden Abschnitt sollen die wichtigsten Sprachelemente von Cypher vorgestellt werden. Dazu werden unter anderem Queries an eine Beispieldatenbank formuliert. Diese Beispieldatenbank soll den Propertygraphen aus Abbildung 2.7 enthalten.

Beim Design von Cypher wurde Wert darauf gelegt, dass es einfach zu lesen und zu verstehen ist [RWE13]. Daher werden in Cypher Nodes und Relationships durch Schriftzeichenpiktogramme repräsentiert. So werden Nodes als zwei runde Klammern dargestellt, die einen Bezeichner einschließen, beispielsweise „(n)“. Relationships entsprechen einem Pfeil aus zwei Bindestrichen und je nach Richtung einem „größer als“- oder „kleiner als“-Zeichen, „-->“ oder „<--“. Bezeichner für Relationships können innerhalb eckiger Klammern in der Mitte der Pfeile angeben werden, wie zum Beispiel „-[r]-> “. Es ist möglich, die Richtung von Relationships zu ignorieren, das notiert man: „--“.

2.4.2 Struktur einer Query

Cypher orientiert sich in seiner Querystruktur an SQL. So besteht auch eine Query in Cypher aus einer Folge von Klauseln [Neo14]. Die Klauseln an sich unterscheiden sich jedoch teilweise erheblich von den SQL-Klauseln. Auch eine Verschachtelung von Cypher-Queries ist nicht möglich.

In Cypher besitzen Queries, die unterschiedliche Funktionen erfüllen, unterschiedliche

Strukturen. Angelehnt an [CRE] werden im Folgenden Ausdrücke in Backus-Naur-Form für die Strukturen von Read-Only Queries, Write-Only Queries und Read-Write-Queries angegeben.[1]

Read-Only:

```
(((MATCH [WHERE]) | (OPTIONAL MATCH [WHERE]))
[WITH [ORDER BY] [SKIP] [LIMIT]])*
RETURN [ORDER BY] [SKIP] [LIMIT]
```

Write-Only:

```
(CREATE [UNIQUE] | MERGE)*
[SET|DELETE|REMOVE|FOREACH]*
[RETURN [ORDER BY] [SKIP] [LIMIT]]
```

(Vergleiche [CRE])

Read-Write

```
((MATCH [WHERE] | OPTIONAL MATCH [WHERE])
[WITH [ORDER BY] [SKIP] [LIMIT]])*
(CREATE [UNIQUE] | MERGE)*
[SET|DELETE|REMOVE|FOREACH]*
[RETURN [ORDER BY] [SKIP] [LIMIT]]
```

Diese Ausdrücke stellen zwar nur gültige Schlüsselwortfolgen dar, mit ihnen können jedoch nicht alle gültigen Cypher-Queries gebildet werden. So ist es zum Beispiel möglich, nach einer create-Klausel eine with-Klausel einzufügen.

Das Ergebnis einer Query wird über die return-Klausel definiert. Mit der return-Klausel können Elemente, die einen Identifier besitzen (siehe Abschnitt 2.4.3, „match-Klausel und Patterns“) und sogenannte Expressions wiedergegeben werden (siehe [Neo14]).

Im Nachfolgenden werden die einzelnen Klauseln nach Funktion geordnet erläutert. Dabei ist zu beachten, dass die Funktion einer Klausel nicht unbedingt dem Typ einer Query entspricht. So werden zum Beispiel lesende Klauseln nicht nur in Read-Only-Queries verwendet.

[1] Die Zeichen (,), |, *, [,] stellen dabei Metazeichen dar.

2.4.3 Lesende Klauseln

match-Klausel und Patterns
Die match-Klausel wird benutzt, um sogenannte Patterns des Graphen zu identifizieren, und ist das Hauptwerkzeug für Graph-Traversierung [Neo14]. Bei den Patterns handelt es sich um Strukturen, die aus Knoten und Kanten bestehen. Ein einfaches Pattern sieht wie folgt aus:

```
()-->()
```

match ermittelt alle Nodes und Relationships, die dem Pattern entsprechen und mit einem sogenannten Identifier versehen sind. Identifier sind Namen, die für Nodes und Relationships vergeben werden können. Das obige Pattern mit Identifiern versehen entspricht diesem Pattern:

```
(a)-[b]->(c)
```

In einem Pattern können zusätzlich zur Graphstruktur auch Labels und Properties für Nodes und Relationships angegeben werden:

```
(a: Label1 {prop1: Wert1, prop2: Wert2})
-[b: Label2 {prop3: Wert3}]->(c: Label3)
```

In dem obigen Pattern entsprechen Label1 bis Label3 den Werten der Labels, prop1 bis prop3 den Namen der Properties und Wert1 bis Wert3 den Werten der Properties.

Innerhalb einer match-Klausel können auch mehrere Patterns verwendet werden, wie zum Beispiel in der folgenden Klausel:

```
match (a)-->(b), (b)-->(c)
```

Mehrere aufeinanderfolgende Relationships gleicher Art können in einem Pattern mit Hilfe eines Sterns ausgedrückt werden:

(a)-[:TYP*$i..j$]->(b)

Mit $i, j \in \mathbb{N}, i \leq j$. Hierbei gibt i die kleinste und j die größte Anzahl an, wie oft eine Relationship vorkommen soll. i oder j können auch weggelassen werden, in diesem Fall werden Standardwerte für i oder j eingesetzt. Der Standardwert für i lautet Eins, der für j Unendlich.

where
Mit der where-Klausel können weitere Selektionen ausgeführt werden. In Verbindung mit der match-Klausel finden diese Selektionen nicht nach der Graph-Traversierung statt, sondern können als Ergänzung des Patterns interpretiert werden [Neo14]. In der where-

Klausel können einschränkende Bedingungen an die Daten formuliert werden. Diese können mit den logischen Operatoren „AND“, „OR“ und „NOT“ verknüpft werden. Eine Beispielquery, die eine where-Klausel enthält, sieht wie folgt aus:

```
MATCH (a: Person)-[:spielt_in]->(b:Verein)
WHERE a.Geschlecht = 'w' AND b.Sport = 'Basketball'
RETURN a, b
```

Diese Query gibt alle weiblichen Personen zurück, die in einem Verein Basketball spielen. Des Weiteren können auch noch Patterns in where-Klauseln eingebaut werden. Dies ist insbesondere deswegen interessant, da so auch Patterns negiert werden können:

```
MATCH (a: Person)-[:spielt_in]->(:Verein), (b: Person)
WHERE NOT a-[:kennt]->b AND b.Name = 'Tim'
RETURN a
```

Dieser Query liefert alle Personen zurück, die in einen Verein spielen und Tim nicht kennen. Neue Identifier können in where-Klauseln allerdings nicht hinzugefügt werden, daher wird in der match-Klausel noch eine Person b benötigt.

Generell empfiehlt es sich, in where-Klauseln nur Selektionen durchzuführen, die keine einfache Gleichheit überprüfen. Gleichheitsselektionen sollten in der oben vorgestellten Patternschreibweise innerhalb der match-Klausel erfolgen, da Neo4j für Queries in dieser Schreibweise besser parametrisierte Ausführungspläne erstellen kann. In der where-Klausel können alle übrigen, insbesondere negierte und Bereichsselektionen, erfolgen.

optional match
Die optional match-Klausel kann benutzt werden, um optionale Patterns anzugeben. Während bei der match-Klausel nur Objekte zurückgegeben werden, die dem angegebenen Pattern entsprechen, wird bei optional match in diesem Fall null für die Elemente des Patterns zurückgegeben, für die es keine Entsprechung im Graph gibt. Die Query

```
MATCH (a: PERSON)
OPTIONAL MATCH n-[:spielt_in]->(b: Verein)
WHERE b.Sport = 'Fußball'
RETURN a.Name, b.Name
```

würde folgendes Ergebnis liefern:

a.Name	b.Name
Karsten	FC Altdorf
Mark	FC Altdorf
Julia	null
Anna	null

start
Mit Hilfe der start-Klausel kann ein Anfangspunkt für ein Pattern angegeben werden. Als Anfangspunkt kann entweder die ID eines Nodes, einer Relationship oder eine Legacy-Indexsuche angegeben werden. Wird als Startpunkt ein Node angegeben, der nicht ins Pattern passt, ist das Ergebniss der match-Klausel leer, unabhängig davon, ob das Pattern an anderer Stelle im Propertygraphen existiert. Wir nehmen an, dass ein Legacy-Auto-Index nach der Property „Name“ existiert. Dann liefert die Query

```
START n=node:auto_index(Name='Mark')
MATCH n-[:befreundet]->(m: Person)
RETURN n.Name, m.Name
```

folgendes Ergebnis zurück:

n.Name	m.Name
Mark	Karsten
Mark	Anna
Mark	Julia

Sei 0 die ID des Nodes der Person mit dem Namen „Mark“ und 1 die ID des Nodes der Person mit dem Namen „Anna“. Dann ist das Ergebnis folgender Query der Node der Person mit dem Namen „Tim “, weil Mark nur Tim „kennt“.

```
START n=node(0)
MATCH n-[:kennt]->(m: Person)
RETURN m
```

Folgende Query hingegen liefert eine leere Ergebnismenge:

```
START n=node(1)
MATCH n-[:kennt]->(m: Person)
RETURN m
```

2.4.4 Aggregationsfunktionen

count
Mit Hilfe der count-Funktion können Zeilen des Ergebnisses gezählt werden. Die Query

```
MATCH (n: Person)
RETURN count(*)
```

würde das Ergebnis 5 liefern. Beim Zählen ist es auch möglich, Duplikate zu ignorieren.

Folgende Query bestimmt die Anzahl verschiedener Geschlechter und liefert das Ergebnis 2:

```
MATCH(n: Person)
RETURN count(DISTINCT n.Geschlecht)
```

sum
Die sum-Funktion summiert alle numerischen Werte einer Spalte. In folgender Query wird das Durchschnittsalter aller Personen berechnet, zu denen ein Alter bekannt ist:

```
MATCH (n: Person)
WHERE has(n.Alter)
RETURN sum(n.Alter)/count(n)
```

Die Funktion has(property) gibt dabei true für Nodes und Relationships zurück, die die angegebene Property besitzen, sonst false.

collect
Mit der collect-Funktion werden Ergebnisse in einer Liste zusammengefasst. Mit dieser Query können alle Freunde einer Person bestimmt werden:

```
MATCH(n: Person)-[:befreundet]->(m: Person)
RETURN n.Name, COLLECT(m.Name)
```

Das Ergebnis sieht wie folgt aus:

n.Name	collect(m.Name)
Karsten	Mark, Anna, Julia
Mark	Karsten, Anna, Julia
Julia	Karsten, Mark, Anna
Anna	Karsten, Mark, Julia

2.4.5 Schreibende Klauseln

create und set
Die create-Klausel fügt neue Nodes oder Relationships in den Propertygraphen ein. Eine neue Person kann zum Beispiel mit dieser Query eingefügt werden:

```
CREATE(n: Person)
```

Es ist sinnvoll, die create-Klausel mit einer set-Klausel zu verbinden. Diese wird benutzt, um Properties eines Nodes zu erstellen oder zu verändern.

```
CREATE(n: Person)
SET n.Name='Lisa', n.Geschlecht='w'
```

Die set-Klausel kann jedoch auch in einem anderen Kontext, zum Beispiel nach einem match, benutzt werden, um bestehende Properties zu verändern oder neue Properties zu einem bestehenden Node hinzuzufügen. Folgende Query ändert eine Property:

```
MATCH (n: Verein)
WHERE n.Name='SV Neustadt'
SET n.Sport='Hockey'
```

Diese Query fügt eine neue Property zu einem Node hinzu:

```
MATCH (n: Person)
WHERE n.Name='Karsten'
SET n.Leibspeise='Spaghetti'
```

Relationships werden ebenfalls unter Benutzung der create-Klausel erzeugt:

```
MATCH(n: Person), (m: Person)
WHERE n.Name='Tim' AND m.Name='Lisa'
CREATE n-[:befreundet]->m, m-[:befreundet]->n
```

Mit der obigen Query werden befreundet-Relationships zwischen dem Personen-Node mit dem Namen Tim und dem Personen-Node mit dem Namen Lisa in beide Richtungen eingefügt.

Auch Schema-Indexe können mit Hilfe der create-Klausel erzeugt werden. Folgende Query legt einen Index für Personen-Nodes nach dem Property „Alter“ an.

```
CREATE index on :Person(Alter)
```

delete
Mit der delete-Klausel können Nodes und Relationships aus dem Propertygraphen entfernt werden. Ein Node kann mit folgender Query entfernt werden:

```
MATCH (n: Person)
WHERE n.Name='Mark'
DELETE n
```

Die obige Query funktioniert nur, wenn der zu entfernende Node keine ein- oder ausgehenden Relationships besitzt. Besitzt er Relationships, müssen diese ebenfalls gelöscht werden [Neo14]:

```
MATCH (n: Person)-[r]-()
WHERE n.Name='Mark'
DELETE n, r
```

Es ist auch möglich, ausschließlich Relationships zu löschen:

```
MATCH (n: Person)-[r: mag_nicht]->(m: Person)
WHERE n.Name = 'Mark' AND m.Name = 'Tim'
DELETE r
```

remove
Properties können nicht mit der delete-Klausel entfernt werden. Um eine Property zu entfernen, muss die remove-Klausel benutzt werden.

```
MATCH (n: Verein)
REMOVE n.Sport
```

Mit der obigen Query wird die Sport-Property aus jedem Verein-Node entfernt.

merge
Die merge-Klausel stellt eine Kombination aus der match- und der create-Klausel dar. Innerhalb einer merge-Klausel kann ein Pattern angegeben werden. Sollte das Pattern im Graphen nicht gefunden werden, werden die fehlenden Nodes und Relationships in den Graphen eingefügt. Zusätzlich kann die Klausel um einen „ON CREATE SET“ oder um einen „ON MATCH SET“ Teil erweitert werden. Mit diesen Teilen können Properties der Elemente des angegebenen Patterns abhängig davon gesetzt werden, ob das gesamte Pattern im bestehenden Propertygraphen gefunden wurde oder erstellt werden musste. Wurde das Pattern gefunden, wird der „ON MATCH SET“-Teil ausgeführt, musste es erstellt werden, der „ON CREATE SET “-Teil. Wir nehmen an, dass im Graphen kein Node mit dem Label „Verein“ und einem Property mit dem Namen „Name“ und dem Wert „TC Altdorf“ existiert. Folgende Query werde zwei mal ausgeführt:

```
MERGE (n: Verein {Name: 'TC Altdorf'})
ON CREATE SET n.Mitglieder=1
ON MATCH SET n.Mitglieder=n.Mitglieder+1
```

Nach der ersten Ausführung existiert ein Verein-Node mit Namen TC Altdorf und der Property Mitglieder mit Wert 1. Nach der zweiten Ausführung wurde die Anzahl der Mitglieder um 1 erhöht.

2.4.6 Weitere Sprachelemente

order by
Das Sortieren von Ergebnissen kann mit der order by-Klausel bewirkt werden. Die Ergebnisse werden standard-mäßig aufsteigend sortiert. Möchte man eine absteigende Reihenfolge erhalten, muss ein „DESC “an das Ende der Klausel angehängt werden. null-Werte werden dabei als größtes Element interpretiert. Ergebnisse können nach einer oder mehreren Properties sortiert werden. Die folgende Query sortiert alle Personen alphabetisch nach ihren Namen:

```
MATCH(n: Person)
RETURN n.Name
ORDER BY n.Name
```

In dieser Query werden die Vereine absteigend nach ihrem Gründungsdatum sortiert:

```
MATCH (n: Verein)
RETURN n.Name
ORDER BY n.Gegründet DESC
```

limit
Mit Hilfe der limit-Klausel kann die Anzahl der Zeilen des Ergebnisses beschränkt werden. Die folgende Query liefert nur die Namen der ersten drei Personen:

```
MATCH(n: Person)
RETURN n.Name
LIMIT 3
```

skip
Die skip-Klausel bewirkt ein Überspringen von Zeilen des Ergebnisses. Da keine Garantie für die Reihenfolge von Ergebnissen existiert, empfiehlt es sich, die skip-Klausel mit einer order by-Klausel zu verbinden. Mit folgender Query wird der in alphabetischer Reihenfolge erste Name von Personen übersprungen, die restlichen werden ausgegeben:

```
MATCH(n: Person)
RETURN n.Name
ORDER BY n.Name
SKIP 1
```

case
Mit case können die Ergebnisse manipuliert werden. Je nach Wert der Ergebnisse können unterschiedliche Aktionen erfolgen. Der Syntax eines case-Blocks sieht wie folgt aus:

```
CASE
WHEN predicate THEN result
[WHEN ...]
[ELSE default]
END
```

(Vergleiche [Neo14])

In folgender Query wird case benutzt um sicherzustellen, dass ein gültiger Eintrag für das Geschlecht einer Person existiert:

```
MATCH (n: Person)
RETURN
CASE
WHEN n.Geschlecht = 'm'
THEN 'männlich'
WHEN n.Geschlecht = 'w'
THEN 'weiblich'
ELSE 'ungültig'
END
```

filter
Mit der filter-Funktion können Selektionen auf sogenannten Collections ausgeführt werden. Collections sind Mengen von Elementen. Sie können zum Beispiel über die collect-Funktion erzeugt werden. Der Syntax der filter-Funktion sieht wie folgt aus:

```
FILTER(identifier in collection WHERE predicate)
```

(Vergleiche [Neo14])

Die folgende Query gibt zu einer Person alle Freunde zurück, die ein anderes Geschlecht besitzen:

```
MATCH(n: Person)-[:befreundet]->(m: Person)
RETURN n,
       FILTER(x in COLLECT(m) WHERE x.Geschlecht <> n.Geschlecht)
```

with
Die with-Klausel kann anstelle einer return-Klausel eingefügt werden, um ein Zwischenergebnis zu erzeugen. Dies kann zum Beispiel dazu genutzt werden, Ergebnisse umzubenennen, Aggregierungen auszuführen oder Zwischenergebnisse zu sortieren. Nach einer with-Klausel sind nur noch die Identifier gültig, die in der with-Klausel genannt werden.

Nicht genannte Identifier können also nach einer with-Klausel erneut vergeben werden. Mit dieser Query wird der Verein bestimmt, der schon am längsten existiert:

```
MATCH (n: Verein)
WITH max(n.Gegründet) as maxDat
MATCH (n: Verein)
WHERE n.Gegründet = maxDat
RETURN n
```

Pfade und foreach
Innerhalb einer match-Klausel können sogenannte „Paths“ deklariert werden. Ein Path ist eine Folge von Nodes, die das angegebene Pattern (inklusive where-Klauseln) erfüllen. Folgende Query definiert einen einfachen Path:

```
MATCH p=(a: Person)-[:kennt*..]->(b: Person)
WHERE a.Name='Karsten'
RETURN p
```

Das Ergebnis dieser Query lautet:

p
(Name: Karsten, Geschlecht: m, Alter: 23), (Name: Tim, Geschlecht: m)
(Name: Karsten, Geschlecht: m, Alter: 23), (Name: Tim, Geschlecht: m), (Name: Mark, Geschlecht: m, Alter: 22)

Auf die Nodes oder Relationships eines Paths kann mit Hilfe der foreach-Klausel zugegriffen werden. Dafür muss entweder die nodes(path)- oder relationships(path)-Hilfsfunktion verwendet werden. Diese geben eine Collection von Nodes bzw. Relationships zurück. Um alle Nodes innerhalb eines Paths zu markieren, kann folgende Query verwendet werden:

```
MATCH p=(a: Person)-[:kennt*..]->(b: Person)
FOREACH (n in nodes(p) | SET n.marked=true)
```

Innerhalb einer foreach-Klausel können folgende Klauseln benutzt werden: create, set, delete und foreach.

Kapitel 3

Transformation

3.1 Motivation

In diesem Kapitel wird ein Algorithmus vorgestellt, der eine relationale Datenbank in eine Graphdatenbank transformiert. Dieser Algorithmus besteht aus zwei Teilen: Zuerst wird ein Schema in Form eines Transformations-Graphen erstellt. Im zweiten Teil werden die Daten gemäß des Schemas zu einem Graphen transformiert.

Vielen relationalen Datenbanken liegt ein ER-Schema zugrunde. Da ein ER-Schema selbst einen Graphen darstellt, bietet es einen guten Anhaltspunkt, wie eine Graphrepräsentation einer relationalen Datenbank aussehen könnte. Da ein umfassendes ER-Re-Engineering allerdings zu aufwendig wäre, wird eine Kompromisslösung angestrebt. Relationen, in denen genau zwei Fremdschlüssel den ganzen Primärschlüssel bilden, repräsentieren oft zweistellige Relationships. Diese Relationen sollen im Transformations-Graphen als einfache Kanten dargestellt werden. Für alle anderen Relationen wird nicht versucht die ursprünglichen ER-Elemente nachzubilden. Hier wird sich darauf beschränkt, für einzelne Fremdschlüsselbeziehungen jeweils eine Kante zu erzeugen.

3.2 Schemaerstellung

In diesem Schritt soll ein Schema erstellt werden, das eine Abbildung von Relationen auf die Elemente eines Graphen beschreibt. Dabei sollen Fremdschlüsselbeziehungen durch Kanten ausgedrückt werden. Sei R die Menge aller Relationen. Aus dieser Menge R sollen nun drei neue Mengen N (nodes), E (edges) und NME (node-multi-edges) gebildet werden, für die gilt:

$$N \dot{\cup} E \dot{\cup} NME = R \tag{3.1}$$

Relationen, deren Tupel...

1. auf einen Knoten abgebildet werden, werden zur Menge N hinzugefügt.
2. auf eine Kante abgebildet werden, werden zur Menge E hinzugefügt.
3. auf einen Knoten und beliebig viele Kanten abgebildet werden, werden zur Menge NME hinzugefügt.

Dazu werden die Relationen anhand ihrer Primär- und Fremdschlüssel klassifiziert:

Relationen, die keinen Fremdschlüssel enthalten, werden auf Knoten abgebildet. Sie werden zur Menge N hinzugefügt.

Relationen, bei denen genau zwei Fremdschlüssel den ganzen Primärschlüssel bilden, werden auf Kanten abgebildet. Sie werden zur Menge E hinzugefügt.

Relationen, die genau zwei Fremdschlüssel besitzen, diese jedoch nicht den Primärschlüssel bilden, können auf zwei Arten abgebildet werden. Entweder kann für jedes Tupel einer solchen Relation eine Kante erstellt werden, oder ein Knoten und zwei Kanten. Sollten andere Relationen mit einem Fremdschlüssel auf den Primärschlüssel verweisen, müssen jeweils ein Knoten und zwei Kanten erstellt werden. Sonst würden für die Kanten, die durch die Fremdschlüssel der anderen Relationen erzeugt werden, keine Zielknoten existieren. Verweisen keine Fremdschlüssel anderer Relationen auf den Primärschlüssel, kann der Benutzer frei bestimmen, auf welche der beiden Arten die Relation abgebildet wird. Diese Relationen werden also entweder zur Menge NME oder zur Menge E hinzugefügt.

Alle anderen Relationen werden zur Menge NME hinzugefügt. Dabei kann es sich entweder um Relationen handeln, die genau einen Fremdschlüssel besitzen oder die drei oder mehr Fremdschlüssel besitzen. Für diese Tupel werden jeweils ein Knoten und genau so viele Kanten erzeugt, wie die entsprechende Relation Fremdschlüssel besitzt.

Insgesamt lässt sich der Schritt der Klassifizierung als Algorithmus 1 zusammen fassen.

Der Transformations-Graph lässt sich aus dieser Aufteilung erzeugen. Dazu wird für jede Relation $n \in (N \cup NME)$ ein Knoten erzeugt. Das Label eines Knotens entspricht dabei dem Namen von n. Für jede Relation $e \in E$ wird eine Kante erzeugt. Diese geht vom Knoten, der dem Ziel des ersten Fremdschlüssels von e entspricht, zum Knoten, der dem Ziel des zweiten Fremdschlüssels von e entspricht. Das Label der Kante wird vom Namen von e übernommen. Für jede Relation $m \in NME$ wird für jeden Fremdschlüssel in m eine Kante erzeugt. Diese führt vom Knoten, der m entspricht, zum Knoten, der dem Ziel des Fremdschlüssel entspricht. Das Label der Kante wird vom Namen des Fremdschlüssels übernommen. Auf diese Weise wird ein Graph erzeugt, der jede Art von Knoten und jede Art von Kanten genau einmal enthält.

Algorithmus 1 classifyRelations

$N = E = NME = \emptyset$
for all $r \in R$ **do**
  if r hat keine Fremdschlüssel **then**
    $N \leftarrow N \cup \{r\}$
  else if r hat genau zwei Fremdschlüssel als Primärschlüssel **then**
    $E \leftarrow E \cup \{r\}$
  else if r hat genau zwei Fremdschlüssel, sonst **then**
    if r ist Ziel von Fremdschlüsseln **then**
      $NME \leftarrow NME \cup \{r\}$
    else if Benutzerentscheidung **then**
      $NME \leftarrow NME \cup \{r\}$
    else
      $E \leftarrow E \cup \{r\}$
    end if
  else
    $NME \leftarrow NME \cup \{r\}$
  end if
end for

3.3 Graph-Erzeugung

Nun soll aus den Daten in den Relationen ein Graph erzeugt werden, der dem in 3.1 erstellten Schema entspricht. Dazu werden zunächst alle Knoten erzeugt. Erst danach werden alle Kanten ergänzt, damit sichergestellt ist, dass Ziel- und Ursprungsknoten existieren.
Zur Erstellung der Knoten wird Algorithmus 2 verwendet: Mit diesem Algorithmus wird

Algorithmus 2 createNodes

for all $r \in N$ **do**
  for all $t \in r$ **do**
    addNode(r.getMetaData(), t)
  end for
end for
for all $r \in NME$ **do**
  for all $t \in r$ **do**
    addNode(r.getMetaData(), t)
  end for
end for

für jedes Tupel jeder Relation aus N und NME ein Knoten erzeugt. Alle im Algorithmus verwendeten Funktionen, insbesondere die Funktion addNode(RelationsMetaDaten, Tu-

pel), werden so nicht von Neo4j angeboten und müssen selbst implementiert werden. (Siehe dazu auch Kapitel 4.)

Die Funktion getMetaData() gibt die für uns relevanten Informationen über eine Relation zurück. Diese beinhalten: Name der Relation, Namen der Spalten, Primärschlüssel und Fremdschlüssel.

Die Funktion addNode(RelationsMetaDaten, Tupel) erzeugt einen Knoten gemäß des Propertygraph-Modells von Neo4j auf folgende Weise: Sei $t = (v_0, v_1, ..., v_m)$ das übergebene Tupel, r der Name der Relation und $c_0, c_1, ..., c_m$ die zugehörigen Spaltennamen. Das Label des erzeugten Knotens entspricht r. Die Namen der Properties werden von $c_0, ..., c_m$ übernommen. Die Werte der Properties entsprechen dann den jeweils passenden Werten aus t. An dieser Stelle spielen Primär- und Fremdschlüssel noch keine Rolle. Diese Information werden später genutzt, um die Kanten richtig im Graphen zu ergänzen.

Dieser Knoten könnte durch folgende Cypher-Query erzeugt werden:

`CREATE (n:` r`)`
`SET n.`c_0`=`v_0`, n.`c_1`=`v_1`, ..., `c_m`=`v_m

Im letzten Schritt müssen die Kanten auf geeignete Weise eingefügt werden. Dazu wird Algorithmus 3 benutzt.

Algorithmus 3 createEdges

```
for all r ∈ E do
    for all t ∈ r do
        addPropertyEdge(r.getMetaData(), t)
    end for
end for
for all r ∈ NME do
    for all t ∈ r do
        addMultiEdge(r.getMetaData(), t)
    end for
end for
```

Analog zu Algorithmus 2 werden hier für jedes Tupel jeder Relation aus E und NME entsprechend viele Kanten erzeugt.

Auch die in diesem Algorithmus verwendeten Funktionen werden nicht von Neo4j angeboten und müssen selbst implementiert werden. Die Funktion getMetaData() ist die gleiche wie in Algorithmus 2.

Die Funktion addPropertyEdge(RelationsMetaDaten, Tupel) fügt eine Kante in den Graphen ein. Seien...

$t = (f_0 \to (r_0), f_1 \to (r_1), v_2, v_3, ..., v_m)$	das übergebene Tupel,
r	der Name der Relation,
$c_0, c_1, ..., c_m$	die Spaltennamen der Relation,
pk_0	der Spaltenname von f_0 in r_0,
pk_1	der Spaltenname von f_1 in r_1,
n_0	der Knoten, der f_0 enthält,
n_1	der Knoten, der f_1 enthält.

Da die Relationen aus der Menge E stammen, ist sichergestellt, dass genau zwei Fremdschlüssel existieren. Die Kante wird nun von n_0 nach n_1 hinzugefügt. Dabei entspricht das Label der Kante r und die Namen der Properties c_0, ..., c_m. Die Werte der Properties werden passend aus $f_0, f_1, v_2, ...v_m$ übernommen.

Diese Kante könnte durch folgende Cypher-Query erzeugt werden:

```
MATCH (n0: r_0), (n1: r_1)
WHERE n0.pk_0 = f_0 AND n1.pk_1 = f_1
CREATE n0-[r: r]->n1
SET r.c_0=f_0, r.c_1=f_1, r.c_2=v_2, ..., r.c_m=v_m
```

Die Funktion addMultiEdge(RelationsMetaDaten, Tupel) erzeugt eine variable Anzahl von Kanten. Seien...

$t = (f_0 \to (r_0), f_1 \to (r_1), ..., f_i \to (r_i), v_{i+1}, v_{i+2}, ..., v_m)$	das übergebene Tupel,
i	die Anzahl der Fremdschlüssel in t,
r	der Name der Relation,
$c_0, c_1, ..., c_m$	die Spaltennamen der Relation,
pk_j	der Spaltenname von f_j in R_j (j=1,..., i),
pk	der Primärschlüssel der Relation r,
pkv	der Wert des Primärschlüssel in t,
n_{pk}	der Knoten, der pkv enthält,
n_j	der Knoten, der f_j enthält.

Diese Funktion wird unter anderem für Relationen aufgerufen, die einen, drei oder mehr Fremdschlüssel besitzen. Für alle $j \in \mathbb{N}$ mit $j \leq i$ wird nun eine Kante von n_{pk} nach n_j hinzugefügt. Das Label der Kante entspricht dabei c_j. Hier existiert bereits ein Knoten für jedes Tupel, der dessen Einträge als Properties enthält (vgl. Algorithmus 1). Die Tupeleinträge müssen also nicht mehr als Properties in den Kanten gespeichert werden.

Die Kanten können durch folgende Cypher-Query erzeugt werden:

for $j = 0; j < i; j \leftarrow j + 1$ **do**
  `MATCH (npk:` r`), (nj:` r_j`)`
  `WHERE npk.`pk `=` pkv `AND nj.`pk_j `=` f_j
  `CREATE npk-[r:` c_j`]->nj`
end for

Um eine relationale Datenbank vollständig zu transformieren, müssen nur noch alle Teilalgorithmen nacheinander ausgeführt werden, dies geschieht in Algorithmus 4.

Algorithmus 4 Transform

classifyRelations()
createNodes()
createEdges()

3.4 Beispiel

Gegeben sei folgendes relationale Schema:

SONG	(SID, Länge, Titel)
SONGGENRE	(Song→ SONG, Genre)
ALBUM	(AID, Titel)
MUSIKER	(MID, Name)
LAND	(LID, Name)
LEBT_IN	(Einreisedatum, Musiker→MUSIKER, Land→LAND)
ERSCHEINT_IN	(Album→ALBUM, Land→LAND, Datum)
SPIELT	(Musiker→MUSIKER, Song→SONG, Album→Album, Instrument)

Abbildung 3.1 zeigt eine Beispieldatenbank, die dem Schema entspricht.

SONG

SID	Länge	Titel
17	7:10	Layla
54	5:02	Bell Bottom Blues

SONGGENRE

SID	Genre
17	Rock
17	Bluesrock
54	Blues

ALBUM

AID	Titel
31	Layla and Other...

MUSIKER

MID	Name
43	Eric Clapton
16	Bobby Whitlock

LEBT_IN

Einreisedatum	Musiker	Land
30.03.1945	43	5
18.03.1948	16	7

ERSCHEINT_IN

Album	Land	Datum
31	5	01.11.1970
31	7	01.11.1970

SPIELT

Musiker	Song	Album	Instrument
43	17	31	Gitarre
16	17	31	Piano
43	54	31	Gesang
16	54	31	Orgel

LAND

LID	Name
5	Großbritannien
7	U.S.A.

Abbildung 3.1: Beispieldatenbank mit Einträgen

Klassifiziert man die Relationen, erhält man folgende Aufteilung:

N = {SONG, ALBUM, MUSIKER, LAND}
E = {ERSCHEINT_IN}
NME = {SONGGENRE, LEBT_IN, SPIELT}

Abbildung 3.2 zeigt den entsprechenden Transformations-Graphen.

Nun wird der vorgestellte Algorithmus benutzt, um diese Datenbank zu transformieren.

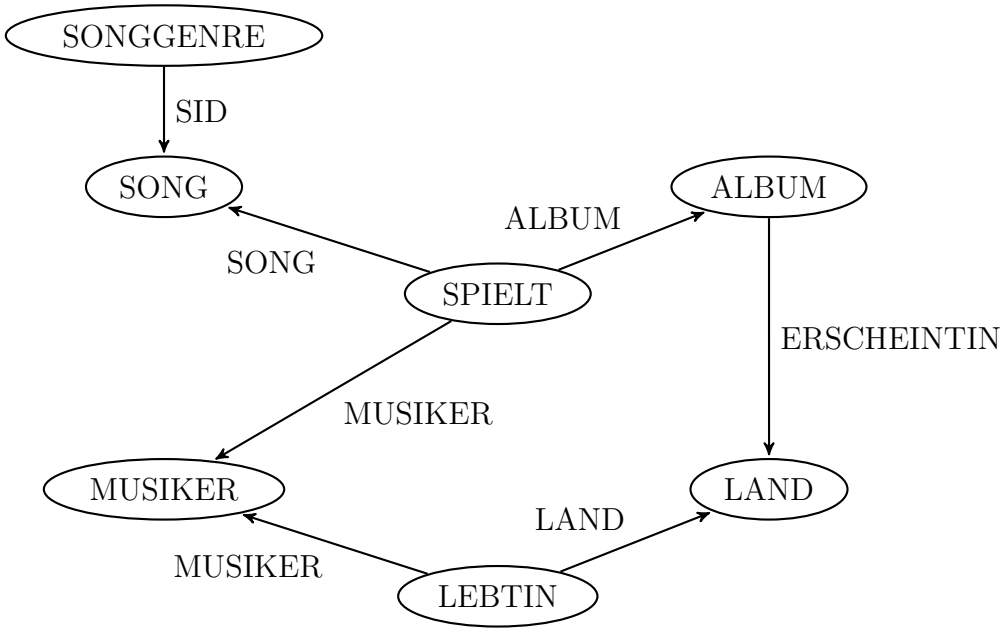

Abbildung 3.2: Transformations-Graph

Daraus entsteht der Propertygraph, der in Abbildung 3.3 zu sehen ist. Die eingefärbten Kanten tragen dabei folgende Attribute:

ERSCHEINT_IN: Album: 31, Land: 5, Datum: 01.11.1970

ERSCHEINT_IN: Album: 31, Land: 7, Datum 01.11.1970

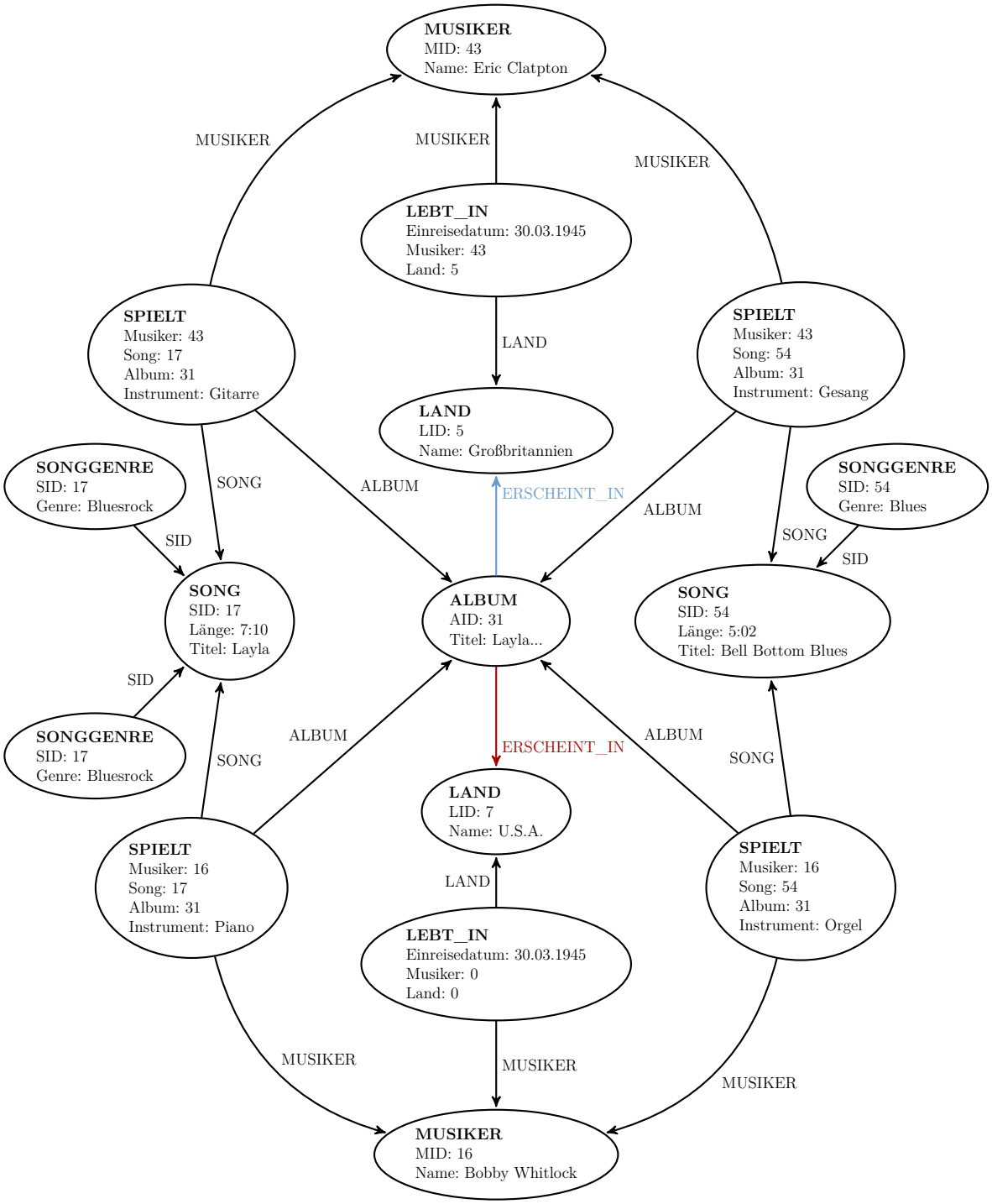

Abbildung 3.3: Graph-Repräsentation der Beispieldatenbank

Kapitel 4

Implementierung der Transformation

Im Rahmen dieser Arbeit wurde die in Kapitel 3 beschriebene Transformation implementiert. In diesem Kapitel soll die Implementierung vorgestellt werden. Dazu wird zunächst der Entwurf einer solchen Anwendung erläutert. Danach werden wichtige Mechanismen dieser Anwendung genauer beschrieben. Zuletzt wird die Bedienung der Anwendung anhand einer Beispieltransformation erklärt.

4.1 Entwurf

In diesem Abschnitt soll eine Anwendung entworfen werden, die es dem Benutzer erlaubt, eine Oracle-SQL-Datenbank in eine Neo4j-Graphdatenbank zu transformieren. Der Transformationsprozess gliedert sich dabei in vier Schritte.

1. Der Benutzer gibt die Verbindungsdaten ein.
2. Eine initiale Einteilung der Relation wird erstellt.
3. Der Benutzer ordnet noch nicht eingeteilte Relationen in E- oder NME-Relationen ein.
4. Die Daten werden transformiert.

Da die Entscheidung zur Einteilung der Relationen eine Interaktion des Benutzers mit der Anwendung erfordert, wird die Anwendung unter Benutzung des Model-View-Controller-Patterns entworfen.

Abbildung 4.1 stellt eine Übersicht über die in der Anwendung verwendeten Packages dar. Das Hauptaugenmerk soll hierbei auf dem Model-Package liegen, da in diesem die eigentliche Transformation implementiert wird. Das Model-Package gliedert sich in drei Unterpackages: ReadData, ConvertData und WriteData. Mit Hilfe des ReadData-Package werden Daten aus der relationalen Datenbank ausgelesen. Das ConvertData-Package wandelt die ausgelesenen Daten in einen Propertygraphen um. Mit dem WriteData-Package wird dieser Graph in der Graph-Datenbank gespeichert.

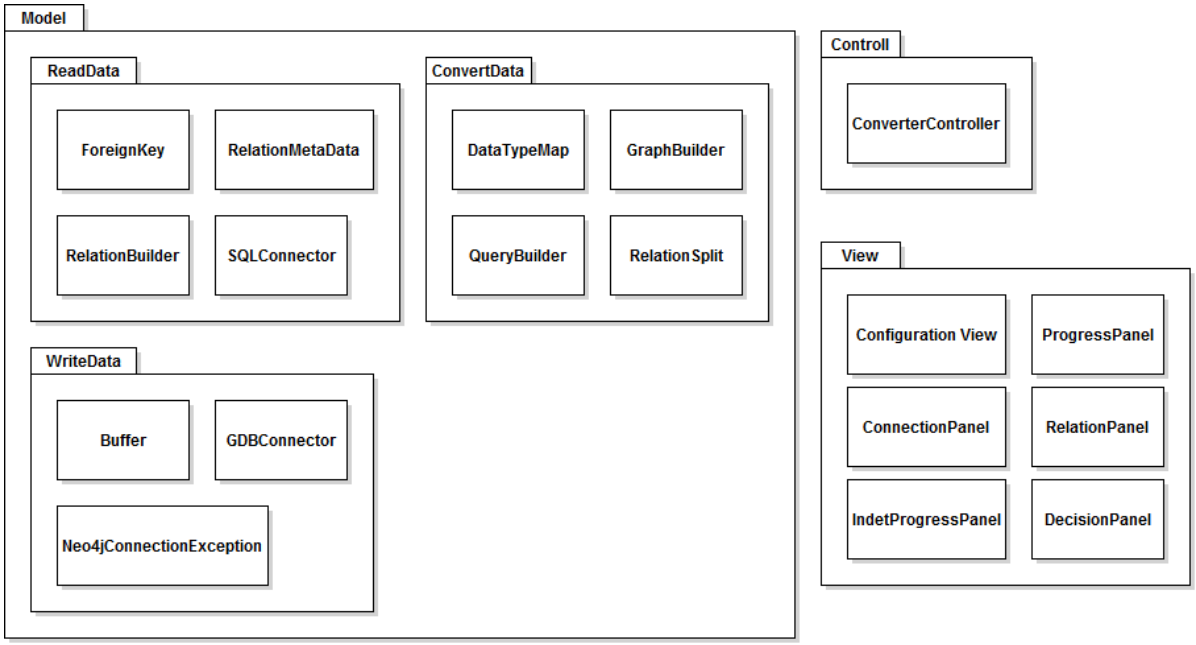

Abbildung 4.1: Package-Diagramm der Anwendung

Abbildung 4.2 zeigt eine Übersicht über die Klassen des Model.ReadData-Packages. Dieses Package soll Daten aus der SQL-Datenbank auslesen. Dies schließt sowohl die in Kapitel 3.3 beschriebenen Meta-Daten als auch die eigentlichen Datensätze aus der Datenbank ein. Die Klasse „RelationMetaData“ entspricht dabei den Meta-Daten, die die Funktion „getMetaData“ zurückgibt. Die Klasse „ForeignKey“ repräsentiert einen Fremdschlüssel. Dabei wird der Index des Fremdschlüssels in der eigenen Relation, die Zielrelation und die Zielspalte gespeichert. Diese Klasse wird von den RelationMetaData-Objekten benutzt, um Fremdschlüssel zu speichern. Die Klasse „SQLConnector“ stellt die Verbindung zur SQL-Datenbank her. Sie führt außerdem alle Queries aus, die an die SQL-Datenbank gerichtet sind. Die Klasse „RelationBuilder“ ruft den SQL-Connector auf und erzeugt die RelationMetaData-Objekte mit Hilfe der durch den SQL-Connector zurückgegebenen Daten.

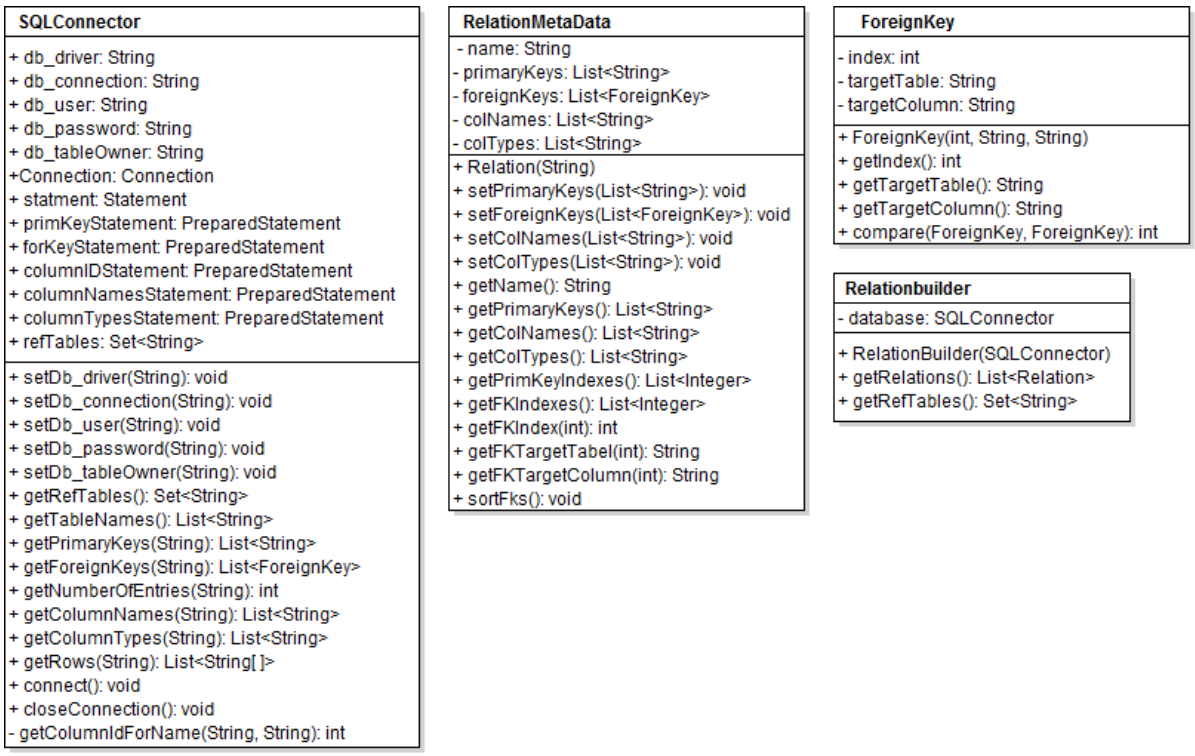

Abbildung 4.2: Klassendiagramm des Model.ReadData-Packages

Abbildung 4.3 zeigt die Klassen, die im Model.ConvertData-Package enthalten sind. Dieses Package soll die Daten, die mit Hilfe des Model.ReadData-Package ausgelesen wurden, in einen Graphen transformieren. Die Klasse „RelationSplit“ beinhaltet die Aufteilung der Relationen auf die Mengen N, E und NME. Die Partitionierung wird dabei durch die RelationSplit vorgenommen. Dafür wird eine Liste mit allen Relationen vom RelationBuilder abgerufen. Der „GraphBuilder“ koordiniert die Übersetzung der Tupel aus der SQL-Datenbank zu Queries, die Kanten und Knoten in der Graph-Datenbank erzeugen. Dazu ruft er gemäß der RelationSplit passende Methoden des „QueryBuilder“ auf. Dieser erzeugt für jedes Tupel, das ihm übergeben wird, jeweils eine Cypher-Query. Der GraphBuilder implementiert auch den getMetaData()-Mechanismus aus Kapitel 3.3, indem er die setCurrentRelation Methode des QueryBuilders aufruft und ihm so RelationMetaData-Objekte übergibt. Die „DataTypeMap“ ist eine statische Klasse. In ihr wird gespeichert, welcher SQL-Datentyp auf welchen Neo4j-Datentyp abgebildet wird. Der QueryBuilder ruft diese Informationen von der DataTypeMap auf, um die Queries entsprechend zu gestalten.

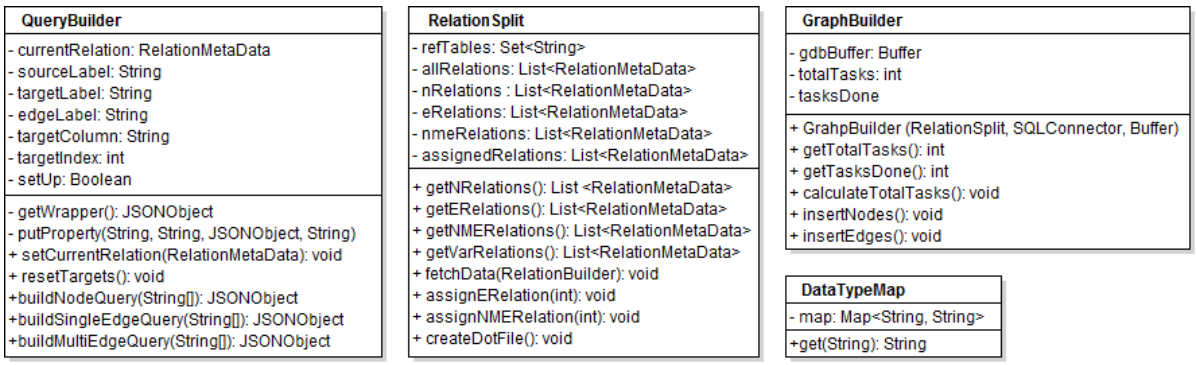

Abbildung 4.3: Klassendiagramm des Model.ConvertData-Packages

Abbildung 4.4 zeigt eine Übersicht über die Klassen des Model.WriteData-Packages. Dieses Package ist dafür verantwortlich, die umgewandelten Daten in der Neo4j-Datenbank zu speichern. Der „GDBConnector“ stellt die Verbindung zu Neo4j her und führt Cypher-Queries aus. Im „Buffer“ werden die vom QueryBuilder erstellten Queries gebündelt. Ist ein Schwellenwert erreicht, werden alle Queries über den GDBConnector gesendet. Tritt ein Fehler beim Verbindungsaufbau zu Neo4j auf, wirft der GDBConnector eine „Neo4jConnectionException“.

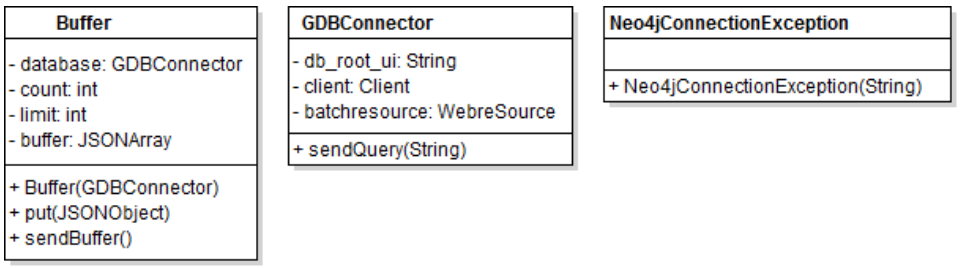

Abbildung 4.4: Klassendiagramm des Model.WriteData-Packages

4.2 Wichtige Mechanismen

In diesem Abschnitt werden wichtige Mechanismen vorgestellt. Dies umfasst die Generierung der Relationen-Metadaten, die Verwendung eines Buffers und die Datentypkonvertierung.

4.2.1 Generierung der Relationen-Metadaten

Die Metadaten einer Relation spielen bei der Erzeugung des Graphen eine entscheidende Rolle. Deswegen soll in diesem Abschnitt erläutert werden, wie sie erzeugt werden.

Die Relationsmetadaten enthalten

- den Namen der Relation,
- die Namen der Spalten,
- die Datentypen der Spalten,
- den Primärschlüssel,
- die Fremdschlüssel.

Die Fremdschlüssel bestehen dabei aus einer Spaltennummer, der Zieltabelle und der Zielspalte (vgl. Abb. 4.2).

Die RelationMetaData-Objekte werden vom RelationBuilder erzeugt. Der RelationBuilder liest hierfür Daten aus dem Data Dictionary der SQL-Datenbank aus (vgl. [Ora14]). Die dafür benötigten Queries stellt der SQLConnector bereit.

Zuerst werden die Namen aller relevanten Relationen ausgelesen. Wir gehen dabei davon aus, dass diese genau einem Benutzer gehören. Sei o dieser Benutzer. Dann können die Namen der Relationen mit folgender Query bestimmt werden:

```
SELECT
    TABLE_NAME
FROM
    ALL_TABLES
WHERE
    OWNER='o'
```

Für jede Relation werden jetzt nacheinander die benötigten Informationen ausgelesen. Dafür werden hauptsächlich die Tabellen ALL_TAB_COLUMNS, ALL_CONS_COLUMNS und ALL_CONSTRAINTS aus dem Data Dictionary benutzt. ALL_TAB_COLUMNS enthält Informationen über alle Spalten aller Tabellen. ALL_CONS_COLUMNS liefert Informationen über alle Spalten, für die mindestens ein Constraint definiert ist. ALL_CONSTRAINTS enthält Informationen über alle Constraints, insbesondere über Primärschlüssel- (CONSTRAINT_TYPE='P') und Fremdschlüsselconstraints (CONSTRAINT_TYPE='R').

Sei r der Name der aktuellen Relation. Die Namen der Spalten können mit folgender Query ermittelt werden:

```
SELECT
    COLUMN_NAME
FROM
    ALL_TAB_COLUMNS
WHERE
    OWNER='o'
    AND TABLE_NAME='r'
```

Die Datentypen der Spalten werden mit Hilfe dieser Query bestimmt:

```
SELECT
    DATA_TYPE,
    DATA_PRECISION,
    DATA_SCALE
FROM
    ALL_TAB_COLUMNS
WHERE
    OWNER='o '
    AND TABLE_NAME='r'
```

Zusätzlich zum Namen des Datentyps DATA_TYPE werden noch DATA_PRECISION und DATA_SCALE abgerufen. Diese werden benötigt, um beim Datentyp „NUMBER“ zwischen ganzzahligen Werten und Fließkommazahlen zu unterscheiden. Diese werden entsprechend auf die Java-Datentypen int oder double abgebildet.

Der Primärschlüssel einer Relation wird mit dieser Query bestimmt:

```
SELECT
    A.COLUMN_NAME
FROM
    ALL_CONS_COLUMNS A
    JOIN ALL_CONSTRAINTS C
        ON (A.CONSTRAINT_NAME = C.CONSTRAINT_NAME
            AND A.OWNER = C.OWNER)
WHERE
    A.OWNER='o'
    AND A.TABLE_NAME='r'
    AND C.CONSTRAINT_TYPE='P'
```

Mit der letzen Query werden die Fremdschlüssel bestimmt:

```
SELECT
    A.COLUMN_NAME FKEY,
    D.TABLE_NAME TARGETTABLE,
    D.COLUMN_NAME TARGETCOLUMN
```

```
FROM
    ALL_CONS_COLUMNS A
    JOIN ALL_CONSTRAINTS B
        ON (A.CONSTRAINT_NAME = B.CONSTRAINT_NAME
            AND A.OWNER = B.OWNER)
    JOIN ALL_CONSTRAINTS C
        ON (B.R_CONSTRAINT_NAME = C.CONSTRAINT_NAME
            AND B.OWNER = C.OWNER)
    JOIN ALL_CONS_COLUMNS D
        ON (C.CONSTRAINT_NAME = D.CONSTRAINT_NAME
            AND C.OWNER = D.OWNER)
WHERE
    A.OWNER='o'
    AND A.TABLE_NAME='r'
    AND B.CONSTRAINT_TYPE='R'
```

In dieser Query werden zunächst alle Spalten einer Tabelle bestimmt, die einen Fremdschlüssel enthalten. In der Query sind das die Tabellen A und B. Dann wird das Ziel der Fremdschlüssel ermittelt, indem nach Constraints gesucht wird, die von den Fremdschlüsseln referenziert werden. Dies leistet Tabelle C. Zum Schluss werden die referenzierten Constraints noch Tabellen und Spalten zugeordnet. Im Query erfolgt dies über den Join der Tabelle D.

Mit Hilfe der so bestimmten Informationen erzeugt der RelationBuilder schließlich die RelationMetaData-Objekte.

4.2.2 Verwendung eines Buffers

Die hier entworfene Anwendung soll mit einer dedizierten Neo4j-Datenbank arbeiten, die Datenbank soll also als eigenständige Anwendung laufen. Bei der Erstellung des Graphen wird für jeden Knoten und jede Kante des Graphen genau eine Query erzeugt. Jede dieser Queries muss an Neo4j übertragen werden.

Die Übertragung soll über die REST-API erfolgen, die Neo4j anbietet. Diese erlaubt es, Cypher-Queries als in HTTPS gekapselte JSON-Objekte zu übertragen. [Neo14] schlägt als Ziel für die HTTPS-Anfragen die Adresse /db/data/cypher vor. Unter dieser Adresse erreicht man einen Cyphercompiler, der einzelne Queries auf der Datenbank ausführen kann.

Allerdings erweist es sich als problematisch, jede Query einzeln auszuführen. Da für jedes Tupel einer N-Relation eine Query, für jedes Tupel einer E-Relation eine Query und für jedes Tupel einer NME-Relation mindestens zwei Queries erzeugt werden, ist mit einer großen Gesamtanzahl an Queries zu rechnen. Laut [HP13] existiert beim Ausführen einer

Query über die REST-API ein konstanter Netzwerk-Overhead. Jede Query einzeln zu übermitteln hieße, für jede Query diesen Overhead in Kauf zu nehmen. Des Weiteren führt die Übermittlung einer großen Anzahl an HTTPS-Anfragen in kurzer Zeit dazu, dass Neo4j abstürzt, ähnlich wie bei einer Denial-of-Service Attacke.

Daher müssen die Queries in irgendeiner Form gebündelt werden. Unter /db/data/batch stellt Neo4j eine Stapelverabeitungsschnittstelle bereit, die auch Cypher-Queries ausführen kann. Auch diese Schnittstelle akzeptiert in HTTPS gekapselte JSON-Objekte. Um die Queries zusammenzufassen wird ein JSON-Array erstellt, dem die Queries als JSON-Objekt hinzugefügt werden. Diese Funktionalität erledigt der Buffer. Er sammelt die Queries, bis ein gewisser Schwellwert erreicht ist. Danach schickt der Buffer die Queries als JSON-Array über eine HTTPS-Anfrage ab. Als geeigneter Schwellwert hat sich in dieser Implementierung eine Anzahl von 500 Queries bewährt.

4.2.3 Datentypkonvertierung

Während der gesamten Konvertierung der Datenbank durchlaufen die Daten drei Systeme: Die SQL-Datenbank, den Konverter und die Neo4j-Datenbank. Daher ist es notwendig sicherzustellen, dass die Datentypen sinnvoll aufeinander abgebildet werden. Neo4j ist in Java implementiert und benutzt daher (eingeschränkte) Java Datentypen. Der Konverter wurde ebenfalls in Java geschrieben, so dass nur eine Datenkonversion von Oracle-SQL auf Java erfolgen muss.

Die von Neo4j unterstützen Datentypen sind: boolean, byte (8-bit integer), short (16-bit integer), int (32-bit integer), long (64-bit integer), float, double, char und String. Die Daten aus der SQL-Datenbank werden mittels JDBC abgerufen. Dazu muss je nach Datentyp eine passende Methode aufgerufen werden, das heißt der Datentyp muss explizit angegeben werden. Alle von Neo4j benutzen Datentypen haben eine vollständige Stringrepräsentation. Deswegen werden im Konverter alle Daten zunächst als String zwischengespeichert. Beim Erzeugen der Queries findet dann die Konversion von String zum endgültigen Datentyp statt. In der DataTypeMap ist gespeichert, welcher SQL-Datentyp auf welchen Neo4j-Datentypen abgebildet wird. Tabelle 4.1 zeigt die Zuordnung der häufigsten Datentypen.

SQL-Datentyp	Neo4j-Datentyp
VARCHAR2	String
NUMBER(x, 0)	Integer
NUMBER(y, z)	Double
DATE	String

Mit $x, z \in \mathbb{N}, y \in \mathbb{N}_0$ und $y \geq z$

Tabelle 4.1: Datentypzuordnung

In den RelationMetaData-Objekten ist auch der Datentyp jeder Spalte gespeichert. Im QueryBuilder wird beim Übersetzen einer Spalte zu einer Property mit Hilfe dieser Information der richtige Zieldatentyp ermittelt und die Query dementsprechend formuliert.

4.3 Bedienung

In diesem Abschnitt wird eine Beispiel-SQL-Datenbank zu einer Neo4j-Datenbank mit Hilfe des Konverters transformiert. Als Beispieldatenbank dient dabei die Modulkatalog-Datenbank des Fachgebietes Datenbanken und Informationssysteme des Instituts für praktische Informatik der Leibniz Universität Hannover.

Das Schema der SQL-Datenbank ist in Abbildung 4.5 zu sehen.

Abbildung 4.6 zeigt den ersten Schritt des Transformationsprozesses. In diesem gibt der Benutzer die Verbindungsdaten der beiden Datenbanken ein.

In Abbildung 4.7 ist der zweite Schritt des Konverterdialogs zu sehen. In diesem Schritt wird die Aufteilung der Relationen erzeugt. Die ersten drei Textfelder zeigen die Relationen, die fest N-, E- oder NME-Relationen zugewiesen wurden. Darunter können (sofern vorhanden) variable Relationen entweder E- oder NME-Relationen zugeordnet werden. Zusätzlich besteht die Möglichkeit den Transformationsgraphen als .dot GraphViz-Datei zu exportieren (Für weitere Informationen siehe [Gra]).

ETHISTORY	(SEMESTER→ SEMESTER, ...)
HISTORY	(SEMESTER→ SEMESTER, ...)
KB	(ID, SG→ STUDIENGANG, GUELTIG_VON → SEMESTER, GUELTIG_BIS → SEMESTER, ...)
KB_MODUL	(KB→ KB, Modul→ MODUL, GUELTIG_VON → SEMESTER, GUELTIG_BIS → SEMESTER, ...)
LV	(ID, GUELTIG_VON → SEMESTER, GUELTIG_BIS → SEMESTER, ...)
LV_PE	(LV→ LV, PE→ PE, GUELTIG_VON → SEMESTER, GUELTIG_BIS → SEMESTER, ...)
MODUL	(ID, GUELTIG_VON → SEMESTER, GUELTIG_BIS → SEMESTER, ...)
NEU_BETREUER	(NEU_LV→ LV, BETREUER→ PERSON)
NEU_DOZENT	(NEU_LV→ LV, DOZENT→ PERSON)
NEU_LV	(ID, APARTNER→ PERSON, KB1_INF_MSC→ KB KB1_TI_MSC→ KB, KB2_INF_MSC→ KB, KB2_TI_MSC→ KB, MODUL_INF_ALT→ MODUL, MODUL_INF_BSC→ MODUL, MODUL_INF_MSC→ MODUL, MODUL_TI_BSC→ MODUL, MODUL_TI_MSC→ MODUL, SEMESTER→ SEMESTER)
NEU_PRUEFER	(NEU_LV→ LV, PRUEFER→ PERSON, ..)
PE	(ID, GUELTIG_VON → SEMESTER, GUELTIG_BIS → SEMESTER, ...)
PERSON	(ID, GUELTIG_VON → SEMESTER, GUELTIG_BIS → SEMESTER, ...)
PRUEFUNG	(MODUL→ MODUL, PE→ PE, GUELTIG_VON → SEMESTER, GUELTIG_BIS → SEMESTER, ...)
SANGEBOT	(ID, GUELTIG_VON → SEMESTER, GUELTIG_BIS → SEMESTER, ...)
SEMESTER	(ID, ...)
STUDIENGANG	(ID, GUELTIG_VON → SEMESTER, GUELTIG_BIS → SEMESTER, ...)
TABLE_APARTNER_LV	(LV→ LV, APARTNER→ PERSON)
TABLE_APARTNER_MODUL	(MODUL→ MODUL, APARTNER→ PERSON)
TABLE_BETREUER	(SANGEBOT→ SANGEBOT, BETREUER→ PERSON)
TABLE_DOZENT	(SANGEBOT→ SANGEBOT, DOZENT→ PERSON)
TABLE_PRUEFER	(SANGEBOT→ SANGEBOT, PRUEFER→ PERSON)

Abbildung 4.5: Relationales Schema der Modulkatalog-Datenbank

Abbildung 4.6: Schritt Eingabe der Verbindungsdaten

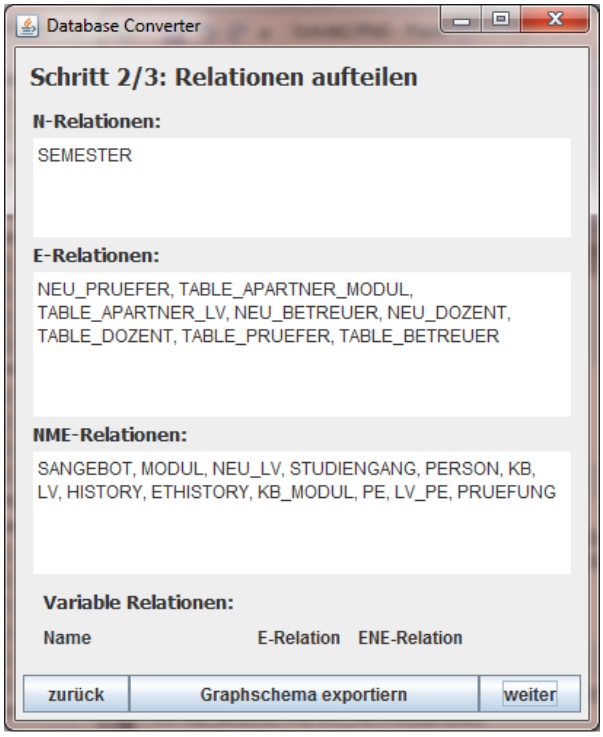

Abbildung 4.7: Aufteilung der Relationen

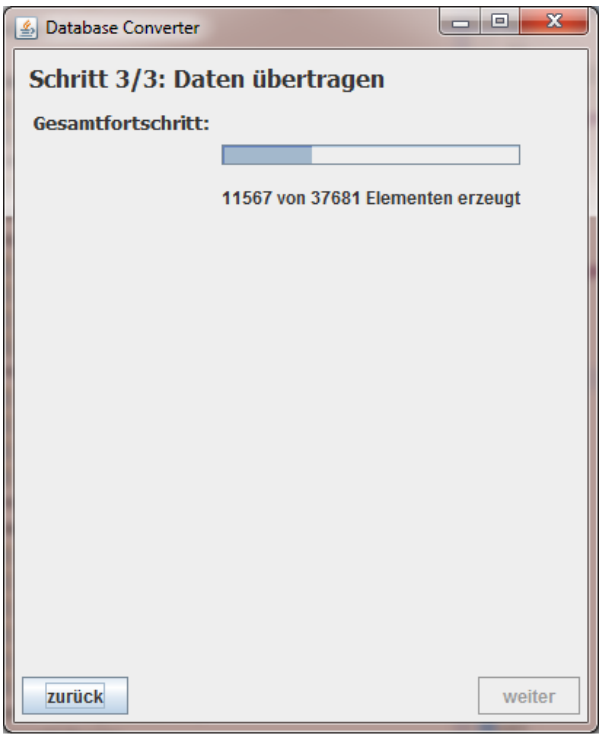

Abbildung 4.8: Erstellen des Graphen

Abbildung 4.8 zeigt den letzten Schritt. In diesem findet die eigentliche Transformation statt. Dabei werden die Daten aus der SQL-Datenbank ausgelesen und die Graphelemente in der Neo4j-Datenbank erzeugt.

Der exportierte Transformations-Graph der Modulkatalog-Datenbank ist in Abbildung 4.9 zu sehen. Die semantisch wichtigste Kette soll hier kurz erläutert werden: Zu einer Lehrveranstaltung (LV) können verschiedene Semesterangebote (SANGEBOT) bestehen. Über LV_PE werden den Lehrveranstaltungen prüfbare Einheiten (PE) zugewiesen. Prüfungen (PRUEFUNG) erfolgen für eine prüfbare Leistung und innerhalb eines Moduls. Ein Modul wird über KB_Modul in einen Kompetenzbereich (KB) eingeordnet.

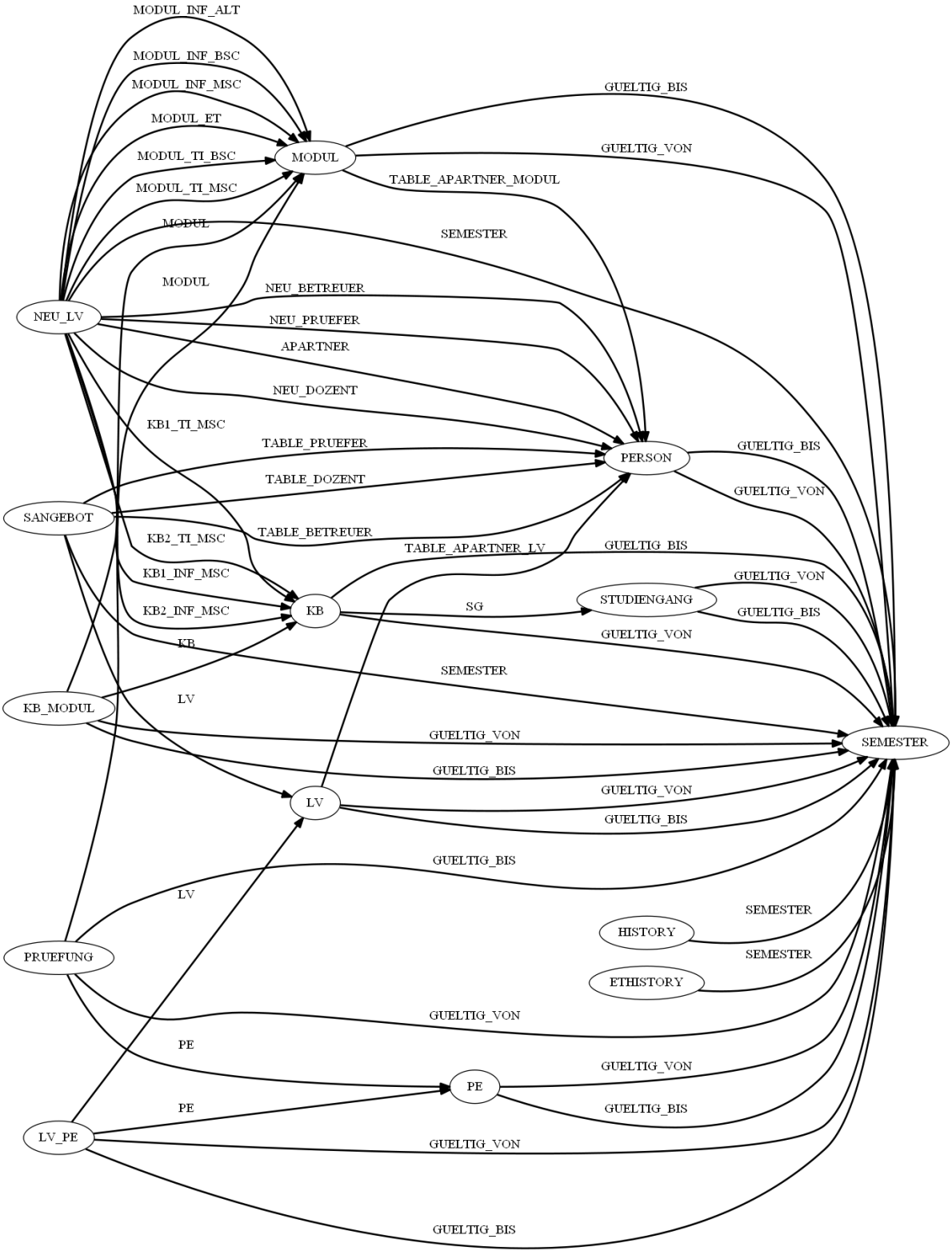

Abbildung 4.9: Transformationsgraph der Modulkatalog-Datenbank

Kapitel 5

Evaluation

In diesem Kapitel soll die Funktionalität und der Funktionsumfang von Neo4j evaluiert werden. Dazu wird zunächst die Ausdrucksfähigkeit der Anfragesprache Cypher beschrieben. Weiterhin werden die Kosten von typischen Graph-Operationen erläutert. Darauf wird der Umfang der Funktionalität die Neo4j bietet betrachtet. Im letzten Abschnitt werden die Ausführungszeiten konkreter einzelner Queries gemessen. Dazu werden Anfragen an unterschiedliche Neo4j-Datenbanken gestellt.

5.1 Ausdrucksfähigkeit von Cypher

In diesem Abschnitt wird ein Eindruck der Ausdrucksfähigkeit von Cypher vermittelt. Dazu wird zunächst beschrieben, wie relationale Operationen auf Cypher übertragen werden können. Danach wird anhand einer Beispielprozedur gezeigt, wie größere schreibende Queries formuliert werden können.

5.1.1 Übertragung relationaler Operationen

In diesem Abschnitt soll aufgezeigt werden, wie typische relationale Operationen auf einer Graphdatenbank ausgeführt werden können, die durch den im Kapitel 3 beschriebenen Algorithmus erzeugt wurde. Als Beispieldatensatz wird dazu die Beispiel-Song-DB aus Abschnitt 3.4 verwendetet. Für jede Operation wird jeweils exemplarisch eine SQL-Query und eine Cypher-Query angegeben. Um die Lesbarkeit zu erhöhen, werden die Gleichheits-Selektionen in einer where-Klausel und nicht im Pattern ausgeführt.

Verbund (Join)
Eine Verbundsanfrage an die Beispieldatenbank kann wie folgt aussehen:

```
SELECT *
FROM MUSIKER
JOIN SPIELT on (MUSIKER.mid = SPIELT.Musiker)
JOIN SONG on (SPIELT.Song = SONG.SID)
```

In der Graphdatenbank werden Fremdschlüsselbeziehungen als Relationship ausgedrückt. Um einen Join mit Gleichheitsbedingung nachzubilden muss man also lediglich einer Relationship folgen. Dies kann durch folgende Cypher-Query erfolgen.

```
MATCH (m: MUSIKER)<-[M: MUSIKER]-(s: SPIELT)-[S: SONG]->(so: SONG)
RETURN m, s, so
```

Joins, die nicht unter einer Gleichheitsbedingung erfolgen, können mit Hilfe einer where-Klausel formuliert werden. Folgende SQL-Query

```
SELECT *
FROM SONG
JOIN SONGGENRE on (SONG.ID <= SONGGENRE.SID)
```

kann wie folgt als Cypher-Query formuliert werden:

```
MATCH (song: SONG), (sg: SONGGENRE)
WHERE song.ID <= sg.SID
RETURN song, sg
```

Projektion
Eine Projektion kann durch folgende SQL-Query ausgeführt werden:

```
SELECT Musiker, Instrument
FROM SPIELT
```

Das Cypheräquivalent sieht dann so aus:

```
MATCH(s: SPIELT)
RETURN s.Musiker, s.Instrument
```

Selektion
Eine Anfrage mit Selektion kann durch diese SQL-Query ausgedrückt werden.

```
SELECT *
FROM SPIELT
WHERE Instrument='Gesang'
```

Die entsprechende Cypher-Query sieht wie folgt aus:

```
MATCH(s: SPIELT)
WHERE s.Instrument='Gesang'
RETURN s
```

Umbenennung
In SQL können sowohl Relationen als auch Attribute umbenannt werden:

```
SELECT Song AS 'Lied'
FROM SPIELT S1
WHERE S1.Instrument='Gesang'
```

In Cypher entspricht dies einer Umbenennung von Nodes und Properties:

```
MATCH(S1: SPIELT)
RETURN s1.Song as Lied
```

Vereinigung
Eine Vereinigung in einer SQL-Query kann folgende Gestalt haben:

```
SELECT *
FROM SPIELT
WHERE Instrument='Gesang'
UNION
SELECT *
FROM SPIELT
WHERE Instrument='Orgel'
```

In Cypher kann die Vereinigung mit Hilfe der filter-Funktion ausgedrückt werden:

```
MATCH (s1)
WHERE s1.Instrument='Gesang'
WITH s1
MATCH (s2)
WHERE s2.Instrument='Orgel'
WITH collect(distinct s1) as s1List,
     collect(distinct s2) as s2List
MATCH (n)
WHERE n in s1List OR n in s2List
RETURN n
```

Differenz
Eine Differenz kann in SQL wie folgt gebildet werden:

```
SELECT *
FROM SPIELT
MINUS
SELECT *
FROM SPIELT
WHERE Instrument='Orgel'
```

In Cypher ist dieser Funktion nicht implementiert, eine ähnliche Wirkung kann aber mit der filter-Funktion erreicht werden:

```
MATCH(s1: SPIELT)
WITH collect(distinct s1) as total
MATCH(s2: SPIELT)
WHERE s2.Instrument='Orgel'
WITH collect(distinct s2) as part, total
WITH filter(x in total WHERE not x in part) as sub
MATCH (n) WHERE n in sub
RETURN n
```

Aggregierung
Auch Aggregierungsfunktionen können in Cypher umgesetzt werden. In folgender SQL-Query wird der Song mit der längsten Spieldauer ermittelt:

```
SELECT *
FROM SONG
WHERE Länge =  (SELECT max(Länge)
                FROM SONG)
```

In Cypher lässt sich eine entsprechende Query formulieren:

```
MATCH (n: SONG)
WITH max(n.Länge) as max
MATCH (n: SONG)
WHERE n.Länge = max
RETURN n
```

Diese Query zählt die Anzahl an Instrumenten, die ein Musiker spielt:

```
SELECT m.Name, count(DISTINCT sp.Instrument)
FROM MUSIKER m
JOIN SPIELT sp on (m.MID = sp.MUSIKER)
GROUP BY m.Name
```

Das gleiche Ergebnis liefer folgende Cypher-Query:

```
MATCH (m: MUSIKER)<-[:MUSIKER]-(sp: SPIELT
RETURN m.Name, count(DISTINCT s.Instrument)
```

Die Gruppierung erfolgt in Cypher nach den restlichen Ausdrücken, die in der return-Klausel angegeben sind. Dies können Nodes, Relationships und einzelne Properties sein.

Verkettung
Um die Operationen zu verketten, muss nur RETURN durch WITH ausgetauscht werden. Das RETURN der letzten Operation muss allerdings erhalten bleiben. Dies liefert zwar nicht immer die kürzeste Query, ist jedoch ein mechanisch sicherer Weg die oben genannten Operationen zu verketten. Zu beachten ist, dass bei dieser Variante Projektionen nur am Schluss erfolgen dürfen.

Eine Verbundoperation auf dem Ergebnis von zwei Selektionen sieht wie folgt aus.

```
MATCH(s: SPIELT)
WHERE s.Instrument='Gesang'
WITH s
MATCH (m: MUSIKER)
WHERE m.Name='Eric Clapton'
WITH s, m
MATCH s<-[M: MUSIKER]-(m: MUSIKER)
RETURN s, m
```

Terme der Relationenalgebra
Im vorherigen wurde exemplarisch gezeigt, wie einzelne relationale Operationen umgesetzt werden können. Terme der Relationenalgebra setzen sich aus verschiedenen relationalen Operationen zusammen. Die Ausführung einzelner relationaler Operationen eines Terms lassen sich immer sequentialisieren. Diese Sequentialisierung kann in Cypher mit Hilfe von with-Klauseln, wie bei der Verkettung gezeigt, umgesetzt werden.

In diesem Vorgehen liegt ein vielversprechender Ansatz, beliebige Terme der Relationenalgebra in Cypher auszudrücken. Dies führt zu der starken Vermutung, dass Cypher relational vollständig ist.

5.1.2 Hinzufügen von Datensätzen

In diesem Abschnitt soll gezeigt werden, wie nicht nur einzelne Elemente sondern auch komplette Datensätze dem Propertygraphen hinzugefügt werden können. Dazu wird die Modulkatalog-Datenbank aus Abschnitt 4.3 benutzt. In dieser existiert die Relation „NEU_LV“. NEU_LV wird benutzt um Daten für Lehrveranstaltungen, die dem Modulkatalog hinzugefügt werden sollen, zwischenzuspeichern. Wenn eine Lehrveranstaltung zum Modulkatalog hinzugefügt werden soll, reicht es nicht, nur einer Relation einen

einzelnen Eintrag hinzuzufügen. Es muss eine Kette von Einträgen in verschiedenen Relationen erzeugt werden. Daher existiert eine PL/SQL-Prozedur, die einen Eintrag aus NEU_LV in eine solche Kette überführt. Die Kette umfasst Einträge in den Relationen LV, LV_PE, PE, PRUEFUNG und SANGEBOT. Diese Einträge werden von der PL/SQL-Prozedur der Reihe nach erzeugt.

Nun soll eine Cypher-Query konstruiert werden, die Einträge aus NEU_LV richtig in den Propertygraphen einfügt. Dabei kann die Funktionalität der PL/SQL-Prozedur nicht im vollen Umfang nachgebildet werden, da zum Beispiel Fallunterscheidungen nicht im gleichen Maße erfolgen können.

Nach Aufrufen der PL/SQL-Prozedur werden die Einträge aus NEU_LV nicht automatisch gelöscht. Um dennoch zu verhindern, dass Einträge doppelt hinzugefügt werden, existieren Spalten mit dem Präfix „GENERATED_“ im Titel. In ihnen wird ein Flag gesetzt, das je nachdem, ob die Lehrveranstaltung schon eingetragen ist, entweder auf „j“ oder „n“gesetzt wird. Ist das Flag auf „j“gesetzt, wurde die entsprechende Lehrveranstaltung schon in den Modulkatalog eingefügt. Ist das Flag auf „n“gesetzt, muss sie noch hinzugefügt werden. In PL/SQL werden diese Flags mit Hilfe von IF-Blöcken überprüft. In Cypher wird dies über eine Selektion sichergestellt.

In folgender Query werden die Einträge für ein Modul mit ID y aus einem NEU_LV-Node mit ID x in eine solche Kette überführt. Diese besteht aus einem LV-, einem PE-, einem LV_PE-, einem PRUEFUNG- und einem SANGEBOT-Node. Diese werden mit passenden Relationships verbunden. Außerdem werden Prüfer, Dozent und Betreuer aus anderen Nodes mit NEU-Präfix ausgelesen und im Propertygraphen eingefügt. Zum Schluss werden die „GENERATED“-Flags auf „j“ gesetzt. Um die Lesbarkeit zu erhöhen, werden die Gleichheits-Selektion in einer where-Klausel und nicht im Pattern ausgeführt.

```
MATCH (nlv: NEU_LV)
WHERE nlv.ID = x
AND   nlv.GENERATED_INF_ALT = 'n'
AND   nlv.GENERATED_INF_BSC = 'n'
AND   nlv.GENERATED_INF_MSC = 'n'
AND   nlv.GENERATED_TI_BSC = 'n'
AND   nlv.GENERATED_TI_MSC = 'n'
AND   nlv.GENERATED_ET_BSC = 'n'
AND   nlv.GENERATED_ET_MSC = 'n'

CREATE (lv: LV)
SET lv.ID=nlv.ID, lv.NAME = nlv.NAME, lv.ENGLISH=nlv.ENGLISH,
    lv.VSWS=nlv.VSWS, lv.USWS=nlv.USWS, lv.LSWS=nlv.LSWS,
    lv.PSWS=nlv.psws, lv.SSWS=nlv.SSWS, lv.FREQUENZ=nlv.FREQUENZ,
    lv.FREQUENZ_SEM = trim(nlv.FREQUENZ_SEM),
    lv.GUELTIG_VON = nlv.SEMESTER, lv.FREIGABE = 'j'
```

```
WITH nlv, lv
MATCH (apartner: PERSON)
WHERE apartner.ID=nlv.APARTNER
CREATE lv-[:TABLE_APARTNER_LV]->apartner

CREATE (pe: PE)
SET pe.NAME= 'Prüfung '+nlv.NAME, pe.LP=nlv.LP,
    pe.PARTEN=nlv.PARTEN, pe.BENOTET=nlv.BENOTET,
    pe.GUELTIG_VON = nlv.SEMESTER,
    pe.ARBEITSAUFWAND=str(nlv.LP*30)+' h',
    pe.ID=id(pe)

CREATE (lvpe: LV_PE)
SET    lvpe.PE =pe.ID, lvpe.LV=lv.ID, pe.GUELTIG_VON=nlv.SEMESTER
CREATE lvpe-[:LV]->lv
CREATE lvpe-[:PE]->pe
WITH nlv, lv, pe, lvpe

MERGE (mod: MODUL ID:_ y)
ON CREATE SET mod.wm='1'
ON MATCH SET mod.wm= 'WP'

CREATE (pr: PRUEFUNG)
SET pr.MODUL=mod.ID, pr.PE=pe.ID, pr.WAHLMERKMAL=mod.wm,
pr.ST_PRFLST=nlv.ST_PRFLST, pr.WEITERF=nlv.WEITERF,
pr.GUELTIG_VON=nlv.SEMESTER
CREATE pr-[:PE]->pe
CREATE pr-[:MODUL]->mod

CREATE (sag: SANGEBOT)
SET sag.VORLESUNG=nlv.VORLESUNG, sag.LV=lv.ID,
    sag.SEMESTER=nlv.SEMESTER, sag.PART=nlv.PART,
    sag.PDAUER=nlv.PDAUER, sag.link=nlv.LINK,
    sag.BEMERKUNG=nlv.BEMERKUNG,
    sag.VORKENTNISSE=nlv.VORKENTNISSE,
    sag.LERNZIELE=nlv.LERNZIELE, sag.LITERATUR=nlv.LITERATUR,
    sag.STOFFPLAN=nlv.STOFFPLAN,
    sag.BESONDERHEITEN=nlv.BESONDERHEITEN,
    sag.NOTIZ=nlv.MITTEILUNG, sag.SEMESTERTHEMA=nlv.THEMA,
    sag.FREIGABE='j', sag.VORLESITUNG=nlv.VORLEISTUNG,
    sag.MODUS_ERGPRF=trim(nlv.MODUS_ERGPRF),
    sag.DAUER_ERGPRF=nlv.DAUER_RGPRF, sag.SPRACHE=nlv.SPRACHE,
    sag.ID=ID(sag)
WITH nlv, lv, pe, lvpe, pr, mod, sag
```

```
MATCH (sem: SEMESTER)
WHERE sem.ID=nlv.SEMESTER
CREATE sag-[:SEMESTER]->sem
CREATE sag-[:LV]->lv

WITH nlv
MATCH nlv-[r1 :NEU_PRUEFER]->(per: PERSON)
CREATE nlv-[r2: PRUEFER]->per
SET r2.LV=r1.NEU_LV, r2.PRUEFER=r1.PRUEFER

WITH nlv
MATCH nlv-[r1 :NEU_DOZENT]->(per: PERSON)
CREATE nlv-[r2: DOZENT]->per
SET r2.LV=r1.NEU_LV, r2.DOZENT=r1.DOZENT

WITH nlv
MATCH nlv-[r1 :NEU_BETREUER]->(per: PERSON)
CREATE nlv-[r2: BETREUER]->per
SET r2.LV=r1.NEU_LV, r2.BETREUER=r1.BETREUER

SET   nlv.GENERATED_INF_ALT = 'j',
      nlv.GENERATED_INF_BSC = 'j',
      nlv.GENERATED_INF_MSC = 'j',
      nlv.GENERATED_TI_BSC = 'j',
      nlv.GENERATED_TI_MSC = 'j',
      nlv.GENERATED_ET_BSC = 'j',
      nlv.GENERATED_ET_MSC = 'j'
```

5.2 Graph-ADT-Operationen

In diesem Abschnitt soll das zeitliche Verhalten von Neo4j beim Ausführen einiger graphtypischer Operationen in Abhängigkeit von der Größe der gespeicherten Daten beschrieben werden. Obwohl sich keine absoluten Aussagen über die Kosten einzelner Operationen ermitteln lassen, können mittels empirischer Messungen grobe Einschätzungen getroffen werden. Die nachfolgenden Erläuterungen basieren maßgeblich auf [PMS13].

getVertex
Diese Operation gibt einen Knoten des Graphen zurück. Welcher Knoten zurückgegeben werden soll, kann zum Beispiel über eine ID oder ein Suchkriterium eingeschränkt werden. Für eine solche Operation muss für jeden Knoten genau einmal bestimmt werden, ob er dem Suchkriterium entspricht. Dies verursacht lineare Kosten in Abhängigkeit der

Anzahl existierender Knoten. Der tatsächliche Zeitbedarf verhält sich allerdings anders. Messungen in [PMS13] zeigen, dass es vier signifikante Abstufungen der benötigten Zeit gibt. Diese entsprechen den Stufen der in 2.3.2 erläuterten Speicherhierarchie. Werden die Daten zu groß für eine Stufe, erhöht sich die benötigte Zeit erheblich. Wenn die Daten nicht mehr in den Arbeitsspeicher passen, verdoppelt sich die benötigte Zeit.

getEdge
Diese Operation gibt eine Kante des Graphen mit einer bestimmten ID zurück. Die Kosten für diese Operation verhalten sich ähnlich wie die Kosten für getVertex. Da die Records der Relationships jedoch erheblich mehr Informationen als die Records der Nodes enthalten, steigt auch die benötigte Zeit in einem größeren Maße, wenn die benötigten Daten zu groß für die Caches werden. In den Messungen in [PMS13] waren die Zugriffszeiten für das Lesen von der Festplatte verglichen mit den Zeiten für das Lesen aus dem Objekt-Cache nahezu um ein fünfzigfaches größer.

getNeighbors
Diese Funktion soll alle Nachbarn eines Knoten zurückgeben. Knoten sind genau dann Nachbarn, wenn sie mit einer Kante verbunden sind. Die Richtung der Kante soll dabei keine Rolle spielen. Um alle Nachbarn zu bestimmen, müssen alle ein- und ausgehenden Kanten eines Knoten bestimmt werden und danach abgelaufen werden.

Für einen Knoten werden alle Kanten als doppelt verkettete Liste in den Relationships abgespeichert. Wir nehmen an, dass die Kosten, einem Pointer zu folgen, konstant sind. Dann sind die Kosten, die Liste mit den Kanten zu durchlaufen, linear in Abhängigkeit der Größe der Liste. Da die Relationships Pointer zu den Ziel- und Ursprungsknoten speichern, sind die Kosten einer Kante zu folgen konstant. Insgesamt sind die theoretischen Kosten für diese Operation also linear.

Praktisch lassen sich diese Kosten allerdings nicht so genau festlegen, da mit Speicherzugriffen eine nicht exakt bestimmbare Komponente dazu kommt. [PMS13] zeigt, dass in der Praxis konstante Initialisierungskosten benötigt werden. In den Messungen ist zu erkennen, dass die Kosten für das Bestimmen der Nachbarn durch eine lineare Funktion in Abhängigkeit der Anzahl der Nachbarn eingeschätzt werden können.

5.3 Funktionsumfang

Neo4j bietet mit Cypher eine für Graph-Traversierungen gut geeignete Anfragesprache. Allerdings können an dieser noch einige Komponenten ergänzt werden. So fehlt zum Beispiel eine Möglichkeit Maxima/Minima von numerischen Werten innerhalb einer where-Klausel zu bilden. Auch Unteranfragen werden nicht unterstützt. Statt dessen müssen Zwischenergebnisse mit einer with-Klausel gebildet werden. Im Nachfolgenden soll zusätzlich der Funktionsumfang von Neo4j außerhalb der Anfragesprache bewertet

werden. Dies umfasst insbesondere auch das Benennen fehlender Funktionen.

Im dedizierten Betrieb bietet Neo4j über eine Weboberfläche die Möglichkeit, Cypher-Queries komfortabel auszuführen. Dabei werden Ergebnisse wahlweise als Graph visualisiert oder in Zeilenform angezeigt.

Neo4j bietet mittles unique-Constraints die Möglichkeit, die Schemalosigkeit des Propertygraphen einzuschränken. Weitere Möglichkeiten gibt es jedoch nicht. Dies bietet auf der einen Seite den Vorteil, den Datenbestand sehr flexibel zu erweitern. Auf der anderen Seite sind die Möglichkeiten, die Konsistenz von neu abgespeicherten Daten zu überprüfen sehr begrenzt. Hier wären weitere Mechanismen wünschenswert. Diese könnten in Form von weiteren Constraints geschaffen werden. Ein zu Triggern ähnlicher Mechanismus ist in Neo4j nicht vorhanden.

Transaktionen werden unter Neo4j über eine spezielle Schnittstelle abgewickelt. Dort können sie mit den üblichen Befehlen „commit“ und „rollback“ gesteuert werden. Dabei werden auch Sperren genutzt und eventuell daraus resultierende Deadlocks behandelt.

Eine Benutzerverwaltung ist unter Neo4j nicht vorhanden. Das bedeutet, dass man ohne Zugangsdaten auf die Graphdatenbank zugreifen kann, sofern die Adresse und der Port bekannt sind. Dadurch entfällt auch eine Rechteverwaltung. Daraus resultiert, dass ein ausschließlich lesender Zugriff nicht möglich ist. Auch das Einschränken des Zugriffs auf bestimmte Elemente des Graphen, zum Beispiel alle Nodes eines Labels, ist nicht möglich. Ein generelles Passwort kann zwar über eine HT-Access-Lösung realisiert werden, diese muss jedoch manuell angelegt werden. Generell sollte ein Datenbanksystem eine solche Funktionalität bereitstellen.

5.4 Zeitliches Verhalten

5.4.1 Messumgebung

In diesem Abschnitt wird der Zeitbedarf konkreter Queries auf verschiedenen Systemen gemessen. Es werden Messungen auf einem Client/Server-System auf dem sowohl Neo4j als auch ein Oracle-SQL-Server läuft, sowie auf einem lokalen Neo4j-System ausgeführt. Der Server verfügt über 2 Intel Xeon CPUs mit jeweils 6 Kernen mit einer Frequenz von 3.06 GHZ, 12 mal 8 GB DDR3 Arbeitsspeicher und 4 600GB SAS-2 Festplatten mit jeweils 15000 Umdrehungen pro Minute. Der Server läuft unter SUSE Linux Enterprise Server 11 SP3. An diesen Server werden von einem Client-PC aus dem lokalen Netzwerk Anfragen an Neo4j und SQL gestellt. Die Neo4j-Anfragen erfolgen über die Cypher-Schnittstelle der REST-API, die SQL-Anfragen über einen JDBC-Treiber. Diese Zeiten sind also direkt vergleichbar.

Das lokale Neo4j-System besteht aus einem einzelnen Desktop-Rechner, auf dem ein dedizierter Neo4j-Server läuft. Der Desktop-Rechner verfügt über einen Intel i5 CPU mit vier Kernen mit einer Frequenz von jeweils 2.3 GHz, 4 GB DDR3 Ram und einem 248GB SATA-3 SSD-Laufwerk. Der Desktop-PC läuft unter Ubuntu 12.04 LTS. Auf diesem Rechner werden lokal Anfragen an dem Neo4j-Server gestellt. Die Anfragen erfolgen über die Cypher-Schnittstelle der REST-API.

Auf beiden Systemen läuft die Neo4j Version 2.0.4. Für die Messungen mit Indexen wurden unter Neo4j ausschließlich Schema-Indexe verwendet.

Um den Zeitbedarf einer Anfrage zu ermitteln wird jede Query hundertmal ausgeführt. Anschließend wird der arithmetische Mittelwert gebildet. Dabei müssen die Ergebnisse der Neo4j-Anfragen auf 1000 Zeilen begrenzt werden, da die Ergebnismengen sonst zu groß werden, um sie auf Applikationsebene vollständig zu laden, da die Neo4j Ergebnisse als JSON-Objekte via HTTPS übermittelt werden.

Die Neo4j-Caches werden vor Beginn der Messung mit einer Warmup-Routine gefüllt. Diese lädt zufällige Nodes, Relationships und deren Properties. Bei der Sichtung der ersten Messergebnisse fiel auf, dass die jeweils erste Ausführung einer Cypher-Query deutlich mehr Zeit benötigte, als die darauf folgenden Ausführung der gleichen Query. Dies ist darauf zurück zu führen, dass die angeforderten Daten ab der zweiten Ausführung in den Caches standen. Um diese Verfälschung der Messergebnisse durch Caches zu verhindern, wird mithilfe der Warmup-Routine eine Cache-Verdrängung zwischen jeder Ausführung einer Cypher-Query aufgerufen.

Bei den SQL-Queries konnte kein ähnlicher Effekt beobachtet werden, daher wird auf diesem System auf eine Cache-Verdrängung verzichtet. Unter Oracle-SQL wird für jedes Primary-Key-Constraint ein Index angelegt. Um trotzdem Messungen ohne Index durchzuführen, wurden die entsprechenden Relationen kopiert und kein Primary-Key-Constraint für die kopierten Relationen erstellt.

5.4.2 Queries an die Modulkatalog-Datenbank

In diesem Abschnitt werden Queries an die Modulkatalog-Datenbank aus Abschnitt 4.3 gesendet. Die Datenbank liegt sowohl in der ursprünglichen relationalen Form sowie in der transformierten Graphdatenbank-Variante vor. Die Anfragen stammen dabei teilweise aus existierenden Anwendungen, die auf die Modulkatalog-Datenbank zugreifen. Zu jeder Anfrage wird die SQL- und die Cypher-Variante und deren Laufzeit angegeben. Der Vernetzungsgrad des Propertygraphen bezeichnet die Summe der Grade aller Nodes geteilt durch die Anzahl aller Nodes. Der Grad eines Nodes ist die Anzahl ein- und ausgehender Relationships. Für die Modulkatalog-Datenbank beträgt der Vernetzungsgrad 7,6.

Query 1
Die erste Anfrage liest alle Einträge einer Relation beziehungsweise alle Nodes mit einem bestimmten Label aus:

SQL:

```
SELECT *
FROM MODKAT.SANGEBOT
```

Cypher:

```
MATCH (n: SANGEBOT)
RETURN n
```

Die Anfrage gibt 5657 Tupel/Nodes zurück.

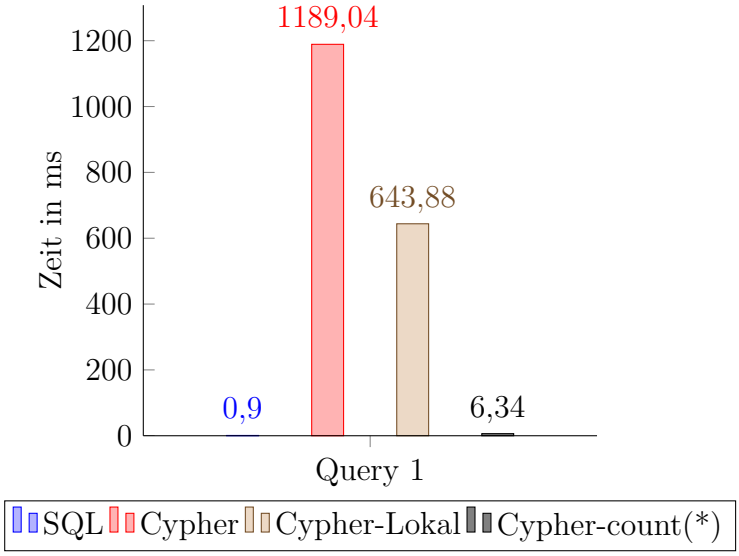

Abbildung 5.1: Zeitbedarf von Query 1

Abbildung 5.1 zeigt die mittleren Ausführungszeiten von Query 1. Erwartungsgemäß ist der SQL-Server bei dieser Anfrage deutlich (über 1000 mal) schneller. Allerdings werden beim Auslesen eines Nodes und seiner Properties auch mehr Informationen geladen als beim Auslesen einer einzelnen Zeile einer Relation. In dem Record eines Nodes wird nicht nur auf die Properties verwiesen, sondern auch auf die Relationships. Diese werden für diese Anfrage nicht benötigt aber trotzdem geladen. Die gleiche Anfrage wurde auf dem lokalen Neo4j-System ausgeführt. Dort wurde sie 546 ms schneller ausgeführt. Das legt die Vermutung nahe, dass ein Großteil des Zeitbedarfs von Query 1 für die Übertragung der Ergebnisse via HTTPS gebraucht wird.

Um die Ergebnismenge bei nahezu gleichem Rechenaufwand zu verringern, wurde die gleiche Query noch einmal mit einer return-Klausel ausgeführt, die ausschließlich

„count(*)“ beinhaltet. Diese Query konnte 186 mal schneller als die ursprüngliche Query ausgeführt werden. Dies erhärtet die Vermutung.

Query 2
In der zweiten Query wird eine einfach Selektion ausgeführt.
SQL:

```
SELECT *
FROM SANGEBOT
WHERE ID=3857
```

Cypher:

```
MATCH (sag: SANGEBOT {ID: 3857})
RETURN sag
```

Die Relation umfasst 5657 Tupel/Nodes. Die Anfrage gibt einen Tupel/Node zurück.

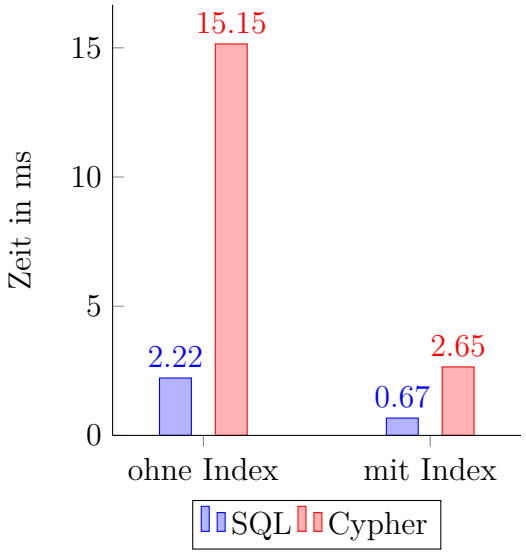

Abbildung 5.2: Zeitbedarf von Query 2

In Abbildung 5.2 sind die mittleren Ausführungszeiten von Query 2 abgebildet. Die Query wurde auf beiden Systemen jeweils einmal ohne und mit Index nach der ID ausgeführt. Auch bei dieser Anfrage ist die SQL-Datenbank schneller als Neo4j: Ohne Index benötigt Neo4j knapp 7 mal so lang wie SQL, mit Index ungefähr 4 mal. Selektionen sind für beide Datenbanksysteme essentielle Operationen und sollten daher von beiden Systemen schnell ausgeführt werden. Query 2 wurde ohne Index unter Neo4j um über 77 mal schneller als Query 1 ausgeführt. Da ohne Index trotzdem alle Tupel/Nodes gelesen werden müssen, lässt dies auch hier vermuten, dass der größte Teil der Ausführungszeit von Query 1 für die Ausgabe von Daten und deren Übertragung benötigt wird. Die Verwendung eines Indexes beschleunigt die Ausführung unter SQL um 69 % und unter

Neo4j um 82 %.

Query 3
Query 3 enthält ebenfalls eine Selektion, diesmal jedoch über einen Bereich:
SQL:

```
SELECT *
FROM SANGEBOT
WHERE ID>3000 AND ID<4000
```

Cypher:

```
MATCH (sag: SANGEBOT)
WHERE sag.ID>3000 AND sag.ID<4000
RETURN sag
```

SANGEBOT enthält 5589 Tupel/Nodes, die Anfrage gibt 281 Tupel/Nodes zurück. Abbildung 5.3 zeigt den Zeitbedarf von Query 3. Auch hier ist die SQL-Variante deut-

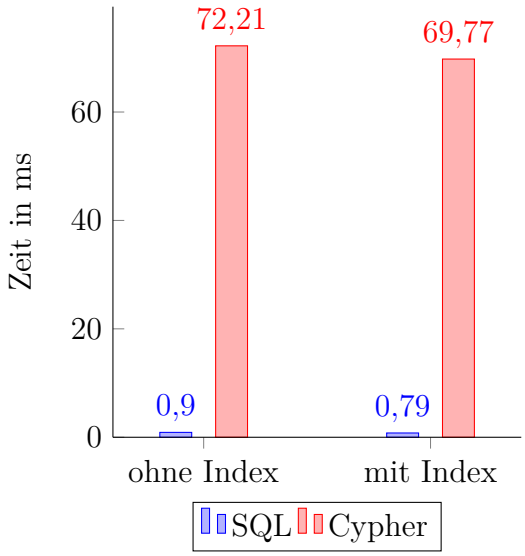

Abbildung 5.3: Zeitbedarf von Query 3

lich schneller als die entsprechende Cypher-Query. Auffällig ist, dass die mittlere Ausführungszeit der Cypher-Query sich mit Index nur wenig verringert. In beiden Fällen braucht Neo4j mehr als 80 mal so lange wie SQL. Dies spricht dafür, dass die Größe des Ergebnisses einen spürbaren Einfluss auf die Ausführungszeit hat, da die Größe der Ergebnisse unabhängig vom Index ist. Um diesen Effekt weiter zu untersuchen wurde die Anzahl der selektierten Nodes unter Neo4j in Query 3 variiert und die Ausführungszeit gemessen. Die Ergebnisse sind in Abbildung 5.4 und 5.5 zu sehen. Es ist sehr deutlich zu erkennen, dass mit der Größe des Ergebnisses auch die Ausführungszeit zu nimmt. Dabei gibt es auch hier kaum einen Unterschied zwischen der Variante mit Index und

ohne Index. Die Ergebnisse legen die Vermutung nahe, dass ein linearer Zusammenhang zwischen der Anzahl zurückgegebener Nodes und der benötigten Zeit existiert. Um dies beurteilen zu können, wurde eine lineare Regression für beide Varianten durchgeführt. In beiden Fällen galt für das Bestyimmtheitsmaß R^2: $R^2 > 0,99$. Das ist ein sehr starkes Indiz dafür, dass ein linearer Zusammenhang besteht. Auch die benötigte Zeit von Query 1 lässt sich in diesen linearen Zusammenhang einordnen.

Dies spricht dafür, dass die meiste Zeit nicht zum Ausführen der Selektion sondern zum Übertragen der Ergebnisse benötigt wird.

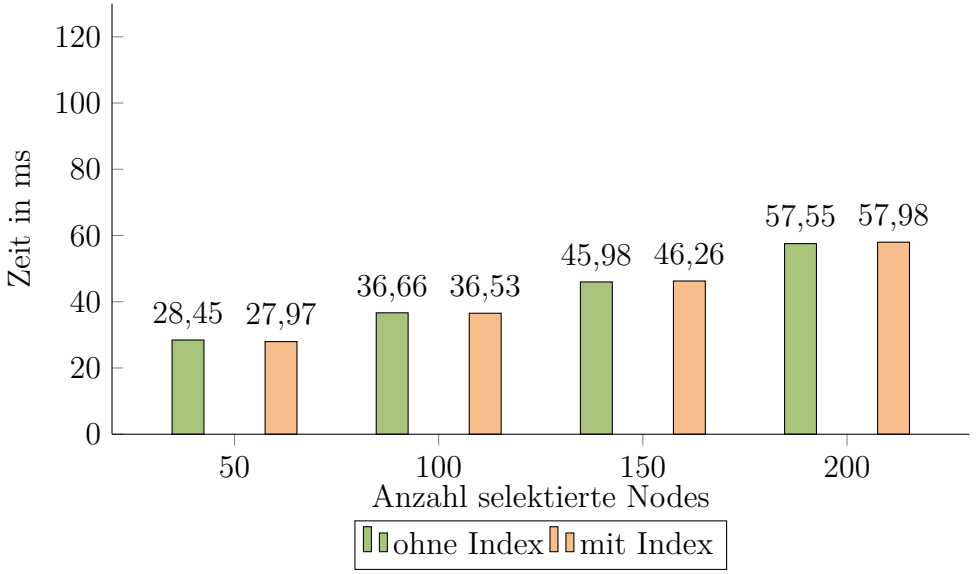

Abbildung 5.4: Zeitbedarf in Abhängigkeit der Selektionsgröße, Teil 1

Query 4 bis Query 9

In der folgenden Serie von Queries sollen die Kosten für das Bilden eines Joins und dem Auflösen einer Relationship bei einem konstanten Startpunkt verglichen werden. Dabei werden 3 Stufen mit jeweils unterschiedlicher Anzahl von Joins/Relationships betrachtet. Für jede Stufe wird der Startpunkt einmal per Selektion auf die ID-Property bzw. die ID-Spalte bestimmt und einmal per Row-/Node-ID. Die Queries lauten wie folgt:

SQL 4

```
SELECT *
FROM MODKAT.SANGEBOT sag
JOIN MODKAT.LV lv on (sag.LV = lv.ID)
JOIN MODKAT.LV_PE lvpe on (lvpe.lv = lv.ID)
JOIN MODKAT.PE pe on (pe.ID = lvpe.PE)
WHERE sag.rowid='AAAVGGAAEAAAK6jAAD'
```

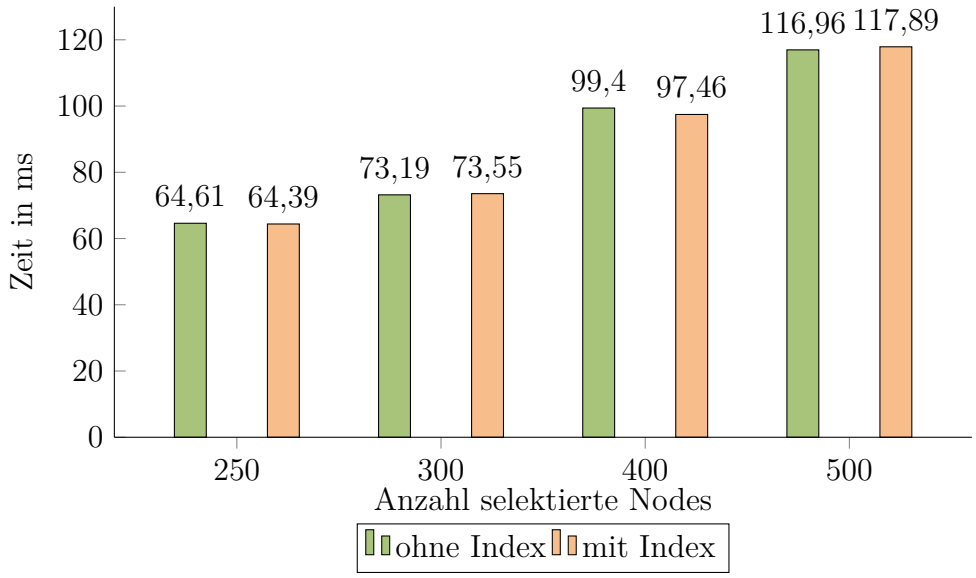

Abbildung 5.5: Zeitbedarf in Abhängigkeit der Selektionsgröße, Teil 2

Cypher 4

```
START sag=node(2727)
MATCH sag-[:LV]->(lv: LV)<-[:LV]-(lvpe: LV_PE)-[:PE]->(pe: PE)
RETURN sag, lv, lvpe, pe
```

SQL 5

```
SELECT *
FROM MODKAT.SANGEBOT sag
JOIN MODKAT.LV lv on (sag.LV = lv.ID)
JOIN MODKAT.LV_PE lvpe on (lvpe.lv = lv.ID)
JOIN MODKAT.PE pe on (pe.ID = lvpe.PE)
WHERE sag.id=5006
```

Cypher 5

```
MATCH (sag: SANGEBOT {ID: 5006})-[:LV]->(lv: LV)<-[:LV]-
       (lvpe: LV_PE)-[:PE]->(pe: PE)
RETURN sag, lv, lvpe, pe
```

Query 4 und 5 geben jeweils eine Zeile zurück.

SQL 6

```
SELECT *
FROM MODKAT.SANGEBOT sag
```

```
JOIN MODKAT.LV lv on (sag.LV = lv.ID)
JOIN MODKAT.LV_PE lvpe on (lvpe.lv = lv.ID)
JOIN MODKAT.PE pe on (pe.ID = lvpe.PE)
JOIN MODKAT.PRUEFUNG pruef on (pruef.PE = pe.ID)
JOIN MODKAT.MODUL mod on (pruef.MODUL = mod.ID)
WHERE sag.rowid='AAAVGGAAEAAAK6jAAD'
```

Cypher 6

```
START sag=node(2727)
MATCH sag-[:LV]->(lv: LV)<-[:LV]-(lvpe: LV_PE)
      -[:PE]->(pe: PE) <-[:PE]-(pruef: PRUEFUNG)
      -[:MODUL]->(mod: MODUL)
RETURN sag, lv, lvpe, pe, pruef, mod
```

SQL 7

```
SELECT *
FROM MODKAT.SANGEBOT sag
JOIN MODKAT.LV lv on (sag.LV = lv.ID)
JOIN MODKAT.LV_PE lvpe on (lvpe.lv = lv.ID)
JOIN MODKAT.PE pe on (pe.ID = lvpe.PE)
JOIN MODKAT.PRUEFUNG pruef on (pruef.PE = pe.ID)
JOIN MODKAT.MODUL mod on (pruef.MODUL = mod.ID)
WHERE sag.id=5006
```

Cypher 7

```
MATCH (sag: SANGEBOT {ID: 5006})-[:LV]->(lv: LV)<-[:LV]-
       (lvpe: LV_PE)-[:PE]->(pe: PE)<-[:PE]-(pruef: PRUEFUNG)
       -[:MODUL]->(mod: MODUL)
RETURN sag, lv, lvpe, pe, pruef, mod
```

Query 6 und 7 geben jeweils 9 Zeilen zurück.

SQL 8

```
SELECT *
FROM MODKAT.SANGEBOT sag
JOIN MODKAT.LV lv on (sag.LV = lv.ID)
JOIN MODKAT.LV_PE lvpe on (lvpe.lv = lv.ID)
JOIN MODKAT.PE pe on (pe.ID = lvpe.PE)
JOIN MODKAT.PRUEFUNG pruef on (pruef.PE = pe.ID)
JOIN MODKAT.MODUL mod on (pruef.MODUL = mod.ID)
JOIN MODKAT.KB_MODUL kbmod on (kbmod.MODUL = mod.ID)
```

```
JOIN MODKAT.KB on (kbmod.KB = kb.ID)
JOIN MODKAT.STUDIENGANG sg on (kb.SG = sg.ID)

WHERE sag.rowid='AAAVGGAAEAAAK6jAAD'
```

Cypher 8

```
START sag=node(2727)
MATCH sag-[:LV]->(lv: LV)<-[:LV]-(lvpe: LV_PE)
      -[:PE]->(pe: PE) <-[:PE]-(pruef: PRUEFUNG)
      -[:MODUL]->(mod: MODUL))<-[:MODUL]-(kbmod: KB_MODUL)
      -[:KB]->(kb: KB)-[:SG]->(sg: STUDIENGANG)
RETURN sag, lv, lvpe, pe, pruef, mod, kbmod, kb, sg
```

SQL 9

```
SELECT *
FROM MODKAT.SANGEBOT sag
JOIN MODKAT.LV lv on (sag.LV = lv.ID)
JOIN MODKAT.LV_PE lvpe on (lvpe.lv = lv.ID)
JOIN MODKAT.PE pe on (pe.ID = lvpe.PE)
JOIN MODKAT.PRUEFUNG pruef on (pruef.PE = pe.ID)
JOIN MODKAT.MODUL mod on (pruef.MODUL = mod.ID)
JOIN MODKAT.KB_MODUL kbmod on (kbmod.MODUL = mod.ID)
JOIN MODKAT.KB on (kbmod.KB = kb.ID)
JOIN MODKAT.STUDIENGANG sg on (kb.SG = sg.ID)
WHERE sag.id=5006
```

Cypher 9

```
MATCH (sag: SANGEBOT {ID: 5006})-[:LV]->(lv: LV)<-[:LV]-
      (lvpe: LV_PE)-[:PE]->(pe: PE)<-[:PE]-(pruef: PRUEFUNG)
      -[:MODUL]->(mod: MODUL))<-[:MODUL]-(kbmod: KB_MODUL)
      -[:KB]->(kb: KB)-[:SG]->(sg: STUDIENGANG)
RETURN sag, lv, lvpe, pe, pruef, mod, kbmod, kb, sg
```

Query 8 und 9 geben jeweils 11 Zeilen zurück.

In Abbildung 5.6 sind die Zeiten der Queries zu sehen, deren Startpunkte mit Hilfe einer Row-/Node-ID festgelegt wurden sind. Die SQL-Zeiten sind so gering, dass die zeitlichen Abweichungen auch durch anfragefremde Faktoren verursacht sein könnten. Neo4j ist bei der Ausführung ein 3-, 4- bzw. 8-faches langsamer als SQL. Da die SQL-Zeiten sich nahezu konstant verhalten, sind vor allem die Neo4j-Zeiten interessant. In Query 4 sind drei Relationships vorhanden, in Query 6 kommen zwei hinzu, in Query 8 weitere 3. Für die Erweiterungen werden zusätzlich einmal etwa 10 ms und einmal etwa

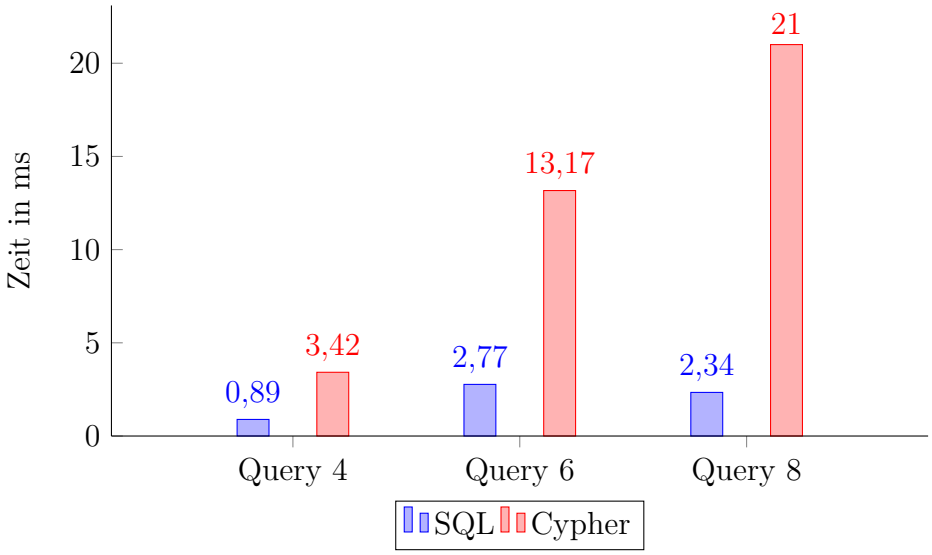

Abbildung 5.6: Zeitbedarf der Queries mit Row-/Node-ID Startpunkt

8 ms benötigt. Man sollte dabei beachten, dass sich mit der Anzahl der Kanten auch die Menge der auszugebenden Daten erhöht, und damit die Kosten für das Auflösen der Relationships selber wahrscheinlich etwas geringer sind.

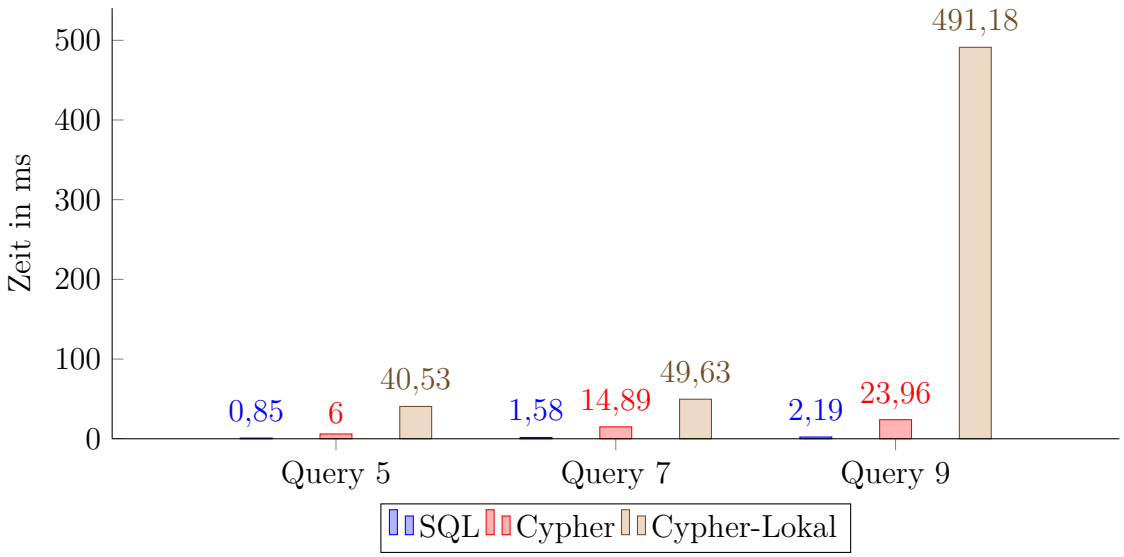

Abbildung 5.7: Zeitbedarf der Queries mit ID-Selektions-Startpunkt

Abbildung 5.7 zeigt die Variante der Queries, bei denen der Startpunkt über eine Selektion auf ID-Properties/Felder erfolgt ist. Auch hier variieren die Neo4j-Zeiten mehr als die SQL-Zeiten, daher sind auch hier eher die Neo4j-Zeiten interessant. Query 5, 7 und 9 benötigen etwa 2-3 ms mehr Zeit als Query 4 und 6. Diese Zeit wird wahrscheinlich von der Selektion beansprucht. Auffällige Ergebnisse konnten beim Ausführen auf dem lokalen Neo4j-System beobachtet werden. Dort benötigt Query 9 signifikant

länger als Query 5 und 7. Diesen Effekt könnte man damit erklären, dass Neo4j die Selektion erst nach dem Graphmatching ausführt. In diesem Fall würde ein große Zwischenergebnismenge entstehen, auf die dann eine Selektion ausgeführt wird. Mithilfe einer with-Klausel könnte man eine frühere Ausführung der Selektion erzwingen, was demnach zu geringeren Ausführungszeiten führen würde. Dies führt zu den Queries 5+, 7+ und 9+:

Cypher 5+

```
MATCH (sag: SANGEBOT {ID: 5006})
WITH  sag
MATCH sag-[:LV]->(lv: LV)<-[:LV]-
       (lvpe: LV_PE)-[:PE]->(pe: PE)
RETURN sag, lv, lvpe, pe
```

Cypher 7+

```
MATCH (sag: SANGEBOT {ID: 5006})
WITH  sag
MATCH sag-[:LV]->(lv: LV)<-[:LV]-
       (lvpe: LV_PE)-[:PE]->(pe: PE)<-[:PE]-(pruef: PRUEFUNG)
       -[:MODUL]->(mod: MODUL)
RETURN sag, lv, lvpe, pe, pruef, mod
```

Cypher 9+

```
MATCH (sag: SANGEBOT {ID: 5006})
WITH  sag
MATCH sag-[:LV]->(lv: LV)<-[:LV]-
      (lvpe: LV_PE)-[:PE]->(pe: PE)<-[:PE]-(pruef: PRUEFUNG)
      -[:MODUL]->(mod: MODUL))<-[:MODUL]-(kbmod: KB_MODUL)
      -[:KB]->(kb: KB)-[:SG]->(sg: STUDIENGANG)
RETURN sag, lv, lvpe, pe, pruef, mod, kbmod, kb, sg
```

Abbildung 5.8 zeigt die benötigten Ausführungszeiten mit und ohne with-Klausel auf dem lokalen, 5.9 auf dem Client/Server-System. Auf beiden Systemen erfolgt die Ausführung mit with-Klausel in kürzerer Zeit. Der Cypher-Anfrageoptimierer scheint also keine Selektionen vorzuziehen. Dies ist ein signifikanter Mangel für ein Datenbanksystem.

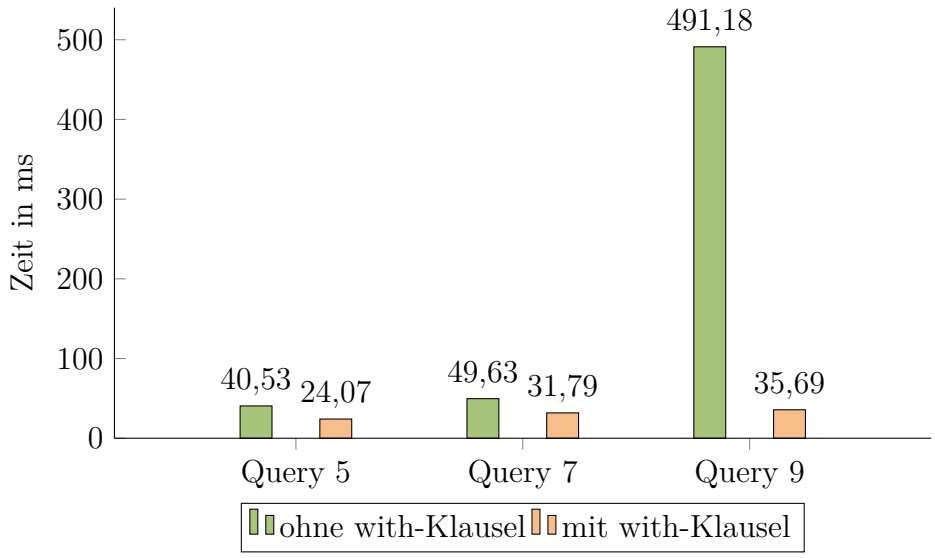

Abbildung 5.8: Zeitbedarf mit und ohne with-Klausel auf dem lokalen System

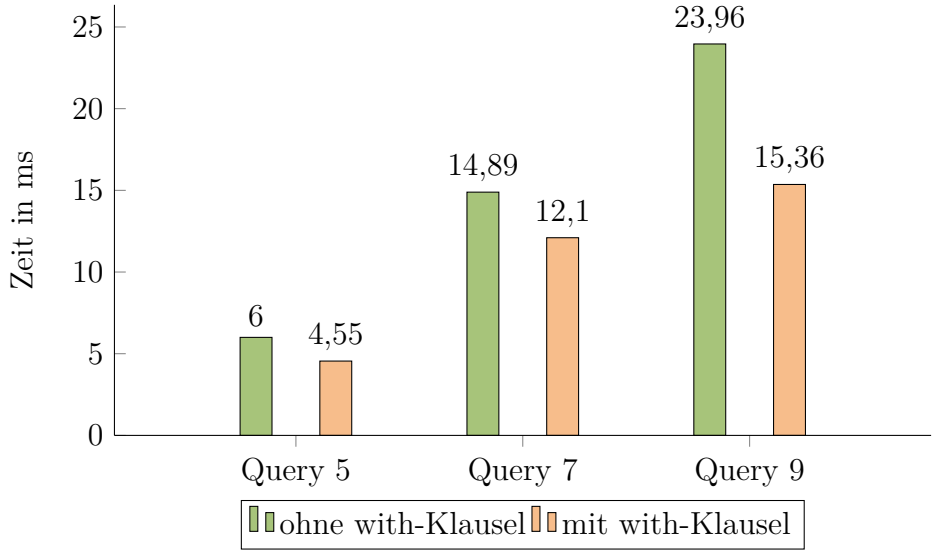

Abbildung 5.9: Zeitbedarf mit und ohne with-Klausel auf dem Client/Server-System

Query 10
In der zehnten Query werden vier Joins und eine Selektion ausgeführt. Der Zeitbedarf der Query wurde mit und ohne Index gemessen.

SQL:

```
SELECT * FROM SANGEBOT sag
JOIN LV lv on (sag.LV = lv.ID)
JOIN LV_PE lvpe on (lvpe.LV = lv.ID)
JOIN PE pe on (lvpe.PE = pe.ID)
```

```
JOIN PRUEFUNG pruef on (pruef.PE = pe.ID)
WHERE sag.ID> 3000 AND sag.ID<4000
```

Cypher:

```
MATCH (sag: SANGEBOT)-[:LV]->(lv: LV)<-[:LV]-(lvpe: LV_PE)-[:PE]->
      (pe: PE)<-[:PE]-(pruef: PRUEFUNG)
WHERE sag.ID > 3000 AND sag.ID < 4000
RETURN sag, lv, lvpe, p, pruef
```

Dies Joins bilden 23035 Tupel, die Anfrage gibt 1033 Zeilen zurück.

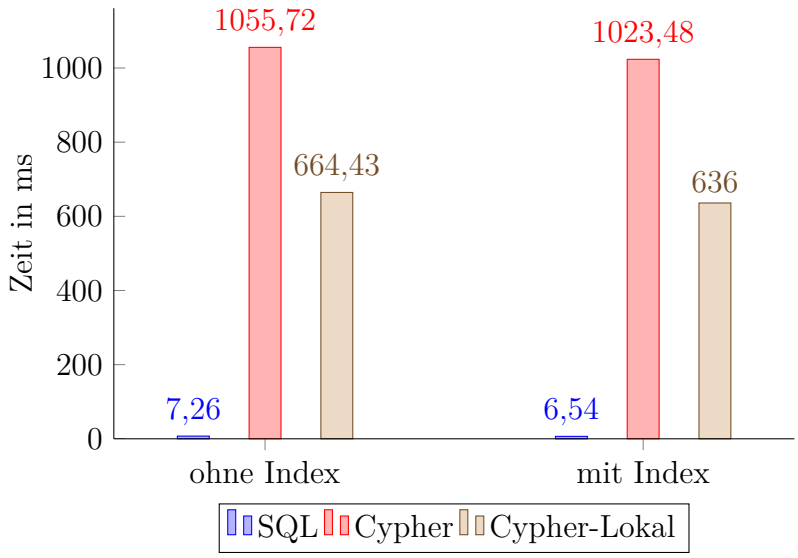

Abbildung 5.10: Zeitbedarf von Query 10

Abbildung 5.10 zeigt die mittleren Ausführungszeiten von Query 10. Hier ist Neo4j deutlich langsamer als SQL: Ohne Index etwa 145 und mit Index etwa 156 mal. Unter SQL standen Indexe für die Primärschlüssel der Relationen LV, LV_PE, PRUEFUNG und SANGEBOT zur Verfügung, unter Neo4j nur ein Index auf Nodes mit dem Label SANGEBOT nach der Property ID, so dass auf beiden Systemen alle Indexe, die für diese Anfrage genutzt werden konnten, auch existierten. In Neo4j müssen nur Relationships ausgelesen werden. Diese Operation ist für ein Graph-Datenbankensystem elementar und sollte deshalb keine hohen Kosten verursachen, während in SQL die vergleichsweise teuren Hash-Joins verwendet wurden. Bei dieser Query, die mit 1033 Tupeln eine relativ große Ergebnismenge liefert, war das lokale Neo4j-System schneller als die Client/Server-Variante. Dies ist ein weiteres Anzeichen dafür, dass die Übertragung der Ergebnisse viel Zeit beansprucht.

Auch für diese Query wurde eine frühere Ausführung der Selektion mit Hilfe einer with-Klausel erzwungen. In Abbildung 5.11 ist zu sehen, dass auch diese Selektion nicht als

erstes im Cypher-Ausführungsplan ausgeführt wird.

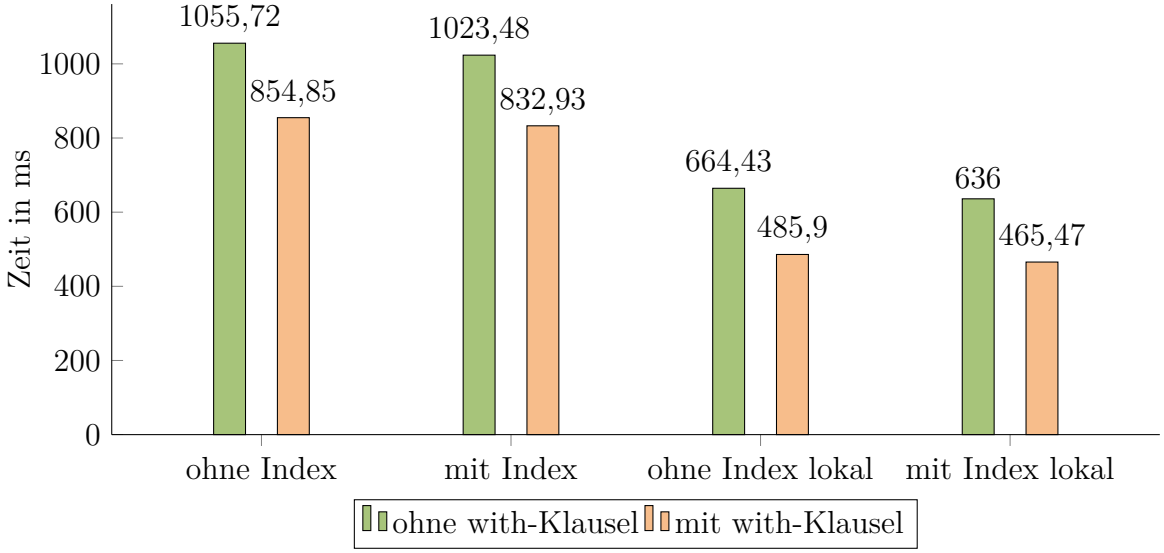

Abbildung 5.11: Query 10 mit erzwungener Selektion

Query 11
Query 11 repräsentiert eine Query aus einer realen Anwendung. Die Query sucht nach Lehrveranstaltungen mit dem spätesten Semesterangebot. Die ursprüngliche Query verwendet mehrere Unteranfragen. Da Unteranfragen in Cypher nicht möglich sind muss ein semantisch äquivalentes Zwischenergebnis mit Hilfe einer WITH-Klausel erzeugt werden. Außerdem richtet sich die Query an eine View, die ebenfalls mit Hilfe von Unteranfragen gebildet wurde. Die in der Query formulierten Joins bilden ein ähnliches Ergebnis.

SQL:

```
SELECT *
FROM LV lvo
   JOIN LV_PE lvpeo on (lvpeo.lv = lvo.id)
   JOIN PE peo on (peo.id = lvpeo.pe)
   JOIN PRUEFUNG pro on (pro.pe = peo.id)
   JOIN MODUL modo on (modo.id = pro.MODUL)
   JOIN KB_MODUL kbmodo on (kbmodo.MODUL = modo.id)
   JOIN KB kbo on (kbo.id = kbmodo.KB)
   JOIN STUDIENGANG sgo on (kbo.SG = sgo.ID)
   JOIN TABLE_APARTNER_LV talvo on (talvo.LV = lvo.id)
   JOIN PERSON pero on (talvo.APARTNER = pero.id)
WHERE
     (sgo.id='20')
     AND (kbo.id='1')
     AND lvo.freigabe LIKE 'j'
```

```
    AND sgo.gueltig_von <= 47
    AND lvo.gueltig_von <=47
    AND lvpeo.gueltig_von <= 47
    AND pro.gueltig_von <= 47
    AND kbmodo.gueltig_von <= 47
    AND sgo.gueltig_bis >= 100
    AND lvo.gueltig_bis >= 100
    AND lvpeo.gueltig_bis >= 100
    AND pro.gueltig_bis >= 100
    AND kbmodo.gueltig_bis >= 100
    AND exists
  (SELECT id
   FROM MODKAT.sangebot
   WHERE
         lv=lvo.id
         AND freigabe LIKE 'j'
         AND semester =
    (SELECT max(semester)
     FROM MODKAT.sangebot
     WHERE
           lv=lvo.id
           AND freigabe LIKE 'j'
           AND semester <= 47)
```

Cypher:

```
MATCH (lv: LV)<-[:LV]-(lvpe: LV_PE)-[:PE]->(pe: PE)<-[:PE]-
      (pruef: PRUEFUNG)-[:MODUL]->(mod: MODUL)<-[:MODUL]-
      (kbmod: KB_MODUL)-[:KB]->(kb: KB)-[:SG]->(sg: STUDIENGANG),
       lv-[:TABLE_APARTNER_LV]->(per: PERSON)
WHERE  sg.ID=20 AND kb.ID=1
       AND sg.GUELTIG_VON <= 47 AND lv.GUELTIG_VON <= 47
       AND lvpe.GUELTIG_VON <= 47 AND pruef.GUELTIG_VON <= 47
       AND kbmod.GUELTIG_VON <= 47
       AND sg.GUELTIG_BIS >= 100 AND lv.GUELTIG_BIS >= 100
       AND lvpe.GUELTIG_BIS >= 100
       AND pruef.GUELTIG_BIS >= 100
       AND kbmod.GUELTIG_BIS >= 100
WITH   lv, lvpe, pe, pruef, mod, kbmod, kb, sg, per
MATCH  lv<-[:LV]-(m: SANGEBOT)
WHERE  m.FREIGABE ='j' and m.SEMESTER <= 47
WITH   lv, max(m.SEMESTER) as mak, lvpe, pe, pruef, mod, kbmod,
       kb, sg, per
MATCH  lv<-[:LV]-(m: SANGEBOT)
```

```
WHERE  m.SEMESTER = mak AND m.FREIGABE ='j' and m.SEMESTER <= 47
RETURN lv, lvpe, pe, pruef, mod, kbmod, kb, sg,  per
```

Die Joins bilden 3460 Tupel, diese Anfrage liefert 32 Zeilen zurück.

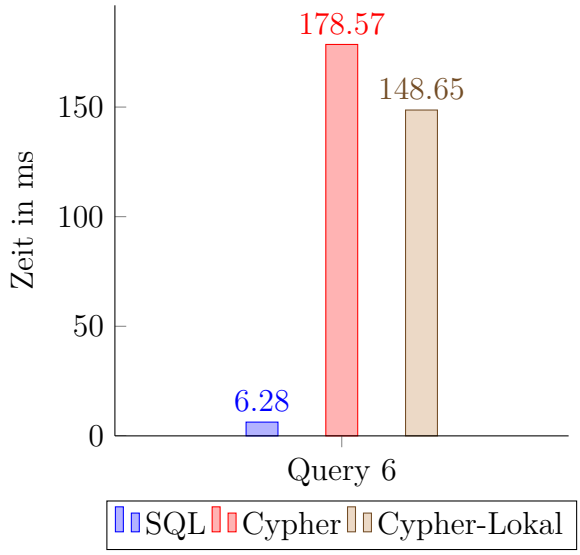

Abbildung 5.12: Zeitbedarf von Query 11

Die mittleren Ausführungszeiten für Query 11 sind in Abbildung 5.12 zu sehen. Hier benötigen beide Systeme unter einer Sekunde. Neo4j benötigt etwa 28 mal so lange wie SQL. Relativ zu den vorherigen Queries schneidet Neo4j hier gut ab, im absoluten Vergleich ist es jedoch um einiges langsamer. Eine erzwungene frühere Ausführung der Selektionen konnte bei dieser Query keine zeitliche Verbesserung bewirken. Um einschätzen zu können, welche Teile der Query welchen Anteil an der Ausführungszeit haben, wurde die Query auch in Teilen ausgeführt. Die beiden with-Klauseln definieren dabei die Grenze zwischen den Teilen und wurden nacheinander durch returns ersetzt. Die Teilqueries wurden auf dem lokalen System ausgeführt, um Verzerrung durch Übertragung zu vermeiden. In Abbildung 5.13 sind die Ergebnisse zu sehen.

Die meiste Zeit wird also für das erste Patternmatching und die erste Selektion benötigt. Diese liefert genau einen LV-Node, auf dem die weiteren matches erfolgen. Dass erklärt die geringen Zeiten, die danach für die anschließenden Schritte benötigt werden.

Query 12

Auch Query 12 stammt aus einer existierenden Anwendung. Sie wird von einer Maske erzeugt, in der einzelne Semesterangebote gesucht werden können. In dieser Query wurden Studiengang, Kompetenzbereich und Modul angegeben. Zusätzlich sollen die Worte „Information Retrieval“ in irgendeiner Form im Semesterangebot enthalten sein. Als Dozent wurde der Name „Neidjl“ angegeben. Auch hier wurde die Anfrage ursprünglich an eine View gerichtet, die Joins über nahezu alle Relationen bildet. In der View wurden in

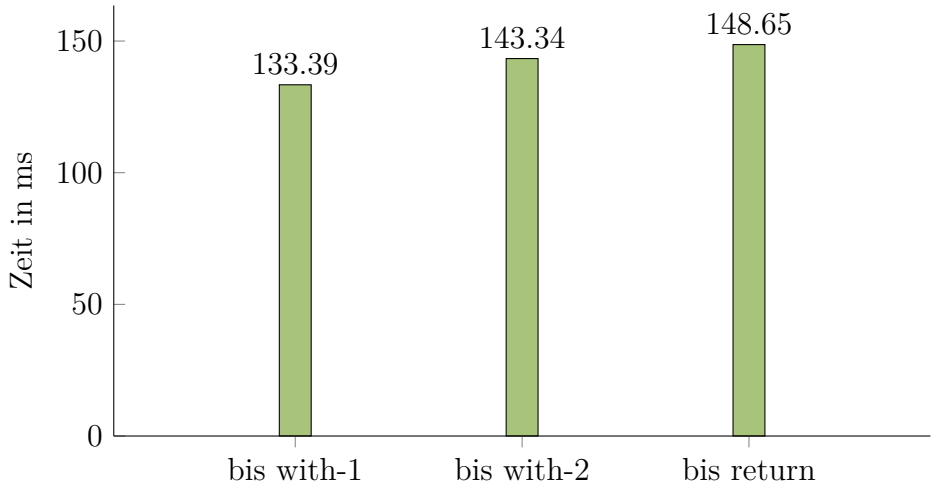

Abbildung 5.13: Teile von Query 11

weiteren Views Unteranfragen gestellt. Um die Queries vergleichbar zu gestalten, wurde die Anfrage daher leicht verändert.

```
SELECT *
FROM
     MODKAT.STUDIENGANG sg
     JOIN MODKAT.KB on (KB.SG = sg.ID)
     JOIN MODKAT.KB_MODUL kbmod on (kbmod.KB = KB.ID)
     JOIN MODKAT.MODUL mod on (kbmod.MODUL = mod.ID)
     JOIN MODKAT.PRUEFUNG pruef on (pruef.MODUL = mod.ID)
     JOIN MODKAT.PE pe on (pe.ID = pruef.PE)
     JOIN MODKAT.LV_PE lvpe on (lvpe.PE = pe.ID)
     JOIN MODKAT.LV lv on (lvpe.LV = lv.ID)
     JOIN MODKAT.SANGEBOT sag on (sag.LV = lv.ID)
     LEFT OUTER JOIN MODKAT.TABLE_DOZENT tdoz
          on (tdoz.SANGEBOT = sag.ID)
     LEFT OUTER JOIN MODKAT.PERSON doz
          on (tdoz.DOZENT= doz.ID)
     LEFT OUTER JOIN MODKAT.TABLE_BETREUER tbet
          on (tbet.SANGEBOT = sag.ID)
     LEFT OUTER JOIN MODKAT.PERSON  bet
          on (tbet.BETREUER = bet.ID)
     LEFT OUTER JOIN MODKAT.TABLE_PRUEFER tpru
          on (tpru.SANGEBOT = sag.ID)
     LEFT OUTER JOIN MODKAT.PERSON  pru
          on (tpru.PRUEFER = pru.ID)
WHERE
    (sg.id='21')
```

```
AND (kb.id='4') AND
(mod.id='96') AND
UPPER(lv.name) LIKE UPPER('%Information%')
AND sprache LIKE 'e'
AND sag.SEMESTER >='43'
AND sag.SEMESTER  <='45'
AND (UPPER(doz.name) LIKE UPPER('%Nejdl%')
     OR UPPER(pru.name) LIKE UPPER('%Nejdl%'))
AND (UPPER(lv.name) LIKE UPPER('%Retrieval%')
     OR UPPER(lv.ENGLISH) LIKE UPPER('%Retrieval%')
     OR UPPER(sag.lernziele) LIKE UPPER('%Retrieval%')
     OR UPPER(sag.stoffplan) LIKE UPPER('%Retrieval%')
     OR UPPER(sag.vorkentnisse) LIKE UPPER('%Retrieval%')
     OR UPPER(sag.literatur) LIKE UPPER('%Retrieval%'))
```

Cypher:

```
MATCH (sag: SANGEBOT)-[:LV]->(lv: LV)<-[:LV]-(lvpe: LV_PE)-[:PE]->
      (pe: PE)<-[:PE]-(pruef: PRUEFUNG)-[:MODUL]->(mod: MODUL)
      <-[:MODUL]-(kbmod: KB_MODUL)-[:KB]->(kb: KB)-[:SG]->
      (sg: STUDIENGANG)
WHERE  sg.ID=21 AND kb.ID=4 AND mod.ID=96 AND sag.SPRACHE='e'
       AND sag.SEMESTER >= 43 AND sag.SEMESTER <= 45
WITH  lv, lvpe, pe, pruef, mod, kbmod, kb, sag
OPTIONAL MATCH sag-[:TABLE_PRUEFER]->(pruefer: PERSON)
OPTIONAL MATCH sag-[:TABLE_DOZENT]->(dozent: PERSON)
OPTIONAL MATCH sag-[:TABLE_BETREUER]->(betreuer: PERSON)
WITH  lv, lvpe, pe, pruef, mod, kbmod, kb, pruefer, dozent,
      betreuer, sag
WHERE upper(lv.NAME)=~ upper('.*Information.*')
      AND (upper(dozent.NAME) =~ upper('.*Nejdl.*')
           OR  upper(pruefer.NAME) =~ upper('.*Nejdl.*'))
      AND(upper(lv.NAME)=~ upper('.*Retrieval.*')
           OR upper(lv.ENGLISH)=~ upper('.*Retrieval.*')
           OR upper(sag.LERNZIELE)=~ upper('.*Retrieval.*')
           OR upper(sag.STOFFPLAN)=~ upper('.*Retrieval.*')
           OR upper(sag.VORKENTNISSE)=~ upper('.*Retrieval.*')
           OR upper(sag.STOFFPLAN)=~ upper('.*Retrieval.*')
           OR upper(sag.LITERATUR)=~ upper('.*Retrieval.*'))
RETURN  lv, lvpe, pe, pruef, mod, kbmod, kb, pruefer, dozent,
        betreuer, sag
```

Die Joins bilden 47693 Tupel, die Anfrage gibt 2 Zeilen zurück.

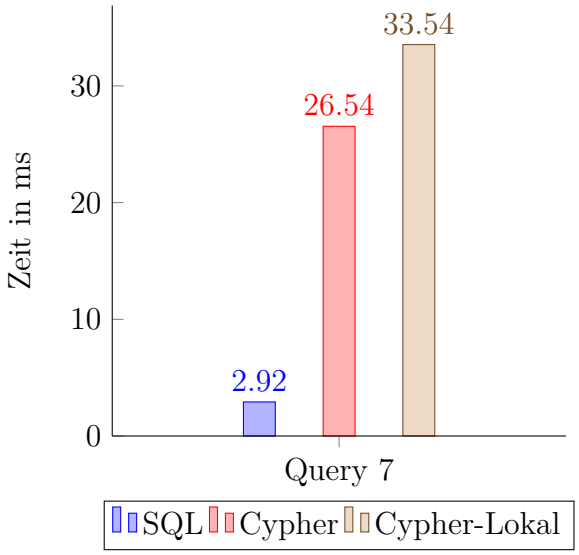

Abbildung 5.14: Zeitbedarf von Query 12

Abbildung 5.14 zeigt den mittleren Zeitbedarf von Query 12. Neo4j benötigt etwa 9 mal so viel Zeit wie SQL. In Relation zu den vorherigen Anfragen ist dies eine relativ gute Zeit. In der Cypher-Query wird mit der with-Klausel ein Zwischenergebnis erzeugt. Durch die hohe Selektivität der ersten where-Klausel wird ein expliziter Startpunkt für das Patternmatching angegeben. Nach der with-Klausel müssen die optional-matches nur auf fünf SANGEBOT-Nodes ausgeführt werden. Dies könnte zur der relativ geringen Laufzeit führen. Für diese Query benötigt das lokale Neo4j-System mehr Zeit als das Client/Server-System. Das lässt darauf schließen, dass für diese Query die meiste Zeit zur Bestimmung der Ergebnismenge benötigt wird und nicht zur Übertragung der Ergebnisse. Auch hier bewirkt eine erzwungene frühere Ausführungen der Selektion keine zeitliche Verbesserung.

Übersicht

Tabelle 5.1 liefert eine Übersicht über die benötigten Zeiten aller Systeme. Die Faktoren geben an, welches Vielfache der Zeit von Oracle-SQL Neo4j benötigt. Für Queries die sowohl mit als auch ohne Index ausgeführt wurden, wurde dabei immer die Variante mit Index gewählt. Für Cypher-Queries wurde immer die Variante mit erzwungener Selektion gewählt, falls vorhanden. Faktoren wurden auf zwei Stellen nach dem Komma gerundet. Alle Zeiten werden in Millisekunden angegeben, die Ergebnisgröße in Zeilen.

In diesen Messungen war Neo4j in keinen einzigen Fall schneller als SQL. In Extremfällen ist SQL um über das 1000-fache schneller als Neo4j. Besonders auffällig war, dass die benötigte Zeit zum Ausführen einer Query zum größten Teil zum Übertragen der Ergebnisse benötigt wurde. Bei kleineren Ergebnismengen verbessert sich zwar das zeitliche Verhalten von Neo4j, im besten Fall benötigt Neo4j jedoch immer noch mehr als das 3-fache der Zeit, die SQL benötigt. Für diese Datenbank würde sich der Einsatz von Neo4j also nicht lohnen.

Query	Ergebnisgröße	SQL	Neo4j	Faktor	Neo4j-Lokal	Faktor
1	5657	0,9	1189,04	1321,16	643,88	715.42
2	1	0,67	2,65	3,96	–	–
3	281	0,79	69,77	88,32	–	–
4	1	0,89	3,42	3,84	–	–
5	1	0,85	4,55	5,35	24,07	28,32
6	9	2,77	13,17	4,75	–	–
7	9	1,58	12,1	7,66	31,79	20,12
8	11	2,34	21,0	8,97	–	–
9	11	2,19	15,36	7,01	35,69	16,3
10	1033	6,54	832,93	127,36	465,47	71,17
11	2	6,28	178,57	28,43	148,65	23,67
12	2	2,92	26,54	9,09	33,54	11,49

Tabelle 5.1: Übersicht über die Ausführungszeiten

5.4.3 Anfragen mit geographischen Daten

In diesem Abschnitt werden die Ausführungszeiten von Queries gemessen, die an eine Datenbank mit geographischen Daten gerichtet sind. Die Daten liegen als Graph gespeichert vor. Relational liegen die Daten in zwei Relationen vor:

Nodes (ID, X, Y)
Edges (source→Nodes, target→ Nodes)

In der Nodes Relation wird jedem Knoten eine eindeutige ID zugeordnet. Außerdem sind dort X- und Y-Koordinaten eines Knotens gespeichert. In der Relation Edges sind die Kanten mit Start- und Ziel-Knoten gespeichert. Der Graph umfasst 10.000 Knoten und 13.399 Kanten. Er hat also einen Vernetzungsgrad von 2,68. Zur Messung wurde das Client/Server-System aus 5.4.1 verwendet.

Die nachfolgende Queries bestimmen für einen zufällig gewählten Startknoten die X- und Y-Koordinaten aller erreichbaren Knoten mit einem bestimmten Abstand zum Startknoten. Für jeden Abstand wurden auf jeder Datenbank zwei Queries ausgeführt: Eine, die die Kantenrichtung berücksichtigt und eine, die die Kantenrichtung ignoriert. Bei jeder Ausführung einer Query wird ein anderer Startpunkt x zufällig gewählt.

Die Queries für die gerichtete Variante lauten wie folgt:
Cypher 1

```
MATCH (n: Knoten {ID: x})-[:Kante]->(m: Knoten)
RETURN m.x, m.y
```

SQL 1

```
SELECT X, Y
FROM
  Edges n1
  JOIN Nodes on(ID = n1.target)
WHERE
 n1.SOURCE = x
```

Cypher 2

```
MATCH (n: Knoten {ID: x})-[:Kante*2..2]->(m: Knoten)
RETURN m.x, m.y
```

SQL 2

```
SELECT X, Y
FROM
  Edges n1
  JOIN Edges n2 on (n1.target = n2.source)
  JOIN Nodes on(ID = n2.target)
WHERE
 n1.SOURCE = x
```

Cypher 3

```
MATCH (n: Knoten {ID: x})-[:Kante*3..3]->(m: Knoten)
RETURN m.x, m.y
```

SQL 3

```
SELECT X, Y
FROM
  Edges n1
  JOIN Edges n2 on (n1.target = n2.source)
  JOIN Edges n3 on (n2.target = n3.source)
  JOIN Nodes on(ID = n3.target)
WHERE
 n1.SOURCE = x
```

Dies sind die Queries für die ungerichtete Variante:

Cypher 4

```
MATCH (n: Knoten {ID: x})-[:Kante]-(m: Knoten)
RETURN m.x, m.y
```

Cypher 5

```
MATCH (n: Knoten {ID: x})-[:Kante*2..2]-(m: Knoten)
RETURN m.x, m.y
```

Cypher 6

```
MATCH (n: Knoten {ID: x})-[:Kante*3..3]-(m: Knoten)
RETURN m.x, m.y
```

SQL 4-6
Die SQL-Queries lauten wie die Queries 1-3. Statt der Relation Edges nutzen die Queries jedoch eine Relation EdgesU, in der die Kanten in beide Richtungen gespeichert sind.

Bei den Cypher-Queries bezeichnet die Schreibweise

```
(n)-[:Kante*x..x]->(m)
```

eine Verbindung von zwei Nodes n und m mit genau x Relationships des Typs „Kante“.

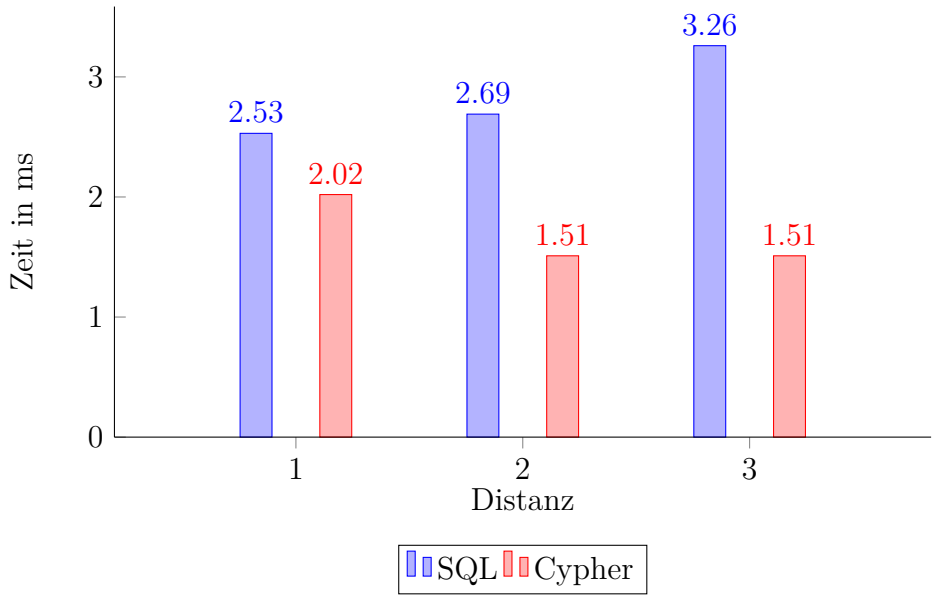

Abbildung 5.15: Ausführungszeiten gerichtet

Abbildung 5.15 zeigt die Ausführungszeiten der Queries, die die Richtung berücksichtigen. In Abbildung 5.16 sind die Ausführungszeiten für die ungerichtete Variante zu sehen. In allen Fällen ist Neo4j schneller als SQL. Neo4j benötigt für Query 2 und 3 weniger Zeit als für Query 1, obwohl zu erwarten ist, dass die benötigte Zeit mit zunehmender Distanz steigt, da auch der Rechenaufwand zunimmt. Allerdings sind die Ausführungszeiten von Neo4j so gering, dass diese Varianz durch Übertragung über

HTTPS verursacht werden könnte. Gleiches gilt für Query 4 bis 6. Dass SQL diesmal langsamer als Neo4j ist, könnte dadurch erklärt werden, dass diesmal nur einzelne große Relationen mit jeweils über 10.000 Einträgen existieren. Mit der Größe der Relationen wachsen auch die Kosten der Joins über diese. Die Kosten für Neo4j steigen für die Selektion zur Bestimmung des Startknotens, diese steigen höchstens linear in Abhängigkeit der Knotenanzahl des Graphens. Das Auflösen der beteiligten Relationships sollte ebenfalls höchstens lineare Kosten verursachen, da hierbei nur das Laden des Zielknotens von der Größe des Graphen abhängt. Dies könnte die kürzeren Ausführungszeiten von Neo4j erklären.

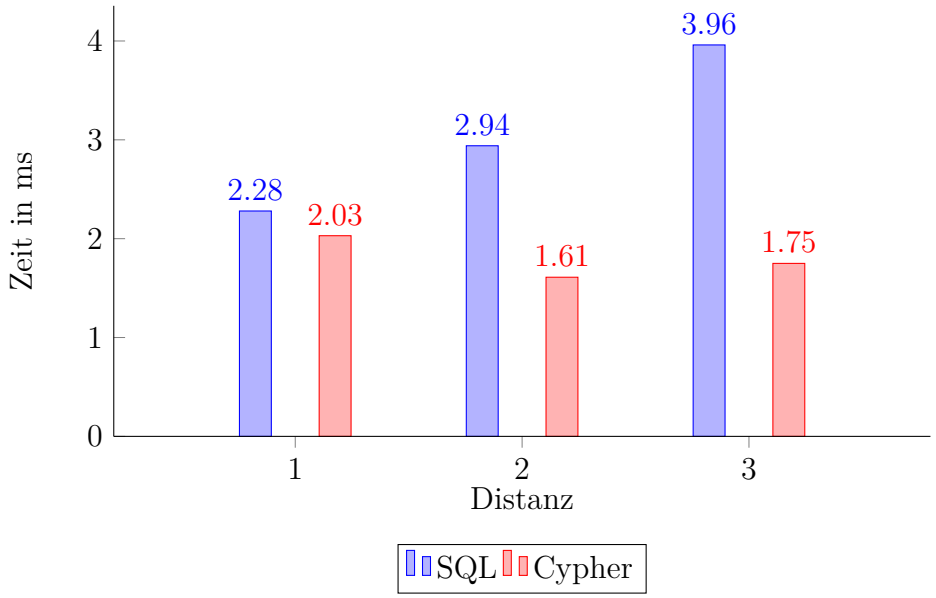

Abbildung 5.16: Ausführungszeiten ungerichtet

5.4.4 Zusätzliche Angebote

Im Folgenden werden Anfragen gestellt, die sich in Neo4j zwar innerhalb einer Query ausdrücken lassen, in SQL jedoch nicht.

Zum Überprüfen, ob eine Verbindung zwischen zwei Nodes besteht kann folgende Query verwendet werden:
Query 1

```
MATCH p=(n: Knoten {ID: x})-[:Kante*..]->(m: Knoten {ID: y})
RETURN p
```

Query 1 bestimmt einen Pfad zwischen einem Node mit der ID x und einem Node mit der ID y. In der Praxis kann diese Query jedoch nicht ausgeführt werden. Er führt dazu,

dass Neo4j eine Exception zurück gibt, da die Ausführung diese Queries zu aufwändig zu seien scheint.

Das Überprüfen, ob eine Verbindung zwischen zwei Nodes besteht, kann jedoch auch mit Query 2 umgesetzt werden:
Query 2

```
MATCH (n: Knoten {ID: x})), (m: Knoten {ID: y})),
       p=shortestPath(n-[:Kante*..]->m)
RETURN p
```

Diese Anfrage berechnet gleich den kürzesten Weg zwischen den beiden Nodes. Sind die Nodes nicht verbunden, ist die Ergebnismenge leer. Für die Testquery wurden die IDs der Nodes zufällig bestimmt.

Unter Neo4j benötigte die Anfrage eine durchschnittliche Ausführungszeit von 2,68 ms. Dies scheint auf eine sehr effiziente Ausführung eines shortestPath-Algorithmus zu deuten, da für diese Query nur unwesentlich mehr Zeit als für die Bestimmung der Nachbarn benötigt wurde. Dass auf der anderen Seite Query 1 nicht zu Ende ausgeführt werden kann, wirkt jedoch unverständlich. Besonders da Query 2 aufzeigt, wie effizient dies theoretisch möglich wäre.

Kapitel 6

Resümee

In diesem Kapitel werden die Ergebnisse dieser Arbeit zusammengefasst. Danach werden weiterführende Themen aufgezeigt.

6.1 Zusammenfassung

Neo4j bietet mit Cypher eine ausdrucksstarke Anfragesprache, mit der Graph-Traversierung komfortabel formuliert werden können. Eine Anfrageoptimierung war hingegen kaum vorhanden, da nicht einmal Selektionen im Ausführungsplan bevorzugt wurden.

Generell können relationale Datenbanken problemlos in Graphdatenbanken transformiert werden. Bestehende SQL-Queries können fast immer in semantisch äquivalente Cypher-Queries übersetzt werden. Eine Ausnahme bilden hier Queries mit Unteranfragen, da diese in Cypher nicht erlaubt sind. Für diese Queries müssen die Ergebnisse der Unteranfragen vorher mit Hilfe von with-Klauseln erzeugt werden.

Die Größe des Propertygraphen spielt in Neo4j erst eine entscheidende Rolle, wenn sie die Größe der Caches überschreitet. Dies ist eine gute Lösung, da das primäre Einsatzgebiet von Graphdatenbanken hochgradig verknüpfte Daten sind und nicht sehr große Datensätze.

Beim Funktionsumfang liegen bei Neo4j teilweise noch große Lücken vor. Insbesondere die fehlende Benutzerverwaltung ist ein großes Defizit.

In den Messungen war Oracle-SQL für die meisten Queries schneller als Neo4j. Nur in den Messungen mit den geographischen Daten konnte Neo4j bessere Ergebnisse erzielen. Dies liegt zum Großteil an der Zeit, die benötigt wird, um die Ergebnisse über HTTPS zu übertragen. Hier liegt ein großer Nachteil von Neo4j im dedizierten Betrieb.

Da Neo4j in Java implementiert ist, unterliegt es auch immer den Grenzen einer Java Virtual Machine, da diese beispielsweise durch den Heap-Size gewisse Randbedingungen vorgibt. Außerdem kommen Faktoren hinzu, die man nicht direkt steuern kann, wie zum Beispiel das Garbage-Collecting. Diese Eigenschaften sind für eine Datenbank, die eine sehr hohe Performance erreichen will, eher hinderlich.

Die Kombination dieser Faktoren erweckt den Eindruck, dass Neo4j eher für den Embedded-Betrieb innerhalb einer Java-Applikation vorgesehen ist. In dieser würden die Benutzerverwaltung und die Übertragung der Daten über HTTPS wegfallen, da man hierfür einen direkten Zugriff auf die Neo4j Java-API nutzt. Da Java-Applikationen sowieso in einer JVM laufen, kann Neo4j diese gleich mit benutzen.

Auf der anderen Seite konnte Neo4j die Ergebnisse bei den Anfragen mit geographischen Daten schneller als SQL bestimmen. Auch können in Cypher graphspezifische Anfragen gestellt werden, die sich nicht innerhalb einer SQL-Query ausdrücken lassen. Ein Beispiel hierfür ist die Ermittlung eines kürzesten Weges.

Daher ist es sinnvoll Neo4j vor allem für Anwendungen einzusetzen, bei denen Daten in Graph-Form vorliegen und viele Knoten gleicher Art existieren, wie zum Beispiel bei der Datenbank mit geographischen Daten. Um die Vorteile der Datenstruktur von Neo4j nutzen zu können, sollten auf dem Graphen auch graphtypische Operationen erfolgen. Das könnte zum Beispiel das Ermitteln eines kürzesten Weges sein. Diese Operation wird zwar von Neo4j gut implementiert, ergibt jedoch nicht auf jedem Datensatz einen Sinn. So hätte es beispielsweise keinen Nutzen, einen kürzesten Weg in der Modulkatalog-Datenbank zu ermitteln.

Weiterhin müssen auch die restlichen Anforderungen an die Anwendung zu den äußeren Bedingungen passen, die Neo4j vorgibt. Dazu zählen insbesondere Schemalosigkeit, fehlende Benutzerverwaltung und die Anfrageschnittstelle. Besonders eignet sich Neo4j für Java-Anwendungen, in denen ein Embedded-Betrieb möglich ist.

Zusammenfassend lässt sich sagen, dass zwar für viele Anwendungsfälle die Verwendung einer anderen Datenbank sinnvoller ist, Neo4j jedoch unter bestimmten Umständen eine interessante Datenbank-Alternative sein kann.

6.2 Ausblick

In dieser Arbeit wurde Neo4j ausschließlich dediziert betrieben, da es mit einem ebenfalls dedizierten Oracle-SQL-Server verglichen werden sollte. Ein anderer Ansatz könnte darin liegen, Messungen mit Neo4j im Embedded-Betrieb durchzuführen.

Des Weiteren könnte Neo4j nicht nur mit relationalen Datenbanken, sondern auch mit

anderen Graphdatenbanken wie zum Beispiel „InfoGrid“ oder „HyperGraphDB“ verglichen werden. Dies würde einen Eindruck vermitteln, wie performant sich Graphdatenbanken überhaupt verhalten können.

Da unterschiedliche Daten in unterschiedliche Graph-Strukturen resultieren, könnten auch noch die Daten und die dazugehörigen Queries variiert werden, um weitere Anwendungsfälle von Neo4j beurteilen zu können.

In dieser Arbeit wurde mit der Community-Edition gearbeitet. Ähnliche Betrachtungen könnten auch für die Enterprise-Edition unternommen werden. Diese verfügt über einen High-Perfomance-Cache und könnte sich daher zeitlich anders verhalten. Auch die High-Availability-Komponente, also das Betreiben einer verteilten Graphdatenbank könnte untersucht werden.

Schließlich hat die Entwicklung von Neo4j zur Zeit noch relativ große Auswirkungen auf sein Verhalten. Daher könnte es sich lohnen, auch das Verhalten neuer Versionen von Neo4j zu untersuchen.

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[CRE] Neo4j Cypher Refcard 2.1. `http://docs.neo4j.org/refcard/2.1/`. Aufgerufen: 22.06.2014.

[EFH+11] S. Edlich, A. Friedland, J. Hampe, B. Brauer, M. Brückner. *NoSQL - Einstieg in die Welt nicht relationaler Web 2.0 Datenbanken.* Carl Hanser Fachbuchverlag, 2011.

[Fie00] R. T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures.* Dissertation, University Of California, Irvine, 2000.

[Gra] Graphviz offiziele Homepage. `http://www.graphviz.org/`. Aufgerufen: 24.05.2014.

[HD13] H. Huang, Z. Dong. Research on architecture and query performance based on distributed graph database Neo4j. *3rd International Conference on Consumer Electronics, Communications and Networks (CECNet)*, 2013.

[HP13] F. Holzschuher, R. Peinl. Perfomance of Graph Query Languages. *Proceedings of the Joint EDBT/ICDT 2013 Workshops*, 2013.

[Hun14] M. Hunger. *Neo4j 2.0 - Eine Graphdatenbank für alle.* entwickler.press, 2014.

[Lin12] T. Lindaaker. An overview of Neo4j Internals. 2012. Aufgerufen: 16.06.2014, URL `http://de.slideshare.net/thobe/an-overview-of-neo4j-internals`.

[N4S] Neo4j Subscriptions. `http://neo4j.com/subscriptions/?_ga=1.108671554.137793134.1395049867`. Aufgerufen: 22.07.2014.

[N4W] Neo4j offizile Website. `http://neo4j.org`. Aufgerufen: 22.07.2014.

[NCS] Neo4j Contributed Software. `http://neo4j.com/contrib/`. Aufgerufen: 22.07.2014.

[Neo14] Neo Technology. *The Neo4j Manual v2.1.0-M02*, 2014. URL `http://docs.neo4j.org/chunked/milestone/`.

[NOR] NoSQL-database.org. http://www.nosql-database.org. Aufgerufen: 04.08.2014.

[NRN] Neo4j Release Notes. https://github.com/neo4j/neo4j/blob/master/packaging/standalone/standalone-community/src/main/distribution/text/community/CHANGES.txt. Aufgerufen: 22.06.2014.

[Ora14] Oracle Cooperation. *Oracle Database Reference12c Release 1*, 2014. URL http://docs.oracle.com/cd/E16655_01/server.121/e17615/toc.htm.

[PMS13] D. M. Peter Macko, M. Seltzer. Perfomance Introspection of Graph Databases. *Proceedings of the 6th International Systems and Storage Conference*, 2013.

[RWE13] I. Robinson, J. Webber, E. Eifrem. *Graph Databases*. O'Reilly, 2013.

[TPG] Tinkerpop Property Graph Model. https://github.com/tinkerpop/blueprints/wiki/Property-Graph-Model. Aufgerufen: 22.06.2014.

Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit und die zugehörige Implementierung selbstständig verfasst und dabei nur die angegebenen Quellen und Hilfsmittel verwendet habe.

Hannover, 07. August 2014

Nicolas Tempelmeier