Heimo Bischofter

Vergleich der Leistungsfähigkeit von Graphen-Datenbanken für Informationsvernetzung anhand der Abbildbarkeit von Berechtigungskonzepten

Masterarbeit

zur Erlangung des akademischen Grades
Diplom-Ingenieur

eingereicht an der
Technischen Universität Graz

Betreuer
Dipl.-Ing. Dr.techn. Roman Kern

Mitbetreuer
Dipl.-Ing. Heimo Gursch und Dipl.-Ing. Markus Zoier

Institut für Wissenstechnologien

Graz, Mai 2016

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Graz, _____ _____

Date Signature

Eidesstattliche Erklärung[1]

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Graz, am _____ _____

Datum Unterschrift

[1] Beschluss der Curricula-Kommission für Bachelor-, Master- und Diplomstudien vom 10.11.2008; Genehmigung des Senates am 1.12.2008

Danksagung

Diese Arbeit entstand am Kompetenzzentrum VIRTUAL VEHICLE in Graz, Österreich. Das Kompetenzzentrum VIRTUAL VEHICLE wird im Rahmen von COMET – Competence Centers for Excellent Technologies durch das Österreichische Bundesministerium für Verkehr und Technologie (BMVIT), das Österreichische Bundesministerium für Wissenschaft, Forschung und Wirtschaft, (BMWFW), die Österreichische Forschungsförderungsgesellschaft mbH (FFG), das Land Steiermark sowie die Steirische Wirtschaftsförderung (SFG) gefördert. Das Programm COMET wird durch die FFG abgewickelt.

An dieser Stelle möchte ich mich herzlich bei all denjenigen bedanken, die mich während der Anfertigung meiner Masterarbeit unterstützt haben.

Mein Dank gebührt den Herrn Dipl.-Ing. Dr. techn. Roman Kern, Dipl.-Ing. Heimo Gursch und Dipl.-Ing. Markus Zoier für die Betreuung meiner Masterarbeit. Für die Unterstützung, die konstruktiven Kritiken und nützlichen Anregungen während der Erstellung der Arbeit möchte ich mich herzlich bedanken.

Bedanken möchte ich mich auch bei meinen Freunden, die mir während meiner gesamten Studienzeit in allen Belangen unterstützend zur Seite gestanden sind und immer ein offenes Ohr für mich hatten.

Schlussendlich möchte ich mich bei meiner Familie, speziell bei meinen Eltern Ernst und Elviera und meinen Geschwistern Gerd und Thomas bedanken, die mich nicht nur während der Masterarbeit, sondern während meines gesamten Studiums, ermutigt und unterstützt haben. Besonderer Dank gilt jedoch meinen Eltern, die durch ihre uneingeschränkte Unterstützung ein Studium erst ermöglicht haben. Danke!

Kurzfassung

Vernetzte Daten und Strukturen erfahren ein wachsendes Interesse und verdrängen bewährte Methoden der Datenhaltung in den Hintergrund. Einen neuen Ansatz für die Herausforderungen, die das Management von ausgeprägten und stark vernetzten Datenmengen mit sich bringen, liefern Graphdatenbanken. In der vorliegenden Masterarbeit wird die Leistungsfähigkeit von Graphdatenbanken gegenüber der etablierten relationalen Datenbank evaluiert. Die Ermittlung der Leistungsfähigkeit erfolgt durch Benchmarktests hinsichtlich der Verarbeitung von hochgradig vernetzten Daten, unter der Berücksichtigung eines umgesetzten feingranularen Berechtigungskonzepts.

Im Rahmen der theoretischen Ausarbeitung wird zuerst auf die Grundlagen von Datenbanken und der Graphentheorie eingegangen. Diese liefern die Basis für die Bewertung des Funktionsumfangs und der Funktionalität der zur Evaluierung ausgewählten Graphdatenbanken. Die beschriebenen Berechtigungskonzepte liefern einen Überblick unterschiedlicher Zugriffskonzepte sowie die Umsetzung von Zugriffskontrollen in den Graphdatenbanken. Anhand der gewonnenen Informationen wird ein Java-Framework umgesetzt, welches es ermöglicht, die Graphdatenbanken, als auch die relationale Datenbank unter der Berücksichtigung des umgesetzten feingranularen Berechtigungskonzepts zu testen. Durch die Ausführung von geeigneten Testläufen kann die Leistungsfähigkeit in Bezug auf Schreib- und Lesevorgänge ermittelt werden. Benchmarktests für den schreibenden Zugriff erfolgen für Datenbestände unterschiedlicher Größe. Einzelne definierte Suchanfragen für die unterschiedlichen Größen an Daten erlauben die Ermittlung der Leseperformance.

Es hat sich gezeigt, dass die relationale Datenbank beim Schreiben der Daten besser skaliert als die Graphdatenbanken. Das Erzeugen von Knoten

und Kanten ist in Graphdatenbanken aufwendiger, als die Erzeugung eines neuen Tabelleneintrags in der relationalen Datenbank. Die Bewertung der Suchanfragen unter der Berücksichtigung des umgesetzten Zugriffkonzepts hat gezeigt, dass Graphdatenbanken bei ausgeprägten und stark vernetzten Datenmengen bedeutend besser skalieren als die relationale Datenbank. Je ausgeprägter der Vernetzungsgrad der Daten, desto mehr wird die `JOIN`-Problematik der relationalen Datenbank verdeutlicht.

Abstract

There is a growing interest on complex connected data, which replaces proven methods of data storage. Graph databases are a new approach, which handles the challenges of the data management of connected data. This thesis evaluates the performance of chosen graph databases and compares them with the well-established relational database model. The performance of the databases is determined by benchmarktests, which evaluate the performance concerning the processing of connected data in consideration of a fine grained authorization concept.

The theoretical scope of the thesis discusses the fundamentals of both graphdatabases and graph theories. These basics provide the foundation for the evaluation of features and functionality of the chosen graph databases. The chapter focusing on the authorization concept offers a short overview on different concepts and implemented authorization functions within graph databases. With the gained theoretical information a Java framework was implemented, which allows the testing of the graph databases and the relational database with the implemented fine grained authorization concept. The execution of test cases allows the evaluation of the databases concerning reading and writing performance. Data imports with different size determine the write performance and defined search queries for each database size evaluate the read performance.

It has been shown, that - concerning writing operations - the relational database scales better than the graph databases. The creation of nodes and edges in the graph databases causes more effort than the insertion of a new table entry in the relational model. The evaluation concerning reading operations in consideration of fine grained authorization concept has shown, that the performance of graph databases concerning the management of connected data is higher compared to relational databases. For each connection a `JOIN` in the relational database is needed. The more connections in

the structure, the more JOIN operations are needed, which comes at the cost of performance for the relational database.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1. Einleitung

Es existieren viele unterschiedliche Daten, die sich in Form eines Graphen darstellen und speichern lassen. Sei es die Struktur eines sozialen Netzwerkes, Verkehrsnetze und -wege oder Daten und deren Abhängigkeiten zu weiteren Informationen. Diese Daten sind in großen Mengen vorhanden und untereinander stark vernetzt. Zur effizienten Verarbeitung der Daten müssen diese in geeigneten Datenbanksystemen gespeichert werden.

Durch das wachsende Interesse an komplexen und datenintensiven Anwendungen werden bewährte Methoden der Datenhaltung immer mehr in den Hintergrund gedrängt (Edlich u. a., 2010). Relationale Datenbanksysteme, der seit den siebziger Jahren de facto-Standard, stoßen bei hochgradig vernetzten Daten an ihre Grenzen. Graphdatenbankmanagementsysteme (GDMBS) gelten als Alternative zu relationalen Datenbanken, um die Herausforderungen bezüglich hochgradig vernetzten Daten zu meistern. Graphdatenbanken sind keine vollkommen neue Technologie. Jedoch gewinnen diese seit einigen Jahren immer mehr an Bedeutung (Vicknair u. a., 2010) (Edlich u. a., 2010). Essentieller Bestandteil der Graphdatenbanken sind die Beziehungen, die neben Knoten und knoten- bzw. kantenbezogenen Eigenschaften (zur Speicherung von Informationen) das Grundgerüst zur Abbildung von Daten in einem Graphen bilden (Robinson, Webber und Eifrem, 2013). Abgesehen von der effizienten Verarbeitung und Speicherung von Daten spielen Berechtigungen, die eine uneingeschränkte Nutzung von Ressourcen regeln, eine bedeutende Rolle in der Informationstechnik. Feingranulare Berechtigungskonzepte spielen jedoch in GDBMS eine untergeordnete Rolle. Sahafizadeh und Nematbakhsh (2015) bezeichnen eine feingranulare Zugriffskontrolle als eine der Herausforderungen in Graphdatenbanken und NoSQL-Lösungen.

Im Rahmen dieser Arbeit soll dargelegt werden, wie ein feingranulares Berechtigungskonzept in Graphdatenbanken umgesetzt werden kann, um

anschließend die weiterführende Forschungsfrage hinsichtlich der Leistungsfähigkeit von Graphdatenbanken im Vergleich zu relationalen Datenbanken zu beantworten.

In diesem Zusammenhang soll zunächst ein theoretischer Überblick der Grundlagen von Datenbanken und Berechtigungskonzepten - mit besonderem Augenmerk auf Graphdatenbanken - gelegt werden. Ausgewählte Graphdatenbanken und die relationale Datenbank MySQL sollen hinsichtlich ihres Aufbaus und Funktionalität gegenübergestellt werden. Neben der theoretischen Untersuchung sollen die Systeme in einem Benchmarktest in Bezug auf ihre Leistungsfähigkeit gegenüber gestellt werden. Anhand der gewonnen theoretischen Kenntnisse wird ein Berechtigungskonzept entwickelt, welches bei der Implementierung des Java-Frameworks umgesetzt wird. Das Java-Framework liefert die Basis für die Bewertung der Leistungsfähigkeit der einzelnen Datenbanken unter der Verwendung des feingranularen Berechtigungskonzepts. Definierte Testalgorithmen führen Operationen auf dem Datenbestand durch und liefern Messwerte zur Bewertung der Verarbeitungsleistung der Datenbanken.

Zu Beginn der Arbeit werden im Kapitel 2 allgemeine Grundlagen zu Datenbanksystemen und notwendige Graphentheorien (zum leichteren Verständnis von Graphdatenbanken) sowie die exemplarisch ausgewählten Graphdatenbanksysteme erörtert. Anschließend werden im Abschnitt 3 Berechtigungskonzepte sowie die Umsetzung von Zugriffskontrollen in den ausgewählten Datenbanktechnologien beschrieben. Im Anschluss erfolgt in Kapitel 4 die Beschreibung der Umsetzung des Java-Frameworks. Es beschreibt den zur Evaluierung ausgewählten Testdatensatz sowie die Realisierung des feingranularen Berechtigungskonzepts, den Aufbau der Software sowie die Umsetzung der Implementierung. Kapitel 5 enthält die Evaluierung der Leistungsfähigkeit anhand der durch das Java-Framework ermittelten Messwerte. Die gewonnen Kenntnisse werden im Abschnitt 6 reflektiert und daraus ein Fazit gezogen.

2. Grundlagen

Graphdatenbanken sind keine vollkommen neue Technologie, jedoch gewinnt diese seit einigen Jahren immer mehr an Bedeutung. Durch den laufenden technischen Fortschritt und dem wachsenden Interesse an komplexen und datenintensiven Anwendungen werden bewährte Methoden der Datenhaltung immer mehr in den Hintergrund gedrängt. Diese Anwendungen zeichnen sich durch stark ausgeprägte vernetzte Daten aus. Graphdatenbanken bieten einen effizienten Umgang für die Speicherung und Abfrage von hochgradig vernetzten Daten. (Edlich u. a., 2010)

Das nachfolgende Kapitel 2.1 beschreibt kurz die allgemeinen Anforderungen an Datenbanken und an Datenbankmanagementsysteme. Kapitel 2.2 liefert einen kurzen Überblick über die Funktionsweise und Intention von relationalen Datenbanken. Die Kapitel 2.3 und 2.4 erläutern kurz und prägnant den Entstehungsprozess von Graphdatenbanken und dienen zur Auffrischung der Grundkenntnisse im Umgang mit Graphen. Anschließend liefert das Kapitel 2.5 eine Übersicht über zwei ausgewählte Produkte von sich am Markt befindlichen Graphdatenbanken. Kapitel 2.6 und 2.7 beschreibt RDF Stores und die relationale Datenbank MySQL, um einen Vergleich zu Graphdatenbanken herstellen zu können.

2.1. Datenbanken und Data Base Management System

Datenbanken dienen der Verwaltung und Speicherung von Daten und Informationen. Die Datenverwaltung muss drei grundlegende Operationen bereitstellen:

- die Eingabe von neuen Daten
- das Löschen von veralteten Daten
- das Verändern und Anpassen von bereits vorhandenen Daten

Für eine sichere und flexible Datenorganisation unterliegt eine Datenbank bestimmten Anforderungen. Einerseits muss dem Benutzer Zugriff auf die Daten ermöglicht werden, andererseits muss verhindert werden, dass Benutzer ohne ausreichender Berechtigung Daten manipulieren. Ein wichtiges Kriterium ist die Reorganisation der Datenbankstruktur. Es soll möglich sein, die Struktur der Datenbank zu ändern bzw. anzupassen, ohne angebundene Applikationen an die Strukturänderung anpassen zu müssen. (Steiner, 2009)

Eine ideale Datenbank wird laut Steiner (2009) wie folgt beschrieben: die Struktur der Datenbank soll redundante Daten verhindern. Eine Reorganisation des Datenbanksystems darf sich auf die Applikationen nicht auswirken und die Datenbank soll ein gewissen Maß an Flexibilität bieten. Die beschriebenen Eigenschaften bilden einen nicht unwesentlichen Bestandteil einer Datenbank. Die Integrität der Daten in der Datenbank hat oberste Priorität. Sie ist laut Steiner (2009) gegeben, wenn Datenkonsistenz, -sicherheit und schutz eingehalten werden:

"Datenintegrität ist dann gegeben, wenn ein Datenbanksystem so funktioniert, dass keine widersprüchlichen Daten entstehen können, Daten nicht verloren gehen und der Datenzugriff geregelt ist."

Der Begriff Datenkonsistenz bezeichnet die Widerspruchsfreiheit der Daten. Die Datensicherheit soll den Datenbestand vor Verlust und Beschädigung schützen. Der Datenschutz beschreibt die Verhinderung von Datenmissbrauch. Dadurch sollen vertrauliche Daten vor Zugriffen von unbefugten Personen geschützt werden.

Im Zuge von Verlässlichkeit und Integrität einer Datenbank wird in der Literatur von Transaktionsmodell **ACID** gesprochen. Eine Transaktion ist eine Einheit von Datenbankanweisungen, die zur Gänze - oder gar nicht - ausgeführt werden. Das heißt, alle in der Einheit zusammengefassten Anweisungen werden erfolgreich beendet oder bei fehlerhaften Verlauf der

Transaktion alle bisher ausgeführten Anweisungen rückgängig gemacht und der Ausgangszustand vor der Transaktion wieder hergestellt. Die Charakterisierung von Transaktionen lässt sich mit dem **ACID**-Akronym zusammenfassen. (Stuber, 2012)

`ATOMICITY`	alle Änderungen einer Transaktion sind unteilbar (atomar), und werden ganz oder gar nicht durchgeführt.
`CONSISTENCY`	die Datenbank ist vor und nach der Transaktion in einem konsistenten und korrekten Zustand.
`ISOLATION`	eine Transaktion wird isoliert ausgeführt, d.h., andere simultan ausgeführte Transaktionen haben keine Auswirkung auf sie.
`DURABILITY`	Änderungen einer erfolgreich durchgeführten Transaktion sind dauerhaft.

Das *Data Base Management System*, kurz DBMS, ist die eigentliche Komponente um die beschriebenen Anforderungen an eine Datenbank zu erfüllen. Das DBMS ist der einzige Prozess, der direkt auf die Daten zugreifen kann und bildet den wesentlichen Bestandteil der Datenverwaltung. Die Manipulation der Daten, sprich "speichern, löschen und ändern" erfolgt nur über das DBMS. Die Verwaltung der Zugriffsrechte sowie die Koordination von mehreren Zugriffen wird ebenso über das Verwaltungssystem abgewickelt, wie auch die Gewährleistung der Datenkonsistenz. Das DBMS stellt zudem Funktionalitäten zur Datensicherung und Datenwiederherstellung zur Verfügung, um nach technischen oder manuellen Fehlern, welche beschädigte Daten nach sich ziehen, Datenverluste und inkonsistente Daten zu verhindern. (Unterstein und Matthiessen, 2012)
Abbildung 2.1 zeigt den vereinfachten Aufbau eines Verwaltungssystems als Schnittstelle zwischen Applikation und Datenbank.

2.2. Relationale Datenbanken

Relationale Datenbanken sind die weit verbreitetsten Datenbanken, welche auf dem von Codd (1970) vorgeschlagenen relationalen Datenbankmodell basieren.

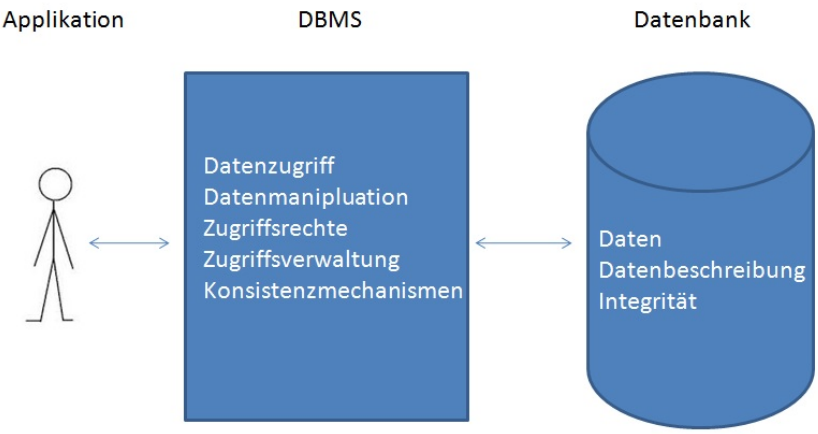


Abbildung 2.1.: DBMS und seine grundlegenden Funktionen als Schnittstelle zwischen Applikation und Datenbank (Quelle: nach Unterstein und Matthiessen, 2012)

Die Grundlage des von Codd (1970) entwickelten Datenbankmodells ist die Relation, welche die Beschreibung einer Tabelle und ihre Beziehung zu anderen Tabellen darstellt. Eine relationale Datenbank besteht aus einer großen Anzahl von Tabellen, in welchen die Informationen entsprechend dem Inhalt gespeichert werden. Diese Tabellen werden in der Fachsprache Relation oder Entität genannt. Objekte, die in einer Relation gespeichert werden, sind durch Attribute beschrieben. Ein "Objekt" Person kann somit beispielsweise, wie in Tabelle 2.1 ersichtlich, durch die Attribute *ID, Vorname, Nachname und Geburtsdatum* beschrieben werden. Ein Datensatz entspricht einer Menge von Attributen und wird auch Tuple genannt. Vereinfacht ausgedrückt entsprechen die Spalten einer Tabelle den Attributen und eine Zeile einem Datensatz.

Eine relationale Datenbank ist somit durch eine Menge von Tabellen gegeben, die zueinander in Beziehung stehen. Die Beziehungen unter den einzelnen Tabellen werden durch Schlüssel hergestellt. Dabei unterscheidet man zwischen Primär- und Fremdschlüssel. Der Primärschlüssel wird durch eine Menge von Attributen festgelegt und identifiziert einen Datensatz in der Tabelle eindeutig (Entitätsregel). Er darf somit in der Tabelle nur einmal auftreten. Der Fremdschlüssel hingegen beschreibt die Verknüpfung verschiedener Tabellen. Ein Fremdschlüssel besteht wie der Primärschlüssel

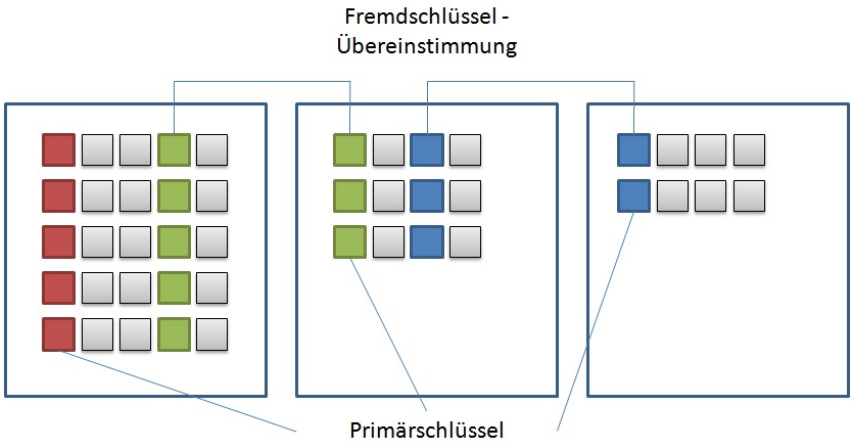


Abbildung 2.2.: Darstellung der Schlüsselübereinstimmung von Primär- und Fremdschlüssel in einer relationalen Datenbank (Quelle: nach cbSolution.net (2011))

aus einem Attribut oder einer Kombination von Attributen und verweist auf den Primärschlüssel einer anderen Tabelle. Abbildung 2.2 veranschaulicht die Verknüpfung des Fremdschlüssel auf dem Primärschlüssel einer Tabelle.

Entstehen Verknüpfungen über Fremdschlüssel, wird referentielle Integrität gefordert. Diese soll Datenkonsistenz und -integrität sicherstellen. Dadurch dürfen Daten ohne eindeutigen Primärschlüssel nicht eingefügt und Datensätze mit Referenzen nicht ohne weiteres gelöscht werden. Das Löschen von Datensätzen könnte zu offenen Referenzen und somit zu inkonsistenten Daten führen.[1] (Unterstein und Matthiessen, 2012)

Für die Definition und Bearbeitung von Datenstrukturen und Daten, sowie für die Abfrage von vorhandenen Datenbeständen steht die von ISO standardisierte Abfragesprache SQL (Structured Query Language) zur Verfügung. Hersteller von relationalen Datenbankmanagementsystemen orientieren sich an diesem Standard und unterstützten SQL. Der "Dialekt" und der Funktionsumfang variieren jedoch von Hersteller zu Hersteller. Piepmeyer (2011) unterteilt Anweisungen in SQL in drei Hauptkategorien ein:

[1] `http://www.datenbanken-verstehen.de/`, aufgerufen am 2015-07-16

Person

ID	Vorname	Nachname	Geburtsdatum
1	Bart	Simpson	01.04.1979
2	Milhouse	van Houten	01.07.1979
3	Nelson	Muntz	-
4	Lisa	Simpson	02.08.1985

Tabelle 2.1.: Beispiel einer Relation "Person" in welcher Entitäten über die Werte ID, Vor- und Nachname sowie Geburtsdatum beschrieben werden

Data Definition Language (DDL) sie umfasst mit *create, drop* und *alter* eingeleitete SQL-Anweisung. Also jene Anweisungen, die die Definition und Manipulation von Datenbankschemata betreffen.

Data Control Language (DCL) dient zur Rechteverwaltung von Benutzern und zur Transaktionskontrolle.

Data Manipulation Language (DML) umfasst die SQL-Anweisungen *update, delete, insert* und *select*. DML dient zur Datenmanipulation und Datenabfrage.

SQL ist eine deklarative Abfragesprache, das heißt, es steht nicht die Frage nach dem "wie?", sondern die Frage nach dem "was?" im Vordergrund. Die Abfragen werden so formuliert um die benötigten Daten zu erhalten. Die Frage nach dem "wie" wird vom Datenbankmanagementsystem erledigt. Das Ergebnis einer SQL-Abfrage ist ähnlich einer Tabelle und kann auch als solche in weiteren verschachtelten Abfragen weiter benutzt werden.

Relationale Datenbanken finden vor allem Einsatz, wenn es darum geht, große Mengen an Informationen nach einer bestimmten Struktur zu speichern. In eingeschränktem Maße ist die relationale Datenbank flexibel, da die Datenstruktur abgeändert und erweitert werden kann, jedoch auf Kosten der Übersichtlichkeit. Nimmt die Komplexität der Daten jedoch zu, sprich die Daten werden unstrukturierter und die Beziehungen unter den Daten werden vernetzter, sind diese schwierig in einer relationalen Datenbank abzubilden. Erfolgt die Datenhaltung in normalisierter Form, führen die Beziehungen unter den Relationen in der Datenbank bei Abfragen zu JOIN-Operationen, welche die Datenbank ressourcen- und leistungstechnisch an ihre Grenzen bringt. Durch JOIN-Operationen können mehrere Tabellen

über Schlüssel- und Fremdschlüsselfelder zu neuen Relationen zusammengefügt werden. Verschiedene Tabellen können somit zusammengefasst und abgefragt werden. Unter der normalisierten Form der Datenhaltung bzw. der Normalisierung von Daten versteht man das redundanzfreie Speichern von Daten innerhalb der Tabellen der Datenbasis, unter der entsprechenden Zuweisung der Attribute zu den einzelnen Tabellen. Eine redundanzfreie Datenspeicherung ist dann gegeben, wenn kein Teil des Datenbestandes weggelassen werden kann, ohne dass dies zu Informationsverlusten führt (Steiner, 2009). Die Dritte Normalform wird als ausreichend empfunden. Sie liefert ein ausreichendes Maß an Redundanz, Performance und Flexibilität für eine Datenbank.[2]

Insgesamt werden im Allgemeinen folgende Vorteile angeführt:

- standardisierte Zugriffssprache
- bewährte Algorithmen unter der Verwendung von SQL
- Daten bleiben unabhängig voneinander
- relativ einfache Modellierung
- Datenkonsistenz kann durch Entitäts- und referentielle Integrität gesichert werden (Kubacki, 2010)

Den Vorteilen stehen nachfolgende Punkte gegenüber:

- Performance und Ressourcen bei komplexen Daten
- Information über mehrere Tabellen verteilt
- unübersichtliche Datenstruktur
- festes Schema muss vorliegen
- während Schema-Änderungen Datenbank häufig nicht nutzbar (Kubacki, 2010)

Kurz zusammengefasst besteht eine relationale Datenbank aus mindestens einer Tabelle, die durch ein oder mehrere Attribute beschrieben wird. Die einzelnen Tabellen können untereinander in Beziehung stehen. Sie eignet sich besonders für die Speicherung von strukturierten Daten. Ausgewählte Datenbanksysteme, speziell für unstrukturierte, dicht vernetzte und flexible Daten werden, nach einer kurzen Erläuterung zu Entstehung von

[2] `http://datenbanken-verstehen.de/datenbanken/datenmodellierung/normalisierung`, aufgerufen am 2015-01-08

Graphen und einem fokussierten Überblick zur Graphentheorie im Kapitel 2.5 beschrieben.

2.3. Entstehung von Graphdatenbanken

Graphdatenbanken gehören zu der neuen Generation von NoSQL Datenbanken, welche Herausforderungen hinsichtlich umfangreichem Informationsvolumen und hohem Vernetzungsgrad effizient bewältigen. Der Begriff NoSQL (not only SQL) beschreibt im wesentlichen vier Kategorien von Datenbanken. Dokumenten-, Key-Value-, spaltenorientierte- und Graphdatenbanken. (MarkLogic, 2014)

NoSQL, speziell Graphdatenbanken, erfahren aktuell eine große Popularität. Edlich u. a. (2010) bescheinigen die Verwendung von Graphdatenbanken bereits in den 80er und 90er Jahren. Der Verwendungszweck beschränkte sich dabei hauptsächlich auf die Modellierung und Verwaltung von Netzen. Der Durchbruch gelang jedoch zur Jahrtausendwende aufgrund umfangreicher Forschung im Bereich von Semantic Web, Wissensrepräsentation, Geoinformationssystemen und weiteren Anwendungen mit stark ausgeprägten, semi-strukturierten und vernetzten Daten. Die Verbreitung der Technologie wird unter anderem durch das auf Skalierung ausgerichtete Konzept von Graphdatenbanken bezüglich großen Datenmengen und die schwierige Darstellung vernetzter Datenstrukturen in relationalen Datenbanken vorangetrieben. (Edlich u. a., 2010) (Stuber, 2012)

2.4. Graphen

In diesem Kapitel wird ein fokussierter Blick auf die notwendige Graphentheorie für Graphdatenbanken gegeben, wodurch das Verständnis bezüglich Graphdatenbanken erleichtert werden soll. Einen ausführlichen Einblick in

die Welt der Graphen liefert das Buch "Graphentheorie"[3] oder die Ausarbeitung des Institut für Mathematik an der Universität Würzburg [4]

Ein Graph ist eine Struktur die durch eine Menge von Objekten und deren Verbindungen zwischen den Objekten gebildet wird. Objekte werden als Knoten und Verbindungen als Kanten bezeichnet. Ein Graph lässt sich in der Theorie als Paar $G = (V, E)$ darstellen, wobei $V \neq \emptyset$ einer Menge von Knoten und E einer Menge von Kanten entspricht. Eine Kante $e = \{v_1, v_2\}$ aus der Menge E, besteht aus zwei Teilmengen von Knoten $v_1, v_2 \in V$. Dabei wird v_1 als Anfangs- und v_2 als Endknoten von e bezeichnet. Von benachbarten oder *adjazenten* Knoten $v_1, v_2 \in V$ spricht man, wenn es eine Kante $e \in E$ gibt, welche die beiden Knoten v_1, v_2 verbindet. (Schwartz, 2013)
Der Aufbau der Graphenstruktur lässt sich nach der Art der Verknüpfung kategorisieren. Dabei unterscheidet man zwischen folgenden Kategorien:

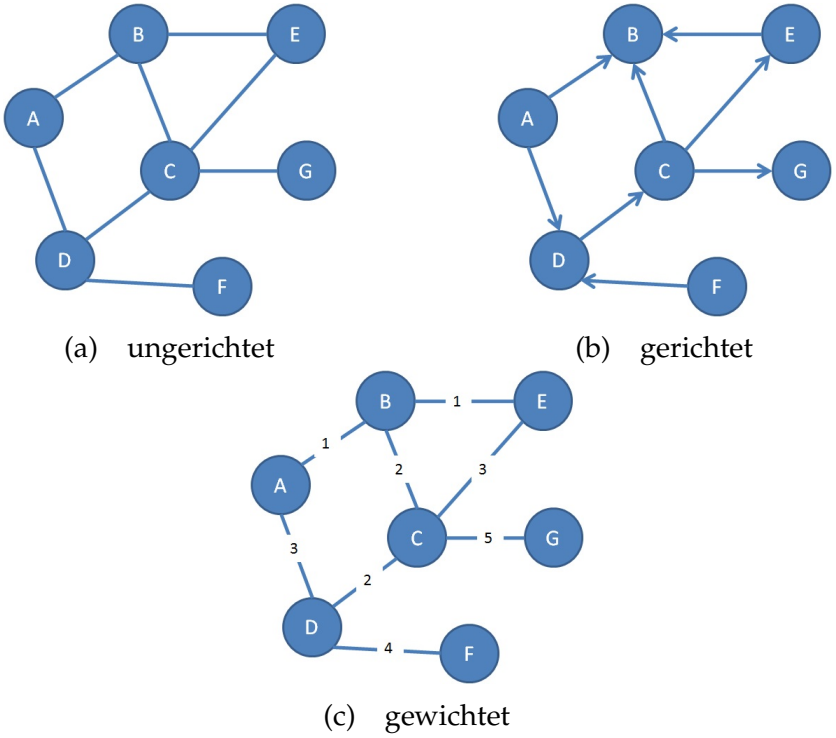
Ungerichtete Graphen Einfache Graphenstruktur, die Knoten beschreibt, welche durch Kanten verbunden sind. Die Kanten sind ungerichtet und verbindet zwei Knoten in beide Richtungen. Die Kanten (v_1, v_2) und (v_2, v_1) beschreiben ein und die selbe Kante, welche die Knoten v_1 und v_2 verbindet. Die Kanten sind dadurch in beide Richtungen "traversierbar[5]". In der Abbildung 2.3a für einen ungerichteten Graphen, gibt es demnach beispielsweise eine Pfad von $< A, B, E >$. Durch ungerichtete Kanten ist die entgegengesetzte Richtung ebenso möglich.

Gerichtete Graphen Im Gegensatz zu den ungerichteten Graphen wird die Verbindung zweier Knoten nur in eine Richtung beschrieben. Wie in Abbildung 2.3b ersichtlich, gibt es dadurch beispielsweise einen Pfad $< A, D, C, G >$. Eine Traversierung der entgegengesetzten Richtung ist jedoch nicht möglich.

[3] `https://www.springer.com/us/book/9783642149115`, aufgerufen am 2015-07-21

[4] `www.mathematik.uni-wuerzburg.de/~schwartz/Lehre/Graphentheorie/SkriptGraphentheorieSchwartz.pdf`, aufgerufen am 2015-07-21

[5] Möglichkeit die einzelnen Knoten des Graphen abzuarbeiten



(a) ungerichtet (b) gerichtet

(c) gewichtet

Abbildung 2.3.: Kategorisierung der Graphen nach der Art der Verknüpfung von Knoten in ungerichteter, gerichteter sowie gewichteter Graph

Gewichtete Graphen Den Kanten, welche einzelne Knoten untereinander verbinden, sind Zahlenwerte, sogenannte Gewichtungen zugeordnet. Die Gewichtungen haben beispielsweise Auswirkungen auf das Durchlaufen einzelner Knoten, abhängig von der Auswahl des Algorithmus. Beispielsweise könnte ein Algorithmus ausgewählt werden, welcher die kürzeste Wegstrecke von Knoten A nach Knoten C berechnet (Abbildung 2.3c). Die Summe der niedrigsten Werte wäre beispielsweise der optimale kürzeste Pfad. Im konkreten Beispiel der Pfad $< A, B, C >$.

2.5. Graphdatenbanken

Informationen werden in relationale Datenbanken, wie bereits in Kapitel 2.2 beschrieben, in Tabellen gespeichert. Beziehungen zwischen gespeicherten Objekten in unterschiedlichen Tabellen werden durch Schlüssel realisiert. Das Web-2.0- und Big-Data Zeitalter mit hohem Aufkommen an unstrukturierten, schnell wachsenden und sich verändernden Daten bringt die relationale Datenbank an ihre Grenzen. Diesen Herausforderungen sind die relationale Datenbanken nur schwer gewachsen, da diese mit solchen Strukturen nur im mäßigen Ausmaß zurecht kommt. Dies führt unter anderem zu Performance- und Verfügbarkeitsproblemen (Edlich u. a., 2010). Dem entgegenwirken sollen Graphdatenbanken. Im Gegensatz zu relationalen Datenbanken werden in Graphdatenbanken miteinander verbundene Daten in Form von Knoten- und Kanten, genannt "connected data", gespeichert (Robinson, Webber und Eifrem, 2013).

Robinson, Webber und Eifrem (2013) beschreiben Graphdatenbanken als ein online Datenbankmanagementsystem welches Daten als Graph darstellt und CRUD Funktionalität (**C**reate[6], **R**ead[7], **U**pdate[8], **D**elete[9]) zur Verfügung stellt. Aufgrund ihrer Leistungsfähigkeit, ihrer Verfügbarkeit und der Gewährleistung von Datenintegrität werden sie in der Online-Transaktionsverarbeitung (OLTP-Systeme), auch Echtzeitransaktionsverarbeitung genannt, eingesetzt. Im Gegensatz zu anderen Datenbanken, wie beispielsweise relationalen Datenbanken, sind im Datenbankmodell der Graphdatenbanken der Hauptbestandteil Beziehungen und werden als echtes Element in der Datenbank gespeichert. Dies hat zwar den Nachteil, dass zum Zeitpunkt des Einfügens die Kosten der Erstellung der Beziehung getragen werden müssen, jedoch stehen die Beziehungen bei den häufig auftretenden Abfragen dann bereits zur Verfügung. Daher eignen sich Graphdatenbanken für Daten, in welchen die Beziehungen im Mittelpunkt stehen und deren Struktur einem Graphen ähnelt.

Die Speicherung der Daten in Graphdatenbanken kann *nativ* oder *nicht-*

[6] engl.: erstellen, erzeugen
[7] engl.: lesen, auslesen
[8] engl.: aktualisieren
[9] engl.: löschen

nativ erfolgen. Die native Speicherung ist speziell für das Speichern und Verwalten von Graphen konzipiert. Nicht native Technologien verwenden relationale- oder objektorientierte Datenbanken zur Speicherung von Graphen. Wie bei der Speicherung von Graphen unterscheidet man auch bei der Verarbeitung von Graphen zwischen nativ und nicht nativ. Die native, auch indexfreie Adjazenz genannt, ist die effizientere Verarbeitung. Dabei erfolgt beispielsweise die Traversierung nicht über globale Indizes, sondern über die direkte Referenz eines Knoten zu seinem Nachbarknoten. Dabei agiert jeder Knoten als eigener "Micro Index". Native Verarbeitung liefert Vorteile bei der Leistung bezüglich Traversierung. Abfragen, die nicht mit der Traversierung zusammenhängen, sind mit Graphdatenbanken ebenso möglich, jedoch deutlich speicherintensiver. (Robinson, Webber und Eifrem, 2013)

Vergleicht man Graphdatenbanken mit relationalen Datenbanken, eignen sich relationale Datenbanken dank ihrer Tabellenstruktur für Abfragen, die Informationen nach bestimmten Restriktionen suchen. Dies ist mit Graphdatenbanken auch möglich, jedoch werden dazu Indexmechanismen benötigt. Ihre Stärke liegt dabei klar bei der Traversierung, ausgehend von einem angegebenen Knoten. Hierzu werden in relationalen Datenbanken Indizes über Fremdschlüssel benötigt, um mittels `JOIN`-Abfragen die Beziehungen zu finden. Die Abbildungen 2.4 und 2.5, sollen anhand eines einfachen Beispiels den Vorteil der Traversierung in Graphdatenbanken veranschaulichen. Für die Selektierung aller Personen, welche im Kernkraftwerk arbeiten, müssen in einer relationalen Datenbank (Abb. 2.4) drei Indizes durchlaufen werden um die Beziehung zwischen Person und Firma herstellen zu können. In einer Graphdatenbank (Abb. 2.5) muss nur ein Index durchlaufen werden, um den Startknoten zu finden. Die Beziehung zwischen Person und Firma wird über die Traversierung zwischen den Knoten hergestellt. In diesem einfachen Beispiel spielt die Leistung nur eine untergeordnete Rolle. Wächst jedoch der Datenumfang, steigt auch die Anforderung an die Leistungsfähigkeit. Ignoriert man in Graphdatenbanken den Indexsuchlauf für den Startknoten, kann man davon ausgehen, dass die Abfrage im optimalen Fall im konkreten Beispiel in konstanter Zeit $\mathcal{O}(1)$ durchgeführt wird. Die Laufzeit in relationalen Datenbanken beläuft sich hingegen für jeden Indexdurchlauf auf $\mathcal{O}(log_2 n)$. Sind in einer Graphdatenbank beispielsweise alle Knoten miteinander verbunden und werden schlechtesten Fall bei der

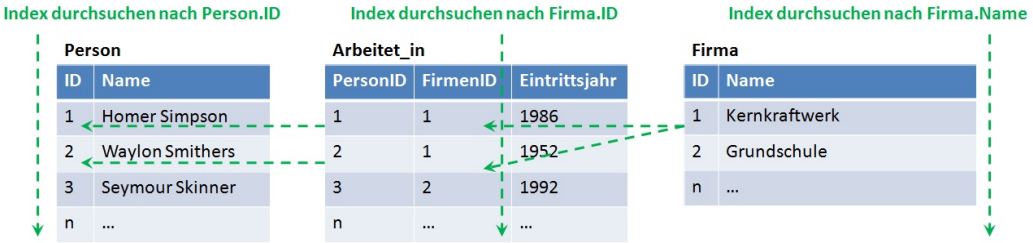


Abbildung 2.4.: Grafische Darstellung einer Traversierung über Beziehungen zwischen unterschiedlichen Relationen in einer relationalen Datenbank (Quelle: nach Domenjoud (2016))

Traversierung alle Knoten durchlaufen, beträgt die Laufzeit einer solchen Abfrage $\mathcal{O}(n)$.(Robinson, Webber und Eifrem, 2013)

Robinson, Webber und Eifrem (2013) charakterisieren Leistungsfähigkeit, Flexibilität und Agilität als große Stärke von Graphdatenbanken.

Leistungsfähigkeit Im Gegensatz zu relationalen Datenbanken oder anderen Technologien für NoSQL Datenbanken bieten Graphdatenbanken einen herausragend leistungsfähigen Umgang mit stark vernetzten Daten. Relationale Datenbanken sind `JOIN`-intensiv. Die Performance verschlechtert sich, je größer die Daten werden. Nicht jedoch bei Graphdatenbanken. Hier bleibt die Performance ansatzweise konstant, da es sich um lokale Operationen direkt am Graph handelt.

Flexibilität Die Datenstruktur ermöglicht es, neue Knoten und Beziehungen hinzu zu fügen, ohne dabei vorhandene Abfragen oder die Funktionalität der Anwendung zu beeinträchtigen. Dies wirkt sich laut Robinson, Webber und Eifrem (2013) positiv auf die Anpassungsfähigkeit und auf Risiken in der Entwicklung aus. Auch relationale Datenbanksysteme unterstützen natürlich das Hinzufügen von neuen Knoten und Beziehungen, jedoch mit dem Unterschied, dass ein neuer Knoten- bzw. Kantentyp unter Umständen eine neue Tabelle darstellt und somit eine Änderung der Architektur und Funktionalität mit sich zieht. Durch die Flexibilität muss die Datenbank nicht bis ins kleinste Detail akribisch modelliert werden. Sie

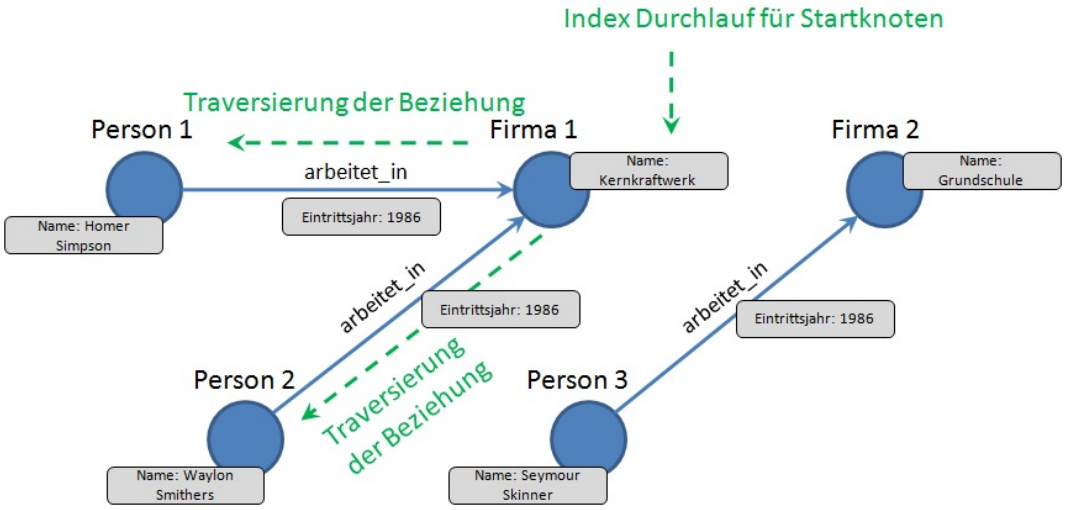


Abbildung 2.5.: Grafische Darstellung einer Traversierung über Beziehungen zwischen unterschiedlichen Knotentypen in Graphdatenbanken (Quelle: nachDomenjoud (2016))

ermöglicht es, auf sich ändernde Geschäftsprozesse und Anforderungen zu reagieren.

Agilität Laut Robinson, Webber und Eifrem (2013) unterstützen Graphdatenbanken den Trend der agilen Softwareentwicklung. Das Datenmodell soll dabei mit den Entwicklungsschritten der Applikation weiterentwickelt werden. Schema-freie Datenbanken, wie es Graphdatenbanken sind, ausgestattet mit einer textbasierten Abfragesprache und einer Programmierschnittstelle (API) unterstützen agile Softwareentwicklungsprozesse optimal.

Das Datenmodell in Graphdatenbanken dient zur Speicherung vernetzter Informationen. Als elementare Eigenschaft für die Erzeugung von Graphen müssen Graphdatenbanken Knoten und Kanten hinzufügen können. Graphdatenbanksysteme erlauben je nach Technologieanbieter sowohl die Definition von gerichteten als auch von ungerichteten Kanten. Alternativ können in den unterschiedlichen Systemen Kanten mit Gewichten versehen werden. Da das Graphenmodell mit Knoten und Kanten im praktischen

Einsatz an seine Grenzen stößt, hat sich im Umfeld von Graphdatenbanken das *Property-Graph-Modell* etabliert. Dabei wird das einfache Datenmodell um einen weiteren Typen, die "Properties [10]" erweitert. Ein Property-Graph besteht also im wesentlichen aus *Knoten, Beziehungen* und *Eigenschaften*. Eigenschaften sind *Key-Value-Pairs*[11], bei welchem der Schlüssel aus einem String und der dazugehörige Wert aus einem beliebigen Datentyp besteht. Eigenschaften werden in Knoten und Kanten eingebunden. Die Beziehungen (Relations) verbinden einzelne Knoten. Sie hat *immer* eine Richtung und verbindet einen Start- und einen Endknoten. In einigen Graphdatenbanksystemen ist es möglich, Knoten und Kanten mit einem eindeutigen Identifier einem sogenannten Label, zu versehen. Identifier (ID's) werden typischerweise vom System automatisch vergeben, um Knoten und Kanten zu identifizieren. Labels können zudem explizit gesetzt werden. So stellt beispielsweise in der Abbildung 2.6 die Beschreibung "bestellt" ein explizit vergebenes Label für die Kante dar. (Robinson, Webber und Eifrem, 2013) (Edlich u. a., 2010)

Abbildung 2.6 zeigt die vereinfachte Modellierung eines Bestellverlaufes als Property Graph.

Die Eigenschaft der Graphdatenbanken, Knoten und Kanten als verbundene Struktur abzubilden, ermöglicht es, beliebige Modelle zu erstellen die den zu realisierenden Aufgabenbereich widerspiegeln. Die daraus resultierenden Modelle sind einfacher und ausdrucksvoller als Modelle, die mit relationalen Datenbanken erstellt werden. Im folgenden Kapitel werden die für die Evaluierung ausgewählten Graphdatenbanken-Technologien, Neo4j und OrientDB, diskutiert.

2.5.1. Neo4j

Die ersten Entwicklungsschritte von Neo4j begannen schon im Jahr 2000, als die Mitbegründer auf ein Graphenproblem stießen, welches mit einer relationalen Datenbank nicht realisierbar war. Über Jahre hinweg wurde am Projekt Neo4j gearbeitet, bis schließlich 2010 die erste Open Source

[10] engl.: Eigenschaften
[11] engl.: Schlüssel-Werte-Paare

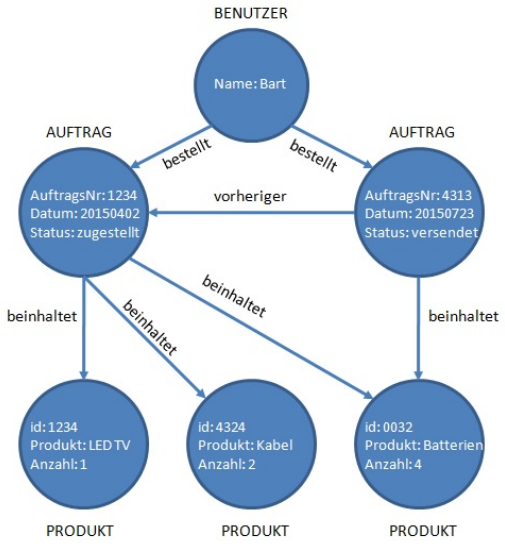


Abbildung 2.6.: Darstellung eines Bestellverlaufs in Form eines Property-Graph inklusive Attribute für Knoten, sowie Knoten- und Kantenlabel (Quelle: nach Robinson, Webber und Eifrem, 2013)

Graphdatenbank Neo4j 1.0 eingeführt wurde. Im Dezember 2013 wurde die neue Version Neo4j 2.0 veröffentlicht. Das in Java geschriebene Datenbankmanagementsystem wird von Neo Technology in zwei verschiedenen Ausführungen angeboten: der Community- und der Enterprise-Edition. Grundlegende Funktionalität liefert die Community-Edition. Die Enterprise-Edition erweitert die Grundfunktionalität um Überwachungsmechanismen, sowie verbesserte Performance- und Skalierungsfunktionen. Die Enterprise-Edition eignet sich für den kommerziellen Einsatz und unterliegt der *Affero General Public License (AGPL) v3* sowie der *Neo4j commerical license*. Die Community-Edition darf nur in freier Software eingesetzt werden und steht unter der freien Lizenz *GNU General Public License (GPL) v3*. (Neo Technology Inc., 2015)
Tabelle 2.2 liefert einen kurzen Überblick über die Charakteristik von Neo4j.

Neo4j bietet zwei verschiedene Betriebsmodi: das System kann sowohl in Form einer eingebetteten Bibliothek als auch in Client-Server-Konfiguration betrieben werden. Die Client-Server-Konfiguration erfordert einen client-

Entwickler	Neo Technology
aktuelle Version	Neo4j 2.2.0, Juli 2015
Datenbankmodell	Graph DBMS
Lizenz	Open Source (GPL, AGPL), Neo4j commerical license
Server Betriebssysteme	Linux, OS X, Windows
Datenschema	schemafrei
APIs	Cypher query language, Java API, RESTful API
Transaktionskonzept	ACID

Tabelle 2.2.: Überblick der Systemeigenschaften von Neo4j (Quelle: nach solid IT GmbH (2015))

seitigen Zugriff über eine REST-Schnittstelle. Entsprechende Schnittstellen, stehen in unzähligen Sprachen wie Java, .NET, Javascript [12] und vielen weiteren Sprachen zur Verfügung. Erfolgt die Verwendung als eingebettete Bibliothek, findet die Kommunikation über die Java-Schnittstelle statt.

Datenmodell

Das Datenbanksystem Neo4j implementiert das Property-Graph-Modell. Dessen zentrale Charakteristiken, wie bereits im Kapitel 2.5 beschrieben, sind Knoten, Kanten und Eigenschaften. Die Knoten wurden im aktuellen Datenbankmodell um ein Label erweitert. Somit können einzelne, mit dem selben Label versehene Knoten zu einer Gruppe zusammengefasst werden. Das Datenbanksystem erlaubt es, Knoten mit mehreren Labels zu versehen. Aus diesem Grunde können Knoten mehreren Gruppen zugehörig sein. Durch das Versehen der Knoten mit Labels können in Anfragen Knotenbezeichnungen miteinbezogen werden, sodass sich die Anfrage nur auf einen Teilgraph und nicht den gesamten Graphen bezieht. Dies ermöglicht eine vereinfachte und effizientere Anfrage. Labels, sowohl für Knoten als auch für Kanten, können während der Laufzeit hinzugefügt und entfernt werden.

[12] `http://neo4j.com/developer/language-guides/`, aufgerufen am 2016-03-22

Im Gegensatz zu Knoten müssen Kanten mit einer Bezeichnung versehen werden. Eine Knotenbezeichnung ist optional.

Kanten beschreiben die Beziehung zwischen den einzelnen Knoten, welche in Neo4j gerichtet sind und aus einem Start- und einem Endknoten bestehen. Eine Schleife, mit anderen Worten, Start und Ende beschreiben den gleichen Knoten, ist erlaubt. Beziehungen können in beide Richtungen traversiert werden. Folglich muss keine weitere Beziehung in die entgegengesetzte Richtung erzeugt werden. Eigenschaften, im Property-Graph-Modell als Properties bezeichnet, sind Schlüssel-Werte-Paare und können Knoten und Kanten optional zugewiesen werden. Wie bereits im vorherigen Kapitel 2 beschrieben, sind die Schlüssel vom Datentyp *String*, während die zugehörigen Werte aus einem zulässigen Java-Datentyp bestehen können. Gleich den relationalen Datenbanken ist der Schlüssel eindeutig und kann nur einmalig vergeben werden.

Das Datenmodell bietet, wie auch relationale Datenbanken, Konzepte zur Datenintegrität. Freistehende Kanten sind nicht erlaubt. Kanten dürfen nur zwischen bereits vorhandenen Knoten erzeugt werden. Analog zur referentiellen Integrität in relationalen Datenbanken erfordert das Löschen eines Knoten das Entfernen von indizierten Kanten. Das heißt, bevor der Knoten gelöscht werden kann, müssen alle ein- und ausgehenden Kanten entfernt werden. Den Knoten und Kanten wird vom Datenbanksystem eine eindeutige Identität zugewiesen, welche einem Primärschlüssel in relationalen Datenbanken entspricht.

Durch das schemalose Datenbankmodell ermöglicht Neo4j beliebige Variationen aus Eigenschaften und Labels für Knoten und Kanten. Nicht nur Eigenschaften und Labels können beliebig kombiniert werden, sondern auch die Verknüpfung von Knoten und Kanten kann beliebig durchgeführt werden. (Neo4 Technology Inc., 2015)

Indexstruktur

Historisch wird in den ersten Versionen von Neo4j im Zusammenhang mit Indizierung der Begriff *Legacy-* oder *manueller Index* verwendet. Die Einführung von Labels für Knoten führte zur Erweiterung der Software

um *Schema-Indizes*. Bei der Umsetzung der Indizierung handelt es sich um zwei unterschiedliche Konzepte, die weder untereinander auswechselbar, noch in irgendeiner Form untereinander kompatibel sind. Legacy/manuelle Indizes sind exakte oder Volltextindizes für Knoten oder Kanten, sowie räumliche Indizes für Knoten, welche durch das externe Indexsystem *Apache Lucene*[13] umgesetzt werden. Der Index besteht aus einem Werte-Schlüssel-Paar, jeder Knoten bzw. jede Kante muss manuell zum Index hinzugefügt werden. Eine automatische Aktualisierung wird nicht unterstützt. Somit müssen gelöschte oder hinzugefügte Knoten und Kanten manuell dem Index mitgeteilt werden. Aus diesem Grund wird der Legacy Index oft auch als manueller Index bezeichnet.

Schema Indizes basieren auf einem einzelnen Attribut eines bestimmten Labels. Der Index wird automatisch verwaltet und vom Datenbanksystem aktualisiert, sobald sich der Graph ändert. Mit der Erzeugung eines neuen Schema Index werden alle Knoten, welche die gleiche Label-Attribut-Kombination besitzen, asynchron und automatisch in den Index aufgenommen. Die Verwendung von Schema-Indizes ermöglicht eine Optimierung von Anfragen, eine Kanten-Indizierung ist jedoch leider nicht möglich. (Hunger, 2015) (Neo4 Technology Inc., 2015) (Small, 2015)

Zugriffskonzepte

Neo4j bietet unterschiedliche Möglichkeiten, um CRUD-Operationen auf die Datenbasis anzuwenden. Die drei wesentlichen Zugriffskonzepte *Java Core API*, *Traversal Framework* (als Erweiterung der Java API) sowie die Abfragesprache *Cypher* sind sowohl im eingebetteten Betriebsmodus als auch via RESTful Schnittstelle im Client-Server Modus möglich. Cypher ist die primäre von Neo Technology entwickelte Abfragesprache. Gremlin wird als sekundäre Abfragesprache zur Verarbeitung, Analyse und Manipulation von Graphen unterstützt (Malette, 2015). Sie ist im Rahmen des Apache TinkerPop Projekts [14] entstanden.

[13] `https://lucene.apache.org/core/`, aufgerufen am 2016-03-22
[14] `http://tinkerpop.incubator.apache.org/`, aufgerufen am 2016-03-22

Java Core API Die Java Core API stellt elementare CRUD-Operationen zur Verfügung und erlaubt Lese- und Schreibzugriff auf Knoten, Kanten und deren Eigenschaften. Dadurch können Knoten erzeugt und optional mit Labels und Eigenschaften versehen werden. Kanten können nur unter Angabe von existierenden Knoten erzeugt werden, wobei die Reihenfolge der Knoten der Richtung der Kante entspricht. Optional können Kanten, ebenso wie Knoten, mit Labels und Eigenschaften versehen werden. Knoten mit ihren Eigenschaften und Labels, sowie ihren aus- und eingehenden Kanten können ausgelesen werden. Ein lesender Zugriff auf Kanten kann das Label, die zugehörenden Eigenschaften, sowie den Start- und den Endknoten zurückliefern. Knoten und Kanten können jederzeit angefügt oder entfernt werden, und eine Manipulation von Eigenschaften ist jederzeit möglich. (Edlich u. a., 2010) (Junghanns, 2014)

Die Java-API von Neo4j bietet für die Berechnung von Pfaden bereits vorgefertigte Algorithmussammlungen. Zudem können jederzeit beliebige weitere Algorithmen anwendungsseitig implementiert und integriert werden (FH-Köln, 2011).

Traversal Framework Das Traversal Framework dient zur unterschiedlichen Traversierung von Daten. Dazu wurde die Core API durch Hilfskonstrukte erweitert, um verschiedene Traversierungsoptionen zu ermöglichen. Das heißt, grundsätzlich werden hier Pfade gesucht und zurückgegeben. Die Implementierung der geeigneten Kriterien erfolgt durch die Verwendung der Java API. Es werden ausschließlich lesende Anfragen auf die Daten unterstützt.

Ausgangspunkt einer Traversierung ist die `TraversalDescription`. Sie dient zur Beschreibung bzw. Initialisierung und gibt an, wie die Abfrage behandelt werden soll. Eine Reihe von zusätzlichen Parametern kann angegeben werden, welche Abfragen beeinflussen. Mit Hilfe des Parameters `PathExpander` wird festgelegt, welche Kantenlabels beim Durchlaufen berücksichtigt werden. Unter Angabe mehrerer Labels, kann durch die Reihenfolge die Priorität der einzelnen Labels festgelegt werden. Zur Berücksichtigung sämtlicher Knoten werden dem Parameter keine Labels übergeben. Filter- und Abbruchkriterien können mit der Schnittstelle `Evaluator` implementiert werden. Im Wesentlichen wird dabei für jede Position festgelegt, ob die

Traversierung fortgesetzt werden soll und ob der Knoten bzw. die Kante im Ergebnis enthalten sein soll. Das Verhalten der Traversierung, das heißt, in welcher Reihenfolge der Übergang auf den nächsten Knoten erfolgt, lässt sich mit `BranchSelector/BranchOrderingPolicy` festlegen. Das Framework stellt Tiefen- und Breitensuche zur Traversierung bereits zur Verfügung. `Uniqueness` legt fest, wie oft Kanten und Knoten beim Durchlaufen besucht werden sollen bzw. können.

Nach der entsprechenden Definition der Abfrage wird unter der Angabe eines Startknotens die Abfrage mittels der Funktion `traverse` gestartet. (Neo4 Technology Inc., 2015) (Edlich u. a., 2010)

CRUD-Operationen mittels Cypher Hunger (2014) beschreibt *Cypher* als proprietäre, deklarative und graphenorientierte Abfragesprache, welche lesenden und schreibenden Zugriff auf die Daten erlaubt. Cypher wurde zur leichteren Verständlich- und Leserlichkeit für die nicht-affine Zielgruppe an die Syntax von SQL angepasst. Die Abfragesprache ermöglicht graphenbasierte Anfragen zur Informationsextraktion sowie das Prüfen von Erreichbarkeit bestimmter Knoten im Graphen und die Berechnung von Pfaden. Nachfolgend werden die grundlegenden Schlüsselwörter beschrieben.

`MATCH`	erlaubt es bestimmte Muster von Identifikatoren für Knoten und Bezeichnungen anzugeben, um Informationen aus der Datenbasis auszulesen
`WHERE`	Selektion ähnlich zur SQL-Klausel. Dient zur Filterung der Ergebnisse anhand von Prädikaten
`RETURN`	ähnlich zu `SELECT` in SQL. Dient zur Projektion der Rückgabewerte
`ORDER BY`	Sortierung
`CREATE`	Knoten- und Kanteninstanzen können erzeugt werden. Das Anlegen einer Kante erfordert bereits vorhandene Knoteninstanzen
`SET`	neue Eigenschaften können zu Knoten- oder Kanteninstanzen hinzugefügt werden oder bereits vorhandene Eigenschaften aktualisiert werden
`REMOVE`	Eigenschaften von Knoten- oder Kanteninstanzen können entfernt werden

`DELETE` im Gegensatz zu `REMOVE` ermöglicht die `DELETE`-Klausel das Entfernen von Knoten und dazugehörigen Beziehungen, bzw. das Entfernen von einzelnen Kanteninstanzen

Beispielsweise wird mit der Anweisung 2.1 ein Knoten mit den Eigenschaften `name` und `alter` erzeugt. Knoten werden in runden Klammern eingeschlossen und die dazugehörigen Eigenschaften befinden sich ähnlich zum JSON-Format in geschwungenen Klammern.

```
1  CREATE (p1:Person {name: "Bart", alter: 10})
2  RETURN p1
```

Listing 2.1: Cypher Code zur Erzeugung eines Knoten vom Typ Person, welche durch die Attribute `name` und `alter` beschrieben wird

Der Cyper-Codeausschnitt 2.2 zeigt das Erzeugen einer Kante zwischen zwei bereits existierenden Knoten. Die Klausel `MATCH` sucht nach zwei Knoten mit dem Label Person, welche mittels der `WHERE`-Klausel eingeschränkt werden. `CREATE` erzeugt die gerichtete Beziehung `BEFREUNDET_MIT` zwischen den Knoten *Bart* und *Milhouse*. Gerichtete Kanten werden als Pfeile `->`, `<-` dargestellt. Die Zusatzinformation für Beziehungen, im konkreten Fall `BEFREUNDET_MIT`, steht in eckigen Klammern.

```
1  MATCH (p1:Person), (p2:Person)
2  WHERE p1.name = "Bart"
3    AND p2.name = "Milhouse"
4  CREATE p1-[r:BEFREUNDET_MIT]->p2
5  RETURN r
```

Listing 2.2: Cypher Code zur Erzeugung einer Kante mit dem Label `BEFREUNDET_MIT` zwischen zwei Knoten vom Typ Person, welche über den Vergleichsoperator selektiert werden

Eine detaillierte Beschreibung zu Cypher und dessen Verwendung liefert das Online Manual von Neo4j[15]. Um sich einen schnellen Überblick über die möglichen Cypher Ausdrücke zu verschaffen, wird die Cypher Referenz Card von Neo4j[16] empfohlen.

[15] `http://neo4j.com/docs/2.2.3/`, aufgerufen am 2015-12-05

[16] `http://neo4j.com/docs/2.2.3/cypher-refcard/`, aufgerufen am 2015-12-05

CRUD-Operationen mittels Gremlin Redmond und Wilson (2012) beschreiben Gremlin als eine in Groovy geschriebene Abfragesprache zur Traversierung von Graphen. Gremlin ermöglicht lesenden und schreibenden Zugriff auf die Datenbasis. Ebenfalls ist es möglich, via Gremlin Muster innerhalb der Datenbasis zu finden. Für die Beschreibung eines Graphen werden in Gremlin allgemeine mathematische Graphen-Begriffe verwendet. Ein Knoten wird in Gremlin als *Vertex* bezeichnet und anstelle einer Kante wird der Begriff *Edge* verwendet.

Nachfolgend werden die grundlegenden CRUD-Funktionen von Gremlin beschrieben. In Gremlin wird per Konvention die Variable `g` angegeben, welche ein Objekt des Graphen repräsentiert. Lesende und schreibende Operationen werden auf das Graphen-Objekt `g` angewendet. So liefert die Funktion `V()` im Codeausschnit 2.3, angewendet auf das Objekt `g` alle Knoten (Vertices) des Graphen zurück. Analog dazu liefert die Funktion `E()` alle Kanten (Edges) des Graphen.

```
1 g.V()
2 // v[1]
3 // v[2]
4 // v[3]
```

Listing 2.3: Ausgabe der gesamten Knoten eines Graphen in der Sprache Gremlin. Die in den eckigen Klammern angegebenen Zahlen entsprechen der von der Datenbank vergebenen ID

Gremlin ermöglicht es, auf einzelne Knoten zuzugreifen, in dem der Methode `v()` ein Index übergeben wird. Durch die Verkettung mit der Funktion `map()` werden die Eigenschaften des ausgewählten Knotens ausgegeben. Im Codeausschnitt 2.4 wird auf den Knoten mit dem Index eins zugegriffen und alle seine dazugehörigen Eigenschaften, im konkreten Fall `name` und `alter`, ausgegeben. Analog dazu kann auf die Eigenschaften von Kanten mittels der Methode `e()`zugegriffen werden.

```
1 gremlin> g.v(1).map()
2 // {name=Bart, alter=10}
```

Listing 2.4: Ausgabe der Eigenschaften des Knoten mit dem von Neo4j vergebenen Index eins, mittels der Gremlin-Funktion `map`

Gremlin erlaubt schreibende Zugriffe. Der einfachste schreibende Zugriff ist das Erzeugen von Kanten oder Knoten. Durch die Funktionen `addVertex()`

und `addEdge()` können Knoten und Kanten hinzugefügt werden. Codeausschnitt 2.5 erzeugt einen neuen Knoten und weist ihn der Variable `millhouse` zu. Anschließend wird auf den Graphen `g` die Funktion `addEdge()` mit vier Parametern aufgerufen. Der erste Parameter stellt den Startknoten da, in diesem Fall der Knoten mit dem Index null. Der Zweite Parameter ist der Endkonten. Die Beschreibung der Kante wird im dritten Parameter mitgegeben. Eigenschaften, sowohl für Kanten und Knoten werden in eckigen Klammern beschrieben und werden beim Erzeugen von Kanten als vierter Parameter mitübergeben. Die Richtung der Kante wird beim Erstellen durch die Reihenfolge der Knoten bestimmt.

```
1 millhouse = g.addVertex([name: 'Millhouse', alter: 10])
2 // v[1]
3 g.addEdge(g.v(0), millhouse, 'befreundet_mit', [seit: 2010])
4 // e[1][0-befreundet->1]
```

Listing 2.5: Erzeugen eines Knoten mit den Attributen `name` und `alter`, sowie das Erzeugen einer Kante mit dem Label `befreundet_mit` zwischen dem zuvor erzeugen Knoten und dem Knoten mit dem vergebenen Index null

Zu den CRUD Operationen zählt aktualisieren und löschen (Update und Delete). Mittels Gremlin kann auf die Eigenschaften des Objekts (Knoten oder Kante) zugriffen werden und sobald das gewünschte Objekt gefunden wurde, die jeweilige Eigenschaft angepasst werden. Ident zum Aktualisieren einer Eigenschaft funktioniert das Hinzufügen einer neuen Eigenschaft. Ein hinzufügen der Eigenschaft `alter` (Codeausschnitt 2.6) erfolgt unter der Voraussetzung, dass die angegebene Eigenschaft nicht vorhanden ist. Besitzt das Objekt die angegebene Eigenschaft, wird diese aktualisiert. Das Entfernen einer Eigenschaft erfolgt durch das Aufrufen der Methode `removeProperty()` auf das jeweilige Objekt unter der Angabe der Eigenschaft. Im Codeausschnitt 2.6 wird ein Knoten mit dem Index eins im Graphen `g` gesucht, um anschließend die Eigenschaft `alter` zu entfernen. Knoten können durch die Methode `removeVertex()` unter der Angabe des zu löschenden Knoten entfernt werden. Analog funktioniert das Löschen von Kanten durch die Funktion `removeEdge()`.

```
1 millhouse.alter = 8
2 // 8
3 g.v(0).removeProperty('alter')
4 // 10
5 g.removeVertex(g.v(1))
6 // null
```

Listing 2.6: Update des Attributes `alters` des zuvor erzeugten Knoten `millhouse`, sowie das Entfernen des Attributes `alter` durch die Methode `removeProperty` vom Knoten mit dem Index null. Der Knoten mit dem Index eins wird vollständig mit der Methode `removeVertex` entfernt.

Damit wurden die grundlegenden lesenden und schreibenden Operationen vorgestellt. Eine detaillierte Übersicht über die Funktionalität liefert die Gremlin Dokumentation[17]. Nähere Information über Gremlin und das TinkerPop Projekt liefert die Wiki Seite von Gremlin [18].

Charakteristik von Neo4j

Durch die Schema- und Typenfreiheit liefert Neo4j hinsichtlich Daten und Beziehungen zwischen den Daten keinerlei Einschränkung und eignet sich daher sehr gut für unstrukturierte Daten. Neo4j unterstützt in der aktuellen Version rund 34 Milliarden Knoten und Beziehungen, sowie maximal 234 Milliarden Eigenschaften und 32.000 Typen an Beziehungen, welche in den meisten Anwendungsfällen mehr als ausreichend sein sollten. (Neo4 Technology Inc., 2015).

Neo4j ist ein OpenSource Projekt, unterstützt wie beschrieben Indexierung mittels Lucene und ebenso eine Reihe von Spracherweiterungen, welche sich in vorhandene Frameworks integrieren lassen. Ein wesentlicher Vorteil gegenüber anderen Datenbanken, speziell relationale Datenbanken, ist ihre Schnelligkeit, insbesondere in Verbindung mit Traversierung beziehungsweise Pfadsuche. In relationalen Datenbanken erfolgt die Traversierung über `JOIN`- und Filteroperationen. In Neo4j hingegen erfolgt die Traversierung mittels der Abfragesprache Cypher oder dem dafür speziell vorgesehen Traversal Framework, welches es ermöglicht, viele der Traversierungsprobleme

[17] `http://gremlindocs.com/`, aufgerufen am 2015-11-26
[18] `https://github.com/tinkerpop/gremlin/wiki`, aufgerufen am 2015-11-26

mittels Breiten- und Tiefensuche zu lösen. Ein eingeschränkter Vorteil ist die Hochverfügbarkeit und das hohe Lesevolumen, welches jedoch nur in der lizenzpflichtigen Enterprise-Edition bereitgestellt wird. Die Enterprise-Edition ermöglicht des weiteren Sharding. Dabei handelt es sich um eine Methodik der Datenbankpartitionierung, welche den Datenbestand in mehrere Teile aufteilt und diese von einzelnen Servern verwaltet werden[19]. (Redmond und Wilson, 2012) (Neo Technology Inc., 2015)

Redmond und Wilson (2012) beschreiben als Schwäche von Neo4j das Erwerben einer Lizenz, um mit Enterprise-Tools wie Backuplösungen und High Avialabiltiy (hohe Verfügbarkeit) arbeiten zu können. Als wesentlichen Nachteil beschreiben Edlich u. a. (2010) die Java-Begrenzung. Die Performance von Neo4j hängt stark von der Leistung der Java-Virtual-Machine auf dem Operativsystem ab. Des weiteren gibt es Begrenzungen hinsichtlich Sharding. Da ein Graph nicht automatisch partitionierbar ist, müssen Optimierungen hinsichtlich der Partitionierungsstrategie manuell durchgeführt werden.

2.5.2. OrientDB

OrientDB ist eine Open Source Dokumenten-Graph Datenbank. Sie bietet native Graphfunktionalität inklusive Eigenschaften und Funktionalitäten von Dokumentendatenbanken. OrientDB wurde von Orient Technologies entwickelt und steht unter der Apache 2.0 Lizenz. Das ermöglicht eine Anwendung im freien als auch im kommerziellen Bereich. Die erste Version der in Java implementierten Graphdatenbank erschien 2010. Durch die Implementierung in Java ist die Ausführung, ähnlich zu Neo4j, an die JVM (Java Virtual Machine) gebunden. Somit ermöglicht OrientDB allen Plattformen mit einer kompatiblen JVM das Datenbanksystem zu verwenden. OrientDB steht in zwei Ausführungen zur Verfügung: in der *Community-* und in der *Enterprise-Edition*. Die Enterprise Edition erweitert die Grundfunktionalität der Community Edition um Analyse- und Aufzeichnungswerkzeuge für Abfragen, Cluster Management, Live Überwachung und 24x7 Support. Das Dokumenten-Graph Datenbanksystem lässt sich als

[19] `http://db-engines.com/de/article/Sharding`, aufgerufen am 2016-01-11

Entwickler	Orient Technologies LTD
aktuelle Version	OrientDB 2.1.0, 05.August 2015
Datenbankmodell	Document Store, Graph DBMS, Key-Value Store, Objekt DBMS
Lizenz	Open Source (Apache2 License), OrientDB commerical license
Server Betriebssysteme	Alle Betriebssysteme mit kompatibler Java Virtual Machine
Datenschema	schemalos, fest und semi-struktuiert
APIs	Java API, RESTful API, Blueprints API, Gremlin
Transaktionskonzept	ACID

Tabelle 2.3.: Überblick Systemeigenschaften OrientDB (Quelle: nach (solid IT GmbH, 2015))

Client-Server-Konfiguration verwenden, oder alternativ eingebettet in einer Java Anwendung betreiben. Neben einer zentralen Verwendung ist auch der Einsatz eines dezentralen Systems, verteilt auf mehre Server möglich. Eine Master-Master Replikation für die dezentrale Verwendung bietet neben einer horizontalen Skalierbarkeit von Lesen und Schreiben auch eine höhere Ausfallsicherheit.

Ein besonderes Merkmal von OrientDB ist die Umsetzung als Multi-Model DBMS. OrientDB bietet die Möglichkeit, verschiedene Datenmodelle zu nutzen und unterstützt **Graphen-, Dokumenten-, Key/Value-** und **Objektmodelle**. Die Speicherung der Information erfolgt generell in Dokumenten und dient als Basis für die Umsetzung als Graphdatenbank. Die Graphdatenbank ermöglicht durch die Definition von Beziehungen von Dokumenten eine Repräsentation als Graph. In allen Datenmodellen ist es möglich, ein Schema für die Anwendungsdomäne zu definieren, auf ein Schema komplett zu verzichten oder eine Kombination eines Schemas mit semi-strukturierten Daten zur Modellierung zu verwenden. Ein weiteres Unterscheidungsmerkmal ist die Datenbanksicherheit. Im Gegensatz zu anderen Graphdatenbanken bietet OrientDB eine systemseitige Rechteverwaltung, basierend auf den Berechtigungskonzept von Benutzern und Rollen. Darüber hinaus setzt sich OrientDB mit der Erweiterung von SQL-Standards anstelle einer nicht standardisierten Abfragesprache von anderen Herstellen ab. Die Informationen

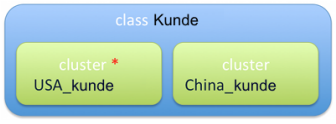
zu OrientDB stammen aus den Ausführungen von Edlich u. a. (2010) und größtenteils aus der von Orient Technologies (2015) erstellten Dokumentation.

Datenmodell

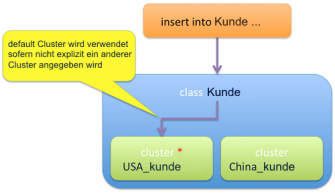
Zur Einführung werden kurz die Modelle Key/Value und Objektmodell erläutert. Die Ausarbeitung behandelt das Thema Graphdatenbank, daher gilt das Hauptaugenmerk dem eine Schicht über dem Dokumentenmodell angesiedelten Graphdatenmodell und der Basis selbst, dem Dokumentenmodell.

Dokumentenmodell Die Basis für die Speicherung von Informationen - sowohl im Dokumentenmodell als auch im Graphenmodell - sind Dokumente, welche sich aus einer Menge von Schlüssel-Wert-Paaren zusammensetzen. Schlüssel sind innerhalb eines Dokuments eindeutige Bezeichner vom Datentyp `String` und erlauben den Zugriff auf die Werte. Den Werten stehen die üblichen Primitive wie Integer, String etc. zur Verfügung oder es wird den Attributen als Wert ein eingebettetes Dokument zugewiesen. Dokumente sind flexibel und leicht zu ändern, da sie nicht zwingender Weise an ein Schema gebunden sind. OrientDB erlaubt, es Dokumente zu gruppieren und verwendet zur Umsetzung der Gruppierung das Konzept von *classes*[20] und *clusters*[21]. Ein Dokument besteht aus Attributen und gehört zur einer *Klasse*. *Cluster* werden zur Gruppierung von Daten verwendet. Somit können bestimmte Gruppen durchsucht werden, ohne dabei eine Abfrage auf die gesamte Klasse durchzuführen. Beim Erstellen einer Klasse erzeugt OrientDB standardmäßig einen Cluster, in dem alle Informationen gespeichert werden, sofern nicht dediziert ein bestimmter Cluster angegeben wird. Abbildung 2.7a zeigt den Aufbau einer Klasse `Customer`, die sich in zwei Cluster unterteilt: Kunden für USA und China. Der als Standard definierte Cluster ist mit einem roten Stern markiert. Operationen ohne Angabe eines bestimmten Clusters werden auf den Standardcluster ausgeführt. Darstellung 2.7b veranschaulicht das Hinzufügen eines neuen

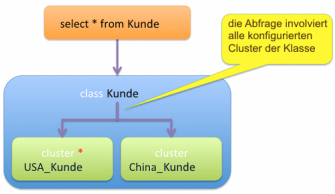
[20] engl.: Klassen
[21] engl.: Gruppen



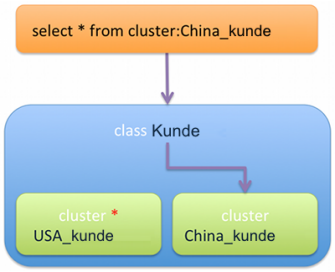
(a) Customer Klasse



(b) Standardmäßiger Cluster beim Einfügen



(c) Selektieren aus Klasse



(d) Selektieren aus Cluster

Abbildung 2.7.: Grafische Darstellung einer Untergliederung der Klasse `Kunde` in zwei unterschiedliche Untergruppen, `USA_Kunde` und `China_Kunde`, die als Cluster bezeichnet werden, sowie die Darstellung des Einfügens und Selektieren aus dem standardmäßig definierten Cluster und unter Angabe eines dedizierten Clusters (Quelle: Orient Technologies, 2015)

Datensatzes, ohne Angabe eines bestimmten Clusters. Die Daten werden in den Standard-Cluster geschrieben. Direkte Abfragen auf eine Klasse, ohne Angabe eines Clusters, involviert alle konfigurierten Cluster. Die Abfrage in der Skizze 2.7c durchsucht die gesamte Klasse. Das Verwenden von Clustern ermöglicht, wie bereits beschrieben, das Gruppieren von Daten. Eine Abfrage gegen einen Cluster erhöht die Performance, da nicht die gesamte Klasse, sondern nur der angegebene Cluster durchsucht werden muss (Abb. 2.7d).

Das Dokumentenmodell ermöglicht es, Dokumente untereinander zu verbinden. Dokumente können dazu in andere Dokumente eingebunden werden, oder mittels einem *LINK* miteinander verbunden werden. Erfolgt ein Zugriff auf ein Dokument, werden die Verbindungen zu anderen Dokumenten von OrientDB automatisch aufgelöst.

Key/Value-Modell Dabei handelt es sich um das einfachste Modell in OrientDB. Jede Information in der Datenbank kann über einen Schlüssel angesprochen werden. Die zum Schlüssel dazugehörige Information kann in einfachen oder komplexen Datentypen gespeichert werden. OrientDB erlaubt des weiteren Dokumente oder Graphen Elemente als Datentyp für die Werte in einem Schlüssel-Werte-Paar. Dies unterscheidet das klassische Schlüssel-Werte-Modell vom Modell in OrientDB. Beziehungen zwischen Schlüssel-Werte-Paaren können mittels sogenannten Links erzeugt werden.

Objektmodell Das Objektmodell entstand durch den Einsatz von objektorientierten Programmiersprachen und soll die klassischen Attribute der objektorientierten Programmierung wie Vererbung, Polymorphie und direkte Bindung zwischen Objekten unterstützen. Im Objektmodell von OrientDB werden Klassen als Cluster und Objekte als Dokumente oder Knoten dargestellt. Eigenschaften von Objekten werden durch in Dokumenten enthaltene Felder oder Eigenschaften von Knoten repräsentiert. Im Gegensatz zum relationalen Datenbankmodell und dem klassischen Objekt Modell, in welchen Verbindungen über "Relationships" bzw. "Pointer" umgesetzt werden, verwendet das Objekt Modell in OrientDB sogenannte Links, um Beziehungen umzusetzen. Dazu speichert das Ausgangsobjekt die Referenz des Zielobjektes.

Graphmodell OrientDB verwendet zur Modellierung von Graphen, wie auch auch Neo4j, das Property-Graphen-Modell (Abb. 2.6). Dieses kann schemalos, nach einem Schema oder semi-strukturiert verwendet werden. Ein Schema wird wie in der objektorientierten Programmierung durch eine Menge von Klassen definiert. Eine Klasse definiert den Typ eines Dokumentes und legt fest, welche Eigenschaften und dazugehörige Integritätsbedingungen das Dokument besitzt. Mittels Integritätsbedingungen lassen sich Wertebereiche unterschiedlicher Primitive einschränken und festlegen, ob eine Eigenschaft obligatorisch ist, oder es den Wert `null` enthalten darf. OrientDB erlaubt des weiteren das Modellieren von Vererbungshierarchien. Dadurch werden Eigenschaften von Oberklassen an alle vererbten Unterklassen weitergegeben.

Die Darstellung von Knoten innerhalb des Graphes erfolgt durch Dokumente. Per Definition besteht der Property-Graph in OrientDB aus Knoten und Kanten. Diese bestehen zumindest aus folgenden Eigenschaften, welche beim Erstellen eines Objekts systemseitig vergeben werden:

- Knoten
  - einem eindeutigen Identifikator
  - einer Menge von eingehenden Kanten
  - einer Menge von ausgehenden Kanten
- Kante
  - einem eindeutigen Identifikator
  - einem Anfangsknoten
  - einem Endknoten
  - einem Label, welches den Typ der Kante zwischen Start- und Endknoten beschreibt

Zusätzlich zu den vom System festgelegten Eigenschaften können sowohl bei Kanten als auch bei Knoten weitere Attribute hinzugefügt werden. Für die Repräsentation von Kanten unterscheidet OrientDB im Graphmodell zwei verschiedene Arten: sogenannte *Lightweight* Kanten-Repräsentation und ordentliche Kanten-Repräsentation. Lightweight Kanten haben keine Identität in der Datenbank und können nur verwendet werden, wenn keine Eigenschaften für die Kante definiert werden und es genau eine Kante zwischen zwei Knoten gibt. Werden Lightweight Kanten verwendet, wird für die Kante vom System kein Dokument erzeugt. Die Knoten (welche als Dokumente repräsentiert werden) beinhalten einen direkten Link auf den jeweiligen anderen Knoten. Seit der Entwicklung von OrientDB 2.0 werden Lightweight Kanten standardmäßig deaktiviert. Unter der Verwendung von regulären Kanten wird für eine Kante ein Dokument erstellt welches Start- und Endknoten verbindet. Der Startknoten hat als ausgehende Kante den eindeutigen Identifikator der jeweiligen Kante in seinen Eigenschaften eingetragen. Vice versa befindet sich der eindeutige Identifikator beim Endknoten in den ausgehenden Kanten.

Indextyp	Durable	Trans-aktionen	Bereichs-abfragen	Hauptmerkmal	Beschreibung
SB-Baum[22]	Ja	Ja	Ja	Konglomerat aus Hash und Lucene	
Hash	Ja	Ja	Nein	Schnelle Suche, geringer Speicher-platz	funktioniert wie HashMap, daher schnelle Punktab-frage, jedoch keine Bereichsabfrage möglich
Lucene	Ja	Ja	Ja	gut geeignet für Volltext-Index und mehrdimensiona-le Index	Kann nur für Volltext- und mehrdimensio-nale Indizes verwendet werden

Tabelle 2.4.: Eigenschaften der drei grundlegenden Indexstrukturen in OrientDB (Quelle: nach Orient Technologies (2015))

Indexstruktur

OrientDB unterstützt drei unterschiedliche Indizes für den effizienten Zugriff auf die in der Datenbank gespeicherte Dokumente. Tabelle 2.4 liefert einen Überblick über die Eigenschaften und beschreibt kurz die drei grundlegenden Indextypen von OrientDB. SB-Baum, Hash und Lucene unterstützen Transaktionen und sind Durable. Das heißt, dass nach einer Durchführung einer Transaktion nicht nur die Daten in die Datenbank geschrieben werden, sondern auch die Daten des Index geschrieben werden. Dies hat einen Mehraufwand bei Schreiboperationen zur Folge, stellt aber konsistente Daten im Index sicher und der Index muss nach etwaigen Ausfällen bzw. Abstürzen nicht neu erstellt werden.

– Die Indizes können für ein Klassen Attribut bzw. für mehrere Klassen-Attribute definiert werden, welche einen eindeutigen Bezeichner besitzen. Attribute, welche an ein Schema gebunden sind, werden automatisch vom System aktualisiert. Semi-strukturierte und schemalose Attribute, welche nicht definiert wurden, müssen manuell durch den Entwickler über die Java

[22]basiert auf dem B-Baum. Optimierungen hinsichtlich Einfügen von Daten und Bereichsabfragen

API oder entsprechende SQL Kommandos aktualisiert werden. Per Java API und SQL Kommandos werden Befehle zur Index-Erzeugung, Löschen von einzelnen Einträgen, dem vollständigen Löschen von Indizes sowie Punkt- und Bereichsabfragen zur Verfügung gestellt.

Ausführliche Beschreibungen zur Verwendung von Indizes liefert das Handbuch von OrientDB [23].

Zugriffskonzepte

OrientDB bietet unterschiedliche Optionen, um CRUD-Operationen auf die Daten anzuwenden. Einerseits kann, wie in Neo4j, die Verwaltung der Daten über Java APIs erfolgen, oder andererseits über die primäre Abfrage Sprache "OrientDB-SQL", welche den SQL-Standard um Graphoperationen erweitert. Auch außerhalb der Java-API und der Abfragesprache ist der Zugriff auf OrientDB möglich. Dazu stehen unzählige Treiber für Skriptsprachen, sowie für die C-Sprachfamilie zur Verfügung. Eine Liste der möglichen Treiber kann der freien Online-Dokumentation von OrientDB [24] entnommen werden. Zudem bietet OrientDB eine RESTful Schnittstelle, die es ermöglicht, auf unterschiedliche Instanzen der Datenbank zuzugreifen. Die Kommunikation erfolgt über Requests. Dazu verwendet OrientDB vier Methoden:

`GET`	um Daten aus der Datenbank zu erhalten
`POST`	um neue Daten in die Datenbank zu schreiben
`PUT`	um existierende Daten in der Datenbank zu ändern
`DELETE`	um Daten zu löschen

Ergebnisse werden im JSON-Format zurückgeliefert. Datensatzupdates oder neue Datensätze gehen ebenso im JSON-Format an den Server zurück. (Orient Technologies, 2015) (Zerbst, 2015)

Nachfolgend werden die Java-API sowie der Zugriff mittels SQL-Dialekt detailliert beschrieben. OrientDB stellt drei unterschiedliche Java API's zur Verwaltung von Daten zur Verfügung. *Graph-, Document-* und *Object API*. Im

[23] `http://orientdb.com/docs/last/Indexes.html`, aufgerufen am 2016-01-05

[24] `http://orientdb.com/docs/last/Programming-Language-Bindings.html`, aufgerufen am 2016-01-06

Rahmen der Ausarbeitung, wird jedoch nur die Java Graph-API berücksichtigt. Die nötigen Informationen für die Ausarbeitung der Zugriffskonzepte liefert die von Orient Technologies (2015) frei verfügbare Dokumentation.

Java Graph-API Die OrientDB Graph-API implementiert die TinkerPop BluePrints Schnittstelle. Dabei handelt es sich um eine Sammlung von Interfaces und Implementierungen sowie Test-Suites für das Property-Graph-Model. Analog zu JDBC (Java Database Connectivity) ist BluePrints eine einheitliche Schnittstelle für Graphdatenbanken und erlaubt dadurch, unterschiedliche GDBMS einzubinden [25]. Mit Hilfe der API ist es möglich, den Server aus einer beliebigen Anwendung anzusprechen. Für das Erzeugen und Lesen von Knoten und Kanten stehen Methoden zur Verfügung, welche als Ergebnis eine Referenz des jeweiligen Objektes zurückliefern. Beim Erzeugen wird das jeweilige Objekt mit einem eindeutigen Identifikator versehen. Kanten können nur zwischen bereits existierenden Knoten erzeugt werden, und die Reihenfolge der Kanten gibt die Richtung der Kante an. Die Angabe eines Bezeichners für die Kante ist verpflichtend und besteht aus dem Datentyp `String`. Attribute, sowohl für Kanten als auch für Knoten, können über zur Verfügung gestellte Methoden angelegt, aktualisiert und entfernt werden. Über eigenständige Methoden ist es möglich, inzidente Kanten sowie benachbarte Knoten eines bestimmten Knotens auszulesen. Analog zu den Knotenfunktionen ist es möglich, von angegebenen Kanten den Start- und Endknoten auszulesen. Sowohl Kanten als auch Knoten lassen sich Löschen. Dabei stellt OrientDB referentielle Integrität sicher.

Die Verwaltung von Klassen erfolgt über Methoden, welche von der Java API zur Verfügung gestellt werden. Klassen können unter der Angabe eines Namens vom Datentyp `String` erstellt werden. Nachfolgend lassen sich Attribute und deren Primitive sowie Integritätsbedingungen für die erstellte Klasse festlegen. Die Zuordnung eines Knoten- oder Kantenobjekts zu einer Klasse erfolgt bei der Erstellung des jeweiligen Objekts unter der Angabe des gewünschten Klassennamens. Wurde die Klasse nach einem Schema definiert, sprich sind alle Klassenattribute zwingend, müssen diese zum Zeitpunkt des Erzeugens angegeben werden. Optionale Attribute hingegen können nachträglich angegeben werden.

[25] `https://github.com/tinkerpop/blueprints/wiki`, aufgerufen am 2016-01-08

Die Verwendung der Java API ermöglicht eine beliebige anwendungsseitige Implementierung von Graphalgorithmen. Traversierung wird von der nativen Java API mittels Tiefen- und Breitensuche unterstützt. Durch Einbinden von SQL ähnlichen Abfragen wird mittels der OrientDB SQL Engine, unter Verwendung von OrientDB SQL-Funktionen, die Berechnung von Distanzen und kürzesten Pfaden ermöglicht. Ausreichende Implementierungen von Suchverfahren und Graphalgorithmen stellt die Blueprints Erweiterung *Furnace* [26] zur Verfügung.

CRUD-Operationen mittels Abfragesprache OrientDB-SQL OrientDB verwendet zur Schemadefinition und zu Datenabfragen und -manipulationen einen eigenen SQL-Dialekt. Aufbauend auf den SQL-Standard, wurde SQL für den Umgang mit Graphen optimiert und erweitert. Die Anlehnung von OrientDB an SQL soll den Einstieg und den Umgang mit Graphdatenbanken erleichtern. Nachfolgend soll an einfachen Bespielen der Umgang mit OrientDB beschrieben werden und der Unterschied zu SQL aufgezeigt werden.

Sowohl Kanten als auch Knoten sind in OrientDB Klassen zugeordnet. OrientDB enthält die Basisklassen `V` (Vertex) und `E` (Edge). Die Erstellung von Klassen erfolgt über den Befehl `CREATE CLASS`. Codausschnitt 2.7 zeigt das Erzeugen einer Klasse `Person`, welche die Klasse des Basisknotens erweitert. Die Klasse `V`, für Vertex, repräsentiert alle Knoten innerhalb des Graphen. `CREATE PROPERTY` erzeugt für die Klasse zwei Eigenschaften: `name` und `alter`. Analog zu SQL kann mittels `ALTER` eine Eigenschaft, eine Klasse, ein Cluster oder die Eigenschaften einer Datenbank modifiziert werden. Selbige können durch das Kommando `DROP` gelöscht werden. Die Definition von Kantentypen erfolgt identisch zu den Knoten, sie erben jedoch von der Basisklasse `E`, welche alle Kanten in einem Graphen darstellt.

```
1  CREATE CLASS Person EXTENDS V;
2  CREATE PROPERTY Person.alter INTEGER;
3  CREATE PROPERTY Person.name MANDATORY true;
```

Listing 2.7: Erzeugen einer von der Basisklasse V abgeleiteten Klasse vom Typ `Person`, welche durch die Eigenschaften `alter` und `name` beschrieben wird, mittels OrientDB SQL

[26] `https://github.com/tinkerpop/furnace/wiki`, aufgerufen 2015-12-08

```
1  ALTER PROPERTY Person.alter INTEGER MIN 18;
2  DROP PROPERTY Person.alter;
```

Listing 2.8: Ändern sowie Löschen des zuvor modifizierten Attributes `alter` der Klasse Person, mittels OrientDB SQL

```
1  CREATE CLASS Befreundet_Mit EXTENDS E;
```

Listing 2.9: Erzeugen einer von der Basisklasse E abgeleiteten Klasse vom Typ `Befreundet_Mit`, mittels OrientDB SQL

Im SQL-92 Standard erfolgt die Manipulation von Daten mittels `INSERT`, `UPDATE` und `CREATE`. Angelehnt an diesen Standard werden die Befehle auch in OrientDB zur Verfügung gestellt. `INSERT` erzeugt einen neuen Eintrag in der Datenbank. Das Erzeugen von Knoten und Kanten erfolgt über eigene Befehle: `CREATE VERTEX` und `CREATE EDGE`. Codeausschnitt 2.10 fügt zwei Knoten vom Typ `Person` hinzu und erstellt eine Verbindung zwischen den Beiden. Das Erzeugen einer Kante erfordert die Angabe von Start- und Endknoten. Die Angabe kann direkt erfolgen, sprich unter der Angabe der eindeutigen Identifizierung, oder durch eine geschachtelte Selektion. Der Startknoten der Kante wird direkt mit dem Identifikator `#13:1` und der Endknoten indirekt mit der verschachtelten Selektion angegeben.

```
1  CREATE VERTEX Person
2          SET name="Bart", alter=10;
3  CREATE VERTEX Person
4          SET name="Milhouse", alter=10;
5
6  CREATE EDGE Befreundet_Mit
7          FROM #13:1
8            TO ( SELECT FROM Person WHERE name='Bart')
```

Listing 2.10: OrientDB SQL Code zur Erstellung von Knoten der Klasse `Person` sowie das Erzeugen einer Kante vom Typ `Berfreundet_Mit`, welche beide zuvor erzeugten Knoten verbindet. Die von OrientDB vergebene ID definiert den Startknoten, während der Endknoten über die `WHERE`-Klausel und den Vergleichsoperator selektiert wurde

Die Datenabfrage erfolgt wie in SQL über das `SELECT`-Statement, welche beliebig verschachtelt werden können. Die Selektion kann mittels bekannter `WHERE`-Klausel erweitert werden. Hierzu stellt das Datenbanksystem vergleichende, bool'sche und mathematische Operatoren zur Verfügung. Anfrage

2.11 selektiert alle Attribute von Knoten, die eine eingehende typunabhängige Kante auf die Person `Bart` besitzen. Als Ergebnis wird eine Menge von Knoten zurückgeliefert, die durch die Funktion `expand()` aufgelöst werden.

```
1  SELECT expand(in())
2    FROM Person
3   WHERE name='Bart';
```

Listing 2.11: OrientDB SQL Code zur Selektion von allen eingehenden Kanten des Knoten vom Typ `Person`, dessen Name `Bart` entspricht.

Die Funktion `in()` legt die Richtung der Kante fest und kann durch `out()` oder `both()` ersetzt werden. Unter Angabe eines Kantenbezeichners können Kanten von bestimmten Typ berücksichtigt werden.

OrientDB bietet neben Schemadefinition, Selektion und Datenmanipulation auch die Möglichkeit, über den SQL-Dialekt eine Traversierung durchzuführen. Unter der Angabe eines konkreten Knoten, mittels direkter Angabe oder Selektion, führt der Befehl `TRAVERSE` eine Traversierung durch. Für die Traversierung stehen zwei Algorithmen zur Verfügung: *Depth-First*[27] und *Breadth-First*[28] Strategie. Beide Algorithmen verhindern, dass Knoteninstanzen mehrfach besucht werden. Ausgehend vom Startknoten `#12:0`, traversiert der Codeausschnitt 2.12 alle ausgehenden Kanten und liefert die Menge aller Knoten zurück, deren Abstand zum Startknoten höchstens die Tiefe zwei beträgt. Aus der zurückgelieferten Menge wird nur das Attribut `name` ausgegeben. Die Traversierungsstrategie kann mittels `STRATEGY` angegeben werden. Standardmäßig wird die Tiefensuche verwendet. Die Funktion `out()` kann wie bereits beschrieben durch `in()` oder `both()` ausgetauscht werden. In den drei jeweiligen Funktionen kann eine Menge von Kantenbeschreibungen angegeben werden. Wird keine Beschreibung der Kante angegeben, so werden alle Kanten berücksichtigt.

[27] engl.: Tiefensuche
[28] engl.: Breitensuche

```
1        SELECT name, $depth AS Distanz
2          FROM (TRAVERSE out("Befreundet_Mit")
3                  FROM #12:0 WHILE $depth <= 2
4                STRATEGY BREADTH_FIRST);
```

Listing 2.12: OrientDB SQL Code zur Traversierung über die ausgehende Kante vom Typ `Befreundet_Mit`, ausgehend vom Knoten mit der ID 12, mit der Bedingung, dass die Traversierung solange durchgeführt wird, bis die Tiefe der Zahl zwei entspricht. Als Traveriserungsstrategie wurde die Breitensuche angegeben.

Groß- und Kleinschreibungen müssen bei Schlüsselwörtern und Klassennamen im Gegensatz zu Attributnamen und Abfragewerten nicht berücksichtigt werden. Der wesentliche Unterschied zwischen OrientDB und relationalen Datenbanken ist die Darstellung von Beziehungen. Während relationale Datenbanken Beziehungen über `JOIN's` abbilden, geschieht dies in OrientDB über `LINK's`. Aus diesem Grund unterscheidet sich der SQL-Dialekt vom SQL-Standard. Die gewohnte `JOIN`-Syntax wird nicht unterstützt und zur Auflösung von Beziehung verwendet OrientDB SQL eine "Punkt-Notation". Codeausschnitt 2.13 zeigt ein Beispiel für die Auflösung einer Beziehung in SQL und den dazugehörigen equivalenten Ausdruck in OrientDB SQL.

```
1    -- SQL JOIN
2       SELECT *
3         FROM Person P, Stadt S
4     WHERE P.stadt = S.id
5       AND S.name = 'Rom'
6
7    -- equivalentes Statement in OrientDB SQL
8    SELECT *
9      FROM Person
10     WHERE stadt.name = 'Rom'
```

Listing 2.13: Equivalente OrientDB SQL Operation zu SQL-JOIN, welche alle Personen selektiert, die in der Stadt `Rom` leben (Quelle: nach Orient Technologies (2015)

Während in SQL zur Protektion aller Spalten die Angabe des $*$-Operators verpflichtend ist, kann dieser in OrientDB optional angegeben werden. Wesentlich unterscheiden sich die beiden Abfragesprachen bei der Verwendung von `DISTINCT`. In OrientDB wird `DISTINCT` als Funktion verwendet, anstelle des Schlüsselwortes. Das Schlüsselwort `HAVING` und das Selektieren von mehreren Zielen, in dem Fall Klassen, wird nicht unterstützt. Dies kann jedoch durch die Verwendung von verschachtelten `SELECT`-Statements umgangen werden.

Charakteristik von OrientDB

Zerbst (2015) beschreibt die flexible und elegante Abfrage sowie die geringe Einstiegshürde durch Abfragen mittels gängigem SQL und der frei verfügbaren Dokumentation als eine der Stärken von OrientDB. Positive Aspekte wie Transaktionssicherheit, Verwaltung von Zugriffsrechten, verteilte Datenbanken und die Möglichkeit der direkten Übernahme von Daten aus relationalen Datenbanken wurden des weiteren angeführt. Die Möglichkeit, Daten in strukturierter, unstrukturierter oder semi-strukturierter Form in unterschiedlichen Datenmodellen zu speichern, ermöglicht eine flexible Anwendung der Graphdatenbank. Dank der Apache Lizenz ist OrientDB für kommerzielle Anwendungen nutzbar. (Zerbst, 2015) (Edlich u. a., 2010)

2.6. RDF und RDF Stores

Dieses Kapitel widmet sich dem *Resource Description Framework (RDF)*, welches vom *World Wide Web Consortium (W3C)* entwickelt wurde, um Informationen im Semantic Web auszudrücken. Der erste Teil soll die Grundlagen von RDF näher betrachten, um anschließend die Funktionsweise von *Apache Jena*, einem Framework für die Verarbeitung von RDF-Daten, zu beschreiben.

2.6.1. Resource Description Framework

Das *Resource Description Framework*, kurz RDF, ist ein Framework für den Austausch und die Repräsentation von Informationen im Semantischen Web. Die Grundstruktur der abstrakten Syntax ist eine Menge von Tripeln. Die einzelnen Tripel bestehen aus einem *Subjekt*, einem *Prädikat* und einem *Objekt*, welche oft auch als Statements bezeichnet werden. Subjekte und Objekte stellen Ressourcen da, welche durch das Prädikat beschrieben werden. Mit Hilfe der Statements wird ein RDF-Graphen Modell aufgebaut. Das Datenmodell besteht im wesentlich aus drei Objekt-Typen:

`Ressource` Als Ressource werden einzelne Dinge, die durch RDF beschrieben werden, bezeichnet. Eine Ressource kann beispielsweise eine Webseite oder ein Teil einer Webseite sein. Es sei darauf hingewiesen, dass es sich bei einer Ressource nicht um ein im Internet erreichbares Objekt handeln muss. Es kann auch eine Person oder ein Fahrzeug sein. Jede Ressource wird durch einen "Internationalized Resource Identifier (IRI)" beschrieben. [29] (Fiedler, 2004)

`Eigenschaft` Eine Charakteristik, ein Attribut oder eine Beziehung mit der eine Ressource beschrieben wird; definiert erlaubte Werte und Typen von Ressourcen die durch sie beschrieben werden. (Fiedler, 2004)

`Statement` Eine spezielle Ressource, zusammen mit einer Eigenschaft und einem Wert für die Ressource, wird als RDF-Statement bezeichnet. Genauer gesagt handelt es sich bei den drei Teilen eines Statements um ein Subjekt, ein Objekt und das Prädikat. Ein Objekt eines Statements kann aus einem Literal, einem primitiven Datentyp oder einer weiteren Ressource bestehen. (Fiedler, 2004)

Folgendes Beispiel zeigt eine graphische Repräsentation eines RDF-Modells. Ressourcen, also Subjekte oder Objekte, sind in der graphischen Repräsentation durch Ellipsen dargestellt. Handelt es sich bei einem Objekt um einen Literal, wird dieses als Rechteck, ansonsten weiterhin als Ellipse abgebildet. Die gerichtete Kante beschreibt die Verbindung zwischen Subjekt und Objekt und wird mit dem Prädikat beschriftet. Abbildung 2.8 zeigt, dass die Ressource (Subjekt), in diesem Fall eine Webseite, einen Titel Namens "Die Simpsons" hat. Das Prädikat wird durch die Eigenschaft "Titel" und das Objekt durch den dedizierten Wert "Die Simpsons", welcher als Literal modelliert ist, beschrieben.

Die Knoten eines RDF Graphen, also Subjekt und Objekt, können entweder IRI-Referenzen oder leere Knoten (Blank Nodes) sein. Ein Leerer Knoten kann jedoch entweder nur ein Subjekt oder ein Objekt eines Triples sein. Ein Objekt Knoten kann des Weiteren aus einem Literal bestehen. Leere

[29] http://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/, aufgerufen am 2015-09-03

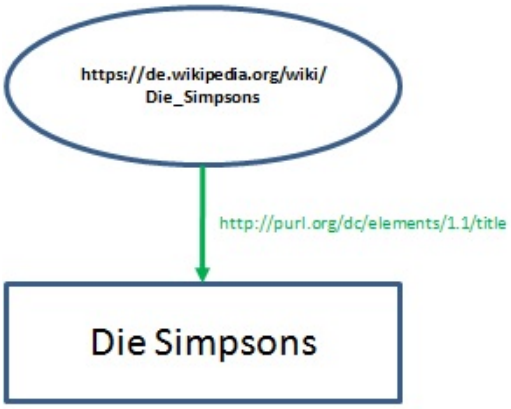


Abbildung 2.8.: Einfacher RDF Graph, welcher das Subjekt als Ellipse, das Prädikat als gerichtete Kante und das Objekt als Rechteck modelliert darstellt

Knoten beinhalten keine Daten. Im Wesentlichen nehmen sie zwei Rollen ein: Objekt des ersten Statements und Subjekt des zweiten Statements.

IRI Referenzen sind eindeutige Identifikatoren und dienen zur maschinellen Identifizierung von Subjekten, Prädikaten und Objekten. Der "Internationalized Resource Identifier (IRI)" ist eine Generalisierung des bereits bekannten URI Konzepts. Jede URI oder URL ist eine IRI. Eine Web-Ressource wird beispielsweise durch eine eindeutige URL identifiziert. IRI liefert den Vorteil einen größeren Bereich von UniCode-Zeichen verwenden zu können. Durch IRIs können unterschiedliche Ressourcen identifiziert werden:

- Dinge und Ressourcen im Web
- Personen, Fahrzeuge, Gegenstände, ... Ressourcen welche nicht im Web erreichbar sind
- Eigenschaften der Ressourcen
- im Generellen Dinge und Ressourcen die beschrieben und identifiziert werden können

Literale sind konstante Werte. Darunter versteht der RDF-Standard, Zeichenketten, Datum oder eine einfache Zahlen. Literale können nur als Objekte in einem RDF Statement verwendet werden. (Cyganiak, Wood und Lanthaler, 2014) (Fiedler, 2004)

2.6.2. RDF Stores

Datenbanken, welche semantische Verbindungen in Form von Subjekt - Prädikat - Objekt speichern können, werden als RDF-Stores oder Triplestores bezeichnet. Sie verwenden zur Speicherung der Daten das RDF-Framework 2.6.1. Dieser Abschnitt soll kurz Konzepte und Ideen hinter RDF-Stores wiedergeben, um anschließend auf den Triplestore - Apache Jena 2.6.3 einzugehen.

Aufgrund der Einfachheit, der Flexibilität und der atomaren Form des RDF-Models ist es möglich strukturierte, semi-strukturierte oder unstrukturierte Daten zu speichern. Zur Strukturierung und Schematisierung werden "Resource Description Framework Schema (RDFS)" und die "Web Ontology Language (OWL)" verwendet. Diese ermöglichen es Ontologien anhand einer Beschreibungssprache zu formalisieren. Busse u. a. (2014) beschreiben eine Ontologie wie folgt: *"die formale Definition von Begriffen und deren Beziehungen als Grundlage für ein gemeinsames Verständnis"*. Als Abfragesprache fungiert oft SPARQL, welche Ähnlichkeiten zu SQL aufweist. Die Entwicklung der Standards wurde von "World Wide Web Consortium (W3C)" vorangetrieben. (Ontotext, 2014)

Für die Datenspeicherung und -repräsentation gibt es laut Stegmaier u. a. (2011) unterschiedliche Ansätze. Die Repräsentation von RDF-Daten kann in unterschiedlichen Formaten erfolgen:

- Notation 3 (N3) ist eine kompakte, aber komplexe Sprache zur Speicherung von Tripeln
- N-Triples wurde aus N3 heraus entwickelt um die Komplexität zu minimieren
- Terse RDF Triple Language (Turtle) wurde entwickelt, um die Ausdrucksstärke von N3 zu erhöhen; wird unter anderem dazu eingesetzt, um Graph-Strukturen in SPARQL zu realisieren.
- RDF/XML die RDF-Tripel werden in einer XML Syntax repräsentiert
- RDF/XML-ABBREV wurde aus der Basis XML Syntax heraus geboren und verwendet eine kompaktere und verkürzte Syntax.

Basierend auf den unterschiedlichen Implementierungsansätzen werden -

laut Stegmaier u. a. (2011) - drei unterschiedlichen Ansätzen für die Speicherung von RDF-Daten unterschieden:

- In-memory: ein bestimmter Anteil des Hauptspeichers wird für die Speicherung von RDF-Daten reserviert; hohe Performance, hat jedoch den Nachteil, dass dieser Ansatz nur für geringe Datenmengen möglich ist.
- Native: im Gegensatz zu *in-memory* werden die RDF-Daten durch geeignete Implementierungen permanent im Dateisystem gespeichert.
- Non-native: auch unter *relational database storage* bekannt - die englische Bezeichnung deutet bereits darauf hin - zur permanenten Speicherung von RDF-Daten werden relationale Datenbanken herangezogen. Dabei unterscheidet man zwei unterschiedliche Strategien. Erstere hält eine Tabelle mit allen RDF-Tripeln. Die zweite Umsetzung speichert die Ontologie in einer dafür erzeugten Tabellen-Struktur; dies führt zu einer großen Anzahl von Tabellen.

2.6.3. Apache Jena

Jena ist ein auf Java basierendes Open-Source Framework zur Entwicklung von Applikationen für das Semantische Web. Jena wurde ursprünglich von HP Labs, einem Teilbereich von Hewlett-Packard, entwickelt und wurde 2010 von der "Apache Software Foundation" übernommen. Tabelle 2.5 liefert einen Überblick über die Systemeigenschaften von Apache Jena. (Apache Software Foundation, 2015)

Das Framework kann unter `https://jena.apache.org/download/index.cgi` in der neusten Version 3.0 heruntergeladen werden und erfordert Java in der Version 8. Das Framework kann nicht eigenständig verwendet werden, sondern stellt Werkzeuge und Java-Bibliotheken zur Verfügung, welche vom Anwender selbst festgelegt werden müssen, um Semantische Web Applikationen und Applikationen für zusammenhängende Daten zu erstellen. Die entsprechenden Java-Bibliotheken mit den nötigen Werkzeugen werden in die Java-Applikation eingebunden. Die Distribution enthält folgende Module:

Entwickler	Apache Software Foundation
aktuelle Version	3.0, August 2015
Datenbankmodell	RDF Store
Lizenz	Apache Lizenz, Version 2.0
Server Betriebssysteme	Linux, OS X, Unix, Windows
Datenschema	ja - RDF-Schemas
APIs	Fuseki (REST-style SPARQL HTTP Interface, Jena RDF API, RIO (RDF Input/Output), SPARQL
Transaktionskonzept	ACID

Tabelle 2.5.: Überblick der Systemeigenschaften von Apache Jena (Quelle: nach solid IT GmbH (2015))

- `jena-core` stellt die Jena RDF APIs, Ontologie APIs und Inference[30] APIs zur Verfügung.
- `jena-arq` SPARQL 1.1, Abfragen und Updates.
- `jena-tdb` skalierbares und performantes Subsystem zu Speicherung für Jena.
- `jena-iri` stellt eine Implementierung für "Internationalized Resource Identifiers (IRIs)" und "Uniform Resource Identifiers (URI)" zur Verfügung

Das Jena Framework besteht aus unterschiedlichen, durch definierte Schnittstellen verbundenen Subsystemen. Abbildung 2.9 zeigt einen Überblick über die Architektur von Apache Jena.

Jenas RDF API bietet Möglichkeiten, um RDF Daten zu erzeugen, zu manipulieren, zu lesen und zu schreiben. Das wesentliche Element zur Verwaltung von RDF Daten in Jena ist das `Model`. Innerhalb des `Models` werden die Daten als `Statements` dargestellt. Das `Model` repräsentiert den gesamten Graphen. Die Zugriffsverwaltung auf Statements und den Graphen, sowie auf die unterschiedlichsten Komponenten erfolgt über die RDF API.

Jena bietet die Möglichkeit, Daten persistent in einer relationalen Datenbank zu speichern und bei Bedarf die Daten zu laden. Dabei unterstützt das Fra-

[30] engl.: Rückschluss

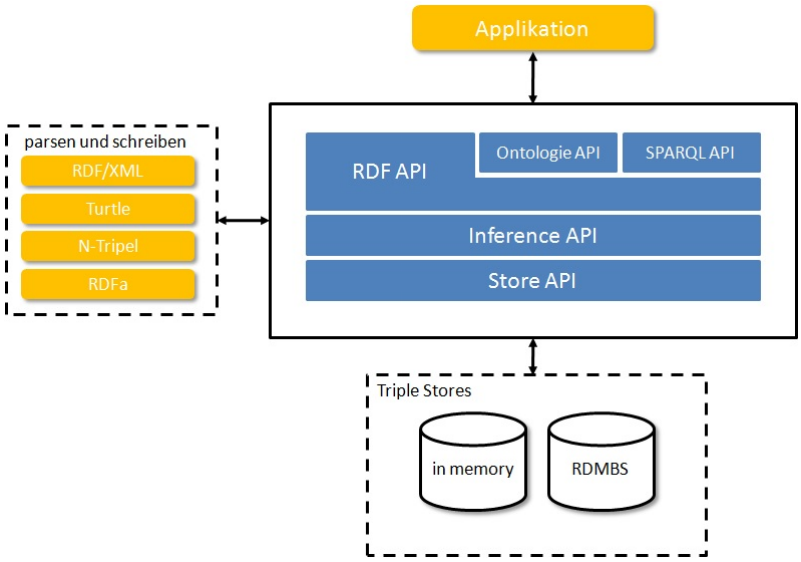


Abbildung 2.9.: Apache Jena Architektur Überblick (Quelle: nach Apache Software Foundation, 2015)

mework unterschiedliche Datenbanksysteme. Zu den bekanntesten zählen MySQL, PostgreSQL und Oracle. Eine vollständige Liste der unterstützten Datenbanksystem kann unter den Dokumentationen auf der Apache Jena Website[31] abgerufen werden. Alternativ zu der Erstellung von persistenten Daten können Daten in den Hauptspeicher (in-memory) geladen werden. Dies bietet einen Geschwindigkeitsvorteil, bringt jedoch den Nachteil des limitierten Speichers mit sich.

Ein wichtiger Bestandteil von Applikationen im Semantischen Web ist das Ableiten/Herleiten von Informationen, welche nicht explizit im Graph angegeben werden. In Jena ermöglicht dies die "Inference API". Jena stellt sogenannte "Inference Engines" und "Reasoner" zur Verfügung. Diese "Engines" und "Reasoner" verarbeiten die von OWL und RDF bereitgestellten Daten über die Semantik. Es werden Beziehungen ausgewertet und die dadurch neu entstandenen Tripel eingefügt. Liefern die vom Framework bereitgestellten Werkzeuge nicht die gewünschte Funktionalität, können wahlweise externe Reasoner eingebunden werden.

[31] `https://jena.apache.org/documentation/sdb/databases_supported.html`, aufgerufen am 2015-09-09

Ontologien im Format RDFS und im ausdruckstärkeren OWL werden von Jena unterstützt. Dafür liefert das "Ontology API" die nötige Funktionalität. Den Umgang mit SPARQL[32], der Abfragesprache für RDF, wird durch die SPARQL API geregelt. (Apache Software Foundation, 2015) (Stegmaier u. a., 2011)

Charakteristik von Apache Jena

Äquivalent zu Graphdatenbanken ist auch in Jena bzw. in RDF Stores die Flexibilität eine der größten Stärken. Aufgrund der Einfachheit des Models lassen sich unstrukturierte, semi-strukturierte bzw. strukturierte Daten speichern. Das Schema lässt sich einfach erweitern, ohne Änderungen vorzunehmen. Im Gegensatz zu NoSQL Datenbanken wie Neo4j und OrientDB verwenden Triple Stores einheitliche Standards. Die Standards reichen von RDF, RDFS, OWL bis hin zur Abfragesprache SPARQL. Ein wesentliche Stärke von Jena im Speziellen ist die Verwendung von Reasoner. Diese ermöglichen es, Beziehungen auszuwerten und durch die Auswertung neu entstandene Tripel einzufügen. Die Möglichkeit in Jena, Daten im Hauptspeicher abzulegen, ermöglicht eine schnelle und effiziente Verarbeitung. Können Daten, aufgrund des Datenumfangs nicht im RAM abgelegt werden, muss auf die persistente Variante zurückgegriffen werden. Durch den großen Umfang an einer Vielzahl von Werkzeugen und Bibliotheken ist der Einstieg mit einer sehr steilen Lernkurve verbunden, die jedoch durch umfangreiche Tutorials und Dokumentationen kompensiert wird. Ist eine Kommunikation über die zur Verfügung gestellte JAVA API nicht gewünscht, erlaubt es der "Apache Jena Fuseki SPARQL Server" über HTTP mit Jena zu kommunizieren. (Apache Software Foundation, 2015) (Ontotext, 2014) [33]

[32] **S**PARQL **P**rotocol **A**nd **Q**uery **L**anguage

[33] `http://www.mkbergman.com/483/advantages-and-myths-of-rdf/`, aufgerufen am 2016-01-11

Entwickler	Oracle Corporation
aktuelle Version	5.6.27, September 2015
Datenbankmodell	Relationales DBMS
Lizenz	Open Source (GPL) und kommerzielle Lizenz
Server Betriebssysteme	Linux, OS X, Windows, Solaris
Datenschema	ja
APIs	JDBC, ODBC, ADO.net
Transaktionskonzept	ACID (abhängig von der verwendeten Datenbankengine - nicht für MyISAM)

Tabelle 2.6.: Überblick der Systemeigenschaften von MySQL (Quelle: nach solid IT GmbH (2015))

2.7. MySQL

Däßler (2013) beschreibt MySQL als ein kompaktes und leistungsfähiges Datenbanksystem, welches auf Grund seines Leistungsumfangs, der Möglichkeit der kostenfreien Nutzung für nicht kommerzielle Zwecke und der Vielfalt an unterstützten Betriebssystemen zu den meist verbreitetsten Datenbanksystemen gehört. Die erste Version von MySQL wurde 1997 vom schwedischen Unternehmen *MySQL AB* veröffentlicht. Im Jahr 2008 wurde das Datenbanksystem von *Sun Microsystems* übernommen, welches 2010 von *Oracle*, dem Marktführer für relationale Datenbanken, gekauft wurde. Derzeit steht MySQL in der Version 5.6.27 als Open-Source Software sowie als Enterprise Edition für kommerzielle Zwecke zur Verfügung. Die Enterprise Edition erweitert die frei verfügbare Community Edition um eine umfangreiche Palette an Funktionalitäten, Werkzeugen und Supportleistungen, welche hohe Skalierbarkeit, Sicherheit und Zuverlässigkeit garantieren. Eine genaue Aufschlüsselung des Funktions- und Supportumfangs der einzelnen Varianten kann unter der Oracle Website [34] abgerufen werden. Tabelle 2.6 liefert einen Überblick über die Charakteristik von MySQL.

Neben dem Support, der exzellenten und umfangreichen Dokumentation nennt Däßler (2013) weitere grundlegende Eigenschaften, die MySQL als

[34] `https://www.mysql.de/products/`, aufgerufen am 2016-10-08

relationales Datenbanksystem attraktiv macht:

- **Leistungsfähige Architektur**, welche Mehrbenutzerzugriffe und eine große Anzahl an gleichzeitigen Datenabfragen unterstützt.
- **Hohe Performance**, im Unterschied zu anderen relationalen Systemen.
- **Kompatibilität**, als Abfragesprache wurde, bis auf wenige Ausnahmen, der SQL-Standard implementiert. Dies ermöglicht eine Portierung auf andere relationale Systeme, die SQL unterstützen.
- **Portabilität**, in nahezu jeder Hardwarekonfiguration betriebsfähig; verfügbar für alle gängigen Betriebssysteme.
- **ODBC-Unterstützung**, Anwendungen können mittels dieser Programmierschnittstele auf den MySQL Datenbankserver zugreifen.
- **Client-Server-Unterstützung**, erlaubt Zugriff von beliebigen Client-Rechnern über das Netzwerk auf den Datenbankserver.
- **Kosten**, duales Lizenzsystem; Open-Source-Lizenz GPL und proprietäre Lizenz; Clientprogramme können kostenlos verwendet werden; im nicht-kommerziellen Bereich kann der MySQL Datenbankserver kostenlos verwendet werden.
- **Flexibilität**, Programmierschnittstellen für unterschiedliche Programmiersprachen.

Die Architektur von MySQL orientiert sich an dem in Kapitel 2.1 beschriebenen Aufbau. Auf dem MySQL-Server, entspricht dem DBMS, werden die Datenbanken erstellt. Für die jeweilige Datenbank können ein oder mehrere Tabellen angelegt werden. Für jede einzelne Datenbank erstellt MySQL physisch einen Ordner auf dem Filesystem des Betriebssystems, in welchen Dateien, die die Struktur und die Daten der einzelnen Tabellen widerspiegeln, abgelegt werden. Der Zugriff auf diese Dateien erfolgt durch sogenannte *Engines*. Da einzelne Tabellen unterschiedliche Typen aufweisen können, sind diese Engines austauschbar. Die Engines regeln und optimieren den Zugriff auf die unterschiedlichen Tabellentypen. Neben *MyISAM* und *InnoDB*, welche zu den bekanntesten Engines gehören, unterstützt MySQL noch weitere. Diese können unter der "Storage-Engine Dokumentation"[35] entnommen werden. (Däßler, 2013) (Oracle Corporation, 2015)

[35] `https://dev.mysql.com/doc/refman/5.6/en/storage-engines.html`, aufgerufen am 2015-10-09

Das Datenmodell, also die logische Organisationsstruktur aller in einer Datenbank gespeicherten Daten, entspricht in MySQL dem in Kapitel 2.2 beschriebenen Datenmodell einer relationalen Datenbank. Es besteht im Wesentlichen aus einer Menge von Tabellen, die aus Spalten und Zeilen bestehen, und deren Beziehungen untereinander über Schlüssel hergestellt werden.

Der Zugriff und die Manipulation der Daten in den Tabellen und der Datenbank selbst erfolgt in MySQL mit der Datenbankabfragesprache *SQL (Structured Query Language)*. SQL erlaubt das Erstellen und Löschen von Datenbanken und Tabellen sowie CRUD-Operationen auf in der Datenbank befindliche Daten. SQL dient somit als Schnittstelle zum Datenbanksystem für den Datenbankentwickler, -administrator und den -benutzer. Die Anweisungen der Abfragesprache werden im Wesentlichen in drei Kategorien unterteilt. Dabei handelt es sich um die Datendefinition, die Datenverarbeitung und die Datenkontrolle. Die Datendefinition befasst sich mit der Struktur und der Definition der Datenbank und ihren Tabellen. Anweisungen hinsichtlich Datenmanipulation und -abfrage fallen in die Kategorie der Datenverarbeitung. Die Datenkontrolle befasst sich im Wesentlichen im Zugriff- und Rechteverwaltung sowie mit der Transaktionskontrolle. Eine Übersicht der einzelnen SQL-Anweisungen kann in der Dokumentation von MySQL bzw. unter der Website von Tutorialspoint [36] gefunden werden.

Für die Evaluierung und Bewertung der Umsetzung von Berechtigungskonzepten, sowie der Auswertung bzw. Bewertung der Effektivität und Effizienz der unterschiedlichen Technologien im Kapitel 5, wurde MySQL als relationale Datenbank ausgewählt. Hauptgrund für die Bewertung einer relationalen Datenbank stellt die Tatsache dar, dass ein RDBMS noch immer die Lösung für den Großteil der Datenbankprobleme ist. Sie haben sich über Jahre etabliert und ermöglichen durch eine große Anzahl von Anbietern und Nutzern sowie Tutorien und der einheitlichen Abfragesprache SQL eine komfortable Nutzung. Graphdatenbanken stellen im Gegensatz dazu vom Aspekt der Verarbeitung eine Minderheit dar.

[36] `http://www.tutorialspoint.com/mysql/`, aufgerufen 2016-01-11

3. Berechtigungskonzepte und Datenbanksicherheit

Berechtigungskonzepte und Zugriffsrechte sollen den Datenbestand vor unbefugtem Zugriff schützen. Die Grundanforderungen der Vertraulichkeit und der Datenintegrität soll durch die datenhaltenden Systeme gewährleistet werden. Das Kapitel beschreibt allgemeine Methoden, um den gesamten oder Teile des Datenbestandes vor unbefugtem Zugriff zu schützen. Abschnitt 3.1 beschreibt die Zugriffskontrolle nach eigenem Ermessen, die sogenannte **Discretionary Access Control (DAC)**. Die Methode der mehrstufigen Sicherheit, **Mandatory Access Control (MAC)**, wird in Abschnitt 3.2 beschrieben. Der Abschnitt 3.3 liefert einen Einblick in die rollenbasierte Zugriffskontrolle, **Role Based Access Control (RBAC)**.

3.1. Benutzerbestimmbare Zugriffskontrolle

Die "benutzerbestimmbare Zugriffskontrolle", auch "Zugriffskontrolle nach eigenem Ermessen" (Discretionary Access Control, DAC), ist eine Benutzer bezogene Zugriffskontrolle und beruht auf dem Konzept der Zuweisung und dem Widerruf von Privilegien. Konkret bestimmt der Eigentümer der Datenbank, in den meisten Fällen der Datenbankadministrator, die Zugriffsrechte der Benutzer, welche auf die gesamte oder Teile der Datenbank zugreifen wollen. Die Verwaltung der Rechte kann mittels einer Zugriffskontrollmatrix (Access Control Matric, ACM) erfolgen, welche die Rechte für jeden Benutzer auf die Objekte verwaltet. Die Zeile der Matrix M entspricht den Subjekten und die Spalten den Objekten. Subjekte sind Benutzer, Programme und ähnliches, welche auf die Subjekte wie Datensätze, Attribute usw. zugreifen. Tabelle 3.1 zeigt ein Beispiel einer Zugriffskontrollmatrix,

Objekt / Subjekt	Tabelle_A	Tabelle_B	Tabelle_C.Attribut_A
Benutzer1	{lesen}		
Bentuzer2		{lesen, schreiben}	{lesen}
Benutzer3		{update}	{schreiben}

Tabelle 3.1.: Zugriffskontrollmatrix zur Verwaltung von Benutzerrechten für Objekte einer relationalen Datenbank, die beschreibt, welcher Benutzer welche Operationen auf die jeweilige Ressource ausführen darf, z.B. darf der Benutzer 1 lesend auf die Tabelle_A zugreifen

angelehnt an Objekte der relationalen Datenbank. Jede Position der Matrix $M(i, j)$ spiegelt die Rechte eines Benutzers i für das Objekt j wider. Bei einer Anfrage des Subjekts werden die Berechtigungen geprüft und je nach Autorisierung die Anfrage abgelehnt oder der Zugriff erlaubt. (Elmasri und Navathe, 2009)

3.2. Vorgeschriebene Zugriffskontrolle

Bei der "vorgeschriebenen Zugriffskontrolle" (Mandatory Access Control, MAC), erfolgt eine Klassifizierung der Subjekte (Benutzer, Programme) und der Datenobjekte in mehrstufige Sicherheitsklassen. Die Einteilung der Sicherheitsklassen erfolgt nach dem vertikalen Informationsfluss (Down und Bottom-Up). Typischerweise erfolgt die Einteilung der Klassen in der Reihenfolge *streng geheim*, *geheim*, *vertraulich* und *öffentlich*, wobei streng geheim als höchste Sicherheitsstufe angesehen wird und öffentlich als die niedrigste.

Die Einteilung der Subjekte und Objekte in Sicherheitsklassen entscheidet darüber, ob ein lesender oder schreibender Zugriff auf ein Datenobjekt erfolgen darf. Ein Subjekt S darf auf ein Objekt O nur dann lesend zugreifen, wenn die Klasse des Subjekts die Klasse des Objekts dominiert. Formell bedeutet dies: $Klasse(S) \geq Klasse(O)$. Dies wird als einfache Sicherheitseigenschaft bezeichnet und verhindert, dass Einsehen von Daten, welche sich über der Sicherheitsklasse eines Benutzers befinden (No-Read-Up). Geschrieben werden darf ein Objekt O nur dann, wenn die Klasse des Objekts

die Sicherheitsklasse des Subjekts dominiert, sprich $Klasse(S) \leq Klasse(O)$. Dies wird als Sterneigenschaft bezeichnet und verhindert das Daten einer höheren Sicherheitsklasse in eine niedere Stufe übertragen werden (No-Write-Down). Eine Missachtung der Sterneigenschaft hätte zur Folge, dass beispielsweise ein Benutzer der höchsten Sicherheitsstufe (geheim) Objekte mit der selben Einstufung kopiert und diese als Objekte mit der niedrigsten Stufe zurückschreibt. Somit wäre das Objekt im gesamten System sichtbar. (Elmasri und Navathe, 2009) (Behr, 2015)

Das vertikale Klassifizierungsschema lässt sich um horizontale Bereiche erweitern. Die Sicherheitsklassen werden als Netz organisiert (Lattice) (Elmasri und Navathe, 2009). Im Falle einer Netzorganisation der Sicherheitsklassen kann ein Zugriff nur dann erfolgen, wenn die Voraussetzungen für den vertikalen und horizontalen Bereich erfüllt sind.

3.3. Rollenbasierte Zugriffskontrolle

In der rollenbasierten Zugriffskontrolle (Role Based Access Control, RBAC) werden dem Subjekt (Benutzer, Programm) im wesentlichen Rollen zugeordnet, welche es dem Subjekt ermöglichen auf ein Objekt zuzugreifen. Die Zugriffskontrolle besteht aus den folgenden Komponenten:

- Subjekt
  einem Benutzer oder Programm, welches über die Rollen Berechtigungen erlangt und diese nutzen kann.
- Rolle
  ein Element, welchem Benutzer und Berechtigungen zugewiesen werden.
- Sitzung
  dynamisches, zeitlich begrenztes Element, welches die Verwendung mehrerer Rollen gleichzeitig ermöglicht.
- Objekt
  Ressourcen die durch bestimmte Berechtigungen geschützt werden.
- Operation
  beschreibt den Objektzugriff, beispielsweise lesen, schreiben, modifizieren.

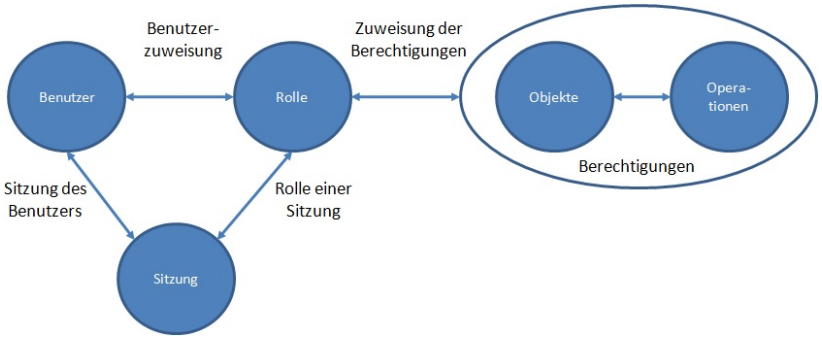


Abbildung 3.1.: Kernmodell des rollenbasierenden Berechtigungskonzeptes, welches Benutzern durch Rollen die Berechtigungen auf Ressourcen zuweist.

- Berechtigung
  ist eine Kombination aus Objekt und Operation.

Die einzelnen Komponenten bilden das in Abbildung 3.1 gezeigte Kernmodell der rollenbasierten Zugriffskontrolle. Die Subjekte bekommen Rollen zugeordnet. Ein Subjekt kann über mehrere Rollen verfügen und umgekehrt. Die Berechtigungen, welche den Rollen zugeordnet werden, ergeben sich aus der Kombination von Objekten und Operationen. Zur Verrichtung der Arbeit eines Subjekts eröffnet dieses eine Sitzung, wobei jedem Subjekt mehrere Sitzungen zugewiesen werden können, aber nur genau ein Subjekt einer Sitzung angehört. In Abhängigkeit seiner Aufgaben wendet das Subjekt innerhalb der Sitzung seine Rollen an um auf das Objekt zuzugreifen. (Tsolkas und Schmidt, 2010)

3.4. Berechtigungskonzepte in Datenbanken

Dieser Teilbereich des Kapitels soll einen Überblick der umsetzbaren Berechtigungskonzepte in den Graphdatenbanken Neo4j und OrientDB sowie in der relationalen Datenbank MySQL verschaffen. Dabei werden die Berechtigungskonzepte der Datenbanksysteme kurz vorgestellt und beschrieben.

3.4.1. Berechtigungskonzepte in Neo4j

Die Graphdatenbank von Neo Technology bietet "out-of-the-box" keine Möglichkeit, ein Berechtigungskonzept umzusetzen [1]. Der Schutz vor unbefugtem Zugriff soll laut Neo4j auf höheren Ebenen des Systems durchgeführt werden[2].

Die Umsetzung eines Berechtigungskonzeptes kann nur direkt am Datenbestand erfolgen. Dazu liefert Neo4j einen generischen Ansatz, basierend auf Zugriffskontrolllisten (Access Control Lists, ACL)[3]. Eine Zugriffskontrollliste regelt, welche Subjekte in welchem Umfang auf Objekte zugreifen können. Dazu werden für Benutzer und Benutzergruppen eigene Knotentypen eingeführt. Ein Benutzer kann mehreren Benutzergruppen angehören, welche durch eine Beziehung miteinander verbunden sind. Den Zugriff auf eine Ressource regelt eine Verbindung zwischen dem Benutzerknoten und dem Knoten der Ressource. Die Zugriffsberechtigungen werden in der Beziehung als Attribut gespeichert und regeln, ob ein Benutzer/Benutzergruppe auf die Ressource zugreifen darf, und in welchem Umfang. Abbildung 3.2 zeigt vereinfacht den generischen Ansatz einer Zugriffskontrollliste in Graphen.

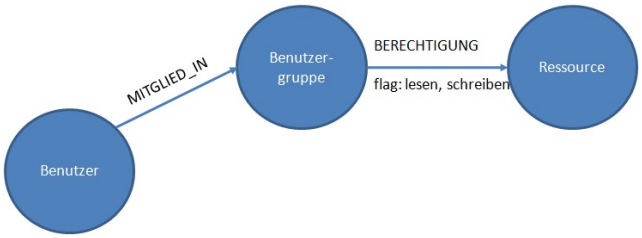


Abbildung 3.2.: Vereinfachte Darstellung der Umsetzung eines Berechtigungskonzepts mittels Zugriffskontrolllisten in Neo4j. Die Eigenschaften der Kanten regeln den Zugriff auf die Ressourcen.

[1] http://db-engines.com/de/system/Neo4j, aufgerufen am 2016-01-28

[2] http://neo4j.com/docs/stable/capabilities-data-security.html, aufgerufen am 2016-01-28

[3] http://neo4j.com/docs/stable/examples-acl-structures-in-graphs.html, aufgerufen am 2016-01-28

3.4.2. Berechtigungskonzepte in OrientDB

Das Sicherheitsmodell von OrientDB basiert auf der rollenbasierten Zugriffskontrolle. Eine Datenbank hat eine Menge von Benutzern, welchen eine oder mehrere Rollen zugewiesen werden. Eine Rolle bestimmt, welche Operationen ein Benutzer auf einer Ressource durchführen darf. Dies hängt in OrientDB von dem definierten Arbeitsmodus und den Regeln einer Rolle ab. Die Regeln werden je nach Arbeitsmodus unterschiedlich interpretiert, sprich sie werden inkludiert bzw. exkludiert. OrientDB unterscheidet dabei zwei Arbeitsmodi, **ALLOW ALL BUT (Rules)** und **DENY ALL BUT (Rules)**. Konkret bedeutet dies, dass eine definierte Rolle alle Operationen auf eine Ressource durchführen darf, außer die festgelegten Ausnahmen, et vice versa, es dürfen nur Operationen durchgeführt werden, welche über Regeln festgelegt wurden. Bei den unterstützten Operationen handelt es sich um die klassischen CRUD-Operationen (Create, Read, Update, Delete), also Hinzufügen, Lesen, Aktualisieren und Löschen.

Das Berechtigungskonzept von OrientDB erlaubt Einschränkungen hinsichtlich unterschiedlicher Ressourcen. So sind Einschränkungen auf gesamte Klassen und Cluster, aber auch differenziertere Einschränkungen auf Satzebene möglich. Dem Benutzer ist es somit nicht möglich, auf bestimmte Datensätze bzw. im Fall eine Graphenrepräsentation, auf bestimmte Knoten zuzugreifen. Restriktionen müssen dediziert für jeden Benutzer und Datensatz angegeben werden[4].

3.4.3. Berechtigungskonzepte in MySQL

MySQL verfolgt das Prinzip einer "benutzerbestimmbaren Zugriffskontrolle". Dadurch wird ein auf Benutzer bezogenes, feingranulares Berechtigungskonzept bis hin zur Attributebene ermöglicht. Rollen und Benutzergruppen werden von MySQL nicht unterstützt. Das Sicherheitskonzept in MySQL erfolgt laut Kofler (2012) in zwei Schritten, wobei durch die Privilegien die Berechtigungen bezüglich der Ressourcen geregelt werden.

[4] `http://orientdb.com/docs/2.1/Database-Security.html`, aufgerufen am 2016-01-29

- Login: Benutzer oder Client-Programm wird überprüft.
- Privilegien: definieren, welche Operationen pro Benutzer ausgeführt werden dürfen oder nicht.

Ein feingranulares Berechtigungskonzept wird laut Kofler (2012) in der Praxis in den seltensten Fällen umgesetzt. Jedoch ist es in MySQL möglich, die Zugriffsrechte für Benutzer auf Datenbanken, Tabellen und Attribute ab zu stufen. Die einzelnen Privilegien eines Benutzers, regeln welche Operationen auf den einzelnen Ressourcen durchgeführt werden dürfen. Die Privilegien reichen von einfachen CRUD-Operationen bis hin zu administrativen Rechten. Eine Übersicht der einzelnen in MySQL unterstützen Privilegien liefert das Online-Handbuch[5].

Die Zugriffsrechte werden in einzelne Ebenen gegliedert. Abbildung 3.3 zeigt die einzelnen Ebenen der Zugriffsrechte, sowie die dazugehörigen Systemtabellen, in welchen die Verwaltung der Benutzerrechte erfolgt. Die erste Stufe zeigt die Benutzer und ist verantwortlich für den Verbindungsaufbau. Auf der Datenbankebene werden die Berechtigungen der Benutzer für die einzelnen Datenbanken festgelegt. Die Ebene der Tabelle und Attribute definiert die Privilegien für den Zugriff auf den Datenbestand.[6]

Privilegien einer übergeordneten Ebene sind in untergeordneten Ebenen gültig und können dort nicht entzogen werden (Kofler, 2012).

Eine Übersicht der in den beschriebenen Datenbanksystemen umgesetzten Berechtigungs- und Zugriffskonzepte liefert die Tabelle 3.2.

3.5. Berechtigungskonzepte in NoSQL und Graphdatenbanken

Eine feingranulare Zugriffskontrolle, stellt Graphdatenbanken bzw. NoSQL-Lösungen vor eine große Herausforderung. Das Problem liegt darin, Daten

[5] `http://dev.mysql.com/doc/refman/5.7/en/privileges-provided.html`, aufgerufen am 2016-01-29

[6] `http://dev.mysql.com/doc/refman/5.7/en/`, aufgerufen am 2016-01-29

Berechtigungskonzept	Datenbank		
	Neo4j	OrientDB	MySQL
DAC			
MAC		X	
RBAC			X

Tabelle 3.2.: Übersicht der umgesetzten Berechtigungs- und Zugriffskonzepte in den Graphdatenbanken Neo4j und OrientDB, sowie in der relationalen Datenbank MySQL

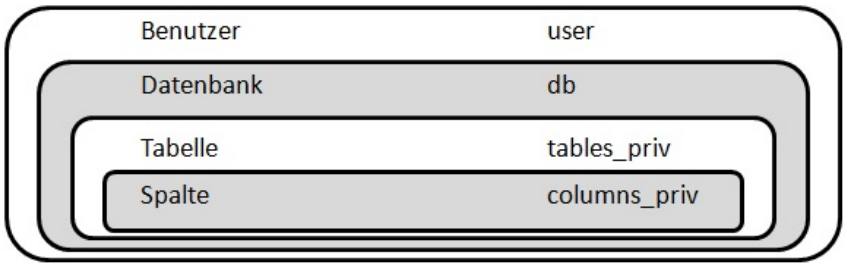


Abbildung 3.3.: Ebenen der Zugriffsrechte und dazugehörige Systemtabellen zur Verwaltung von Benutzerrechten in MySQL

vor unbefugten Benutzern zu schützen, welche keinen Zugriff auf individuelle Daten besitzen. Herkömmliche Berechtigungskonzepte stoßen dabei an ihre Grenzen (Sahafizadeh und Nematbakhsh, 2015).

Cloud Security Allience (2013) bezeichnen eine feingranulare Zugriffskontrolle als eine der zehn größten Herausforderungen in Big-Data. Der Online-Artikel für Datenvirtualisierung und Graphdatenbanken[7] beschreibt, dass Zugriffskontrollen nicht zu den Stärken von Graphdatenbanken zählen. Datenvirtualisierung, also eine Schicht zwischen Datenbank und Applikation kann eine vielseitige Zugriffskontrolle auf Basis von Knotentypen, Attributen usw. zur Verfügung stellen. Cloud Security Allience (2013) beschreibt, dass die Umsetzung der Zugriffskontrolle hauptsächlich auf Ebene der Applikation umgesetzt wird.

Zugriffskontrollen, welche in der Datenbank umgesetzt werden, spiegeln im Großteil das rollenbasierte Berechtigungskonzept wider. Durch die Schemafreiheit erlaubt die Graphdatenbank eine Abbildung großer Mengen von dicht vernetzten, untereinander verbunden Benutzern, Rollen und Ressourcen und somit die Realisierung einer Zugriffskontrolle. Das Konzept erlaubt Top-Down und Bottom-Up Abfragen hinsichtlich der Zugriffsicherheit. Konkret kann ermittelt werden, welcher Benutzer auf welche Ressource Zugriff hat (Top-Down) bzw. wer kann ausgehend von einer gegebenen Ressource die Berechtigungen ändern (Bottom-up). (Robinson, Webber und Eifrem, 2013)

Die Realisierung von Berechtigungen über Benutzer und Rollen kann zum Problem führen, dass Knoten mit einer großen Anzahl an Beziehungen entstehen, wodurch eine Traversierung erschwert wird[8]. Der alternative Lösungsansatz für die Umsetzung eines Zugriffskonzepts wäre die Speicherung der Berechtigung individuell an jedem Knoten. Rechte des Benutzers

[7] `http://www.datavirtualizationblog.com/thinking%2Doutside%2Dthe%2Dgraph%2Ddata%2Dvirtualization%2Dand%2Dgraph%2Ddatabases/`, aufgerufen am 2016-01-31

[8] in einem Kommentar zur Diskussion über die Umsetzung von Berechtigungen: *"The down side is that eventually you will run into a dense node problem, which is a node with too many relationships making traversal across it very hard. But you'll only run into this when you start having a node with relationships past a several hundred thousand"* - `http://stackoverflow.com/questions/17411071/permissions-to-be-stored-as-a-node-or-a-property`, aufgerufen am 2016-01-31

würden somit mit den Berechtigungen der einzelnen Knoten verglichen werden.

Durch die Realisierung der Berechtigungen über Attribute wird jedoch das von De Virgilio, Maccioni und Torlone (2014) beschriebene Ziel der Modellierungsstrategie für Graphdatenbanken - die Minimierung der Datenbankzugriffe während der Traversierung, missachtet. Jeder Knoten der bei der Traversierung durchlaufen wird, muss auf seine Berechtigungen überprüft werden. Dies bringt einen Zugriff auf das Berechtigungsattribut eines einzelne Knoten mit sich.

4. Implementierung

Aus einer Vielzahl von Graph-Datenbankmanagementsystemen (Graph DBMS) wurden zwei Datenbanklösungen ausgewählt, welche für die technische Umsetzung verwendet wurden. Ziel der Implementierung ist die Umsetzung eines feingranularen Berechtigungskonzepts für die ausgewählten Datenbanklösungen, welches es ermöglicht, Testdatensätze zu importieren und in etablierten Datenbank-Technologien abzubilden. Die Software bietet des weiteren die Funktionalität, lesend auf die Datenbanken zu zugreifen und nach Inhalten in der jeweiligen Datenbank zu suchen.

Die Funktionalität "Lesen" und "Suchen" kann unter der Berücksichtigung von festgelegten Berechtigungen durchgeführt werden oder zur Gänze ohne definierte Berechtigungen erfolgen. Das Berechtigungskonzept erlaubt es somit, nur Daten zu sehen, für die der Nutzer berechtigt ist. Dies bedeutet im Falle von einer Suche ohne Berücksichtigung definierter Berechtigungen in einer Graphstruktur, dass der Ausgangsknoten nur dann auf den mit ihm direkt verbundenen Endknoten zugreifen darf, wenn es eine Überschneidung von Berechtigungen zwischen den jeweiligen Knoten gibt. Wird eine Suche mit definierten Berechtigungen initiiert, wird eine Überschneidung der Berechtigungen eines Ausgangsknoten und dessen direkt und indirekt verbundenen Knoten gegen die definierten Berechtigungen geprüft. Das Ergebnis einer Suche entspricht einem Graphen, welcher aus Knoten besteht, die ausgehend von einem Ausgangsknoten aufgrund der Berechtigungen eingesehen werden darf. Konkret wird vom Ausgangsknoten eine Route über die Beziehungen und Knoten verfolgt, bis die Berechtigungen eines Knotenpaares einer Kante nicht mehr übereinstimmen bzw. sich die Berechtigungen eines Knoten nicht mehr mit den definierten Berechtigungen decken. Das Kapitel 4.3 liefert einen genauen Überblick über die Umsetzung und Funktionsweise des Berechtigungskonzepts in Graphen. Anhand eines Evaluierungsschwerpunktes soll die Leistungsfähigkeit der einzelnen

Datenbank-Technologien sowie die Umsetzungsmöglichkeiten von Berechtigungskonzepten festgestellt werden. Durch sorgfältiges Vergleichen der einzelnen Graph DBMS hinsichtlich ihrer Funktionalität und Verwendbarkeit fiel die Wahl auf die Graphdatenbanklösungen von Neo4j und OrientDB. Ausschlaggebend für die Auswahl von Neo4j und OrientDB war, dass beide DBMS laut Ranking von DB-Engines [1] zu den populärsten Datenbanken zählen und sie über die Möglichkeit einer Java Programmierschnittstelle verfügen. Im Zuge der Implementierung wird neben den Graphdatenbanken MySQL als etablierter Vertreter von relationalen Datenbanken mit einbezogen.

Die Implementierung erfolgte ausschließlich in Java. Die Testdatensätze werden in einer relationalen Datenbank gespeichert, welche in weiterer Folge als Ausgangsdatenbank bezeichnet wird. Die Ausgangsdatenbank ermöglicht es, einen Block aus unterschiedlichen, miteinander verbundenen Knotentypen auszulesen. Die ausgelesenen Daten werden in einer allgemeinen Datenstruktur gespeichert, welche die Daten als Graph repräsentiert. Die Datenstruktur ermöglicht eine variable Verwendung der unterschiedlichen Datenbanksysteme. Ein Austausch der Quellenanbindungen ist möglich, sprich Daten können aus Datenbanken unterschiedlichen Typs gelesen und geschrieben werden.

4.1. Datensatz

Der von Sinha u. a. (2015) veröffentlichte Datensatz "Microsoft Academic Graph" dient als Datenbasis. Dabei handelt es sich um einen Graphen, welcher Knoten unterschiedlichen Typs verbindet. Er enthält Daten aus dem wissenschaftlichen Bereich. Dazu zählen Publikationen und deren Zitierung in weiteren Publikationen, sowie Metainformationen bezüglich deren Forschungsgebiete, Autoren, Institutionen, deren Veröffentlichungen bzw. Präsentation in Journals und Konferenzen. Die Daten sind in separaten Textdateien gespeichert. Die Textfiles stellen Knoten unterschiedlichen Typs

[1] `http://db-engines.com/de/ranking/graph+dbms`, aufgerufen am 2016-03-20

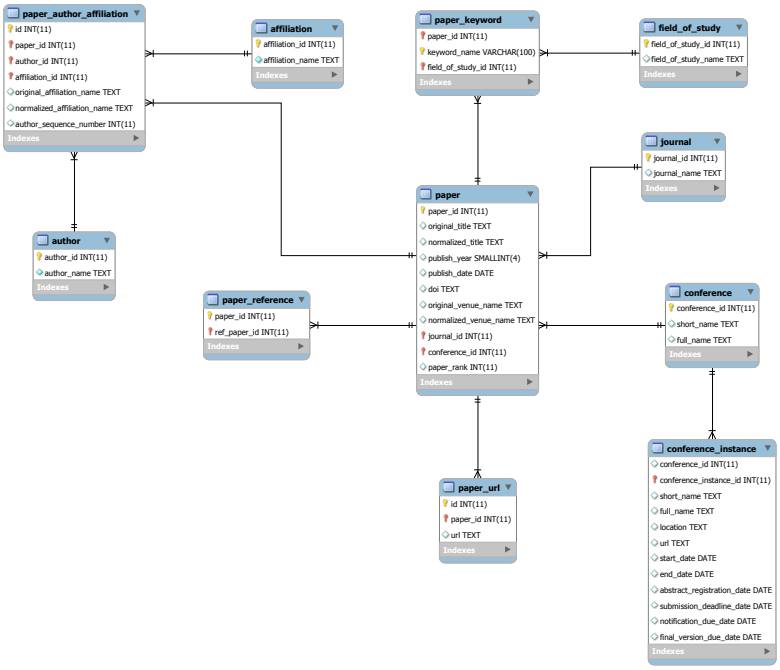


Abbildung 4.1.: Entity-Relationship-Modell zur Speicherung des gesamten Microsoft Academic Graph in der relationalen Datenbank

dar. Der Datensatz steht via HTTP zum Download verfügbar[2]. Für die Implementierung wurde der Datensatz von 06. November 2015 verwendet.

Die Daten der Textdateien werden in die relationale Datenbank gespeichert. Dabei entspricht eine einzelne Textdatei einer Relation, die Zeilen einem Datensatz in der jeweiligen Relation und die durch Tabulatoren in der Textdatei getrennten Zeichenfolgen den Attributen einer Tabelle. Das Entity-Relationship-Modell (Abb. 4.1) zeigt die notwendigen Relationen zur Speicherung der Testdatensätze in der relationalen Datenbank. Die Gesamtgröße der Datenbank entspricht ~166 Gigabyte.

Tabelle 4.1 liefert einen Überblick über die Anzahl der einzelnen Datensätze der jeweiligen Relationen. Fehlerhafte Datensätze, welche nicht interpretier-

[2] `http://research.microsoft.com/en-us/projects/mag/`, aufgerufen am 2016-01-31

Entität	Beschreibung	Anzahl
affiliation	Institutszugehörigkeit	19.849
author	Autoren	119.892.201
field_of_study	Forschungsbereich	53.834
journal	Zeitschriften	23.568
conference	Konferenzen	1275
conference_instance	einzelne Tagungen einer Konferenz	50.202
paper_url	Webseiten der Publikationen	349.947.403
paper_keyword	Schlüsselwörter der Publikationen	157.052.442
paper_reference	Referenzen der Publikationen	952.364.264
paper_author_affiliation	Verbindung zwischen Publikation, Autor und Institut	312.177.278
paper	Publikationen	120.887.883

Tabelle 4.1.: Entitäten und Anzahl der eingefügten Datensätze des Microsoft Academic Graph

bare Zeichen enthalten, wurden beim Einlesen verworfen.

4.2. Datenmodell

Das Datenmodell legt die Struktur der eingelesenen Daten in Form eines Graphen fest. Abbildung 4.2 zeigt die Repräsentation der Daten als Graphstruktur. Ellipsen stellen einen Knoten und ein abgerundetes Rechteck die Eigenschaften des jeweiligen dar. Knoten und Kanten sind mit einem Label versehen. Es ist auch möglich, Kanten Eigenschaften zuzuweisen. Die Eigenschaften einer Kante sind als Aufzählungspunkte unter der Kante modelliert. Berechtigungen der einzelnen Knoten sind als Eigenschaften direkt am Knoten modelliert. Zur Speicherung der Daten in Graphdatenbanken, wie Neo4j und OrientDB, kann das Datenmodell unverändert übernommen werden.

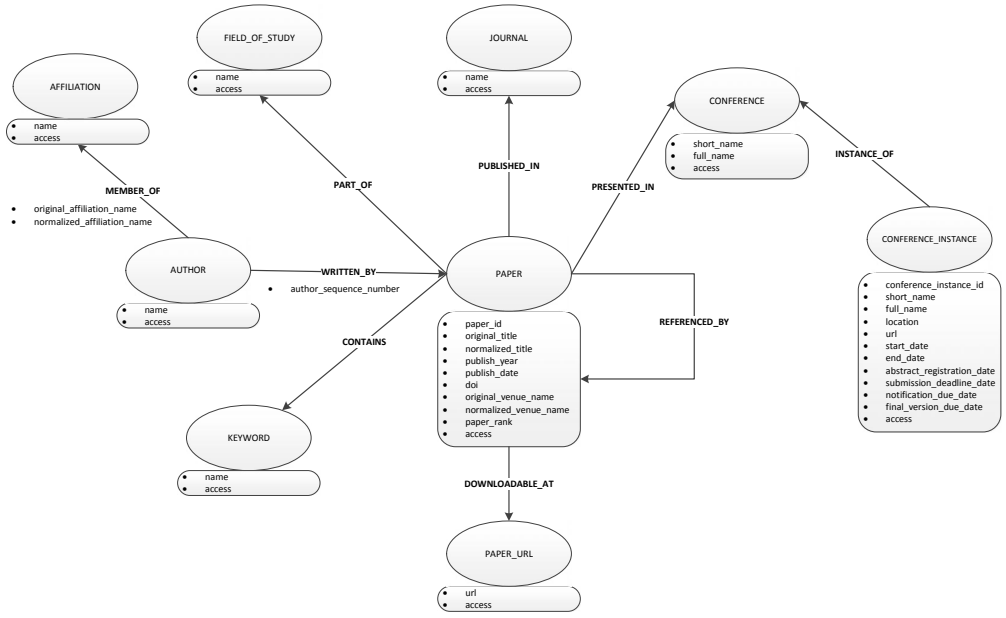


Abbildung 4.2.: Datenmodell zur Abbildung der eingelesenen Testdaten in den Evaluierungsdatenbanken

Angelehnt an die Graphstruktur wurde das Datenbankmodell für die relationale Datenbank modelliert. Die Abbildung 4.3 zeigt ein Entity-Relationship-Modell zur Beschreibung des Datenbankmodells. Die neun unterschiedlichen Knotentypen werden jeweils als ein Entitätstyp mit den jeweiligen Attributen realisiert. Die Auflösungstabellen spiegeln die Beziehung zwischen den einzelnen Knoten wieder. So stellt beispielsweise die Tabelle `author_affiliation` die Verbindung zwischen einem Autor und einer Mitgliedschaft dar. Das Berechtigungskonzept wurde in eigene Entitätstypen ausgelagert, da die Berechtigungen als Attribut nicht verglichen werden können. Die Tabelle `access` speichert die gesamten Berechtigungen. Sämtliche Entitätstypen wie `author`, `paper`, `journal`, etc. besitzen eine eigene Berechtigungstabelle, welche die Berechtigung einer einzelnen Entität speichert. Dadurch können Übereinstimmungen zwischen Berechtigungen verschiedener Typen überprüft werden.

4.3. Berechtigungskonzept

Das Berechtigungskonzept soll die eingeschränkte Nutzung von Ressourcen regeln. Im Fall von Daten in Form einer Graphstruktur dürfen nur Knoten eingesehen werden, welche die geeigneten Zugriffsberechtigungen besitzen. Die Realisierung der Zugriffskontrolle in einer Graphstruktur erfolgt über Berechtigungen, welche direkt an jedem Knoten gespeichert werden. Jeder Knoten erhält eine Menge an Berechtigungen, welche im Attribut `access` gespeichert werden (Abb. 4.2). Das Attribut repräsentiert eine Liste von Zugriffsberechtigungen, in welcher eine Berechtigung genau einmal vorkommen darf. Erfolgt ein Zugriff auf eine Ressource, welche durch einen Knoten repräsentiert wird, ohne definierte Berechtigungen, so darf der Ausgangsknoten A nur dann auf den mit ihm verbundenen Endknoten E zugreifen, wenn die Schnittmenge der Berechtigungen aus A und E keine leere Menge ergibt. Erfolgt eine Operation unter Berücksichtigung von definierten Berechtigungen, so darf auf den Startknoten und den mit ihm direkt und indirekt verbundenen Endknoten nur dann zugegriffen werden, sofern eine Schnittmenge zwischen den definierten Berechtigungen und den Berechtigungen der einzelnen Knoten existiert.

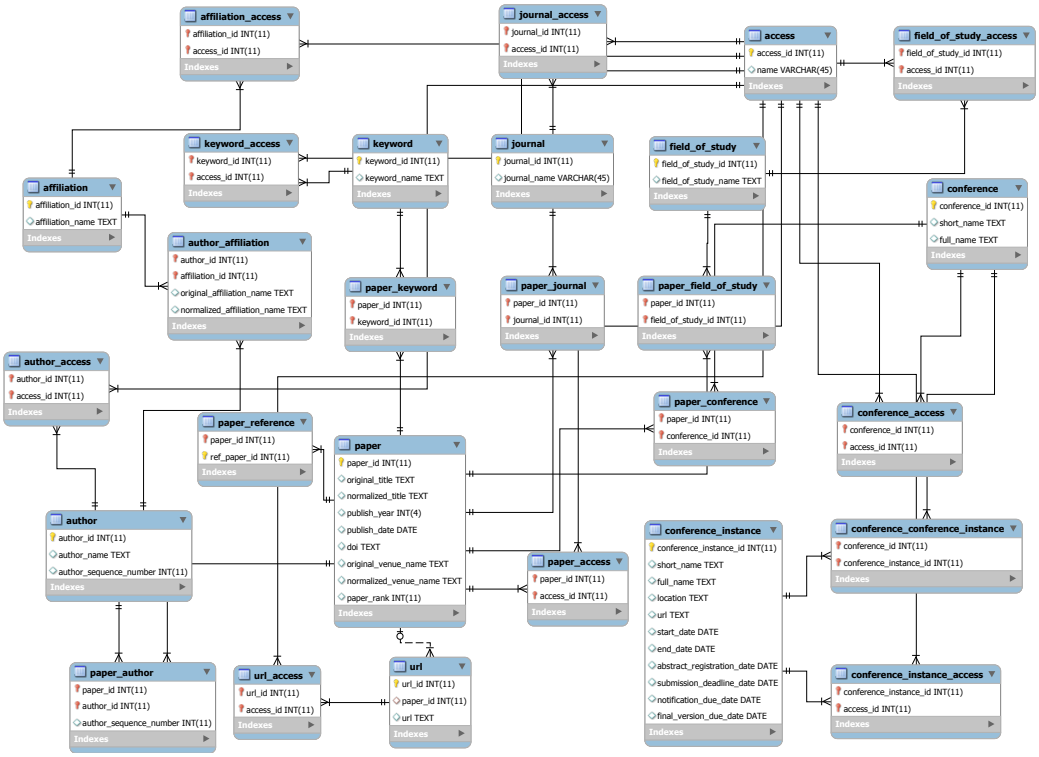


Abbildung 4.3.: Entity-Relationship-Modell zur Beschreibung des Datenbankmodells für die Speicherung der Testdatensätze als Graphstruktur in einer relationalen Datenbank

Abbildung 4.4.: Zerlegen eine Webseite in Teilbereiche für die Umsetzung eines Berechtigungskonzepts für Daten in Form einer Graphstruktur

Die Schnittmenge zweier Mengen ist die Menge aller Elemente, die sowohl in A als auch in E vorkommen: $A \cap E = \{x \mid x \in A \wedge x \in B\}$.

Die Webseiten der Publikationen und die der Konferenzinstanzen bilden die Grundlage der Berechtigungen. Eine Webseite wird in ihre Teilbereiche, ausgehend vom Protokoll bis hin zur Top-Level-Domain, zerlegt. Abbildung 4.4 zeigt das Zerlegen einer Webseite bis zum Teilbereich der Top-Level-Domain. Verzeichnisse welche der Top-Level-Domain folgen, werden ignoriert. Die zerlegten Teilbereiche ergeben die Berechtigungen.

Ausgehend von den Webseiten der Publikation und der Webseite einer Konferenzinstanz werden die Berechtigungen für die weiteren Knoten erstellt. Aus Gründen der Einfachheit wird die Erzeugung der Berechtigungen für die einzelnen Knoten am konkreten Beispiel des Datenmodell (Abb. 4.2) erklärt. Die Berechtigungen für den Knoten `PAPER` werden aus den Webseiten der `PAPER_URL`-Knoten erzeugt. Die mit dem `PAPER` direkt verbundenen Knoten, außer der Knoten `CONFERENCE`, erben die Berechtigungen der jeweiligen Publikation. `AFFILIATION`-Knoten erben über die direkte Verbindung zu Autoren. Die Zugriffsberechtigung von `CONFERENCE`-Knoten erfolgt über die als Attribut gespeicherte Webseite einer Konferenzinstanz. Ein `CONFERENCE`-Knoten erbt nicht die Berechtigung einer Publikation.

Besitzt ein Ausgangsknoten A mehrere direkte Verbindungen zu den Endknoten $E_1, E_2, ...E_n$, ergeben sich seine Berechtigungen B_A aus der Vereinigungsmenge der Berechtigung der einzelnen Knoten $B_{E_1}, B_{E_2}, ...B_{E_n}$. Formell bedeutet dies: $B_A = B_{E_1} \cup B_{E_2} ... \cup B_{E_n} = \{x \mid x \in B_{E_1} \vee x \in B_{E_2} ... \vee x \in B_{E_n}\}$

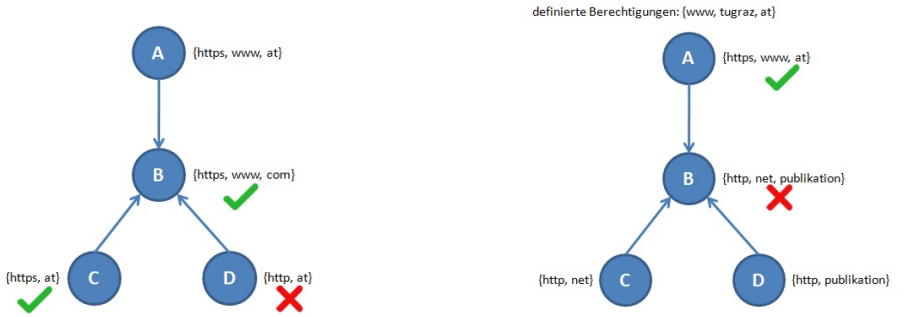
Die Abbildung 4.5 beschreibt an konkreten Beispielen die Funktionsweise

des Berechtigungskonzepts in Graphen. Ausgehend vom Startknoten `A` zeigt die Grafik 4.5a die Ermittlung des Endergebnisses ohne definierte Berechtigungen. Es werden alle Routen durchlaufen, solange die Schnittmenge der Berechtigungen keiner leeren Menge entspricht. Die Menge der Berechtigungen des Knoten `A` geschnitten mit den Mengen der Berechtigungen von `B` ergibt die Ergebnismenge *https*, *www*. Ausgehend von Knoten `B` werden die nächsten Berechtigungen überprüft. Die Schnittmenge der Berechtigungen `B` und `C` ergibt eine Menge welche als Element *https* enthält und somit mindestens eine gültige Berechtigung. Der Durchschnitt aus den Mengen von `B` und `D` ergibt eine leere Menge, wodurch der Knoten `D` nicht in das Ergebnis aufgenommen wird. Der daraus entstandene Teilgraph, in diesem Fall `A, B, C` entspricht jenen Knoten, auf welche der Ausgangsknoten zugreifen darf. Abbildung 4.5b zeigt die Ermittlung eines Suchergebnisses unter der Berücksichtigung von definierten Berechtigungen. Die Berechtigungen *www*, *tugraz*, *at* definieren die Restriktionen. Der Knoten `A` bestimmt den Ausgangspunkt. Die Berechtigungen des Ausgangsknoten werden mit der Menge der definierten Berechtigungen geschnitten. Ergibt der Schnitt mindestens eine gültige Menge, werden die direkt und indirekt mit ihm verbundenen Knoten weiter auf eine Übereinstimmung der Berechtigungen geprüft. Im konkreten Beispiel wird der Knoten `B` geprüft, welcher keine Übereinstimmung zurückliefert und dadurch die mit ihm verbundenen Knoten nicht weiter geprüft werden. Der Knoten `A` entspricht dem Teilgraph, auf welchem zugegriffen werden darf.

4.4. Architektur

Bei der Implementierung handelt es sich um eine clientseitige Java Anwendung welche über geeignete Treiber mit den jeweiligen Datenbankservern kommuniziert. Der Verbindungsaufbau zur relationalen Datenbank sowie die Kommunikation mit dem Datenbankserver erfolgt mit dem MySQL JDBC Treiber. Neo4j und OrientDB werden im eingebetteten Modus betrieben. Dazu wurden die nötigen Bibliotheken von Neo4j und OrientDB in die Java-Anwendung mittels Maven[3] eingebunden. Die Architektur wird in

[3] `https://maven.apache.org/`, aufgerufen am 2016-01-18



(a) ohne Berechtigungen (b) mittels Berechtigungen

Abbildung 4.5.: Funktionsweise des Berechtigungskonzepts ohne definierte Berechtigungen und unter der Berücksichtigung von Berechtigungen anhand der Ermittlung der Schnittmenge der einzelnen Knoten in einem Graphen

zwei Schichten unterteilt: die Quellenanbindung und die Logik. Abbildung 4.6 liefert eine vereinfachte Darstellung der Architektur. Der Importer ist die essentielle, Komponente um Daten aus den im Kapitel 4.1 beschriebenen Textdateien einzulesen und so aufzubereiten, dass diese in den unterschiedlichen Technologien gespeichert werden können. Des weiteren erfolgt über den Importer das Speichern der Testdatensätze inklusive der Berechtigungskonzepte, sowie das Lesen und Suchen in den unterschiedlichen zu evaluierenden Datenbanksystemen. Die Quellenanbindung dient als Anbindung zu den einzelnen Datenbanken und verwendet die von den einzelnen Technologien zur Verfügung gestellten Java Schnittstellen. Über die Quellenanbindung werden die spezifischen Operationen für die Ausgangsdatenbank (lesen, schreiben) sowie für die Evaluierungsdatenbanken (lesen, schreiben, suchen) aufgerufen. Die Ausgangsdatenbank dient zur Zwischenspeicherung der Testdatensätze, um diese anschließend für die Speicherung als Knoten und Kanten in den Evaluierungsdatenbanken aufzubereiten.

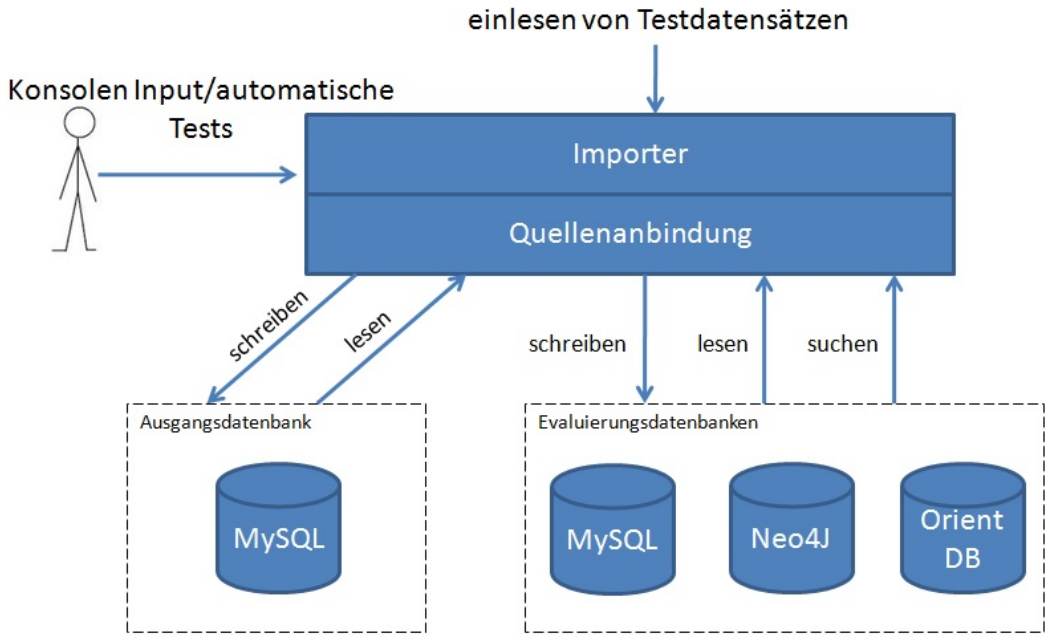


Abbildung 4.6.: Vereinfachte grafische Darstellung der Java-Applikation zur Kommunikation mit den unterschiedlichen Datenbanksystemen.

4.5. Verarbeitung des Microsoft Academic Graph

Nachfolgende Kapitel illustrieren die Abläufe der Verarbeitung des Microsoft Academic Graph, ausgehend von den Rohdaten bis hin zu der Abbildung der Informationen in Form von Graphen in den unterschiedlichen Datenbanksystemen. Der Abschnitt 4.5.1 beschreibt den Import der Daten aus den Textdateien in die relationale Datenbank. Die relationale Datenbank dient als Ausgangsdatenbank um die Informationen nachfolgend auszulesen und so aufzubereiten, dass diese in den ausgewählten Datenbanksystemen Neo4j, OrientDB und der relationalen Datenbank MySQL, gespeichert werden können. Das Auslesen und Aufbereiten der importierten Testdaten wird im Kapitel 4.5.2 beschrieben. Anschließend beschreiben die Kapitel 4.5.3, 4.5.4 und 4.5.5 die Abbildung der Testdaten in den zwei Graphdatenbanken, Neo4j und OrientDB, sowie in der relationalen Datenbank, welche als Gradmesser bezüglich Datenbanksysteme anzusehen ist. Die Beschreibung der Datenabfrage, unter Berücksichtigung der Berechtigung, erfolgt für Neo4j im Kapitel 4.5.6. Die Datenabfrage für OrientDB wird im Kapitel 4.5.7 beschrieben. Die relationale Datenbank in Abschnitt 4.5.8 komplettiert das Kapitel der Datenverarbeitung.

4.5.1. Datenimport in die relationale Datenbank

Das Einlesen der Testdatensätze erfolgt über den von `java.util` zur Verfügung gestellten `Scanner`. Dabei handelt es sich um einen einfachen Scanner, der es ermöglicht, Text unter der Verwendung von Regular Expressions zeilenweise einzulesen[4]. Aufgrund der einfachen Umsetzung und der Tatsache, dass Methoden für das zeilenweise Einlesen von Textdateien zur Verfügung gestellt werden, fiel die Wahl auf die `Scanner`-Klasse. Ein byteweises Einlesen wäre die Alternative, welche es ermöglichen würde, nicht interpretierbare Zeichen zu prüfen und diese spezifisch zu behandeln. Der Nachteil wäre die im Vergleich zum Scanner aufwändigere Umsetzung. Eingelesene

[4] `https://docs.oracle.com/javase/7/docs/api/java/util/Scanner.html`, aufgerufen am 2016-01-14

Zeilen mit nicht interpretierbaren Zeichen werden bei der Umsetzung mittels der `Scanner`-Klasse ignoriert. Die abstrakte Klasse `ParserScanner` stellt Methoden für das Einlesen von Daten zur Verfügung, welche in den jeweiligen Subklassen implementiert werden. Die abstrakte Methode `parseNext` wird in den Subklassen implementiert und regelt das spezifische Einlesen der unterschiedlichen Daten. Dies ermöglicht eine flexible und einfache Erweiterung um typspezifische Parser. Zur Vermeidung von Speicherproblemen werden die Zeilen der jeweiligen Testdaten blockweise eingelesen sowie anschließend blockweise in die initiale MySQL Ausgangsdatenbank geschrieben. Die Klasse `ImporterScanner` regelt das Einlesen sowie das Schreiben in die Datenbank. Je nach einzulesender Textdatei, welche einen speziellen Datentyp repräsentiert, regelt der geeignete Parser das Einlesen der Daten und retourniert eine Liste von Referenzen auf Objekte des jeweiligen Datentyps. Erfolgt beispielsweise das Einlesen von Autoren, enthält die Liste Referenzen von `Author`-Objekten (Abb. 4.7). Die Elemente der Liste werden anschließend in die Datenbank geschrieben.

Für das Schreiben in die Datenbank stellt das Interface `DataBaseWriter` einfache Methoden zur Verfügung, um die Daten geeignet in die Datenbank zu speichern. Die Umsetzung via Interface-Implementierung ermöglicht eine flexible Erweiterung für unterschiedliche Datenbanksysteme. Die Klasse `DataBaseWriterMySql` implementiert das Interface und schreibt den zuvor eingelesenen Block an Daten in die Datenbank. Abbildung 4.7 zeigt die essentiellen Komponenten für den Datenimport. Die Klassen `Author` und `Journal` sowie die spezifischen Parser `AuthorParser`, `JournalParser` und `PaperParser` werden als Beispiel angeführt und stehen repräsentativ für die gesamte Menge an Parser-Klassen und Klassen, welche die einzelnen Informationen der eingelesenen Daten zwischen speichern.

Der Codeausschnitt 4.1 der Klasse `ImporterScanner` zeigt das Einlesen und Schreiben von Datensätzen am Beispiel eines Autors. In Zeile 5 wird die Subklasse `AuthorParser` zurückgegeben, welche das Einlesen von Autoren regelt. Anschließend wird in Zeile 10 das Schreiben initiiert. Entsprechend den Datentyp wird die implementierte Interface-Methode aufgerufen, welche das Schreiben in die Datenbank durchführt (Zeile 38).

```
1  public void doImport(Set<DataType> types) {
2      for (DataType type : types) {
3
4          try {
5              ParserScanner parser = this.getParserForType(type);
6
7              while (parser.hasNext()) {
8                  List batch = parser.parseNext();
9
10                 writeBatchToDatabase(type, batch);
11             }
12         }
13     }
14 }
15
16 private ParserScanner getParserForType(DataType type) throws IOException {
17
18     switch (type) {
19     case AUTHORS:
20         return new AuthorParser(filePathConfig.getAuthors(), batchSize);
21     default:
22         throw new IllegalArgumentException("Unknown DataType '" + type.toString()
               + "'");
23     }
24 }
25
26 private void writeBatchToDatabase(DataType type, List list) {
27
28     switch (type) {
29     case AUTHORS:
30
31         /*
32          * check type safety; if first element of list is from the type all
33          * other should also be from the type Author
34          */
35         if (!(list.get(0) instanceof Author))
36             throw new IllegalArgumentException("Wrong DataType for list ...");
37
38         writer.writeAuthors(list);
39         break;
40     default:
41         throw new IllegalArgumentException("Unknown DataType '" + type.toString()
               + "'");
42     }
43 }
```

Listing 4.1: Einlesen der Textdateien des Microsoft Academic Graph und schreiben der unterschiedlichen Datensätze in der Klasse `ImporterScanner` mittels Java

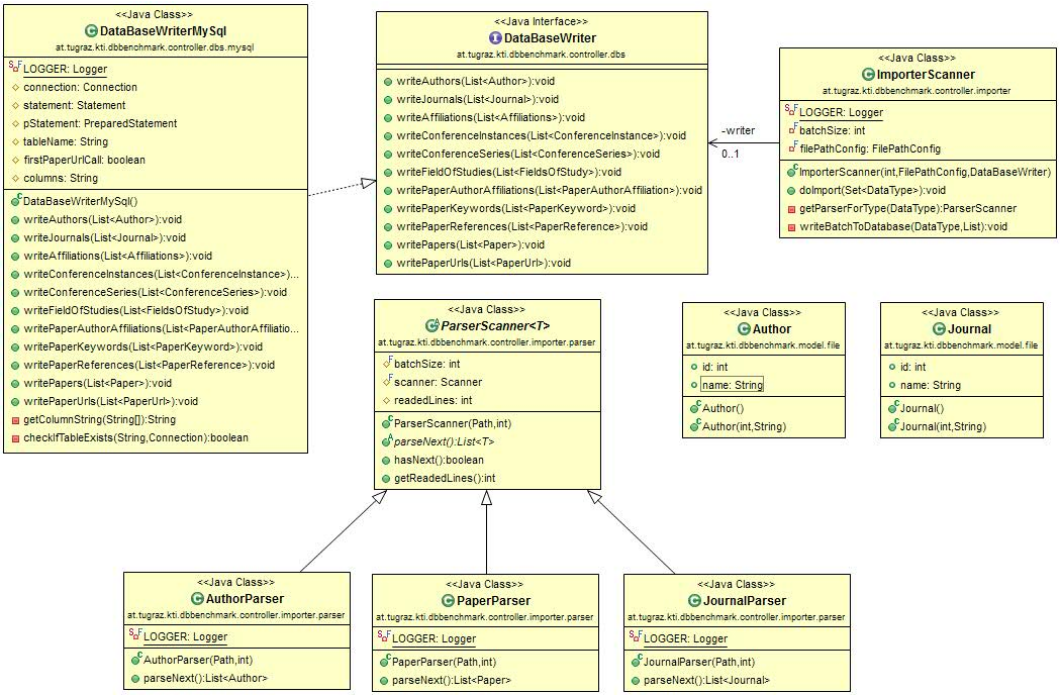


Abbildung 4.7.: UML-Diagramm Ausschnitt der einzelnen Komponenten für das Einlesen des Microsoft Academic Graph in die MySQL Datenbank

4.5.2. Export und Aufbereitung der Testdaten

Bevor die Testdaten in die Evaluierungsdatenbanken von Neo4j, OrientDB und MySQL geschrieben werden können, müssen diese aus der Ausgangsdatenbank gelesen werden. Das Auslesen erfolgt über die Klassen `ImporterMySQL` und `DataBaseReaderMySql`. Die Klasse `DataBaseReaderMySql` stellt die Quellenanbindung zur Ausgangsdatenbank dar. Die `ImporterMySQL` Klasse steuert das Auslesen. Es wird solange ein Block von Datensätzen ausgelesen, bis die gewünschte auszulesende Anzahl an Datensätzen erreicht wurde. Aus Performance Gründen werden die Daten schrittweise aus der Datenbank ausgelesen und in einer abstrakten Graphstruktur gespeichert. Der Graph wird über eine `HashMap<Integer, GraphLayer>` abgebildet. Die zu einer Publikation gehörenden Metainformationen wie Autoren, Journal etc. sind mit dieser verbunden. Aus diesem Grund wurde eine `HashMap` verwendet, da diese die Möglichkeit bietet, Publikationen ohne vorhandene

Metainformationen zu speichern und diese vergleichsweise schnell in der Struktur zu Suchen. Die Speicherung in der abstrakten Struktur erlaubt des weiteren eine Austauschbarkeit. In der Struktur gespeicherte Daten können in beliebige Datenbanken geschrieben werden. Der Schlüssel der `HashMap` entspricht der Identifikationsnummer einer Publikation. Das dazugehörige `GraphLayer`-Objekt repräsentiert die mit einer Publikation verbundenen Metainformationen. Die Metainformationen für die Publikation werden in den Member-Variablen `paperMetaData` und `paperConferenceData` des `GraphLayer`-Objekts gespeichert. Das Berechtigungskonzept 4.3 wird beim Aufbau der abstrakten Graphstruktur realisiert. Die Zugriffsberechtigungen für Metainformationen von Publikationen werden beim Auslesen der Webseiten einer Publikation erzeugt. Die Berechtigungen für Konferenzen werden aus den Webseiten der einzelnen Konferenztagungen generiert.

Das Einlesen der Daten - Publikation und dazugehörige Metainformationen - sowie die Abbildung des Graphen in der abstrakten Datenstruktur wurde in einzelne Phasen des Einlesens eingeteilt. Grund dafür ist, die `JOIN`-Operationen pro Auslesephase möglichst gering zu halten. Theoretisch wäre es möglich, alle zu einer Publikation gehörenden Informationen auszulesen, jedoch mit deutlich negativen Auswirkungen auf die Performance. Die Phasen werden für jede Publikation, die ausgelesen wird, durchgeführt, bis die Anzahl der gewünschten Publikationen erreicht wurde.

1. **Publikationen:** Eine Publikationen wird aus der relationalen Datenbank ausgelesen. Für jede Publikation wird ein neues Schlüssel-Werte-Paar in der abstrakten Graphstruktur angelegt. Als Schlüssel dient die Identifikationsnummer der Publikation. Als Wert wird ein neues `GraphLayer`-Objekt mit den bereits vorhandenen Informationen der Publikation erzeugt. Dadurch wird für die gelesene Publikation bereits ein `GraphLayer`-Objekt angelegt, welches anschließend weitere Metainformationen einer Publikation speichert.
2. **Autoren und Institution:** Anschließend werden für die Publikation, die in Schritt eins ausgelesen wurde, Informationen zu Autoren und deren Institut ausgelesen und im `GraphLayer`-Objekt der jeweiligen Publikation gespeichert. Die Informationen werden im `PaperMetaData`-Objekt in einer eigenen Datenstruktur gespeichert.
3. **Informationen zu der Publikation:** In der dritten Phase werden das Forschungsgebiet, Schlüsselwörter und die Webseiten der jeweiligen

Publikation ausgelesen und im `GraphLayer`-Objekt der dazugehörigen Publikation gespeichert. Jede Webseite einer Publikation wird in die Teile Protokoll und Host zerlegt. Wobei die Teile des Host wiederum in durch Punkt getrennte Teile zerlegt werden. Die einzelnen Teile spiegeln die Berechtigungen wider und werden der `accessList` hinzugefügt. Die Datenstruktur ist eine `SortedSet`, in welcher jede Berechtigung nur einmal gespeichert wird.

4. **Konferenzinformationen:** Für die im ersten Schritt gelesen Publikation werden die Konferenzinformationen ausgelesen. Ein `GraphLayer`-Objekt hält die Informationen zu der Konferenz in einer eigenen Member-Variable vom Datentyp `ConferenceSerieAndInstance`. Jede Konferenz kann mehrere Konferenztagungen besitzen. Diese werden als `ConferenceInstance` in der Datenstruktur `HashMap<Integer, ConferenceInstance>` gespeichert. Auch hier besteht der Key aus der Identifikationsnummer der Konferenztagung und der Wert besteht aus der `ConferenceInstance`, welche die Informationen zu der jeweiligen Tagung speichert. Die Berechtigungen werden identisch zu den Berechtigungen des Papers erzeugt. Aus jeder Webseite einer `ConferenceInstance` werden die Berechtigungen extrahiert.
5. **Referenzen:** Als letzter Schritt werden die Referenzen, in welchen die ausgelesene Publikation referenziert wird, ausgelesen. Die Referenzen werden in der Member-Variable `paperMetaData` des `GraphLayer`-Objekts der jeweiligen Publikation gespeichert. Existiert die Referenzpublikation in der abstrakten Graphstruktur nicht, wird ein Schlüssel-Werte-Paar mit der Identifikationsnummer der Referenzpublikation hinzugefügt. Metainformationen aus den Schritten zwei bis fünf werden für die Referenzen nicht ausgelesen.

Das UML-Diagramm in der Abbildung 4.8 zeigt die einzelnen Elemente, welche für das Auslesen und die Datenhaltung in der abstrakten Graphstruktur notwendig sind. Die Klassen `Affiliation` und `Author` sind nicht zu verwechseln mit den Klassen aus Abbildung 4.7. Die Klassen `Author`, `Journal`, etc. in Abbildung 4.7 dienen ausschließlich zur Datenhaltung nach dem einlesen aus den Textdateien der Testdatensätze.

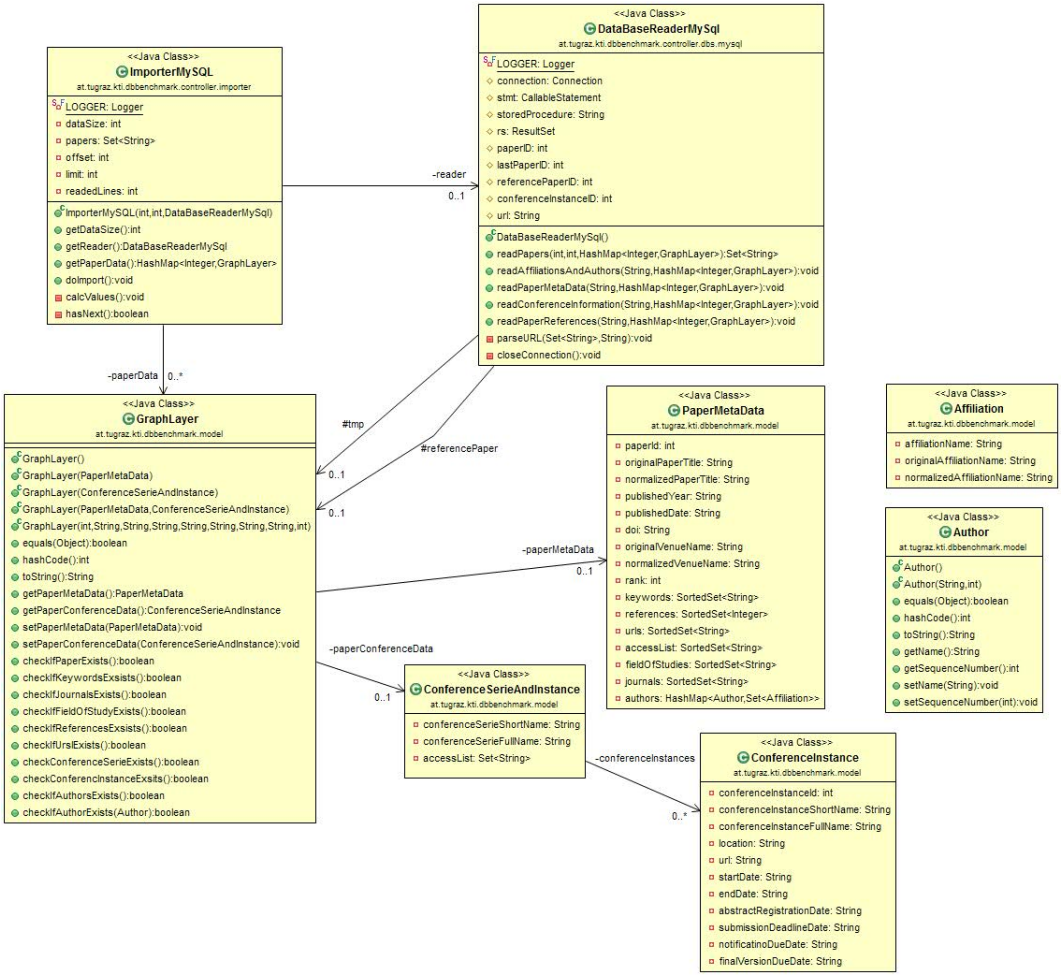


Abbildung 4.8.: UML-Diagramm der einzelnen Elemente für das Auslesen aus der MySQL Ausgangsdatenbank und zur Speicherung in der abstrakten Graphdatenstruktur

4.5.3. Abbildung der Daten als Graph in Neo4j

Die in Kapitel 4.5.2 beschriebene abstrakte Graphdatenstruktur ist Ausgangspunkt, um die Daten in Neo4j zu speichern. Durch die Datenstruktur ist es möglich, die Informationen so zu verarbeiten, dass sie als Knoten und Kanten gespeichert werden können. Für das Schreiben der Knoten und Kanten in die spezifische Datenbank stellt das Interface `GraphDataBaseWriter` die Methode `writeData` zur Verfügung, welche von der Quellenanbindung der einzelnen Datenbank implementiert wird (Abb. 4.9).

Dazu wird über die gesamte Datenstruktur iteriert. Jeder Eintrag repräsentiert eine Publikation und seine dazugehörigen Metainformationen. Das Berechtigungskonzept der einzelnen Knoten wird über die in Teile zerlegten Webseiten realisiert. Eine Konferenz erbt die Berechtigungen seiner einzelne Konferenztagungen (`ConferenceInstances`). Die übrigen Knoten erben die Berechtigungen der Publikation. Die Berechtigungen werden als Attribut in jedem einzelnen Knoten gespeichert. Für jeden Eintrag werden folgende Schritte durchgeführt.

1. Erstellen eines Knoten mit dem Label `PAPER` für die Publikation und der zum Publikationsknoten gehörenden Information.
2. Über die einzelnen Identifikationsnummer der Referenzen der Publikation iterieren; ist der `PAPER`-Knoten mit der Identifikationsnummer noch nicht vorhanden, wird ein `PAPER`-Knoten nur mit der Identifikationsnummer erstellt. Anschließend wird eine Kante mit dem Label `REFERENCED_BY` zwischen Publikation und Referenzpublikation erzeugt. Dadurch werden rekursive Suchoperationen auf die Map vermindert. Die Metainformationen können später agerufen werden, da die Referenz in der abstrakten Struktur vorkommen muss.
3. Anschließend werden die `KEYWORD`-Knoten erzeugt. Dazu wird über jeden Schlüsselwort-Eintrag iteriert und geprüft ob ein Knoten mit dem Schlüsselwort bereits vorhanden ist. Existiert bereits ein Knoten, jedoch ohne Verbindung zwischen Publikation und Schlüsselwort, wird eine Kante erzeugt. Andernfalls wird der Knoten mit dem Schlüsselwort erzeugt und die Beziehung zwischen Publikation und Schlüsselwort erstellt.

4. Nach dem selben Schema wie in Schritt drei werden die Knoten `JOURNAL` (für die veröffentlichte Zeitschrift) und `FIELD_OF_STUDY` (für den Forschungsbereich der Publikation) erstellt. Die `PAPER_URL`-Knoten werden ohne zu prüfen erstellt, da die URL immer eindeutig ist. Dies erspart Abfrageoperationen auf den Datenbestand.
5. Für das Erstellen von `AUTHOR`- und `AFFILIATION`-Knoten wird über die Schlüssel-Werte-Paare der Member-Variable `authors` iteriert. Ein Schlüssel-Werte-Paar repräsentiert einen Autor und seine dazugehörigen Mitgliedschaften. Ist der Autorenknoten mit dem Namen des Autors nicht vorhanden, wird dieser erzeugt; selbiges gilt für die Mitgliedschaften. Anschließend wird die Beziehung zwischen den beiden Knoten hergestellt, sofern diese nicht vorhanden ist. Abschließend wird der Autorenknoten mit der Publikation verbunden.
6. Als letzter Schritt werden die Knoten der Konferenz, `CONFERENCE` und `CONFERENCE_INSTANCE`, erzeugt. Die Metainformationen einer Konferenz sind in der Klasse `ConferenceSerieAndInstance` gespeichert. Der Konferenzknoten und seine Metainformation wird erstellt, sofern eine Konferenz mit dem Namen noch nicht vorhanden ist. Besteht die Beziehung zwischen Publikation und Konferenz noch nicht, wird diese ebenso erzeugt. Nachfolgend wird der Knoten `CONFERENCE_INSTANCE` erzeugt. Es wird geprüft, ob ein Knoten mit der Identifikationsnummer der Instanz bereits in der Datenbank vorkommt. Ist dies nicht der Fall, wird der Knoten, sowie die Beziehung zwischen Konferenz und der spezifischen Tagung hergestellt.

Das Erzeugen von Knoten des Typs `PAPER` und `JOURNAL` wird im Codeausschnitt 4.2 erläutert. Die Erzeugung der weiteren Knotentypen und Beziehungen erfolgt identisch zu den gezeigten. Das Schreiben der Daten erfolgt unter der Berücksichtigung von Transaktionen. Die verwendete Heapgröße hätte es ermöglicht, eine große Transaktion für die gesamten Datenimport zu verwenden, aber zur Vermeidung von großen Transaktionen und Reduzierung der Anzahl an Transaktionen wurde als Kompromiss eine Transaktion pro 20.000 gelesener Publikationen abgeschlossen. Vereinfacht gesagt wird nach dem Lesen von 20.000 Publikationen und den dazugehörigen Metainformationen die Transaktion abgeschlossen und eine neue Transaktion erzeugt. Anschließend werden die Einträge der Datenstruktur für den Graphen durchlaufen. In Zeile 18 und 19 werden die Metainforma-

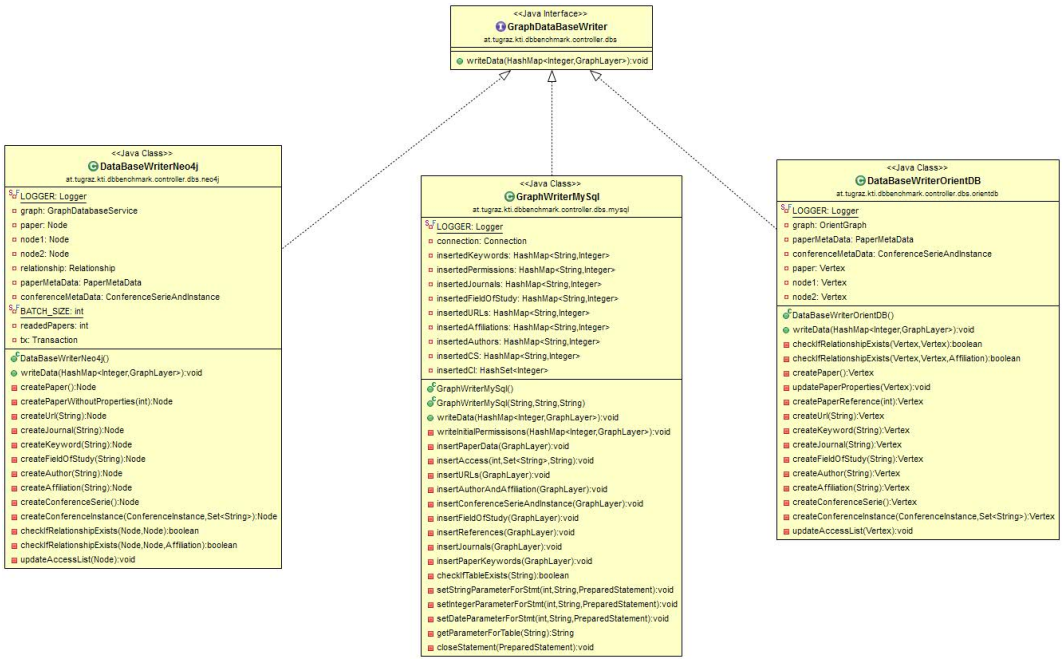


Abbildung 4.9.: UML-Diagramm der Quellenanbindungen zum Schreiben der Daten in die Datenbanken Neo4j, OrientDB und MySQL

tionen für die jeweilige Publikation in der entsprechenden Membervariable gespeichert. Anschließend wird die Funktion `createPaper` aufgerufen, welche einen Knoten für die Publikation erzeugt und den erzeugten Knoten zurückliefert. Liefert die Suche für eine Publikation in Zeile 38 einen Knoten zurück, werden die Attribute aktualisiert. Dies ist der Fall, wenn zuvor bereits eine Referenz der Publikation ohne Attribute erzeugt wurde. Knoten, welche nicht vom Typ `PAPER` stammen, werden, sollten diese bereits vorhanden sein, nicht aktualisiert.

Nach der Erzeugung einer Publikation, werden die Knoten für die Metainformation, welche nicht direkt als Attribut am Knoten für die Publikation gespeichert werden, erzeugt. Repräsentativ werden in Zeile 25 die zur Publikation gehörenden Zeitschriften erzeugt. Da mehrere Publikationen in einer Zeitschrift veröffentlicht werden können, besteht die Möglichkeit, das der Knoten mit dem Titel der Zeitschrift bereits in der Datenbank vorhanden ist. In diesem Fall müssen die Berechtigungen aktualisiert werden (Zeile 81), da der Knoten die Berechtigungen einer direkt verbundenen Publikation erbt (Kapitel 4.3). In Zeile 28 wird geprüft, ob die Beziehung zwischen Start- und Endknoten bereits existiert. Dazu wird über alle Verbindungen des Startknotens iteriert (Zeile 92) und geprüft, ober der übergebene Endknoten dem Endknoten der Verbindung entspricht.

```
1  public class DataBaseWriterNeo4j implements GraphDataBaseWriter {
2
3      private GraphDatabaseService graph;
4      private Node paper, node1, node2;
5
6      private Relationship relationship;
7
8      private PaperMetaData paperMetaData;
9      private ConferenceSerieAndInstance conferenceMetaData;
10
11     public void writeData(HashMap<Integer, GraphLayer> data) {
12
13         graph = Neo4jConfig.getNeo4jDB();
14         try (Transaction tx = graph.beginTx()) {
15             // iterate over each paper in the graph structure
16             for (GraphLayer object : data.values()) {
17
18                 paperMetaData = object.getPaperMetaData();
19                 conferenceMetaData = object.getPaperConferenceData();
20
21                 paper = createPaper();
22
23                 // if we have some journals, create journal node and
24                 // relationship between paper and journal
```

```
25                  for (String journal : paperMetaData.getJournals()) {
26                      node1 = createJournal(journal);
27
28                      if (!checkIfRelationshipExists(paper, node1))
29                          relationship = paper.createRelationshipTo(node1,
                                RelationshipLabel.PUBLISHED_IN);
30                  }
31
32                  // do the same for other nodes
33              }
34          }
35      }
36
37      private Node createPaper() {
38          Node node = graph.findNode(NodeLabel.PAPER, NodeProperties.PAPER_ID,
                paperMetaData.getPaperId());
39
40          if (node == null) {
41              LOGGER.log(Level.INFO,
42                      "Create node " + NodeLabel.PAPER + " for paper <" +
                            paperMetaData.getPaperId() + ">.");
43              node = graph.createNode(NodeLabel.PAPER);
44              node.setProperty(NodeProperties.PAPER_ID, paperMetaData.getPaperId());
45          }
46          // update the data for the paper
47          if (paperMetaData.getOriginalPaperTitle() != null)
48              node.setProperty(NodeProperties.ORIGINAL_TITLE,
                        paperMetaData.getOriginalPaperTitle());
49          if (paperMetaData.getNormalizedPaperTitle() != null)
50              node.setProperty(NodeProperties.NORMALIZED_TITLE,
                        paperMetaData.getNormalizedPaperTitle());
51          if (paperMetaData.getPublishedYear() != null)
52              node.setProperty(NodeProperties.PUBLISH_YEAR,
                        paperMetaData.getPublishedYear());
53          if (paperMetaData.getPublishedDate() != null)
54              node.setProperty(NodeProperties.PUBLISH_DATE,
                        paperMetaData.getPublishedDate());
55          if (paperMetaData.getDoi() != null)
56              node.setProperty(NodeProperties.DOI, paperMetaData.getDoi());
57          if (paperMetaData.getOriginalVenueName() != null)
58              node.setProperty(NodeProperties.ORIGINAL_VENUE_NAME,
                        paperMetaData.getOriginalVenueName());
59          if (paperMetaData.getNormalizedVenueName() != null)
60              node.setProperty(NodeProperties.NORMALIZED_VENUE_NAME,
                        paperMetaData.getNormalizedVenueName());
61          if (paperMetaData.getRank() > 0)
62              node.setProperty(NodeProperties.PAPER_RANK, paperMetaData.getRank());
63          if (paperMetaData.getAccessList() != null)
64              node.setProperty(NodeProperties.ACCESS,
65                      paperMetaData.getAccessList().toArray(new
                            String[paperMetaData.getAccessList().size()]));
66
67          return node;
68      }
69
```

```
70      private Node createJournal(String journal) {
71          Node node = graph.findNode(NodeLabel.JOURNAL, NodeProperties.NAME,
                journal);
72
73          if (node == null) {
74              LOGGER.log(Level.INFO, "Create node " + NodeLabel.JOURNAL + " for
                    name <" + journal + ">.");
75              node = graph.createNode(NodeLabel.JOURNAL);
76              node.setProperty(NodeProperties.NAME, journal);
77              node.setProperty(NodeProperties.ACCESS,
78                      paperMetaData.getAccessList().toArray(new
                            String[paperMetaData.getAccessList().size()]));
79          } else {
80              // update the access list
81              updateAccessList(node);
82          }
83
84          return node;
85      }
86
87      private boolean checkIfRelationshipExists(Node startNode, Node endNode) {
88
89          if (startNode.getRelationships() == null)
90              return false;
91
92          for (Relationship rel : startNode.getRelationships()) {
93              if (rel.getOtherNode(startNode).equals(endNode)) {
94                  return true;
95              }
96          }
97
98          return false;
99      }
100
101     private void updateAccessList(Node node) {
102
103         // update the access list
104         Set<String> tmp = new HashSet<String>();
105         tmp.addAll(Arrays.asList((String[])
                node.getProperty(NodeProperties.ACCESS)));
106
107         tmp.addAll(paperMetaData.getAccessList());
108         node.setProperty(NodeProperties.ACCESS, tmp.toArray(new
                String[tmp.size()]));
109
110     }
111 }
```

Listing 4.2: Schreiben von `PAPER`- und `JOURNAL`-Knoten in Neo4j

4.5.4. Abbildung der Daten als Graph in OrientDB

Das Einlesen der Testdatensätze in OrientDB funktioniert identisch zu der Graphdatenbank Neo4j (Kapitel 4.5.3). Ähnlich zu Neo4j werden auch in OrientDB die Daten unter Berücksichtigung von Transaktionen in die Datenbank geschrieben, jedoch ist die in Neo4j gewählte Batchgröße für die Transaktion in OrientDB nicht umsetzbar. Da bei gewählten Batchgrößen von 20.000 und 10.000 Publikationen die Stackgröße aufgrund rekursiver Aufrufe der OrientDB Java API nicht ausreicht, wurde die Transaktion nach jeder Publikation und den dazugehörigen Metainformationen abgeschlossen und eine neue Transaktion erstellt. Die Quellanbindung `DataBaseWriterOrientDB` regelt den Verbindungsaufbau und schreibt die Daten aus der abstrakten Graphstruktur in die Datenbank. Die abstrakte Graphstruktur `HashMap<Integer, GraphLayer>` wird durchlaufen und die Daten werden dementsprechend verarbeitet, um die Information in Knoten und Kanten (Abb. 4.2) speichern zu können. Die Knoten- und Kantenelemente werden vor dem Erzeugen (wie bei Neo4j) überprüft, ob diese bereits existieren. Nicht vorhandene Knoten sowie nicht existente Kanten zwischen zwei Knoten werden hinzugefügt.

Ein weiterer wesentlicher Unterschied bei der Implementierung zeigt der Codeausschnitt 4.3. Knoten sind vom Datentyp `Vertex`. Der Funktion `getVertices` für die Suche von Knoten muss das Label des Knoten, die Namen der zu suchenden Attribute und die Werte der Attribute übergeben werden (Zeile 9). Die Suche liefert ein `Iterable` über die gefundenen Knoten zurück. Da der zu suchenden Knoten immer eindeutig sind, kann der erste Knoten des `Iterable` als gewünschtes Ergebnis angesehen werden (Zeile 16). Im Gegensatz zur Überprüfung der Beziehung zwischen zwei Knoten (in Neo4j) muss in OrientDB die Richtung der Beziehung angegeben werden. Es wird über alle Beziehungen beider Richtungen iteriert (Zeile 54) und geprüft, ob der Endknoten der ausgehenden oder der eingehenden Beziehung dem zu prüfenden Endknoten entspricht.

```
1 private Vertex createPaper() {
2
3     // check first if a paper with all properties exists
4     // if so return
5     // if an paper just with the id exists, must be a paper reference
6     // so set the known properties
7     // if nothing exists, create a new vertex with the properties
```

```
8       Iterable<Vertex> resultIterator =
            graph.getVertices(OrientDbModel.VERTEX_PAPER,
9               new String[] { NodeProperties.PAPER_ID }, new Object[] {
                    paperMetaData.getPaperId() });
10
11      LOGGER.log(Level.INFO,
12              "Create Node " + OrientDbModel.VERTEX_PAPER + " for paper id <" +
                    paperMetaData.getPaperId() + ">");
13
14      Vertex v;
15      if (resultIterator.iterator().hasNext()) {
16          v = resultIterator.iterator().next();
17          updatePaperProperties(v);
18      } else {
19          v = graph.addVertex("class:" + OrientDbModel.VERTEX_PAPER);
20          v.setProperty(NodeProperties.PAPER_ID, paperMetaData.getPaperId());
21
22          if (paperMetaData.getOriginalPaperTitle() != null)
23              v.setProperty(NodeProperties.ORIGINAL_TITLE,
                        paperMetaData.getOriginalPaperTitle());
24          if (paperMetaData.getNormalizedPaperTitle() != null)
25              v.setProperty(NodeProperties.NORMALIZED_TITLE,
                        paperMetaData.getNormalizedPaperTitle());
26          if (paperMetaData.getPublishedYear() != null)
27              v.setProperty(NodeProperties.PUBLISH_YEAR,
                        paperMetaData.getPublishedYear());
28          if (paperMetaData.getPublishedDate() != null)
29              v.setProperty(NodeProperties.PUBLISH_DATE,
                        paperMetaData.getPublishedDate());
30          if (paperMetaData.getDoi() != null)
31              v.setProperty(NodeProperties.DOI, paperMetaData.getDoi());
32          if (paperMetaData.getOriginalVenueName() != null)
33              v.setProperty(NodeProperties.ORIGINAL_VENUE_NAME,
                        paperMetaData.getOriginalVenueName());
34          if (paperMetaData.getNormalizedVenueName() != null)
35              v.setProperty(NodeProperties.NORMALIZED_VENUE_NAME,
                        paperMetaData.getNormalizedVenueName());
36          if (paperMetaData.getRank() > 0)
37              v.setProperty(NodeProperties.PAPER_RANK, paperMetaData.getRank());
38          // gets the access list of the URL node
39          if (paperMetaData.getAccessList() != null) {
40              v.setProperty(NodeProperties.ACCESS,
41                      paperMetaData.getAccessList().toArray(new
                            String[paperMetaData.getAccessList().size()]));
42          }
43      }
44
45      graph.commit();
46      return v;
47  }
48
49  private boolean checkIfRelationshipExists(Vertex v1, Vertex v2) {
50
51      if (v1.getEdges(Direction.BOTH) == null)
52          return false;
```

```
53
54      for (Edge e : v1.getEdges(Direction.BOTH)) {
55          if (e.getVertex(Direction.OUT).equals(v2) ||
                e.getVertex(Direction.IN).equals(v2))
56              return true;
57      }
58
59      return false;
60  }
```

Listing 4.3: Codeausschnitt für das Speichern von Publikationsknoten in OrientDB

4.5.5. Abbildung der Daten als Graph in MySQL

MySQL gilt als etablierter Vertreter von relationalen Datenbanken. Die relationale Datenbank wurde ausgewählt, um bei der Evaluierung der Leistungsfähigkeit von Graphdatenbanken einen Vergleich herstellen zu können.

Relationalen Datenbanken ist es nicht möglich Daten in Form von Knoten und Kanten, wie dies bei Graphdatenbanken erfolgt, zu speichern. Aus diesem Grund weicht das Prozedere des Daten Imports von dem der Graphdatenbanken ab. Das Schreiben der Testdaten regelt für MySQL die Implementierung `GraphWriterMySql` des Interface `GraphDataBaseWriter`.

Ähnlich zu den Abläufen in Neo4j und OrientDB (Kapitel 4.5.3 und 4.5.4) wird über die gesamte Datenstruktur iteriert und die Daten unter der Verwendung von Transaktionen in die Datenbank geschrieben. Analog zu OrientDB wurde aufgrund von Limitierungen die Transaktionsgröße nach jeder Publikation abgeschlossen. Im Unterschied zu den Graphdatenbanken werden eingangs jedoch die gesamten Berechtigungen aller Publikationen und Konferenzen in die Berechtigungstabelle `access` geschrieben. Dadurch können die mit einem automatischen Schlüssel erzeugten Berechtigungen in einer `HashMap` gespeichert werden. Dies ermöglicht es, beim anschließenden Erzeugen der Knoten den jeweiligen Schlüssel aus der `HashMap` zu lesen. Anderenfalls müssten Leseoperationen auf der Datenbank ausgeführt werden. Anschließend wird jeder Eintrag in der abstrakten Datenstruktur abgearbeitet. Daten die in der Datenbank gespeichert wurden, werden mit

ihrem primären Schlüssel in den assoziativen Speicher geschrieben. Dadurch erfolgen die Suchoperationen zur Überprüfung, ob ein Knoten schon vorhanden ist, nicht mittels SQL-Abfragen auf den Datenbestand, sondern mittels `containsKey` Methode der spezifischen `HashMap`. Dadurch können Operationen auf die Datenbank minimiert werden. Folgende Schritte werden dabei für jeden Eintrag, welcher eine Publikation und ihre Metadaten darstellt, durchgeführt.

1. Einfügen der Publikation in die Tabelle `paper`: Die Berechtigungen für die Publikation werden in die Tabelle `paper_acces` geschrieben.
2. Jedes zur Publikation gehörende Schlüsselwort wird in die Tabelle `keyword` geschrieben, sollte es noch nicht vorhanden sein. Die Verbindung zwischen Publikation und eingefügtem Schlüsselwort bzw. bereits vorhandenem Schlüsselwort wird in der Tabelle `paper_keyword` abgelegt. Die Berechtigungen der einzelnen Schlüsselwörter wird anschließend in die Tabelle `keyword_access` geschrieben. Bereits vorhandene Berechtigungen für jedes Schlüsselwort werden ignoriert, neue werden hinzugefügt.
3. Einlesen der Journals. Erfolgt identisch zu den Schlüsselwörtern. Journals werden in der Tabelle `journal` gesichert, sollten diese noch nicht vorhanden sein. Die Beziehung zwischen Publikation und Journal wird in `paper_journal` gespeichert. Die Zugriffsberechtigungen werden in der Tabelle `journal_access` abgelegt, sollte sie noch nicht vorhanden sein.
4. Hinzufügen der Webseiten der Publikation in die Tabelle `url`, sowie der Berechtigungskonzepte in die Tabelle `url_access`: Die Verbindungen werden nicht in eine dedizierte Tabelle geschrieben, da die Tabelle `url` ein Attribut für die Identifikationsnummer der Publikation besitzt.
5. Hinzufügen der Forschungsgebiete einer Publikation sind identisch wie in Schritt zwei und drei. Die Forschungsgebiete werden in der Tabelle `field_of_study` gespeichert. Die Beziehung zwischen dem jeweiligen Paper und dem Forschungsgebiet wird in der Tabelle `paper_field_of_study` und die Zugriffsberechtigungen, sollten diese noch nicht vorhanden sein, in `field_of_study_access` abgelegt.
6. Hinzufügen von Autoren und Instituten der jeweiligen Publikation: die Autoren werden in der Tabelle `autor` gespeichert, sollten diese noch nicht vorhanden sein. Die Zugangsberechtigungen der einzelnen

Autoren werden, sollten diese ebenfalls noch nicht vorhanden sein, in die Tabelle `autor_access` geschrieben. Das selbe Prozedere erfolgt für die Institute. Diese werden jedoch in die Entitäten `affiliation` und `affiliation_access` geschrieben. Die Beziehungen zwischen den Knoten Publikation und Autor, sowie Autor und Institut werden in den Tabellen `paper_author` und `author_affiliation` gespeichert.

7. Einlesen der Konferenzen und Konferenzinstanzen der jeweiligen Pulblikation erfolgt identisch zu Autoren und Instituten. Die Konferenz wird in der Tabelle `conference` und die Instanzen in der Tabelle `conference_instance` gespeichert, sollten diese noch nicht vorhanden sein. Die jeweiligen Berechtigungen werden in die Entitäten `conference_access` und `conference_instance_access` geschrieben. Verbindungen für Publikation und Konferenz, sowie für Konferenz und die Konferenzinstanz; werden in den Tabellen `paper_conference` und `conference_confernce_instance` gespeichert.

4.5.6. Datenabfrage in Neo4j

Die Suche, welche implizit einen lesenden Zugriff auf die Datenbank bewirkt, wird über das Interface `GraphDataBaseReader` geregelt. Es stellt einfache Methoden für die Suche nach Autoren, Journalen und Publikationen zur Verfügung. Die Abbildung 4.10 zeigt den Aufbau der Klassen für den lesenden Zugriff der unterschiedlichen Datenbanksysteme. Die Klasse `DataBaseReaderNeo4j` implementiert die Methoden des Interfaces und dient als Quellenanbindung für den lesenden Zugriff auf die Datenbank von Neo4j.

Erfolgt eine Journal- oder Autorensuche, werden im ersten Schritt die dementsprechenden Knoten mit den jeweiligen Namen des Journals bzw. des Autors gesucht. Abhängig von definierten Berechtigungen, wird die entsprechende Suchmethode `searchAuthor` bzw. `searchJournal` gestartet. Anschließend werden die mit dem jeweiligen Knoten verbundenen Publikationen abgefragt. Bei einer Suche nach einer Publikation, über die Methode `searchPaper`, bleibt diese Abfrage aus. Die Ermittlung der mit einem Ausgangsknoten verbundenen Publikationen erfolgt über das von Neo4j zur Verfügung gestellte “Traversal Framework“. Es ermöglicht die Definition

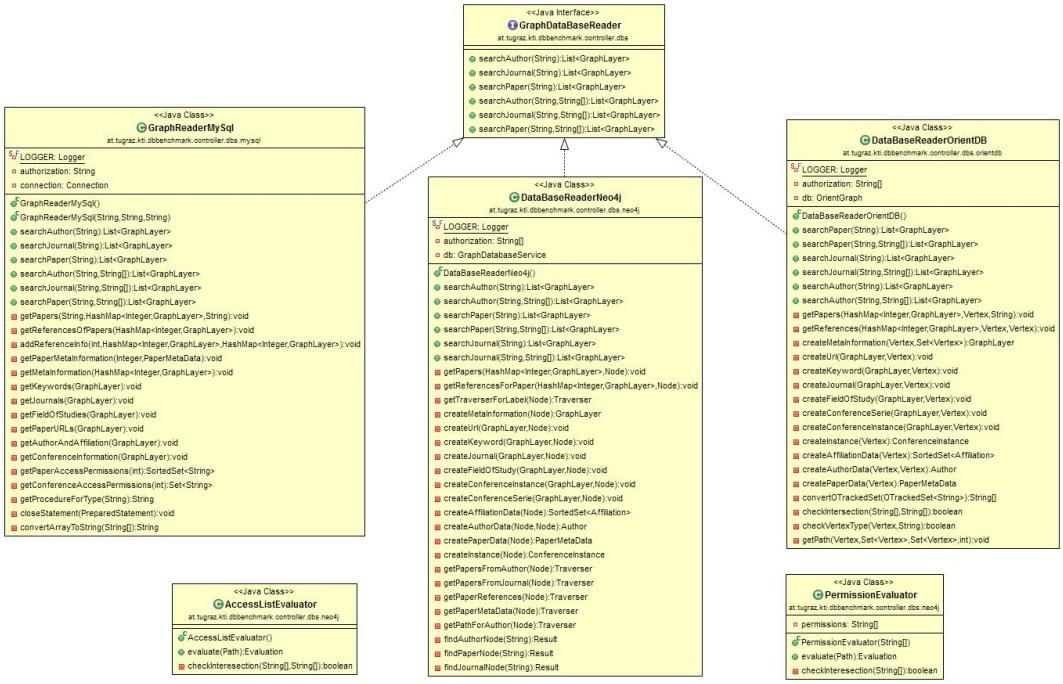


Abbildung 4.10.: Aufbau des lesenden Zugriff auf unterschiedliche Datenbanksysteme in Form eines UML-Diagramms

und Ausführung einer Pfadsuche mittels Tiefen- bzw. Breitensuche am Datenbestand. Der Codeausschnitt 4.4 zeigt die Definition einer Traversierung, in Abhängikeit zur Suchvariante, ausgehend von einem Autor als Startknoten. Die Pfadsuche, ausgehend von einem Journal-Knoten, erfolgt identisch zu der gezeigten. Sie unterscheidet sich nur durch die Definition des Kantentyps und der Richtung der Kante. In Zeile 6 und 10 wird mittels einer `TraversalDescription` die Traversierung spezifiziert. Als Algorithmus wird die Tiefensuche angegeben. Da alle möglichen erreichbaren Knoten, ausgehend vom Startknoten, gesucht werden, spielt die Wahl des Algorithmus keine wesentliche Rolle. Autoren besitzen nur ausgehende Kanten vom Typ `WRITTEN_BY`, welche eine Verbindung zu einer Publikation herstellt. Die Traversalbeschreibung kann einen `Evaluator` enthalten. Er dient als Abbruchkriterium und definiert, ob die Traversierung fortgesetzt werden soll bzw. der aktuelle Knoten in den Pfad mitaufgenommen werden soll. Die Traversierung wird bei unzureichenden Berechtigungen bzw. beim Erreichen einer Pfadtiefe der Länge eins abgebrochen.

Unzureichende Berechtigungen werden durch die Implementierung eines eigenen Evaluators geprüft. Die Klassen `AccessListEvaluator` und `PermissionEvaluator` implementieren das Interface `Evaluator` und regeln die Traversierung unter der Berücksichtigung von Berechtigungen. Während `AccessListEvalutor` die Berechtigungen zwischen Knoten prüft, werden beim `PermissionEvaluator` die Berechtigungen des Knoten mit den vor der Suche definierten Berechtigungen geprüft. Der Java Quellcode 4.5 zeigt die Implementierung des Evaluator-Interfaces. Ist die Pfadlänge null, also am Beginn der Pfadsuche, wird der Startknoten nicht in den Pfad aufgenommen, die Traversierung jedoch fortgesetzt (Zeile 29). Bei einer Pfadlänge größer eins, werden die Berechtigungen zwischen Start- und Endknoten einer Kante überprüft (Zeile 21). Gibt es eine Schnittmenge der Berechtigungen, wird der Endknoten in den Pfad aufgenommen und die Traversierung fortgesetzt. Unzureichende Berechtigungen, also keine Übereinstimmung zwischen den Berechtigungen von Start- und Endknoten, beendet die Traversierung und der Knoten wird nicht in das Ergebnis der Pfadsuche aufgenommen.

Der `PermissionEvaluator` arbeitet identisch, jedoch werden nicht die Berechtigungen zwischen den Knoten geprüft, sondern die Berechtigungen eines jeden Knoten werden gegen die definierten Berechtigungen geprüft.

Der Aufruf der Methode `traverse` in Zeile 14 initiiert die Pfadsuche, ausgehend von einem Startknoten, unter der Berücksichtigung der zuvor definierten Regeln. Zurückgeliefert wird ein `Traverser`-Objekt, welches es ermöglicht, die Schritte der Traversierung zu durchlaufen.

```
1  private Traverser getPapersFromAuthor(final Node node) {
2      TraversalDescription td = null;
3      // if we have no authorizations, we compare the permissions between nodes
4      if (this.authorization == null) {
5          td = db.traversalDescription().depthFirst()
6                  .relationships(RelationshipLabel.WRITTEN_BY, Direction.OUTGOING)
7                  .evaluator(new
                       AccessListEvaluator()).evaluator(Evaluators.toDepth(1));
8      } else {
9          td = db.traversalDescription().depthFirst()
10                 .relationships(RelationshipLabel.WRITTEN_BY, Direction.OUTGOING)
11                 .evaluator(new PermissionEvaluator(this.authorization))
12                 .evaluator(Evaluators.toDepth(1));
13     }
14     return td.traverse(node);
15 }
```

Listing 4.4: Definition einer Traversierung unter Verwendung der implementierten Evaluatoren `AccessListEvaluator` und `PermissonEvaluator`

```
1  /**
2   * The class implements the Evaluator class of Neo4j and controls
3   * what should be returned from a traversal. If an intersection
4   * on the access list of two nodes exists, the node will be
5   * added to the path.
6   *
7   * @author bischoft
8   * @date 2015-16-11
9   */
10 public class AccessListEvaluator implements Evaluator {
11
12     @Override
13     public Evaluation evaluate(Path path) {
14         if (path.length() != 0) {
15
16             String[] startAccess = (String[])
                       path.lastRelationship().getStartNode().getProperty("access");
17             String[] endAccess = (String[])
                       path.lastRelationship().getEndNode().getProperty("access");
18
19             // check if an intersection between the start and the
20             // end node of the relation exists
21             if (checkInteresection(startAccess, endAccess)) {
22                 // include the end node and continue traversal
23                 return Evaluation.INCLUDE_AND_CONTINUE;
24             } else {
25                 // exclude and stop the traversal
26                 return Evaluation.EXCLUDE_AND_PRUNE;
27             }
```

```
28          }
29          return Evaluation.EXCLUDE_AND_CONTINUE;
30      }
31
32      /**
33       * Check if an intersection on the access list of two
34       * nodes exists.
35       *
36       * @param accessNode1
37       * @param accessNode2
38       * @return true if an intersection exists, end if not
39       */
40      private boolean checkInteresection(String[] accessNode1, String[]
            accessNode2) {
41          for (int i = 0; i < accessNode1.length; i++) {
42              for (int j = 0; j < accessNode2.length; j++) {
43                  if (accessNode1[i].equals(accessNode2[j])) {
44                      return true;
45                  }
46              }
47          }
48          return false;
49      }
50  }
```

Listing 4.5: Implementierung des Interfaces `Evaluator` zur Überprüfung der Berechtigungen zwischen verbunden Knoten

Die weiteren publikationsbezogenen Informationen werden aus den restlichen, mit der Publikation verbundenen Knoten gewonnen. Dies erfolgt gleichermaßen mittels Traversierung der im Datenmodell beschriebenen Knoten (Abb. 4.2), jedoch ohne Berücksichtigung der `AFFILIATION`-Knoten. Diese werden eigenständig bei der Extraktion der Information eines Autors behandelt, da die Verbindung zwischen `AUTHOR`- und `AFFILIATION`-Knoten Attribute beinhaltet, welche dediziert verarbeitet werden müssen. Die separate Traversierung stellt die Zuordnung der einzelnen Knoten zu einer Publikation sicher und gewährleistet, dass die Informationen zu dieser Publikation gehören. Würden ausgehend von einem Startknoten zur Gänze alle möglichen Knoten traversiert werden, könnten die Knoten nicht der korrekten Publikation zugeordnet werden und ein Speichern der zu einer Publikation gehörenden Metainformationen in der abstrakten Graphstruktur wäre nicht möglich. Der Quellcodeauschnitt 4.6 zeigt die Definition der Pfadsuche zur Ermittlung der mit einer Publikation verbundenen Knoten.

Die Referenzen einer Publikation sowie deren Metainformationen werden anschließend rekursiv abgearbeitet. Die Gesamten zu einer Publikation

gehörenden Informationen werden in ein `GraphLayer`-Objekt geschrieben, welches genau eine Publikation repräsentiert. Die einzelnen `GraphLayer`-Objekte werden einer `HashMap` hinzugefügt, welche die einzusehenden Knoten eines Graphen darstellt.

```
1  private Traverser getPaperMetaData(final Node node) {
2      TraversalDescription td = null;
3      // if we have no authorizations, we compare the permissions between nodes
4      if (this.authorization == null) {
5          td = db.traversalDescription().depthFirst()
6                  .relationships(RelationshipLabel.CONTAINS, Direction.OUTGOING)
7                  .relationships(RelationshipLabel.DOWNLOADABLE_AT,
                       Direction.OUTGOING)
8                  .relationships(RelationshipLabel.INSTANCE_OF)
9                  .relationships(RelationshipLabel.PART_OF, Direction.OUTGOING)
10                 .relationships(RelationshipLabel.PRESENTED_IN, Direction.OUTGOING)
11                 .relationships(RelationshipLabel.PUBLISHED_IN, Direction.OUTGOING)
12                 .relationships(RelationshipLabel.WRITTEN_BY, Direction.INCOMING)
13                 .evaluator(new AccessListEvaluator());
14     } else {
15         td = db.traversalDescription().depthFirst()
16                 .relationships(RelationshipLabel.CONTAINS, Direction.OUTGOING)
17                 .relationships(RelationshipLabel.DOWNLOADABLE_AT,
                       Direction.OUTGOING)
18                 .relationships(RelationshipLabel.INSTANCE_OF)
19                 .relationships(RelationshipLabel.PART_OF, Direction.OUTGOING)
20                 .relationships(RelationshipLabel.PRESENTED_IN, Direction.OUTGOING)
21                 .relationships(RelationshipLabel.PUBLISHED_IN, Direction.OUTGOING)
22                 .relationships(RelationshipLabel.WRITTEN_BY, Direction.INCOMING)
23                 .evaluator(new PermissionEvaluator(this.authorization));
24     }
25     return td.traverse(node);
26 }
```

Listing 4.6: Traversierungsregeln zur Ermittlung der Metainformationsknoten einer Publikation unter Berücksichtigung der Zugriffsberechtigungen

4.5.7. Datenabfrage in OrientDB

Der lesende bzw. suchende Zugriff auf den Datenbestand von OrientDB wird über die Klasse `DataBaseReaderOrientDB` (Abb. 4.10) geregelt. Die Klasse implementiert das Interface `DatabaseReader` und stellt Methoden zur Suche von Journals, Autoren und Publikationen zur Verfügung.

Der Ablauf der Suche erfolgt identisch zu Neo4j, unter Berücksichtigung der Implementierung und Funktionalität von OrientDB. Abbildung 4.11

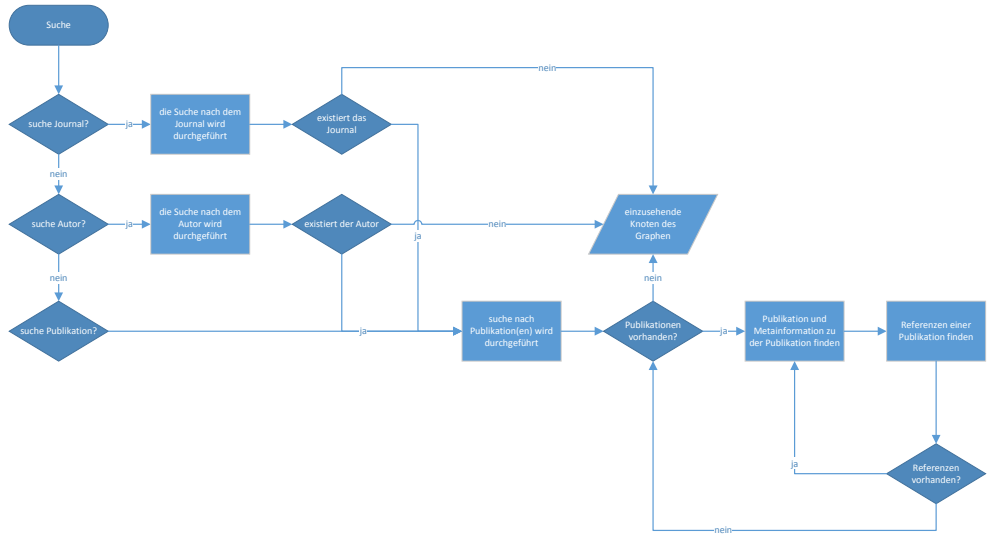


Abbildung 4.11.: Flussdiagramm zur schematischen Darstellung von Suchabfragen in den Graphdatenbanken

zeigt den schematischen Ablauf der Suche. Es erfolgt eine Suchabfrage für ein Journal, einen Autor oder eine Publikation. Anschließend werden, nur bei der Suche von Journalen und Autoren, die mit den jeweiligen Knoten verbundenen Publikationen gesucht. Im nächsten Schritt werden die Metainformationen für die Publikation ausgelesen. Die Referenzen, welche ebenso Publikationen darstellen, werden nachfolgend abgefragt. Rekursiv wird für jede Referenz das Auslesen der Metainformationen initiiert. Sollten nach dem Auslesen der Referenzen keine Publikationen mehr vorhanden sein, wird das Ergebnis zurückgeliefert.

Der wesentliche Unterschied liegt in der Traversierung. Eine Pfadsuche über die Beziehungen einzelner Knoten konnte über die Java API nicht umgesetzt werden. Im Gegensatz zu Neo4j war es nicht möglich, bei der Traversierung Abbruchbedingungen bezüglich der Berechtigungen zu de-

finieren, um die geeigneten Knoten des Pfades zu erhalten. Daraus ergab sich die Konsequenz einer eigenen Implementierung der Traversierung. Der Javacode 4.7 zeigt die Methode der Pfadsuche aus der Klasse `DataBaseReaderOrientDB`.

```
1  private void getPath(Vertex start, Set<Vertex> path, Set<Vertex> visited, int
       depth_limit) {
2      visited.add(start);
3
4      // get the vertices on each depth step
5      Iterable<Vertex> vertexIterator = start.getVertices(Direction.BOTH,
6              new String[] { OrientDbModel.REL_CONTAINS,
                   OrientDbModel.REL_DOWNLOADABLE_AT,
7                      OrientDbModel.REL_INSTANCE_OF, OrientDbModel.REL_PART_OF,
                           OrientDbModel.REL_PRESENTED_IN,
8                      OrientDbModel.REL_PUBLISHED_IN, OrientDbModel.REL_WRITTEN_BY
                           });
9      // got vertices one step deeper decrement depth
10     depth_limit--;
11
12     // if the authorization is not given, take the access stuff from the start
           node
13     String[] startAccess = (this.authorization != null) ? authorization
14             : convertOTrackedSet((OTrackedSet<String>)
                   start.getProperty(NodeProperties.ACCESS));
15
16     for (Vertex v : vertexIterator) {
17         String[] vAccess = convertOTrackedSet((OTrackedSet<String>)
               v.getProperty(NodeProperties.ACCESS));
18
19         // check if we're already visited the vertex and if an intersection
20         // exists and the node is not from type paper
21         if (!visited.contains(v) && checkIntersection(startAccess, vAccess)
22                 && !checkVertexType(v, OrientDbModel.VERTEX_PAPER)) {
23
24             path.add(v);
25
26             // if depth not reached, recursive call of getPath
27             if (depth_limit > 0)
28                 getPath(v, path, visited, depth_limit);
29         }
30     }
31 }
```

Listing 4.7: Implementierung der Tiefensuche zur Traversierung der verbundenen Knoten in OrientDB, unter der Berücksichtigung des feingranularen Berechtigungskonzepts

Der Methode `getPath` wird ein Startknoten, in unserem Fall immer eine Publikation, eine Liste von Knoten welche den Pfad entspricht, sowie eine Liste von Knoten welche bereits besucht wurden und eine maximale Tiefe

für die Suche übergeben. Diese wird benötigt, um nicht den gesamten Graphen zu traversieren, sondern nur die zur Publikation gehörenden Knoten (Metainformationen). Umgesetzt wurde der Algorithmus der Tiefensuche, da dieser auch in Neo4j bei der Traversierung verwendet wird. Zu Beginn jedes Aufrufs werden die mit dem Startknoten, über definierte Kantentypen verbundene Endknoten selektiert (Zeile 5). In der Zeile 14 wird festgelegt, ob ein Vergleich von Berechtigungen zwischen Knoten durchgeführt werden soll, oder die Berechtigungen der einzelnen Knoten gegen definierte Berechtigungen geprüft werden. Wurde eine Suche ohne definierte Berechtigungen gestartet, enthält die Member-Variable `authorziation` keine Werte. Die Zugriffsberechtigungen jedes Endknotens werden mit den in der Variable `authorization` gespeicherten Berechtigungen verglichen. Treffen die drei Bedingungen *"Endknoten noch nicht besucht"*, *"Berechtigungen stimmen überein"* und *"kein Knoten vom Typ"* `PAPER` zu, wird der Knoten dem Pfad hinzugefügt. Wurde die maximale angegebene Tiefe nicht erreicht, wird die Methode `getPath` in Zeile 28, mit dem aktuellen Knoten als neuen Startknoten, rekursiv aufgerufen. Der dadurch erhaltene Pfad spiegelt die Knoten der Metainformationen einer Publikation wider.

Die Informationen zu den Publikationen werden aus den Attributen der jeweiligen Knoten spezifisch ausgelesen und in einem `GraphLayer`-Objekt gespeichert. Jede Publikation, programmtechnisch jedes `GraphLayer`-Objekt, wird in einer `HashMap` Datenstruktur gespeichert, welche die auf Grund der Berechtigungen einzusehenden Knoten enthält.

4.5.8. Datenabfrage in MySQL

Die Implementierung `GraphReaderMySql` des Interfaces `DataBaseReader` stellt Methoden für den lesenden Zugriff auf den Datenbestand in MySQL zur Verfügung. Im Gegensatz zu den Graphdatenbanken OrientDB und Neo4j erfolgt die Traversierung gänzlich mittels der Abfragesprache SQL. Das Flussdiagramm in Abbildung 4.12 zeigt schemenhaft den Ablauf der Suche. Der Ablauf unterscheidet sich von den Graphdatenbanken im Wesentlichen dadurch, dass zuerst die Referenzen der Publikationen ausgelesen werden und erst anschließend die Metainformationen für die gesamten Publikationen selektiert werden. Grund hierfür ist auf der einen

Seite eine einfachere Implementierung der Rekursion in der Datenbankanbindung für MySQL und auf der anderen Seite die Tatsache, dass dieser Ablauf in den Graphdatenbanken ressourcen- und zeitintensiver gewesen wäre, da die Publikationsknoten erneut ausgelesen werden hätten müssen, um einen Bezug zu den Metainformationen herstellen zu können.

Die gesuchten Publikationen bzw. die mit einem Autor oder Journal verbundenen Publikationen werden in der Datenstruktur `HashMap<Integer, GraphLayer>` gespeichert. Für jede ausgelesene Publikation wird ein neuer Map-Eintrag erzeugt. Der Schlüssel besteht aus der Identifikationsnummer der Publikation, der Wert aus einem neuen `GraphLayer`-Objekt. Die zur Publikation gehörenden Referenzen werden rekursiv ermittelt und ebenso in die Datenstruktur eingetragen. Spezifisch werden im Anschluss die Metainformationen für die in der `HashMap` eingetragenen Publikationen, unter der Berücksichtigung der Berechtigungen und einer vorhanden Verbindung, ausgelesen. Das Auslesen der Informationen aus dem Datenbestand erfolgt über sogenannte "Stored Procedures". Dabei handelt es sich um direkt am Datenbankserver gespeicherte Funktionen, die vom Client ausgeführt werden können[5].

Gespeicherte Funktionen bieten eine höhere Effizienz. Sie bringen in den meisten Anwendungsfällen Leistungssteigerung, da eine geringere Datenmengen zwischen Client und dem Datenbankserver ausgetauscht werden muss. Des weiteren bieten "Stored Procedures" Sicherheitsvorteile, da der Client ausschließlich die Möglichkeit hat, vorgefertigte Abfragen auszuführen[6]. Die Sicherheitsvorteile spielen in dieser Umsetzung der Applikation keine Rolle.

Der Codeausschnitt 4.8 zeigt die "Stored Procedure" zur Selektion der Publikationen, unter der Berücksichtigung von Berechtigungen des Start- und Endknoten. Die Selektion der Publikationen für Journals erfolgt identisch, unter der Angabe der spezifischen Tabellen. Die `JOIN`-Operation in Zeile 5 auf die Tabelle `paper_author` repräsentiert die etwaigen Verbindungen zwischen dem gesuchten Autor und seinen Publikationen. In der Zeile 9 werden Übereinstimmungen in den zuvor mittels `JOIN` selektierten Berechtigungen von Autor und Publikation geprüft. Existieren keine Übereinstimmungen

[5] `http://glossar.hs-augsburg.de/Stored_Procedure`, aufgerufen am 2016-01-21

[6] `http://db-engines.com/de/article/Stored+Procedure`, aufgerufen am 2016-01-22

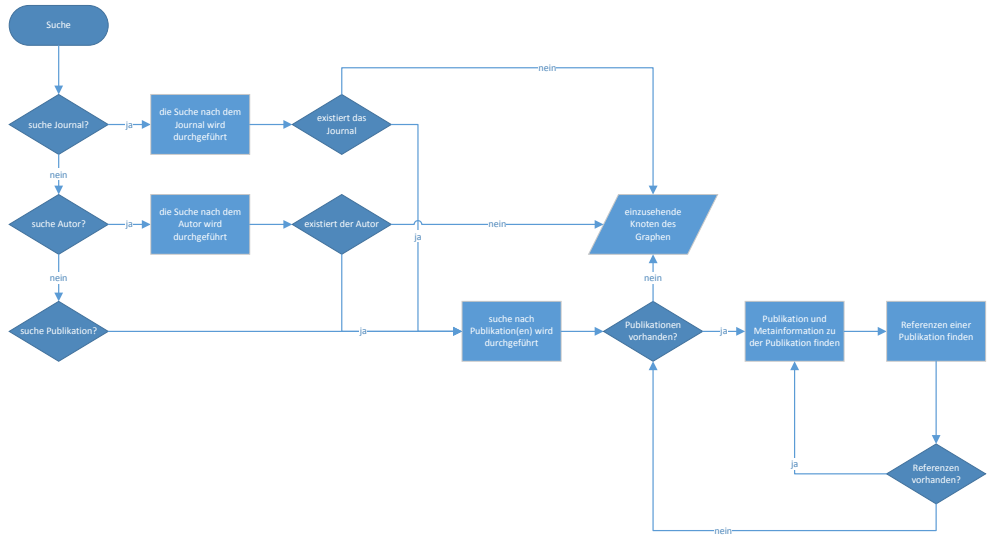


Abbildung 4.12.: Flussdiagramm zur schematischen Darstellung von Suchabfragen in der MySQL Graphdatenbank

bezüglich der Berechtigung bzw. gibt es keine gültige Kante zwischen Autor und Publikation, liefert das gesamte `SELECT`-Statement keine Ergebnisse zurück. Die Modellierung der Berechtigungen in eigenen Tabellen (Abb. 4.3) ermöglicht die Umsetzung von Kanten zwischen unterschiedlichen Knoten, die beispielsweise nicht ausreichend Berechtigungen besitzen. Vereinfacht gesagt bedeutet das, es existiert eine Verbindung zwischen Knoten, der Startknoten darf jedoch nicht auf den Endknoten zugreifen. Des weiteren ist es nicht möglich, in der relationalen Datenbank MySQL Attribute einer Tabelle als Collections, wie in den Graphdatenbanken, zu modellieren.

Dieses Szenario tritt auf, wenn Berechtigungen für die Suche definiert wurden. Dadurch ist es möglich, dass eine Kante zwischen zwei Knoten existiert, jedoch auf den Endknoten aufgrund mangelnder Berechtigungen nicht zugegriffen werden darf. Die Selektion von Publikationen und weiteren "Knotentypen", unter der Berücksichtigung von definierten Berechtigungen, divergiert von der zuvor beschriebenen. Bei der Selektion müssen die übergebenen Berechtigungen berücksichtigt werden. Der Codeausschnitt 4.9 zeigt die Selektion von Publikationen mittels Berechtigungen. Die Selektion unterscheidet sich im wesentlich durch zwei Merkmale: das wegfallen einer `JOIN`-Operation, da keine Berechtigungen zwischen Start- und Endknoten geprüft werden; und das `FIND_IN_SET`-Statement in der Zeile 10, welches die Übereinstimmung der Berechtigungen prüft.

```
1  CREATE DEFINER=`root`@`localhost` PROCEDURE `getPapersOfAuthor`(IN authorName
       VARCHAR(300))
2  BEGIN
3      SELECT p.paper_id, p.original_title, p.normalized_title, p.publish_year,
           p.publish_date, p.doi, p.original_venue_name, p.normalized_venue_name,
           p.paper_rank
4        FROM author a
5  INNER JOIN paper_author pa ON pa.author_id = a.author_id
6  INNER JOIN paper p ON p.paper_id = pa.paper_id
7  INNER JOIN author_access aa ON aa.author_id = a.author_id
8  INNER JOIN paper_access pac ON pac.paper_id = p.paper_id
9  INNER JOIN access ac ON ac.access_id = aa.access_id AND ac.access_id =
       pac.access_id
10     WHERE a.author_name LIKE CONCAT("%", authorName, "%")
11   GROUP BY p.paper_id;
12 END
```

Listing 4.8: Selektierung der mit einem Autor verbundenen Publikationen unter Berücksichtigung der Berechtigungen zwischen Start- und Endknoten

```
1  CREATE DEFINER=`root`@`localhost` PROCEDURE `getPapersOfAuthorWithPermissions`(IN
       authorName VARCHAR(300), IN permissions VARCHAR(300))
2  BEGIN
3                    SELECT p.paper_id, p.original_title, p.normalized_title,
                          p.publish_year, p.publish_date, p.doi, p.original_venue_name,
                          p.normalized_venue_name, p.paper_rank
4                      FROM author a
5            INNER JOIN paper_author pa ON pa.author_id = a.author_id
6            INNER JOIN paper p ON p.paper_id = pa.paper_id
7            INNER JOIN paper_access pac ON pac.paper_id = p.paper_id
8            INNER JOIN access ac ON ac.access_id = pac.access_id
9                     WHERE a.author_name LIKE CONCAT("%", authorName, "%")
10                      AND FIND_IN_SET(ac.name, permissions)
11             GROUP BY p.paper_id;
12 END ;;
```

Listing 4.9: Selektierung der mit einem Autor verbundenen Publikationen unter Berücksichtigung der definierten Berechtigungen

Die Selektion der Knoten, welche mit einer Publikation verbunden sind, werden ebenso über die am SQL-Server gespeicherten Funktionen ausgelesen. Die Berücksichtigung der Zugriffsberechtigungen erfolgt identisch zu den `JOIN`-Operationen in Zeile 7-9 des Codeausschnitts 4.8 unter der Angabe der spezifischen `access`-Tabellen der jeweiligen Knoten.

5. Evaluierung

In diesem Kapitel werden die ausgewählten Datenbanksysteme unterschiedlichen Benchmarktests unterzogen. Ziel des Evaluierungsschwerpunktes ist die Auswertung bzw. Bewertung der Effektivität und Effizienz der unterschiedlichen Technologien unter der Berücksichtigung von Berechtigungskonzepten. Zunächst werden die Testumgebung und die Konfiguration der einzelnen Datenbankmanagementsysteme beschrieben, um anschließend auf die Testergebnisse einzugehen. Darauf folgend wird auf die Methodik bei der Durchführung der Messungen eingegangen um in weiterer Folge die Ergebnisse zu bewerten.

5.1. Testumgebung

Die Datenbanken und das für die Benchmarktests auszuführende Java-Programm, welches Tests bezüglich schreibenden und lesenden Zugriff durchführt, wurden auf einer virtuellen Maschine installiert bzw. der Programmcode dort ausgeführt. Eine virtuelle Maschine emuliert in der Regel ein physische IT-Umgebung. Anfragen für CPU, Speicher, Festplatten und andere Hardware-Ressourcen werden von der Virtualisierungsebene verwaltet. Die Virtualisierungsebene übersetzt die Ressourcen-Anfragen für die darunter liegende physische Hardware (Zimmer, 2005). Ausgewählte Tests bezüglich Schreiben und Lesen wurden zur Ermittlung von Vergleichswerten hinsichtlich unterschiedlicher Systemkonfiguration, auf einer zweiten alternativen virtuellen Maschine durchgeführt. Die Intention dahinter ist, die Auswirkung unterschiedlicher Hardware, im Speziellen die Auswirkung der Leistung von SSD-Festplatten und herkömmlichen festplattenbasierenden Massenspeichern, auf die Leistungsfähigkeit der Technologien zu untersuchen. Zu Gunsten der Lesbarkeit wird in den nachfolgenden Kapiteln die

Basisinformationen über die virtuelle Maschine ‘‘vm_sdd‘‘	
Betriebssystem	Windows Server 2008 R2 Datacenter - Service Pack 1
Systemtyp	64 Bit-Betriebssystem
Prozessor	Intel(R) Xeon(R) CPU 2.80 GHz (4 Prozessoren)
Virtualisierung	VMWare Workstation
Arbeitsspeicher	15,6 GB
Festplattencontroller	HP Smart Array P410i, 1 GB Lese-/Schreibcache
Harddisk	Samsung Pro 850 1 TB, 500 GB der VM zugewiesen

Tabelle 5.1.: Allgemeine Informationen zur Hardwareleistung des mit der Solid-State-Drive konfigurierten virtuellen Systems (vm_ssd) zur Durchführung der Benchmarktests

Hardwarekonfiguration, in welcher die SSD-Festplatte verwendet wird als vm_ssd bezeichnet und die virtuelle Maschine mit dem herkömmlichen Massenspeicher als vm_hdd angeführt. Die Basisinformationen zu verwendeten Betriebssystemen, Prozessor und Speicher der virtuellen Maschinen sind den Tabellen 5.1 und 5.2 zu entnehmen.

5.1.1. Systemkonfiguration Neo4j

Das Graphdatenbankmanagementsystem wird in der Community Edition 2.3.2 in einer eingebetteten Konfiguration verwendet. Der Datenimport erfolgt über die Java-API. Während des Imports erfolgt die Erstellung bzw. Aktualisierung des Lucene-Index für definierte Knotenattribute. Die Suche von Knotenelementen erfolgt über definierte Cypher-Abfragen sowie durch die von der Java-API zur Verfügung gestellten Suchmethoden. Die Abfrage zusammenhängender Knoten wurde mittels Traversal-Framework unter der Verwendung der Breitensuche implementiert. Die eigenen Evaluator-Klassen (Codeausschnitt 4.5) vergleichen bei der Traversierung die Berechtigungen. Für das Auslesen von Daten aus der Graphdatenbank wird Neo4j in einem “Read-Only-Modus“ betrieben.

Die Basiseinstellungen des GDBMS werden unverändert übernommen. Die

Basisinformationen über die virtuelle Maschine ‘‘vm_hdd‘‘	
Betriebssystem	Windows Server 2008 R2 Datacenter - Service Pack 1
Systemtyp	64 Bit-Betriebssystem
Prozessor	AMD Operator Processor 6278, 2.40 GHz (4 Prozessoren)
Virtualisierung	VMWare Workstation
Arbeitsspeicher	12 GB
Festplattencontroller	HP P2000 G3 SAS, 2 GB Lese-/Schreibcache
Harddisk	12x 500GB 6G Serial Attached SCSI im RAID 50 Verbund, 500 GB der VM zugewiesen

Tabelle 5.2.: Allgemeine Informationen zur Hardwareleistung des mit dem herkömmlichen Massenspeicher konfigurierten Systems (vm_hdd) zur Durchführung der Benchmarktests

Größe des Heap-Speichers für die “Java Virtual Machine“ wird entsprechend den Leistungsrichtlinien von Neo4j[1] auf einen Mindestwert von 8 GB festgelegt. Die Stackgröße entspricht dem Standardwert von 1024 MB. Darüber hinaus wird ein Page-Cache mit einer Größe von 2048 MB verwendet. Als Garbage-Collector wird der Concurrent Mark and Sweep Compactor verwendet.

5.1.2. Systemkonfiguration OrientDB

OrientDB, in Community Edition - Version 2.1, wird ebenfalls in einer eingebetteten Konfiguration betrieben. Das GDBMS ermöglicht Operationen auf den Datenbestand mittels Transaktionen, oder gänzlich ohne. Für den Vergleich der unterschiedlichen Technologien wird das System im Transaktionsmodus verwendet. Der Datenimport erfolgt wie in Neo4j über die Java-API. Ein Datenbankindex für definierte Attribute bestimmter Knotentypen wird beim Einlesen der Daten erstellt bzw. aktualisiert. Die Suche

[1] http://neo4j.com/docs/stable/performance-guide.html, aufgerufen am 2016-03-05

erfolgt mittels der proprietären SQL-Abfragesprache und den von der Java-API zur Verfügung gestellten Suchfunktionen. Zusammenhängende Knoten werden über die eigenständige Implementierung der Breitensuche ermittelt (Codeausschnitt 4.7).

Die Heap-Größe in OrientDB wurde mit 6 GB und der Disc-Cache mit 8 GB festgelegt[2].

5.1.3. Systemkonfiguration MySQL

Das relationale Datenbanksystem wird in der Community Edition 5.7 verwendet. Als Speichersubsystem wird InnoDB verwendet. Der Zugriff auf die Datenbank erfolgt über den MySQL-JDBC Treiber. Das Schreiben der Graphstruktur in die Datenbank erfolgt über Prepared Statements. Die Suche sowie die Abfrage zusammenhängender Knoten erfolgt über Stored Procedures. Beim Schreiben werden Indizies erstellt bzw. aktualisiert.

Die Standardkonfiguration für MySQL wurde weitestgehend übernommen. Auf die Definition von Fremdschlüssel wurde verzichtet, um Constraints zu vermeiden. Die Anzahl an Hintergrundprozesse für lesen und schreiben wurden auf 16 festgelegt (Parameter `innodb_read_io_threads` und `innodb_write_io_threads`). Der Parameter zur Nebenläufigkeit von Threads `innodb_thread_concurrency` wurde auf den Wert 0 gesetzt. Dabei entscheidet die InnoDB-Engine selbständig über die optimale Anzahl an Threads, die gleichzeitig auf die InnoDB zugreifen können.

5.1.4. Datengrundlage

Der von Sinha u. a. (2015) veröffentliche Datensatz wird zur Gänze in einer relationalen Datenbank gespeichert. Dies ermöglicht das Extrahieren von Teilgraphen unterschiedlicher Größe, welche als heterogene, zusammenhängende Informationen in den unterschiedlichen Datenbanksystemen abgebildet werden können. Die Informationen müssen dementsprechend

[2] `http://orientdb.com/docs/2.1/Performance-Tuning.html`, aufgerufen am 2016-03-05

Klassen	Anzahl der Publikationen x 1.000				
	1	5	25	50	100
PAPER	17.162	74.757	354.579	670.402	1.265.970
JOURNAL	492	1.700	4.659	6.642	8.884
KEYWORD	1.638	5.197	12.929	18.001	23.904
PAPER_URL	6.867	34.944	173.434	347.417	693.382
FIELD_OF_STUDY	1.451	4.243	9.556	12.709	16.055
CONFERENCE	7	25	104	182	283
CONFERENCE_INSTANCE	7	25	104	182	283
AUTHOR	3.244	16.641	78.500	153.360	297.584
AFFILIATION	364	1.224	3.095	4.351	5.859
$\sum$	31.232	138.756	636.960	1.213.246	2.312.204
Wachstumsfaktor		3,44	3,59	0,90	0,91

Tabelle 5.3.: Anzahl der unterschiedlichen Knotenklassen der verschiedenen Datenbestände in Abhängigkeit zur Publikationsanzahl

aufbereitet werden, um eine Abbildung der Daten in der jeweiligen Datenbank zu ermöglichen. MySQL bietet im Gegensatz zu Neo4j und OrientDB keine Möglichkeit, die Informationen als Graph zu verwalten. Unterschiedliche Knoten- und Kantentypen werden daher in eigenen Relationen der Datenbank abgebildet. Attribute eines Knoten bzw. einer Kante werden als Attribut der Relation gespeichert.

Ein wichtiges Kriterium bei der Durchführung der Tests ist das Leistungsverhalten der Datenbanken bei steigendem Datenvolumen. Die Größe der Teilgraphen wird über die Anzahl der Publikationen geregelt. Durch die Anzahl der Publikationen wurde versucht, ein möglichst konstantes Wachstum hinsichtlich Knoten- und Kantenanzahl zu erreichen. Die Tabellen 5.3 und 5.4 zeigen den Wachstum der Knoten und Kanten, gruppiert nach ihren jeweiligen Klassen.

Klassen	Anzahl der Publikationen x 1.000				
	1	5	25	50	100
MEMBER_OF	1.034	5.533	25.504	50.921	99.864
WRITTEN_BY	3.248	16.763	80.922	161.599	325.369
PART_OF	1.958	9.278	45.993	93.438	186.866
CONTAINS	2.085	9.823	48.537	98.581	196.038
PUBLISHED_IN	578	2.940	14.712	29.301	58.620
DOWNLOADABLE_AT	6.867	34.944	173.434	347.417	693.382
REFERENCED_BY	16.269	71.661	357.552	704.615	1.424.386
PRESENTED_IN	7	25	135	297	595
INSTANCE_OF	7	25	104	182	283
$\sum$	32.053	150.992	746.893	1.486.351	2.985.403
Wachstumsfaktor		4,71	4,95	1,99	2,01

Tabelle 5.4.: Anzahl der unterschiedlichen Kantentypen der verschiedenen Datenbestände in Abhängigkeit zur Publikationsanzahl

5.2. Methodik

Für die Durchführung der Leistungstests wurde im Rahmen der Arbeit ein Java-Framework entwickelt, welches es erlaubt, individuelle Benchmarktests durchzuführen. Die Ausführungsreihenfolge kann beliebig gewählt werden. Die Leistungstests wurden mit Hilfe des Java-Frameworks JUnit [3] implementiert. Die Benchmarktests wurden für jede Kombination aus Graphengröße und Datenbanktechnologie durchgeführt. Als kritischen Punkt bei Benchmarktests bezüglich Datenbanken bezeichnen Dominguez-Sal u. a. (2011) den Umgang mit Caches, welche von Hardware, Betriebssystem und DBMS genutzt werden. Zur Gewährleistung der Vergleichbarkeit der Tests, wurden die Operationen für jede Kombination jeweils dreimal wiederholt. Das arithmetische Mittel der gemessenen Werte liefert das Endergebnis. Gemessen werden die Ausführungszeit und die CPU-Auslastung sowie geschriebene und gelesene Bytes während der Ausführungszeit. Die Ermittlung der Ausführungszeit erfolgt direkt im Java-Framework über die statische Methode `System.nanoTime()`. Die CPU-Auslastung und E/A in Bytes werden mittels den von Windows zur Verfügung gestellten "Performance Monitor"[4] aufgezeichnet. Überwacht werden zwei Prozesse: einerseits die Java Virtual Machine und andererseits MySQL als eigenständiger Prozess. Neo4j und OrientDB werden im eingebetteten Modus betrieben, dadurch erfolgen die Operationen direkt in der JVM. Dem zu Folge wird für die Graphdatenbanken nur die JVM überwacht. MySQL hingegen benötigt neben der JVM einen eigenständigen Prozess für Schreib- und Leseoperationen, wodurch eine Überwachung des eigenständigen MySQL-Prozesses ebenso von Nöten ist.

Für jede Technologie und Datenbankgröße erfolgt im ersten Schritt der Datenimport, um anschließend Suchabfragen unter der Berücksichtigung des feingranularen Berechtigungskonzepts und ohne Vergleich von definierten Berechtigungen absetzen zu können. Die Anfragen werden sequentiell ausgeführt, wodurch beim Import ein konkurrierender Zugriff auf die Datenbasis ausgeschlossen wird und alle Ressourcen, soweit als möglich, vollständig der jeweiligen Technologie zur Verfügung stehen.

[3] `http://junit.org/`, aufgerufen am 2016-03-07

[4] `https://technet.microsoft.com/en-us/library/cc749249.aspx`, aufgerufen am 2016-03-07

Die Benchmarktests werden für jede Graphgröße (Anzahl an Publikationen) nach folgendem Protokoll sequentiell durchgeführt. Die unterschiedlichen Konfigurationen an Suchdurchläufen werden zum Nachweis der Funktionalität des feingranularen Berechtigungskonzepts durchgeführt. Differenzen in der Summe der zurückgegebenen Entitäten im Endergebnis der Suchanfrage demonstrieren die Funktionalität. Der in Punkt drei definierte Suchlauf dient zur Analyse der Laufzeit in Abhängigkeit der Anzahl der Knotentypen. Gesucht wird der identische Datensatz, ausgehend von unterschiedlichen Startknoten.

1. Import der Datengröße N = Anzahl der Publikationen
    a) drei Wiederholungen des Imports pro Technologie
2. zwei Durchläufe an Suchanfragen; erster Durchlauf ohne Berechtigungen, zweiter unter Berücksichtigung von Berechtigungen; die Suchwerte werden für jede Datengröße zufällig gewählt; Berechtigungen werden je Suchwert zufällig definiert
    a) drei Wiederholungen der Suche nach Autoren je Datenbanktechnologie
    b) drei Wiederholungen der Suchanfrage nach Publikationen je Technologie
    c) drei Wiederholungen der Suchanfrage nach Journals je Datenbanktechnologie
3. zwei Durchläufe an Suchanfragen für einen definierten Datensatz; Suchwerte für Autoren, Journal und Publikation sind für jede Datengröße identisch und liefern für jede Suchanfrage den identischen Datensatz je Datenbankgröße und Technologie; erster Durchlauf erfolgt ohne Berücksichtigungen der Berechtigungen, zweiter inkl. Berechtigungen; Berechtigungen werden definiert und für jede Datengröße verwendet
    a) drei Wiederholungen der Suche nach Autoren je Datenbanktechnologie
    b) drei Wiederholungen der Suchanfrage nach Publikationen je Technologie
    c) drei Wiederholungen der Suchanfrage nach Journals je Datenbanktechnologie

Die ermittelten Messwerte werden gespeichert und zur Berechnung und Generierung von Statistiken und Plots in Microsoft Excel und Python weiterverarbeitet.

Tabelle 5.5 beschreibt die jeweiligen JUnit-Tests zur Leistungsbewertung von Einfüge- und Suchoperationen der unterschiedlichen Technologien. Der Import erfolgt über die Java-API der jeweiligen Technologie. Die Suchanfragen wurden mittels der proprietären Abfragesprachen realisiert. Die Traversierung der verbunden Knoten erfolgt in Neo4j mittels dem Traversal-Framework. Eine Traversierung mittels Framework wird in OrientDB und MySQL nicht unterstützt. Die Ermittlung der vernetzten Knoten erfolgt in OrientDB mittels eigenständiger Implementierung der Breitensuche (siehe Kapitel 4). Vernetzte Knoten werden in MySQL mittels `SELECT`-Statements ausgelesen. Erfolgt eine Suche unter Berücksichtigung des feingranularen Berechtigungskonzepts, muss es eine Überschneidung der Berechtigung des Knoten und der für die Suche definierten Berechtigungen gegeben, um in die Ergebnisliste aufgenommen werden zu können. Für die Suche ohne Berechtigungen wird eine Überschneidung der Berechtigungen des Start- und Endknoten einer Kante überprüft. Bei einer Schnittmenge der Berechtigungen wird der Endknoten in das Ergebnis aufgenommen. Eine detaillierte Beschreibung des umgesetzten Berechtigungskonzeptes ist im Kapitel 4.3 zu finden.

Name	Beschreibung
`doImportNeo4j`	Importieren der Testdaten in die Graphdatenbank Neo4j unter Berücksichtigung der Transaktionsgröße. Die Transaktion wird nach 20.000 Publikationen abgeschlossen und eine neue Transaktion erstellt
`doImportOrientDB`	Importieren der Testdaten in die Graphdatenbank OrientDB. Ein Abschluss der Transaktion erfolgt nach der Erstellung eines Knoten.

`doImportMySQL`	Importieren der Testdaten in die relationale Datenbank MySQL. Jede Insert-Operation wird per Transaktion abgeschlossen.
`doAuthorSearch`	Suche nach einem spezifischen Autorenknoten und den mit ihm direkt und indirekt verbundenen Subknoten unter der Berücksichtigung von Berechtigungen. Die Suche kann nach einem exakten Autor oder einem Muster erfolgen.
`doAuthorSearchPermissions`	Suche nach einem spezifischen Autorenknoten und den mit ihm direkt und indirekt verbundenen Subknoten unter der Berücksichtigung von definierten Berechtigungen. Die Suche kann nach einem exakten Autor oder einem Muster erfolgen.
`doPaperSearch`	Exakte- oder Mustersuche nach einer Publikation und der mit ihr direkt und indirekt verbunden Subknoten unter der Berücksichtigung von Berechtigungen.
`doPaperSearchPermissions`	Exakte- oder Mustersuche nach einer Publikation und der mit ihr direkt und indirekt verbunden Subknoten unter der Berücksichtigung von definierten Berechtigungen welche als Parameter übergeben werden.
`doJournalSearch`	Suche nach einem exakten Journal, oder Mustersuche eines Journals und dessen direkt und indirekt verbundenen Subknoten unter der Berücksichtigung von Berechtigungen.

`doJournalSearchPermissions`	Suche nach einem exakten Journal, oder Mustersuche eines Journals und dessen direkt und indirekt verbundenen Subknoten unter der Berücksichtigung von definierten Berechtigungen, welche als Parameter übergeben werden.

Tabelle 5.5.: Name und Beschreibung der im Java-Framework definierten Benchmarktests zur Messung von Ausführungszeit und Auslastung für Datenimport und Suche

5.3. Ergebnisse

Im nachfolgenden Kapitel werden die Ergebnisse der Messungen des Benchmarks vorgestellt und diskutiert. Die errechneten Mittelwerte der Messergebnisse werden entweder in Diagrammform oder in Tabellen visualisiert.

Import Die Größe des Graphen für die Ermittlung der Ausführungszeit des Datenimport wurde über die Anzahl an Publikationen geregelt. Die Abbildungen 5.1 und 5.2 veranschaulichen das Datenvolumen und die Größe der Datenbanken. Einerseits in absoluter Häufigkeit der erstellten Elemente in Abhängigkeit zu der Anzahl an Publikationen. Andererseits in der tatsächlich beanspruchten Speichergröße der einzelnen Testsysteme. Es hat sich gezeigt, dass mit zunehmender Anzahl an Publikationen die Menge an zu erstellender Knoten sinkt, jedoch der Anteil an Kanten steigt. Grund hierfür ist der verhältnismäßig geringe Datenbestand an Knoten vom Typ `JOURNAL`, `CONFERENCE`, `CONFERENCE_INSTANCE`, `AFFILILIATION` und `FIELD_OF_STUDY`, sowie die Tatsache, dass eine Vielzahl an Publikationen bereits als Referenz einer anderen Publikation existieren. Dadurch müssen nur noch neue Kanten zwischen den bestehenden Knoten erstellt werden, jedoch nicht die Knoten an sich.

Die Ermittlung der Ausführungszeit für den Datenimport der definierten Graphgrößen wurde auf der dafür vorgesehenen Standardhardware (vir-

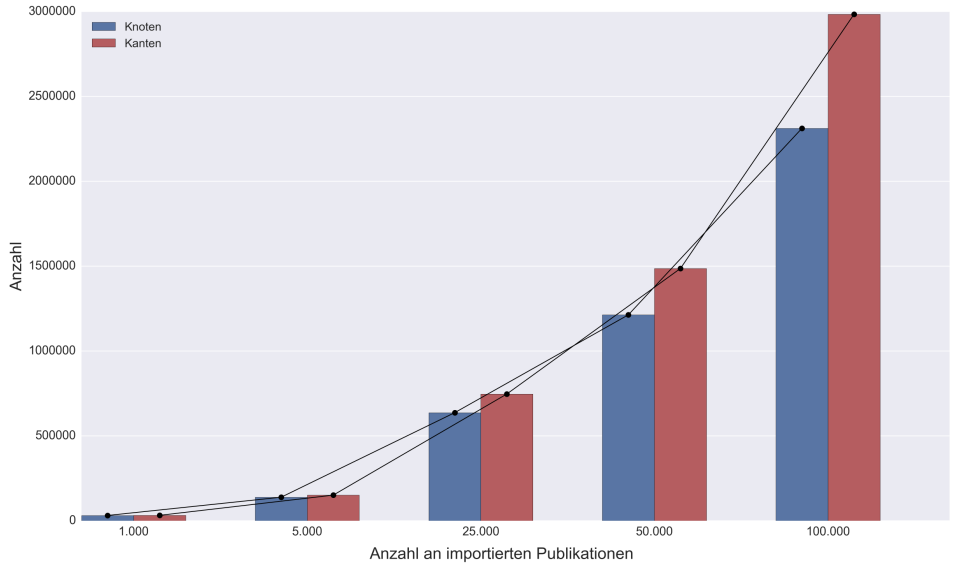


Abbildung 5.1.: Anzahl erstellter Knoten und Kanten in Abhängigkeit zur Publikationsanzahl

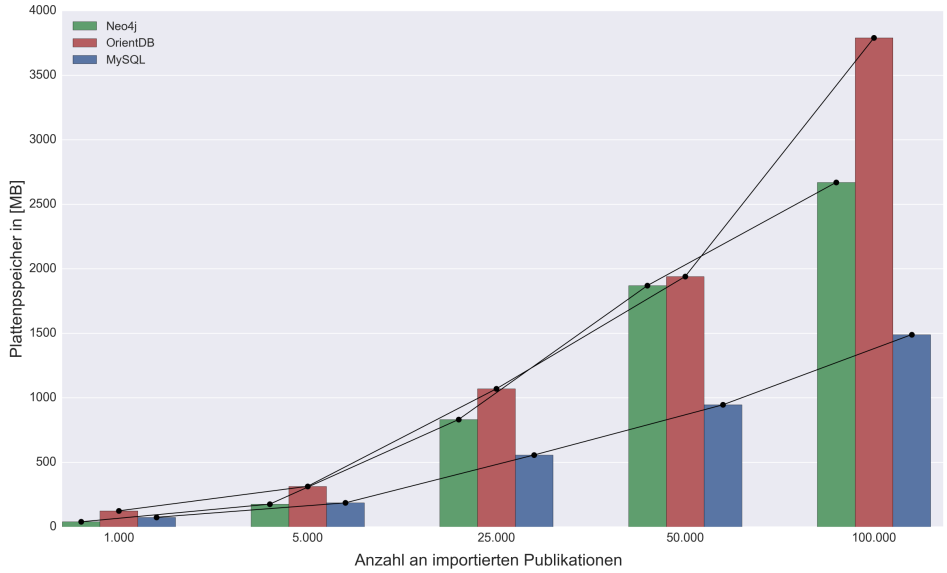


Abbildung 5.2.: Verwendeter Festplattenspeicher der Datenbanken in Abhängigkeit der eingefügten Publikationen

tuelle Maschine `vm_ssd`) durchgeführt. Vergleichswerte der Laufzeiten für den Import von 1.000 und 5.000 Publikationen wurden auf der alternativen Hardware `vm_hdd` (Tab.: 5.2) ermittelt. Die Ausführungszeit wurde aus dem arithmetischen Mittel der einzelnen Durchläufe je Datenbanktechnologie errechnet. Die Berechnung des arithmetischen Mittels der Ausführungszeit des Datenbankimports konnte für Neo4j bei einer Datenbankgröße von 50.000 und 100.000 nicht durchgeführt werden, da ein Import der Daten einen Wert größer 48 Stunden überschritt. Die Daten wurden einmalig für Suchanfragen importiert und die exakte Messwerte des Importvorganges als Ergebnis herangezogen. Die Abbildung 5.4 zeigt die logarithmisch skalierte Dauer für das Einfügen von Knoten und Kanten. OrientDB und MySQL zeigen dabei ein annähernd lineares Verhalten. Es wird verdeutlicht, dass die Graphdatenbank Neo4j mit zunehmender Datenmenge bedeutend schlechter skaliert. Dies ist vermutlich auf die für Neo4j verwendetet Transaktionsgröße zurückzuführen. Im Gegensatz zu Neo4j wird in MySQL und OrientDB, aufgrund von Software abhängigen Limitierungen, eine Transaktion nach jeder Publikation abgeschlossen. In Neo4j erfolgt ein Abschluss der Transaktion jedoch erst nach 20.000 Publikationen. Dies hat zur Folge, dass große Transaktionen geschrieben werden müssen. Für große Mengen an Entitäten sollte der Ansatz von Transaktionen vermieden werden[5]. Deutlich zu erkennen ist, dass die relationale Datenbank am Geringsten von steigender Knoten- und Kantenanzahl beeinflusst wird.

Unterschiede ergeben sich bei der Laufzeit für den Datenimport auf den unterschiedlichen Konfigurationen. Die Ausführungszeiten für den Datenimport auf unterschiedlichen Konfigurationen der virtuellen Maschine sind tabellarisch in 5.6 aufgelistet. Die in Klammer stehenden Werte entsprechen dem Faktor des Geschwindigkeitsvorteils der Konfiguration `vm_hdd` gegenüber der virtuellen Maschine `vm_ssd`.

Die exakten Gründe für den erkennbaren Laufzeitunterschied lassen sich nur schwer eruieren, da die Festplattencontroller und die Festplatten der Konfigurationen mit weiteren virtualisierten Maschinen geteilt werden. Ein markantes Merkmal in den Konfigurationen ist jedoch der Cache des Storage, der sich mit großer Wahrscheinlichkeit positiv auf die Laufzeit des

[5] `http://jexp.de/blog/2013/05/on-importing-data-in-neo4j-blog-series/`, aufgerufen am 2016-03-14

Anzahl Publikationen	Neo4j		OrientDB		MySQL	
	vm_ssd	vm_hdd	vm_ssd	vm_hdd	vm_ssd	vm_hdd
1.000	37,69	32,66 (1,15)	350,8	313,23 (1,12)	87,8	63,45 (1,38)
5.000	666,72	445,88 (1,5)	1.317,43	1.114,57 (1,18)	423,03	300,53 (1,41)

Tabelle 5.6.: Ausführungszeiten in Sekunden für den Datenimport auf den unterschiedlichen Hardwarekonfigurationen. Die Zahl in Klammer entspricht dem Fatkor des Geschwindigkeitsvorteils in Prozent gegenüber der alternativen Konfiguration.

Importvorganges auswirkt. Der direkte Vergleich der Schreibrate zeigt bei der `vm_hdd` Konfiguration während der Importdauer im Schnitt eine größere Anzahl an Bytes pro Sekunde an. Abbildung 5.3 zeigt beispielsweise den direkten Vergleich des Datenimports von 1.000 Publikationen in MySQL. Der schnellere zwei Gigabyte Cache erlaubt höhere Schreibraten pro Sekunde, welche kausal mit der Laufzeit des Einfügevorgangs der einzelnen Datenbanktechnologien zusammenhängt. Die direkten Vergleiche der CPU-Auslastung und der E/A-Bytes für die weiteren Datenbanktechnologien sind im Anhang unter B.6 und B.7 ersichtlich.

Die Diagramme 5.5 und 5.6 zeigen die Leistungsdaten der einzelnen Datenbanksysteme während der Dauer eines Datenimports für 1.000 Publikationen. Die Anzahl an erstellten Knoten und Kanten, kann den Tabellen 5.3 und 5.4 entnommen werden. Verdeutlicht wird die höhere CPU-Auslastung bei den Graphdatenbanken. Dies ist mit großer Wahrscheinlichkeit auf die Einfügeoperationen der Graphdatenbank zurückzuführen. Das Einfügen von Knoten und Kanten ist in Graphdatenbanken wesentlich aufwendiger, als das Erzeugen eines neuen Eintrags in der relationalen Datenbank. Diese Operationen werden direkt am Graph ausgeführt und erklären die im Durchschnitt höhere CPU-Auslastung im Vergleich zur relationalen Datenbank. Auffällig ist zudem die weitaus höhere Anzahl an geschriebenen Eingabe/Ausgabe-Bytes in OrientDB. Werden die zwei Graphdatenbanksysteme untereinander verglichen, ist ersichtlich, dass Neo4j nur einen großen Peak innerhalb des Import-Vorgangs aufweist, welcher sich auf das Schreiben der Daten bei Beendigung der Transaktion zurückführen lässt. Die stetig geschriebenen Bytes in OrientDB und MySQL erklären sich aus der Tatsache, dass jede Publikation und die zugehörigen Metainformationen in einer eigenen Transaktion bearbeitet wird. Der Grund der signifikant

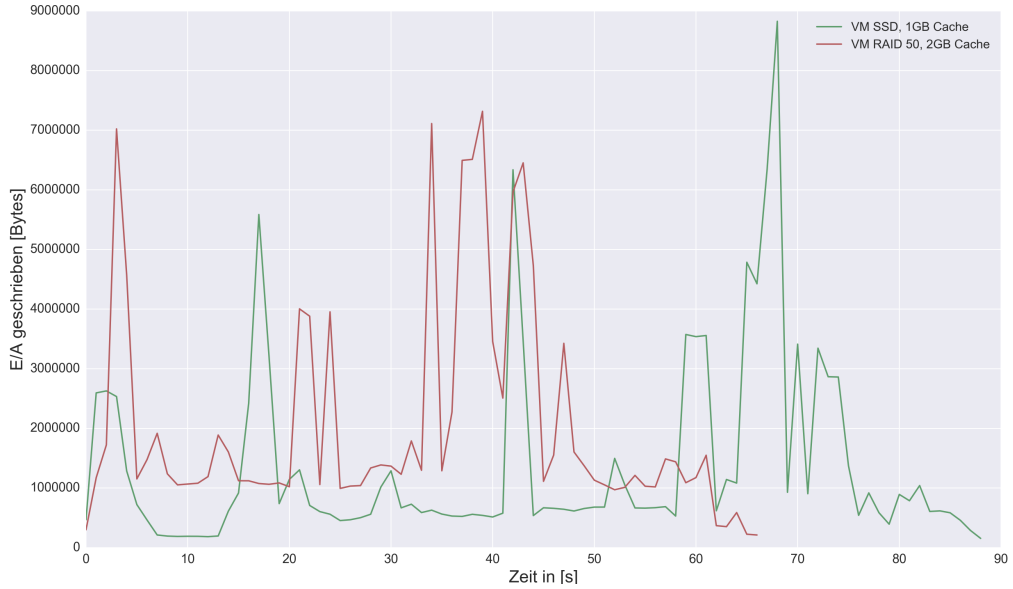


Abbildung 5.3.: Direkter Vergleich der Schreibrate zwischen den beiden Systemkonfigurationen `vm_ssd` und `vm_hdd` während des Importvorganges von 1.000 Publikationen in der relationalen Datenbank MySQL

höheren Anzahl an geschriebenen Bytes in OrientDB lässt sich nicht eindeutig feststellen. Als Ursache kommen das von OrientDB standardmäßig verwendete Cache-Management und die Validierung[6] in Frage. OrientDB behält die am häufigsten verwendeten Elemente im Cache und führt bei jeder Modifikation des Graphen eine Validierung durch, ob bestimmte Regeln für den Graphen eingehalten werden.

Die exakten Werte der Importvorgänge können dem Anhang entnommen werden. Es konnte festgestellt werden, dass die relationale Datenbank bei großen Datenvolumen unter der Verwendung von ACID Transaktionskonzepten und Indizes klare Vorteile gegenüber Graphdatenbanken liefert. Transaktionen und die Erstellung von externen Indizes wie Lucene auf eine große Menge von Knotenattribute sind aufwendig und kosten in GDBMS Zeit.

[6] `http://orientdb.com/docs/2.1/Performance-Tuning-Graph.html`, aufgerufen am 2016-03-14

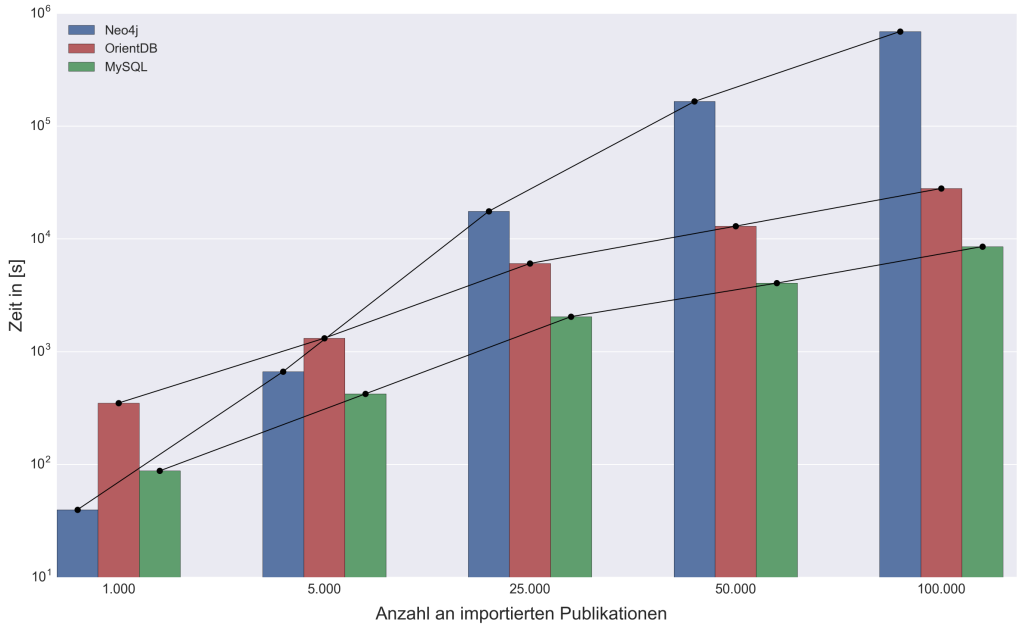


Abbildung 5.4.: Importdauer in Abhängigkeit zur Datenmenge der unterschiedlichen Datenbanksysteme

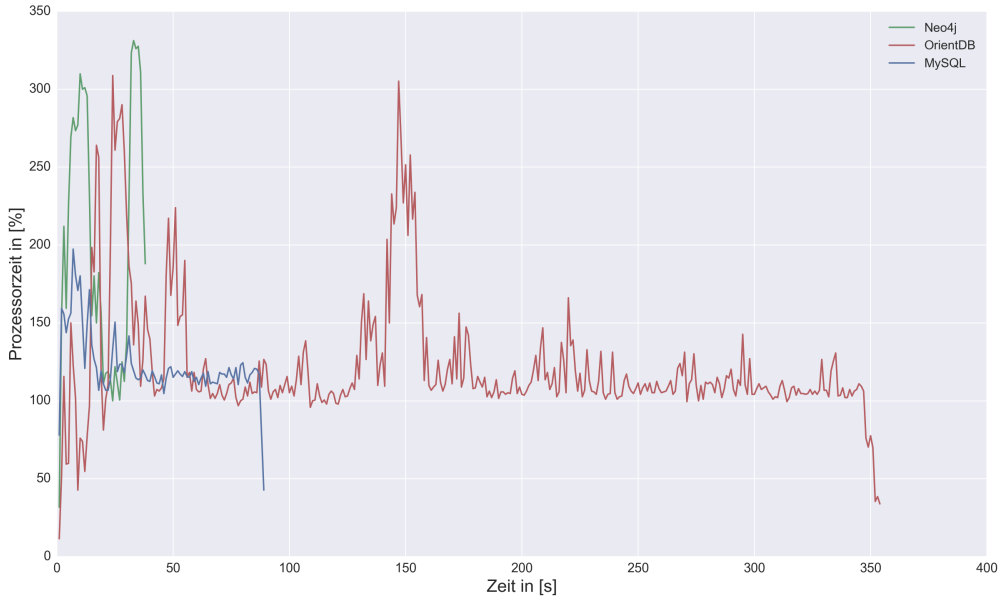


Abbildung 5.5.: CPU Auslastung, während des Importvorganges von 1.000 Publikationen, zur Verarbeitung der Instruktionen der einzelnen Datenbanksysteme

Suche Die Suche in Datenbanksystemen impliziert einen lesenden Zugriff. Für den Benchmark wurden zufällige Startknoten pro Datenbankgröße vom Typ `AUTHOR`, `PAPER` und `JOURNAL` mit einem hohem Vernetzungsgrad ausgewählt. Die Suchanfragen wurden ohne Berechtigungen und unter Berücksichtigung von definierten Berechtigungen durchgeführt, um die Funktionsweise des feingranularen Zugriffskonzepts zu veranschaulichen. Die Abbildung 5.8 zeigt die Antwortszeit der in Tabelle 5.7 definierten Suchanfragen für ein Datenvolumen von 1.000 Publikationen. Die in Klammer beschriebenen Werte entsprechen den zurückgelieferten Entitäten unter Berücksichtigung der Berechtigungen. Zu Gunsten der Lesbarkeit wurde für die gesamten Datenbanktechnologien die Summe an Knoten und Kanten angegeben. Korrekterweise müsste jedoch bei der relationalen Datenbank die Summe der Knoten mit einem Multiplikator von zwei multipliziert werden, um die exakte Summe an Typen von Knoten zu erhalten. Der Multiplikator von zwei ergibt sich aus der Tatsache, dass für die Ermittlung der Knoten auf die typspezifische Berechtigungstabelle eines Knoten zugegriffen werden muss, um die Schnittmenge der Berechtigungen zu ermitteln.

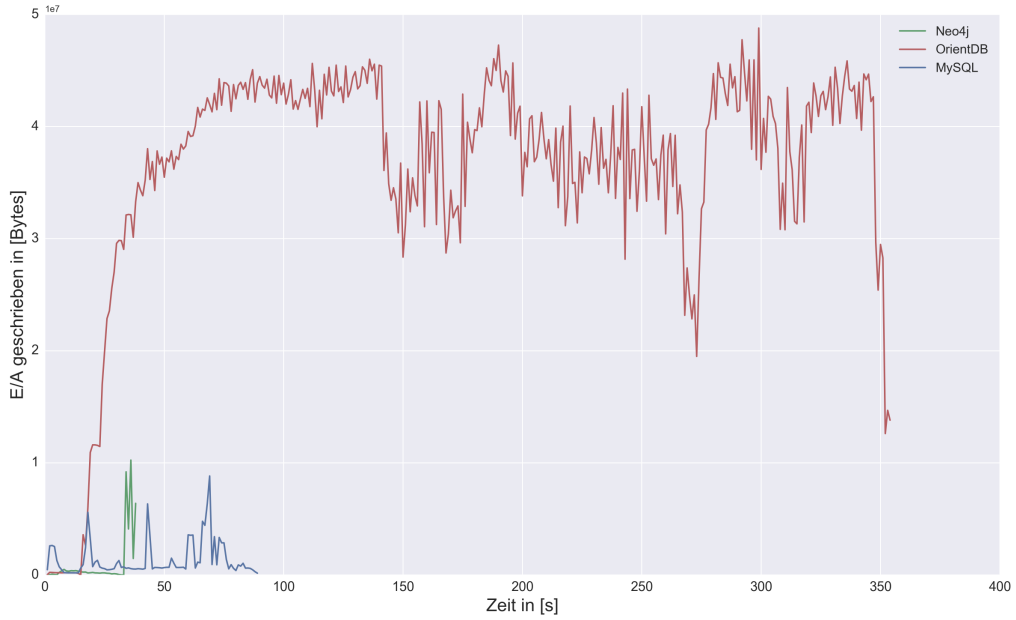


Abbildung 5.6.: Anzahl der Bytes die während dem Importvorgang von 1.000 Publikationen von den einzelnen Datenbanksystemen geschrieben wurden

Exakte Rückgabewerte, gelistet nach Knoten- und Kantentypen können dem Anhang A entnommen werden.

In der Abbildung 5.8 ist ersichtlich, dass die relationale Datenbank bei einem geringen Datenbestand und einem relativ geringen Vernetzungsgrad am besten skaliert. Durch den geringen Vernetzungsgrad sind die `JOIN`-Operationen auf ein Minimum beschränkt. Die Tatsache des geringen Datenbestandes und der wenigen Verbundoperationen spielt der relationalen Datenbank in die Karten und liefert eindeutige Vorteile gegenüber der Traversierung in Graphdatenbanken, welche sich in der Laufzeit widerspiegeln. Vergleicht man die beiden Graphdatenbanktechnologien, hinkt Neo4j bei der Traversierung des Graphen hinterher. Grund für die Differenz ist die Implementierung der Auftragsausführung der Java API in Neo4j. Probeläufe haben ergeben, dass die initiale Suche des Startknotens mittels der proprietären Abfragesprache Cypher in Java bedeutend langsamer ist als die identische Abfrage im Browser Interface. Zum Vergleich zeigt Tabelle 5.8 die Ausführungszeit für die Suche des Autors `yannick letawe` in Java

Suchanfrage	Value	Berechtigung	∑ Kanten	∑ Knoten
Autor	yannick letawe	`com, uk`	24 (24)[7]	25 (25)
Publikation	Efficiently Implementing a Large Number of LL/SC Objects	`doi, dx, org`	10 (8)	9 (7)
Journal	BMC Evolutionary Biology	`https, at, uk`	35 (0)	34 (0)

Tabelle 5.7.: Definition und Rückgabewerte der Suchanfragen für den Datenbestand von 1.000 Publikationen

mittels der Methode `execute()` und mittels der identischen Cypher Query im Browser. Gemessen wurde in Java die Zeit für die Ausführung der Abfrage mittels `execute()`, ohne Berücksichtigung von Verbindungsauf- und abbau.

Die für 1.000 Publikationen definierten Suchanfragen wurden zur Ermittlung der Kosten des Berechtigungskonzepts gänzlich ohne Berechtigungen durchgeführt. Das heißt, ausgehend vom Startknoten werden alle erreichbaren Knoten zurückgegeben. Abbildung 5.7 zeigt die Laufzeiten der Suchanfragen für die unterschiedlichen Datenbanken unter der Berücksichtigung von definierten Berechtigungen, dem Vergleich von Berechtigungen von Start- und Endknoten einer Kante, sowie gänzlich ohne Berechtigungen ("Admin-Rechte"). Die Zahlen über den einzelnen Balken entsprechen der Anzahl der zurückgelieferten Entitäten. Die linke Zahl entspricht den zurückgegebenen Kanten und die Rechte den Knoten. Die tatsächlichen Kosten für das feingranulare Berechtigungskonzept können nicht ermittelt werden, da Suchanfragen mit "Administratorrechten" eine größere Anzahl an Entitäten zurückliefern als Suchanfragen unter der Berücksichtigung von Berechtigungen. Das umgesetzte Berechtigungskonzept wirkt sich auf die Laufzeit der Suchanfragen aus. Das Ausmaß der Auswirkung hängt von der Größe des Datenbestandes und des Vernetzungsgrades ab. Vergleicht man die Laufzeiten der relationalen Datenbank, zeigt die Abbildung längere Laufzeiten bei der Suche mit "Administratorrechten". Der Laufzeitunterschied von wenigen Millisekunden ist auf die größere Anzahl von `JOIN`-Operationen zurückzuführen. Eine Größere Anzahl an zurückgelieferten Entitäten impliziert eine größere Anzahl von `JOIN`-Operationen.

[7]entspricht der Summe von zurückgelieferten Knotenentitäten in GDBMS. Im Bezug

Abbildung 5.7.: Vergleich der Antwortzeiten der unterschiedlichen Suchanfragen für 1.000 Publikationen unter der Berücksichtigung der drei Bedingungen: Suche ohne definierte Berechtigungen, Suche mit definierten Berechtigungen und Suche ohne Berechtigungen (Administratorrechte). Die Zahlen über den Balken entsprechen den zurückgelieferten Entitäten. Die linke Zahl entspricht den Knoten und die Rechte den Kanten.

Ausführungszeit Java [s]	Ausführungszeit Browser [s]
5,91	1,19

Tabelle 5.8.: Ausführungszeiten in Sekunden, der in der Java ausgeführten Cypher Abfrage und der identischen Cypher Abfrage im Browser-Interface

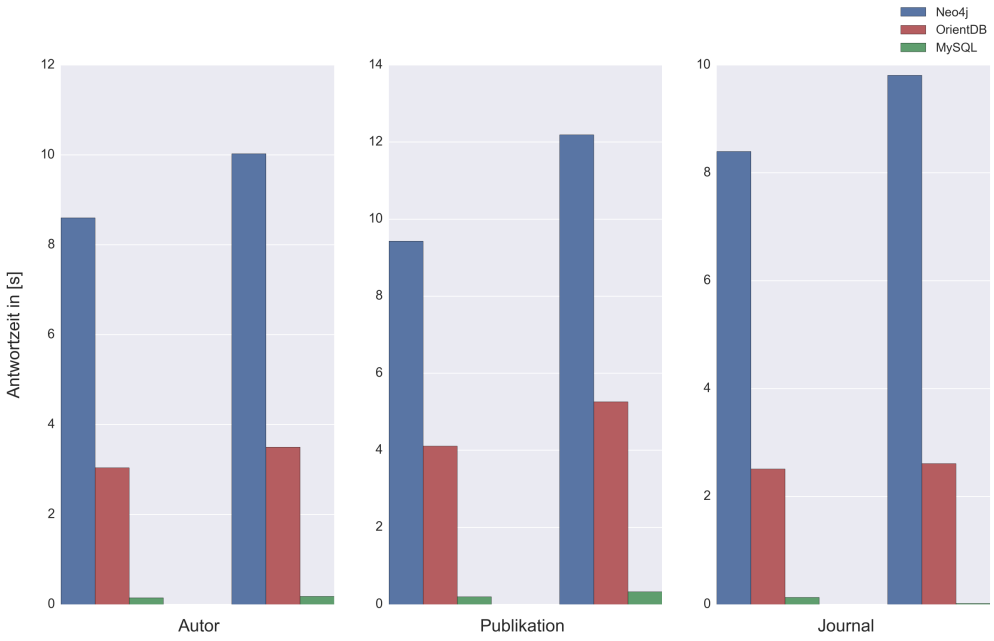


Abbildung 5.8.: Antwortszeit der für einen Datenbestand von 1.000 Publikationen definierten Suchanfragen der jeweiligen Technologien (links: ohne Berechtigungen rechts: mit Berechtigungen)

Steigt die Größe des Datenbestandes und direkt dadurch auch der Vernetzungsgrad des abzubildenden Graphen, verschieben sich die Antwortzeiten. Die relationale Datenbank liefert, wie in Abbildung 5.9 ersichtlich, für die Suchanfrage eines Autors (Tabelle 5.9) schneller das Ergebnis, wird jedoch bei den weiteren Suchanfragen bezüglich Publikation und Journal von den Graphdatenbanken in die Schranken gewiesen. Grund für die schnellere Antwortszeit bei der Autorensuche ist auch hier, wie bereits zuvor beschrieben, der geringe Vernetzungsgrad des Endergebnisses. Des weiteren hat sich gezeigt, dass die relationale Datenbank eine bessere Performance hinsichtlich der Antwortszeit liefert, wenn die Diskrepanz zwischen Knotentypen gering ist. Vereinfacht gesagt, werden große Mengen des gleichen Knotentyps gelesen, wirkt sich dies positiv auf die Laufzeit der relationalen Datenbank aus. Dies ist bei der Autorensuche im geringen Ausmaß der Fall, da nur Publikations- und URL-Knoten sowie ein Autorenknoten gelesen wird.

Auffällig ist bei der Autorensuche der deutliche Laufzeitunterschied zwischen einer Suche mit und ohne definierten Berechtigungen, obwohl das identische Ergebnis zurückgeliefert wird. Grund für den Unterschied in Neo4j und MySQL ist die Implementierung zur Ermittlung der Schnittmenge bei definierten Berechtigungen. Für die Suche unter der Berücksichtigung von definierten Berechtigungen werden unter Neo4j zur Ermittlung der Schnittmenge zusätzliche Operationen auf den Graph ausgeführt, die sich negativ auf die Laufzeit auswirken. Ähnlich verhält sich dabei MySQL. Die Ermittlung der Schnittmenge erfolgt für die Berücksichtigung von Berechtigungen mittels String-Vergleich. Die Summe der String-Vergleiche nimmt allen Anschein nach mehr Zeit in Anspruch, als die Summe der Integer-Vergleiche. In OrientDB unterscheidet sich die Implementierung der Suchvarianten nicht, wodurch sich der Laufzeitunterschied (im Testlauf für die Autorensuche bei einem Datenvolumen von 25.000 Publikationen) schwierig begründen lässt. Die weiteren Suchabfragen ergeben in OrientDB keine signifikanten Laufzeitunterschiede bei einer Suche ohne und mit de-

auf RDBMS muss die Summe an Knoten mit zwei multipliziert werden, um die tatsächliche Anzahl an zurückgelieferten Entitäten aus der relationalen Datenbank zu erhalten. Der Grund dafür ergibt sich aus der Tatsache, dass die Berechtigungen eines Knotentyps in einer eigenen Berechtigungstabelle gespeichert werden und zur Ermittlung der Berechtigungen eines Knoten auf diese zugegriffen werden muss.

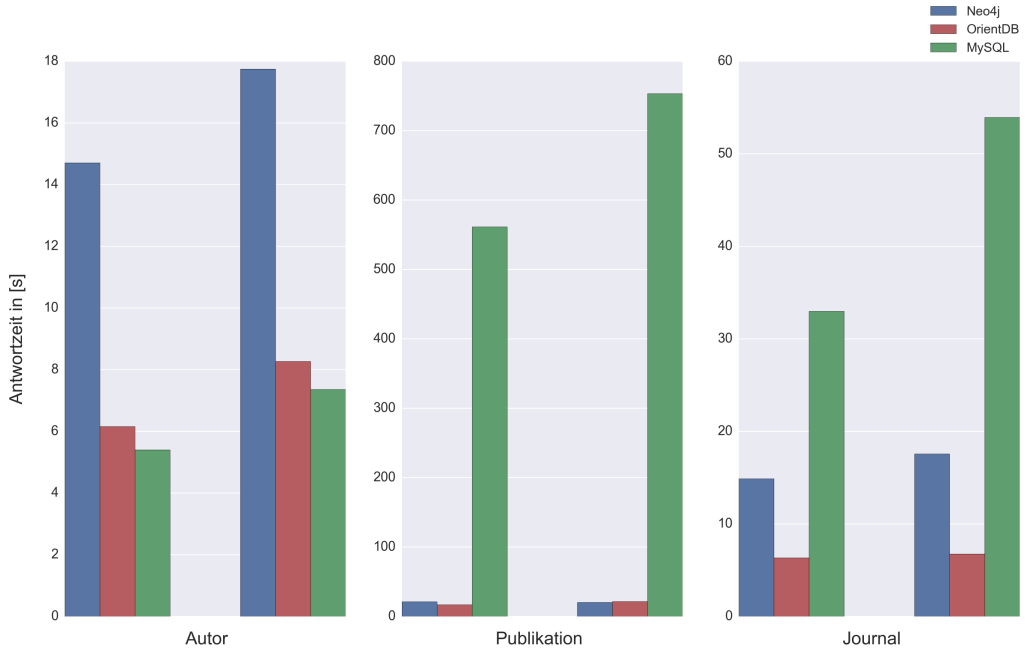


Abbildung 5.9.: Antwortszeit der für einen Datenbestand von 25.000 Publikationen definierten Suchanfragen der jeweiligen Technologien (links: ohne Berechtigungen rechts: mit Berechtigungen)

finierten Berechtigungen, weshalb der große Laufzeitunterschied bei der Autorensuche in Abbildung 5.9 vermutlich durch Ressourcenengpässe zu Stande kommt. Die weiteren Testläufe der unterschiedlichen Graphengrößen sind dem Anhang (B) angefügt.

Wie erwartet, kommt es bei höherem Vernetzungsgrad und größerer Divergenz von Knotentypen (Publikations- und Journalsuche Tab. 5.9) zu erheblichen Performanceinbußen der relationalen Datenbank gegenüber der Graphdatenbank. `JOIN`-intensive Abfragen auf große Datenmengen, zur Ermittlung der verbundenen Knoten, verlangsamen die Suchabfrage. Auch Vicknair u. a. (2010), Robinson, Webber und Eifrem (2013), sowie Edlich u. a. (2010) kommen zu dem Ergebnis, dass Traversierung mittels `JOIN`-Operationen in relationalen Datenbanken die Performance mindern und das eigentlich Designziel von relationalen Datenbanken nicht auf Traversierung und Graphen ausgelegt ist.

Suchanfrage	Value	Berechtigung	∑ Kanten	∑ Knoten
Autor	lee russell	`www`	37 (37)	36 (36)
Publikation	rotation	`http, cambridge, ebooks, org, www`	933 (696)	692 (692)
Journal	BMC Evolutionary Biology	`bmj, com, gut, cabdirect`	3558 (1050)	4036 (1114)

Tabelle 5.9.: Definition und Rückgabewerte der Suchanfragen für den Datenbestand von 25.000 Publikationen

Die Analyse der Leistungsdaten hinsichtlich CPU-Auslastung und verarbeiteten Bytes während einer Suchanfrage wurde an der Autoren- und Publikationssuche, ohne Berücksichtigung des feingranularen Zugriffskonzepts, bei einem Datenbestand von 25.000 Publikationen durchgeführt. Die beiden Suchanfragen ermöglichen eine Analyse hinsichtlich unterschiedlicher Vernetzungsgrade, bei einer durchschnittlichen Datenbankgröße von rund 600.000 Knoten und rund 750.000 Kanten. Die Abbildung 5.10 und 5.11 zeigen die CPU-Auslastung und geschriebenen Bytes bei der Suchanfrage des spezifischen Autors `lee russell`. Die Abbildungen 5.12 und 5.13 zeigt die Leistungsdaten hinsichtlich der Suche nach den Publikationen. Die grafischen Auswertungen für die restlichen Suchanfragen kann dem Anhang B entnommen werden.

Es wird verdeutlicht, dass die CPU-Auslastung für die Dauer der Suchanfrage für Graphdatenbanken wesentlich höher ist als die Auslastung für die relationale Datenbank, unabhängig vom Vernetzungsgrad des Suchergebnisses. Ein Zusammenhang besteht zwischen Vernetzungsgrad und geschriebenen Bytes. Je höher die Dichte, sprich die Anzahl an Knoten und Kanten, desto mehr Bytes werden geschrieben. Immense Auswirkungen hat der Vernetzungsgrad auf die geschriebenen Bytes der relationalen Datenbank. Die Vernetzung erfordert zur Ermittlung verbundener Knoten eine große Anzahl an `JOIN`-Operationen, welche sich auf die Schreibrate auswirken. Die Evaluierung deutet bei der verwendeten Konfiguration und Implementierung darauf hin, dass Suchanfragen in Graphdatenbanken CPU-lastig und in relationalen Datenbanken speicherlastig sind.

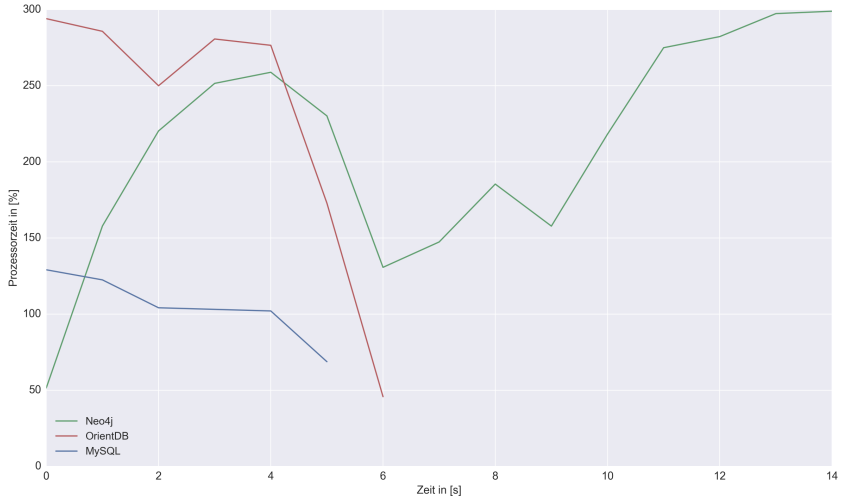


Abbildung 5.10.: CPU-Auslastung der unterschiedlichen Datenbanken während der Suche nach dem Autor `lee russell` und seinen vernetzten Knoten, ohne Berücksichtigung der Berechtigungen bei einer Datenbankgröße von 25.000 Publikationen

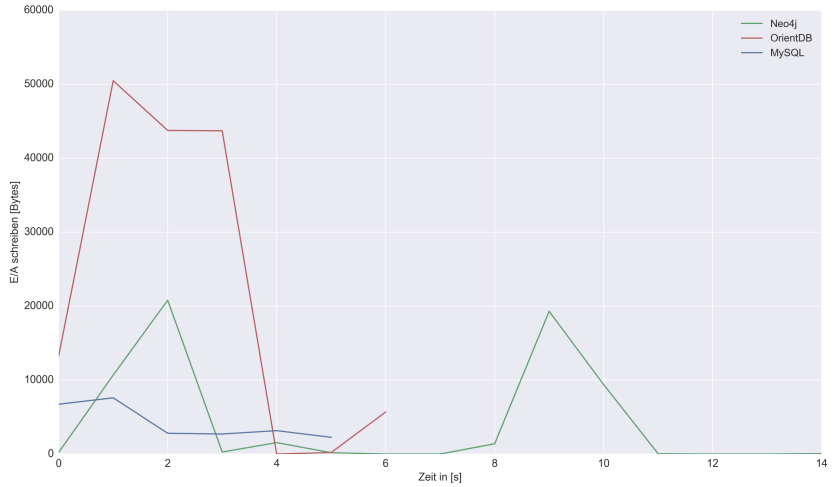


Abbildung 5.11.: Geschriebene Bytes der unterschiedlichen Datenbanken während der Suche nach dem Autor `lee russell` und den mit ihm direkt und indirekt verbundenen Knoten, ohne Berücksichtigung der Berechtigungen bei einer Datenbankgröße von 25.000 Publikationen

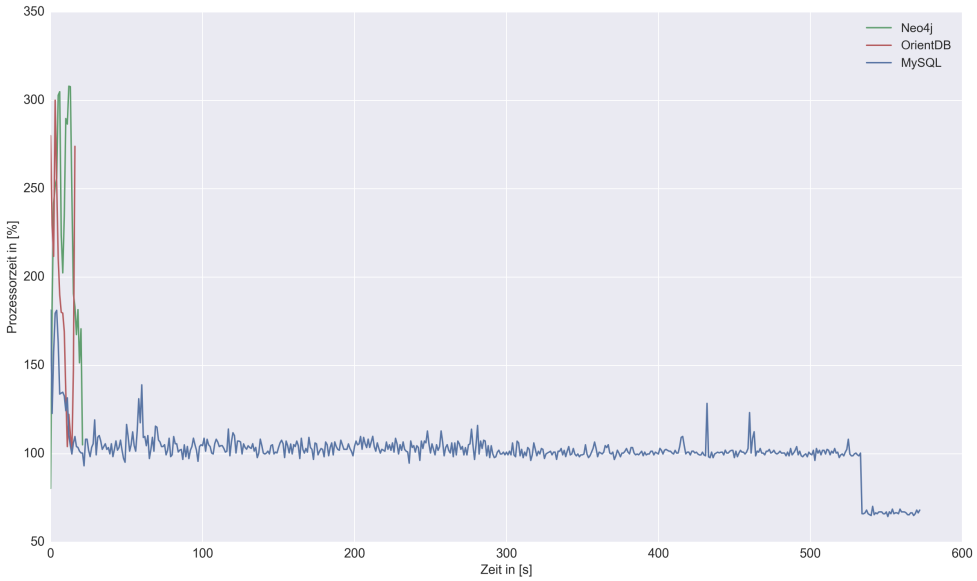


Abbildung 5.12.: CPU-Auslastung der unterschiedlichen Datenbanken während der Suche nach Publikationen mit dem Titel `rotation` und ihren vernetzten Knoten, ohne Berücksichtigung der Berechtigungen bei einer Datenbankgröße von 25.000 Publikationen

Für den Erhalt von Vergleichswerten bezüglich dem Laufzeitverhalten der Suche auf der alternativen Konfiguration `vw_hdd` wurden die in Tabelle 5.9 definierten Suchanfragen ohne Berücksichtigung definierter Berechtigungen auf der selbigen ausgeführt. Wie zu erwarten konnte das Laufzeitverhalten optimiert werden, mit Ausnahme der Suchanfrage für Journals. Diese entwickelte sich in die Gegenrichtung und wurde bedeutend schlechter. Ein mehrmaliges Ausführen der Suche bewirkte keine Verbesserung des Laufzeitverhaltens. Ein Grund für die längere Laufzeit der Suchanfrage konnte nicht ermittelt werden. Tabelle 5.10 zeigt die Laufzeiten der Suchanfragen auf den unterschiedlichen Konfigurationen. Die Werte in Klammer entsprechen dem Faktor der Geschwindigkeitsverbesserung.

Identisch zur Suche von 1.000 Publikationen wurden die definierten Suchanfragen für 25.000 Publikationen unter der Berücksichtigung von Administratorrechten durchgeführt. Abbildung 5.14 zeigt die Laufzeiten der unterschiedlichen Datenbanken und die zurückgelieferten Entitäten der Suchanfrage. Die Zahlen über den Balken stehen für die zurückgelieferten

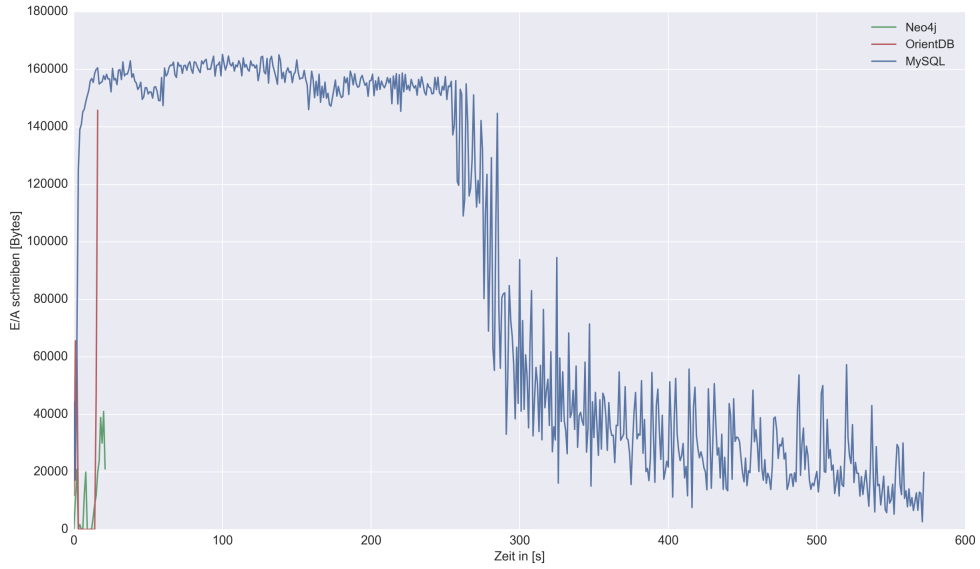


Abbildung 5.13.: Geschriebene Bytes der unterschiedlichen Datenbanken während der Suche nach Publikationen mit dem Titel `rotation` und ihren vernetzten Knoten, ohne Berücksichtigung der Berechtigungen bei einer Datenbankgröße von 25.000 Publikationen

Entitäten, wobei die Linke für die Anzahl der Knoten und die Rechte für Anzahl der Knoten steht. Auch hier sind die tatsächlichen Kosten des Berechtigungskonzepts schwer zu ermitteln, da eine unterschiedliche Anzahl an Entitäten zurückgeliefert wird. Die Autorensuche liefert jedoch für die drei Suchvarianten die selben Ergebnisse. Es hat sich gezeigt, das sich in Neo4j die Umsetzung des Berechtigungskonzepts negativer auf die Laufzeit auswirkt als in der Graphdatenbank OrientDB. Die Zugriffe auf die Attribute der einzelnen Knoten scheinen in OrientDB besser zu skalieren als in Neo4j.

Werden die Zugriffsvarianten für die Autorensuche bei der relationalen Datenbank verglichen, weisen die Suche nach definierten Berechtigungen und die Suche mit Administratorrechten keine Laufzeitunterschiede auf. Die Suche mit definierten Berechtigungen ist jedoch langsamer. Dies zeigt, dass der Vergleich von Strings anscheinend schlechter skaliert als der Vergleich von Integer-Zahlen. Für die Suche nach definierten Berechtigungen wird überprüft, ob die Berechtigungen in der entsprechenden Berechti-

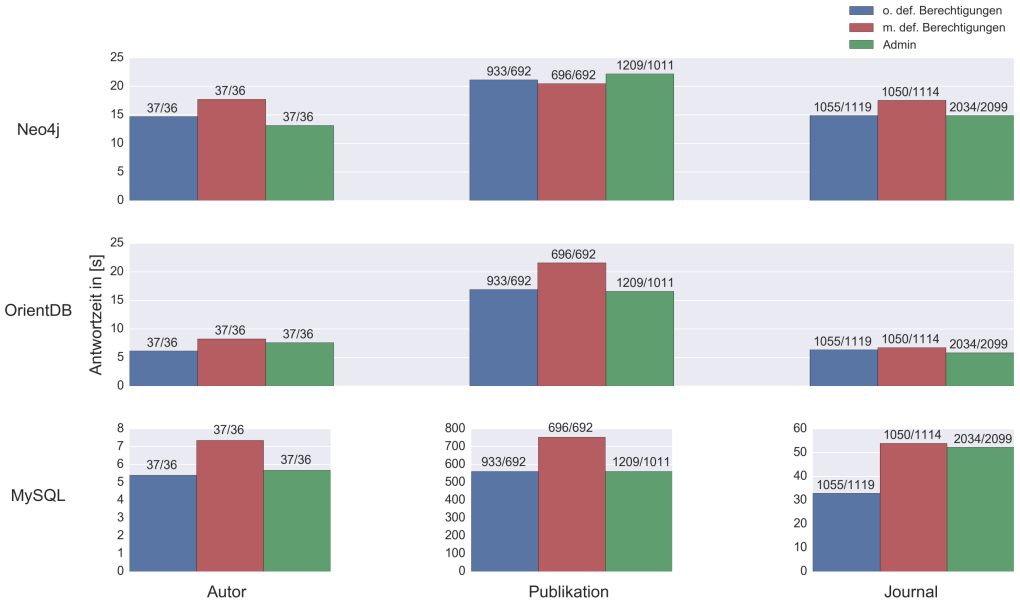


Abbildung 5.14.: Vergleich der Antwortzeiten der unterschiedlichen Suchanfragen für 25.000 Publikationen unter der Berücksichtigung der drei Bedingungen: Suche ohne definierte Berechtigungen, Suche mit definierten Berechtigungen und Suche ohne Berechtigungen (Administratorrechte). Die Zahlen über den Balken entsprechen den zurückgelieferten Entitäten. Die linke Zahl entspricht den Knoten und die Rechte den Kanten.

gungstabelle des jeweiligen Knotentyps vorkommen (Codeausschnitt 4.9). Betrachtet man die Suchanfragen für Publikationen und Journals, ist ersichtlich, dass die Suche mit Administratorrechten, bei einer größeren Anzahl an zurückgegebenen Entitäten, annähernd gleich bzw. besser skaliert als die Suchanfragen mit Berechtigungen. Das lässt darauf schließen, dass die Überprüfung von Berechtigungen bei großen Datenmengen und stark vernetzten Daten wesentlich mehr Zeit in Anspruch nimmt, als bei weniger ausgeprägten Daten.

Gesamtergebnis Sowohl die relationale Datenbank MySQL als auch die Graphdatenbank OrientDB erzielten Ausführungszeiten für den Import von Graphen unterschiedlichen Größenausmaßes, welche für die Evaluierung als akzeptabel angesehen werden können. Einzig Neo4j benötigt lange Ant-

Suchanfrage	Neo4j		OrientDB		MySQL	
	vm_ssd	vm_hdd	vm_ssd	vm_hdd	vm_ssd	vm_hdd
Autor	14, 71	9,68 (1,52)	6,16	5,34 (1,15)	5,4 (1,09)	5,9
Publikation	21,15	18,05 (1,17)	16,91	13,91 (1,21)	531,53	472,17 (1,19)
Journal	14,88	10,95 (1,35)	6,35	4,13 (1,53)	32,99 (1,36)	44,88

Tabelle 5.10.: Laufzeiten in Sekunden, der für 25.000 Publikationen definierten Suchanfragen, auf den unterschiedlichen Hardwarekonfigurationen `vm_ssd` und `vm_hdd`. Die Werte in den Klammern entsprechen den Faktor der Geschwindigkeitsverbesserung in Prozent.

wortszeiten bei Graphen ab einer Größe von rund 1,2 Millionen Knoten und rund 1,5 Millionen Kanten. Dies hängt unter anderem mit der verwendeten Konfiguration und der gewählten Transaktionsgröße zusammen, jedoch ist das Argument, dass die Verwendung von Transaktionen nicht der geeignete Ansatz für große Datenmengen ist[8], nicht außer Acht zu lassen. Generell ist jedoch festzustellen, dass die Verwendung von Transaktionen bei Graphdatenbanken nicht das geeignete Konzept für Datenmengen im Millionenbereich ist. OrientDB wurde hinsichtlich der Transaktionen ähnlich umgesetzt wie MySQL, für jede Publikation und und deren Metainformationen eine neue Transaktion, skaliert jedoch deutlich schlechter als die relationale Datenbank. Im Gegensatz zu MySQL verwenden die GDBMS externe Indexmechanismen, welche sich bei einer Vielzahl an zu indizierenden Knotenattributen augenscheinlich auf das Laufzeitverhalten beim Importvorgang auswirkt. Unabhängig von der verwendeten Hardware ist festzustellen, dass die getestete relationale Datenbank hinsichtlich des Schreib- und Indizierungsvorganges von großen Datenmengen, in der umgesetzten Implementierung den Graphdatenbanken überlegen ist und wesentlich weniger von der Knoten- und Kantenanzahl abhängt.

In Bezug auf Suchanfragen erzielt die relationale Datenbank bei geringen Datenbestand und geringem lokalen Clusterkoeffizienten der Suchknoten schnellere Antwortzeiten. Der Umstand, dass relationale Datenbanken bei Abfragen nach bestimmten Restriktionen gegenüber Graphdatenbanken Vorteile genießt und die Tatsache der wenigen `JOIN`-Operationen bei

[8] `http://jexp.de/blog/2013/05/on-importing-data-in-neo4j-blog-series/`, aufgerufen am 2016-03-16

einem geringen Clusterkoeffizienten spiegeln sich in der Laufzeit wider. Bei höherem Vernetzungsgrad leidet die relationale Datenbank unter dem `JOIN`-Problem. Dies kann unter Umständen mit dem verwendeten Datenbankschema zusammenhängen, welches Daten redundant speichert und sich nicht zur Gänze an die Normalform hält. Mehraufwände der Graphdatenbanken im initialen Datenimport werden durch die erzielten Laufzeiten bei Suchanfragen auf ausgeprägte vernetzte Daten kompensiert. Ein steigender Datenbestand wirkt sich ebenso wie bei einer relationalen Datenbank auf die Suchleistung aus, zum ausgeprägten Teil jedoch nur auf die initiale Suche der Startknoten. Das Laufzeitverhalten bei der Suche nach initialen Startknoten kann anhand des verwendeten Indexmechanismus (Lucene, SB-Tree, Hash-Index) stark variieren. Nach der Selektion der Startknoten spielt die Graphdatenbank ihre Vorteile gegenüber der relationalen Datenbank klar aus. Die Ermittlung der direkt und indirekt mit den Startknoten verbundenen Knoten erfolgt direkt am Graphen über die verbundenen Kanten mittels Traversierung, wodurch performanceintensive `JOIN`-Operationen entfallen. Je höher der Vernetzungsgrad, desto höher der Vorteil der Graphdatenbanken gegenüber der relationalen Datenbank. Dabei hat sich bei der gewählten Implementierung gezeigt, dass die Traversierung graphähnlicher Datenstrukturen nicht zu den Stärken von relationalen Datenbank zählt. Wie zu erwarten war wird deutlich, dass Graphdatenbanken zur Suche von Daten in Graphstrukturen und zur Ermittlung von Pfaden konzipiert wurden.

Vergleicht man die beiden GDBMS im Bezug auf Suchanfragen untereinander, hat sich gezeigt, dass die proprietäre Abfragesprache Cypher von Neo4j gegenüber der SQL ähnlichen Sprache von OrientDB, auch bei der Verwendung ohne Index, Geschwindigkeitsvorteile aufweist. Jedoch ist die Ausführung von Cypher-Code in der Java-API um ein Wesentliches langsamer als die Ausführung der Abfrage im von Neo4j zur Verfügung gestellten Browserinterface. Aus dieser Tatsache heraus ergeben sich Unterschiede bezüglich der Geschwindigkeit bei Suchanfragen zwischen OrientDB und Neo4j.

Die Evaluierung hat gezeigt, dass sich die verwendete Hardware deutlich auf die Leistungsfähigkeit der unterschiedlichen Technologien auswirkt. Es konnte unter der Verwendung unterschiedlicher Festplattenkonfigurationen festgestellt werden, dass die Leistungsfähigkeit der unterschiedlichen Daten-

banktechnologien mit der Leistungsfähigkeit der Hardware steigt. Unter der verwendeten Implementierung hat sich gezeigt, dass die Graphdatenbanken, vor allem zu Beginn des Datenimports, mehr CPU gebunden sind als die relationale Datenbank. Für die Dauer des Datenimports ist OrientDB wesentlich E/A gebundener als die weiteren evaluierten Datenbanktechnologien. Vergleicht man dich Abhängigkeit von CPU und E/A über die Laufzeit der Suche, sind die Graphdatenbanken für die Dauer der Suche vergleichsweise mehr CPU abhängig als die relationale Datenbank. Bei geringem Vernetzungsgrad zeigt OrientDB für die Dauer der Suche höhere E/A-Abhängigkeit als Neo4j und MySQL. Steigt jedoch der Vernetzungsgrad des gesuchten Knoten und dadurch auch die Anzahl der `JOIN`-Operationen, wächst die E/A-Abhängigkeit der relationalen Datenbank.

Die Verwendung von Graphdatenbanken biete sich an, sobald Daten in Form einer Graphstruktur vorhanden sind und die Strukturanalyse von Graphen, sprich die die Traversierung und Ermittlung von Zusammenhängen, im Mittelpunkt der Anwendung steht. Ist eine gewissen Flexibilität des Datenmodells gewünscht, kann die Graphdatenbank ebenso Vorteile bieten. Die in einer Graphdatenbank verwendete Datenstruktur ermöglicht es, neue Entitäten hinzuzufügen, ohne dabei vorhandene Abfragen oder die Funktionalität von Anwendungen zu beeinträchtigen. Die Erweiterung des Datenmodells wird natürlich auch von relationalen Datenbanken unterstützt, jedoch mit dem Unterschied, dass neue Entitäten unter Umständen neue Tabellen darstellen und dies somit eine Änderung der Architektur und Funktionalität nach sich zieht. Für schreibintensive Anwendungen oder Anwendungen im Data-Warehouse Bereich mit großen Datenmengen, sind relationale Datenbanken zu bevorzugen. Sie sind bei Schreiboperationen unter der Verwendung von Transaktionen wesentlich performanter. Stehen Abfragen von Informationen nach bestimmten Restriktionen im Mittelpunkt, wie beispielsweise “finde alle Autoren die mit `a` beginnen“, ist eine relationale Datenbank zu favorisieren, da diese gegenüber Graphdatenbanken wesentlich besser skaliert.

6. Zusammenfasssung, Fazit und Ausblick

Als Ziel der Arbeit wurde eingangs der Vergleich der unterschiedlichen Graphdatenbanksysteme und deren Berechtigungskonzepte formuliert, um diese anschließend anhand ihrer Schreib- und Leseleistung unter Berücksichtigung eines implementierten feingranularen Berechtigungskonzept mit der relationalen Datenbank MySQL zu vergleichen.

Nach der Ausarbeitung der notwendigen Grundlagen betreffend der Funktionsweise von Datenbankmanagementsystemen und grundlegenden Theorien zu Graphen wurden die GDBMS mit dem Fokus auf Neo4j und OrientDB hinsichtlich Datenmodell, Indexstruktur, Zugriffskonzepte und Funktionsumfang verglichen. Anschließend wurden die Berechtigungskonzepte der einzelnen DBMS und die Umsetzung von Berechtigungskonzepten in Graphdatenbanken diskutiert.

Anhand der gewonnenen Basiskenntnisse und eines geeigneten Testdatensatzes wurden Datenmodell und Berechtigungskonzept des Testgraphen definiert, um anschließend mit Hilfe von festgelegten Anforderungen ein Java-Framework zur Evaluierung der Leistungsfähigkeit hinsichtlich Lese- und Schreibperformance der einzelnen Datenbanken zu entwickeln. Das Java-Framework erlaubt es Testdaten in die ausgewählten Testdatenbanken zu schreiben und diese mit oder ohne Berücksichtigung des umgesetzten feingranularen Berechtigungskonzepts auszulesen. Während dem Schreib- bzw. Lesevorgangs werden Laufzeit sowie Leistungsdaten bezüglich CPU-Auslastung und geschriebene Eingabe-/Ausgabebytes pro Sekunde aufgezeichnet.

Die Evaluierungskomponente des Java-Frameworks erlaubt das Ausführen von unterschiedlichen Benchmarktests. Mit Hilfe der Benchmarktests

wurden divergente Größen an Testdaten in die Datenbanktechnologien importiert, um die Schreibleistung derselben zu messen. Zur Gewährleistung der Vergleichbarkeit der Tests wurden die Importvorgänge pro Datengröße und Technologie jeweils dreimal durchgeführt und als tatsächliches Endergebnis der arithmetische Mittelwert der Testläufe berechnet. Suchanfragen ermitteln die Leseperformance unter der Berücksichtigung des umgesetzten feingranularen Berechtigungskonzepts und wurden je Datenbestand randomisiert so gewählt, das ein möglichst hoher Vernetzungsgrad des Endergebnisses sichergestellt wird. Nach dem selben Prinzip wie bei den Importvorgängen werden Suchanfragen wiederholt durchgeführt und aus den Messwerten der Mittelwert für das Endergebnis berechnet.

Im Zuge der Analyse wurden ausgewählte Schreib- und Lesetestverfahren zu Demonstrationszwecken des Einflusses der Hardware in der selben Datenbankkonfiguration auf unterschiedlicher Systemhardware durchgeführt.

Fazit Die evaluierten GDBMS implementieren das Property-Graph-Modell, und verzichten dabei auf ein striktes Schema. OrientDB erlaubt im Gegensatz zu Neo4j die Definition eines Schemas. Die relationale Datenbank verwendet ein striktes Schema. Hinsichtlich der Flexibilität ist die relationale Datenbank gegenüber den Graphtechnologien eingeschränkter. Erweiterungen um Entitäten ziehen meist Änderungen am Datenmodell, etwa durch das Hinzufügen von Tabellen und zusätzlichen Attributen, nach sich. Graphdatenbanken lassen sich ohne Änderungen am verwendeten Datenmodell um Entitäten erweitern, ohne dabei Funktionalität und Anwendungen zu beeinflussen.

Im Zuge der Evaluierung hat sich gezeigt, dass die Schreibperformance beim Einfügen von Knoten und Kanten in die Graphdatenbanken schlechter skaliert als die relationale Datenbank. Schreiboperationen sind in GDBMS wesentlich aufwendiger als das Hinzufügen von Zeilen in einer relationalen Datenbank. Speziell in Neo4j hängt die Laufzeit für die Importdauer signifikant von der Anzahl der erzeugten Knoten und Kanten sowie von der Größe der gewählten Transaktion ab. Beide Graphdatenbanken unterstützen ACID, jedoch ist die Verwendung des Transaktionsmechanismus, zur Sicherung der Datenintegrität beim Import von großen Datenmengen

zu überdenken. Externe Indexmechanismen in Neo4j und OrientDB wirken sich augenscheinlich negativ auf die Laufzeit aus, wodurch sich die relationale Datenbank beim Schreiben von großen Datenmengen klare Vorteile gegenüber der Graphdatenbanken verschafft. Relationale Datenbanken sind für schreibintensive Anwendungen, welche konsistente Daten erfordern, zu favorisieren.

Seit den siebziger Jahren wird die relationale Datenbank als etablierte Datenbank zur Verwaltung von Daten verwendet. Mit dem Wachstum des Datenvolumens und der Zunahme der Vernetzung der Daten untereinander wird die relationale Datenbank vor neue Herausforderungen gestellt. Eine generelle Erkenntnis aus der Evaluierung und der Auswertung der Messergebnisse hat ergeben, dass relationale Datenbanken bedeutend schlechter mit stark ausgeprägten vernetzten Daten, also Daten unterschiedlichen Typs die untereinander verbunden sind, umgehen. Relationale Datenbanken weisen einen Vorteil gegenüber Graphdatenbanken bezüglich Abfragen, welche Daten ähnlichen Typs oder Informationen nach bestimmten Restriktionen selektieren, auf. Jedoch ist die Suche nach verbundenen Daten weitaus zeitintensiver. Durch einen ausgeprägten Vernetzungsgrad der Daten leidet die relationale Datenbanken am `JOIN`-Problem. Dies kann mitunter am verwendeten Datenbankschema, welches sich nicht vollständig an die Normalform hält, liegen. Eine Vielzahl an `JOIN`-Operationen, welche für die Ermittlung verbundener Daten ausgeführt werden müssen, wirken sich negativ auf das Laufzeitverhalten bei Suchoperationen aus. Für Abfragen auf einen Datenbestand mit stark ausgeprägten Vernetzungsgrad zwischen den Daten, bzw. Daten in Form von Graphstrukturen eignen sich Graphdatenbanken besser. In diesem Bereich können sie ihre Stärken bezüglich der Strukturanalyse von Graphen ausspielen. Die Traversierung mittels zur Verfügung gestelltem Framework bzw. in Java ausführbaren Operationen auf den Graphen, sowie den proprietären Abfragesprachen ist wesentlich performanter als die Ermittlung eines Pfades mittels SQL in der relationalen Datenbank.

Die Umsetzung des feingranularen Berechtigungskonzepts erhöht die Anzahl der `JOIN`-Operationen in der relationalen Datenbank. Während die Berechtigungen in den GDBMS als Attribut umgesetzt wurden, werden die Berechtigungen in der relationalen Datenbank in einer eigenen Relation gespeichert. Dies bringt weitere `JOIN`-Operationen für die relationale Datenbank mit sich, die sich wiederum auf die Laufzeit auswirken. Generell

<table>
<tr><th></th><th>Neo4j</th><th>OrientDB</th><th>MySql</th></tr>
<tr><td rowspan="2">Vorteile</td><td>• Vielzahl an unterstützten APIs</td><td>• schnelle Laufzeit bei Datenimport unter der verwendeten Implementierung
• schnelle Strukturanalyse von Graphen</td><td rowspan="2">• kurze Importdauer bei großen Datenmengen
• geringe CPU- und E/A-Abhängigkeit bei Datenimport</td></tr>
<tr><td colspan="2">• eignen sich für Speicherung von Daten in Form von Graphstrukturen
• Analyse von Graphstrukturen</td></tr>
<tr><td>Nachteile</td><td>• skaliert schlecht unter der verwendeten Implementierung bei großen Datenmengen
• Abfragen in Java API langsamer als in Browser-Interface aufwändige Optimierung der Performance</td><td>• E/A gebunden bei Datenimport
• hoher Aufwand bei Optimierung der Performance</td><td>• Strukturanalyse von Graphen - skaliert schlechter als Graphdatenbanken bei großen Datenmengen
• E/A gebunden bei Strukturanalysen stark vernetzter Daten</td></tr>
</table>

Tabelle 6.1.: Überblick der Vor- und Nachteile der Datenbanken, anhand der Ergebnisse der Evaluierung

ist die Umsetzung des Berechtigungskonzepts zu überdenken. Berechtigungen könnten in der Graphdatenbank ebenso als Knoten und Verbindungen modelliert werden, um Zugriffsoperationen auf die Knotenattribute zu vermeiden. Jedoch hätte dies weitere Einfüge-Operationen von Entitäten zur Folge, welche sich wiederum negativ auf die Ausführungsdauer des Einfügeprozesses in der Graphdatenbank auswirken. Aus dem Umstand heraus, das Berechtigungen als eigener Knoten modelliert werden, könnte sich das Problematik ergeben, dass Knoten eine Vielzahl von Verbindungen besitzen, welche eine Traversierung erschweren und deren Laufzeit beeinträchtigt. Die Tabelle 6.1 zeigt die Vor- und Nachteile der ausgewählten Datenbanken in der umgesetzten Implementierung anhand der Ergebnisse der Evaluierung.

Ausblick Eine umfassendere Bewertung der Leistungsfähigkeit wäre durch ein breiteres Spektrum an ausgewählten Datenbankgrößen möglich. Eine

Stichprobe von fünf unterschiedlichen Datenbeständen stellt eine vergleichsweise wenig repräsentative Anzahl an unterschiedlichen Größen für die Messung der Laufzeit des Schreibvorganges dar. Größere Stichproben erlauben eine demonstrativ exaktere Einschätzung der Skalierung der Laufzeit des Schreibvorganges in Abhängigkeit von Knoten und Kantenanzahl. Die verwendeten Cachingmechanismen der Datenbanktechnologien wirken sich auf das Laufzeitverhalten im Bezug auf Leseoperationen aus, wodurch die Messergebnisse ein leicht verzerrtes Bild liefern. Erstrebenswert wäre eine größere Anzahl an Testdurchläufen sowie die Berücksichtigung des Caches der einzelnen getesteten Datenbanken. Ausführungsreihenfolgen und unterschiedliche Warm-Up Prozedere des Caches können zu divergenten Ausführungszeiten führen. Aus zeitlichen Gründen war es jedoch im Rahmen der Arbeit nicht möglich, die einzelnen Testabläufe unter Berücksichtigung eines "kalten" bzw. "warmen" Caches zur Ermittlung der Laufzeit durchzuführen. In Anbetracht der gewählten Messgrößen von CPU-Auslastung in Prozent und geschriebenen Eingabe-/Ausgabebytes pro Sekunde ist zu hinterfragen, ob geeignete Rückschlüsse auf Speicher- und Prozessorauslastung während der Ausführungszeit von Lese- und Suchoperationen gezogen werden können. Gegebenenfalls wäre es in Erwägung zu ziehen, die Messgrößen nicht während eines Durchlaufs mittels der von Microsoft zur Verfügung gestellten Leistungsüberwachung zu ermitteln, sondern CPU und verwendete Heapgröße mit Java Profilingtools zu ermitteln. Die Ermittlung mittels Profilingtools müsste jedoch in einem unabhängigen Testlauf durchgeführt werden, da sich die Analysetools negativ auf das Laufzeitverhalten von Lese- und Schreiboperationen auswirkt. Dadurch könnten jedoch konkretere Rückschlüsse auf CPU- oder speicherintensive Operationen der einzelnen Datenbanktechnologien gezogen werden.

Im Rahmen der Implementierung konnte aus zeitlichen Gründen nur ein Modell eines feingranularen Berechtigungskonzepts umgesetzt werden. Zur Bewertung der Effektivität und Effizienz des umgesetzten Modells würde sich ein konträrer Ansatz eines Berechtigungskonzepts in Graphdatenbanken anbieten. Die Umsetzung eines weiteren Berechtigungskonzepts in Graphdatenbanken, beispielsweise die Modellierung der Berechtigungen als eigener Knoten, sowie der Vergleich der Effektivität und Effizienz der einzelnen Berechtigungskonzepte bieten Ansätze für weitere Arbeiten. Ein weiterer Punkt im Bezug auf Einschränkungen der Implementierung ist die

verwendete Konfiguration der einzelnen Datenbanktechnologien. Aufgrund des hohen Recherche- und den damit verbundenen hohen Zeitaufwand für die Ermittlung der optimalen Konfiguration der Datenbanken wurden die Technologien im Allgemeinen in den Standardkonfigurationen betrieben. Einzelne Parameter wie Heap- und Stackgröße, Cache und Garbage-Collector wurden für die Optimierung der Lese- und Schreibperformance angepasst. In Anbetracht der unzähligen Konfigurationsmöglichkeiten liefert die Thematik die Möglichkeit, sich mit der Fragestellung zu befassen, welche Konfiguration in Abhängigkeit zur Graphengröße (Knoten- und Kantenanzahl) am geeignetsten erscheinen.

Appendix

Anhang A.

Testprotokolle

A.1. Messergebnisse Datenbankimport

Publikationen x 1000	Testlauf	Datenbankgröße	Startzeit	Laufzeit [s]
1	1	39 MB	Feb 16, 2016 3:04:25 PM	36,46
	2	39 MB	Feb 16, 2016 3:07:18 PM	38,40
	3	39 MB	Feb 16, 2016 3:09:30 PM	38,23
5	1	176 MB	Feb 17, 2016 9:41:06 AM	664,84
	2	176 MB	Feb 17, 2016 9:56:26 AM	682,32
	3	176 MB	Feb 17, 2016 10:12:26 AM	653,02
25	1	832 MB	Feb 18, 2016 10:52:17 PM	17.327,69
	2	832 MB	Feb 19, 2016 8:01:44 AM	17.590,01
	3	832 MB	Feb 21, 2016 4:28:46 PM	17.924,96
50	1	1,87 GB	Feb 26, 2016 5:39:07 PM	16.5802,87
	2	n/a	n/a	n/a
	3	n/a	n/a	n/a
100	1	2,67 GB	Mär 04, 2016 6:59:45 PM	692.115,8
	2	n/a	n/a	n/a
	3	n/a	n/a	n/a

Tabelle A.1.: Messergebnisse des Datenimports in Neo4j

Publikationen x 1000	Testlauf	Datenbankgröße	Startzeit	Laufzeit [s]
1	1	124 MB	Feb 16, 2016 3:12:10 PM	351,36
	2	136 MB	Feb 16, 2016 3:19:39 PM	347,26
	3	136 MB	Feb 16, 2016 3:27:44 PM	353,79
5	1	314 MB	Feb 17, 2016 8:20:39 AM	1.304,63
	2	314 MB	Feb 17, 2016 8:45:22 AM	1.334,22
	3	315 MB	Feb 17, 2016 9:11:11 AM	1.313,43
25	1	1,07 GB	Feb 17, 2016 8:54:44 PM	6.056,84
	2	1,10 GB	Feb 17, 2016 11:10:00 PM	6.085,05
	3	1,08 GB	Feb 18, 2016 9:08:51 AM	6.049,64
50	1	1,94 GB	Feb 25, 2016 4:44:23 PM	12.532,93
	2	1,94 GB	Feb 26, 2016 3:09:29 AM	13.174,16
	3	1,98 GB	Feb 26, 2016 10:56:10 AM	13.252,37
100	1	3,79 GB	Mär 02, 2016 10:10:45 PM	34.972,36
	2	3,75 GB	Mär 15, 2016 4:01:28 PM	25.884,04
	3	3,75 GB	Mär 22, 2016 12:12:29 AM	23.279,87

Tabelle A.2.: Messergebnisse des Datenimports in OrientDB

Publikationen x 1000	Testlauf	Datenbankgröße	Startzeit	Laufzeit [s]
1	1	73.6 MB	Feb 16, 2016 3:37:35 PM	87,67
	2	73.6 MB	Feb 16, 2016 3:40:44 PM	86,80
	3	73.6 MB	Feb 16, 2016 3:43:32 PM	88,94
5	1	186 MB	Feb 17, 2016 10:47:47 AM	421,15
	2	186 MB	Feb 17, 2016 11:11:30 AM	429,03
	3	186 MB	Feb 17, 2016 11:28:10 AM	418,92
25	1	557 MB	Feb 20, 2016 11:51:28 AM	2.056,47
	2	557 MB	Feb 20, 2016 2:18:23 PM	2.038,60
	3	557 MB	Feb 20, 2016 3:36:49 PM	2.052,60
50	1	946 MB	Feb 22, 2016 5:57:31 PM	4.145,87
	2	946 MB	Feb 22, 2016 10:25:49 PM	4.029,51
	3	946 MB	Feb 23, 2016 10:16:39 AM	3.994,64
100	1	1,49 GB	Mär 02, 2016 6:08:34 AM	8.599,77
	2	1,49 GB	Mär 02, 2016 8:31:57 AM	8.068,25
	3	1,49 GB	Mär 02, 2016 10:46:29 AM	9.018,95

Tabelle A.3.: Messergebnisse des Datenimports in MySQL

A.2. Suchanfragen

Nachfolgend werden die in den Tabellen A.7 und A.8 verwendeten Abkürzungen beschrieben. Sie beschreiben das eindeutige Label der einzelnen Entitäten, welche von den Suchanfragen zurückgeliefert werden.

`P`	...	PAPER	`m`	...	MEMBER_OF
`K`	...	KEYWORD	`w`	...	WRITTEN_BY
`J`	...	JOURNAL	`co`	...	CONTAINS
`F`	...	FIELD_OF_STUDY	`po`	...	PART_OF
`U`	...	PAPER_URL	`pi`	...	PUBLISHED_IN
`C`	...	CONFERENCE	`r`	...	REFERENCED_BY
`CI`	...	CONFERENCE_INSTANCE	`d`	...	DOWNLOADABLE_AT
`A`	...	AUTHOR	`pr`	...	PRESENTED_IN
`AF`	...	AFFILIATION	`i`	...	INSTANCE_OF

Anzahl Papers	Suche	Suchwert	Testlauf	Berechtigungen	Laufzeit m. B. [s]	Laufzeit o. B. [s]
1.000	A	yannick letawe	1	com, uk	8,67	10,11
			2		8,51	11,35
			3		8,64	8,64
	P	Efficiently Implementing a Large Number of LL/SC Objects	1	doi, dx, org	9,67	12,48
			2		9,58	12,31
			3		9,04	11,8
	J	BMC Evolutionary Biology	1	https, at, uk	8,68	10,5
			2		8,28	9,11
			3		8,26	9,83
5.000	A	smith	1	org, ieee, ieeexplore, discovery, journals	11,49	16,99
			2		11,59	15,67
			3		11,76	15,61
	P	security	1	edu, core, ieee, google	13,10	14,85
			2		13,18	13,04
			3		12,94	13,59
	J	research	1	uni-muenchen, nih, harvard, cambridge	11,52	14,26
			2		11,51	16,16
			3		12,28	16,12
25.000	A	lee russell	1	www	14,67	17,92
			2		14,49	17,4
			3		14,97	17,95
	P	rotation	1	http, cambridge, ebooks, org, www	22,06	21,25
			2		21,78	19,88
			3		19,63	20,37
	J	Gut	1	bmj, com, gut cabdirect	15,76	16,63
			2		13,33	18,36
			3		15,57	17,69
50.000	A	admin	1	net, org, co, fi, fr	14,97	38,07
			2		15,29	27,29
			3		16,09	25,16
	P	camera	1	edu, icm, ieee, org, adsabs	62,78	22,48
			2		37,39	23,12
			3		32,94	22,67
	J	Chemical Communications	1	infosience, au, orca, nlm, hub, nl	17,73	16,7
			2		15,04	14,29
			3		16,33	14,54
100.000	A	harris ewing	1	www, http	16,53	25,26
			2		15,71	23,22
			3		16,63	23,64
	P	blogging	1	ieee, cit, org, unizg	27,81	25,01
			2		29,42	25,09
			3		30,37	23,68
	J	Surface and Interface Analysis	1	library, edu, inist, uk	15	15,53
			2		13,14	13,65
			3		13,26	13,89

Tabelle A.4.: Laufzeiten in Sekunden der definierten Suchanfragen mit und ohne Berücksichtigung der festgelegten Berechtigungen für die Testdatenbank Neo4j

Anzahl Papers	Suche	Suchwert	Testlauf	Berechtigungen	Laufzeit m.B. [s]	Laufzeit o.B. [s]
1.000	A	yannick letawe	1	com, uk	3,02	3,26
			2		3,08	3,66
			3		3,02	3,57
	P	Efficiently Implementing a Large Number of LL/SC Objects	1	doi, dx, org	4,07	5,48
			2		4,11	5,50
			3		4,16	4,79
	J	BMC Evolutionary Biology	1	https, at, uk	2,50	2,88
			2		2,55	2,50
			3		2,47	2,46
5.000	A	smith	1	org, ieee, ieeexplore, discovery, journals	5,76	8,60
			2		5,91	6,11
			3		5,88	6,09
	P	security	1	edu, core, ieee, google	6,43	8,80
			2		6,35	7,64
			3		6,45	7,71
	J	research	1	uni-muenchen, nih, harvard,cambridge	6,02	5,29
			2		6,18	6,01
			3		6,17	5,76
25.000	A	lee russell	1	www	6,31	9,43
			2		6,04	7,82
			3		6,13	7,56
	P	rotation	1	http, cambridge, ebooks, org, www	17,05	27,97
			2		16,79	17,80
			3		16,88	18,96
	J	Gut	1	bmj, com, gut, cabdirect	6,65	6,63
			2		5,90	7,00
			3		6,49	6,63
50.000	A	admin	1	net, org, co, fi, fr	11,97	11,05
			2		13,42	12,10
			3		13,09	12,42
	P	camera	1	edu, icm, ieee, org, adsabs	40,27	28,88
			2		35,26	29,70
			3		29,44	30,92
	J	Chemical Communications	1	infosience, au, orca, nlm, hub, nl	10,43	5,29
			2		9,07	4,77
			3		10,34	4,69
100.000	A	harris ewing	1	www, http	17,07	17,08
			2		16,10	17,68
			3		16,87	17,29
	P	blogging	1	ieee, cit, org, unizg	42,13	40,71
			2		40,51	42,70
			3		42,86	40,53
	J	Surface and Interface Analysis	1	library, edu, inist, uk	6,45	4,66
			2		5,06	4,89
			3		5,28	4,80

Tabelle A.5.: Laufzeiten in Sekunden der definierten Suchanfragen mit und ohne Berücksichtigung der festgelegten Berechtigungen für die Testdatenbank OrientDB

Anzahl Papers	Suche	Suchwert	Testlauf	Berechtigungen	Laufzeit m.B. [s]	Laufzeit o.B. [s]
1.000	A	yannick letawe	1	com, uk	0,17	0,17
			2		0,15	0,2
			3		0,15	0,17
	P	Efficiently Implementing a Large Number of LL/SC Objects	1	doi, dx, org	0,21	0,46
			2		0,20	0,28
			3		0,20	0,27
	J	BMC Evolutionary Biology	1	https, at, uk	0,14	0,02
			2		0,13	0,03
			3		0,12	0,02
5.000	A	smith	1	org, ieee, ieeexplore, discovery, journals	6,68	10,54
			2		6,48	6,19
			3		6,45	6,27
	P	security	1	edu, core, ieee, google	43,58	3,02
			2		43,04	2,4
			3		43,57	2,45
	J	research	1	uni-muenchen, nih, harvard, cambridge	22,06	13,05
			2		24,38	12,56
			3		21,65	12,81
25.000	A	lee russell	1	www	5,26	7,52
			2		5,51	7,24
			3		5,45	7,32
	P	rotation	1	http, cambridge, ebooks, org, www	572,77	757,62
			2		572,96	764
			3		538,88	738,24
	J	Gut	1	bmj, com, gut, cabdirect	33,04	53,42
			2		32,96	54,7
			3		32,99	53,7
50.000	A	admin	1	net, org, co, fi, fr	75,29	40,19
			2		71,02	36,41
			3		72,86	35,76
	P	camera	1	edu, icm, ieee, org, adsabs	1145,56	847,39
			2		1472,12	836,7
			3		1501,09	857,66
	J	Chemical Communications	1	infosience, au, orca, nlm, hub, nl	482,45	31,33
			2		440,81	31,04
			3		472,9	28,56
100.000	A	harris ewing	1	www, http	82	81,99
			2		78,47	82,44
			3		81,51	81,05
	P	blogging	1	ieee, cit, org, unizg	119,67	55,95
			2		119,94	48,92
			3		120,41	48,48
	J	Surface and Interface Analysis	1	library, edu, inist, uk	54,35	10,14
			2		47,01	10,11
			3		47,04	10,08

Tabelle A.6.: Laufzeiten in Sekunden der definierten Suchanfragen mit und ohne Berücksichtigung der festgelegten Berechtigungen für die Testdatenbank MySQL

Anzahl Papers	Value	Datensatz		Entiäten																
		∑ Knoten	∑ Kanten	P	K	J	F	U	C	CI	A	AF	m	w	co	po	pi	r	d	pr	i
1.000	Autor	24	25	1	3	1	3	11	0	0	3	2	4	3	3	3	1	0	11	0	0
	Paper	10	9	1	1	0	1	1	1	1	2	2	2	2	1	1	0	0	1	1	1
	Journal	35	34	1	10	1	9	8	0	0	3	3	3	3	10	9	1	0	8	0	0
5.000	Autor	1514	1901	41	59	25	54	593	1	1	715	25	445	716	61	56	27	593	1	1	1
	Paper	213	125	89	21	1	20	53	1	1	23	4	4	23	21	21	1	0	53	1	1
	Journal	3199	3289	174	346	104	328	1318	0	0	776	153	223	776	379	419	174	0	1318	0	0
25.000	Autor	37	36	10	0	0	0	26	0	0	1	0	0	10	0	0	0	0	26	0	0
	Paper	933	692	262	180	17	176	186	1	1	81	29	41	81	186	182	17	0	183	1	1
	Journal	1055	1119	56	105	1	93	498	0	0	273	29	42	273	130	120	56	0	498	0	0
50.000	Autor	307	297	58	27	0	26	179	0	0	15	2	2	59	29	28	0	0	179	0	0
	Paper	955	789	228	138	14	129	287	1	1	129	28	48	130	157	151	14	0	287	1	1
	Journal	3558	4036	272	66	8	64	1760	0	0	1094	294	711	1102	94	93	272	4	1760	0	0
100.000	Autor	213	212	35	0	0	0	177	0	0	1	0	0	35	0	0	0	0	177	0	0
	Paper	162	134	30	20	5	20	59	2	2	18	6	8	18	20	20	5	0	59	2	2
	Journal	260	284	16	32	1	32	88	0	0	59	32	53	59	34	34	16	0	88	0	0

Tabelle A.7.: Anzahl der zurückgelieferten Entitäten der definierten Suchanfragen, ohne Berücksichtigung der Berechtigungen, aufgeschlüsselt nach ihrem Knoten- und Kantentyp. Die Abkürzungen der unterschiedlichen Typen, sowie die definierten Suchanfragen können dem Anhang entnommen werden

Anzahl Publikationen	Value	Datensatz		Entitäten																
		∑ Knoten	∑ Kanten	P	K	J	F	U	C	CI	A	AF	m	w	co	po	pi	r	d	pr	i
1.000	Autor	24	25	1	3	1	3	11	0	0	3	2	4	3	3	3	1	0	11	0	0
	Paper	8	7	1	1	0	1	1	0	0	2	2	2	2	1	1	0	0	1	0	0
	Journal	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5.000	Autor	571	566	27	51	18	46	281	1	1	125	21	37	125	53	48	20	0	281	1	1
	Paper	68	64	5	11	0	10	33	0	0	9	0	0	9	11	11	0	0	33	0	0
	Journal	1234	1268	53	162	28	152	505	0	0	272	62	107	272	169	162	53	0	505	0	0
25.000	Autor	37	36	10	0	0	0	26	0	0	1	0	0	10	0	0	0	0	26	0	0
	Paper	696	692	28	180	17	176	183	1	1	81	29	41	81	186	182	17	0	183	1	1
	Journal	1050	1114	55	105	1	93	496	0	0	271	29	42	271	130	120	55	0	496	0	0
50.000	Autor	103	99	27	10	0	10	47	0	0	7	2	2	28	11	11	0	0	47	0	0
	Paper	603	637	20	123	9	114	232	1	1	80	23	38	80	141	135	9	0	232	1	1
	Journal	283	328	16	4	2	4	140	0	0	82	35	80	82	5	5	16	0	140	0	0
100.000	Autor	213	212	35	0	0	0	177	0	0	1	0	0	35	0	0	0	0	177	0	0
	Paper	97	95	4	16	1	16	44	0	0	11	5	7	11	16	16	1	0	44	0	0
	Journal	77	82	3	15	1	15	22	0	0	14	7	11	14	16	16	3	0	22	0	0

Tabelle A.8.: Anzahl der zurückgelieferten Entitäten der definierten Suchanfragen unter der Berücksichtigung von Berechtigungen, aufgeschlüsselt nach ihrem Knoten- und Kantentyp. Die Abkürzungen der unterschiedlichen Typen sowie die definierten Suchanfragen können dem Anhang entnommen werden

Suche	Suchwert	Testlauf	Laufzeit [s]		
			Neo4j	OrientDB	MySQL
Autor	yannick letawe	1	10,00	3,30	0,90
		2	10,29	3,31	0,84
		3	9,86	3,58	0,88
Publikation	Efficiently Implementing a Large Number of LL/SC Objects	1	10,87	4,15	0,65
		2	10,72	4,03	0,61
		3	10,17	3,95	0,62
Journal	BMC Evolutionary Biology	1	8,88	2,55	0,71
		2	8,51	2,59	0,71
		3	8,84	2,51	0,72

Tabelle A.9.: Laufzeit in Sekunden der für 1.000 Publikationen definierten Suchanfragen, ohne Berücksichtigung von Berechtigungen - Administratorrechte

Suche	Suchwert	Testlauf	Laufzeit [s]		
			Neo4j	OrientDB	MySQL
Autor	lee russell	1	13,18	8,48	5,84
		2	13,24	7,02	5,34
		3	13,11	7,36	5,85
Publikation	rotation	1	29,28	18,89	560,97
		2	19,91	15,76	547,90
		3	17,40	15,15	574,10
Journal	Gut	1	14,59	6,34	52,61
		2	13,23	5,58	53,02
		3	13,27	5,59	51,07

Tabelle A.10.: Laufzeit in Sekunden der für 25.000 Publikationen definierten Suchanfragen ohne Berücksichtigung von Berechtigungen - Administratorrechte

Anzahl Publikationen	Value	Datensatz		Entitäten															
		∑ Knoten	∑ Kanten	P	K	J	F	U	C	CI	A	AF	m	w	co	po	pi	r	d	pr	i
1.000	Autor	89	90	66	3	1	3	11	0	0	3	2	4	3	3	3	1	65	11	0	0
	Paper	43	44	35	1	0	1	1	1	2	2	2	2	1	1	0	34	1	1	1	1
	Journal	91	90	57	10	1	9	8	0	0	3	3	3	3	10	9	1	56	8	0	0
25.000	Autor	37	36	10	0	0	0	26	0	0	1	0	10	0	0	0	0	0	26	0	0
	Paper	1209	1011	541	180	17	176	183	1	1	81	29	41	81	186	182	17	319	183	1	1
	Journal	2034	2099	1035	105	1	93	498	0	0	273	29	42	273	130	120	56	980	498	0	0

Tabelle A.11.: Anzahl der zurückgelieferten Entitäten der definierten Suchanfragen für die Suche ohne Berechtigungen (Administratorrechte) aufgeschlüsselt nach ihrem Knoten- und Kantentyp. Die Abkürzungen der unterschiedlichen Typen sowie die definierten Suchanfragen können dem Anhang entnommen werden

Anhang B.

Grafische der Messergebnisse

B.1. Auswertung der Suchanfragen

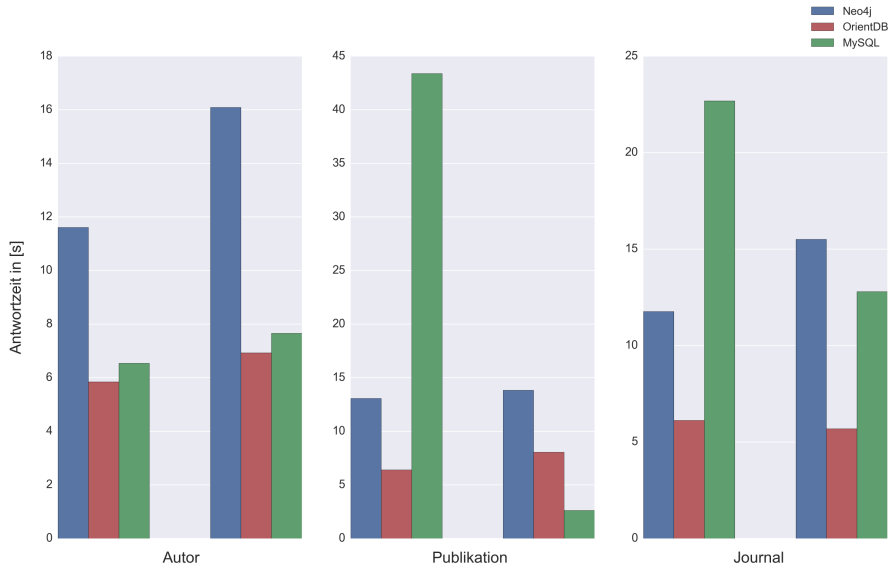


Abbildung B.1.: Antwortszeit der für einen Datenbestand von 5.000 Publikationen definierten Suchanfragen der jeweiligen Technologien (links: ohne Berechtigungen rechts: mit Berechtigungen)

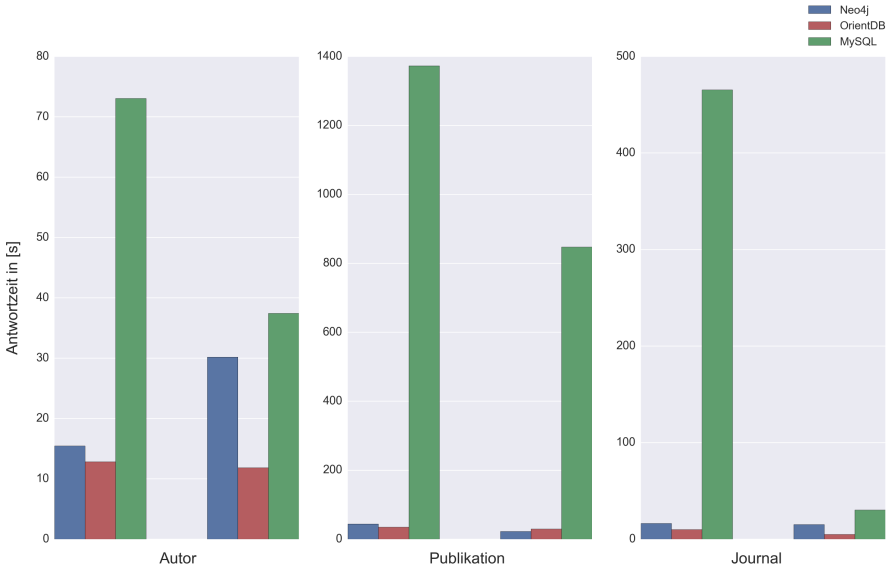


Abbildung B.2.: Antwortszeit der für einen Datenbestand von 50.000 Publikationen definierten Suchanfragen der jeweiligen Technologien (links: ohne Berechtigungen rechts: mit Berechtigungen)

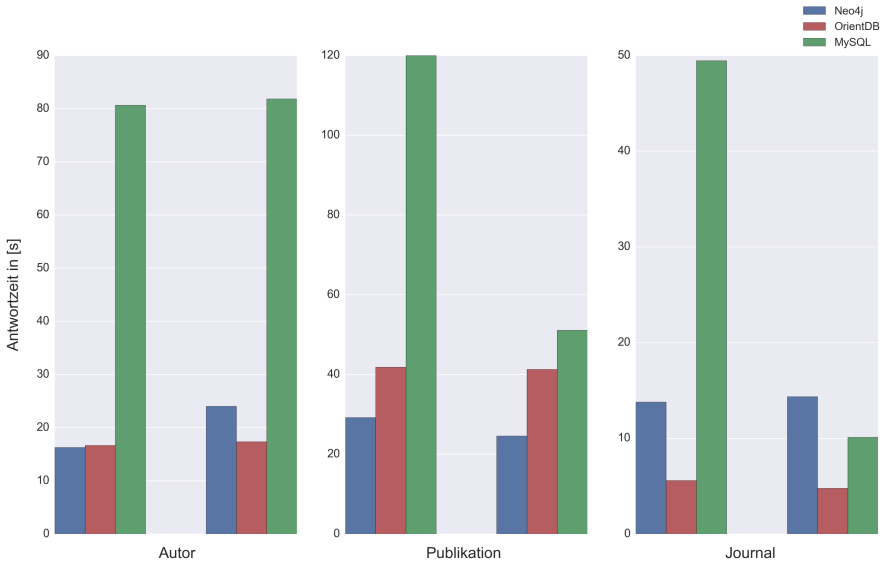


Abbildung B.3.: Antwortszeit der für einen Datenbestand von 100.000 Publikationen definierten Suchanfragen der jeweiligen Technologien (links: ohne Berechtigungen rechts: mit Berechtigungen)

B.2. Prozessorzeit der Importvorgänge auf der Standardkonfiguration vm_hdd

Die Auswertung der Prozessorauslastung für den Datenimport von 50.000 und 100.000 Publikationen wurde auf Grund der Laufzeit von größer 48 Stunden und der damit verbundenen Unübersichtlichkeit der grafischen Darstellung, dem Anhang nicht angefügt.

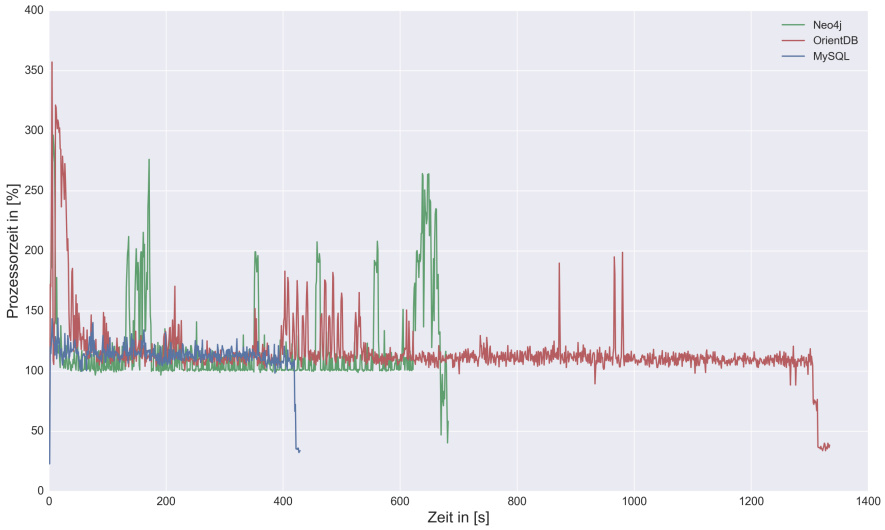


Abbildung B.4.: CPU Auslastung während des Importvorganges von 5.000 Publikationen

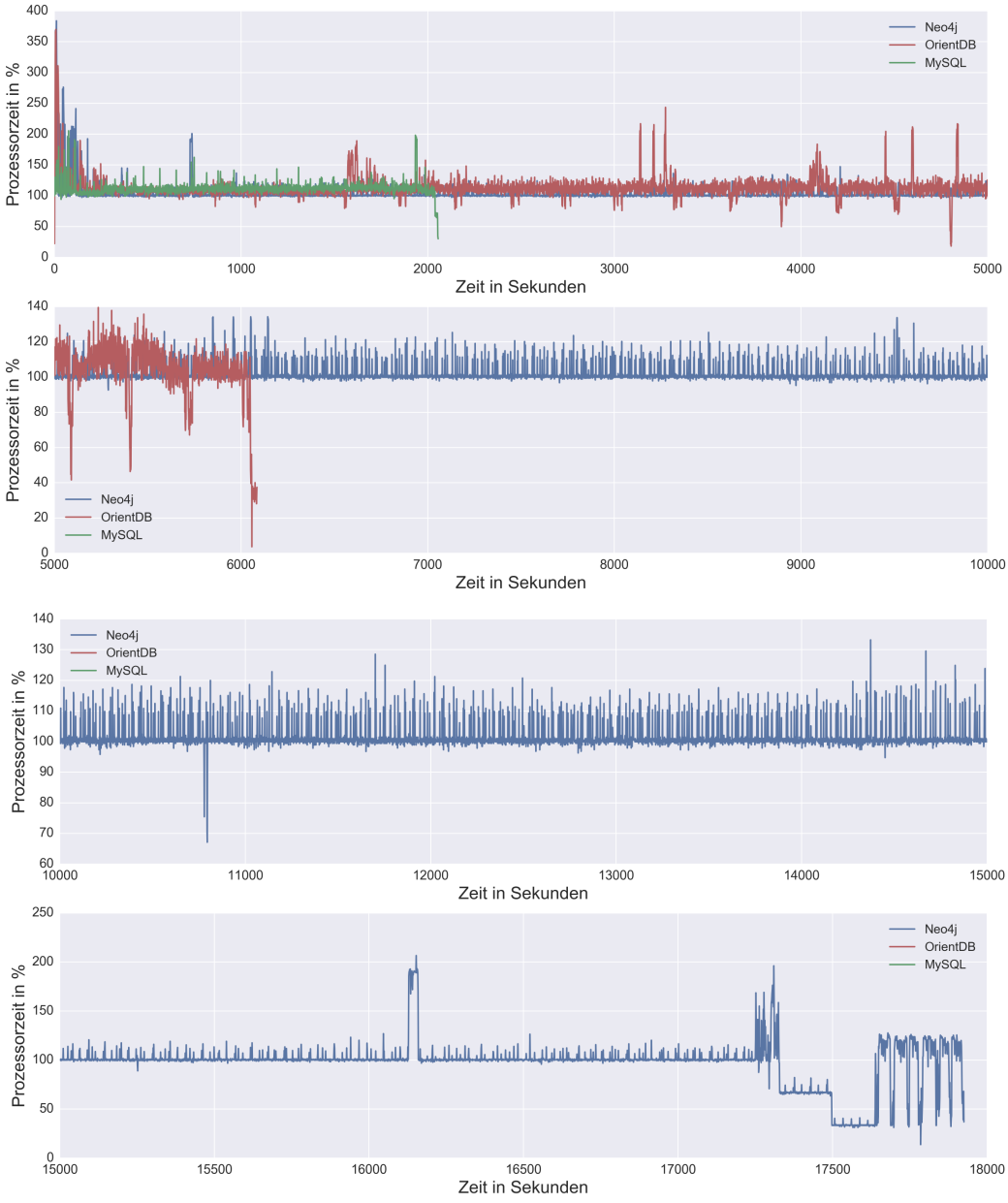


Abbildung B.5.: CPU Auslastung während des Importvorganges von 25.000 Publikationen

B.3. Geschriebene E/A-Bytes der Importvorgänge auf der Standardkonfiguration vm_hdd

Die Auswertung der geschriebenen Eingabe- Ausgabebytes für den Datenimport von 50.000 und 100.000 Publikationen wurde, auf Grund der Laufzeit von größer 48 Stunden und der damit verbundenen Unübersichtlichkeit der grafischen Darstellung, dem Anhang nicht angefügt.

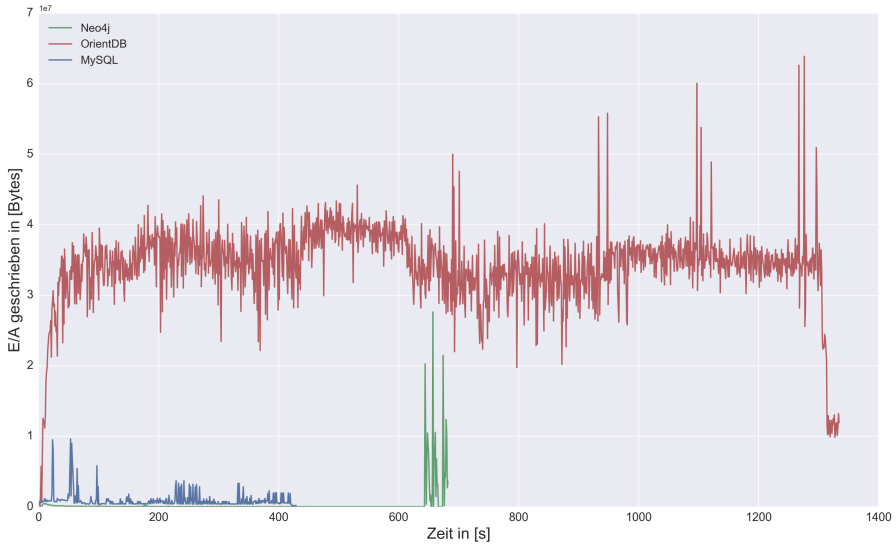


Abbildung B.6.: Anzahl der geschriebenen E/A-Bytes während des Importvorganges von 5.000 Publikationen

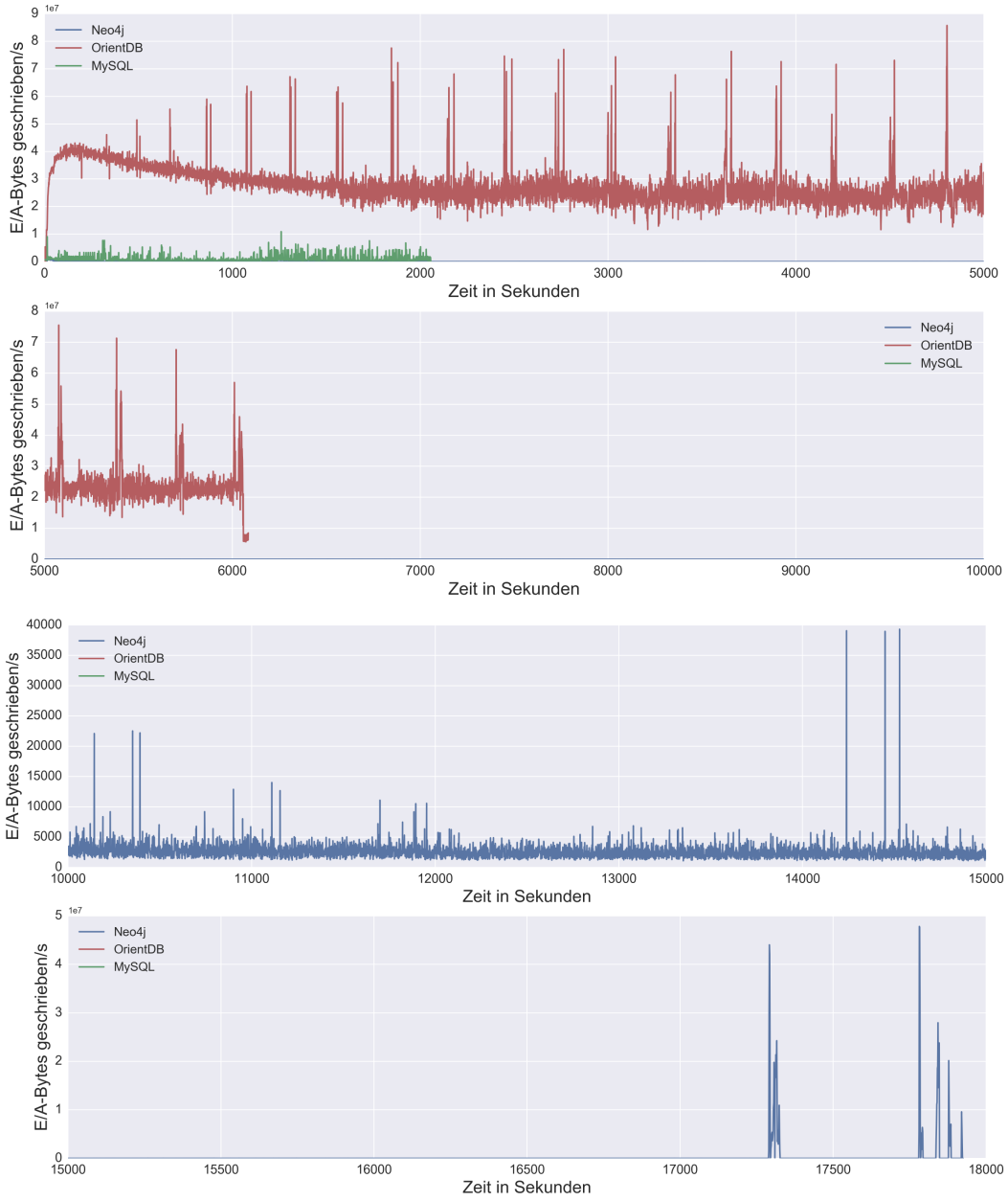


Abbildung B.7.: Anzahl der geschriebenen E/A-Bytes während des Importvorganges von 25.000 Publikationen

B.4. Prozessorzeit der Suchanfragen auf der Standardkonfiguration vm_hdd

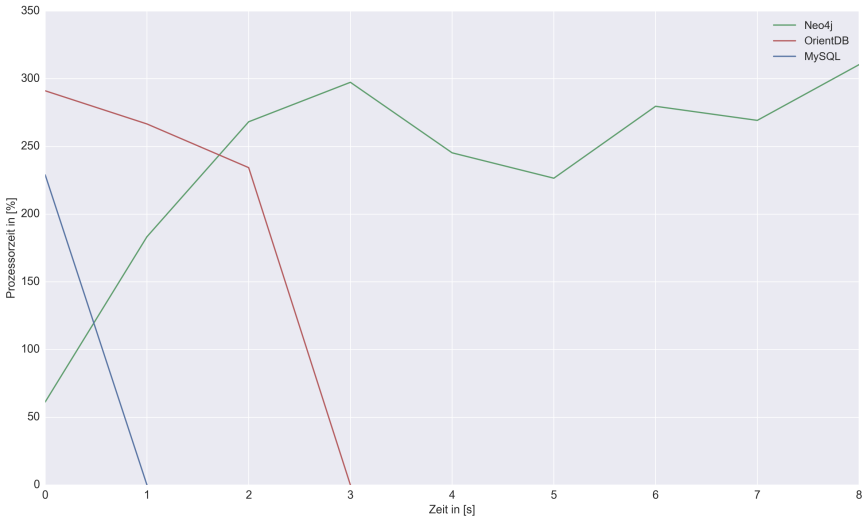


Abbildung B.8.: Prozessorzeit während der Suchanfrage des Autors yannick letawe bei einem Datenbestand von 1.000 Publikationen

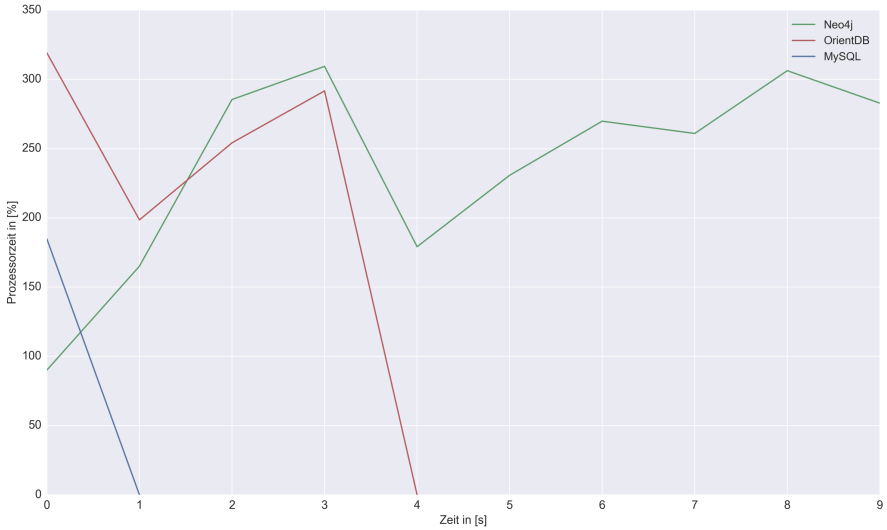


Abbildung B.9.: Prozessorzeit während der Suchanfrage der Publikation `Efficiently Implementing a Large Number of LL/SC Objects` bei einem Datenbestand von 1.000 Publikationen

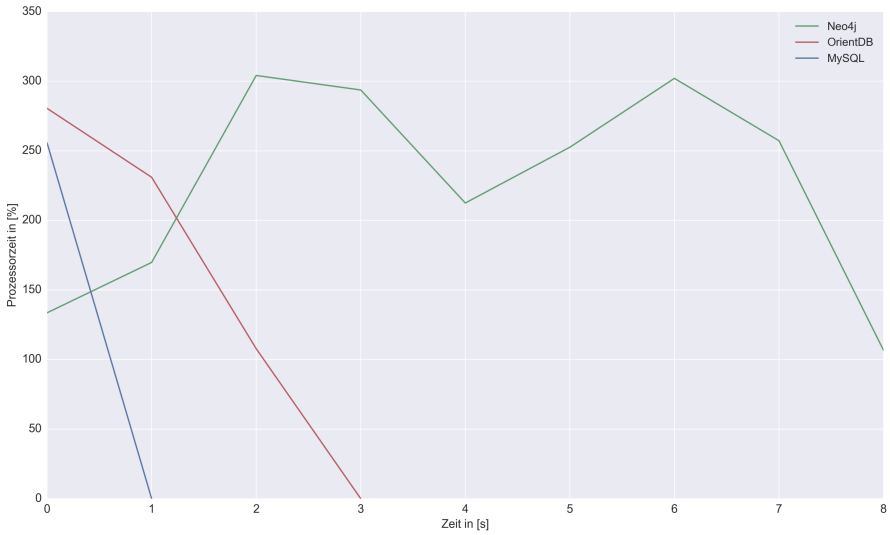


Abbildung B.10.: Prozessorzeit während der Suchanfrage des Journals `BMC Evolutionary Biology` bei einem Datenbestand von 1.000 Publikationen

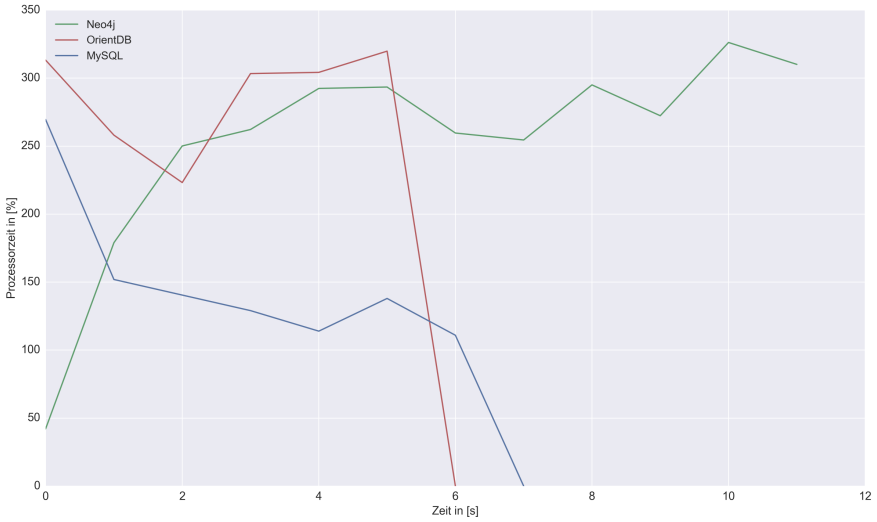


Abbildung B.11.: Prozessorzeit während der Suchanfrage des Autors `smith` bei einem Datenbestand von 5.000 Publikationen

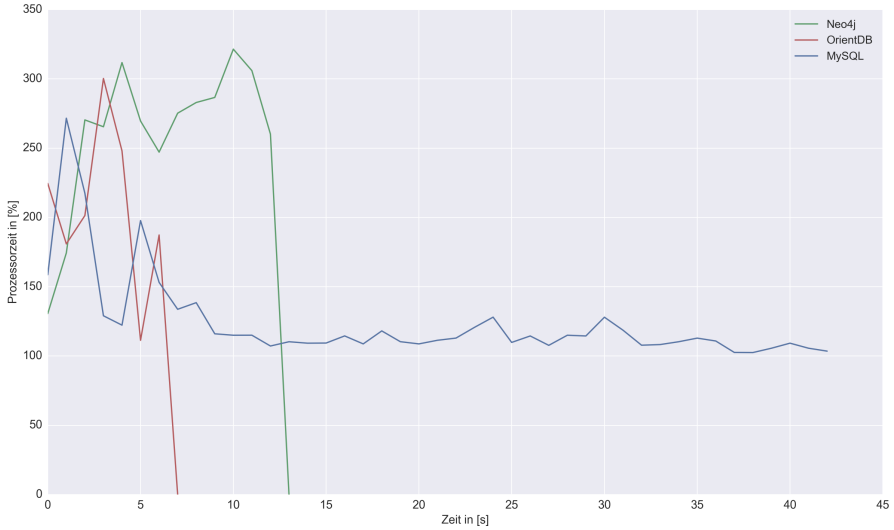


Abbildung B.12.: Prozessorzeit während der Suchanfrage der Publikation `security` bei einem Datenbestand von 5.000 Publikationen

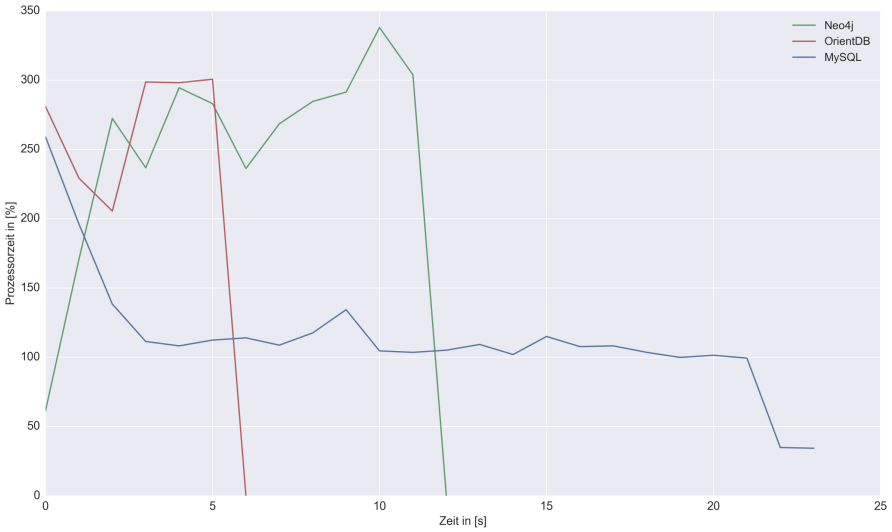


Abbildung B.13.: Prozessorzeit während der Suchanfrage des Journals `research` bei einem Datenbestand von 5.000 Publikationen

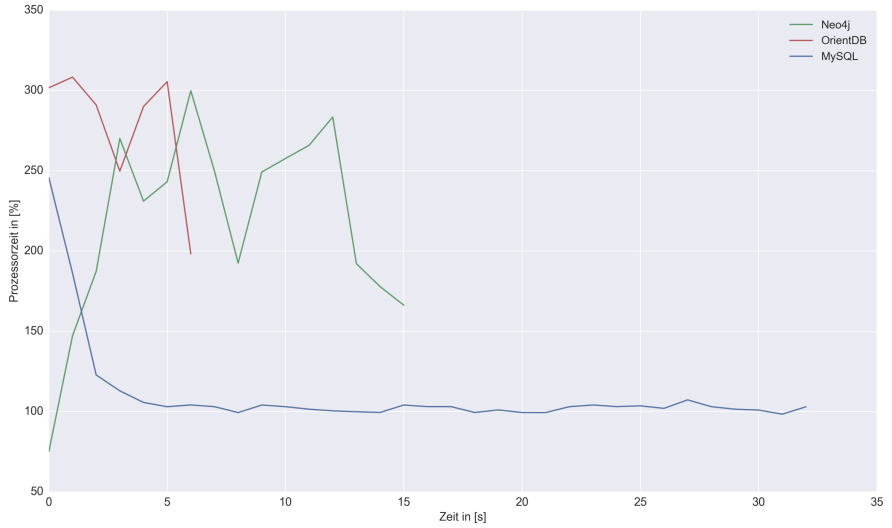


Abbildung B.14.: Prozessorzeit während der Suchanfrage des Journals `Gut` bei einem Datenbestand von 25.000 Publikationen

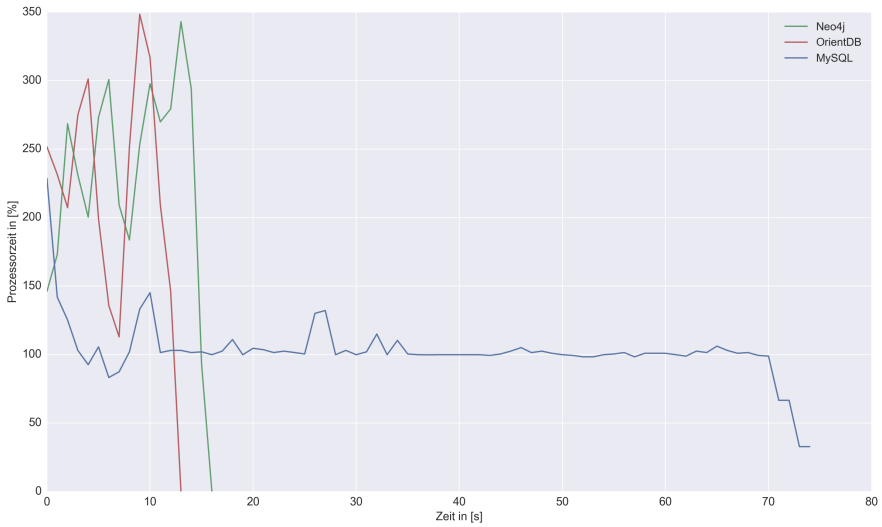


Abbildung B.15.: Prozessorzeit während der Suchanfrage des Autors `admin` bei einem Datenbestand von 50.000 Publikationen

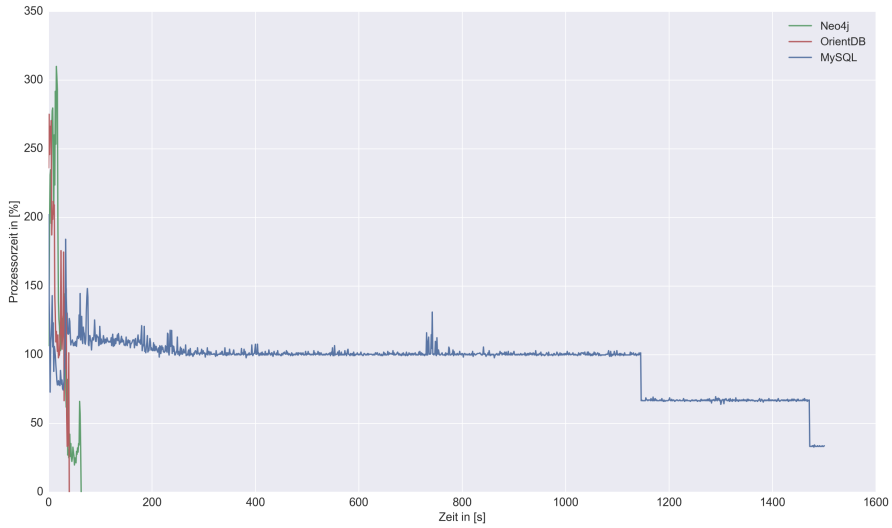


Abbildung B.16.: Prozessorzeit während der Suchanfrage der Publikation `camera` bei einem Datenbestand von 50.000 Publikationen

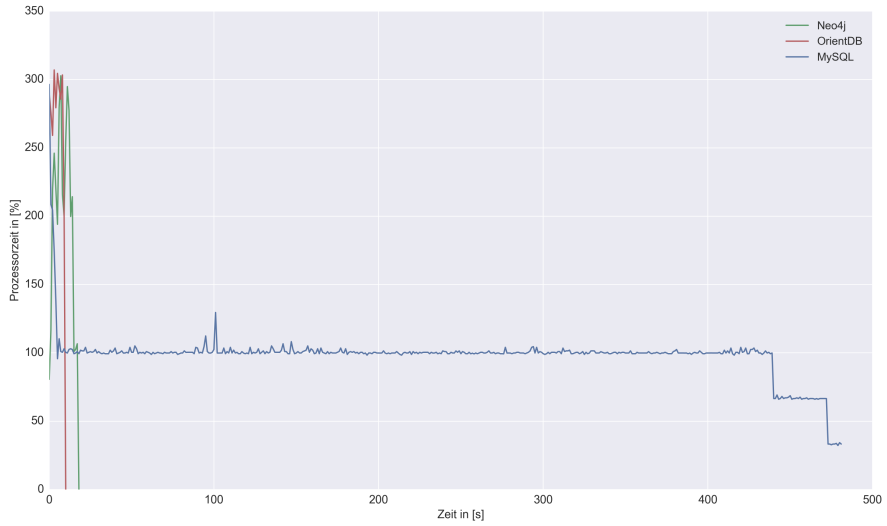


Abbildung B.17.: Prozessorzeit während der Suchanfrage des Journals `Chemical Communications` bei einem Datenbestand von 50.000 Publikationen

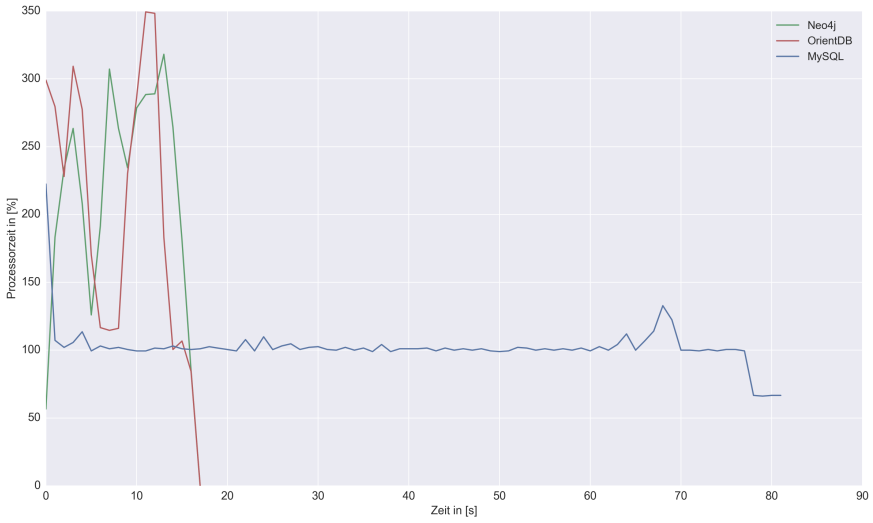


Abbildung B.18.: Prozessorzeit während der Suchanfrage des Autors `harris ewing` bei einem Datenbestand von 100.000 Publikationen

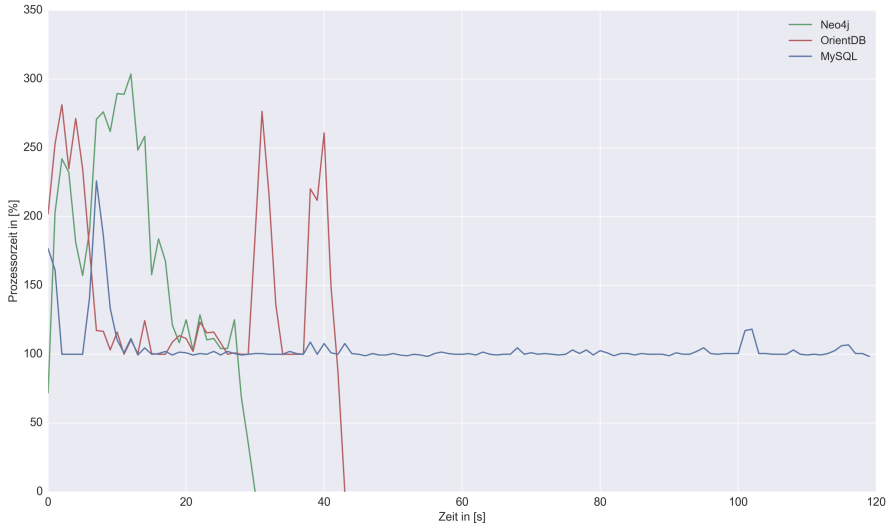


Abbildung B.19.: Prozessorzeit während der Suchanfrage der Publikation `blogging` bei einem Datenbestand von 100.000 Publikationen

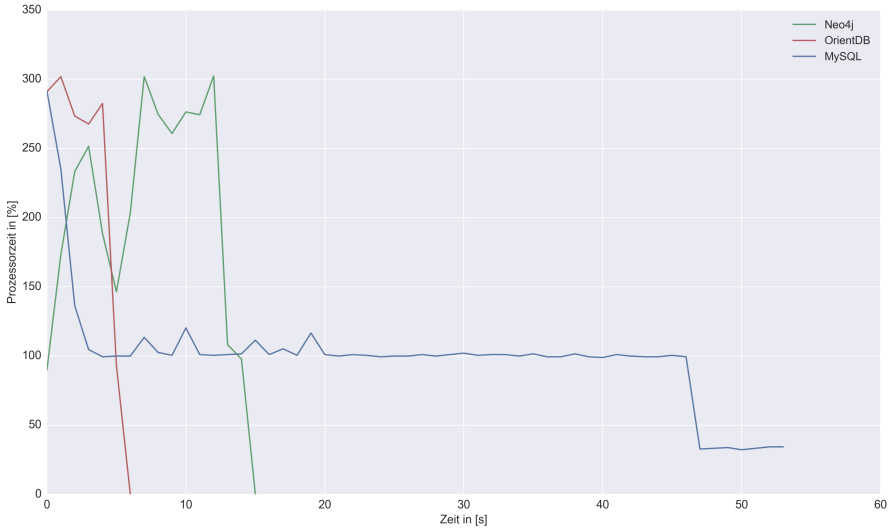


Abbildung B.20.: Prozessorzeit während der Suchanfrage des Journals `Surface and Interface Analysis` bei einem Datenbestand von 100.000 Publikationen

B.5. Geschriebene E/A-Bytes der Suchanfragen auf der Standardkonfiguration `vm_hdd`

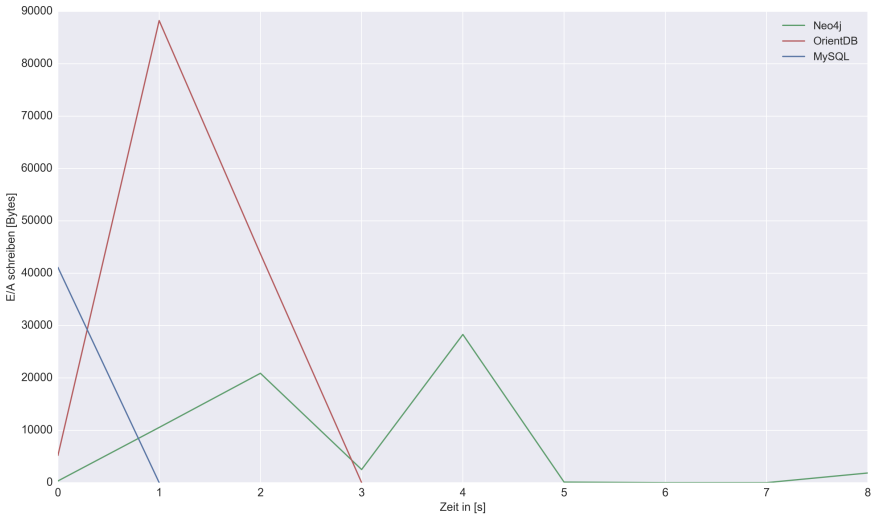


Abbildung B.21.: Geschriebene E/A-Bytes während der Suchanfrage des Autors `yannick letawe` bei einem Datenbestand von 1.000 Publikationen

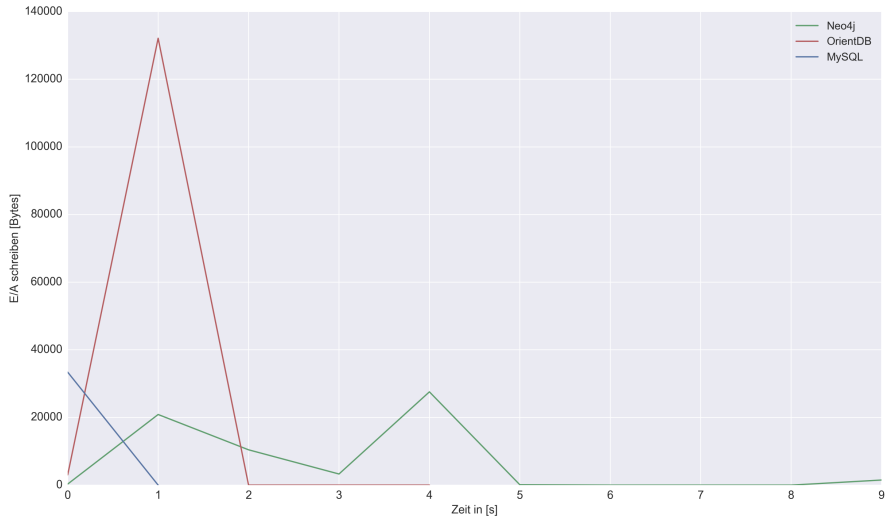


Abbildung B.22.: Geschriebene E/A-Bytes während der Suchanfrage der Publikation `Efficiently Implementing a Large Number of LL/SC Objects` bei einem Datenbestand von 1.000 Publikationen

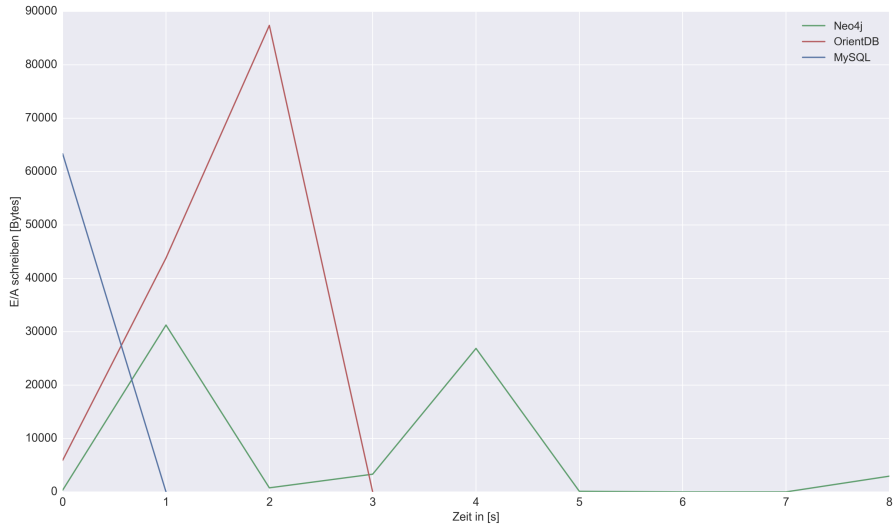


Abbildung B.23.: Geschriebene E/A-Bytes während der Suchanfrage des Journals `BMC Evolutionary Biology` bei einem Datenbestand von 1.000 Publikationen

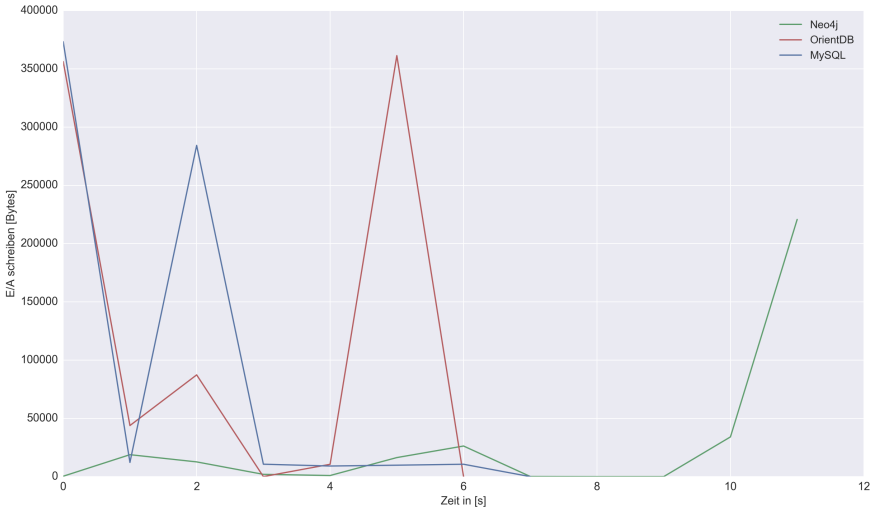


Abbildung B.24.: Geschriebene E/A-Bytes während der Suchanfrage des Autors `smith` bei einem Datenbestand von 5.000 Publikationen.

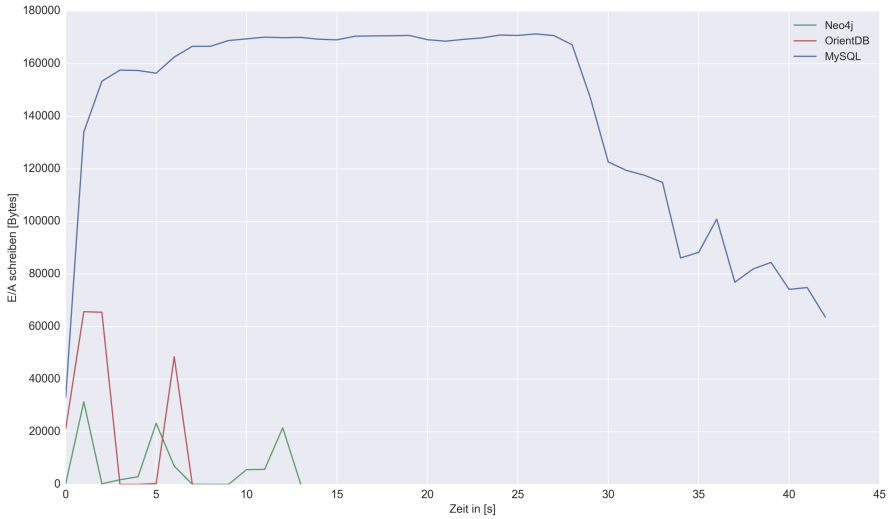


Abbildung B.25.: Geschriebene E/A-Bytes während der Suchanfrage der Publikation `security` bei einem Datenbestand von 5.000 Publikationen

B.5. Geschriebene E/A-Bytes der Suchanfragen auf der Standardkonfiguration `vm_hdd`

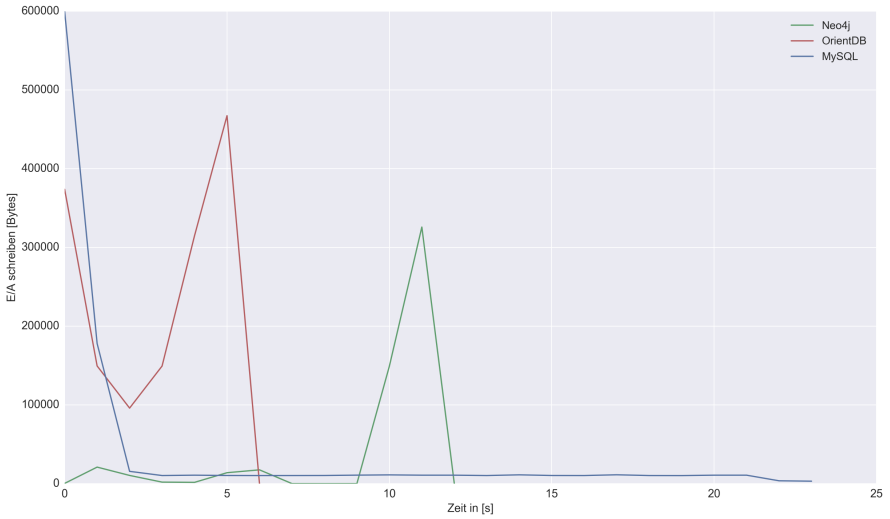


Abbildung B.26.: Geschriebene E/A-Bytes während der Suchanfrage des Journals `research` bei einem Datenbestand von 5.000 Publikationen

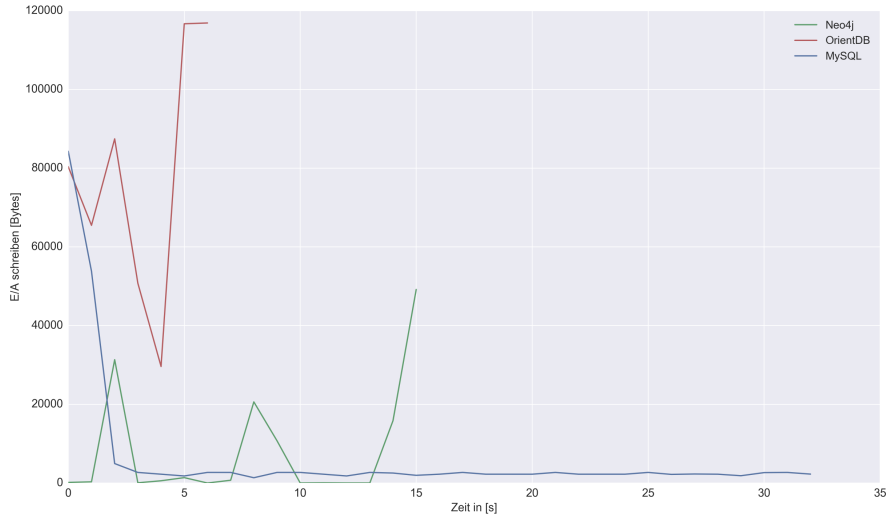


Abbildung B.27.: Geschriebene E/A-Bytes während der Suchanfrage des Journals `Gut` bei einem Datenbestand von 25.000 Publikationen

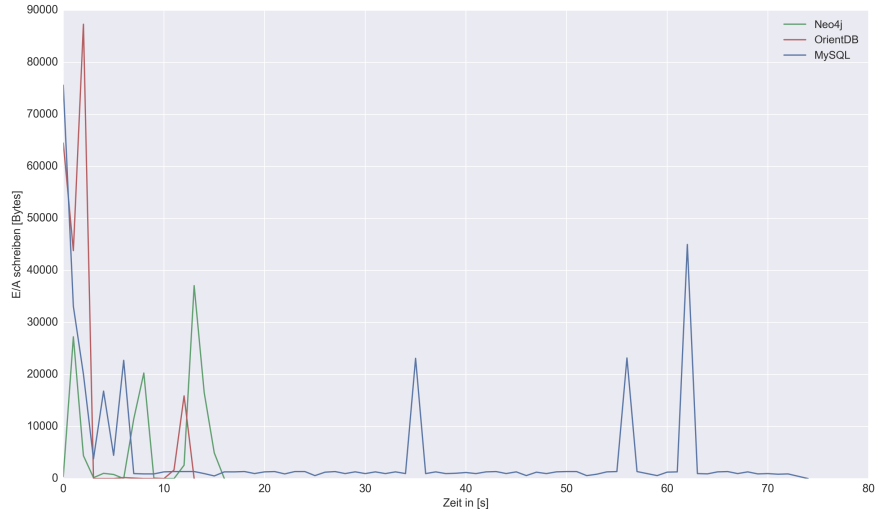


Abbildung B.28.: Geschriebene E/A-Bytes während der Suchanfrage des Autors `admin` bei einem Datenbestand von 50.000 Publikationen

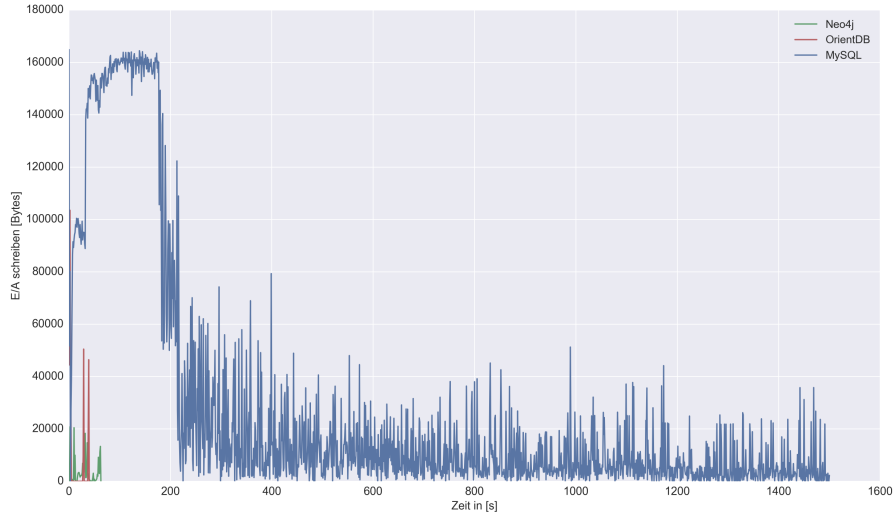


Abbildung B.29.: Geschriebene E/A-Bytes während der Suchanfrage der Publikation `camera` bei einem Datenbestand von 50.000 Publikationen.

B.5. Geschriebene E/A-Bytes der Suchanfragen auf der Standardkonfiguration `vm_hdd`

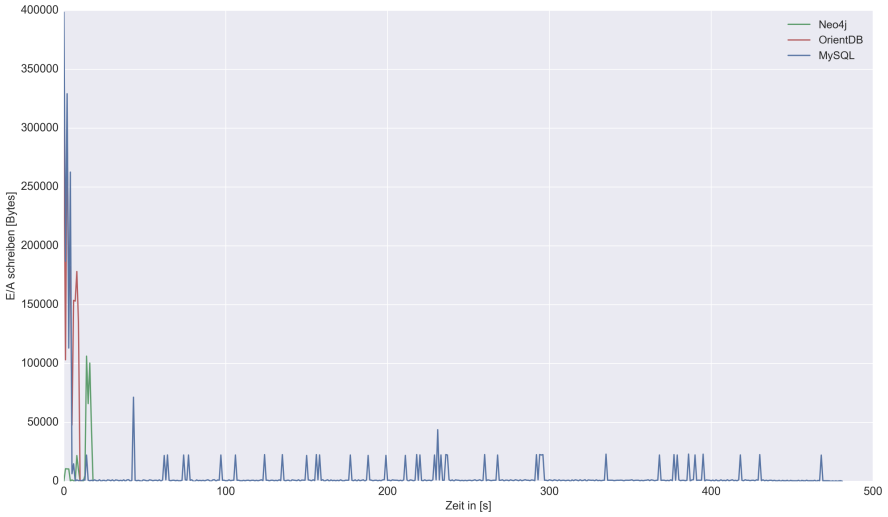


Abbildung B.30.: Geschriebene E/A-Bytes während der Suchanfrage des Journals `Chemical Communications` bei einem Datenbestand von 50.000 Publikationen

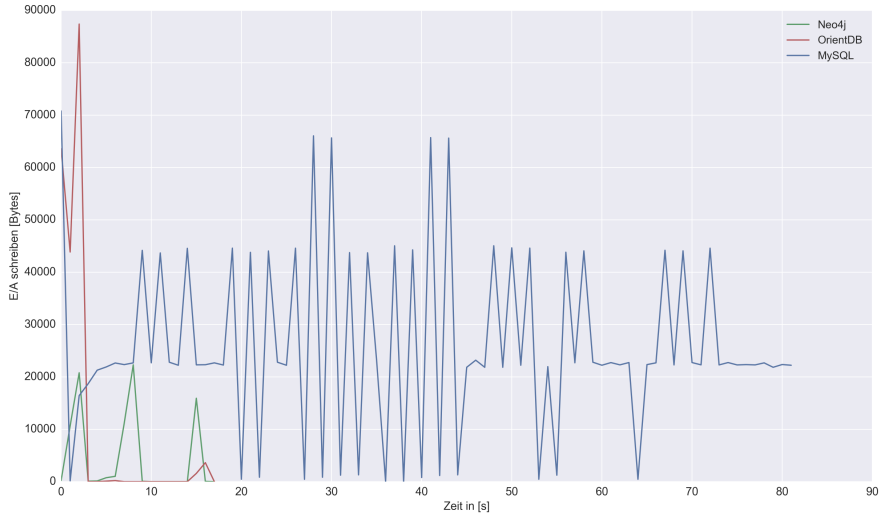


Abbildung B.31.: Geschriebene E/A-Bytes während der Suchanfrage des Autors `harris ewing` bei einem Datenbestand von 100.000 Publikationen

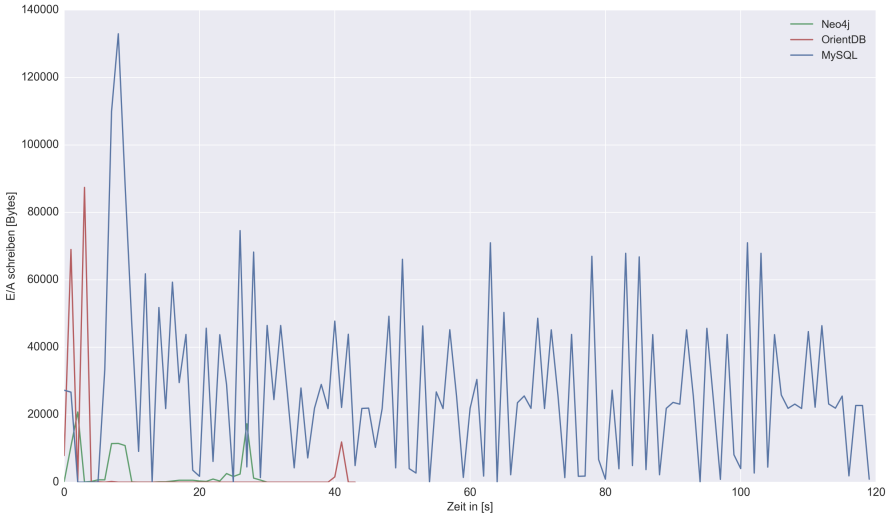


Abbildung B.32.: Geschriebene E/A-Bytes während der Suchanfrage der Publikation `blogging` bei einem Datenbestand von 100.000 Publikationen

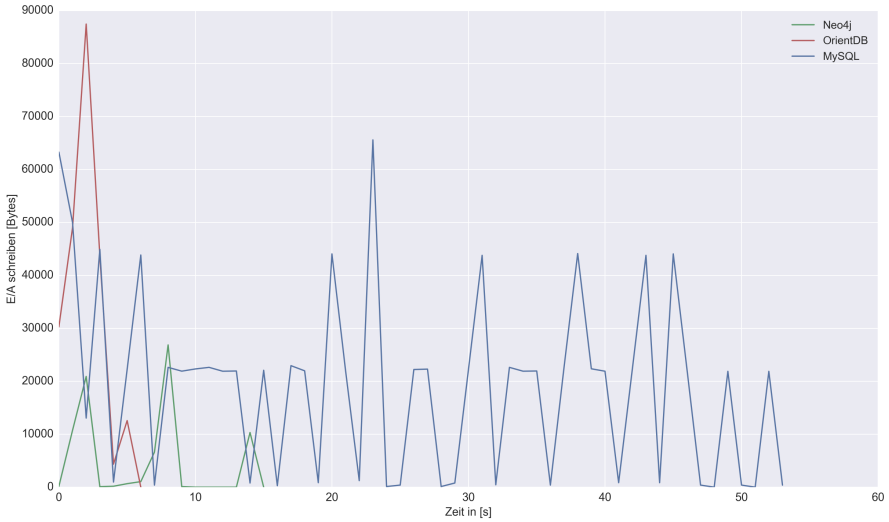


Abbildung B.33.: Geschriebene E/A-Bytes während der Suchanfrage des Journals `Surface and Interface Analysis` bei einem Datenbestand von 100.000 Publikationen

B.6. Prozessorzeit der Technologien auf den unterschiedlichen Hardwarekonfigurationen

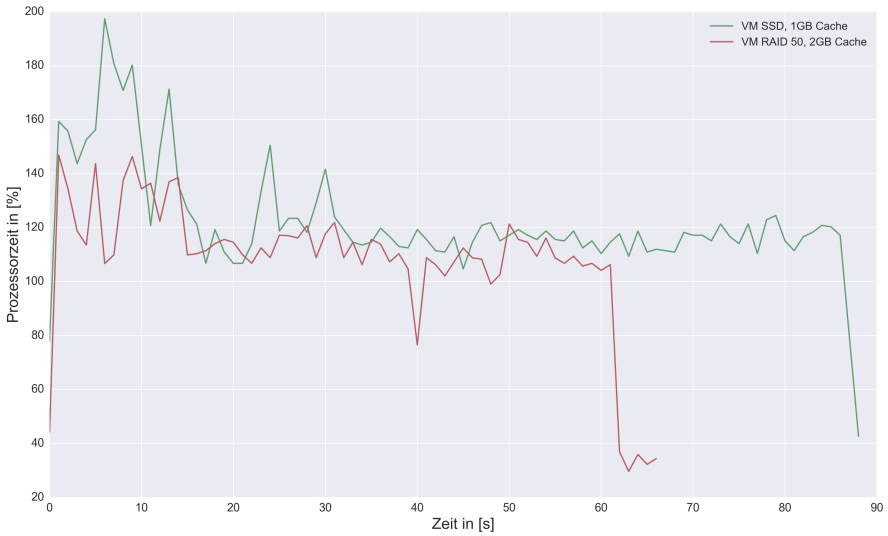


Abbildung B.34.: Direkter Vergleich der Prozessorauslastung, der unterschiedlichen Hardwarekonfigurationen (`vm_ssd` und `vm_hdd`) während des Importvorganges von 1.000 Publikationen in der relationalen Datenbank MySQL

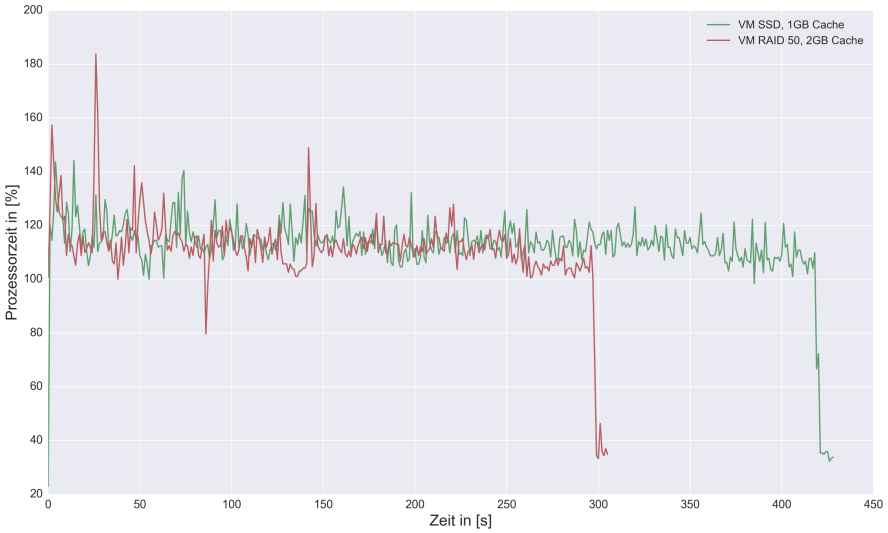


Abbildung B.35.: Direkter Vergleich der Prozessorauslastung, der unterschiedlichen Hardwarekonfigurationen (`vm_ssd` und `vm_hdd`) während des Importvorganges von 5.000 Publikationen in der relationalen Datenbank MySQL

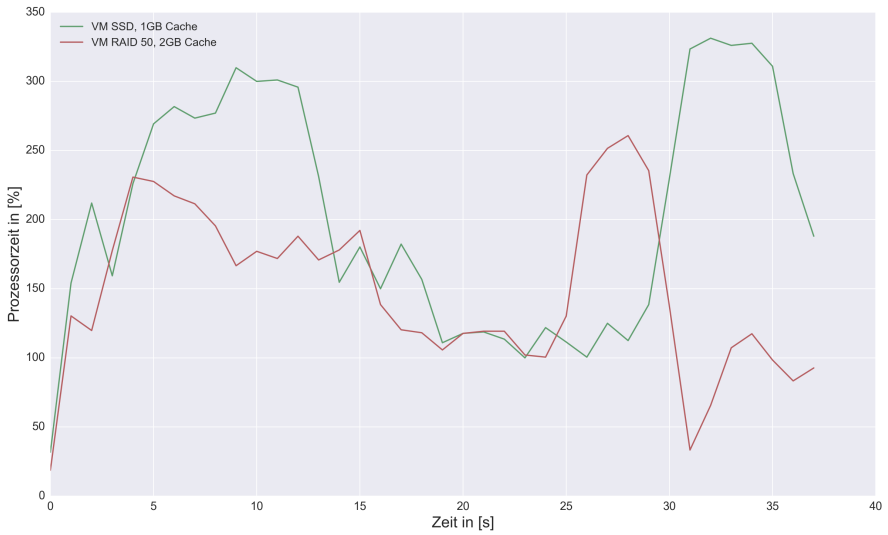


Abbildung B.36.: Direkter Vergleich der Prozessorauslastung, der unterschiedlichen Hardwarekonfigurationen (`vm_ssd` und `vm_hdd`) während des Importvorganges von 1.000 Publikationen in der Datenbank Neo4j

Abbildung B.37.: Direkter Vergleich der Prozessorauslastung, der unterschiedlichen Hardwarekonfigurationen (`vm_ssd` und `vm_hdd`) während des Importvorganges von 5.000 Publikationen in der relationalen Datenbank Neo4j

Abbildung B.38.: Direkter Vergleich der Prozessorauslastung, der unterschiedlichen Hardwarekonfigurationen (`vm_ssd` und `vm_hdd`) während des Importvorganges von 1.000 Publikationen in der Datenbank OrientDB

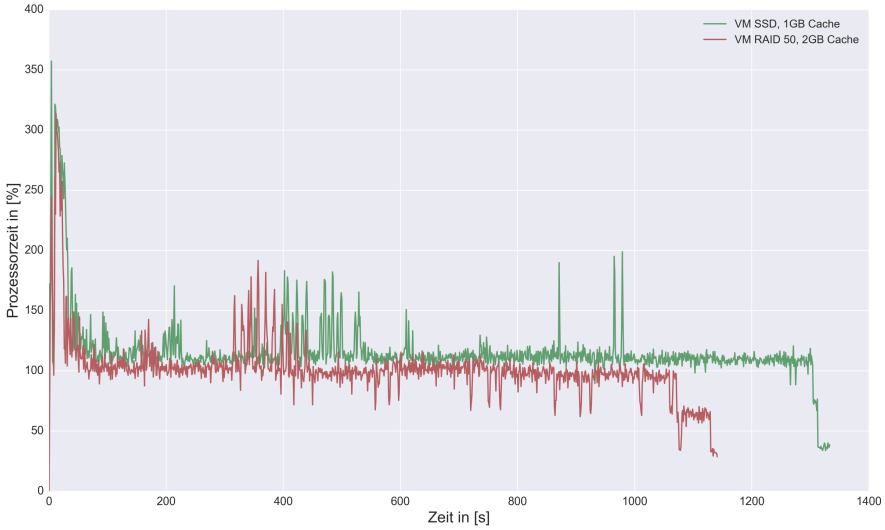


Abbildung B.39.: Direkter Vergleich der Prozessorauslastung, der unterschiedlichen Hardwarekonfigurationen (`vm_ssd` und `vm_hdd`) während des Importvorganges von 5.000 Publikationen in der relationalen Datenbank OrientDB

B.7. Geschriebene E/A-Bytes der Technologien auf den unterschiedlichen Hardwarekonfigurationen

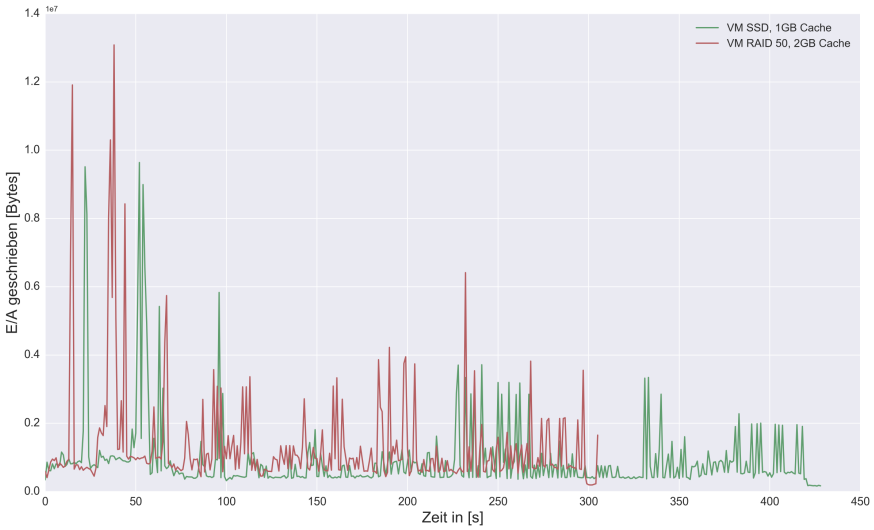


Abbildung B.40.: Direkter Vergleich der Schreibrate, der unterschiedlichen Hardwarekonfigurationen (`vm_ssd` und `vm_hdd`) während des Importvorganges von 5.000 Publikationen in der relationalen Datenbank MySQL

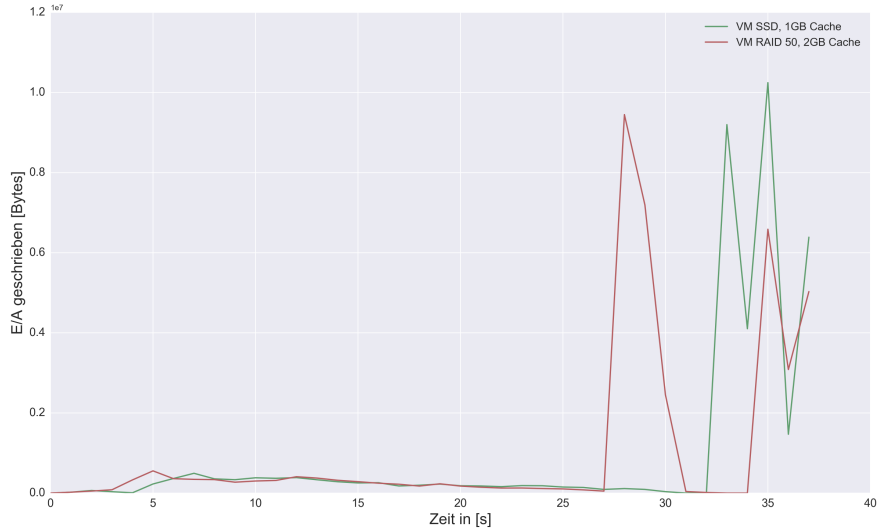


Abbildung B.41.: Direkter Vergleich der Schreibrate, der unterschiedlichen Hardwarekonfigurationen (`vm_ssd` und `vm_hdd`) während des Importvorganges von 1.000 Publikationen in Neo4j

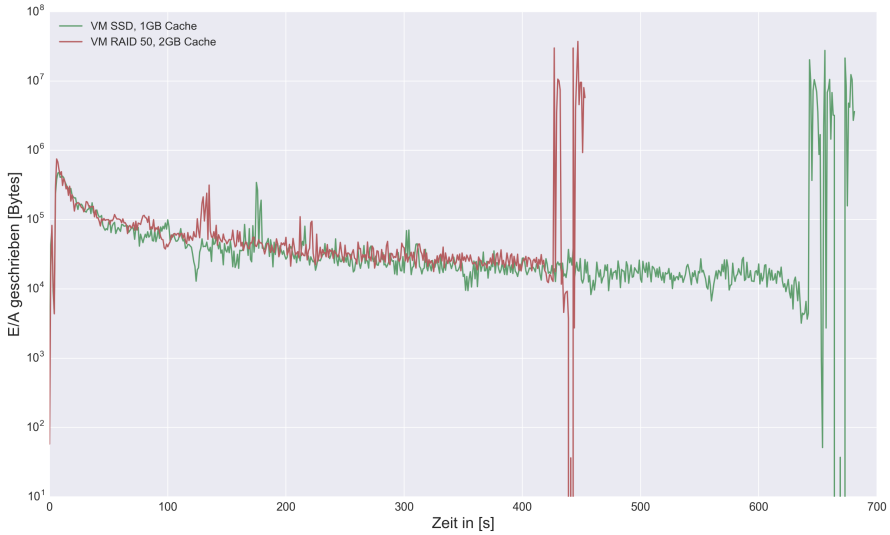


Abbildung B.42.: Direkter Vergleich der Schreibrate, der unterschiedlichen Hardwarekonfigurationen (`vm_ssd` und `vm_hdd`) während des Importvorganges von 5.000 Publikationen in Neo4j

Abbildung B.43.: Direkter Vergleich der Schreibrate, der unterschiedlichen Hardwarekonfigurationen (`vm_ssd` und `vm_hdd`) während des Importvorganges von 1.000 Publikationen in OrientDB

Abbildung B.44.: Direkter Vergleich der Schreibrate, der unterschiedlichen Hardwarekonfigurationen (`vm_ssd` und `vm_hdd`) während des Importvorganges von 5.000 Publikationen in OrientDB

Literatur

Apache Software Foundation (2015). *Apache Jena. A free and open source Java framework for building Semantic Web and Linked Data applications*. Englisch. URL: https://jena.apache.org/index.html (besucht am 08. 09. 2015) (siehe S. 45, 47, 48).

Behr, Thomas (2015). *Sicherheit und Privatspähre in Datenbanksystemen. Seminarband zum Kurs 1912/19912 im WS 2014/2015*. Deutsch. Fernuniversität in Hagen. URL: http://dna.fernuni-hagen.de/Lehre-offen/Seminare/1912-WS1415/Seminarband.pdf (besucht am 26. 01. 2016) (siehe S. 55).

Busse, Johannes u. a. (2014). »Was bedeutet eigentlich Ontologie?« Deutsch. In: *Informatik-Spektrum* 37.4, S. 286–297. ISSN: 0170-6012. DOI: 10.1007/s00287-012-0619-2. URL: http://dx.doi.org/10.1007/s00287-012-0619-2 (siehe S. 44).

cbSolution.net (2011). *Dabase: relational vs. object vs. graph vs. document*. URL: http://www.cbsolution.net/techniques/ontarget/databases_relational_vs_object_vs (besucht am 17. 07. 2015) (siehe S. 7).

Cloud Security Allience (2013). »Top Ten Big Data Security and Privacy Challenges«. In: URL: https://cloudsecurityalliance.org (besucht am 31. 01. 2016) (siehe S. 61).

Codd, E. F. (1970). »A Relational Model of Data for Large Shared Data Banks«. In: *Commun. ACM* 13.6, S. 377–387. DOI: 10.1145/362384.362685. URL: http://doi.acm.org/10.1145/362384.362685 (siehe S. 5, 6).

Cyganiak, Richard, David Wood und Markus Lanthaler (2014). *RDF 1.1 Concepts and Abstract Syntax*. URL: http://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/ (besucht am 03. 09. 2015) (siehe S. 43).

Däßler, Rolf (2013). *MySQL 5. Das Einsteigerseminar*. 2. Aufl. bhv. ISBN: 978-3-8266-7510-2 (siehe S. 49, 50).

De Virgilio, Roberto, Antonio Maccioni und Riccardo Torlone (2014). »Model-driven design of graph databases«. In: *Conceptual Modeling*. Springer, S. 172–185 (siehe S. 62).

Domenjoud, Michel (2016). *Graph databases: an overview*. Englisch. URL: http://blog.octo.com/en/graph-databases-an-overview/ (besucht am 11. 01. 2016) (siehe S. 15, 16).

Dominguez-Sal, David u. a. (2011). »Performance Evaluation, Measurement and Characterization of Complex Systems: Second TPC Technology Conference, TPCTC 2010, Singapore, September 13-17, 2010. Revised Selected Papers«. In: Hrsg. von Raghunath Nambiar und Meikel Poess. Berlin, Heidelberg: Springer Berlin Heidelberg. Kap. A Discussion on the Design of Graph Database Benchmarks, S. 25–40. ISBN: 978-3-642-18206-8. DOI: 10.1007/978-3-642-18206-8_3. URL: http://dx.doi.org/10.1007/978-3-642-18206-8_3 (siehe S. 111).

Edlich, Stefan u. a. (2010). *NoSQL - Einstieg in die Welt nichtrelationaler Web 2.0 Datenbankend*. Hanser. ISBN: 978-3-446-42355-8 (siehe S. 1, 3, 10, 13, 17, 22, 23, 28, 30, 41, 127).

Elmasri, Ramez A. und Shamkant B. Navathe (2009). *Grundlagen von Datenbanksystemen. Bachelorausgabe*. 3. Aufl. Pearson Studium. ISBN: 978-3-86894-012-1 (siehe S. 54, 55).

Fiedler, Wolfgang (2004). »Verwalten von Metadaten in webbasierten Informationssystemen mit RDF«. Diplomarbeit. Technische Universität Graz: Instiut für Informationsverarbetiung und Computergestützte Neue Medien (siehe S. 42, 43).

Hunger, Michael (2014). *Neo4j 2.0. Eine Graphdatenbank für alle*. entwickler.press. ISBN: 978-3-86802-128-8 (siehe S. 23).

Hunger, Michael (2015). *On Neo4j Indexes, Match and Merge*. URL: http://jexp.de/blog/2015/04/on-neo4j-indexes-match-merge/ (besucht am 20. 08. 2015) (siehe S. 21).

Junghanns, Martin (2014). »Untersuchung zur Eignung von Graphdatenbanksystemen für die Anaylse von Informationsnetzwerken«. Masterarbeit. Instiut für Informatik, Universität Leipzig (siehe S. 22).

Kofler, Michael (2012). *MySQL 5.5 & 5.6. Band II: Administration*. ebooks.kofler. ISBN: 978-3-902643-08-7 (siehe S. 58, 59).

FH-Köln (2011). *Datenbanken Online Lexikon*. URL: http://wikis.gm.fh-koeln.de/wiki_db/Datenbanken/Neo4j (besucht am 28. 07. 2015) (siehe S. 22).

Kubacki, W. Mark (2010). »SQL vs. NoSQL« (siehe S. 9).

Malette, Stephen (2015). *Gremlin*. URL: https://github.com/tinkerpop/gremlin/wiki (besucht am 27. 07. 2015) (siehe S. 21).

MarkLogic (2014). *Die NoSQL-Generation: Das Dokumentenmodell*. Whitepaper. Skyper Villa, Taunusanlage 1, Frankfurt 60329: MarkLogic Corporation (siehe S. 10).

Neo Technology Inc. (2015). *Neo4j, the world's leading graph database*. Neo Technology Inc. URL: http://neo4j.com/ (besucht am 24. 07. 2015) (siehe S. 18, 28).

Neo4 Technology Inc. (2015). *The Neo4j Manual v2.2.3*. Neo Technology Inc. URL: http://neo4j.com/docs/ (besucht am 24. 07. 2015) (siehe S. 20, 21, 23, 27).

Ontotext (2014). *The truth about Triplestores. The top 8 things you need to know when considering a Triplestore*. Englisch. URL: http://ontotext.com/documents/white_papers/The-Truth-About-Triplestores.pdf (besucht am 11. 01. 2016) (siehe S. 44, 48).

Oracle Corporation (2015). *MySQL 5.6 Reference Manual*. Englisch. URL: http://dev.mysql.com/doc/refman/5.6/en/ (besucht am 09. 10. 2015) (siehe S. 50).

Orient Technologies (2015). *OrientDB manual - version 2.1*. Orient Technologies LTD. URL: http://orientdb.com/docs/last/index.html (besucht am 06. 08. 2015) (siehe S. 30, 31, 34–36, 40).

Piepmeyer, Lothar (2011). *Grundkurs Datenbanksysteme. Von den Konzepten bis zur Anwendungsentwicklung*. Hanser. ISBN: 978-3-446-42354-1 (siehe S. 7).

Redmond, Eric und Jim R. Wilson (2012). *Sieben Wochen, sieben Datenbanken. Moderne Datenbanken und die NoSQL-Bewegung*. Übers. von Peter Klicman. O'Reilly. ISBN: 978-3-86899-791-0 (siehe S. 25, 28).

Robinson, Ian, Jim Webber und Emil Eifrem (2013). *Graph Databases*. O'Reilly. ISBN: 978-1-449-35626-2 (siehe S. 1, 13–18, 61, 127).

Sahafizadeh, Ebrahim und Mohammad Ali Nematbakhsh (2015). »A Survey on Security Issues in Big Data and NoSQL«. In: *Advances in Computer Science: an International Journal* 4.4, S. 68–72 (siehe S. 1, 61).

Schwartz, Alexandra (2013). *Einführung in die Graphentheorie*. Vorlesungsskript. Emil-Fischer-Straße 30, Würzburg: Institut für Mathematik, Julius-Maximilians-Universität Würzburg. URL: http://www.mathematik.uni-wuerzburg.de/~schwartz/Lehre/Graphentheorie/SkriptGraphentheorieSchwartz.pdf (besucht am 21. 07. 2015) (siehe S. 11).

Sinha, Arnab u. a. (2015). »An Overview of Microsoft Academic Service (MAS) and Applications«. In: *Proceedings of the 24th International Conference on World Wide Web*. WWW '15 Companion. Florence, Italy: ACM, S. 243–246. ISBN: 978-1-4503-3473-0. DOI: 10.1145/2740908.2742839. URL: http://doi.acm.org/10.1145/2740908.2742839 (siehe S. 64, 108).

Small, Nigel (2015). *Neo4j Index Confusion*. URL: http://nigelsmall.com/neo4j/index-confusion (besucht am 27. 07. 2015) (siehe S. 21).

solid IT GmbH (2015). *DB-Engines*. URL: http://db-engines.com/de/ (besucht am 24. 07. 2015) (siehe S. 19, 29, 46, 49).

Stegmaier, Florian u. a. (2011). *Evaluation of Current RDF Database Solutions*. Paper. Universtiät Passau: Chair of Distributed Information Systems (siehe S. 44, 45, 48).

Steiner, Rene (2009). *Grundkurs Relationale Datenbanken. Einführung in die Praxis der Datenbankentwicklung für Ausbildung, Studim und IT-Beruf*. 7. Aufl. Vieweg+Teubner. ISBN: 978-3-8348-0710-6 (siehe S. 4, 9).

Stuber, Marcus (2012). *Eine Einführung in das Graphdatenbankmodell*. Seminararbeit. Institut für Informatik, Universität Leipzig (siehe S. 5, 10).

Tsolkas, Alexander und Klaus Schmidt (2010). *Rollen und Berechtigungskonzepte. Ansätze für das Identity- und Access Management im Unternehmen*. 1. Aufl. ISBN: 978-3-8348-1243-8. DOI: 10.1007/978-3-8348-9745-9 (siehe S. 56).

Unterstein, Michael und Günter Matthiessen (2012). *Relationale Datenbanken und SQL in Theorie und Praxis*. 5. Aufl. Springer Vieweg. ISBN: 978-3642289859. DOI: 10.1007/978-3-642-28986-6 (siehe S. 5–7).

Vicknair, Chad u. a. (2010). »A Comparison of a Graph Database and a Relational Database: A Data Provenance Perspective«. In: *Proceedings of the 48th Annual Southeast Regional Conference*. ACM SE '10. Oxford, Mississippi: ACM, 42:1–42:6. ISBN: 978-1-4503-0064-3. DOI: 10.1145/1900008.1900067. URL: http://doi.acm.org/10.1145/1900008.1900067 (siehe S. 1, 127).

Zerbst, Carsten (2015). »Die Graphdatenbank Orient DB. Verknüpfte Welten«. In: *Linux Magazin*, S. 86–90 (siehe S. 35, 41).

Zimmer, Dennis (2005). *VMware und Microsoft Virtual Server: Virtuelle Server im professionellen Einsatz*. Galileo Computing. ISBN: 3898427013 (siehe S. 105).