Institut für Parallele und Verteilte Systeme

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Umsetzung anwendungsspezifischer ETL-Prozesse im Data Lake

Magdalena Winckler

Studiengang:	Softwaretechnik
Prüfer/in:	PD Dr. rer. nat. habil. Holger Schwarz
Betreuer/in:	Corinna Giebler, M.Sc.
Beginn am:	12. August 2019
Beendet am:	19. Februar 2020

Kurzfassung

In der heutigen Zeit entstehen tagtäglich immer mehr Daten. Um diese nutzen zu können, müssen sie gespeichert, verarbeitet und analysiert werden. Da dies bei immer größer werdenden Datensätzen, welche gleichzeitig heterogen sein können, viele neue Anforderungen mit sich bringt, entstehen auch immer wieder neue Technologien. So wird es immer schwerer relevante Daten herauszufiltern oder diese zu bereinigen. Eine dieser Technologien ist der Data Lake. Ein Data Lake ist ein System in dem jegliche Arten von Daten gespeichert und zu einem späteren Zeitpunkt bei Verwendung bereitgestellt werden können. Oft wird jedoch ein Data Lake mit einem Hadoop-Cluster gleichgesetzt. Um einen Data Lake effizienter zu konzeptionieren, wird in der Literatur deshalb oft darauf hingewiesen, dass auch auf andere Technologien innerhalb eines Data Lakes zurückgegriffen werden soll. Gleichzeitig kann ein Data Lake in unterschiedliche Zonen aufgeteilt werden, um diesen effizienter zu nutzen. Die Aufteilung eines Data Lakes in Zonen, wird allgemein als Zonenmodell bezeichnet. Bisher gibt es jedoch nur wenige wissenschaftliche Arbeiten, welche sich sowohl mit der Konzeptionierung als auch mit der Implementierung eines Data Lakes mit mehreren Technologien beschäftigt.

In dieser Arbeit wird ein Konzept eines Data Lakes mit mehreren Technologien entworfen und dieses prototypisch umgesetzt. Um dies zu erreichen werden unterschiedliche Zonenmodelle und bereits in der Literatur vorhandene Anwendungen dieser auf einen Data Lake betrachtet. Anschließend wird ein Beispieldatensatz vorgestellt, anhand dessen unterschiedliche Anwendungsfälle entworfen werden. Um den Data Lake mit anderen Technologien zu erweitern, werden zudem unterschiedliche Datenbankarten vorgestellt. Darauf folgt in einer prototypischen Umsetzung die Übertragung der Daten von einem Hadoop-Cluster in die unterschiedlichen Datenbanken. Abschließend folgt eine Evaluation und eine Zusammenfassung der Arbeit.

Inhaltsverzeichnis

1 Einleitung **13**

2 Hintergrund **17**
2.1 Data Lakes 17
2.2 Das Zonenmodell 17

3 Verwandte Arbeiten **21**
3.1 Zikopoulos - Big Data Beyond the Hype 21
3.2 Vor- und Nachteile des Konzepts 24

4 Konzeptionierung eines Data Lakes **25**

5 Anwendungsfall **27**
5.1 Beispieldatensatz 27
5.2 Anwendungsfälle 28
5.3 Datenbanken 30

6 Implementierung **33**
6.1 Aufsetzen Hadoop und Datenbanken 33
6.2 Übertragung der Daten von Hadoop zu NoSQL-DBs 33
6.3 Übertragung der Daten von Hadoop zu Neo4J 41

7 Evaluation **45**
7.1 Datenbankabfragen über Schlüssel 47
7.2 Datenbankabfragen für häufig gemeinsam abgefragte Attribute 49
7.3 Datenbankabfragen auf spärlich besetzten Attributen 50
7.4 Datenbankabfragen auf stark vernetzten Daten 51

8 Zusammenfassung und Ausblick **55**

Literaturverzeichnis **57**

Abbildungsverzeichnis

2.1 Eines der verschiedenen Zonenmodelle [Madsen] 18
2.2 Zonenmodell von Madsen in anwedungsfall(un)abhängigen Bereich aufgeteilt . . 19

4.1 Grafische Darstellung des in diesen Kapitel entworfenen Konzepts 26

5.1 Auszug aus dem Beispieldatensatz 28
5.2 Beziehungen zwischen den Beispieldatensätzen 29

6.1 Beziehungen zwischen title.basics und title.episodes 39
6.2 ER-Digramm zur Visualisierung des Zusammenhangs zwischen Name und Title . 42

7.1 Vergleich der Abfragedauern von Spark und HBase 48
7.2 Vergleich der Abfragedauern von Spark und MongoDB 50
7.3 Vergleich der Abfragedauern von Spark und Cassandra 51
7.4 Vergleich der Abfragedauern von Spark und Neo4J 53

Tabellenverzeichnis

3.1 Zikopolous Zonenmodell mit verwendeten Technologien 23

Verzeichnis der Listings

6.1 Allgemeiner Aufbau der Klasse Processor 34
6.2 run()-Methode der Klasse DBImporter 35
6.3 createTable()-Methode für HBase 36
6.4 process()-Methode für HBase 37
6.5 createTable()-Methode für MongoDB 38
6.6 process()-Methode für MongoDB 38
6.7 Unterdokumente in MongoDB erstellen 39
6.8 createTable()-Methode für Cassandra 40
6.9 process()-Methode für Cassandra 41
6.10 Erstellen von Name-Knoten mit Cypher 43
6.11 Erstellen von Title-Knoten mit Cypher 43
6.12 Erstellen von Kanten zwischen Title-Knoten und Name-Knoten mit Cypher . . . 44
6.13 Datenbankabfrage mit Cypher 44

7.1 PerformanceTester 45
7.2 PerformanceTester Berechnungen 46
7.3 Abfrage für HBase in Spark 47
7.4 Abfrage für MongoDB in Spark 49
7.5 Abfrage für Cassandra in Spark 50
7.6 Datenbankabfrage mit Cypher 51
7.7 Abfrage für Neo4J in Spark 52

1 Einleitung

In der heutigen Zeit entstehen mehrere Trillionen Bytes digitale Daten pro Tag [Kro18]. Nach Schätzungen der Firma Seagate wird sich somit die jährliche Größe der globalen Datensphäre von 33 Zetabyte im Jahr 2018 auf etwa 175 Zetabyte im Jahr 2025 erhöhen [GRR19]. Dieser Trend bringt die Herausforderung mit sich, dass eine sehr große Datenmenge je nach Nutzen gespeichert, verarbeitet und analysiert werden muss.

Aus diesem aufsteigenden Trend entstand der Begriff Big Data, welcher ein massives Volumen an strukturierten, halb-strukturierten und unstrukturierten Daten beschreibt [Jac09]. Aus dieser ständig wachsenden Datenmenge ist es schwer, ohne größeren Aufwand, relevante Daten herauszufiltern und diese für spätere Anwendungsfälle zu reinigen und aufzuwerten. Die vielen unterschiedlichen Wege an Daten zu kommen, gepaart mit den vielen Möglichkeiten der Speicherung und unterschiedlichen Formen der Darstellung, machen es oft unmöglich diese Rohdaten gleichermaßen direkt zu verarbeiten [WMS94]. Es entstehen viele heterogene Daten, welche nur einen teilweise bekannten oder ganz unbekannten Wert haben und erstmals bereinigt werden müssten, um einen Nutzen aus ihnen zu ziehen.

Ein naheliegendes Ziel ist es vorerst alle Daten mit möglichst wenig Aufwand zu verarbeiten, um so alle Daten zu einem späteren Zeitpunkt zur Verfügung zu haben und weiterverwenden zu können [Mad15]. Dabei wird zunächst nicht danach unterschieden, welchen Wert die Daten haben.

In einem ursprünglichen Ansatz wurden die Daten meist zuerst aufbereitet, um diese darauf in eine Speicherstruktur, wie z.B. ein Enterprise Data Warehouse, einzupflegen [Mad15]. Das vorhergehende Aufbereiten bringt jedoch einen hohen zeitlichen Aufwand mit sich und macht Daten, deren Wert als einmal “nicht von Nutzen” eingeschätzt wurde und somit nicht in das Data Warehouse eingepflegt wurden, nicht erneut zugänglich [BG13].

Um dieses Problem zu adressieren wird ein Speichersystem benötigt, in welchem Daten auch unaufbereitet gespeichert werden können. So kann zu einem späteren Zeitpunkt noch auf alle Daten zugegriffen werden und die Daten können einem spezifischen Anwendungsfall entsprechend, aufbereitet und verarbeitet werden. Um dies umzusetzen, wurde der Data Lake entworfen, welcher in der Lage ist jegliche Arten von Daten zu speichern und diese für eine flexible Analyse ohne vordefinierten Anwendungsfall verfügbar zu machen [TD16].

Der Data Lake umfasst ein System, in dem jegliche Daten gespeichert werden, welche ein Unternehmen analysieren und verarbeiten möchte [Mat17]. Dabei basiert der naive Ansatz eines Data Lakes darauf alle Daten im Originalformat in einem Hadoop-Cluster zu sammeln und zu speichern. Damit können Analysten die Daten verständlich machen, wenn diese sie brauchen [Mad15]. Die Arbeit mit Rohdaten ist jedoch oft ineffizient und langsam, weist zudem eine sehr schlechte Wiederverwendbarkeit auf und macht den Data Lake nicht für alle Nutzergruppen zum Durchsuchen geeignet [Mad15].

Um den Data Lake für unterschiedliche Daten und Use Cases verwendbar zu machen, sollte dieser nicht mit einem Hadoop-Cluster gleichgesetzt werden. Stattdessen handelt es sich bei dem Data Lake um ein Konzept, welches mit verschiedensten Technologien umgesetzt werden kann. In der Literatur wird oft sogar vorgeschlagen, den Data Lake aus mehreren Speichertechnologien aufzubauen [GM19]. Bei der Speicherung, in unterschiedlichen Speichertechnologien, kann je nach Art der Daten die passendste Technologie gewählt werden. Zusätzlich können Daten vorverarbeitet werden, um die Effizienz zu erhöhen. Auch ist es oft sinnvoll Daten, dem Anwendungsfall entsprechend, in unterschiedlichen Speichertechnologien zu speichern [GM19]. Dies hat den Vorteil, dass Daten, welche einmal verarbeitet wurden, in diesem Verarbeitungsgrad einen schnelleren Zugriff erlauben und bereits in der, für den Anwendungsfall, am besten passenden Speicherform vorliegen können.

Das Modell, welches den Data Lake nach Use Case und Verarbeitungsgrad aufteilt, wird allgemein hin als Zonenmodell bezeichnet und steigert die Effizienz des Data Lakes [GM19]. Es bleibt allerdings unklar, wie die polystrukturierte Speicherlandschaft mit dem Zonenmodell zusammen passt. So kann es beispielsweise sein, dass Daten beim Wechsel von einer Zone in die andere auch das Speichersystem wechseln. Da bisher in nur wenig Literatur überhaupt von mehreren Speichersystemen innerhalb eines Data Lakes gesprochen wird, gibt es keine detaillierten wissenschaftlichen Ansätze, um die Daten innerhalb und zwischen den verschiedenen Zonen sinnvoll zu verwalten.

Das Ziel der Arbeit ist es daher, die Datenübertragung zwischen verschiedener Speichersysteme, auf der Basis eines Zonenmodells, zu untersuchen.

Um dies zu erreichen werden folgende Beiträge in dieser Arbeit geleistet:

- An einem Beispielkontext werden verschiedene Data Lake Use Cases hergeleitet und untersucht.
- Es wird eine an die Rahmenbedingungen angepasste Speicherlandschaft erstellt.
- Darauf folgt die prototypische Implementierung der Datenbewegung zwischen den Speichersystemen im Rahmen der Vorverarbeitung.
- Zuletzt folgt eine Evaluation.

1.0.1 Gliederung

Die Gliederung der Arbeit ist wie folgt:

Kapitel 2: Hintergrund: In diesem Kapitel werden kurz die Begrifflichkeiten Data Lake und Zonenmodell veranschaulicht, damit mit diesen anschließend gearbeitet werden kann.

Kapitel 3: Verwandte Arbeiten: Bevor wir mit der Konzeptionierung eines Data Lakes anfangen können, wird in diesem Kapitel auf bereits vorhandene Konzepte eines Data Lakes eingegangen.

Kapitel 4: Konzeptionierung eines Data Lakes: In diesem Kapitel wird anhand der gewünschten Anforderungen ein neues Konzept für einen Data Lake entworfen.

Kapitel 5: Anwendungsfall: Um das neue Konzept prototypisch umsetzten zu können, wird ein Anwendungsfall gebraucht. Dazu wird zuerst ein Beispieldatensatz vorgestellt, aus welchem anschließend mögliche Anwendungsfälle erarbeitet werden. Zusätzlich werden in diesem Kapitel unterschiedliche Datenbanken vorgestellt, mit welchen anschließend die erarbeiteten Anwendungsfälle abgedeckt werden können.

Kapitel 6: Implementierung: Da nun Konzept und Beispieldatensatz stehen, wird in diesem Kapitel nun ein Data Lake prototypisch umgesetzt und die Umsetzung beschrieben.

Kapitel 7: Evaluation: Dieses Kapitel dient zur Evaluation des in Kapitel 4 konzeptionierten Data Lakes. Hierzu werden Messungen aus Datenbankabfragen und direkten Abfragen auf einem Hadoop-Cluster gegeneinander verglichen.

Kapitel 8: Zusammenfassung und Ausblick: Als Abschluss dieser Arbeit wird diese hier kurz zusammengefasst und es wird auf andere mögliche Technologien zum Umgang mit Big Data eingegangen.

2 Hintergrund

Um einen Data Lake sinnvoll konzeptionieren zu können, sollte man sich zuerst die Herkunft des Begriffes anschauen. In diesem Kapitel wird kurz erklärt, was ein Data Lake ist, siehe Abschnitt 2.1 und es wird auf bereits vorhandene Zonenmodelle des Data Lakes eingegangen, siehe Abschnitt 2.2.

2.1 Data Lakes

Der Begriff des Data Lakes entstand mit dem Aufkommen von Big Data. Big Data bringt die Herausforderung mit sich, dass immer mehr Daten gespeichert und zum Durch- und Untersuchen bereitgestellt werden sollen [Zik15]. Hierbei liegt insbesondere nicht nur eine große Menge an Daten vor, sondern die Daten können auch in verschiedenen Variationen und aus unterschiedlichen Datenquellen vorliegen [HGQ16].

Der Data Lake ermöglicht die Verwaltung und Analyse dieser Daten. Dabei ist es nicht notwendig, Use Cases vorzudefinieren, da der Data Lake zu jedem Zeitpunkt beliebige Analysen auf den verwalteten Daten erlaubt. Dazu werden in einem Data Lake Rohdaten gespeichert und Funktionen für die On-Demand-Integration bereitgestellt [HGQ16]. Das Zugreifen auf Rohdaten wird insbesondere durch eine Metadatenbeschreibung sichergestellt [HGQ16]. Das bedeutet, dass möglichst früh möglichst viele Metadaten extrahiert werden sollten, um den Informationsgehalt des Datensatzes zu erhalten [HGQ16]. Insgesamt stellen Data Lakes somit die Brücke zwischen der Speicherung von heterogenen Daten und der Möglichkeit diese Daten durch- und untersuchen zu können, dar [HGQ16]. Hierbei werden die Daten meist vorerst im Originalformat gespeichert. Insgesamt müssen Data Lakes somit drei Anforderungen entsprechen. Sie müssen Daten erfassen, Daten verarbeiten und Daten zur Nutzung bereitstellen [Mad15].

2.2 Das Zonenmodell

Um die in Abschnitt 2.1 genannten Anforderungen zu erfüllen ist es sinnvoll Data Lakes in unterschiedliche Zonen zu unterteilen. Eine Zone ist hierbei ein unterteilter Bereich zum Speichern, Bearbeiten und Zugreifen auf Informationen, basierend auf bestimmten Charakteristiken und Zweck [Zik15]. Dabei agieren die unterschiedlichen Zonen grundsätzlich unabhängig voneinander, so können Daten zu einer Zeit erfasst und zu einer anderen verarbeitet werden [Mad15]. Entschieden in welcher Zone Daten gespeichert werden, wird je nach Langlebigkeit und Verarbeitungsstufe. Hierbei werden in der Literatur unterschiedliche Zonenmodelle beschrieben, welche letztendlich jedoch sehr ähnliche Ansätze haben [Gor16; PWD17; Sha18]. Eines dieser Zonenmodelle wird auch von [Mad15] in seinem Paper beschrieben.

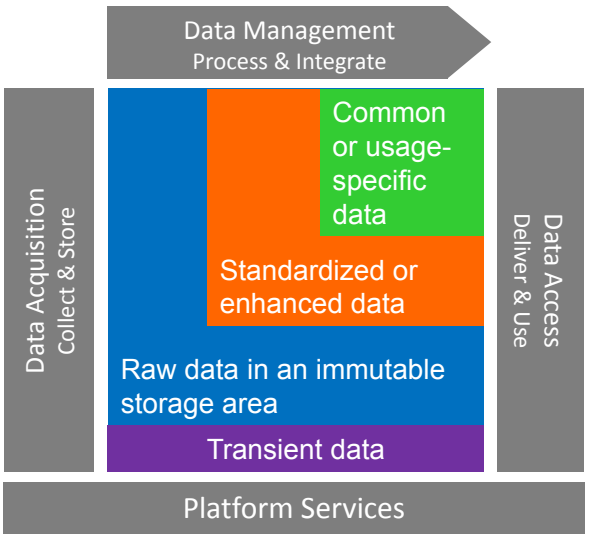

Abbildung 2.1: Eines der verschiedenen Zonenmodelle [Madsen]

Hierbei unterteilt sich das Zonenmodell in vier unabhängige Zonen, siehe Abbildung 2.1. Die erste Zone ist die *'Collection Zone'*, welche dafür gedacht ist, Rohdaten zu sammeln. Hier werden die Daten bei Ankunft möglichst schnell gespeichert, um gleich wieder für den Zugriff erreichbar zu sein. In dieser Zone befinden sich nur bisher unbearbeitete Daten, welche erst zu einem späteren Zeitpunkt verarbeitet werden. In der Abbildung 2.1 stellt der blaue Bereich die Collection Zone dar.

Die nächste Zone ist die sogenannte *'Management Zone'*. Diese wird in der Abbildung 2.1 als orangener Bereich dargestellt. In dieser Zone befinden sich Daten, welche bereits an Standards angepasst, verbessert und abgeleitet wurden. Durch das Standardisieren und Bereinigen der Daten wird eine bessere Verfügbarkeit von Informationen und ein leichterer Zugriff gewährleistet.

Als Nächstes kommt die *'Delivery Zone'*, der grüne Bereich, welche die Zone für häufig benutzte oder auch anwendungsspezifische Daten darstellt. In dieser Zone sollten Daten in einheitlichen Formaten und mit standardisierten Namen vorliegen, um die Verwendung möglichst einfach zu gestalten.

Zuletzt gibt es noch die *'Transient Zone'*, welche für kurzlebige oder auch Daten mit nur einmaliger Nutzung gedacht ist. Sie stellt einen Zwischenspeicherplatz für Daten dar, um mit diesen arbeiten und für einen anderen Zweck vorbereiten zu können. Hierbei ist es nicht vonnöten diese Daten auf Dauer zu speichern. Sie wird als lila Bereich in der Abbildung 2.1 dargestellt und kann in jeder Stufe der Datenverarbeitung als Hilfsmittel verwendet werden.

Allgemein lassen sich alle Zonenmodelle in zwei Bereiche unterteilen: anwendungsabhängige und anwendungsunabhängige Zonen (z.B. in [Gor16; PWD17]). In dem Zonenmodell von Madsen, siehe Abschnitt 2.2, stellt hierbei die Collection Zone den anwendungsfallunabhängigen Teil dar und die Management Zone zusammen mit der Delivery Zone stellen den anwendungsfallabhängigen Teil dar, vgl. Abbildung 2.2. Hierbei werden in der Collection Zone, unabhängig vom späteren Anwendungsfall, sämtliche Daten gespeichert und stehen bisher in keiner Verbindung zu ihrem späteren Gebrauch. Im Gegensatz zu diesem anwedungsfallunabhängigen Bereich werden die

Daten in der Management Zone bereinigt und an Standards angepasst, dies kann nicht geschehen, ohne bereits zu wissen, wofür die Daten später verwendet werden sollen. Somit sind je nach Anwendungsfall andere Informationen wichtig, welche aus dem Datensatz herausgefiltert werden sollen, was automatisch mit einer anwendungsfallabhängigen Bereinigung einhergeht. Auch ist es oft sinnvoll, je nach späteren Anwendungsfall, die Daten in bestimmte Formate zu bringen, um diese später möglichst schnell und einfach erreichbar zu machen. Bei der Übertragung der Daten von der Management Zone zu der Delivery Zone bleiben die Daten in ihrem anwedungsfallspezifischen Kontext bestehen und somit ist diese gleichfalls ein anwendungsfallabhängiger Bereich. Da die Transient Zone hauptsächlich als Zwischenspeicher verwendet wird und somit für alle Zonen von Nutzen ist, wird diese keinem der beiden Teile zugeordnet.

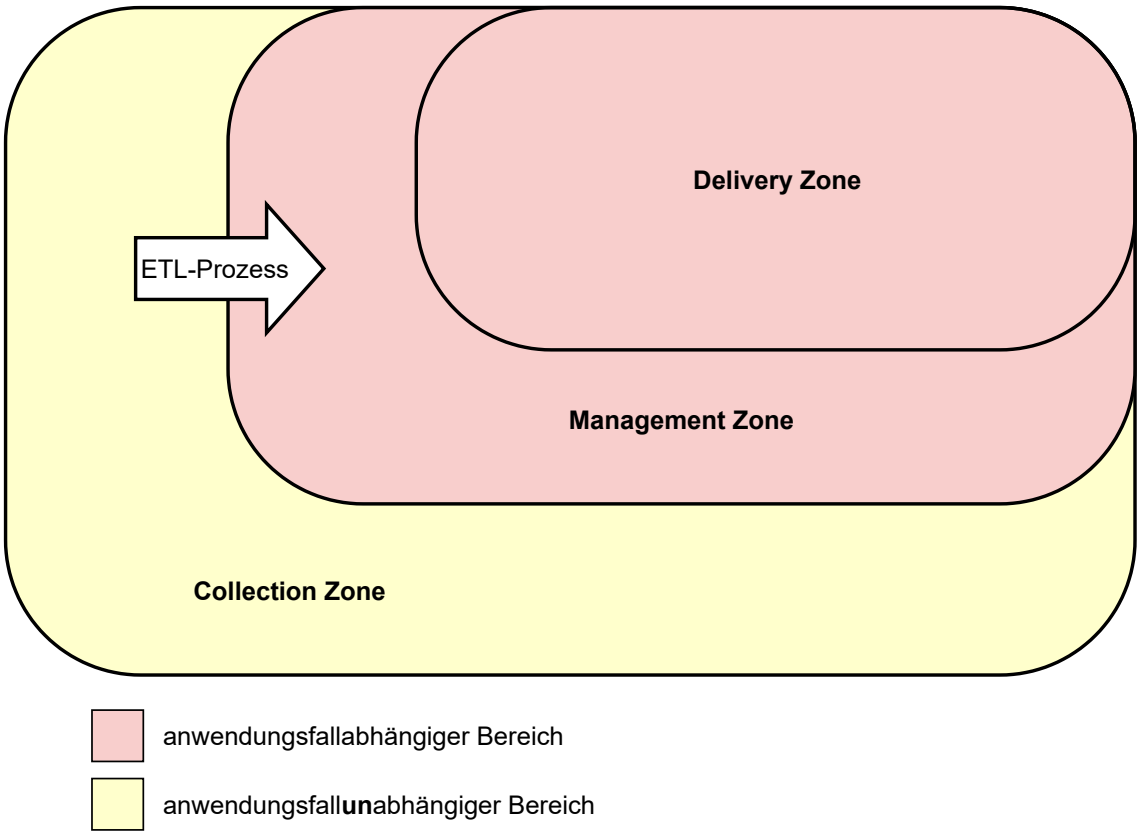

Abbildung 2.2: Zonenmodell von Madsen in anwedungsfall(un)abhängigen Bereich aufgeteilt

Für die in dieser Arbeit geleisteten Beiträge ist die Unterteilung in diese beiden Bereiche ausreichend. Im Rest der Arbeit wird darum auf eine weitere Detaillierung des verwendeten Zonenmodells verzichtet. Gleichzeitig liegt der Fokus der Arbeit auf der Datenübertragung zwischen den beiden Bereichen, siehe Pfeil in Abbildung 2.2.

3 Verwandte Arbeiten

Trotz vielfältiger Zonenmodellen gibt es kaum wissenschaftliche Arbeiten, welche das Konzept des Zonenmodells auf einen Data Lake anwenden. Somit stellt sich die Frage, wie das Zonenmodell mit der heterogenen Speicherlandschaft vereinbar ist und auf welche Aspekte bei der Implementierung eines Data Lakes Rücksicht genommen werden sollten.

3.1 Zikopoulos - Big Data Beyond the Hype

Eine mögliche Anwendung eines Zonenmodels auf einen Data Lake wird von Zikopoulos u.a. in seinem Buch "Big Data Beyond the Hyp" beschrieben [Zik15]. Hierbei handelt es sich um die einzige wissenschaftliche Arbeit, welche sowohl das Zonenmodell als auch die Implementierung eines Data Lakes behandelt. Dabei verwendet Zikopoulos das Zonenmodell, um eine bessere Balance zwischen physischer Umsetzung, ökonomischer Kostenkurve und der Datenaufbereitung zu finden. Hierbei wird ein Data Lake beschrieben, auf welchem zusätzlich zu den typischen Analysen ein Data Warehouse aufgebaut wird. Dabei geht er auch auf den Entstehungsprozess von gereinigten Daten ein, der darin besteht Daten aus mehreren, gegebenenfalls auch unterschiedlich strukturierten, Datenquellen zu extrahieren, diese den Anwendungsfall und den Speicheranforderungen bedingt zu transformieren und die Daten letztendlich in eine Datenbank einzupflegen. Der Prozess der Datenbeschaffung wird allgemeinhin als ETL-Prozess (Extract-, Transform-, Load-Prozess) bezeichnet. Bei der Verwendung eines Data Warehouses kam es aufgrund der Möglichkeit, dass Daten auch innerhalb der Speichertechnologie transformiert werden können, oftmals zu der Anpassung des Datenbeschaffungsprozesses hin zu einem ELT-Prozess.

Um auch mit großen Datenmengen umgehen zu können, hat Zikopolous das Data Warehouse mit anderen Speichertechnologien erweitert und die Speicherlandschaft in Zonen, vergl. Tabelle 3.1 aufgeteilt. Allgemein beschreibt Zikopolous acht Zonen in seinem Konzept. Die erste Zone ist die sogenannte *'Landing Zone'*, welche mit der in Abschnitt 2.2 beschriebenen 'Collection Zone' vergleichbar ist. Auch hier werden Rohdaten gesammelt und liegen in ihrer ursprünglichen Speicherform vor. Dies hat wieder den Vorteil, dass alle Daten gespeichert werden können und erst zu einem späteren Zeitpunkt auf ihren Informationsgehalt geprüft werden müssen und somit auch falls zuerst als irrelevant eingestuft zu einem späteren Zeitpunkt immer noch zur Verfügung stehen. Anders als bei einer 'Staging Area', in welcher Daten einfach nur als Kopie vorliegen, werden Daten hier in Hadoop gespeichert, welches selbst bereits eine Landing Zone mit möglichen Suchfunktionalitäten darstellt. Trotzdem ist es laut Zikopolous nicht zu empfehlen direkt auf Hadoop zuzugreifen, sondern die Daten vorher aufzubereiten.

Die nächste Zone ist die *'Exploration Zone'*. Daten werden hier noch nicht in ein bestimmtes Schema gebracht, somit bleiben mögliche Verknüpfungen zwischen den Daten bestehen und können "erforscht" werden. Auch diese Zone ist mit Hadoop realisiert, welche die schemafreie Speicherung

unterstützt. Zikopolous setzt diese Zone mit einem Sandkasten gleich, einem Spielraum, um Daten zu testen und deren Umfeld kennenzulernen. Hierbei soll nicht nur auf offensichtliche Verlinkungen, wie über Fremdschlüssel geachtet werden, sondern auch über Verbindungen, welche nicht unbedingt im ersten Augenblick offensichtlich werden, wie Korrelationen zwischen Attributen. Durch die Möglichkeit die Daten in dieser Zone beliebig verändern zu können, soll man einen anderen Blick auf den Datensatz bekommen. Ziel ist es den Daten ein erstes Schema zu geben und den Informationsgehalt erreichbar zu machen. Falls in der Exploration Zone Daten nicht mehr benötigt werden, können diese einfach gelöscht werden, da weiterhin ein Backup aller Daten in der Landing Zone vorliegen.

Um Daten noch genauer untersuchen zu können, gibt es die sogenannte *'Deep Analytic Zone'*. Diese zielt darauf ab tiefere und performantere Analysen zu betreiben als es in Hadoop direkt möglich wäre. Um dies zu ermöglichen, werden Daten hier in RDBMS-Technologien übertragen, welche für ihre gute Performanz bekannt sind. Ein Beispiel für eine Datenbank, welche für Analysen gut geeignet ist, ist die dashDB.

Ähnlich der Deep Analytic Zone ist die *'New Staging Zone'*. Hierbei gibt es zwei unterschiedliche Ansätze. Entweder hat man die Daten zur weiteren Analyse in ein anderes System übertragen, um diese dort zu transformieren und ans Zielsystem auszuliefern oder man hat die Daten erst ins Zielsystem übertragen und innerhalb der Datenbank weiter verarbeitet. Bei einem hohen Datenaufkommen kann das Übertragen jedoch lange dauern und es entstehen viele Abhängigkeiten zwischen den einzeln übertragenen Datensätzen. Da das bearbeiten von Daten in zwei unterschiedlichen System jedoch zu Komplexität und Zeitverlust führt, bleibt Zikopolous für diesen Verarbeitungsschritt weiterhin in Hadoop, da Hadoop gleichfalls eine parallele Verarbeitung erlaubt und gleichzeitig die Metadaten der Informationen erhält. Ziel der New Staging Zone ist es hauptsächlich die Daten in eine Struktur zu bringen, welche mit der Struktur des Zielsystems übereinstimmt.

Um die nun strukturierten Daten zu archivieren, gibt es die *'Queryable Archive Zone*. Diese wird gleichfalls mit Hadoop realisiert. Hierbei muss an jedoch berücksichtigen, dass Hadoop keine vergleichbar performante Lösung darstellt, wie eine relationale Umsetzung. Jedoch bietet Hadoop die Möglichkeit einer 'Lower Cost Storage' Lösung, da die Daten weiterhin in Hadoop gespeichert werden können und nicht gleich verarbeitet werden müssen. Auch bietet Hadoop mit passenden Tools, wie beispielsweise IBM Big SQL, einen vollen SQL-Zugriff, welcher die Daten im Speichersystem erreichbar macht. Durch diese Möglichkeiten, die Hadoop bietet, ist es ohne größeren Aufwand möglich aus der Staging Zone eine Queryable Zone zu machen. Eine vergleichbare Zone zum dauerhaften Speichern von Daten wird in dem Zonenmodell von Madsen, siehe Abschnitt 2.2, nicht berücksichtigt.

Falls jedoch Daten öfters gebraucht werden, ist eine zuverlässige, robuste und leistungsfähige Umgebung vonnöten. Diese wird in der *'Trusted Data Zone'* durch ein Enterprise Data Warehouse gestellt. Hierbei werden die Informationen in gesammelter, konsolidierter Ansicht bereitgestellt und ermöglichen somit eine zuverlässige und schnelle Abfrage. In der Trusted Data Zone sollten nur wichtige und aktuelle Daten vorliegen. Daten, die nicht mehr verwendet werden, können jederzeit bedenkenlos gelöscht werden, da diese immer noch, falls nötig, als Backup in Hadoop vorliegen. Da sich in dieser Zone häufig verwendete Daten, welche bereits in ein Data Warehouse eingepflegt wurden, befinden, kann man diese mit der Delivery Zone aus Abschnitt 2.2 vergleichen. Auch hier befinden sich nur Daten, welche häufig gebraucht werden und somit möglichst schnell und einfach erreichbar sein sollten.

Zone	Verwendete Technologien
Collection Zone	Hadoop
Exploration Zone	Hadoop
Deep Analytic Zone	dashDB (RDBMS)
New Staging Zone	Hadoop
Queryable Archive Zone	Hadoop
Trusted Data Zone	Enterprise Data Warehouse
Business Reporting Zone	dashDB (RDBMS), BLU Acceleration
Real-Time Processing And Analytic Zone	IBM's Info-Sphere Stream

Tabelle 3.1: Zikopolous Zonenmodell mit den verwendeten Technologien

Ebenfalls ähnlich der Delivery Zone aus Abschnitt 2.2 ist die *Business Reporting Zone*. In dieser Zone liegt der Schwerpunkt auf die in Unternehmen benötigten unterschiedlichen Datenstrukturen und Sichten. Also wie die benötigten Daten dem Anwender präsentiert werden. Hierzu können Daten aus dem Data Warehouse oder aus Hadoop geladen werden. Um an Daten aus Hadoop zu gelangen, können unterschiedliche RDBMS-Systeme, wie dashDB, oder auch die Sammlung BLU Acceleration, welche von IMB für analytische Datenbank-Workloads entworfen wurde, verwendet werden. Insgesamt ist das Ziel dieser Zone die Abfragezeiten zu verringern, hierzu können auch unterschiedliche In-Memory-Datenbanken verwendet werden.

Zuletzt gibt es noch die *'Real-Time Processing And Analytic Zone'*, welche nochmals ein extra Augenmerk auf die bestehenden Verbindungen zwischen den einzelnen Daten richtet. Da eine höhere Menge an Daten, welche zudem oftmals nur halb oder gar ganz unstrukturiert ist, es unmöglich macht alles zu speichern, wird dies oft gar nicht erst versucht. Dabei gehen häufig wichtige Aspekte unter, da erst gar keine Analyse stattfindet. Um möglichst vielen Daten Informationen zu entlocken, werden diese nicht mühsam aufbereitet und gepflegt, sondern direkt analysiert und transformiert. Um dies als Echtzeit Verarbeitung auf Hadoop zu realisieren, verwendet Zikopolous IBM's Info-Sphere Stream Technologie. Somit bekommt diese Zone als Input reine Rohdaten und versucht aus diesen gefilterte und kausal zusammenhängende neue Daten zu erzeugen, welche wiederum in eine der anderen Zonen eingepflegt werden kann. Diese Zone ist mit der Transient Zone aus Abschnitt 2.2 vergleichbar. Auch hier werden die Daten aufbereitet und bearbeitet, um sie anschließend für die Weiterverarbeitung an die anderen Zonen weiterzugeben. Allerdings geht die Real-Time Processing And Analytic Zone noch einen ganzen Schritt weiter als die Transient Zone. Hier werden Daten nicht nur auf den offensichtlichen Informationsgehalt untersucht, sondern auch auf Informationen aufgrund des Zusammenspiels von Daten mit anderen Daten. Somit wird sichergestellt, dass einem keine kausalen Zusammenhänge entgehen.

Im Rahmen der Arbeit liegt der Fokus auf der Batch-Verarbeitung, weshalb kein extra Augenmerk auf Streaming-Technologien gelegt wird.

3.2 Vor- und Nachteile des Konzepts

Insgesamt erzeugt Zikopolous ein Konzept, welches auf zwei altbewährten Konzepten aufbaut, die des Enterprise Data Warehouses und die des Data Lakes auf der Basis von Hadoop. Hierbei macht sich das Konzept jeweils die Vorteile der einzelnen Technologien zunutze und versucht die Nachteile durch die Verwendung der jeweils andren Technologie wett zu machen. Durch die Verwendung dieser beiden Technologien wird mit dem EDW bereits eine sehr effiziente Technologie mit schnellen und einfachen Zugriff auf die Daten verwendet. Gleichzeitig wird durch die Verwendung von einem HDFS bereits sichergestellt, dass das Konzept auch für halb- und unstrukturierte Daten verwendbar ist. Zusätzlich zeigt sich bei Zikopolous eine recht klare Richtung der Datenverarbeitung. Die Daten kommen in der Rohform an und werden dann von Zone zu Zone feiner verarbeitet und mehr an die Anforderungen der Zielarchitektur angepasst.

Nachteil des Konzepts ist, dass in den Vorverarbeitungsstufen fast ausschließlich Hadoop verwendet wird. Trotzdem wird davon ausgegangen, dass die Daten auch innerhalb von Hadoop mehrere Zonen durchwandern. Es wird jedoch nicht je nach Aufbereitungsart eine andere, passendere Speichertechnologie verwendet. Auch schränkt Zikopolous sein Konzept stark ein, da er grundsätzlich von einem Data Warehouse als Zielsystem ausgeht, was je nach Datenart und -eigenschaften nicht immer sinnvoll ist. Dazu sollte man in seinem Konzept beispielhaft auch andere insbesondere neuere Technologien bedenken, wie zum Beispiel No-SQL Datenbanken, welche selbst wiederum Vorteile mit sich bringen. Zusätzlich zu der Erweiterung durch andere Technologien ist es auch nicht immer sinnvoll eine Zone an eine Technologie zu knüpfen. So sollten sich die Zonen auf die Eigenschaften der momentanen Verarbeitungsstufe der Daten beziehen und nicht auf die Eigenschaften des Speichersystems.

Somit stellt Zikopolous Konzept bereits eine solide Umsetzung eines Zonenmodells dar, jedoch gibt es auch noch einige Verbesserungsmöglichkeiten.

4 Konzeptionierung eines Data Lakes

In diesem Kapitel wird ein beispielhaftes Speicherkonzept für einen Data Lake vorgestellt. Hierzu wird ein Konzept erstellt, welches eine geeignete Datenhaltung innerhalb eines Data Lakes konzeptioniert.

Um den Data Lake sowohl für anwendungsfallabhängige, sowie für anwedungsfallunabhängige Daten als Speicherlandschaft geeignet zu machen, wird dieser in verschiedene Technologien unterteilt.

Hierbei bleibt bestehen, dass die frisch extrahierten Rohdaten, welche unabhängig von dem späteren Anwendungsfall vorliegen, vorerst in Hadoop gespeichert werden, siehe anwedungsfallunabhängigen Bereich inAbbildung 4.1. Das hat den Vorteil, dass alle Daten gespeichert werden können und die Daten vorerst keiner bestimmten Struktur entsprechen müssen.

Um aus den Daten Informationsgehalt zu entnehmen, müssen diese bereinigt und aufgewertet werden. Dies geschieht nun je nach späterem Anwendungsfall. Im Gegensatz zu Zikopolous Konzept in Kapitel 3 werden hier die Daten nicht nur in ein Data Warehouse übertragen, sondern je nach spezifischen Anforderungen des Anwendungsfalls in unterschiedliche Zielsysteme, siehe Abbildung 4.1. Hierbei ist es wichtig, dass die Daten für spätere Anfragen erreichbar und verständlich gemacht werden. So muss durch die richtige Interpretation der Daten der Informationsgehalt eindeutig gemacht werden. Dies geschieht insbesondere durch die Verwendung von Metadaten.

Ist sichergestellt, welche Anforderungen an den Datensatz gestellt werden, wird nun eine möglichst passende Speicherform benötigt. Hier bieten sich Datenbanken an, welche ermöglichen auch komplexe und flexible Abfragen auf unstrukturierten Daten zu machen und welche auch bei großen Datensätzen durch ihre horizontale Skalierbarkeit verwendbar sind.

In Zikopolous Data Lake hatten die Daten zudem eine Verarbeitungsrichtung von grob zu fein. Dementsprechend mussten Rohdaten, falls diese die Grundlage für unterschiedliche Informationen waren, immer von Anfang an wieder aufgearbeitet werden. In diesem Konzept sollen jedoch einmal verarbeitete Daten und Ergebnisse von Analysen erneut erreichbar gemacht werden. Um dies zu ermöglichen muss gegeben sein, dass Daten auch zwischen den unterschiedlichen Speicherformen hin- und hergeschoben werden können.

Unabhängig von den Datenbanken kann weiterhin ein Data Warehouse als zusätzliches Zielsystem verwendet werden, siehe Abbildung 4.1. Dies kann sowohl direkt aus Hadoop befüllt werden als auch aus den Ergebnissen von Abfragen auf den Datenbanken beruhen.

Um eine für den Anwendungsfall passende Datenbank auswählen zu können, muss man sich vorab die Vor- und Nachteile der einzelnen Datenbanken anschauen.

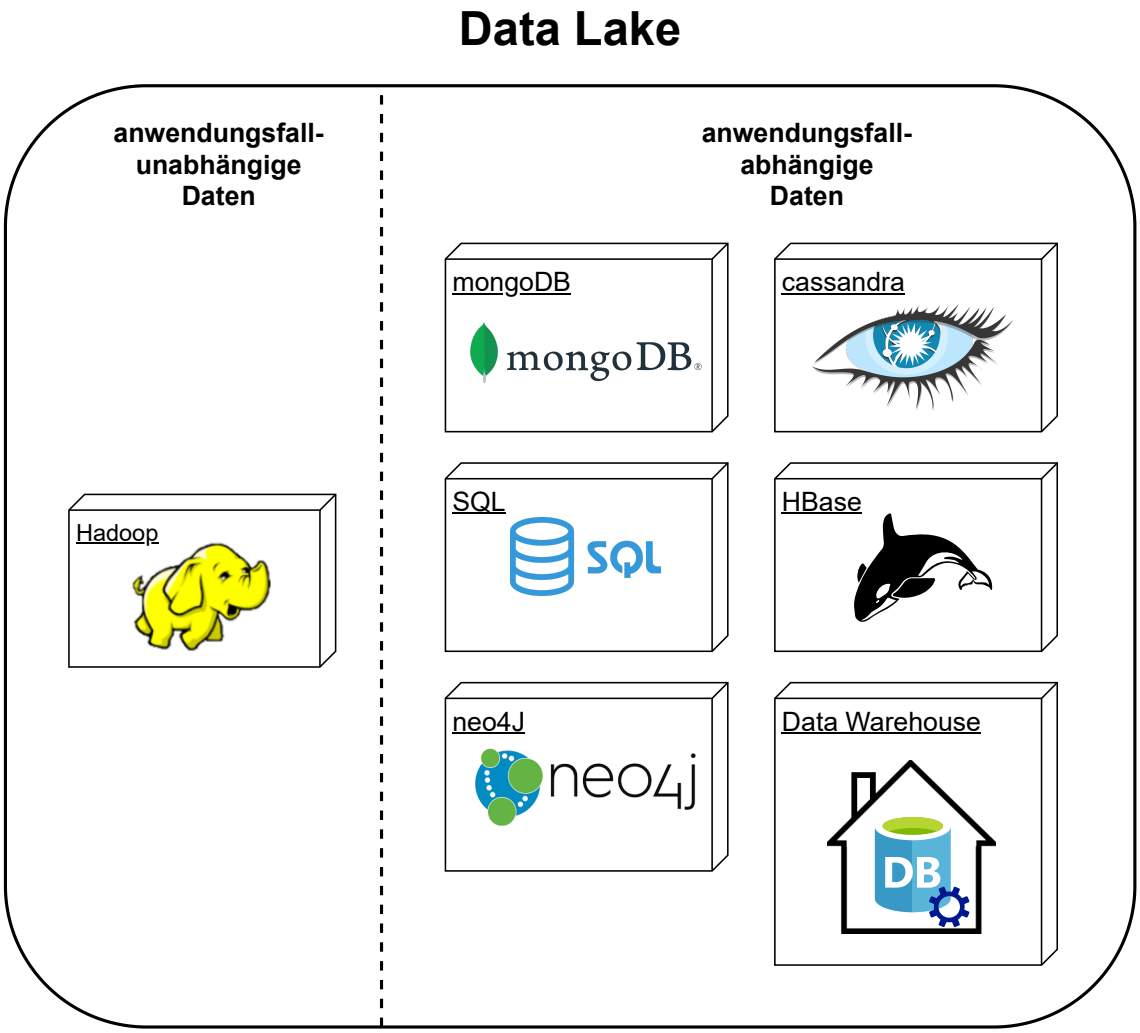

Abbildung 4.1: Grafische Darstellung des in diesen Kapitel entworfenen Konzepts

5 Anwendungsfall

In diesem Kapitel wird zuerst der für die prototypische Umsetzung gewählte Beispieldaten in Abschnitt 5.1 angeschaut, um an diesen unterschiedliche Anwendungsfälle herauszuarbeiten, siehe Abschnitt 5.2.

5.1 Beispieldatensatz

Um beispielhaft einen Data Lake implementieren zu können, braucht man einen Datensatz. Hierzu verwenden wir als Beispieldatensatz echte Daten aus der *Internet Movie Database*[1], kurz IMDb. Die IMDb ist eine Datenbank zu Filmen, Fernsehserien, Videoproduktionen und Computerspielen sowie über Personen, die daran mitgewirkt haben. Hierbei hatte diese im Dezember 2019 Einträge über 6,5 Mio. Filmproduktionen und zu über 10,4 Millionen Film- und Fernsehschaffenden. Bei dem Datensatz handelt es sich um sieben unterschiedliche Teildatensätze, welche jeweils formatiert als tab-separated-values (TSV) und UTF-8 vorliegen. Dabei stellen die Daten unterschiedliche Eigenschaften von Filmen/Serien dar, unter anderem besitzen alle Filme eine eindeutige ID, einen Titel, die Schauspielbesetzung und den Originaltitel. Der Datensatz wurde gewählt, da er sehr anschaulich und leicht verständlich ist und man somit gut die Relationen zwischen den einzelnen Daten nachvollziehen kann. Ein Auszug aus dem Datensatz ist in Abbildung 5.1 zu sehen. In dieser Datei werden Filmtitel mit ihren unterschiedlichen Eigenschaften, wie die Art der Produktion und das Genre, dargestellt. Hierbei ist jeder Titel anhand einer eindeutigen alphanumerischen Nummer, der sogenannten tconst, erkennbar. Referenzen zu anderen Dateien geschehen hierbei immer über eine dieser eindeutigen IDs.

Wie in Abbildung 5.1 zu sehen ist, kann die erste Zeile des Dokuments jeweils als 'Header', welcher beschreibt, was in den Spalten für Werte stehen, interpretiert werden. Bei Werten die nicht gesetzt wurden, steht ein '\N' als Platzhalter und bezeichnet sogenannte null-Werte.

Um einen Überblick über die unterschiedlichen Dateien und ihre Beziehungen zueinander zu bekommen, kann Abbildung 5.2 betrachtet werden. In dem Entity-Relationship-Diagramm gibt es jeweils für jede einzelne Datei eine Entität. Der Übersicht halber werden in dieser Abbildung nicht alle Attribute dargestellt.

Dabei stellen in Abbildung 5.2 blau hinterlegte Entitäten Dateien mit Informationen über den Titel dar und orange hinterlegte Entitäten Dateien mit Informationen über Personen. Wie in dem Diagramm zu sehen ist, stellt hierbei title.basics das Zentrum der Dateien dar, da alle andern Dateien Beziehungen zu dieser haben. Während die Dateien über Titelinformationen nur mit title.basics verbunden sind, sind die Dateien mit Personeninformationen auch untereinander verbunden.

[1] https://www.imdb.com/

1	tconst	titleType	primaryTitle	genres
2	tt0000001	short	Carmencita	Documentary,Short
3	tt0000002	short	Le clown et ses chiens	Animation,Short
4	tt0000003	short	Pauvre Pierrot	Animation,Comedy,Romance
5	tt0000004	short	Un bon bock	Animation,Short
6	tt0000005	short	Blacksmith Scene	Comedy,Short
7	tt0000006	short	Chinese Opium Den	Short
8	tt0000007	short	Corbett and Courtney Before the Kinetograph	Short,Sport
9	tt0000008	short	Edison Kinetoscopic Record of a Sneeze	Documentary,Short
10	tt0000009	movie	Miss Jerry	Romance
11	tt0000010	short	Exiting the Factory	Documentary,Short
12	tt0000011	short	Akrobatisches Potpourri	Documentary,Short
13	tt0000012	short	The Arrival of a Train	Action,Documentary,Short

Abbildung 5.1: Auszug aus dem Beispieldatensatz

Hierbei enthält *title.basics* die grundlegenden Informationen über den Titel. Ergänzt werden diese Informationen durch *title.akas*, welche Namen enthält, unter denen der Titel auch veröffentlicht wurde, *title.episode*, welche bei Serien gefüllt ist mit der Episoden- und Staffelnummer, und *title.ratings*, welche Bewertungen des Titels enthält.

Bei den Personen steht *name.basics* im Mittelpunkt, welche grundlegende Informationen über Personen, welche am Titel beteiligt sind, enthält. Ergänzt werden diese Informationen durch die Crew des Titels, welche in *title.crew* zu finden ist und die Hauptdarstellerbesetzung, welche in *title.principals* gespeichert ist.

5.2 Anwendungsfälle

Anhand des Beispieldatensatzes können unterschiedliche Anwendungsfälle erhoben werden. Hierbei gibt es eine Vielzahl an unterschiedlichen Abfragemöglichkeiten, weshalb wir uns auf die Unterteilung in Kategorien mit je einer exemplarischen Anfrage beschränken.

Kategorie 1: Zugriff über Schlüssel. Diese Kategorie enthält sehr simple Anfragen, bei denen man direkt von einem Schlüsselwert auf ein Datenobjekt zugreift. Da es sich hierbei um vergleichsweise leichte Abfragen handelt, sollten diese möglichst schnell und konsistent sein. Ein Beispiel für so eine Abfrage ist es, einen Film nach dessen ID zu suchen. Hierbei handelt es sich bei der ID des Films um ein eindeutiges Identifikationsmerkmal, welches direkt auf das Film-Objekt verweist.

Kategorie 2: Zugriff auf spärlich besetzte Attribute. Diese Kategorie enthält insbesondere Abfragen auf Attributen, welche oft unbekannte Werte haben oder aus anderen Gründen nur spärlich belegt sind. Es liegt zwar ein direkter Zugriff vor, allerdings wäre es hierbei ineffizient auch die nicht belegten Werte abzuspeichern und diese später wieder durchsuchen zu müssen. Eine mögliche Abfrage ist es, Filme nach dem Verfasser des Buches zu suchen, da heutzutage nicht mehr alle Filme auf Grundlage eines Buches verfilmt werden oder dieser oft einfach nicht bekannt oder vermerkt ist.

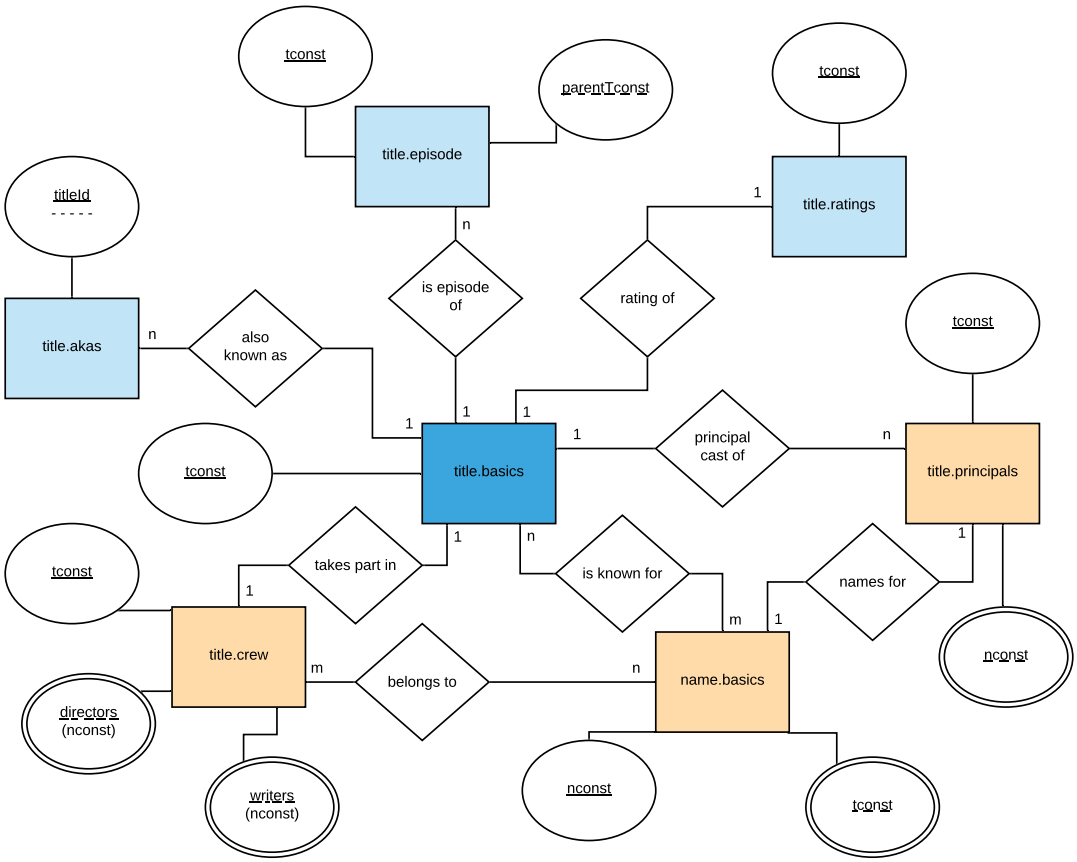

Abbildung 5.2: Beziehungen zwischen den Beispieldatensätzen

Kategorie 3: Zugriff auf stark vernetzte Daten. Bei dieser Kategorie besteht die Schwierigkeit aufgrund von vielen Relationen zwischen den einzelnen Attributen. Bei herkömmlichen relationalen Datenbanken kann diese Relationen nur über viele 'join-Operationen' dargestellt werden, was eine hohe Unleserlichkeit und eine hohe Fehleranfälligkeit der Abfragen mit sich bringt. Zudem ist es oft notwendig, Relationen unterschiedlich stark zu gewichten, da einige wichtigere Merkmale darstellen als andere. Diese unterschiedliche Gewichtung der Relationen ist in relationalen Datenbanken nicht bzw. nur sehr schwer darstellbar. Ein mögliches Beispiel für diese Kategorie ist das Suchen eines Films mit ähnlicher Besetzung. Weil viele Schauspieler in mehreren Filmen mitspielen, entstehen viele Relationen zwischen diesen beiden Entitäten. Hinzu kommt, dass nicht alle Schauspieler oder andere Teammitglieder die gleiche Wichtigkeit für den Nutzer, der die Nachfrage durchführt, haben. So ist es zum Beispiel wichtiger, dass die Hauptdarsteller übereinstimmen, als dass die gleichen Statisten mitspielen.

Kategorie 4: Zugriff auf häufig gemeinsam abgefragte Attribute. Diese letzte Kategorie umfasst Abfragen auf Attributen, welche oftmals gemeinsam abgerufen werden und trotzdem nicht als ein Attribut gespeichert werden sollten. Zusätzlich können diese Attribute komplexere Strukturen haben und zum Beispiel mehrere Werte pro Attribut enthalten. Ein Beispiel für solche Attribute ist

das Suchen nach allen Episoden einer Serie. Hierbei entsprechen die Episoden und die Serien je einzelnen Attributen und werden dennoch sehr häufig gemeinsam aufgerufen. Somit müssen beide immer geladen werden und können, da häufig zusammen abgerufen, gemeinsam geladen werden.

5.3 Datenbanken

Um die beschriebenen Anwendungsfälle möglichst performant und effizient auszuführen, benötigt es passende Speichertechnologien mit dazugehörigen Abfragemöglichkeiten. Hierbei eignen sich insbesondere Datenbanken, welche auf die unterschiedlichen Charakteristiken der Anwendungsfälle abgestimmt sind. Dazu betrachten wir unterschiedliche No-SQL Datenbanken. Da es mittlerweile eine sehr hohe Anzahl an NoSQL-Datenbanken auf dem Markt gibt, werden in diesem Kapitel zuerst die unterschiedlichen Datenbanktypen betrachtet.

5.3.1 Key-value store

Die erste NoSQL-Variante sind die *key-value store Datenbanken*, welche auf einer schemafreien Datenspeicherung basieren [CY15]. Hierbei bestehen die Werte aus einem 'key', welcher ein String ist und einem 'value', welcher dem 'Ist-Werten' entspricht und aus einem primitiven Typen bestehen kann [CY15]. Wie die meisten NoSQL-Datenbanken sind auch die key-value store DBs gut horizontal skalierbar [CY15]. Zusätzlich besitzen sie eine sehr hohe Datenkonsistenz über die einzelnen Knoten [CY15]. Aufgrund der Einfachheit, dieser Art von Datenbanken, haben diese insbesondere ihren Vorteil in der Schnelligkeit [CY15]. Allerdings unterstützen key-value Stores keine Transaktionen, was bei Abfragen, welche nicht atomar sind, zu Inkonsistenzen innerhalb der Daten führen kann [CY15].

Bekannte Beispiele für diese Art von Datenbanken sind Apache HBase[2] und Redis[3].

5.3.2 Document-store

Auch bei den *document-store Datenbanken* werden die Daten schemafrei gespeichert und der Zugriff erfolgt über einen einzigartigen Schlüssel, welcher als String vorliegt [CY15]. Die unterliegende Speicherstruktur besteht aus semi-strukturierten Daten. So basieren Datenbanken auf unterschiedlichen standardisierten Formaten, wie JASON, XML, BISON, PDF oder auch Microsoft Office [CY15]. Durch vorgefertigte APIs und Query-Languages sind dokumentenbasierte Datenbanken sehr effizient im Finden von Werten [CY15].

Bekannte Beispiele für diese Art von Datenbanken sind MongoDB[4] und Apache CouchDB[5].

[2] https://hbase.apache.org/
[3] https://redis.io/
[4] https://www.mongodb.com/de
[5] https://couchdb.apache.org/

5.3.3 Graphdatenbanken

Auch bei den *Graphendatenbanken* handelt es sich um eine schemalose Datenbankform [CY15]. Hierbei werden die in der Datenbank gespeicherten Daten als Graph verwaltet, wobei der Graph Knoten, sowie Kanten mit bestimmten Eigenschaften besitzt [CY15]. Graphdatenbanken sind besonders für Daten geeignet, welche stark miteinander vernetzt sind [CY15]. Aufgrund von unterschiedlichen Graphalgorithmen könne so komplizierte Datenbankabfragen stark vereinfacht werden [CY15].

Bekannte Beispiele für diese Art von Datenbanken sind Neo4J[6], Apache Giraph[7], sowie Google's Pregel[8].

5.3.4 Wide-Column-store

Zuletzt gibt es noch die sogenannten *column-store Datenbanken*. Hier werden alle Werte, welche in einer Spalte vorliegen zusammengefügt/serialisiert [CY15]. Aufgrund des ähnlichen Aufbaus zu zeilen-orientierten Datenbanken sind diese auch vergleichsweise effizient [CY15]. Spalten-orientierte Datenbanken finden insbesondere ihren Vorteil, wenn es um spärlich besetzte Datensätze geht, da sie nur gefüllte Werte speichern und somit auch nur diese durchsuchen müssen [CY15].

Bekannte Beispiele für diese Art von Datenbanken sind Apache Cassandra[9] und Hypertable[10].

[6] https://neo4j.com/
[7] https://giraph.apache.org/
[8] https://www.arangodb.com/docs/stable/graphs-pregel.html
[9] http://cassandra.apache.org/
[10] https://hypertable.org/

6 Implementierung

In diesem Kapitel wird der in Kapitel 4 konzeptionierte Data Lake prototypisch umgesetzt. Hierzu wird zuerst in Abschnitt 6.1 kurz beschrieben, wie ein Hadoop-Cluster aufgesetzt wird. Darauf folgt in Abschnitt 6.2 der zentrale Teil der Datenübertragung von Hadoop zu den unterschiedlichen NoSQL-Datenbanken. Um abschließend in Abschnitt 6.3 noch einmal extra auf die Graphdatenbank Neo4J einzugehen, da diese bei der Umsetzung besondere Ansprüche hat.

6.1 Aufsetzen Hadoop und Datenbanken

Hadoop[1] wurde für die prototypische Implementierung in einem Single-Node-Cluster mit pseudo-verteilten-Modus aufgesetzt. Der pseudo-verteilte-Modus simuliert hierbei drei Knoten, wobei jeder Hadoop-Hintergrundprozess in einem separatem Java-Prozess läuft. Zusätzlich wurden die NoSQL-Datenbanken HBase[2], Neo4J[3], Cassandra[4] und MongoDB[5] aufgesetzt, eine jeweilige Anleitung zum Aufsetzten befindet sich in den Fußnoten.

6.2 Übertragung der Daten von Hadoop zu NoSQL-DBs

Im nächsten Schritt der Implementierung wird die Übertragung der Daten von Hadoop zu den einzelnen NoSQL-Datenbanken betrachtet. Da der Beispieldatensatz aus TSV-Dateien besteht, ist es vergleichsweise einfach das Datenmodell zu interpretieren. So wird bei den Dateien jeweils die oberste Zeile als Header interpretiert, welcher beschreibt, was sich in der jeweiligen Spalte für Werte befinden und somit eine Tabelle bzw. ein Dokument aufgebaut. Hierzu dient die im Rahmen dieser Arbeit entstandene Klasse DBImporter, welche es ermöglicht die unterschiedlichen Datenbanken über Hadoop anzusprechen, und die ebenfalls im Rahmen dieser Arbeit entstandene Klasse Processor, welche abhängig von der jeweiligen Datenbank sich um die eigentliche Datenübertragung der Daten kümmert.

Der DBImporter kann über die Kommandozeile mit `hadoop jar DBImporter.jar DATABASE [FILE..]` aufgerufen werden. Dabei kann für die Variable `DATABASE` ein Wert aus dem Enum der Klasse Processor gewählt werden kann, siehe Listing 6.1 (Zeile 1-13) und `FILE` durch den Namen der entsprechenden zu importierenden TSV-Datei ersetzt werden muss.

[1] https://hadoop.apache.org/docs/r3.0.3/hadoop-project-dist/hadoop-common/SingleCluster.html
[2] https://hbase.apache.org/book.html#quickstart
[3] https://neo4j.com/docs/operations-manual/current/installation/linux/debian/
[4] http://cassandra.apache.org/download/
[5] https://docs.mongodb.com/manual/tutorial/install-mongodb-on-ubuntu/

Bei der Klasse Processor handelt es sich um ein Enum, welches für die unterschiedlichen Datenbanken je eine Datenbankverbindung herstellen kann, siehe Listing 6.1 Zeile 15, diese auch wieder schließen kann (Zeile 18), die Daten übertragen kann (Zeile 24) und je nach Datenbankart eine Tabelle oder ein Dokument erstellen kann (Zeile 21). Da die Implementierung dieser Methoden Datenbank spezifisch sind, werden wir uns vorerst den DBImporter genauer anschauen.

```
1  public enum Processor {
2
3    MONGODB {
4      ...
5    },
6
7    HBASE {
8      ...
9    },
10
11   CASSANDRA {
12     ...
13   };
14
15   public void init() throws Exception {
16   }
17
18   public void close() throws Exception {
19   }
20
21   public void createTableIfNecessary(String name, String[] someHeaders) throws Exception {
22   }
23
24   public void process(List<String[]> lines) throws Exception {
25   }
26 }
```

Listing 6.1: Allgemeiner Aufbau der Klasse Processor

Der DBImporter hat als Hauptaufgabe einen Hadoop-Job zu erstellen, welcher die Daten aus der übergebenen Datei blockweise in die angegebene Datenbank überträgt. Dies geschieht in der run()-Methode, siehe Listing 6.2. Dazu wird zuerst über den Processor eine Datenbankverbindung hergestellt (Zeile 8). Anschließend werden die zu übertragenden Dateien geladen und die Daten eingelesen. Dazu wird ein BufferedReader verendet, welcher diese blockweise speichert und später übergibt (Zeile 12). Um eine Tabelle oder ein Dokument erstellen zu können, wird zuerst die erste Zeile der eingelesenen Datei ausgelesen und gespeichert (Zeile 14). Diese stellen später die sogenannten Header dar und fungieren als Tabellenköpfe bzw. Attributnamen. Im nächsten Schritt wird über den Processor eine Tabelle bzw. ein Dokument angelegt, falls dieses noch nicht existiert (Zeile 15). Anschließend werden die restlichen Daten immer blockweise ausgelesen (Zeile 21 ff.). Damit in der Datei angegebene \N-Werte als richtige null-Werte erkannt werden, werden diese in Zeile 25 und 26 ersetzt. Die vorverarbeiteten Zeilen werden anschließend einem Buffer hinzugefügt (Zeile 29). Im nächsten Schritt wird der vorbereitete Block in die Datenbank übertragen, falls die gewünschte Größe erreicht wurde (Zeile 32). Hierzu werden die gepufferten Daten dem Processor übergeben, damit dieser die Daten in die Datenbank schreiben kann, siehe Zeile 33. Der Buffer wird

nach der Übertragung geleert (Zeile 34) und ein die nächsten Zeilen können eingelesen werden. Letztendlich wird in dem finally-Block immer die Datenbankverbindung wieder über den Processor geschlossen (Zeilen 39-41).

```
1  public int run(String[] args) throws Exception {
2      ...
3      // Find DB Processor by DB name
4      Iterator<String> argumentIterator = Arrays.asList(args).iterator();
5      Processor processor = Processor.valueOf(argumentIterator.next().toUpperCase());
6
7      try {
8        processor.init();
9          ...
10
11         // Open file an read data
12         try (BufferedReader reader = new BufferedReader(...) {
13           ...
14           String[] headers = reader.readLine().split(DELIMITER);
15           processor.createTableIfNecessary(name, headers);
16           ...
17           String line;
18
19           // Read lines blockwise
20           List<String[]> buffer = new ArrayList<String[]>(Processor.BLOCK_SIZE);
21           while ((line = reader.readLine()) != null) {
22             String[] values = line.split(DELIMITER);
23             for (int i = 0; i < values.length; i++) {
24               // Replace \N by null value
25               if (values[i].matches(NULL_VALUE)) {
26                 values[i] = null;
27               }
28             }
29             buffer.add(values);
30
31             // Write block to database
32             if (++rowCount % Processor.BLOCK_SIZE == 0) {
33               processor.process(buffer);
34               buffer.clear();
35             }
36           }
37         }
38       }
39     } finally {
40       processor.close();
41     }
42     return 0;
43   }
```

Listing 6.2: run()-Methode der Klasse DBImporter

Da der DBImporter an mehreren Stellen den Processor aufruft,wird nun zuerst die Implementierung der Klasse Processor betrachtet. Weil die Implementierung abhängig von der jeweiligen Datenbank ist, geschieht dies in einzelnen Unterkapiteln. Der Processor stellt jedoch auch zwei Methoden,

welche nicht explizit zur Datenübertragung verwendet werden und somit in dieser Arbeit nicht genauer vorgestellt werden. Es reicht aus zu wissen, dass die init()-Methode, siehe Listing 6.1 Zeile 15, jeweils eine Verbindung zu der gewünschten Datenbank herstellt und sich um sonstiges Setup kümmert, falls dies benötigt wird und die close()-Methode (Zeile 18) die Datenbankverbindung wieder schließt und falls benötigt Aufräumarbeiten macht.

Die anderen Methoden werden im Folgenden vorgestellt.

6.2.1 HBase

Die erste betrachtete Datenbank ist die key-value store Datenbank HBase. Diese überschreibt, wie alle anderen Datenbanken auch, die beiden Methoden `public void createTableIfNecessary(String name, String[] someHeaders)` und `public void process(List<String[]> lines)` der Klasse Processor.

Hierbei erstellt die createTableIfNecessary()-Methode eine Tabelle anhand der übergebenen Headers, siehe Listing 6.3 Zeile 2, falls diese nicht bereits existiert. Hierzu werden die für HBase nötigen Attribute befüllt (Zeilen 3-7) und es wird geprüft, ob die gewünschte Tabelle neu angelegt werden muss (Zeile 9) oder ob eine bereits vorhandene Tabelle weiter befüllt werden soll (Zeile 8). Anschließend wird die erstellte Tabelle gespeichert, um diese später befüllen zu können (Zeile 11) und die Header werden sich gemerkt, damit diese bei der Übertragung der Daten gesetzt werden können (Zeile 12).

```
1  @Override
2  public void createTableIfNecessary(String name, String[] someHeaders) throws Exception {
3    family1 = Bytes.toBytes("columns");
4    admin = connection.getAdmin();
5    TableName tableName = TableName.valueOf(name);
6    TableDescriptorBuilder tableBuilder = TableDescriptorBuilder.newBuilder(tableName);
7    tableBuilder.setColumnFamily(ColumnFamilyDescriptorBuilder.of(family1));
8    if (!admin.tableExists(tableName)) {
9      admin.createTable(tableBuilder.build());
10   }
11   table = connection.getTable(tableName);
12   headers = Arrays.stream(someHeaders).map(Bytes::toBytes).toArray(byte[][]::new);
13 }
```

Listing 6.3: createTable()-Methode für HBase

Nachdem die Tabelle erstellt wurde, werden dem Processor immer blockweise Daten aus dem DBImporter übergeben. Um diese in die Datenbank zu schreiben, wird die process()-Methode des Processors aufgerufen, siehe Listing 6.4. Diese Methode ist dafür zuständig der Datenbank neue Zeilen hinzuzufügen. Hierzu bekommt die Methode die im Processor zusammengestellten `lines` übergeben (Zeile 2), über welche anschließend iteriert wird (ab Zeile 4). Dabei werden sogenannte `Puts` erstellt, welche später eine neue Zeile in der Datenbank darstellen (Zeile 5). Diese `Puts` werden anschließend mit den gewünschten Daten befüllt (Zeilen 6+7) und der Tabelle hinzugefügt (Zeile 11).

```
1  @Override
2  public void process(List<String[]> lines) throws Exception {
3    List<Put> puts = new ArrayList<>(lines.size());
4    for (String[] values : lines) {
5      Put newRow = new Put(Bytes.toBytes(values[0]));
6      for (int i = 1; i < headers.length; i++) {
7        newRow.addColumn(family1, headers[i], values[i] == null ? null : Bytes.toBytes(values[i
   ]));
8      }
9      puts.add(newRow);
10   }
11   table.put(puts);
12 }
```

Listing 6.4: process()-Methode für HBase

Da man in diesem Fall aus einer tsv-Datei, welche tabellarisch interpretiert werden kann, die Daten in eine tabellarische Datenbank überführt, wird der volle Informationsgehalt der Daten erhalten. Jedoch wird durch die automatische Übertragung der Daten auch kein zusätzlicher Informationsgehalt erschaffen. Somit bleibt die Übertragung von Daten von Hadoop zu einer HBase-Datenbank verlustfrei und der Zugriff auf die Daten kann nun über die Datenbank geschehen.

Um den Zugriff auf die entstandene Datenbank zu zeigen, wird erneut der Anwendungsfall aus Abschnitt 5.2 betrachtet. Hierbei war das Ziel, einen Film über den Filmnamen zu identifizieren. Dabei ist der Filmname anhand seiner tconst eindeutig erkennbar und stellt somit den key dar.

Die Datenbankabfrage sieht wie folgt aus:

```
get 'title.basics', "tt0068646"
```

Hierbei wird ein `get` auf die Datei 'title.basics' ausgeführt und aus dieser wird der Film mit der eindeutigen ID "tt4052886" zurückgegeben.

6.2.2 MongoDB

Als nächstes wird die dokumentorientierte Datenbank MongoDB betrachtet. Auch hier werden erneut die beiden Methoden `public void createTableIfNecessary(String name, String[] someHeaders)` und `public void process(List<String[]> lines)` der Klasse Processor überschrieben.

Im Gegensatz zu HBase wird jedoch bei MongoDB keine Tabelle, sondern ein Dokument angelegt, dies ist in Listing 6.5 zu sehen. Hierzu wird in Zeile 3 ein neues Dokument erstellt und in Zeile 4 ein `ObservableSubscriber`[6] instanziiert. Zusätzlich werden sich auch hier die übergebenen Header gespeichert.

[6] https://github.com/mongodb/mongo-java-driver-reactivestreams/blob/1.0.x/examples/tour/src/main/tour/SubscriberHelpers.java

```
1  @Override
2  public void createTableIfNecessary(String name, String[] someHeaders) throws Exception {
3    collection = mongoClient.getDatabase(DATABASE_NAME).getCollection(name);
4    subscriber = new ObservableSubscriber<>();
5    headers = someHeaders;
6  }
```

Listing 6.5: createTable()-Methode für MongoDB

Um anschließend die Daten der Datenbank hinzuzufügen, kann Listing 6.5 betrachtet werden. Auch hier wird der process()-Methode die vom DBImporter zusammengestellten `lines` übergeben (Zeile 2), über welche iteriert werden und je `line` wird ein neues Dokument erzeugt (Zeile 5). Dieses Dokument wird anschließend in einer Schleife mit Feldname und zugehörigen Wert befüllt (Zeile 7). Die befüllten Dokumente werden anschließend einer Collection in der Datenbank hinzugefügt (Zeile 11). Dies wird durch den von MongoDB gestellten Subscriber überwacht, falls hierbei ein Fehler auftritt, wird eine Exception geworfen (Zeilen 11-15).

```
1  @Override
2  public void process(List<String[]> lines) throws Exception {
3    List<Document> docs = new ArrayList<>(lines.size());
4    for (String[] values : lines) {
5      Document doc = new Document();
6      for (int i = 0; i < headers.length; i++) {
7        doc.append(headers[i], values[i]);
8      }
9      docs.add(doc);
10   }
11     collection.insertMany(docs).subscribe(subscriber);
12   try {
13     subscriber.await();
14   } catch (Throwable e) {
15     throw new Exception(e);
16   }
17 }
```

Listing 6.6: process()-Methode für MongoDB

Auch bei der Übertragung der Daten von einer tsv-Datei in eine dokumentenorientierte Datenbank wird der Informationsgehalt der Daten erhalten. Hierbei ändert sich nur der Aufbau der Daten, da die Werte nicht mehr tabellarisch dargestellt werden, sondern als Felder. Als Anwendungsfall für diesen Datenbanktyp wurde nach einer Staffel einer Serie mit allen zugehörigen Folgen gesucht.

Um den Zusammenhang zwischen title.basics und title.episodes in Erinnerung zu holen, kann Abbildung 6.1 betrachtet werden. Hier sieht man, dass in title.episodes jeweils die tconst eines Filmes enthalten ist und die sogenannte parentTconst, welche auf eine tconst von title.basics referenziert. Dabei stellt das Attribut parentTconst eine ID dar, welche auf die Staffel vereist, in welcher die Episode enthalten ist.

Da der Anwendungsfall beinhaltet, dass die einzelnen Folgen immer zusammen mit der Staffel abgerufen werden, ist es sinnvoll die Folgen innerhalb des Staffel-Dokuments zu speichern. Dies kann wie in Listing 6.7 geschehen.

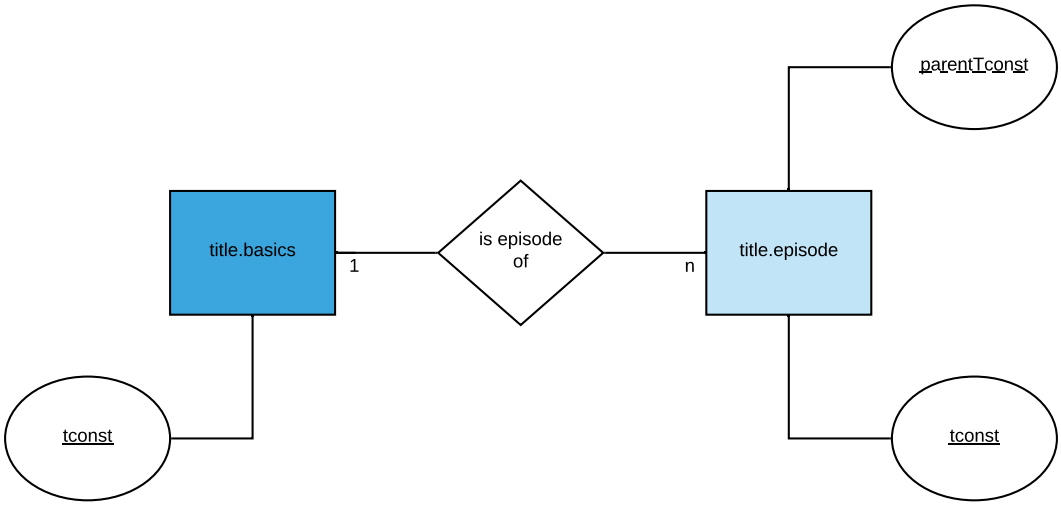

Abbildung 6.1: Beziehungen zwischen title.basics und title.episodes

```
1  db.title.basics.aggregate([
2     {
3        $lookup:
4           {
5              from: "title.episode",
6              localField: "tconst",
7              foreignField: "parentTconst",
8              as: "episode"
9           }
10    },
11    {
12       $match: { episode: {$not: {$size: 0}} }
13    },
14    {
15       $lookup:
16          {
17             from: "title.basics",
18             localField: "episode.tconst",
19             foreignField: "tconst",
20             as: "titles"
21          }
22    }
23 ]).toArray().forEach(function(myTitle){
24    db.title.basics.update({ _id: myTitle._id}, { $set: {titles: myTitle.titles}})
25 });
```

Listing 6.7: Erstellen von Episoden-Unterdokumenten im Staffel-Dokument

Um in einem Staffel-Dokument ein Unterdokument zu erstellen, welches alle Episoden enthält, wird eine `aggregate`-Funktion ausgeführt, siehe Listing 6.7 (Zeiel 1), welche zwei Dokumente aneinanderfügt. Dazu werden zuerst alle Episoden einer Staffel rausgesucht (Zeilen 3-8). Hierzu

wird ein `lookup` gemacht, welches Titel raus sucht, deren parentTconst in title.episodes (Zeilen 5+7) mit der tconst aus title.basics (Zeilen 1+6) übereinstimmt. Das Ergebnis wird temporär in einer Variable namens episode gespeichert (Zeile 8). Im nächsten Schritt werden mit einem `match` alle Titel, welche keine Episoden besitzen, da es zum Beispiel Filme sind oder auch Staffeln selbst, entfernt (Zeile 12). Anschließend werden in dem zweiten `lookup` zu den übriggebliebenen Episoden die gesamten Titeldokumente geladen (Zeilen 15-20). Dies geschieht anhand der tconst. Auch hier wird das Ergebnis temporär in eine Variable namens titles gespeichert. Abschließend werden in den Zeilen 23 und 24 die title.basics-Dokumente upgedatet und die rausgesuchten Episoden als Unterdokument gespeichert.

Um nun auf eine Staffel mit deren Episoden zugreifen zu können, kann folgende Datenbankabfrage verwendet werden:

```
db.title.basics.find({tconst: "tt4052886"})
```

Dabei wird mit `db` die Datenbank angesprochen und mit `title.basics` die gewünschte Collection. Auch hier reicht ein einfaches `find`, welches anhand einer tconst das gesamte Dokument sucht und zurückgibt.

6.2.3 Cassandra

Die letzte mit dem DBImporter umgesetzte Datenbank ist Cassandra, eine Wide-Column-store Datenbank.

Auch diese überschreibt die beiden Methoden `public void createTableIfNecessary(String name, String[] someHeaders)` und `public void process(List<String[]> lines)` der Klasse Processor.

Zuerst wird die createTableIfNecessary()-Methode betrachtet, welche in Listing 6.8 zu sehen ist. Hierzu müssen zuerst beim übergebenen Tabellennamen (Zeile 2) alle Punkte durch Unterstriche ersetzt werden, um Cassandra's Namensschema zu entsprechen (Zeile 3). Anschließend wird, falls nicht bereits vorhanden, ein `Keyspace` erstellt, in welchem die angelegten Tabellen anzufinden sind (Zeilen 5-7). Im darauffolgenden Schritt werden die Tabellen erzeugt (Zeilen 7+8), welchen anhand der Header die Zeilennamen übergeben werden (Zeilen 9+10). In Zeile 12 werden die zusammengestellten Tabellen endgültig angelegt.

```
1  @Override
2  public void createTableIfNecessary(String name, String[] someHeaders) throws Exception {
3    tableName = name.replaceAll("\\.", "_");
4    columns = someHeaders;
5    CreateKeyspace createKs = SchemaBuilder.createKeyspace(DATABASE_NAME).ifNotExists().
   withSimpleStrategy(1);
6    session.execute(createKs.build());
7      CreateTable createTable = SchemaBuilder.createTable(DATABASE_NAME, tableName).ifNotExists
   ()
8        .withPartitionKey(someHeaders[0], DataTypes.UUID);
9    for (int i = 0; i < someHeaders.length; i++) {
10     createTable = createTable.withColumn(someHeaders[i], DataTypes.TEXT);
11   }
12   session.execute(createTable.build());
13 }
```

Listing 6.8: createTable()-Methode für Cassandra

Um die erzeugten Tabellen zu befüllen, wird erneut die process-Methode aufgerufen. Diese ist in Listing 6.9 zu sehen. Auch bei Cassandra wird über die, in Zeile 2, übergebenen `lines` iteriert (ab Zeile 3), um diese einzeln in einer Schleife zu befüllen (ab Zeile 6). Hierbei wird bei Cassandra zuerst geprüft, ob es sich um den Kopf des Inserts handelt, da nun zuerst ein insert-Statement angelegt werden muss. Falls dieses bereits existiert, wird der Wert an das vorhandene insert-Statement an gehangen (Zeilen 6-9). Abschließend wird das insert-Statement in der Datenbank eingetragen (Zeile 11).

```
1  @Override
2  public void process(List<String[]> lines) throws Exception {
3    for (String[] values : lines) {
4      InsertInto insertPrefix = QueryBuilder.insertInto(DATABASE_NAME, tableName);
5      RegularInsert insertQuery = null;
6      for (int i = 0; i < columns.length; i++) {
7        insertQuery = Objects.isNull(insertQuery)
8            ? insertPrefix.value(columns[i], QueryBuilder.literal(values[i]))
9            : insertQuery.value(columns[i], QueryBuilder.literal(values[i]));
10     }
11     session.executeAsync(insertQuery.build());
12   }
13 }
```

Listing 6.9: process()-Methode für Cassandra

Auch bei Cassandra handelt es sich um eine tabellarische Datenbankform. Deswegen ist auch hier davon auszugehen, dass der Informationsgehalt bei der Übertragung bestehen bleibt.

Um einen Zugriff auf die entstandene Datenbank zu demonstrieren, wird auch hier wieder der in Abschnitt 5.2 entworfenen Anwendungsfall umgesetzt. In diesem wurde nach einem Autor eines Filmes gesucht. Hierzu kann die folgende Abfrage verwendet werden:

```
SELECT tconst FROM testdb.title_crew WHERE writers LIKE '%nm1125275%';
```

Bei der Datenbankabfrage wird nach einer Film-ID (tconst) gesucht Dazu wird mit `testdb` der Namespace festgelegt, in welchem gesucht werden soll. Welche Tabelle durchsucht werden soll, wird mit `title_crew` übergeben. Nun wird in der Zeile `writers` die gesucht, welche Attribute die nconst “nm0085156” enthalten. Das `LIKE` ist notwendig, da mehrere Werte eingetragen sein können. Anschließend wird der zugehörige Titel zurückgegeben.

6.3 Übertragung der Daten von Hadoop zu Neo4J

Zuletzt wird die NoSQL-Datenbank Neo4J betrachtet, welche eine Graphendatenbank ist. Diese besteht aus Knoten mit Attributen, welche durch Kanten verbunden werden. Ein Installationsleitfaden kann unter *https://neo4j.com/docs/operations-manual/current/installation/linux/debian/#debian-add-repository* gefunden werden. Da der Aufbau dieser Datenbank etwas komplexer war, wurde diese nicht vollständig mit dem DBImporter umgesetzt. Eine vollständige Umsetzung wurde jedoch per Skript erreicht, weswegen in diesem Kapitel auf die Umsetzung mit dem DBImporter verzichtet wird.

Um den Aufbau einer Graphendatenbank zu verstehen betrachten wir exemplarisch die Umsetzung für die beiden Dateien name.basics.tsv und title.basics.tsv, da diese in unserem späteren Anwendungsfall Verwendung finden werden. In den Dateien geht es um die Verbindung zwischen den Namen, einer am Film teilhabenden Person, und den Titel eines Filmes.

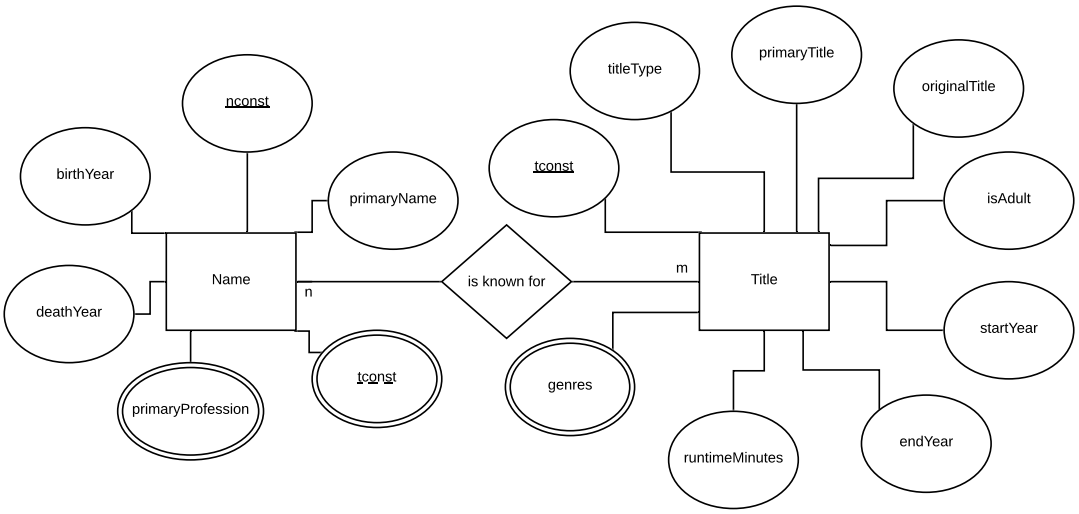

Abbildung 6.2: ER-Digramm zur Visualisierung des Zusammenhangs zwischen Name und Title

Abbildung 6.2 stellt den Zusammenhang zwischen Namen und Titel dar. Hierbei wird eine tsv-Datei, aus unserem Beispieldatensatz, durch eine Entität im ER-Diagramm und ein Header als Attribut dargestellt.

In der Graphdatenbank werden die beiden Entitäten später als Knoten umgesetzt. Dabei besitzt die Entität Name einen eindeutigen Schlüssel namens nconst und ein mehrwertiges Attribut namens tconst, welches auf den eindeutigen Schlüssel von der Entität Title verweist. Durch diesen Verweis zwischen den beiden Entitäten entsteht eine Relation. Zu beachten ist, dass diese Relation namens “isKnownFor” in der Graphdatenbank gleichfalls als eine Kante dargestellt wird. Da die beiden Entitäten eine n:m - Beziehung haben, bedeutet dies, dass mehrere Kanten zwischen Namen-Knoten und Title-Knoten bestehen können. Zusätzlich zu den beiden Knoten Name und Title, werden auch die mehrwertigen Attribute, welche keine Fremdschlüssel sind, als Knoten verstanden.

Um dieses Modell umzusetzen, hatten wir vorerst auch versucht den Aufruf über den Hadoop Job im DBImporter zu realisieren. Da jedoch der Java Driver von Neo4J keine generischen Methoden zum Aufrufen anbietet, sondern ein Datenbankaufruf ein per String zusammengestellter Cypher Querying Language - Befehl ist, war dies sehr unkomfortabel. Zudem gab es keine rein Java basierte Lösung, welche eine saubere Verarbeitung der bestehenden Relationen zwischen den Daten aufarbeiten würde, ohne zuvor künstliche Metadaten zu erschaffen. Darum wurde stattdessen die für Neo4J spezifische skripting Sprache Cypher verwendet. Cypher bringt jedoch den Nachteil mit sich, dass die Daten nicht direkt aus Hadoop ausgelesen werden können, sondern diese zuerst in ein lokales Filesystem geladen werden müssen.

Unabhängig von der Art der Umsetzung erfordert das Erstellen einer Graphdatenbank nähere Kenntnisse über den Datensatz als bei den vorherigen Datenbanken. Dies bringt es mit sich, dass die Umsetzung explizit auf die verschiedenen Dateien angepasst wurde und einen kaum generischen Ansatz mit sich bringt. Gleichzeitig entsteht durch die spezifische Umsetzung ein sehr sauberes Datenmodell, dass die logischen Verknüpfungen zwischen den Daten exakt widerspiegelt.

Im Folgenden wird die Umsetzung der Graphdatenbank anhand der beiden vorgestellten Dateien verdeutlicht. Es wird ein Cypher-Skript verwendet, welches sowohl Knoten als auch Kanten aus den übergebenen Daten erzeugt.

Zuerst werden in Listing 6.10 Knoten für Names (Personen) mit den zugehörigen Attributen erstellt.

```
1 CREATE CONSTRAINT ON (name:Name) ASSERT name.nconst IS UNIQUE;
2 USING PERIODIC COMMIT 10000
3 LOAD CSV WITH HEADERS FROM "file:///name.basics.tsv" as row
4 FIELDTERMINATOR '\t'
5 MERGE (name:Name {nconst: row.nconst}) set name.primaryName=(case row.primaryName when "\\N"
  then null else row.primaryName end), ...;
```

Listing 6.10: Erstellen von Name-Knoten mit Cypher

Hierzu wird in Zeile 1 in Listing 6.10 ein unique Constraint erstellt, um sicherzustellen, dass die Namens-ID nconst einzigartig ist und einen Knoten der Art Name anlegt. Dies hat gleichzeitig den Vorteil, dass durch das ASSERT ein Index auf nconst angelegt wird, was das MERGE aus Zeile 5 der Abbildung merklich beschleunigt. In diesem Beispiel werden die Daten aus der tsv-Datei periodisch in 1000er Schritten übertragen, siehe Zeile 2. Hierzu werden sie aus dem lokalen Speicher mit zugehörigen Headern geladen (Zeile 3) und als Feldtrennungszeichen ein Tab gesetzt (Zeile 4). Zuletzt wird durch das MERGE in Zeile 5 der Knoten letztendlich angelegt, falls dieser einzigartig ist. Hierzu werden die Schlüssel, dargestellt in der geschweiften Klammer, mit den im Graphen vorhandenen Schlüsseln verglichen und geprüft, ob dieser bereits existiert. Falls der Knoten angelegt wird, werden auch die zum Schlüssel zugehörigen Attribute angelegt. Hierzu werden die Werte der Attribute vorerst auf \N Werte geprüft, welche durch echte null-Werte ersetzt werden.

Gleich zu den Name-Knoten werden die Title-Knoten angelegt, siehe Listing 6.11.

```
1 CREATE CONSTRAINT ON (title:Title) ASSERT title.tconst IS UNIQUE;
2 USING PERIODIC COMMIT 10000
3 LOAD CSV WITH HEADERS FROM "file:///title.basics.tsv" as row
4 FIELDTERMINATOR '\t'
5 MERGE (title:Title {tconst: row.tconst}) set title.primaryTitle=(case row.primaryTitle when
  "\\N" then null else row.primaryTitle end), ...;
```

Listing 6.11: Erstellen von Title-Knoten mit Cypher

Um die beiden Knotenarten logisch zu verbinden, wird zuletzt in Listing 6.12 gezeigt, wie Kanten gelegt werden.

```
1 USING PERIODIC COMMIT 10000
2 LOAD CSV WITH HEADERS FROM "file:///name.basics.tsv" as row
3 FIELDTERMINATOR '\t'
4 UNWIND split(row.knownForTitles, ',') AS titleId
5 MATCH (name:Name {nconst: row.nconst})
6 MATCH (title:Title {tconst: titleId})
7 MERGE (name)-[known:IS_KNOWN_FOR]->(title);
```

Listing 6.12: Erstellen von Kanten zwischen Title-Knoten und Name-Knoten mit Cypher

Hierzu wird erneut die verwendete Datei geladen. Da knownForTitles ein mehrwertiges Attribut ist, werden in Zeile 4 die titleIds zuerst anhand des Kommas getrennt. Anschließend werden mit einem MATCH der title anhand seiner tconst und der name anhand seiner nconst gesucht (Zeile 5+6), um anschließend in Zeile 7 eine Kante zwischen title und name zu legen. Diese Kante besitzt den Namen IS_KNOWN_FOR und zeigt von name in Richtung title.

Anschließend wird auch bei der Graphdatenbank Neo4J der in Abschnitt 5.2 entworfene Anwendungsfall umgesetzt. Dieser sucht anhand eines Filmes andere Filme mit ähnlicher Schauspielbesetzung. Die Datenbankabfrage kann wie folgt aussehen, vgl. Listing 7.6. Hierbei wird mit einem `MATCH` nach einem Titel gesucht, an welchem auch Schauspieler aus dem Titel mit der tconst “tt0109686” arbeiten (Zeile 1). Hierzu wird zuerst mit der Variable result ein Title definiert, welcher in einer `WORKED_ON`-Beziehung zu einem `Name` steht. Dieser `Name` muss gleichzeitig in einer `WORKED_ON`-Beziehung zu dem `Title` mit der gewünschten `tconst` stehen. Da in dem Anwendungsfall nur die Schauspielbesetzung berücksichtigt werden soll, wird in Zeile 2 die jeweilige `WORKED_ON`-Beziehung, anhand der Kategorie, auf Schauspieler und Schauspielerinnen beschränkt. Abschließend wird in Zeile 3 das Ergebnis in der `result`-Variable zurückgegeben.

```
1 MATCH (result:Title)<-[work1:WORKED_ON]-(:Name)-[work2:WORKED_ON]->(:Title {tconst: "tt0109686
  "})
2 WHERE work1.category IN ["actor", "actress"] AND work2.category IN ["actor", "actress"]
3 RETURN result;
```

Listing 6.13: Datenbankabfrage nach Filmen mit ähnlicher Schauspielbesetzung

7 Evaluation

Um die Erweiterung des konzeptionierten Data Lakes zu evaluieren, wird in diesem Kapitel der Zugriff auf Daten direkt in Hadoop mit dem Zugriff der Daten innerhalb einer der NoSQL-Datenbanken verglichen. Hierzu werden die Abfragen, für die in Abschnitt 5.2 erarbeiteten Anwendungsfälle, jeweils anhand der Zugriffszeit und der Komplexität der Abfrage an sich verglichen. Dabei wird je die Abfrage direkt aus der NoSQL-Datenbanken mit einer äquivalenten Abfrage direkt auf Hadoop verglichen. Um direkt auf Hadoop zugreifen zu können, wird Apache Spark verwendet [1] mit der Abfragesprache Scala.

Um den Vergleich der zeitlichen Komplexität für eine hohe Anzahl an Datenbankzugriffen zu automatisieren, wurden die Java Klassen `PerformanceTester` und `PerformanceTesterSpark` geschrieben.

Hierzu können die `PerformanceTester`-Klassen per Konsole mit dem Befehl `java -jar PerformanceTester.jar COUNT TESTCASE` bzw. `java -jar PerformanceTesterSpark.jar COUNT TESTCASE` aufgerufen werden. Zuerst wird gewählt, ob ein Testfall direkt auf Hadoop mit Spark ausgeführt werden soll oder auf der Datenbank und jeweils die entsprechende Testklasse ausgewählt. Mit `TESTCASE` kann der auszuführende Testfall ausgewählt werden und mit `COUNT` die Anzahl der Ausführungen für die Datenbankabfrage. Der PerformanceTester, siehe Listing 7.1, führt nun die Abfrage so oft aus, wie in `COUNT` übergeben (Zeile 6) und speichert die Messzeiten für die Ausführungsdauer der Abfrage in einer Datei (Zeile 9). Die Datei ist unter `measure_<count>_<timestamp>.txt` zu finden. Wobei `count` erneut für die Anzahl der Abfragen steht und `timestamp` für einen Zeitstempel, wann die Abfrage gemacht wurde.

```
1  try (TestCase testCase = TestCase.valueOf(argumentIterator.next().toUpperCase());) {
2    System.out.println("Measuring " + testCase.name() + " for " + count + " executions...");
3    testCase.init();
4    long startTime;
5    testCase.executeQuery();
6    for (int i = 0; i < count; i++) {
7      startTime = System.nanoTime();
8      testCase.executeQuery();
9      times[i] = System.nanoTime() - startTime;
10   }
11 }
```

Listing 7.1: Klasse PerformanceTester, welcher automatisch mehrere Datenbankabfragen ausführt

Gleichzeitig berechnet der PerformanceTester das arithmetische Mittel, den Median und den Modus der Abfragezeiten, siehe Listing 7.2. Hierbei ist das arithmetische Mittel (Zeilen 3+23) besonders gut geeignet, um den Durchschnitt mit allen Abfragewerten widerzuspiegeln [Kup11]. Der Median (Zeilen 6-10, 25) ist ebenfalls ein Durchschnitt, jedoch wird dieser nicht so stark durch sogenannte

[1] https://spark.apache.org/

Ausreißer beeinflusst [Kup11]. So kann die erste Abfrage zum Beispiel länger dauern, da diese Zuerst die Datenbankverbindung herstellen muss. Um zu wissen, welche Abfragezeit am häufigsten vorkommt, wird der sogenannte Modus (Zeilen 11-20, 27) verwendet [Kup11]. Dieser ist zum Beispiel gut, um zu wissen, mit welcher Abfragezeit man am häufigsten rechnen kann. Abschließend schreibt der PerformanceTester die Ergebnisse der Berechnungen in eine Datei. Die Berechnungen des PerformanceTesterSpark laufen äquivalent zu denen des PerformanceTesters.

```
1  try (BufferedWriter theWriter = Files.newBufferedWriter(theResultPath)) {
2    if (times.length > 1) {
3      OptionalDouble average = Arrays.stream(times).average();
4      Arrays.sort(times);
5      double median;
6      if (times.length % 2 == 0) {
7        median = ((double) times[times.length / 2] + (double) times[times.length / 2 - 1]) / 2;
8      } else {
9        median = (double) times[times.length / 2];
10     }
11     long maxValue = 0, maxCount = 0;
12
13       for (int i = 0; i < times.length; ++i) {
14       long timerCount = 0;
15       for (int j = 0; j < times.length; ++j) {
16         if (times[j] == times[i]) ++timerCount;
17       }
18       if (timerCount > maxCount) {
19         maxCount = timerCount;
20         maxValue = times[i];
21       }
22     }
23     theWriter.append("Average: " + formatDuration((long) average.getAsDouble()));
24     theWriter.newLine();
25     theWriter.append("Median: " + formatDuration((long) median));
26     theWriter.newLine();
27     theWriter.append("Modus: " + formatDuration(maxValue));
28
29     for (int i = 0; i < times.length; i++) {
30       long l = times[i];
31       if (l < 0) {
32         break;
33       }
34       theWriter.newLine();
35       theWriter.append(formatDuration(l));
36     }
37     theWriter.newLine();
38
39     System.out.println("Result written to \"" + theResultPath.getFileName().toString() +
"\".");
40   }
41 } catch (Exception e) {
42   e.printStackTrace();
43 }
```

Listing 7.2: automatische Berechnungen der Klasse PeformanceTester

Da die gewählten Beispieldatensätze vergleichsweise klein sind für Big Data, wurden für die Performance Messungen jeweils die Speicher begrenzt. Für alle Messungen aus Spark wurde hierzu der Speicher (RAM) des aktuellen Users auf der VM limitiert. Dies geht durch die Eingabe `ulimit -m 524288` in der Konsole. Es wurde das Limit 512MB gewählt, da Spark keine stärkere Begrenzung des Speichers erlaubt.

Um die Datenbankabfragen mt den Abfragen in Hadoop vergleichen zu können, wurde auch deren Speicher limitiert. Bei HBase musste hierzu die Datei `~/hbase/hbase-2.2.0/conf/hbase-env.sh` editiert werden. Hierzu muss die Datenbank gestoppt sein. Nun kann in der Datei der Befehl `export HBASE_HEAPSIZE=512M` auskommentiert werden und die Heapgröße auf 512MB gesetzt werden.

Ähnlich geht die Speicherlimitierung bei Cassandra. Hier muss die Datei `/etc/cassandra/jvm.options` editiert werden. Dazu muss erneut die Datenbank gestoppt sein und die folgenden beiden Befehle auskommentiert und auf 512MB gesetzt werden: `-Xms512M` und `Xmx512M`. Dies setzt die initiale Java Memory und die maximale Java Memory jeweils auf 512 MB.

Auch um die Datenbank Neo4J anzupassen, muss diese gestoppt sein. Jetzt kann die Datei `/etc/neo4j/neo4j.conf` editiert werden. Hier müssen die beiden Einstellungen `dbms.memory.heap.initial_size=512m` und `dbms.memory.heap.max_size=512m` jeweils auf 512 MB gesetzt werden.

Zuletzt wird der Speicher von MongoDB angepasst, da diese keine explizite Möglichkeit anbietet den Speicher zu beschränken wird hier eine sogenannte Control Group verwendet. Hierzu muss MongoDB laufen. Anschließend kann über die Konsole mit dem Befehl `sudo cgcreate -g memory:DBLimitedGroup` eine Control Group erstellt werden. Um deren Speicher anzupassen muss zuerst mit `code` der bisherige (zu hohe) Speicherbedarf aufgeräumt werden und anschließend wird mit `sudo cgclassify -g memory:DBLimitedGroup` `` `pidof mongod` `` dem mongod-Prozess der Control Group ein Memorylimit zugewiesen.

7.1 Datenbankabfragen über Schlüssel

Zuerst werden die beiden Datenbankabfragen, welche über einen Schlüssel auf ein Attribut zugreifen, miteinander verglichen. Ein möglicher Anwendungsfall war das Suchen einen Films nach dessen tconst. Hierzu wird die Datenbankabfrage aus HBase, welche

```
get 'title.basics', "tt0068646"
```

war, mit der äquivalenten Abfrage für Spark verglichen, siehe Listing 7.3

```
1 spark.sqlContext.read.format("csv").option("delimiter", "\t")
2   .option("header", "true").load("movies/title.basics.tsv")
3   .filter("tconst = 'tt0068646'");
```

Listing 7.3: Äquivalente Abfrage für den HBase Anwendungsfall in Spark

Bei der Abfrage muss bei Spark erst genau definiert werden, welches Dateiformat vorliegt. Da Spark nur einen Import für CSV-Dateien bereit stellt, muss das Trennzeichen explizit auf Tab gesetzt werden, siehe Listing 7.3 Zeile 1. Anschließend muss Spark übergeben werden, was als Attributname verwendet werden soll, wie auch bei dem Import in die Datenbanken werden hier die Header verwendet (Zeile 2). Nun kann die Datei anhand des Dateipfads geladen werden (Zeile 2) und nach dem Wert “tt0068646” in der Spalte tconst durchfiltert werden (Zeile 3).

Da die Angabe des Dateiformats und des Speicherorts bei HBase implizit geschieht, ist die Anfrage geringfügig kürzer und somit leichter zu lesen.

Um die Abfragezeiten der beiden Abfragen zu vergleichen, kann Abbildung 7.1 betrachtet werden. Diese stellt die Abfragedauer von 20 Abfragen dar. Auf der x-Achse wird die Abfragedauer in Sekunden dargestellt, wobei diese logarithmisch ist. Die blaue Linie spiegelt die Abfragedauern direkt auf Hadoop über Spark wieder und die orangene Linie die der Abfrage in HBase. Hierbei ist zu erkennen, dass die Abfragen über HBase zwischen 2ms und 8ms schwangen. Dies liegt bedeutend unter den Abfragezeiten über Spark, welche alle zwischen 27s und 29s liegen. Während bei HBase sowohl das Mittel als auch der Median bei 4,2ms liegen, liegt das arithmetische Mittel bei HBase bei 27,6s und der Median bei 27,5s. Dies zeigt, dass die Abfragen kaum durch Ausreißer beeinflusst wurden und somit die Abfrage über Spark ungefähr mit dem Faktor 6500 langsamer sind. Dieser hohe Unterschied kann damit erklärt werden, dass HBase eine key-value store Datenbank ist und für Anfragen, welche über einen Schlüssel auf ein Attribut zugreifen, optimiert wurde.

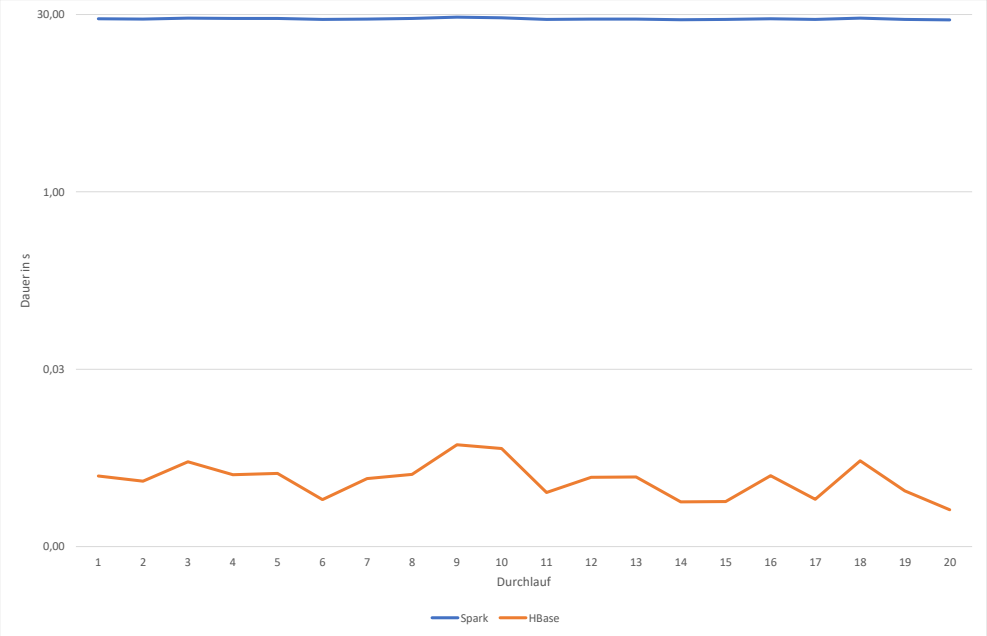

Abbildung 7.1: Vergleich der Abfragedauern von Spark und HBase

Insgesamt bewährt sich somit das Laden der Dateien in eine key-value-store Datenbank, falls diese anschließend für einen entsprechenden Anwendungsfall verwendet werden.

7.2 Datenbankabfragen für häufig gemeinsam abgefragte Attribute

Anschließend werden die Datenbankabfragen für Attribute, welche oft gemeinsam abgefragt werden, verglichen. Ein möglicher Anwendungsfall war es eine Staffel mit all ihren Episoden zu laden. Diese Abfrage konnte in MongoDB wie folgt gemacht werden:

```
db.title.basics.find({tconst: "tt4052886"}).
```

Um eine äquivalente Abfrage direkt auf Hadoop in Spark zu erzeugen, werden die folgenden Abfragen benötigt, siehe Listing 7.4. Hierzu wird zuerst die Datei title.basics geladen (Zeile 1) und diese nach der gesuchten Staffel anhand der tconst gesucht (Zeile 2). Um nun anschließend die zugehörigen Episoden zu finden, wird die Datei title.episode geladen und anhand der parentTconst, welche gleich mit der Staffel tconst sein muss, durch filtert (Zeile 3). Zuletzt werden die raus gefilterten Episoden an die gesuchte Staffel gejoint (Zeile 4).

```
1 var titles = spark.sqlContext.read.format("csv").option("delimiter", "\t").option("header", "
  true").load("movies/title.basics.tsv");
2 titles.filter("tconst = 'tt4052886'");
3 var episodes = spark.sqlContext.read.format("csv").option("delimiter", "\t").option("header",
  "true").load("movies/title.episode.tsv").filter("parentTconst = 'tt4052886'").select("tconst")
  ;
4 episodes.join(titles).where(episodes("tconst") === titles("tconst"));
```

Listing 7.4: Äquivalente Abfrage für den MongoDB Anwendungsfall in Spark

Bereits bei der Datenbankabfragekomplexität sieht man, dass die Abfrage in MongoDB bedeutend kürzer und somit leichter lesbar ist.

Wird nun die Abfragedauer der beiden Abfragen verglichen, sieht man, dass die Abfrage in MongoDB auch um einiges performanter ist. Um die beiden Abfragen zu vergleichen, kann Abbildung 7.2 betrachtet werden. Auch hier wurden wieder 20 Abfragen in MongoDB (orangene Linie) mit 20 äquivalenten Abfragen in Spark verglichen (blaue Linie). In dem Diagramm ist erneut auf der x-Achse die Abfragedauer in Sekunden, welche logarithmisch dargestellt wird. Auch wenn die Abfragezeiten in MongoDB leicht schwanken, haben sie ein arithmetisches Mittel von 0,9ms, was bedeutend unter dem Mittel von 1min 25s in Spark liegt. Hierbei kann die schnellere Abfrage in MongoDB damit erklärt werden, dass alle Episoden innerhalb des Staffel-Dokuments gespeichert sind und man somit nur eine Abfrage benötigt. Gleichzeitig müssen bei der Abfrage über Spark die Episoden mit der Staffel zusammen gejoint werden und Joins sind bei Datenbankabfragen sehr aufwändige Operationen. Somit wird die Dauer durch die mehreren Einzelabfragen, den Join und der Tatsache, dass Hadoop nicht so gut optimiert ist, wie die Datenbank MongoDB, bedeutend langsamer.

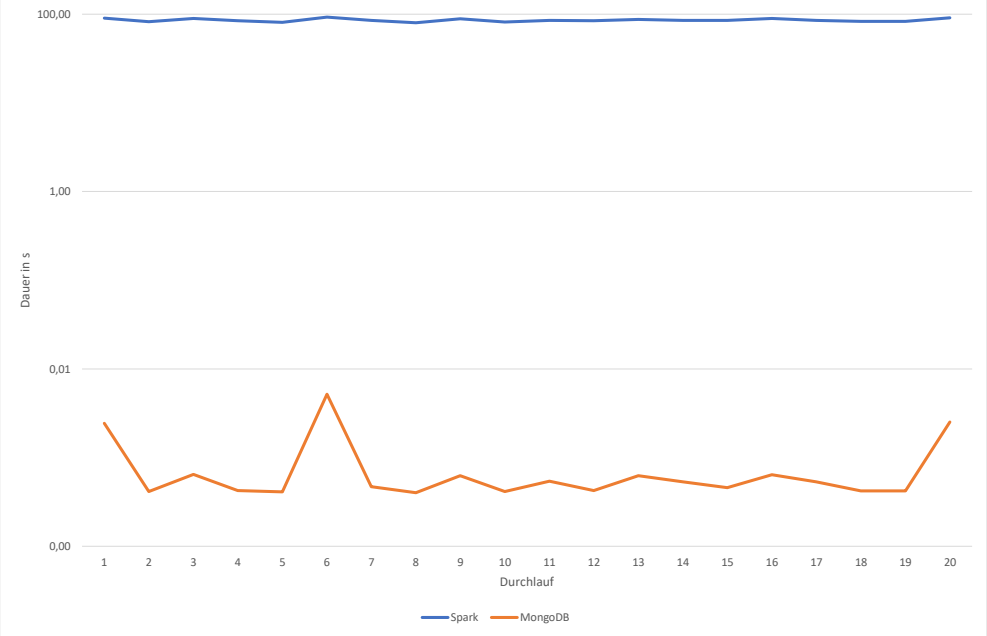

Abbildung 7.2: Vergleich der Abfragedauern von Spark und MongoDB

Somit bewährt sich das Laden von zusammengehörigen Attributen in ein Dokument, sowohl was die Abfragekomplexität als auch was die Abfragezeiten angeht.

7.3 Datenbankabfragen auf spärlich besetzten Attributen

Auch bei den Datenbankabfragen auf spärlich besetzte Attribute wird die zuerst die Abfragezeit von Abfragen auf Cassandra mit denen über Spark direkt in Hadoop verglichen. Als mögliche Abfrage wurde nach dem Autor eines Films gesucht. In Cassandra ging die Abfrage wie folgt:
`SELECT tconst FROM testdb.title_crew WHERE writers LIKE '%nm1125275%';.`

Eine äquivalente Abfrage in Spark, kann wie in Listing 7.5 aussehen. Hier wird erneut zuerst die zu durchsuchende Datei geöffnet und das Dateiformat auf TSV gesetzt (Zeilen 1+2). Anschließend wird die Datei nach Autoren mit der gewünschten nconst durchsucht und der zugehörige Film zurückgegeben.

```
1 spark.sqlContext.read.format("csv").option("delimiter", "\t")
2   .option("header", "true").load("movies/title.crew.tsv")
3   .filter("writers like '%nm1125275%'").select("tconst");
```

Listing 7.5: Äquivalente Abfrage für den Cassandra Anwendungsfall in Spark

Anschließend werden auch hier die Abfragezeiten verglichen, hierzu kann Abbildung 7.3 betrachtet werden. Auch bei diesem Diagramm stellt die x-Achse die Abfragezeit in Sekunden dar und ist logarithmisch. Es werden erneut 20 Durchläufe sowohl in Cassandra als auch in Spark gemacht. Hierbei belaufen sich die Zeiten für die Abfrage auf Cassandra zwischen 18,5ms und 66,6ms und die Abfragezeiten in Spark zwischen 13s und 15s. Somit ist die Abfrage über Spark erneut bedeutend langsamer als in der NoSQL-Datenbank Cassandra, da hier die Attribute, welche null sind, gar nicht erst gespeichert werden und somit auch nicht durchsucht werden müssen.

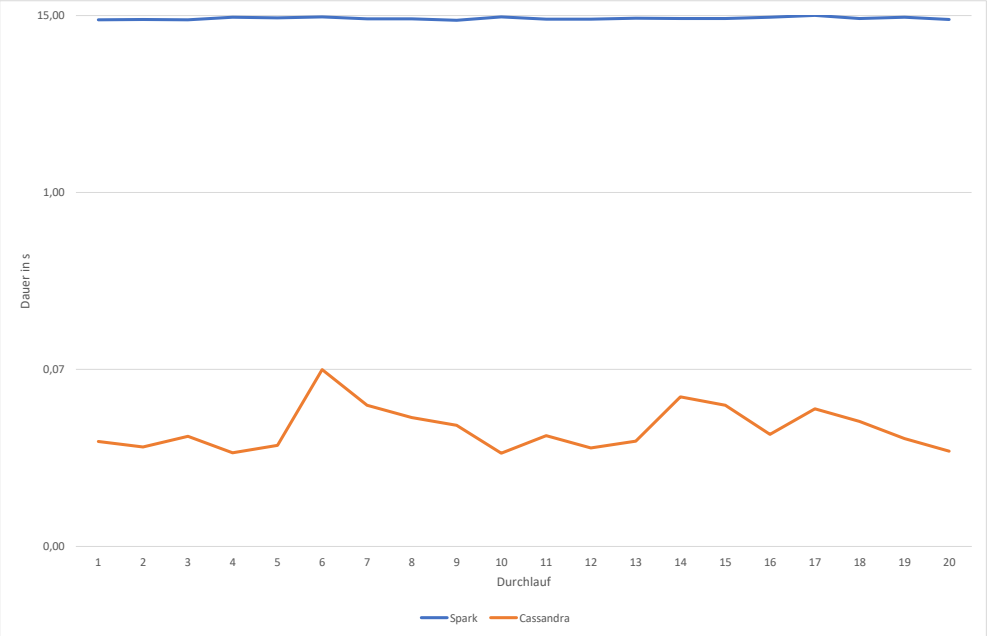

Abbildung 7.3: Vergleich der Abfragedauern von Spark und Cassandra

7.4 Datenbankabfragen auf stark vernetzten Daten

Zuletzt werden auch die Abfragen auf stark vernetzten Daten miteinander verglichen. Als Beispielanwendungsfall wurden Filme mit ähnlicher Schauspielbesetzung gesucht. Die Abfrage für Neo4J ist erneut in Listing 7.6 zu sehen.

```
1 MATCH (result:Title)<-[work1:WORKED_ON]-(:Name)-[work2:WORKED_ON]->(:Title {tconst: "tt0109686
  "})
2 WHERE work1.category IN ["actor", "actress"] AND work2.category IN ["actor", "actress"]
3 RETURN result;
```

Listing 7.6: Datenbankabfrage nach Filmen mit ähnlicher Schauspielbesetzung

In Listing 7.7 ist die äquivalente Abfrage für Spark zu sehen. Hierbei müssen mehrere Einzelschritte gemacht werden. Zuerst wurde die Datei title.principals geladen und nach Schauspieler/innen durch filtert (Zeile 1). Anschließend wurden die übriggebliebenen Werte anhand der tconst für den gesuchten Film durch filtert und die Schauspieler/innen, welche an dem Film teilgenommen haben, zurück gegeben (Zeile 2). Nun werden alle Filme aus der Datei title.basics geladen und in `movies` gespeichert (Zeile 3). Zuletzt werden die jeweiligen Schauspieler/innen mit den zugehörigen anderen Filmen, an denen diese gleichfalls teilgenommen haben, aneinander gejoint (Zeile 4).

```
1 var participals = spark.sqlContext.read.format("csv").option("delimiter", "\t").option("header
  ", "true").load("movies/title.principals.tsv").filter("category in ('actor', 'actress')").as("
  participals");
2 var actors = participals.filter("tconst = 'tt0109686'").select("nconst").as("actors");
3 var movies = spark.sqlContext.read.format("csv").option("delimiter", "\t").option("header", "
  true").load("movies/title.basics.tsv").as("movies");
4 var result = actors.join(participals, $"actors.nconst" === $"participals.nconst").join(movies,
   $"participals.tconst" === $"movies.tconst");
```

Listing 7.7: Äquivalente Abfrage für den Neo4J Anwendungsfall in Spark

Um die Abfragedauern der beiden Abfragen zu vergleichen kann Abbildung 7.4 betrachtet werden. Hier wird erneut die Abfrage über Spark mit einer blauen und die Datenbankabfrage in Neo4J mit einer orangenen Linie dargestellt. Auch bei diesem Diagramm stellt die x-Achse die Abfragedauer in Sekunden dar, welche logarithmisch ist und die y-Achse die Anzahl der Testdurchläufe. Da die Abfragezeiten in Neo4J zwischen 23ms und 218ms schwanken, mit einem Mittel von 64ms, sind diese bedeutend schneller als über Spark. In Spark hat die äquivalente Abfrage ein arithmetisches Mittel von 4min 26s und schwankt zwischen 4min 20s und 4min 39s. Auch bei dieser Abfrage müssen gleich mehrere join-Operationen in Spark gemacht werden, welche die einzelnen Abfragen zusammen führt. In Neo4J wird die gleiche Abfrage mit nur einem einzelnen Datenbankaufruf gemacht und gleichzeitig wurde diese für Abfragen über Relationen optimiert.

Somit ist sowohl die Datenbankabfrage für Neo4J kürzer und leichter lesbar, als auch die Abfragedauer bedeutend geringer. Allerdings muss bei Neo4J berücksichtigt werden, dass das Einlesen der Werte in die Datenbank einen höheren Aufwand als bei den anderen verglichenen Datenbanken mit sich bringt. Jedoch entsteht dadurch auch ein sehr sauberes Datenmodell.

Insgesamt muss je nach späterem Anwendungsfall und Häufigkeit der Verwendung, von der eingepflegten Datenbank, entschieden werden, ob sich die Übertragung der Daten in eine Graphdatenbank bezahlt macht.

Insgesamt lohnt sich somit die Erweiterung eines Data Lakes mit weiteren Technologien, wie NoSQL-Datenbanken. Da SQL-Datenbanken eine schemafreie Speicherstruktur haben, ermöglichen diese eine kostengünstige Möglichkeit der Übertragung von Daten aus Hadoop. Dies wiederum ermöglicht schnelle und einfach lesbare Abfragen auch für komplexere Anwendungsfälle.

Jedoch muss je nach Anwendungsfall die Wahl der Datenbank bewusst geschehen, da zum Beispiel die Übertragung von Daten in eine Graphendatenbank auch einiges an Aufwand mit sich bringt. Gleichzeitig wird die Übertragung der Daten durch deren Metadaten beeinflusst. Sind so zum Beispiel ein oder auch mehrere Werte aus einer Datei nicht eindeutig interpretierbar, wird es schwer diese automatisch in eine Datenbank zu übertragen. Somit sollte bei einem hohen manuellen Aufbereitungsaufwand nicht die Möglichkeit der Übertragung in ein Data Warehouse vergessen werden.

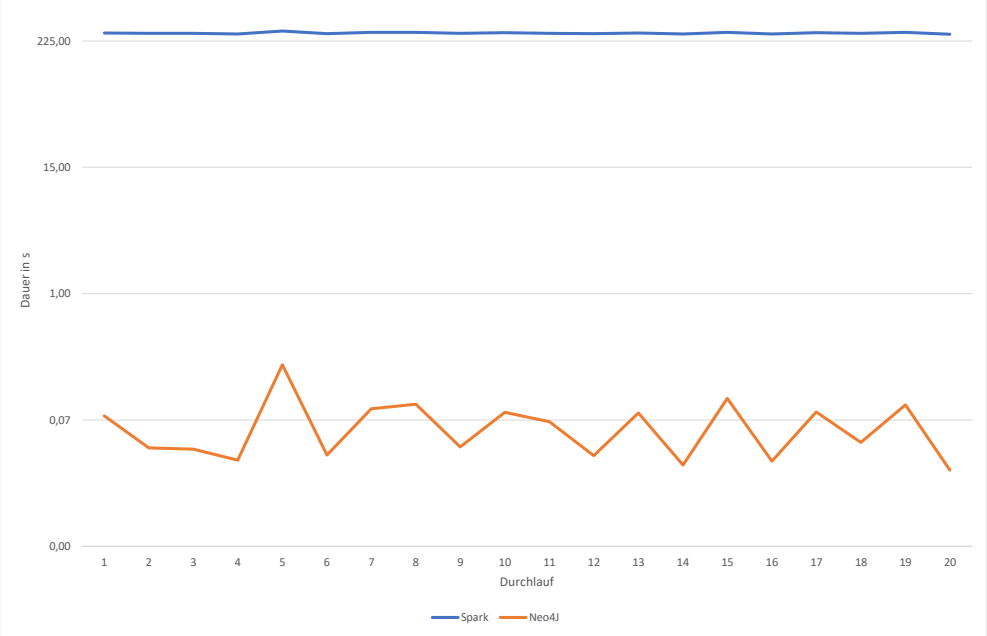

Abbildung 7.4: Vergleich der Abfragedauern von Spark und Neo4J

8 Zusammenfassung und Ausblick

In dieser Arbeit wurde ein Konzept eines Data Lakes entworfen, um mit einem großen Volumen an strukturierten, halb-strukturierten und unstrukturierten Daten arbeiten zu können.

Hierzu wurde zuerst das Konzept Data Lake und ein Zonenmodell vorgestellt. Mithilfe einer verwandten Arbeit, wurde ein bestehendes Konzept untersucht, welches noch einige Nachteile aufwies. Damit diese Nachteile ausgeglichen wurden, wurde ein eigenes Konzept eines Data Lakes entworfen. Das neue Konzept beinhaltet dabei die Aufteilung in anwendungsfallabhängige und anwendungsfallunabhängige Datenhaltung. So werden anwendungsfallunabhängige Daten zuerst in einem Hadoop-Cluster gespeichert. Dies hat den Vorteil einer Speicherung, ohne dass Daten zuvor aufbereitet oder transformiert werden müssen. Wenn Daten nun zur Verwendung bereitgestellt werden sollen, geschieht dies anhand des späteren Anwendungsfalls. So werden die Daten in ein für den Anwendungsfall passendes System übertragen. Als mögliche Systeme wurden die aus der verwandten Arbeit entnommenen Enterprise Data Warehouses und SQL-Datenbanken übernommen. Zu diesen Speichermöglichkeiten kamen unterschiedliche NoSQL-Datenbanken, welche den Vorteil einer schemalosen Speicherung und einer guten horizontalen Skalierung mit sich bringen und somit gut für Big Data geeignet sind.

Um das entworfene Konzept prototypisch umsetzten zu können, wurde ein Beispieldatensatz gesucht, anhand dessen unterschiedliche Anwendungsfälle erarbeitet wurden. Hier wurde der Datensatz der *Internet Movie Database*, kurz IMDb, genommen, da die Relationen zwischen den unterschiedlichen Datensätzen somit leicht verständlich und nachvollziehbar sind. Weil das entworfenen Konzept auch darauf beruht unterschiedliche Datenbanken zu verwenden, wurden einige Datenbankarten vorgestellt. Dabei lag der Fokus auf den unterschiedlichen NoSQL-Datenbanken, welche in unterschiedliche Gruppen aufgeteilt werden können. Jetzt konnten den erarbeiteten Anwendungsfällen ein passendes Speichersystem zugeordnet werden.

Anschließend wurde der Data Lake prototypisch umgesetzt. Hierzu wurde die Datenübertragung aus einem Hadoop-Cluster in die unterschiedlichen Datenbanken implementiert und entsprechend des Anwendungsfalls eine Datenbankabfrage geschrieben.

Zuletzt wurde die Datenhaltung in den unterschiedlichen NoSQL-Datenbanken gegen die Haltung in einem HDFS evaluiert. Dazu wurden mit Spark Abfragen direkt auf Daten in Hadoop geschrieben und diese gegen die Datenbankabfragen evaluiert. Die Abfragen wurden hierzu anhand der Komplexität der Abfrage und anhand der Zeitkomplexität verglichen. Dies zeigt das die Datenübertragung bei entsprechendem Anwendungsfall sowohl leichtere, als auch schnellere Abfragen ermöglicht.

Ausblick

Da die Erweiterung des Data Lakes mit unterschiedlichen NoSQL-Datenbanken erfolgreich war, kann darüber nachgedacht werden, ob es noch andere sinnvolle Datenspeicher gibt, mit denen man den Data Lake erweitern kann. So wird es auch zukünftig immer wieder neue Technologien geben, welche den Umgang mit großen Datensätzen ermöglichen und sich somit als Erweiterung für den Data Lake eignen. Auch sollte über eine sinnvolle Übertragung der Daten innerhalb der Zonen nachgedacht werden. So können einmal erarbeitete Verarbeitungsschritte auch für andere Anwendungsfälle, welche die Speicherung in einer anderen Technologie voraussetzten, wiederverwendet werden. Dies würde die Datenverarbeitungskosten möglichst gering halten und den Data Lake effizienter machen.

Literaturverzeichnis

[BG13] A. Bauer, H. Günzel. *Data-Warehouse-Systeme: Architektur, Entwicklung, Anwendung*. dpunkt. verlag, 2013 (zitiert auf S. 13).

[CY15] R. Casado, M. Younas. „Emerging trends and technologies in big data processing“. In: *Concurrency and Computation: Practice and Experience* 27.8 (2015), S. 2078–2091 (zitiert auf S. 30, 31).

[GM19] C. Giebler, B. Mitschang. „Leveraging the Data Lake“. In: (2019) (zitiert auf S. 14).

[Gor16] A. Gorelik. *The Enterprise Big Data Lake*. Hrsg. von T. McGovern. O'Reilly Media, Inc., 2016 (zitiert auf S. 17, 18).

[GRR19] J. Gantz, D. Reinsel, J. Rydning. *The Digitization of the World From Edge to Core*. 2019 (zitiert auf S. 13).

[HGQ16] R. Hai, S. Geisler, C. Quix. „Constance: An intelligent data lake system“. In: *Proceedings of the 2016 International Conference on Management of Data*. ACM. 2016, S. 2097–2100 (zitiert auf S. 17).

[Jac09] A. Jacobs. „The pathologies of big data“. In: *Communications of the ACM* 52.8 (2009), S. 36–44 (zitiert auf S. 13).

[Kro18] M. Kroker. *2,5 Trillionen Bytes Daten pro Tag erzeugt – Speicherkapazität von 36 Millionen iPads*. 2018. URL: https://blog.wiwo.de/look-at-it/2018/03/21/25-trillionen-bytes-daten-pro-tag-erzeugt-speicherkapazitaet-von-36-millionen-ipads/ (zitiert auf S. 13).

[Kup11] H. Kuper. „Quantitative Auswertungsverfahren“. In: *Empirische Bildungsforschung*. Springer, 2011, S. 121–129 (zitiert auf S. 45, 46).

[Mad15] M. Madsen. „How to Build an enterprise data lake: important considerations before jumping in“. In: *Third Nature Inc* (2015) (zitiert auf S. 13, 17).

[Mat17] C. Mathis. „Data lakes“. In: *Datenbank-Spektrum* 17.3 (2017), S. 289–293 (zitiert auf S. 13).

[PWD17] P. Patel, G. Wood, A. Diaz. „Data Lake Governance Best Practices“. In: *The DZone Guide to Big Data - Data Science & Advanced Analytics* 4 (2017), S. 6–7 (zitiert auf S. 17, 18).

[Sha18] B. Sharma. *Architecting Data Lakes - Data Management Architectures for Advanced Business Use Cases*. 2. Aufl. O'Reilly Media, Inc., 2018 (zitiert auf S. 17).

[TD16] P. Tyagi, H. Demirkan. „Data lakes: The biggest big data challenges“. In: *Analytics Magazine* (2016), S. 56 (zitiert auf S. 13).

[WMS94] F. P. Williams III, M. D. McShane, D. Sechrest. „Barriers to effective performance review: The seduction of raw data“. In: *Public Administration Review* (1994), S. 537–542 (zitiert auf S. 13).

[Zik15] P. Zikopoulos. *Big data beyond the hype: A guide to conversations for today's data center*. McGraw-Hill Education, 2015 (zitiert auf S. 17, 21).

Alle URLs wurden zuletzt am 16. 02. 2020 geprüft.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift