MARTIN-LUTHER
UNIVERSITÄT
HALLE-WITTENBERG

Bachelorarbeit

# Deklaratives Textmining mit Spark

Dmitry Trofimov
Matrikelnummer: 213222170

Wissenschaftlicher Betreuer und Gutachter: Dr. Alexander Hinneburg
Zweitgutachter: Prof. Dr. Stefan Brass

Abgabetermin: 03.10.2016

# Inhaltsverzeichnis

# 1 Einleitung

Die Menge an Informationen, die heutzutage tagtäglich veröffentlicht werden, ist enorm – allein die Anzahl an Blogposts, die an einem Tag ins Internet gestellt werden, beläuft sich auf mehrere Millionen (etwa 2 345 000 Blog-Einträge wurden am 03.08.2016 bis 12:37 erstellt [1]). Die Werkzeuge, welche man als Nutzer verwendet, um darin etwas zu finden, beschränken sich im Groben auf die Suche und die Verweise („Links“) auf die suchbezogenen Inhalte [2, S. 77]. Doch die Informationen, in denen gesucht wird, sind zum größten Teil unstrukturiert, was negative Auswirkungen auf die Suchqualität haben kann.

Man stelle sich vor, es gäbe eine Themenübersicht, innerhalb deren flexibel navigiert werden könnte, und man wäre in der Lage, die interessanten und/oder relevanten Themenbereiche selbst auswählen [2, S. 77]. Das Gewünschte wäre dann vermutlich leichter zu finden, wenn man die Suche auf einen vom Nutzer selbst festgelegten Kontext einschränkt. Besonders für die wissenschaftliche Arbeit wäre beispielsweise die Nachvollziehbarkeit der Verbindungen zwischen den Themen wünschenswert.

Der Sinn von einem solchen Konzept ist plausibel; doch um es zu bewerkstelligen, brächte man eine Instanz, die den Wörtern in allen verfügbaren Textquellen Themen zuweist – eine Aufgabe, die von menschlicher Hand unmöglich zu bewältigen wäre. Aus diesem Grund ist die probabilistische Themenmodellierung (eng. *probabilistic topic modelling*) entstanden – ein Begriff für algorithmische Ansätze zum Vervollständigen großer Datensätze mit thematischen Informationen. Die Daten werden analysiert, um Aussagen darüber zu treffen, welche Themen an welchen Stellen ans Licht kommen, wie sie miteinander in Verbindung stehen und wie sie sich im Laufe der Zeit ändern. So könnten textuelle Archive oder aber auch andere Ressourcen wie Bilder, genetische Daten oder gar soziale Netzwerke automatisiert organisiert werden [2, S. 77]. Für diesen Zweck sind sogenannte *Topic-Model-Browser* gebräuchlich – eine Software, die für Themenmodelle eine grafische Oberfläche bietet. Hierzu zählt auch TopicExplorer, entwickelt an der MLU Halle-Wittenberg. Es wird in dieser Bachelorarbeit darum gehen, die Implementierung der Vorberechnungen für diese Software durch den Austausch von ihrem Back-End zu beschleunigen.

## 1.1 Zielstellung

**Ist-Zustand** TopicExplorer ist einer der wenigen Topic-Model-Browser, die nach dem Paradigma der *deklarativen* Programmierung entwickelt wurden: der Programmierer muss bei diesem Ansatz nur formulieren, *was* er berechnet haben möchte, und nicht *wie* [3]; der Rechenweg wird bei vom System automatisch ermittelt [4, S. 181].

Warum der deklarative Ansatz? Zum einen wurde er gewählt, um der Implementierung möglichst viel Raum zur Skalierung zu bieten und die Implementierung (das Back-End) mit wenig Aufwand austauschen zu können [5, S. 1136]. Der Programmierer formuliert in SQL eine Beschreibung des Problems, und die Ebene darunter kümmert sich um die Berechnung der Lösung zu diesem Problem. Zum anderen will man sich auch die Möglichkeit einräumen, von der Übersetzung grafischer Darstellungen probabilistischer Themenmodelle in Entity-Relationship-Modelle zu profitieren – diese bildet eine Brücke zwischen den Themenmodellen und den Datenbanken, in denen die themenmodellbezogenen Daten gespeichert werden.

Ursprüngliche Umsetzung der TopicExplorer-Vorberechnung mit MySQL-Back-End ließ viele Wünsche offen, hauptsächlich in Hinsicht auf eine unangemessen lange Rechenzeit. Die Rechnerressourcen (CPU und RAM) wurden bei der Vorberechnung in der MySQL-Umgebung kaum ausgenutzt: eine der Restriktionen von MySQL ist die Regel „ein Prozessorkern pro Abfrage“ [6]. Das mag bei vielen Abfragen mit kleinem Rechenaufwand vielleicht noch von Vorteil sein, trifft auf unsere Anwendungsfall mit großen Daten jedoch nicht zu, und war die Hauptursache für Ineffizienz der Vorberechnung.

Im TopicExplorer werden nur wenige Abfragen getätigt, die aber sehr viele Daten auf einmal verarbeiten. Das liegt daran, dass wir mit großen Textmengen auf Wort-Ebene arbeiten, d.h. wir behandelt mit unseren Abfragen fast jedes Wort mit spezieller Relevanz (anders gesagt, jedes Wort, das kein Funktionswort [7] ist). Bei sehr großen Datensätzen erweist sich MySQL in seltenen, nicht reproduzierbaren Fällen sogar als instabil.

**Soll-Zustand** Aus dieser Problematik ruht die Zielstellung dieser Bachelorarbeit: unter Beibehaltung der Deklarativität der Implementierung einen signifikanten Zeitgewinn bei den Vorberechnungen für TopicExplorer durch eine bessere Hardwareauslastung zu verschaffen.

Dieses Ziel sollten über die Umstellung von MySQL auf **Apache Spark** erreicht werden. Spark ist ein Framework für *cluster computing* mit einer SQL-Schnittstelle, womit man sich gegenüber MySQL eine effizientere Rechnerressourcenauslastung, mehr Stabilität und unter Umständen die Möglichkeit der Implementierung neuer Funktionen (wie „Framing“) verspricht.

Der Umstellung liegt folgende Idee zugrunde: der MySQL-Umsetzung wird die Problemformulierung entnommen, um sie auf einem Spark-Cluster ausrechnen zu lassen. Dadurch ändert sich theoretisch nur das Back-End-System, welches direkt für die Kontrolle der Berechnung sorgt. Das Ergebnis der Berechnung bleibt bei gleicher Problemformulierung unverändert, und die Deklarativität der Umsetzung bleibt erhalten. Im Soll-Zustand mit Spark ist gegenüber der MySQL-basierten Version mit schnellerer Ergebnisermittlung sowie der gewonnenen Stabilität des Berechnungsverlaufs zu rechnen.

## 1.2 Herausforderungen

**Kernprobleme bei der Umsetzung** Die Tatsache, dass man im Modul Spark SQL (in der Spark-Version 1.6.1) nicht alle SQL-Abfragen ausführen kann, stellt eine Herausforderung dar. Solche Teile der Topic-Explorer-Vorberechnung mussten deshalb in äquivalente SQL-Statements umgeschrieben werden. Die Komponente Spark SQL in Spark 1.6.1 kann keine Abfragen mit beispielsweise *UPDATE*- oder *ALTER TABLE*-Befehlen ausführen, welche in den originalen Abfragen vorkommen. Damit hängt die Tatsache zusammen, dass die Tabellen in Spark unveränderlich sind: soll eine Tabelle geändert werden, muss an ihrer Stelle eine neue, modifizierte Tabelle erzeugt werden; deshalb kann man ihre Struktur nicht wie die der MySQL-Tabellen verwalten (erweitern, kürzen, aktualisieren usw.). Im TopicExplorer werden die Tabellen aber oft erweitert, was einen Umbau der Abfragen bedurfte.

Eine weitere Herausforderung betrifft den ursprünglichen Quellcode der Implementierung. Manche Abfragen waren in mehrere unabhängige Statements aufgeteilt, und mussten aus den for-Schleifen eines Java-Konstrukts heraus aufgerufen werden. Diese Stückelung der Anfragen musste rückgängig gemacht werden, indem solche Konstrukte zusammengefasst wurden, zu je einer großen Abfrage pro Abfragen-Schleife. Die SQL-Statements waren bei der alten Implementierung außerdem auf eine unübersichtliche Weise im Java-Quellcode eingebettet. Dies war zum Teil der Arbeitsweise des Eclipse-Formatters geschuldet, da die Autoformatierung auf eine aufwändige Weise umgangen werden musste, damit die Abfragen an MySQL im passenden Textformat an den Server geschickt werden. Neue Festlegung in diese Hinsicht lautete, eine Struktur in den Quellcode hineinzubringen: die SQL-Statements sollen in einzelne Textdateien ausgelagert werden, wo sie in einem bestimmten, für Menschen gut lesbaren Format gespeichert und bei Bedarf leicht wiederzufinden und zu erweitern sind. Die strukturellen Besonderheiten der MySQL-Umsetzung hatten sich erst bei der Umstellung auf Spark als Nachteile erwiesen, weil sie den Back-End-Umtausch etwas erschwert hatten.

Die letzte Herausforderung: die Umsetzung der Topic-Explorer-Vorberechnungen in Spark erforderte neben gewünschten Fähigkeiten (SQL, Java) mehr Lernaspekte wie die Aneignung einer funktionale Sprache (Scala) und der funktionalen Besonderheiten von Spark. Man muss als Spark-Programmierer selbst entscheiden, wann die Daten materialisiert werden müssen – alle Cache-Punkte werden deshalb manuell gesetzt. Eine automatische Lösung zur Berechnung der sinnvollsten Stellen für Cache-Punkte gibt es zurzeit nicht, und so musste eine Cache-Strategie für die Spark-basierte Vorberechnung ausgearbeitet werden.

**Fazit zu den Herausforderungen** Beim deklarativen Ansatz hängt die Formulierung der zu berechnenden Probleme theoretisch nicht von deren low-level-Umsetzung ab. Praktisch ließ sich das SQL-Abfragenkonstrukt jedoch nicht unverändert von MySQL zu Spark übernehmen. Die Ursache für diesen Aufwand waren strukturelle Nachteile der ursprünglichen MySQL-Umsetzung, die z.T. fehlenden Funktionalitäten in Spark (Version 1.6.1) sowie spezielle Besonderheiten von Spark-SQL-Programmierkonzepten.

# 2 Verwandte Arbeiten / Related Work

## 2.1 TopicExplorer und andere Topic-Model-Browser

Statistische Themenmodelle können die Auswertung der Inhalte großer Informationsmengen durch die Automatisierung vereinfachen. Ihre Ausgabedaten erscheinen jedoch für einen nicht eingeweihten Nutzer sehr unübersichtlich. Um die durch Textanalyse entstandenen Daten verständlich und validierbar zu machen, benötigt man spezielle Visualisierungswerkzeuge – Topic-Model-Browser.

Dazu wurde neben *TopicExplorer* [8] eine Reihe anderer Systeme* entwickelt. Sie entstanden meist durch die Zusammenarbeit von Informatikern und Sprachwissenschaftlern, und wurden für die spefizischen Anwendungszwecke entworfen. Zu den international bekannten Beispielen gehören das US-amerikanische *Serendip*-System [9] zur Analyse von englisch-sprachigen Textarchiven sowie *LDAvis* [10] und *topic-explorer* [11]. Beim letzten Topic-Model-Browser könnte durchaus eine Verwechslungsgefahr bestehen: topic-explorer kommt aus den USA, TopicExplorer ist dagegen eine deutsche Entwicklung. Die aktuell am besten ausgebauten Werkzeuge sind Serendip und TopicExplorer (Stand: Mitte 2015). Im Vergleich bieten diese beiden einen ähnlichen Funktionalitätsumfang in Themenrepräsentation und -Ranking, wenngleich deren Kommunikation mit dem Nutzer mit zwei unterschiedlichen Techniken implementiert ist. TopicExplorer kann seinerseits mit wichtigen Zusatzfunktionen überzeugen: beispielsweise lässt sich der zeitliche Verlauf von Themen bildlich rekonstruieren, und die Themen können interaktiv zusammenfasst werden [11].

Vergleichen lassen sich die verschiedenen Topic-Model-Browser in folgenden „Disziplinen“: maschinelles Lernen der zeitlichen Themenentwicklung, deren Visualisierung, die hierarchische Themengliederung und die Software-Organisation.

**Zeitliche Themenentwicklung** Die zeitliche Themenentwicklung erfolgt bei fast allen Visualisierungssystemen (TopicExplorer, TIARA [12], EvoRiver [13], RoseRiver [14]) prinzipiell in zwei Schritten: zuerst werden die Themen ohne Rücksicht auf die Zeitstempel gelernt, und anschließend wird eine Darstellung der Themenentwicklung mit den Zeitangaben erarbeitet. Das Aachener Projekt *D-VITA* [15] zeigt aber einen anderen, rechenintensiveren Ansatz, wo die Zeitstempel beim Lernen von Themen gleich mitberücksichtigt werden. Bisher lässt sich nicht sagen, welche Herangehensweise sich besonders vorteilhaft für die Inhaltsanalyse herausstellt.

**Visualisierung** Alle Systeme bieten eine interaktive Übersicht zur Themenentwicklung. Zusätzlich dazu präsentieren TIARA, EvoRiver und RoseRiver (noch) prototypische Darstellungen für die Formung der Beziehungen zwischen Themen. Für spätere Versionen von TopicExplorer ist dies ebenfalls geplant, mit dem Ziel, neuentstehende Verbindungen zwischen den Themen erkennen zu können.

**Hierarchische Themengliederung** Die hierarchische Themengliederung wird in LDAvis, TopicPanorama [16] und HierarchicalTopicGrid [17] prototypisch visualisiert. Der Schwerpunkt dieser Musterumsetzungen liegt in der Untersuchung von speziellen Visualisierungs- und Interaktionstechniken. Die Hierarchie und zeitliche Entwicklung von Themen werden in TopicExplorer, RoseRiver und HierarchicalTopics [18] vereint (Anmerkung: letzteres ist nur eine prototypische Studie). RoseRiver ist dabei eine spezielle Visualisierung für wenige vorausgewählte Themen; diese kann nach einer Begutachtung im TopicExplorer umgesetzt werden.

**Software-Organisation** Das Software-Design sowie solche Aspekte wie Wartung, Erweiterbarkeit und Konfigurierbarkeit finden eine ausdrückliche Betonung nur in den Projekten TopicExplorer und D-VITA. In den bisherigen Publikationen anderer Topic-Browser-Entwicklerteams wurde darauf kein besonderes Augenmerk gerichtet.

Objektiv besitzt TopicExplorer im Vergleich zu anderen Systemen die größte Funktionsvielfalt und deckt fast alle aktuellen Forschungsaspekte zur grafischen Darstellung und Interaktion mit Themenmodellen ab. Ferner ist TopicExplorer eins der wenigen Systeme mit einer deklarativen Implementierung. Das bringt einen zusätzlichen Vorteil, dass weitere Funktionalität leicht hinzugefügt und die Ausführungsebene ausgetauscht werden kann.

*** Anmerkung:** die oben aufgeführten Vergleiche zwischen verschiedenen Topic-Model-Browsern wurden von Dr. Alexander Hinneburg und Prof. Dr. Christian Oberländer erarbeitet, und stammen aus einer persönlichen Mitteilung.

## 2.2 Probabilistische Themenmodelle

Die *probabilistische Themenmodellierung* ist die Grundlage für Topic-Model-Browser – und ist somit einer der wichtigsten theoretischen Bestandteile von **TopicExplorer**, wo die Themen nach dem Wahrscheinlichkeitsmodell **Latent Dirichlet Allocation** modelliert werden.

### 2.2.1 Latent Dirichlet Allocation

**Latent Dirichlet Allocation** (LDA, deutsche Übersetzung: verborgene Dirichlet-Verteilung) ist ein Themenmodell im Gebiet der probabilistischen Themenmodellierung [19, S. 10] [2, S. 77]. Die grundlegende Idee hinter diesem Modell ist es, dass die Dokumentensammlungen (Korpora) eine Mischung aus verschiedenen Themen in bestimmten Proportionen beinhalten. Es wird angenommen, dass die Dokumente aus einem imaginären, zufälligen Generierungsprozess heraus entstanden seien [2, S. 78], und deshalb eine bestimmte verborgene (Themen-)Struktur aufweisen, welche durch das auf den statistischen Annahmen basierende Rückgängigmachen von dem Generierungsprozess aufzudecken gilt. Die Antwort auf die Frage, warum der generative Prozess nur imaginär ist, ist einfach – es ist unwahrscheinlich, dass die echten auszuwertenden Textkorpora wirklich „erwürfelt“ wurden.

**Annahme: Generierungsprozess für Texte** Dieser generative Prozess kann für jeden einzelnen Text aus der Datensammlung wie folgt beschrieben werden; seien die Wörter in zwei Schritten generiert worden:

1. Wähle zufällig eine Verteilung über Themen (so beinhaltet jedes Dokument die Themen in unterschiedlicher Proportion)
2. Für jedes Wort im Text/Dokument
   a) wähle zufällig ein Thema aus der Verteilung über Themen (variiert je nach Dokument, siehe Schritt 1).
   b) wähle zufällig ein Wort aus diesem Thema.

Es gibt nur einen gemeinsamen Themen-Menge für alle Dokumente aus der Sammlung, die Themen in jedem einzelnen Dokument werden jedoch in unterschiedlichen Maßen gedeckt.

Ein Thema sei eine Verteilung über ein festes Vokabular: zum Beispiel, das Thema „Computer“ beinhaltet mit hoher Wahrscheinlichkeit Wörter, welche die Rechnersysteme (Computer) betreffen, wie z.B. das Wort „Prozessor“. Angenommen wird, dass alle Themen noch vor der Generierung der Dokumente spezifiziert wurden.

In dem für die Theorie angenommenen generativen Prozess spielt die *Dirichlet-Verteilung* eine zentrale Rolle. Sie lässt sich an einem gut vorstellbaren Beispiel erklären.

Man stelle sich die Zufallsexperimente mit einem echten, sechsseitigen Spielwürfel als einen Beispielkontext vor: um eine ungefähre Wahrscheinlichkeitsfunktion zu ermitteln, kann man mit (ausreichend vielen) Stichproben arbeiten. Die echten Würfel sind, genau betrachtet, jedoch nicht exakt gleich, und die Wahrscheinlichkeitsfunktion für jeden Würfel weicht vom Idealfall etwas ab. Somit wäre ein Beutel mit vielen echten Würfeln ein Beispiel, das man mit einer Dirichlet-Verteilung modellieren kann: ziehen wir zufällig einen Würfel aus dem Beutel, so ziehen wir eine Wahrscheinlichkeitsfunktion – da jeder Würfel ungleich dem anderen ist. Wenn wir auf den generativen Prozess zurückkommen, gleicht eine Themenverteilung für einen Artikel einem solchen Beutel mit Würfeln, und ein Thema können wir als einen Würfel vorstellen, mit einem Wort an jeder seiner Kanten.

Das Ergebnis der Dirichlet-Verteilung wird im generativen Prozess dazu genutzt, um die Wörter aus dem Dokument verschiedenen Themen zuzuordnen (siehe Histogramm* in der Abb. 1).

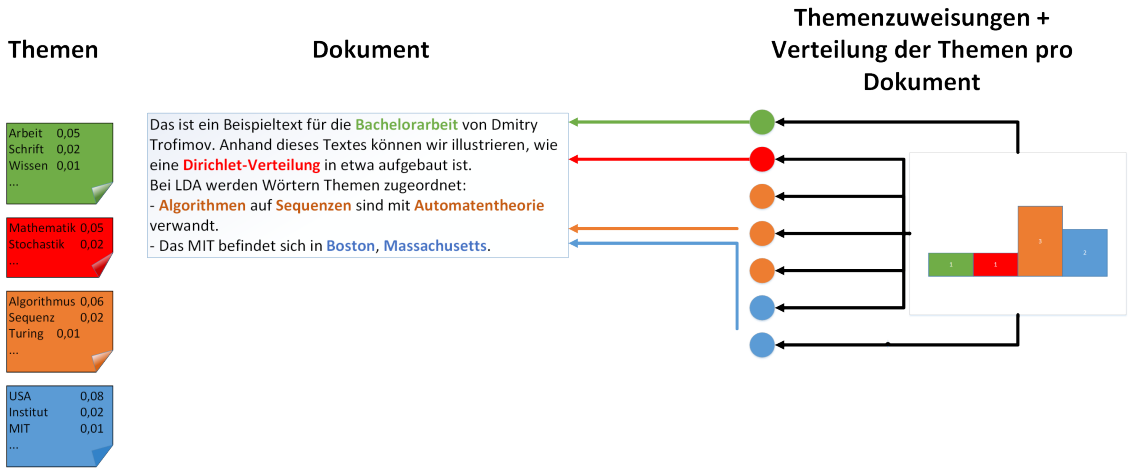



Abbildung 1: Die LDA-Annahmen: Themen seien Verteilungen über Wörter und gegeben für die ganze Dokumentensammlung (links). Jedes Dokument sei wie folgt generiert worden: wähle eine Verteilung über Themen (Histogramm rechts), wähle danach eine Zuweisung der Themen (Farbkreise Mitte rechts), wähle anschließend ein Wort aus dem Thema.

*** Anmerkung zur Abbildung:** Der Text sowie die Verteilungen in der Abbildung stellen keine echten, ausgewerteten Daten dar und dienen nur zur Veranschaulichung.

Die Dirichlet-Verteilung ist einsetzbar, um die Wortverteilung in Texten zu modellieren. Hätte man ein Wörterbuch mit $k$ möglichen Wörtern, dann könnte man *einen* bestimmten Text als eine Wahrscheinlichkeitsfunktion der Länge $k$ repräsentieren, welche durch die Normalisierung der empirischen (d.h. auf vorliegenden Daten basierende) Worthäufigkeit entsteht. *Eine Gruppe* von Texten würde eine Sammlung an Wahrscheinlichkeitsfunktionen erzeugen, und man kann darüber eine Dirichlet-Verteilung aufstel-

len, um die Variabilität dieser Wahrscheinlichkeitsfunktionen festzuhalten. Die Dirichlet-Verteilung würde dann wiedergeben, wie wahrscheinlich es ist, dass eine bestimmte Verteilung über Wörter auftritt. Verschiedene Dirichlet-Verteilungen können genutzt werden, um Publikationen mehrerer Autoren oder Arbeiten über etliche Themen zu modellieren [20, S. 3].

Die Autoren des Modells, D. Blei, A. Ng und M. Jordan, liefern folgende Definition [21, S. 993] (übersetzt aus dem Englischen):

> LDA ist ein dreischichtiges hierarchisches Bayes'sches Modell, bei dem jedes Element einer Sammlung als eine endliche Mischung von den Themen behandelt wird, die aus einer (der Sammlung zugrundeliegenden) Menge an Themenwahrscheinlichkeiten hervorgehen. Im Rahmen des Kontexts der Textmodellierung liefern die Wahrscheinlichkeiten für das Auftreten der Themen eine explizite Darstellung eines bestimmten Dokuments.

**Verborgene Themenstruktur** Die Texte/Dokumente für die Themenmodellierung werden vom Nutzer vorgelegt, die Themenstruktur – inkl. Themenmenge, Themenverteilung pro Dokument sowie Wort-Themen-Zuweisung pro Dokument – ist anfangs allerdings nicht gegeben; sie ist verborgen.

Das zentrale Problem für die Modellierung der Themen ist es, diese verborgene Themenstruktur aufzudecken, indem man den angenommenen zufälligen Generierungsprozess der Dokumente „rückgängig" macht. Man versucht also zu erraten, welche verborgene Themenstruktur für die Generierung der vorliegenden Datensammlung potenziell zuständig gewesen sein könnte.

Die algorithmisch erschlossene verborgene Struktur ähnelt der thematischen Struktur der Dokumentensammlung, und ergänzt jedes Dokument mit statistisch erschlossenen Vermerken, welche für verschiedene Zwecke genutzt werden können. Beispiele für den Nutzen sind Dokumentenklassifizierung und *Information Retrieval* [2, S. 79].

**Variablen der Wahrscheinlichkeitsverteilung** In der generativen Themenmodellierung, zu der LDA gehört, werden die Daten so behandelt, als wären sie durch einen Prozess mit den darin involvierten verborgenen Variablen erzeugt. Dieser Erzeugungsprozess definiert eine multivariate (d.h. mehrere unbestimmte Variablen enthaltend) Verteilung über sowohl beobachtbare als auch verborgene Variablen. Für die Analyse nutzt man diese multivariate Verteilung und die gegebenen, beobachtbaren Variablen, um eine bedingte Wahrscheinlichkeitsverteilung der verborgenen Variablen zu errechnen. Diese bedingte Verteilung nennt sich auch *A-posteriori*-(Wahrscheinlichkeits-)verteilung aus dem Bereich der Bayes'schen Statistik [2, S. 80].

Die beobachtbaren Parameter sind die Wörter im Dokument, die verborgenen bilden ihrerseits die Themenstruktur. Will man die letztere errechnen, ist das prinzipiell ein Berechnungsproblem einer A-posteriori-Verteilung – einer bedingten Verteilung der verborgenen Variablen in einem Dokument [2, S. 80].

Wovon hängt diese bedingte Verteilung ab? Sie wird durch verschiedene Abhängigkeiten definiert. Die Zuweisung der Themen richtet sich z.B. nach dem Themenverhältnis pro Dokument; ein bestimmtes Wort im Text ist gebunden an sein zugewiesenes Thema, und es hängt von der Themenverteilung im Dokument sowie im ganzen Korpus ab. Diese Abhängigkeiten sind im Texterzeugungsprozess in den statistischen Vermutungen verklausuliert. Sie spiegeln sich auch in den *probabilistischen grafischen Modellen* für LDA wieder [2, S. 80].

**Vorgänger von LDA** LDA wurde entwickelt, um die Probleme mit dem zuvor konzipierten probabilistischen Modell pLSI (*probabilistic latent semantic analysis*) zu beheben. Dieses war eine probabilistische Version einer maßgebenden Arbeit an der latenten semantischen Analyse (eng. latent semantic analysis) [2, S. 80] [22, 391-?407]. Diese Publikation dokumentierte die Nützlichkeit der Zerlegung von Dokument-Begriff-Matrizen in Einzelwerte [2, S. 80]. Aus der Perspektive der Matrizenfaktorisierung kann LDA deshalb auch als eine Art von Hauptkomponentenanalyse für diskrete Daten angesehen werden [2, S. 80] [23] [24].

**Algorithmen für Themenmodellierung** Welche Algorithmen zum *Topic Modeling* sind geläufig? Allgemein lassen sich die algorithmischen Ansätze zur Themenmodellierung in zwei Kategorien unterteilen: in variations- und in stichprobenbasierte Algorithmen.

Bei den stichprobenbasierten Algorithmen wird versucht, aus der A-posteriori-Verteilung Stichproben zu sammeln, um diese Verteilung mit einer empirischen Verteilung zu approximieren [2, S. 81]. Ein Beispiel dafür wäre Gibbs-Sampling [25, S. 8]: man konstruiert eine Markow-Kette – eine Sequenz aus Zufallsvariablen, wo jede Zufallsvariable von der vorhergehenden abhängt. Die asymptotische Verteilung dieser Markow-Kette ist die A-posteriori-Verteilung. Die Markow-Kette wird für die verborgenen Variablen in einem bestimmten Dokument definiert. Der Algorithmus hierfür besteht darin, für eine ausreichend lange Zeit Stichproben aus der asymptotischen Verteilung zu sammeln und sie als Vektoren aufzufassen, um anschließend die gesuchte Verteilung mit Hilfe der gesammelten Stichproben zu approximieren [2, S. 81]. Gibbs-Sampling kommt im TopicExplorer in der Komponente *Mallet* zum Einsatz.

Die variationsbasierten Algorithmen könnten als eine deterministische Alternative zu den stichprobenbasierten Verfahren erwähnt werden [26, S. 183-233] [27, 1?-305]. Es soll angemerkt werden, dass diese für TopicExplorer nicht relevant sind, und ihre Erwähnung nur zur Vervollständigung des Gesamtbildes für theoretische Hintergründe dient. Anstatt die A-posterior-Verteilung zu approximieren, stellen die variationsbasierten Methoden eine parametrisierte Familie von Verteilungen über die verborgene Struktur auf [2, S. 82]. Anschließend suchen sie nach einem Glied der Verteilungsfamilie, welcher am nähesten (gemessen nach dem Kullback-Leibler-Abstand, einem Maß fur die Unterschiedlichkeit zweier Wahrscheinlichkeitsverteilungen [28, S. 13]) zur A-posterior-Verteilung steht [2, S. 82].

Beide Algorithmentypen implementieren die Suche über die Themenstruktur; die Dokumentensammlung (der Korpus) liegt für beide als unveränderliche Quelle für beob-

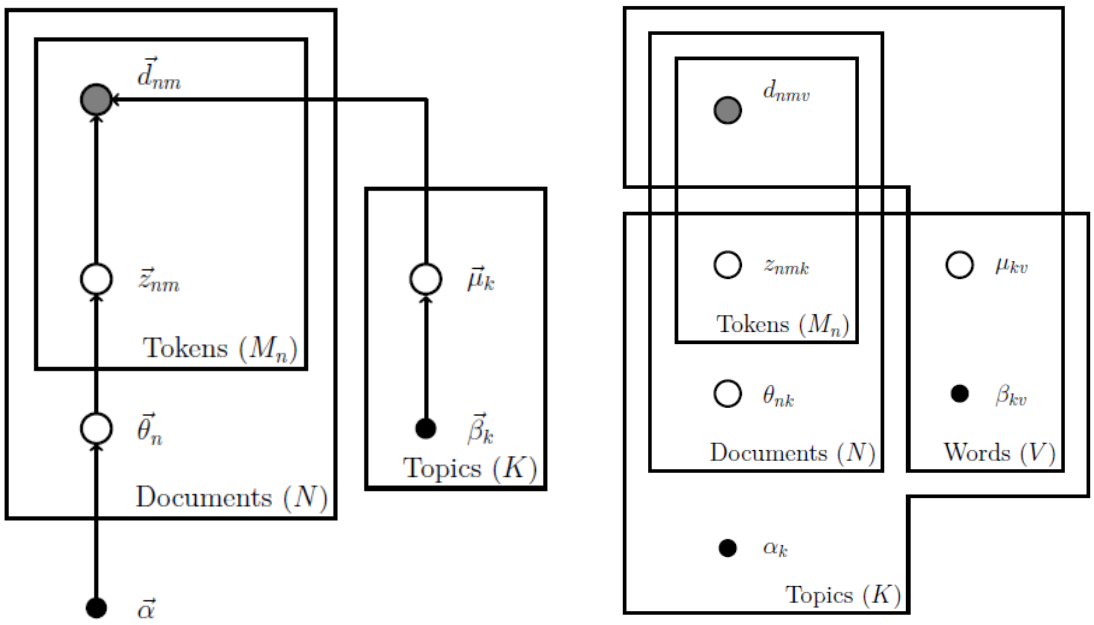
achtbare Zufallsvariablen vor, und dient als eine Art Leitfaden für die Suche nach den verborgenen Strukturen [2, S. 82]. Die Antwort auf die Frage, welcher Ansatz besser geeignet ist, hängt von dem anzuwendenden Wahrscheinlichkeitsmodell ab, und verbleibt noch als Grund für akademische Debatte [2, S. 82].

Das in diesem Kapitel behandelte LDA-Modell steht insofern in Verbindung mit Datenbanken und dem deklarativen Textmining, dass sich LDA in ein Entity-Relationship-Modell – eine häufig verwendete Weise für Designbeschreibung der relationalen Datenbanken – umwandeln lässt.

### 2.2.2 Übersetzung der LDA- in Entity-Relationship-Modelle

Die probabilistischen grafischen Modelle bringen den Aufbau der Familien von Wahrscheinlichkeitsverteilungen auf eine grafische Weise zum Ausdruck [2, S. 80]. Sie sind Graphen, in denen die Knoten Zufallsvariablen repräsentieren, und die An- bzw. Abwesenheit der Kanten auf bedingte Annahmen zu deren Abhängigkeit hindeuten [29] – eine Art „Heirat zwischen Wahrscheinlichkeits- und Graphentheorie“ (Zitat aus [30, S. 1]).

Eine Weise der zweidimensionalen grafischen Darstellung ist die sogenannte *plate*-Notation (oder auch *plate model* genannt, frei übersetzt: *Tafelmodell*). Das ist eine kompakte Form zur Darstellung komplexer grafischer Modelle. Jeder Teilgraph ist zwingend ein Teil jedes Elements seines Obergraphen [31, S. 48]. Beispiele zur Veranschaulichung findet man in der Abbildung 2. Mehr zu *plate notation* findet man in [32, Kapitel 14.3, Probabilistic Relational Models].




(a) LDA als *plate model* (b) LDA als atomares *plate model*

Abbildung 2: Grafische Darstellungen des probabilistischen Modells von TopicExplorer (LDA), **Quelle**: [33, S. 33]

**Erklärung* zu den Abbildungen** Ein Korpus ist ein Satz von Dokumenten. Ein Artikel $n$ aus dem Korpus $N$ besteht aus einem Token-Satz $M_n$. Jeder Token ist genau ein Wort, beschrieben durch den Vektor $\vec{d}_{nm}$. Die Variable $\vec{z}_{nm}$ ist 1-aus-K-codiert. Sollte der Token $m$ im Dokument $n$ dem Thema $k$ zugewiesen werden, hat die Variable $\vec{z}_{nm}$ eine Eins an der mit dem Thema $k$ assoziierten Stelle stehen. Jedes Thema $k$ kommt aus einer endlichen Menge von Themen $K$ und besitzt seine eigene, durch $\vec{\mu}_k$ parametrisierte Wortverteilung. Die Themenverhältnisse/-Proportionen pro Dokument sind repräsen-

tiert durch $\vec{\theta}_n$. Die Vektoren $\vec{\alpha}$ und $\vec{\beta}_k$ sind A-priori-Parameter (oder Hyper-Parameter) nach dem Bayesschen Wahrscheinlichkeitsbegriff.

Jeder Knoten steht für eine Zufallsvariable. Die verborgenen Variablen (Themenproportionen, -zuweisungen, sowie die Themen selbst) sind helle Knoten. Die beobachtbaren Variablen (Wörter im Dokument) sind graue Knoten. Die Rechtecke stellen nach der *plate notation* die Wiederholung der Komponenten dar. Das *Tokens*-Rechteck befindet sich z.B. im *Dokuments*-Rechteck drin, was bedeutet, dass jedes Dokument seine eigenen Tokens besitzt, und es keine Tokens ohne ein Dokument geben kann.

*** Anmerkung**: übersetzt aus [33, S. 32-33].

Frank Rosner hat sich in seiner Master-Arbeit [33] u.a. damit beschäftigt, wie man grafische probabilistische Modelldarstellungen in Entity-Relationship-Modelle übersetzen kann. Seine Fallstudie [33, S. 32-39] betrifft Topic Explorer und beschreibt, wie man LDA – das probabilistische Modell hinter TopicExplorer – in ein ER-Modell mit Hilfe von eine von ihm beschriebenen algorithmischen Ansatz [33, S. 19-29] umwandeln kann.

Die Übersetzung einer vorliegenden *plate notation* (siehe Abb. 2a) erfolgt in drei Schritten:

1. Führe die mehrdimensionalen Variablen auf ihre Bestandteile zurück. Das Ergebnis in diesem Schritt sei APM (eng. *atomic plate model*, übersetzt: *atomares Tafelmodell*; siehe Abb. 2b).
2. Wandle APM in ein Entity-Relationship-Modell um.
3. Reduziere das ER-Modell, um Übersetzungsartefakte zu vermeiden. (siehe Abb. 3)

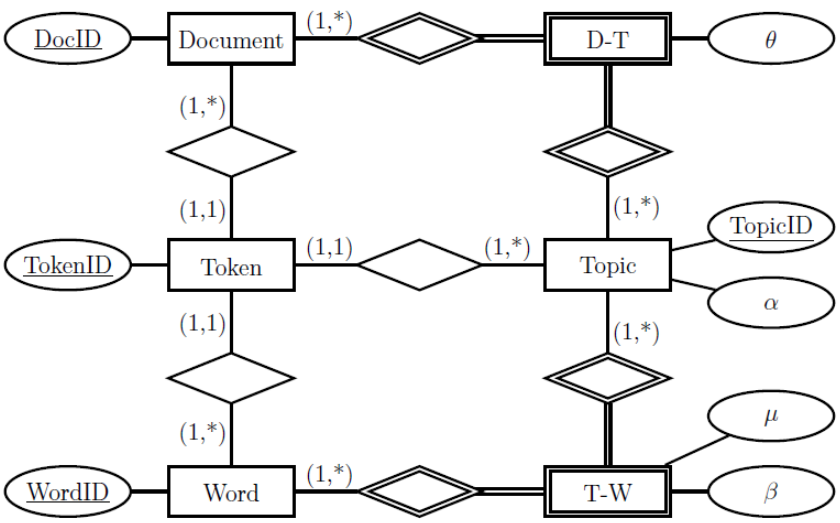



Abbildung 3: Das aus dem grafischen LDA-Modell übersetzte ER-Modell nach der Reduktion. **Quelle**: [33, Fallstudie zu TopicExplorer, S. 35]

Ein auf einem solchen Weg erhaltenes ER-Modell dient als „Kern“ für den Entwurf eine relationaler Datenbank, die wiederum eine Grundlage für den deklarativen Textmining-Ansatz von TopicExplorer ist. Liegen die probabilistischen Daten der Textvorverarbeitung in Form der Datenbankeinträge vor, so kann man sie mit Hilfe von SQL-Abfragen analysieren.

Auf diese Weise lässt sich der Bogen zur Zielstellung dieser Bachelor-Arbeit schließen: gegeben ist die Implementierung für ein Themenmodell; dieses lässt sich in ein ER-Diagramm überführen, mittels dessen eine Datenbank entworfen werden kann; gewollt ist Effizienz und Stabilität der Analyse von den durch die Modellierung erhaltenen und in der Datenbank gespeicherten Daten mit Hilfe von SQL-Abfragen. Dies soll durch den Einsatz von Spark errungen werden.

## 2.3 Spark

### 2.3.1 Apache Spark allgemein

Apache Spark ist eine in Scala geschriebene [34] Allzweck-Engine für *cluster computing* (Cluster = Rechnerverbund), welche die Zwischenergebnisse der Berechnungen fast ausschließlich im Hauptspeicher ablegt, mit mehreren Prozessorkernen parallel arbeitet, und seitens SQL für die TopicExplorer-Vorberechnung eine viel höhere Rechenressourcenauslastung als MySQL verspricht. Spark verfügt neben einer SQL-API über Programmierschnittstellen in den Sprachen Scala, Java und Python sowie über Bibliotheken für Graphenverarbeitung, Machine Learning und Verarbeitung der Datenströme (*streaming*) [35] [36, S. 1].

Die erste Version von Spark wurde im Jahr 2010 veröffentlicht. Nach dem Stand von Ende Mai 2015 (relativ zur Veröffentlichung des Papers *Spark SQL: Relational Data Processing in Spark* [36]) ist Spark das aktivste Open-Source-Projekt für Big-Data-Verarbeitung. Am 26. Juli 2016 wurde mit Spark 2.0.0 eine neuer Entwicklungsmeilenstein gesetzt: die neue Version brachte die Unterstützung vom SQL-2003-Standard, viele Leistungsverbesserungen und eine große Anzahl an Fehlerkorrekturen mit sich [37].

**Vergleiche zu anderen Frameworks** Geschwindigkeit ist ein wichtiger Punkt, weil es bei TopicExplorer um die Verarbeitung großer textueller Datenmengen geht.

In der Studie [38] wurde Spark mit Apache Hadoop verglichen – einer Open-Source-Implementierung von *MapReduce*. Im Hinblick auf Geschwindigkeit war Spark meist doppelt so schnell als sein Gegenüber. Begründen lässt sich das zum Teil mit Sparks Durchführung von Berechnungen im Arbeitsspeicher sowie dem zurückhaltenden Verhalten bei der Datenmaterialisierung im Gegensatz zu Hadoop [38, S. 2120]: letzteres schreibt die Zwischenergebnisse auf die Festplatte [39, S. 721], Spark vermeidet es und legt die Rechenergebnisse dagegen nur dann im Hauptspeicher ab, wenn das explizit verlangt wird [39, S. 722]. Auch komplexe, auf Festplatte(n) laufende Anwendungen sind mit Spark schneller als MapReduce [40, Kapitel 1, S. 19]. Im direkten Vergleich schnitt Spark nur beim Sortieren schlechter ab als MapReduce [38, S. 2114].

Die hash-basierte Aggregationskomponente [38, S. 2113] sowie der reduzierte CPU- und Festplattenzugriffs-Aufwand durch das Caching von Datensätzen [38, S. 2110] sind weitere Gründe für die hohe Performanz von Spark.

Hohe Leistung von Spark bedarf eines hohen Arbeitsspeicherbedarfs [39, S. 727]. Eigene Beobachtungen können dies nur bestätigen: Spark kann nicht mit großen Daten umgehen, wenn der Arbeitsspeicher zu knapp gemessen ist (Beispiel: weist man Spark statt der gut geratenen 50 Gb RAM nur 8 Gb zu, wird das Rechnen im standardmäßigen MEMORY_ONLY-Speichermodus abgebrochen, sobald der Speicher nicht mehr ausreicht). Es ist jedoch schwierig, den Speicherbedarf für die konkrete Aufgabe einzuschätzen; zum Stand der Forschung in dieser Hinsicht: es wird seit 2013 an einem Modell für die Vorhersage des Trade-Offs zwischen Speicher- und Zeitbedarf bei iterativen Operationen gearbeitet [39, S. 727].

Im Allgemeinen ist Spark für die Unterstützung einer breiten Palette an Einsatzzwecken konzipiert, wo man zuvor separate verteilte Systeme benötigte: interaktive Warteschlangen, Stapelverarbeitung von Daten oder Streaming von Inhalten, um einige zu nennen. Spark lässt sich mit anderen Werkzeugen für Big Data integrieren, u.a. kann es auf Hadoop-Clustern (HDFS, HBase [39, S. 721]) laufen und mit jeder Hadoop-Datenquelle arbeiten [40, Kapitel 1, S. 19].

Spark SQL ist die für diese Bachelorarbeit wichtigste Spark-Komponente. Sie gehört seit der Version 1.0 zu Sparks Werkzeugen, und stellt einen Weg dar, mit strukturierten (mit einem Schema versehenen) sowie semistrukturierten Daten zu arbeiten. Spark SQL unterstützt verschiedene Eingabe-Datentypen (u.a. CSV und parquet). Eine Besonderheit von Spark SQL ist das Fehlen der zwingenden Notwendigkeit, die Daten aus der Quelle vollständig lesen zu müssen, solange Spark mit dem Datenschema vertraut ist: nur die Daten, welche das System braucht, werden gelesen – andere jedoch nicht [40, Kapitel 5].

### 2.3.2 Spark SQL

Spark SQL ist ein Modul für Apache Spark, welches die Funktionalität für relationale Datenverarbeitung in die API von Spark integriert. Konkret für TopicExplorer repräsentiert dieses Modul eine Möglichkeit deklarativer Datenbank-Abfragen mit Spark. Ferner erlaubt es dem Nutzer, die Funktionalität durch das Hinzufügen von Programmbibliotheken zu erweitern [36, S. 1]. Eine Bibliothek MLlib [41] kann beispielsweise Spark um *machine learning*-Pipelines ergänzen [36, S. 8]. Eine andere Bibliothek namens *spark-csv*, erlaubt einen schnellen und unkomplizierten Import von CSV-Tabellen [42]. Letztere Erweiterung erwies sich als hilfreich für die Spark-Version von TopicExplorer, da die Quell-Tabellen im CSV-Format importiert wurden.

Im Vergleich zu anderen, älteren Softwareprodukten bietet Spark SQL zwei wesentliche Vorteile [36, S. 1]:

- Eine viel engere Integration zwischen relationaler und prozeduraler Datenverarbeitung (mit Hilfe der DataFrame API).
- Vorhandensein eines eingebauten Optimierers (*Catalyst*), welcher sich um die Effizienz der Ausführung von Rechenoperationen kümmert.

Spark SQL baut auf einem vorhergehenden Projekt der Spark-SQL-Entwickler auf – dem System Shark, wo die SQL-Funktionalität erstmals in Spark umgesetzt war. Statt den Nutzer zwischen relationaler (mit SQL-Befehlen) und prozeduraler (mit *map, filter* o. ä.) API wählen zu lassen, erlaubt Spark SQL es, diese beiden APIs nahtlos miteinander zu kombinieren [36, S. 1]. Hierfür sind die DataFrame-Schnittstelle sowie die Programmbibliothek/Klasse SQLContext (umbenannt zu SparkSession ab Spark 2.0) zuständig. Somit lassen sich relationale Operationen sowohl auf externen Datenquellen (Hive, relationale DB, JSON)[36, S. 3] als auch auf den in Spark eingebauten verteilten Datensätzen ausführen.

Sparks DataFrame-API kann man als eine Haupt-Programmierschnittstelle in Spark SQL ansehen. Sie wertet die Operationen „faul“ aus (Stichwort: *Lazy Evaluation*), sodass der Optimierer seinerseits noch Verbesserungen zum Ausführungsplan für relationale Datenoperationen hinzufügen kann. Was bedeutet diese Art der Auswertung für den Anwender? Ein DataFrame-Objekt repräsentiert schließlich nur einen logischen Plan, nach dem der Satz der im DataFrame enthaltenen Daten bearbeitet werden *soll*; die Abarbeitung der Daten erfolgt aber erst dann, wenn das Rechenergebnis materialisiert werden muss.

**DataFrame-API** Der **DataFrame** ist ein Tabellen-Datensatz und der Name für eine API zur Planung relationaler Abfragen. Ein solcher Plan wird bei Verwendung von Spark SQL angelegt und an den Optimierer (Catalyst) weitergegeben, der sich um die Ausführung kümmert. Die Abfragen können sowohl als Strings als auch als funktionale Befehle formuliert werden [34]. Da die DataFrames die zurzeit stabilste und für relationale Operationen am besten optimierte Spark-API zu sein scheint [34] sowie die wichtigste Rolle in Spark SQL einnimmt, waren DataFrames die erste Wahl für die Umsetzung der SQL-Abfragenkonstukte für TopicExplorer.

Dieses Konzept ähnelt dem des *Data Frame* [43] aus der Programmiersprache R [44] – in R hat diese Schnittstelle jedoch keinen automatischen Optimierer. Prinzipiell sind DataFrames Sätze von *strukturierten* Einträgen, was bedeutet, dass sie ein Schema besitzen.

Sie können direkt aus den RDD – in Spark integrierten verteilten Datensätzen von Java-/Python-Objekten – erzeugt werden, was die relationale Datenverarbeitung in den bestehenden Spark-Programmen möglich macht. Die Struktur ist ein wichtiger Unterschied zu RDD; das spiegelt sich in („SchemaRDD“) wieder, dem alten Namen von Dataframes in früheren Spark-Versionen. Die DataFrames verfügen neben einer fester Struktur über den Zugriff auf eine domänenspezifische Sprache, welche alle üblichen relationalen Operationen erfasst (Aggregation, Join, Group-By usw.). Durch die Speicherung der Daten in Tabellenform fällt der Platzverbrauch von DataFrames im Vergleich zu Java- oder Python-Objekten deutlich geringer aus. DataFrames lassen es SQL-seitig zu, mehrere Aggregationen mit einem SQL-Statement in einem Arbeitsgang zu berechnen – eine Vorgehensweise, die sich mit den traditionellen funktionalen APIs schwierig gestalten lässt [36, S. 1]. Außerdem sind solche DataFrame-Operationen wie Pipelining, Prädikatenkellernutzung (eng. *predicate pushdown*) und automatische JOIN-Auswahl schwierig selbst für RDD nachzuprogrammieren; mit DataFrames sind sie jedoch schnell verfügbar und werden im Ausführungsplan noch zusätzlich optimiert [36, S. 10]. Fairerweise sollte man erwähnen, RDD *müssen nicht* in DataFrames umgewandelt werden, denn man kann auch mit RDD aus Row-Objekten (d.h. Objekten, die Tabellenzeilen darstellen) arbeiten – allerdings mit Verzicht auf die DataFrame-Vorteile. Die Leistungsvergleiche [36, S. 10] fallen zugunsten des Ansatzes mit DataFrames. Die Konvertierung RDD-zu-Dataframe sollte man in Spark SQL stets, wenn möglich, vorziehen, und die Rückkonvertierung möglichst vermeiden. Während der Arbeit an der TopicExplorer-Vorberechnung stellte sich heraus, dass Catalyst beim „Hin-und-Her-Springen“ zwischen RDD und Dataframe

scheinbar die Übersicht über den Verlauf der Rechenoperationen verliert, was sich im schlecht optimierten Ausführungsplan resultiert.

Nicht nur Spark SQL, sondern auch andere Spark-Komponenten (z.B. machine learning) nutzen DataFrames.

**Catalyst-Optimierer** Catalyst ist neben DataFrames einer der Hauptgründe für die schnelle Rechenleistung mit Spark SQL. Er ist vielseitig erweiterbar, und, ausgestattet mit charakteristischen Merkmalen der Programmiersprache Scala, lässt den Nutzer auf eine unkomplizierte Weise zusammensetzbare Regeln für die Code-Generierung hinzufügen sowie Stellen für vorgesehene Erweiterungen definieren. Mit Catalyst steht dem Nutzer eine Reihe an Funktionen zur Verfügung, die den heutigen Anforderungen für Datenanalyse genügen soll (wie z.B. Schnittstellen für verschiedene Arten von *machine learning*, Schemainterferenz für JSON, Abfragenzusammensetzung für externe Datenbanken usw.) [36, Kap. 4, Catalyst Optimizer, S. 4].

Catalyst profitiert von den guten Seiten (wie der musterbasierten Suche, eng. *Pattern Matching*) von Scala, wenn es darum geht, die zusammensetzbaren Regeln in einer Turing-vollständigen Sprache auszudrücken. Dieser Optimierer offeriert außerdem ein Framework zur Baumtransformation: dieses verwendet man zum Analysieren, Planen und Generieren des Codes während der Laufzeit. Mit diesem Framework kann Catalyst und somit auch Spark SQL um neue Datentypen (JSON, CSV, Avro, parquet) [36, S. 1] erweitert werden. Der Nutzer kann hinzukommend selbst Datentypen (*user-defined types*, UDT) und Datenquellen definieren [36, S. 7].

Andere, ältere erweiterbare Optimierer benötigten meist eine komplexe domänenspezifische Sprache zur Regelnformulierung, und dazu einen speziellen Compiler, um diese Regeln in einen ausführbaren Code zu übersetzen [45] [46] – ein Ansatz, welcher viel Wartungskosten mit sich bringt. Catalyst lässt den Entwickler dagegen die Programmiersprache Scala ohne Einschränkungen nutzen und die Regeln entsprechend vergleichsweise einfach definieren – und scheint sogar der erste in einer funktionalen Sprache geschriebene, reife Abfragenoptimierer zu sein [36, S. 4].

Die Wahl von Scala als Programmiersprache für die Catalyst-Implementierung ist damit zu begründen, dass sie als eine funktionale Sprache gut für den Compilerbau geeignet sind. In solchen Sprachen kann der Programmierer das Konzept der *Monaden* anwenden, was ihm z.B. bei der Erweiterung eines Compilers um neue Fehlerbehandlungsmethoden o.a. viel Arbeit abnehmen kann [47, S. 24-52]. Ein Programm in einer funktionalen Sprache besteht aus einer Menge von Gleichungen, und der Wert jedes Ausdrucks hängt nur von dessen freien Variablen ab. So wird der Datenfluss innerhalb des Programms somit explizit beschrieben, und so kann sichergestellt werden, dass die Reihenfolge der Berechnungen irrelevant ist [47, S. 24-52] – definitiv ein Vorteil bei der faulen Auswertung, welche in Spark für die Schnelligkeit der Berechnungen sorgt.

Dieser Optimierer arbeitet in seiner Kernfunktionalität mit Bäumen für formale Ausdrücke und Regeln, mit denen man diese Bäume transformieren kann [36, S. 5]. Jeder Knoten im Baum ist unveränderlich, besitzt einen Typ und kann Kinderknoten haben. Die Regeln sind Funktionen mit einem Baum als Ein- und Ausgabe, z.B. *transform*;

sie werden partiell angewendet, d.h. nur auf die Teilbäume, wo die Regeln anwendbar sind – an dieser Stelle kommt der Musterabgleich (*pattern matching*) zum Einsatz. Die Teilbäume, in denen die Regeln keine Anwendung finden, werden übersprungen. Ferner fasst Catalyst die Regeln zu Gruppen zusammen, und wendet jede einzelne Gruppe an einem Baum solange an, bis der Baum sich nicht mehr verändert [36, S. 5].

Über der „Baum- und Regeln"-Kernebene stehen speziell für relationale Abfragenverarbeitung entwickelte Programmbibliotheken (z.B. Bibliotheken für logische Abfragenpläne oder relationale Ausdrücke) sowie einige Regeln zum Behandeln der Abfragenabarbeitung. Eine Abfrage läuft die Phasen *Analyse, logische Optimierung, physische Planung und Codegenerierung* durch.

Die höhere (und äußerste) Ebene im Catalyst-Konzept ist ein Bestandteil der Scala-Sprache – die sogenannten „Quasiquotes" [48]. Sie werden zur Beschreibung von abstrakten syntaktischen Bäumen in Scala genutzt, und erlauben die Bytecode-Erzeugung aus den zusammensetzbaren Ausdrücken während der Laufzeit [36, S. 6]. Die abstrakten syntaktischen Bäume werden von Catalyst aus den Ausdrucksbäumen für SQL-Statements generiert.

Die Verarbeitung einer Abfrage läuft in dieser Reihenfolge ab [36, S. 6]:

1. Erstelle aus den Daten (DataFrame) oder einer Abfrage (SQL-Statement) einen logischen Plan.
2. Analysiere und löse die Referenzen in diesem logischen Plan auf.
3. Optimiere den aufgelösten logischen Plan.
4. Erstelle daraus einen oder mehrere physische Pläne und optimiere sie.
5. Wähle einen physischen Plan aus mit Hilfe eines Kostenmodells.
6. Generiere daraus einen ausführbaren Bytecode, kompiliere ihn und führe ihn aus.

Neue Umsetzung der Topic-Explorer-SQL-Vorberechnungen profitiert von der Kombination *Catalyst-DataFrame*, weil diese Methode sich gegen den möglichen, aber unpraktischen Ansatz mit RDD durchsetzt, da sie die meisten Optimierungen bietet.

### 2.3.3 Weitere Einsatzfelder von Spark

Bei der Wahl des zum TopicExplorer passenden Frameworks für Auswertung großer Datenmengen ist Spark nicht nur durch seine effizientere Ressourcenverteilung, sondern auch durch seine Vielseitigkeit aufgefallen. In diesem Kapitel wird über SQL etwas hinausgegangen; es werden Einsatzbereiche für Spark vorgestellt, die sich auf *Big Data* als ein potenzielles Interessengebiet für ähnliche Projekte ausdehnen.

**Spark und Datalog** Iterative Anwendungen sind eine Stärke von Spark, rekursive dagegen (wie z.B. Suche nach dem kürzesten Pfad in Graphen) eher weniger: für die Rekursion nutzt Spark einen iterativen Ansatz [5, S. 1135]. Für jede Iteration wird ein

neuer Job gestartet, und so erhält Spark nicht die gesamte Übersicht über die Ausführung der Anwendung. Um rekursive Algorithmen in Spark effizient zu gestalten, muss der Programmierer selbst nicht nur äußerst gut den zu implementierenden Algorithmus verstehen, sondern auch Sparks API und seine Interna. Das kann einen höheren Aufwand als nötig bedeuten, jedoch sollte man beachten, dass Spark andererseits auch einige für rekursive Verfahren essentiellen Eigenschaften mit sich bringt [5, S. 1135], wie z.B. das Caching eines Datensatzes, oder geringe Ressourcenkosten für das Starten eines Jobs. Eine Studie [5] hat untersucht, wie man Spark mit seinen Gegebenheiten dazu bringen kann, rekursive Anwendungen effizient zu unterstützen – und im Rahmen dieser Forschung wurde eine rekursive Datenbanken-Programmiersprache *Datalog* implementiert. Das aus dieser Arbeit entstandene Spark-Modul nennt sich *BigDatalog*.

Datalog ist eine deklarative Programmiersprache für deduktive Datenbanken. Eine deduktive Datenbank ist eine mit Regeln und Fakten erweiterte relationale Datenbank, welche das Herleiten von Schlussfolgerungen erlaubt [49]. Ein Datalog-Programm ist nichts anderes als ein endlicher Satz an Axiomen [5, S. 1136]. Programmiert wird darin, indem man eine Menge an Regeln erstellt. Die Auswertung erfolgt durch die Abfrage dieser Axiome und die sogenannte *Fixpunktberechnung* ohne explizite Angabe der Rekusionstiefe [49]. Der Fixpunkt ist dabei ein Punkt, ab dem durch weitere Auswertung keine neuen Fakten mehr hinzukommen.

Die Umsetzung von Datalog auf Spark gestaltete sich laut der Studie komplexitätsbedingt nicht immer leicht: manche Teile der Funktionalität waren entgegenkommend, andere Besonderheiten standen dafür im Weg.

Produktiv war die Tatsache, dass Spark eine API für SQL bietet. Das bedeutet, dass man die gegebenen Schnittstellen für eine weitere „Query Language“ samt des spracheigenen Compilers und Evaluierers wiederverwenden kann. An dieser Stelle nimmt die Datalog-Implementierung ihren Ursprung: *Spark SQL*'s logische und physische relationale Operatoren werden übernommen bzw. erweitert (so wurde die Komponente SqlContext ein Teil von BigDatalogContext [5, S. 1139], und die Optimierer-Operatoren werden mit Hilfe von Catalyst-Framework erweitert, sodass auch Datalog-Ausführungspläne optimiert werden). Durch den Wunsch der Wiederverwendung von Spark SQL ergaben sich allerdings weitere Schwierigkeiten: zum einen würde Sparks Aufgabenplaner/Scheduler monotone, ohnehin konsistente Datalog-Programme unnötig nach Ausführungsetappen koordinieren [5, S. 1139]; zum anderen ist Spark SQL nur für azyklische Pläne ausgelegt. So bräuchte eine neue rekursive, regelbasierte Syntax ein anderes Front-End als das von Spark SQL (Sparks SqlContext und Catalyst unterstützen standardmäßig keine rekursiven Operatoren) [5, S. 1139]. Weniger Vorteilhaft war auch die unveränderliche Beschaffenheit der zur BigDatalog-Umsetzung genutzten RDD, was bei rekursiven Ansätzen für große Daten ein gutes Speichermanagement sinnvoll macht – welches selbst zu implementieren galt [5, S. 1139].

BigDatalog verwendet eine an Spark angepasste und dafür optimierte, parallelisierte Version der von Datalog bekannten *semi-naiven Auswertung* [5, S. 1138]: zum einen gegebenen Zeitpunkt kann nur eine Iteration stattfinden, und mit der nächsten wird erst begonnen, wenn die vorhergehende abgearbeitet wurde. Die Regeln zum Ausstieg aus der Rekursion (die *exit rules*) und die rekursiven Regeln werden physisch separat geplant [5,

S. 1139]. Die *exit rules* werden nur am Anfang einmal ausgewertet, die rekursiven Regeln dagegen unterliegen wiederholter Auswertung bis zum Fixpunkt. Der physische Ausführungsplan wird seinerseits aus einem logischen Plan erstellt, und besteht aus Teilen von Spark SQL und den BigDatalog-Operatoren, welche RDD generieren; sollen die RDD mehrmals benutzt werden, finden sie ihren Weg in den Cache [5, S. 1140]. Für die parallele semi-naive Auswertung werden keine Standard-RDD, sondern SetRDD verwendet, um getrennte Tabellenzeilen speichereffizient unterzubringen. Die bei dieser Auswertung genutzten Methoden für die RDD-Transformation wurden ihrerseits durch effizientere ersetzt, welche mit SetRDD umgehen können und den Eingabenaustausch zwischen den Partitionen dem BigDatalog-Compiler überlassen [5, S. 1141]. Es wurden bei BigDatalog außerdem in Hinsicht auf lineare Rekursion Optimierungen für JOIN-Befehle vorgenommen [5, S. 1142]. Es gibt viel mehr technische Feinheiten von BigDatalog, die man in [5] nachlesen kann.

Die Leistungsvergleiche der Datalog-Umsetzung auf Spark können überzeugen: BigDatalog zeigte sich darin meist schneller als andere moderne Datalog-Systeme, und in Hinsicht auf rekursive Abfragen schlägt dieses sogar das ebenfalls Spark-basierte GraphX [5, S. 1147].

Mit BigDatalog kommen neue Möglichkeiten zur Aufgabendeklaration in Spark hinzu: in erster Linie Vorteile der logischen Programmierung, oder neue Ansätze wie z.B. *Tabled Logic Programming* (siehe [50]). Die Frage, welche Vorteile BigDatalog gegenüber Spark SQL in Hinsicht auf Geschwindigkeit beim Einsatz in einem Topic-Model-Browser wirklich bieten kann, bleibt noch offen: BigDatalog hätte wahrscheinlich das Potenzial, eine weitere Möglichkeit für deklarative Beschreibung der Themenmodell-Vorberechnungen zu werden; dies soll jedoch neben anderen Varianten noch abgewogen werden, wozu bisher keine Arbeit geleistet wurde.

**OLTP-OLAP-kombinierte Datenbanken** Unterschiedliche Datenbankensysteme sind für verschiedene Zwecke entwickelt, und besitzen dementsprechend unterschiedliche Eigenschaften. OLAP („*Online Analytical Processing*“) und OLTP („*Online Transaction Processing*“) sind zwei Arten von Datenbankverwaltung.

Bei OLTP bestehen die Datenbankprozesse aus atomaren Transaktionen auf momentan aktuellen Daten. Sie involvieren häufige, kurze Lese- und Schreibezugriffe auf wenige Datensätze über Primärschlüssel. Die Einzeldaten in der Datenbank liegen meist detalliert und unverdichtet vor. OLTP eignet sich gut für operative Systeme (z.B. für Administrationssysteme) [51].

OLAP ist hingegen analyseorientiert: der Zugriff erfolgt meist nur lesend, und an erster Stelle stehen historische, aggregierte Informationen [51]. Die vorliegenden Daten werden mit Hilfe von „Snapshots“ in regelmäßigen Zeitabständen aktualisiert. Dieser Ansatz ist für eine geringe Zugriffshäufigkeit ausgelegt, und die Lesetransaktionszeit ist meist durch komplexere Anfragen wesentlich länger als bei OLTP. Ein mögliches Einsatzfeld für OLAP sind Data Warehouses [51].

Durch moderne Tendenzen wie „Internet of Things“ steigt der Bedarf an Anwendungen, welche sich stark auf den Echtzeitbetrieb konzentrieren, und dabei einen intensiven

Datenaustausch handhaben können/müssen. Deshalb stechen in der Software für Big-Data-Anwendungen drei Schwerpunkte heraus [52, S. 2153]: (1) kontinuierliche Datenstromverarbeitung mit Echtzeitwarnungen, (2) Umgang mit schreibintensiven Aufgaben und Transaktionen sowie (3) interaktive Datenanalyse mit SQL nach dem OLAP-Ansatz. Diese Bereiche vereinen die Anforderungen sowohl an OLAP- als auch OLTP-Systeme. Eine Lösung, die alle drei genannten Schwerpunkte effizient abdeckt, sei aber nicht gegeben [52, S. 2153].

Die sogenannten SQL-auf-Hadoop-Lösungen (u.a. Spark SQL und Hive) decken nur den dritten Punkt ab. Sie nutzen Tabellenformate und OLAP-Optimierungsmethoden, um die Abfragen über große *statische* Datensätze zu tätigen [52, S. 2153]. Für Echtzeitoperationen an Datenbanken sind solche Systeme jedoch wenig geeignet. Die Tabellendaten durch Transaktionen zu verändern und dabei Datenkonsistenz beizubehalten wäre bei denen zu umständlich. Spark, als Beispiel hierzu, arbeitet mit unveränderlichen Datensätzen – sobald sie materialisiert sind, impliziert die Tabellenveränderung viel neue Rechen- und Speicherarbeit bis zur Neumaterialisierung der aktualisierten Daten. Ferner sind die Lesezugriffe, und folglich auch die Transaktionen bei solchen Systemen aufgrund fehlender Indexierung [52, S. 2153] langsamer. Das liegt vor allem am Grundkonzept dieser Software: Spark ist kein System für Datenmanagement, sondern eins für Datenverarbeitung (genauer: für Verarbeitung der Daten in Batches/Stapeln). Die Indexierung als eine verfügbare Optimierungsoption kann man z.B. in MySQL finden, aber nicht in Spark.

Essenziell für IoT-Anwendungen wäre auch Management der Nebenläufigkeit von Anfragen. Bei den erwähnten Systemen stellt dies ein Problem dar: Nebenläufigkeit von Transaktionen spielte bei der Konzipierung von Spark SQL und Hive eine geringe Rolle, weil sie zum Zweck effizienter Ausführung der Anfragen auf „großen“ Daten geschaffen wurden.

Als eine bessere Lösung für solche *Lücken* in der Funktionalität bieten sich hybride Systeme an (HTAP, *Hybrid Transaction/Analytical Processing*). Sie vereinen in sich sowohl OLTP als auch OLAP, indem sie die Informationen in zwei Formaten abspeichern: als Zeilen auf der Festplatte oder im Pufferspeicher, und als komprimierte Tabellen [52, S. 2153]. Nachteil: zur Unterstützung der Datenströmverarbeitung benötigen sie eine parallel dazu laufende *streaming engine* (wie Storm, Kafka oder Confluent) [52, S. 2153]. So werden die Schwerpunkte (2) und (3) berücksichtigt, für (1) wird man jedoch eine separate Lösung suchen müssen. Das wäre nicht sehr kritisch, insofern man eine ausreichend tiefe Datenstromanalyse erhalten würde; das ist leider nicht der Fall, weil die Werkzeuge zur Datenstromverarbeitung (z.B. Apache Samza) eine zu oberflächliche Auswertung beherrschen [52, S. 2153]. Für komplexe Analyse mit dieser Art der Tools sind OLAP-Optimierungen – wie effiziente Operatoren für JOIN, GROUP und Aggregationen – erforderlich [52, S. 2153] [53] [54].

Die Zusammensetzung eines Systems aus mehreren existierenden Software-Produkten zur Verwirklichung aller drei Zielaspekte – kontinuierliche Datenstromverarbeitung mit Echtzeitwarnungen, Umgang mit schreibintensiven Aufgaben und interaktive SQL-Analyse – kann sich in vielen Hinsichten negativ auswirken. Eine Erhöhung der Komplexität, die daraus folgenden höheren Implementierungs- und Wartungskosten und geringere Leis-

tung betonen den Wunsch nach einem *unifizierten* System. In diese Lücke positioniert sich **SnappyData**, ein Cluster-System auf Basis von Apache Spark.

SnappyData integriert Spark als Big-Data-Rechenplattform mit GemFireXD (in diesem Fall – als dessen Fork namens Snappy-Store [55]), einer transaktionalen In-Memory-Datenbank mit horizontaler SQL-Skalierung [56].

Spark versucht bei der Ausführung eines Jobs stets alle Prozessorkerne so stark wie möglich auszunutzen. Diese Vorgehensweise wird durch SnappyData ergänzt: kleinere Jobs können die anderen Operationen mit größerer Verzögerungszeit überlappen und eine höhere Priorität erhalten, ohne einen Scheduler einzuschalten [52, S. 2155]. Zur Unterstützung einer dichten Nebenläufigkeit während der Laufzeit wurde ein sogenannter „Job Server“ hinzugefügt, um die Anwendungen von den Datenbankservern zu entkoppeln (wie man es oft von den relationalen Datenbanken kennt) und die Zustandsinformationen zwischen Anwendungen und Clients zu übermitteln [52, S. 2155].

Die Datenstromverarbeitung, OLAP und OLTP haben in SnappyData eine gemeinsame API [52, S. 2155]. Spark SQL wird durch OLTP-Funktionen erweitert: an den Tabellen können die INSERT-, UPDATE-, DELETE-Operationen angewendet werden. Ferner sind *Constraints* (NOT NULL, UNIQUE usw.) sowie Indexierung möglich. Zum Schluss erhält Spark SQL damit auch die Möglichkeit der deklarativen Datenstromverarbeitung in SQL. Diese Funktionalitäten werden im Modul *SnappyContext* bereitgestellt, welches in SnappyData das native *SqlContext*-Modul in den Hintergrund drängt.

Aufgrund der hohen Anforderungen an die Geschwindigkeit der Datenstromanalyse ist SnappyData mit aktuellen AQP-Techniken (Approximate Query Processing) ausgestattet. Die Genauigkeit der Analyse soll mit Hilfe von *high-level accuracy contracts (HAC)* [57] auf eine intuitive Weise einstellbar sein. Für beliebig komplexe Abfragen auf Datenströmen kann SnappyData automatisch die statistischen Verzerrungen korrigieren; außerdem versorgt es den Nutzer mit Fehlerschätzungen [52, S. 2155].

SnappyData ist ein Ansatz, Spark als Big-Data-Plattform mit den Ideen von Hochverfügbarkeit, schnellem Transaktionsmanagement und Echtzeitdatenanalyse zu vereinen – d.h. mit Faktoren, für die Spark ursprünglich nicht konzipiert war. Das bezeugt von einer tiefgreifenden Erweiterbarkeit dieses Systems. Typische Anwendungsfälle beinhalten viel interaktives Big Data (wie z.B. Finanzmarktforschung). TopicExplorer ist zurzeit nach dem OLAP-Prinzip gestaltet: man lädt ein Snapshot der Dokumentensammlung in Form der Grundtabellen rein, und analysiert diesen. Sollte man im Laufe der Entwicklung dieser Software mehr Wert auf Echtzeitdaten legen, ließen sich die Vorteile von SnappyData theoretisch aber auch für deklaratives Textmining der sozialen Netzwerke nutzen.

# 3 TopicExplorer mit Spark

Der vollständige Umtausch des Back-End (MySQL gegen Spark) – einer der deklarativen Beschreibung untergeordneten funktionalen Ebene – bedurfte einige Anpassungen. Es wurden zwei Methoden für die Vorberechnung ausgearbeitet: eine ggü. der MySQL-Funktionalität vollständige, an Spark angepasste Herangehensweise, sowie eine neue, experimentelle und daher nicht vollständige Vorberechnungsmethode, welche zum Teil ausgetestet wurde.

**Vorgenommene Anpassungen** Im Wesentlichen bestand die alte MySQL-Umsetzung der TopicExplorer-Vorberechnung darin, den deklarativen Teil – die SQL-Befehle – aus den Java-Quellcodekonstrukten heraus aufzurufen. Die Java-Konstrukte wurden in der Spark-Umsetzung durch kompakteren Scala-Code ausgetauscht. Die SQL-Statements wurden übersichtlich formatiert und in einzelne Textdateien ausgelagert. Das Schema der Tabellenerzeugung wurde zwecks der Kompatibilität mit Spark etwas überarbeitet (die UPDATE- bzw. ALTER TABLE-Befehle wurden hinfällig, da Spark SQL mit ihnen nicht arbeiten kann).

Viele kleinere SQL-Statements waren in der MySQL-Version in *for*-Schleifen gepackt, konnten aber in der Spark-Version in jeweils eine einzelne, große Abfrage pro Schleife zusammengefasst werden. Spark kommt mit wenigen, aber rechenintensiven Abfragen besonders gut zurecht – mit vielen kleinen Statements würde man an dieser Stelle zusätzlichen Aufwand für Neuimplementierung der Schleifen betreiben und das Leistungspotenzial dieses Frameworks nicht ausnutzen.

Die Optimierungsmethoden, die einem in MySQL zur Verfügung standen, waren die Indexierung und das Aufsplitten der großen Abfragen in kleinere, unabhängige Statements. Erstere Methode kann die Arbeit von MySQL beschleunigen: u.a. verkürzt es die nötige Zeit für SELECT-Zugriffe und JOIN-Operationen. Die Indizes gewähren in MySQL einen sortierten Zugriff auf die Tupeln, der die *„GROUP BY“*-Abfragen – eine in TopicExplorer häufig genutzte Operation – schneller macht [58]. Spark hingegen verfügt über keine Indexierung: somit fielen die für Indexierung verantwortlichen Teile der Aufgabendeklaration weg. Hingegen bedarf Spark einer taktisch sinnvollen, manuellen Setzung von Tabellen-Cache-Punkten – einer anderen Art von Optimierung. Im Allgemeinen ist es in Spark sinnvoll, die Daten, welche man später nochmal braucht, in den Cache zu verschieben – die Daten liegen dann im Hauptspeicher und müssen bei Bedarf nicht von der Festplatte gelesen werden. Nicht-gecachte Daten müssten bei erneutem Abruf, solange sie nicht explizit abgespeichert wurden, nochmal ausgerechnet werden, was Zeit und Rechnerressourcen kostet.

Um das hierarchische Clustering von Themen kümmert sich ein R-Skript (für die Sprache R findet sich hierzu ein hilfreiches Paket *hclust* [59]). Das R-Skript musste an das neue Schema ebenfalls angepasst werden, seine Funktionsweise blieb aber im Wesentlichen gleich: die Themen liegen zuerst als „Blätter“ vor, und Ähnlichkeiten zwischen ihnen werden mit Hilfe spezieller hierarchischer Statements paarweise untersucht. Daraus wächst ein Baum für Themenhierarchie, welcher für die spätere Themenmodellanzeige eine Rolle spielt.

## 3.1 Ursprüngliche Tabellenerzeugung

In diesem Abschnitt wird die ältere Version der Tabellenerzeugung beschrieben, welche zuerst mit MySQL, und später mit Spark implementiert wurde.

Als Erstes wird die Datenbank mit Text- und Themenmodell-Informationen (Tokens, Begriffen usw.) gefüllt. Daraus entstehen die Quell-Tabellen *orgTable_text* und *DOCUMENT_TERM_TOPIC*. Mit ihrer Hilfe werden weitere Tabellen erzeugt (siehe Abb. 4 ).

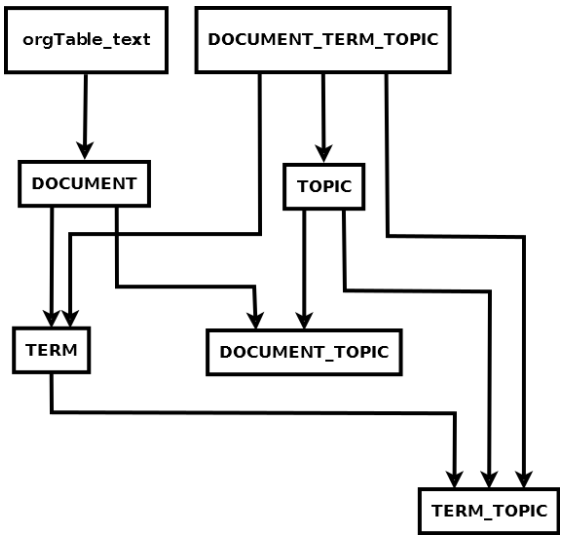



Abbildung 4: Flussdiagramm zur Tabellenerzeugung aus den Grundtabellen

Die Tabelle *orgTable_text* ist das Ergebnis des sogenannten Crawlings, d.h. dort befinden sich nur Textinhalte. Wenn z.B. eine Web-Seite gecrawlt wird, dann werden die für den Leser relevanten Seiteninhalte automatisch von den HTML-Textteilen getrennt und abgespeichert. Die Quell-Tabelle *DOCUMENT_TERM_TOPIC* ist das Resultat der Datenvorverarbeitung der Textinhalte, wo wie folgt vorgegangen wird:

- Die Texte werden in einzelne Wörter zerlegt.
- Ein spezielles computerlinguistisches Programm überführt die Wörter in ihre Grundform.
- Aufgrund der gelegentlich auftretenden Fehlschlüsse in der automatischen Lösung sortiert der Anwender per Hand das Vokabular aus.
- Das Themenmodell wird trainiert, einzelnen Token werden Wahrscheinlichkeitsbelegungen für Themenzuordnung zugewiesen.

Die Skizze zur Aufbaureihenfolge lässt allerdings viele Details außen vor. Einen Einblick in die Datenbank-Struktur gewährt ein ER-Diagramm in Abb. 5:

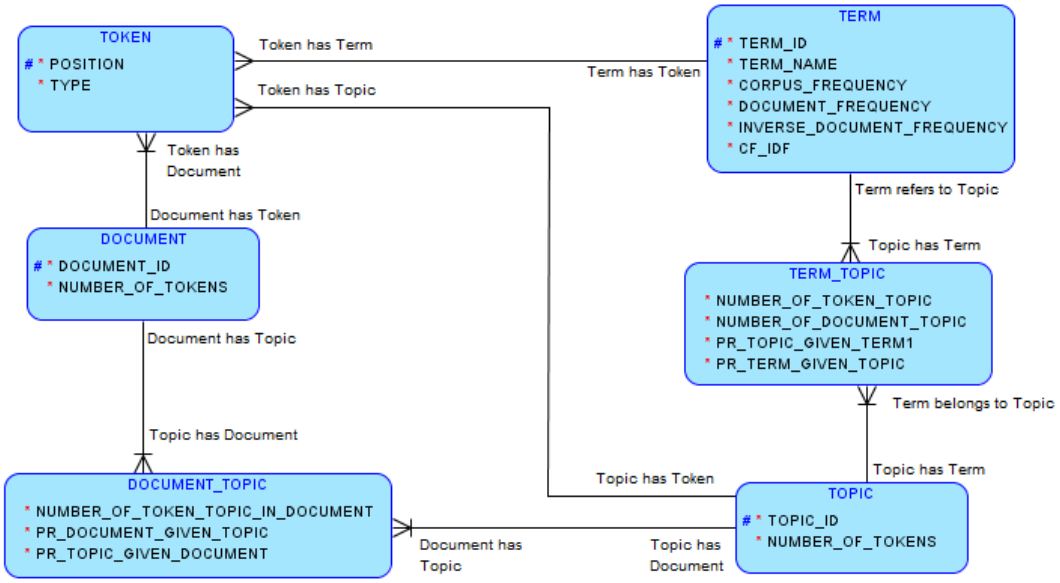



Abbildung 5: Entity-Relationship-Diagramm zum Datenbank-Aufbau

**Erklärung der Tabellenattribute** DOCUMENT_ID, TOPIC_ID und TERM_ID sind eindeutige Identifikatoren (jeweils für Dokumente, Themen und Wörter/Begriffe). NUMBER_OF_TOKENS ist die Anzahl der Tokens, die mit einem Dokument, Thema oder Wort in Vebrindung stehen (z.B. NUMBER_OF_TOKEN_TOPIC ist die Anzahl der Tokens pro Thema, und NUMBER_OF_DOCUMENT_TOPIC – die Anzahl der Dokumente, in denen sich ein bestimmtes Thema wiederspiegelt). Ein Token ist dabei ein Wort, welches mit einem bestimmten Thema assoziiert wird. Das Attribut CORPUS_FREQUENCY gibt die Häufigkeit für das Auftreten eines bestimmten Wortes im ganzen Korpus (d.h. in der ganzen Dokumentensammlung) an. DOCUMENT_FREQUENCY ist die Häufigkeit „Wort pro Dokument“, und INVERSE_DOCUMENT_FREQUENCY ist die invertierte Version davon. Sie wird beispielsweise im Bereich *Information Retrieval* für die Indexierung von Dokumenten gebraucht; man sucht mit deren Hilfe nach Wörtern, die durch hohe Auftrittshäufigkeit in wenigen Dokumenten diese gut repräsentieren können. CF_IDF ist CORPUS_FREQUENCY multipliziert mit INVERSE_DOCUMENT_FREQUENCY. PR_DOCUMENT_GIVEN_TOPIC ist die Wahrscheinlichkeit, mit der ein Dokument mit einem gegebenen Thema assoziiert wird (genau umgekehrt ist es bei PR_TOPIC_GIVEN_DOCUMENT).

In Spark gab es allerdings u.a. Probleme mit der Tabelle TERM. Deren Attribut TERM_ID war in MySQL eine *Auto-Increment*-Spalte. Spark SQL besitzt diese funktionelle Spaltenbeschreibung nicht. Dies wurde vorläufig mit einer RDD-Funktion implementiert, welche sich allerdings als ein *„Pipeline-Breaker“* erwies; *Pipelining* bedeutet „direktes Weiterreichen der einzelnen Tupeln ohne Zwischenspeichern“ [60]. Im Fall dieser selbstimplementierten Funktion wurde vom DataFrame-Format zu einem viel weniger

optimierten RDD umgeschaltet, eine *map*-Operation ausgeführt, und die Daten wurden anschließend aus RDD wieder zurück zu DataFrame konvertiert – Sparks Optimierer kam mit dem doppelten API-Wechsel nicht besonders gut zurecht. Dieser Aufwand war mit Zwischenspeicherung der Daten verbunden und kostete zwecks einer einzigen Index-Spalte unnötig Rechenzeit, weshalb dies in der neuen Version der Tabellenerzeugung entfernt wurde. Da Spark sowieso keine Vorteile durch Indexierung erringt, lässt sich Attribut TERM_NAME der Tabelle TERM als Primärschlüssel anstelle von TERM-_ID verwenden.

Auf der Suche nach möglichen Verbesserungen wurde ein anderer Ansatz herausgearbeitet, bei dem die Reihenfolge der Tabellenerzeugung sowie die Tabellenattribute von der Original-Variante abweichen. Diese andere Methode für die Vorberechnung sollte etwas kompakter gestaltet werden, und ihr Schwerpunkt fiel auf die für die Themenmodell-Anzeige (Front-End) wichtigen Eigenschaften.

## 3.2 Neugestaltung der Tabellenerzeugung

Im neuen Konzept der Vorberechnung wird genauso wie in der alten davon ausgegangen, dass die Quell-Tabellen aus der linguistischen Datenvorberechnung bereits gegeben sind. Die überarbeitete Tabellenerzeugung, die in der aktuellen Spark-Implementierung verwendet wird, kann in zwei Schritte aufgeteilt werden: in Vorwärts- und Rückwärtsschritt. Die richtungsbasierte Vorgehensweise ist vergleichbar mit dem Aufbau von Bayesschen Netzen. Ein Bayessches Netz ist ein gerichteter, azyklischer Graph mit Zufallsvariablen als Knoten und ZV-Abhängigkeiten als Kanten. Der tiefste Pfad geht vom äußeren Knoten aus (dem A-priori-Parametervektor für die Themenverteilung pro Dokument) zu dem tiefsten Knoten (einem Tokenvektor); empfehlenswert an dieser Stelle wäre ein Rückblick auf die Abbildung 2a, welche diese Erklärung zusätzlich schematisiert.

Ähnlich geht man im **Vorwärtsschritt** vor: dort werden die jeweiligen Tokenanzahlen (versehen mit einem bei diesem Ansatz universellen Attribut-Namen NUMBER_OF_TOKENS) für die Tabellen ermittelt. Später, im optionalen **Rückwärtsschritt**, werden diese Tokenanzahlen für probabilistische Berechnungen verwendet. Gemeint ist damit die Ermittlung von solchen statistischen Attributen wie (1) die Wahrscheinlichkeit, dass ein Thema einen bestimmten Begriff beinhaltet (PR_TOPIC_GIVEN_TERM), (2) die Wahrscheinlichkeit, mit der man im Rahmen eines bestimmten Themas auf einen bestimmten Begriff trifft (PR_TERM_GIVEN_TOPIC), oder (3-5) Wahrscheinlichkeiten, mit denen man die Verteilung von bestimmten Themen/Wörtern/Dokumenten im gesamten Korpus beschreibt (PR_TOPIC, PR_TERM, PR_DOCUMENT).

Die Reihenfolge der derartigen Tabellenerzeugung kann wie folgt dargestellt werden:

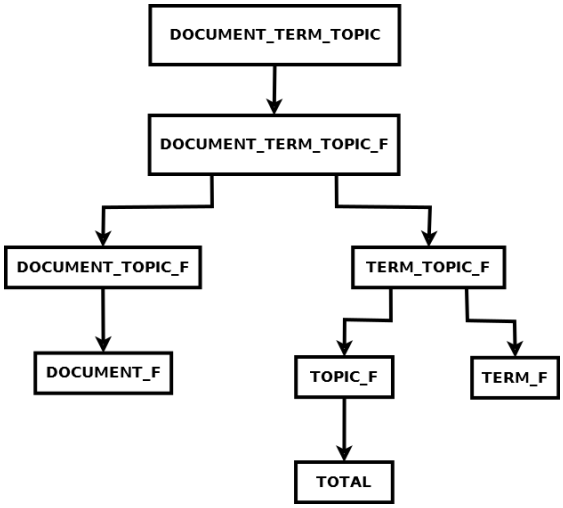



Abbildung 6: Überarbeitete Reihenfolge der Tabellenerzeugung aus den Grundtabellen

Als Tabellenattribute im neuen Ansatz dienen zum einen die Schlüssel (künstliche numerische oder natürliche Identifikatoren), und zum anderen die Tokenanzahl. Der Tabellenname selbst erläutert die *Beziehung*, welche die Tabelle definiert: wenn die Tabelle sich beispielsweise *DOCUMENT_TOPIC_F* nennt, dann stellt sie das Dokument-Themen-

Verhältnis dar. Deren Attribute DOCUMENT_ID und TOPIC_ID sind die Schlüssel, und das Attribut NUMBER_OF_TOKENS – die Anzahl der Token, welche mit dem Dokument mit der DOCUMENT_ID und dem Thema mit der TOPIC_ID in Verbindung stehen. Der Zusatz *F* in den Tabellennamen ist die Abkürzung für *Forwards*, als Hinweis auf den Vorwärtsschritt.

Die Grundtabelle *DOCUMENT_TERM_TOPIC* wird im Vorwärtsschritt etwas reduziert: gleiche Tokens innerhalb eines Dokuments werden zusammengefasst, und man erhält eine neue Grundtabelle *DOCUMENT_TERM_TOPIC_F*, mit der weiter gerechnet wird. Wenn man mehrere gleiche Tokens zusammenfassen, können sie allerdings nicht auf eine einzige Position zeigen, woraufhin dieses Attribut bei den Tokens wegfallen muss. Dadurch bildet sich eine neue, kompaktere (ohne Positionsangabe) Token-Darstellung TOKEN_F heraus (siehe Abb. 7). Die hinzugekommene Tabelle TOTAL beinhaltet nur ein Attribut und ein Tupel – die Gesamtanzahl der Token für die ganze Dokumentensammlung (NUMBER_OF_TOKENS), damit man später bei Bedarf die Wahrscheinlichkeiten ausrechnen kann. Die Wahrscheinlichkeitsattribute sind in folgender Abbildung nicht wiedergegeben, weil der Rückwärtsschritt zur Ermittlung dieser Angaben optional ist.

Folgendes Entity-Relationship-Diagramm beschreibt den neuen Aufbau der Datenbankstruktur:

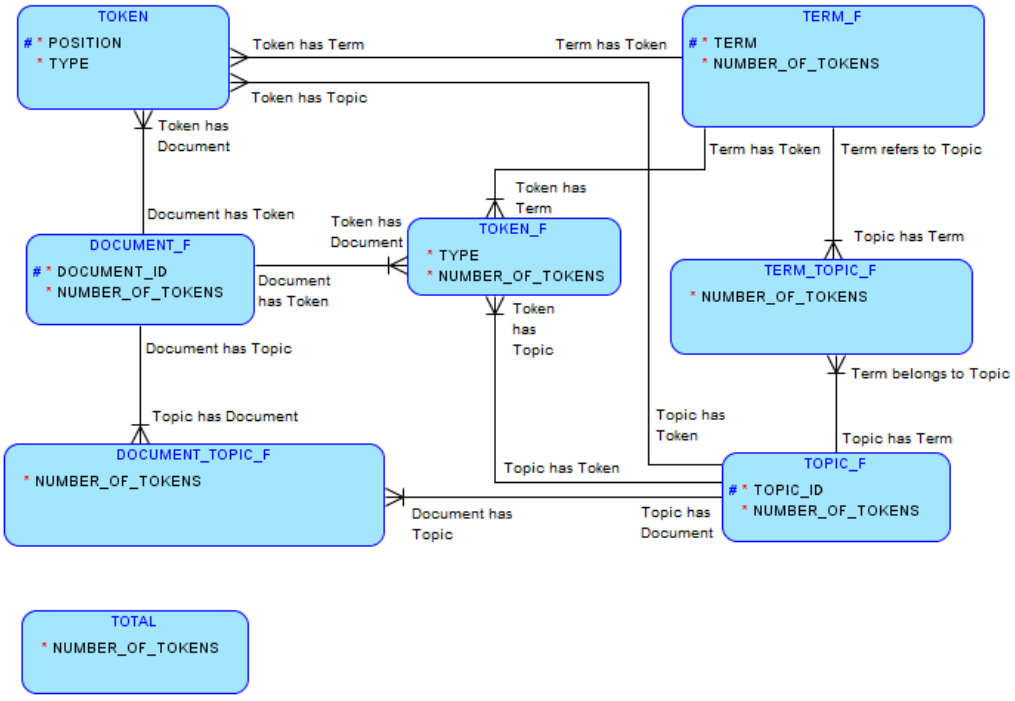



Abbildung 7: Entity-Relationship-Diagramm zum neugestalteten Datenbank-Aufbau

Warum können die Informationen (wie Häufigkeiten oder Wahrscheinlichkeiten), die beim ursprünglichen Ansatz in den Tabellen gespeichert werden (und beim neuen Ansatz im Rückwärtsschritt optional hinzukommen können), nur allein durch die Tokenanzahl ersetzt werden?

Die kurze Antwort auf diese Frage: auf diese Weise erhält man schnell nötige Informationen für die Themen- bzw. Wort-Ranglisten, die schließlich dem Nutzer im Front-End präsentiert werden.

Diese Erklärung lässt sich selbstverständlich ausbauen. Das Attribut für die probabilistische Angabe für die Themensortierung nennt sich in der alten Version PR_TOPIC_GIVEN_TERM (die Wahrscheinlichkeit, dass ein bestimmter Begriff einem bestimmten Thema zuzuordnen ist). Sie rechnet man wie folgt aus (Thema $T$ und Begriff $B$ sind für die jeweilige Wahrscheinlichkeit $P(T \mid B)$ fest):

$$\text{PR\_TOPIC\_GIVEN\_TERM} = P(T \mid B) = \frac{\mid K_{T \cap B} \mid}{\mid K_B \mid}$$

$\mid K_{T \cap B} \mid$ ist die Anzahl der Token mit Thema $T$ und Begriff $B$, $\mid K_B \mid$ ist die Anzahl aller dem Begriff $B$ zuzuordnenden Token.

Das Attribut zur späteren Wörtersortierung spiegelt die Wahrscheinlichkeit wieder, mit der man im Rahmen eines bestimmten Themas auf einen bestimmten Begriff trifft. Sie erhält man mit

$$\text{PR\_TERM\_GIVEN\_TOPIC} = P(B \mid T) = \frac{\mid K_{T \cap B} \mid}{\mid K_T \mid}$$

Diese Formeln verkörpern den allgemeinen Ansatz für die probabilistischen Rechenoperationen in der Datenbank, der bei der alten Herangehensweise sowie in der überarbeiteten Version auf den Plan tritt: eine Wahrscheinlichkeit kann als ein Bruch dargestellt werden, wo oben im Nenner ein spezifischer Ausdruck mit mehr Parametern steht, und unten im Zähler – eine weniger spezifische Bezeichnung.

Zwecks eines Beispiels mit echten Daten könnte man den Begriff „Regierungschef“ aus dem TopicExplorer-Testdatensatz nehmen. Dieses Wort kommt in den Themen Nr. 2 und 29 vor, wir beschränken uns hierbei nur auf das Thema Nr. 29. Die sogenannte CORPUS_FREQUENCY von dem Begriff „Regierungschef“ beträgt 65 Token – auf dieses Wort trifft man im ganzen Dokumentensatz nämlich genau 65 Mal. Ferner gehören insgesamt 7567 Token zum Thema Nr. 29, davon stehen 48 Token in Verbindung sowohl mit dem Begriff „Regierungschef“ als auch mit dem Thema 29. Diese Informationen reichen aus, um PR_TOPIC_GIVEN_TERM zu erhalten: **48** – die Anzahl der Token mit Thema 29 und Begriff „Regierungschef“ (alias der spezifische Ausdruck und eine Untermenge) – dividiert man durch **65**, die Anzahl aller dem Begriff „Regierungschef“ zugewiesenen Token (alias eine Obermenge, beschrieben durch den unspezifischeren Ausdruck). Dadurch, dass die im Zähler beschriebene Menge eine Untermenge der im Nenner beschriebenen Menge ist, kann solch ein Bruch (und entsprechend auch eine Wahrscheinlichkeit) nie größer sein als 1.

Wie bereits erwähnt, dienen die probabilistischen Angaben PR_TOPIC_GIVEN_TERM und PR_TERM_GIVEN_TOPIC zur Themen- bzw. Wörter-*Sortierung*. Die

Formeln für diese Angaben lassen in diesem Zusammenhang beobachten, dass man beim Sortieren der Wörter oder Themen nach den Wahrscheinlichkeiten (also nach kompletten Brüchen) die gleiche Reihenfolge erhält wie beim Sortieren nur nach den Token-Anzahlen (d.h. nur nach den Bruchzählern). Der Grund hierfür ist die Tatsache, dass die jeweilige Obermenge im Nenner immer dieselbe für alle Untermengen im Bruchzähler bleibt. Eine Rolle für die Rangordnung der Themen bzw. Wörter (eng. *„Rankings“*) spielt nur die *Reihenfolge* der zugehörigen Wahrscheinlichkeitsangaben; die konkreten Wahrscheinlichkeiten hingegen eher weniger. Deshalb kann man bei gleicher Sortierreihenfolge auch nur die Tokenanzahlen nehmen (d.h. nur die Zahlen im Bruchzähler). Den Rückwärtsschritt, wo die anschließende Division durch den Bruchnenner durchgeführt wird, kann man daher theoretisch auch weglassen, solange man sich nur auf die „Rankings“/Ranglisten der Themen und Wörter sowie ihre grafische Darstellung konzentriert.

Sollten die konkreten Wahrscheinlichkeiten doch gebraucht und zu einem bestimmten Zweck ausgewertet werden, kann man den Rückwärtsschritt hinzunehmen und diese noch optional ermitteln. Dazu werden die Informationen zur Tokenanzahl in den jeweiligen Tabellen benötigt; diese liegen nach dem Vorwärtsschritt schon vor.

Sowohl für die ursprüngliche als auch für die *Forwards*-Herangehensweise wurden nach der Optimierung die Rechenzeiten gemessen und miteinander vergleichen. Im nächsten Kapitel wird es darum gehen, welche Faktoren bei der Optimierung berücksichtigt wurden.

## 3.3 Optimierung der TopicExplorer-Vorberechnung

In Spark existieren zurzeit nur wenige Verfahren, die Operationen an Tabellen zu beschleunigen (Stand: Spark-Version 1.6.1 [61]):

- Nutzung des Tungsten-Back-Ends [62] (eine Alternative zum DataFrame-Back-End, Hauptziel: Optimierung der CPU-Effizienz), standardmäßig eingeschaltet
- Änderung der Anzahl von Partitionen für Datenaustausch-Operationen (bei Joins oder Aggregationen)
- Nutzung des Hauptspeichers zur Ablage von Tabellen („Caching“)

Es ist möglich, dass die ersten zwei Verfahren in den zukünftigen Spark-Versionen automatisiert werden könnten [61], und sie werden fortan nicht näher erläutert. Der letzte Ansatz hingegen bedarf näherer Betrachtung.

**Methoden für Tabellenmaterialisierung** Das Abspeichern von Tabellen im Hauptspeicher erfolgt, wie bereits angesprochen, durch manuelle (d.h. durch den Programmierer) Festlegung der Cache-Punkte. Man setzt einen Cache-Punkt durch das Hinzufügen von einem entsprechenden Befehl in den Programmcode. So nimmt der Programmierer die Speicherorganisation für Tabellen durch das Caching-Management selbst in die Hand. Die Performanz der Tabellenberechnung hängt dabei von der Geschicktheit der Setzung von Cache-Punkten ab.

Zur Setzung der Cache-Punkte gibt es die Methode *cacheTable("Tabellenname")* aus Sparks Modul *sqlContext*. Alternativ geschieht dies durch den Aufruf *d.cache()* für ein DataFrame *d* mit der zu materialisierenden Tabelle, ansonsten ließe sich die Methode *cache()* auch mit *persist(StorageLevel.MEMORY_ONLY)* ersetzen. Die Methode *persist* erlaubt die Festlegung der Caching-Speicherebene („Storage Level“); die Variante MEMORY_ONLY ist nur eine der vielen Möglichkeiten.

Spark verfügt zum Caching-Zweck über folgende Speicherebenen:

| **Storage Level** | **Bedeutung** |
|---|---|
| MEMORY_ONLY | Standard-Speicherebene (wird bei der Methode *cache()* verwendet); besitzt die beste CPU-Effizienz. Objekte werden innerhalb einer Java Virtual Machine nicht-serialisiert untergebracht. Sollte ein Objekt nicht vollständig in den Hauptspeicher reinpassen, werden die nicht gecachten Partitionen bei Bedarf neu berechnet. |

| | |
|---|---|
| MEMORY_AND_DISK | Objekte werden nicht-serialisiert untergebracht. Sollte ein Objekt nicht in den Hauptspeicher reinpassen, werden die nicht gecachten Partitionen davon auf die Festplatte geschrieben und bei Bedarf von dort ausgelesen. |
| MEMORY_ONLY_SER | Objekte werden serialisiert gespeichert (ein Byte-Array pro Partition). Platzsparender als ohne Serialisierung, jedoch mit mehr Rechenaufwand auf CPU-Kosten. |
| MEMORY_AND_DISK_SER | ähnlich zu MEMORY_ONLY_SER, aber mit Verlagerung der nicht in den Hauptspeicher passenden Partitionen auf die Festplatte, anstatt ihrer Neuberechnung bei Bedarf. |
| DISK_ONLY | Speicherung nur auf der Festplatte. |
| MEMORY_ONLY_2, MEMORY_AND_DISK_2 usw. | gleiches Prinzip wie bei den Versionen ohne 2 im Namen, hierbei wird aber jede Partition des Objekts auf zwei Knoten des Clusters redundant abgespeichert. |
| OFF_HEAP | ähnlich zu MEMORY_ONLY_SER, die Unterbringung der Daten erfolgt allerdings im Off-Heap-Speicher. Zurzeit eine experimentelle Speicherebene. |

**Auswirkungen vom Caching** Aufgrund des von Spark genutzten Prinzips der „faulen Auswertung“ ist ein Cache-Befehl nur ein zusätzlicher Vermerk im internen Ausführungsplan. Allein durch solchen einen Befehl wird keine Materialisierung erzwungen. Erst wenn das Programm an einer Stelle ankommt, an der die zum Cachen vermerkte Tabelle wirklich materialisiert werden muss (z.B. konkrete Tabellenwerte werden abgerufen, oder die Tabelle soll exportiert werden), wird die genannte Tabelle ausgerechnet, komprimiert und im Hauptspeicher abgelegt. So muss diese Tabelle nicht bei jedem Abruf der darin enthaltenen Daten immer wieder ausgerechnet werden – sie liegt nach dem Caching solange im Hauptspeicher, bis sie entweder automatisch verdrängt bzw. manuell

aus dem Cache wieder rausgenommen wird, oder bis die Anwendung beendet wird.

Diejenigen DataFrames/Tabellen, die innerhalb einer Anwendung mehrfach wiederverwendet werden, sollen zur Beschleunigung der Anwendung und Reduzierung des Rechenaufwands in den Hauptspeicher geladen werden. Ein anderer, im Cluster-Betrieb wichtiger Anwendungsfall für das Caching ist das *Absichern der Ergebnisse teurer Rechenoperationen*, sodass die Daten nicht neu berechnet werden müssen bzw. viel leichter wiederhergestellt werden können, falls eine Betriebsstörung an einem Cluster-Rechenknoten auftritt. Infolge der Materialisierung einer Tabelle wird die Datenherkunft automatisch vermerkt. Sollte ein Knoten im Cluster-Betrieb Störungen aufweisen, können die dadurch verlorenen Daten mit Hilfe der Datenherkunftsinformationen leichter wiederhergestellt werden, indem sie nach den Herkunftsplänen neu berechnet werden [63].

**Automatische und manuelle Hauptspeicherverwaltung** Spark überwacht die Cache-Nutzung (im Cluster-Betrieb – an jedem Cluster-Knoten), und gibt bei Bedarf den Hauptspeicher nach dem „Am längsten nicht verwendet"-Prinzip (LRU, least-recently-used) frei [64], solange sich der Cache-Speicher nur auf die *MEMORY_ONLY*-Ebene (mit oder ohne Optionen wie Serialisierung bzw. Redundanz) beschränkt. Der für die Schreib- und Lesezugriffe schnelle Arbeitsspeicher kann bei einer Big-Data-Anwendung wie TopicExplorer schnell voll werden, woraufhin die lange nicht verwendeten Daten entweder verworfen (standardmäßiger Vorgang) oder auf die Festplatte geschrieben werden müssen (z.B. in der MEMORY_AND_DISK-Ebene). Unter Umständen soll der Programmierer selbst dafür sorgen, dass der Hauptspeicher genug Platz für wichtige Tabellen bietet, indem die nicht mehr benötigten Daten rechtzeitig aus dem Cache rausgenommen werden. Hierzu ein Beispiel, welches für die MEMORY_ONLY-Speicherebene zutreffen würde: eine große Tabelle wird kurz nach dem Anwendungsstart ausgerechnet und gecacht, weil sie am Ende noch benötigt wird. Dabei soll definitiv vermieden werden, dass sie zwischen Anwendungsstart und -Ende aufgrund der Speicherdefizite aus dem Cache verdrängt wird – sonst müsste sie am Ende nochmal materialisiert werden, was unnötige Rechenzeit kosten würde. Mit der MEMORY_AND_DISK-Tabellenpersistenz wäre das Platzproblem in allermeisten Fällen zwar nicht mehr von Bedeutung, weil die Daten, welche nicht mehr in den Hauptspeicher reinpassen, auf die Festplatte geschrieben worden wären. Doch leider kosten die Festplattenzugriffe viel mehr Zeit als die Hauptspeicher-Zugriffe, und sollten daher vermieden werden – so wäre die Auswahl der MEMORY_AND_DISK-Speicherebene keine hervorragende Lösung für das Problem in diesem Beispiel. In solchen Sonderfällen wie diesem sollte man als Programmierer bestenfalls selbst für ausreichend Platz im Speicher sorgen, indem man versucht, die Tabellen, sobald sie nach mehrmaliger Wiederverwendung nicht nochmal genutzt werden, aus dem Hauptspeicher verwerfen zu lassen.

**Gewählte Caching-Strategie** Die Vorberechnung für TopicExplorer ist aufgrund dieser Überlegungen nach folgender MEMORY_ONLY-Caching-Strategie ausgelegt: wenn die Tabelle mehr als einmal verwendet wird, soll sie materialisiert werden, und, sobald sie definitiv nicht mehr gebraucht wird, wird die Tabelle aus dem Cache manuell

verworfen (wobei die automatische LRU-Verdrängungsstrategie trotzdem weiterhin bestehen bleibt). Die Tabellen, welche nur *einmal* wiederverwendet werden, werden nicht im Cache zwischengespeichert, sondern bei Bedarf einmalig nach dem Prinzip der „faulen Auswertung“ ausgerechnet – die Wahrscheinlichkeit. Der Catalyst-Optimierer soll in beiden von diesen Fällen eine Chance haben, Optimierungen durch das Überspringen und/oder Zusammenführen von Rechenoperationen im Ausführungsplan zu leisten. Diese wörtlich formulierte Caching-Strategie wurde in beiden Vorberechnungsansätzen in Spark verwendet. Leider führte das durch den Befehl *uncache* explizite „Entmaterialisieren“ von Tabellen zu einem Fehler, dessen Ursprung (noch) unklar ist – dazu muss aber gesagt werden, dass dies die Arbeit nicht gehindert hat: der Testserver besaß auch für die Abarbeitung des großen Test-Datensatzes genug Hauptspeicher, und der im letzten Beispiel beschriebene Sonderfall drohte nicht vorzukommen.

Zuletzt sollte angemerkt werden, dass das Materialisieren von Tabellen, welche oft wiederverwendet werden (z.B. DOCUMENT_TERM_TOPIC, TERM_TOPIC), in erwarteter Weise die am stärksten maßgebenden Geschwindigkeitsoptimierungen mit sich brachte.

## 3.4 Zeitvergleiche

Die Auswertung der Rechenzeiten, welche Spark und MySQL für die Vorberechnung benötigen, veranschaulicht die Unterschiede zwischen diesen beiden Back-Ends und legt den Grundstein zum Fazit dieser Bachelorarbeit.

**Technische Gegebenheiten** Die Zeitvergleiche wurden auf einem Linux-Server mit 32 Prozessorkernen und 50 GB Arbeitsspeicher durchgeführt. Da dies nur ein einziger Rechner ist, beziehen sich alle Ergebnisse auf einen *pseudo-cluster*-Betriebsmodus, d.h. mit einem einzigen Knoten ist kein echter Cluster gegeben, und der Datenaustausch findet innerhalb eines Systems statt – somit bleibt die Untersuchung der horizontalen Skalierung (*scale out*) für TopicExplorer-Zwecke nicht getestet. Im echten Cluster (d.h. mit paralleler Berechnung auf mehreren Rechnerknoten) käme mit der Inter-Knoten-Kommunikation noch der Faktor der Netzwerkbelastung hinzu.

**MySQL vs. Spark** In dieser Tabelle sind Zeitangaben zusammengefasst, die durch Experimente mit verschiedenen Vorberechnungsansätzen auf verschieden großen Datensätzen entstanden sind.

Die Messungen mit dem ursprünglichen Ansatz und verschiedenen Back-Ends (inkl. Startzeit für die MySQL- bzw. Spark-Shell):

| | MySQL | Spark |
|---|---|---|
| 2721 Artikel | 152 s | 378 s |
| 268861 Artikel | 39498 s | 3723 s |

Anhand dieser Tabelle wird deutlich, dass mit Spark auf großen Daten ein sehenswerter Zeitgewinn erzielt wurde. Ersichtlich wird außerdem, dass sich Apache Spark nicht gut für die Auswertung kleiner Datensätze geeignet ist: allein das Laden der Spark-Shell dauert spürbar lange (eine bis anderthalb Minuten), weshalb die Umstellung auf Spark keine oder kaum Zeitersparnisse bei wenigen auszuwertenden Daten bringen und sich deshalb nicht lohnen würde. Wichtig ist hier die Anmerkung, dass bei diesem Vergleich zwischen MySQL und Spark der ursprüngliche Vorberechnungsansatz (ohne Vorwärtsschritt) genutzt wurde. Der neue Ansatz (mit Vorwärtsschritt) hingegen ist zum aktuellen Zeitpunkt noch nicht vollständig umgesetzt; somit ist ein vollwertiger Vergleich der ursprünglichen deklarativen Problemstellung und der neuen Herangehensweise zum aktuellen Zeitpunkt noch nicht möglich, weil andere TopicExplorer-Bestandteile (wie die Zeitkomponente und die Themenhierarchie-Berechnung) an den neuen Ansatz noch angepasst werden müssten.

Um trotzdem einen Vergleich zwischen den beiden Spark-basierten Vorberechnungskonzepten zu ermöglichen, wurden am alten Ansatz folgende temporäre Änderungen vorgenommen:

- Die SQL-Statements wurden fair gestaltet, d.h. die Berechnungen der Wahrscheinlichkeiten und die damit einhergehenden JOIN-Operationen wurden rausgenommen, weil diese im Vorwärtsschritt nicht geschehen. Prinzipiell beschränkte man

sich auf die inneren Statements, also auf die Unterabfragen, die das Gerüst der Abfragen im alten Ansatz bilden.

- Die Zeit-Komponente (orig. Name: „time preprocessing“; sie berechnet, welche Wörter pro welches Thema in welcher Woche am häufigsten vorkommen) wurde ebenfalls rausgekürzt, da es von ihr noch keine an den Vorwärtsschritt angepasste Version gibt.
- Die Themenhierarchie-Berechnung fiel weg, aus dem gleichen Grund wie die Zeitkomponente (fehlende Neu-Anpassung).

Unter der Berücksichtigung dieser Rahmenbedingungen stellten sich experimentell folgende Rechenzeiten heraus:

| | ursprünglich* | neu (mit Vorwärtsschritt) |
|---|---|---|
| 2721 Artikel | 111 s | 166 s |
| 268861 Artikel | 241 s | 283 s |

*** Anmerkung:** mit oben beschriebenen Änderungen.

Aus diesen Werten lässt sich ableiten, dass der ursprüngliche Ansatz verglichen mit der neuen Version schneller oder gleichschnell (theoretisch könnten die Systemprozesse die Zeitmessung geringfügig beeinflusst haben) ist, weshalb das Nebeneinanderstellen von diesen zwei Herangehensweisen zur TopicExplorer-Vorberechnung einige offene Fragen aufbringt. Zu untersuchen gälte noch folgendes:

- Woran liegt die gleiche bzw. schlechtere Leistung mit mit dem neuen Ansatz gegenüber dem alten? Liegt es an der geänderten Reihenfolge der Tabellenerzeugung, weshalb vielleicht mehr Rechenarbeit nötig ist, oder gibt es hierfür andere Gründe?
- Wie schnell würden sich mit dem neuen Ansatz die rechenintensiven Zeitkomponente und Themenhierarchie ausrechnen lassen, falls sie daran noch angepasst werden würden/könnten? Würden sie dann den Zeitvergleich zugunsten des neuen Ansatzes umschwenken, oder die Überlegenheit der alten Technik zusätzlich verstärken?
- Ließe sich der neue Ansatz ohne Potenzialverlust (also ohne Wegschneiden von nützlicher Information aus dem Schema) so modifizieren, dass er schneller als der ursprüngliche wird?

Bis diese Fragen geklärt sind, sollte der bereits fertige ursprüngliche Art für die TopicExplorer-Vorberechnung genutzt werden, da sie zum aktuellen Zeitpunkt die besten Ergebnisse liefert und die meiste Funktionalität unterstützt.

# 4 Fazit

**Neuerungen** Die interne Struktur wurde weitgehend überarbeitet, was sowohl den Quell-Code als auch die Anfragen betrifft. Sämtliche textuelle Teile wurden deutlich übersichtlicher gestaltet; SQL wurde vom sonstigen Code (Scala, R) getrennt, und die Abfragen sind über einzelne Textdateien zu erreichen. Die früher aus den for-Schleifen heraus aufgerufenen SQL-Statements wurden umformuliert und brauchen keine Schleifen mehr. Die Tabellenaktualisierungen sind weggefallen: Tabellen mussten persistent gestaltet werden, da man in Spark die bestehenden Tabellen nicht mehr ändern kann. Die MySQL-spezifische Optimierung in Form von Indexierung wurde entfernt aufgrund der Überfälligkeit in Spark, und wurde durch eine andere Art der Optimierung – das Setzen der Cache-Punkte – ersetzt. Statt eines Java-Skripts zum Einleiten der Vorberechnung ist nun ein Scala-Skript hierfür zuständig.

**Zukünftige Arbeit** Ein Flaschenhals, welcher zur Vorberechnung mit dem neuen Back-End nicht hinzukam, ist die Implementierung der linguistischen „Frames“ (mehr zu diesem Konzept in [65]). Diese Implementierung war schon von der Arbeit mit Spark angestrebt. MySQL schien jedoch die Aufgabe nicht bzw. sehr ineffizient zu bewältigen. Spark bietet in dieser Hinsicht gute Aussichten, und das wäre eine mögliche Richtung für die weitere TopicExplorer-Entwicklung.

Ein weiterer Punkt betrifft die zwei Techniken für die TE-Umsetzung mit Spark. Die offenen Fragen aus dem vorhergehenden Kapitel sollen im diesem Bezug noch beantwortet werden, um eine Entscheidung darüber zu treffen, ob man sich für die neue, experimentelle Herangehensweise mit dem Vorwärtsschritt entscheiden sollte. Falls dafür entschieden wird, müssten noch die Komponente für die zeitliche Themenentwicklung sowie die Themenhierarchie-Berechnung angepasst werden.

Zuletzt sollte für die zukünftige Arbeit erwähnt werden, dass eine Aktualisierung von Spark auf die Version 2.0 oder höher (aktuelle Version auf dem Server: 1.6.1) günstig für die weitere Entwicklung wäre, weil man damit eine zusätzliche Beschleunigung der SQL-Abfragen erwartet und die Vorteile des unterstützten SQL-2003-Standards nutzen könnte. Zu beachten ist allerdings, dass diese Aktualisierung mit einige API-Änderungen einhergeht – diese wären zu berücksichtigen.

**Schlussbetrachtung** Als Ziel dieser Arbeit galt ein deutlicher Zeitgewinn bei den TopicExplorer-Vorberechnungen mit Hilfe von Apache Spark. Die Deklarativität der Implementierung musste dabei beibehalten werden. Dieses Ziel ist erreicht, was der vorhergehende Zeitvergleich bestätigt: die TE-Vorberechnung auf großen Daten benötigt nur etwa ein Zehntel der Zeit, die sie vor der Spark-basierten Umsetzung gebraucht hat.

Die Umstellung des Back-Ends von TopicExplorer von MySQL auf Apache Spark ist nahezu vollständig gelungen, da die meisten „Flaschenhälse“ in der TE-Vorberechnung beseitigt wurden. Der deklarative Aufbau von TopicExplorer hat im Laufe der Arbeit am Back-End-Tausch als vorteilhaft herausgestellt, was für dieses Aufbaukonzept spricht. Dieser Topic-Model-Browser erringt nun mit der signifikant schnelleren Spark-basierten Vorberechnung noch mehr Potenzial gegenüber seiner Konkurrenz.

## Literatur

[1] InternetLiveStats.com. 2016. URL: http://www.internetlivestats.com (siehe S. 1).

[2] David M. Blei. „Probabilistic Topic Models“. In: *Commun. ACM* 55.4 (Apr. 2012), S. 77–84. ISSN: 0001-0782. DOI: 10.1145/2133806.2133826. URL: http://doi.acm.org/10.1145/2133806.2133826 (siehe S. 1, 6, 8, 9, 10, 11).

[3] Frans Coenen. *2CS24 – TOPICS IN INFORMATION PROCESSING: DECLARATIVE LANGUAGES*. 1999. URL: http://cgi.csc.liv.ac.uk/~frans/OldLectures/2CS24/declarative.html#detail (siehe S. 2).

[4] P.C.Y. Sheu. *Software Engineering and Environment: An Object-Oriented Perspective*. Software Science and Engineering. Springer US, 2012. ISBN: 9781461559078. URL: https://books.google.de/books?id=KN%5C_vBwAAQBAJ (siehe S. 2).

[5] Alexander Shkapsky u. a. „Big Data Analytics with Datalog Queries on Spark“. In: *Proceedings of the 2016 International Conference on Management of Data*. SIGMOD '16. San Francisco, California, USA: ACM, 2016, S. 1135–1149. ISBN: 978-1-4503-3531-7. DOI: 10.1145/2882903.2915229. URL: http://doi.acm.org/10.1145/2882903.2915229 (siehe S. 2, 18, 19, 20).

[6] Alexander Rubin. *Using Apache Spark and MySQL for Data Analysis*. 2015. URL: https://www.percona.com/blog/2015/10/07/using-apache-spark-mysql-data-analysis/ (siehe S. 2).

[7] Duden. *Bedeutung des Wortes „Funktionswort/Synsemantikon“*. 2016. URL: http://www.duden.de/rechtschreibung/Synsemantikon (siehe S. 2).

[8] Alexander Hinneburg, Rico Preiss und Rene Schroeder. „TopicExplorer: Exploring Document Collections with Topic Models“. In: *Proceedings of the 2012 European Conference on Machine Learning and Knowledge Discovery in Databases - Volume Part II*. ECML PKDD'12. Bristol, UK: Springer-Verlag, 2012, S. 838–841. ISBN: 978-3-642-33485-6. DOI: 10.1007/978-3-642-33486-3_59. URL: http://dx.doi.org/10.1007/978-3-642-33486-3_59 (siehe S. 4).

[9] Eric Alexander u. a. „Serendip: Topic model-driven visual exploration of text corpora“. In: *Visual Analytics Science and Technology (VAST), 2014 IEEE Conference on*. Okt. 2014, S. 173–182. DOI: 10.1109/VAST.2014.7042493 (siehe S. 4).

[10] Carson Sievert und Kenneth Shirley. „LDAvis: A method for visualizing and interpreting topics“. In: *Proceedings of the Workshop on Interactive Language Learning, Visualization, and Interfaces*. Baltimore, Maryland, USA: Association for Computational Linguistics, Juni 2014, S. 63–70. URL: http://www.aclweb.org/anthology/W/W14/W14-3110 (siehe S. 4).

[11] The Indiana Philosophy Ontology project. *InPho Topic-Explorer*. 2015. URL: http://inphodata.cogs.indiana.edu (siehe S. 4).

[12] Shixia Liu u. a. „Interactive, Topic-based Visual Text Summarization and Analysis“. In: *Proceedings of the 18th ACM Conference on Information and Knowledge Management*. CIKM '09. Hong Kong, China: ACM, 2009, S. 543–552. ISBN: 978-1-60558-512-3. DOI: 10.1145/1645953.1646023. URL: http://doi.acm.org/10.1145/1645953.1646023 (siehe S. 4).

[13] Guodao Sun u. a. „EvoRiver: Visual Analysis of Topic Coopetition on Social Media“. In: *Visualization and Computer Graphics, IEEE Transactions on* 20.12 (Dez. 2014), S. 1753–1762. ISSN: 1077-2626. DOI: 10.1109/TVCG.2014.2346919 (siehe S. 4).

[14] Weiwei Cui u. a. „How Hierarchical Topics Evolve in Large Text Corpora“. In: *Visualization and Computer Graphics, IEEE Transactions on* 20.12 (Dez. 2014), S. 2281–2290. ISSN: 1077-2626. DOI: 10.1109/TVCG.2014.2346433 (siehe S. 4).

[15] Dr. Michael Derntl. *D-VITA: Dynamic Visual Topic Analytics*. 2016. URL: http://dbis.rwth-aachen.de/cms/research/ACIS/D-VITA (siehe S. 4).

[16] Shixia Liu u. a. „TopicPanorama: A full picture of relevant topics“. In: *Visual Analytics Science and Technology (VAST), 2014 IEEE Conference on*. Okt. 2014, S. 183–192. DOI: 10.1109/VAST.2014.7042494 (siehe S. 5).

[17] N. Jojic, A. Perina und D. Kim. „Hierarchical learning of grids of microtopics“. In: *ArXiv e-prints* (März 2015). arXiv: 1503.03701 [stat.ML] (siehe S. 5).

[18] Wenwen Dou u. a. „HierarchicalTopics: Visually Exploring Large Text Collections Using Topic Hierarchies“. In: *Visualization and Computer Graphics, IEEE Transactions on* 19.12 (Dez. 2013), S. 2002–2011. ISSN: 1077-2626. DOI: 10.1109/TVCG.2013.162 (siehe S. 5).

[19] Christoph Carl Kling u. a. „Topic Model Tutorial: A Basic Introduction on Latent Dirichlet Allocation and Extensions for Web Scientists“. In: *Proceedings of the 8th ACM Conference on Web Science*. WebSci '16. Hannover, Germany: ACM, 2016, S. 10–10. ISBN: 978-1-4503-4208-7. DOI: 10.1145/2908131.2908142. URL: http://doi.acm.org/10.1145/2908131.2908142 (siehe S. 6).

[20] Bela A. Frigyik, Amol Kapila und Maya R. Gupta. *Introduction to the Dirichlet Distribution and Related Processes*. Techn. Ber. 2010 (siehe S. 8).

[21] David M. Blei, Andrew Y. Ng und Michael I. Jordan. „Latent Dirichlet Allocation“. In: *J. Mach. Learn. Res.* 3 (März 2003), S. 993–1022. ISSN: 1532-4435. URL: http://dl.acm.org/citation.cfm?id=944919.944937 (siehe S. 8).

[22] Scott C. Deerwester u. a. „Indexing by Latent Semantic Analysis“. In: *JASIS* 41.6 (1990), S. 391–407. DOI: 10.1002/(SICI)1097-4571(199009)41:6<391::AID-ASI1>3.0.CO;2-9. URL: http://dx.doi.org/10.1002/(SICI)1097-4571(199009)41:6%3C391::AID-ASI1%3E3.0.CO;2-9 (siehe S. 9).

[23] Wray L. Buntine. „Variational Extensions to EM and Multinomial PCA“. In: *Machine Learning: ECML 2002, 13th European Conference on Machine Learning, Helsinki, Finland, August 19-23, 2002, Proceedings*. 2002, S. 23–34. DOI: 10.1007/3-540-36755-1_3. URL: http://dx.doi.org/10.1007/3-540-36755-1_3 (siehe S. 9).

[24] Wray Buntine und Aleks Jakulin. „Discrete Component Analysis“. In: *Proceedings of the 2005 International Conference on Subspace, Latent Structure and Feature Selection*. SLSFS'05. Bohinj, Slovenia: Springer-Verlag, 2006, S. 1–33. ISBN: 3-540-34137-4, 978-3-540-34137-6. DOI: 10.1007/11752790_1. URL: http://dx.doi.org/10.1007/11752790_1 (siehe S. 9).

[25] Mark Steyvers und Tom Griffiths. „Probabilistic Topic Models“. In: *Latent Semantic Analysis: A Road to Meaning*. Hrsg. von T. Landauer u. a. Laurence Erlbaum, 2006. URL: http://cocosci.berkeley.edu/tom/papers/SteyversGriffiths.pdf (siehe S. 9).

[26] Michael I. Jordan u. a. „An Introduction to Variational Methods for Graphical Models“. In: *Machine Learning* 37.2 (1999), S. 183–233. DOI: 10.1023/A:1007665907178. URL: http://dx.doi.org/10.1023/A:1007665907178 (siehe S. 9).

[27] Martin J. Wainwright und Michael I. Jordan. „Graphical Models, Exponential Families, and Variational Inference“. In: *Foundations and Trends in Machine Learning* 1.1-2 (2008), S. 1–305. DOI: 10.1561/2200000001. URL: http://dx.doi.org/10.1561/2200000001 (siehe S. 9).

[28] Till Decker. „Datenklassifikation mittels Bayestechniken“. Diplomarbeit. Technische FH Berlin, 2005. URL: http://www.itwm.fraunhofer.de/fileadmin/ITWM-Media/Abteilungen/BV/Pdf/Abschlussarbeiten/Diplomarbeit_Decker.pdf (siehe S. 9).

[29] Kevin Murphy. *A Brief Introduction to Graphical Models and Bayesian Networks*. 1998. URL: http://www.cs.ubc.ca/~murphyk/Bayes/bnintro.html (siehe S. 11).

[30] M.I. Jordan. *Learning in Graphical Models*. Adaptive computation and machine learning. London, 1998. ISBN: 9780262600323. URL: https://books.google.de/books?id=zac7L4LbNtUC (siehe S. 11).

[31] G. Carenini, R. Ng und G. Murray. *Methods for Mining and Summarizing Text Conversations*. Synthesis digital library of engineering and computer science. Morgan & Claypool, 2011. ISBN: 9781608453900. URL: https://books.google.de/books?id=oSWizUw%5C_oxgC (siehe S. 11).

[32] D.L. Poole und A.K. Mackworth. *Artificial Intelligence: Foundations of Computational Agents*. Artificial Intelligence: Foundations of Computational Agents. Cambridge University Press, 2010. ISBN: 9780521519007. URL: https://books.google.de/books?id=eALhh%5C_tkpv4C (siehe S. 11).

[33] Frank Rosner. „Integrating Probabilistic and Database Models for Rapidly Building Customized Machine Learning Applications“. Thesis submitted in partial fulfillment of the requirements for the degree Master of Science. Martin-Luther-University Halle-Wittenberg – Faculty of Law, Economics und Business, 2014 (siehe S. 11, 12).

[34] Andy Grove. *Apache Spark: RDD, DataFrame or DataSet?* 2016. URL: http://www.agildata.com/apache-spark-rdd-vs-dataframe-vs-dataset/ (siehe S. 14, 16).

[35] The Apache Software Foundation. *Apache Spark project*. 2016. URL: http://spark.apache.org (siehe S. 14).

[36] Michael Armbrust u. a. „Spark SQL: Relational Data Processing in Spark“. In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*. 2015, S. 1383–1394. DOI: 10.1145/2723372.2742797. URL: http://doi.acm.org/10.1145/2723372.2742797 (siehe S. 14, 15, 16, 17, 18).

[37] The Apache Software Foundation. *Spark 2.0 Release Notes*. 2016. URL: http://spark.apache.org/releases/spark-release-2-0-0.html (siehe S. 14).

[38] Juwei Shi u. a. „Clash of the Titans: MapReduce vs. Spark for Large Scale Data Analytics“. In: *Proc. VLDB Endow.* 8.13 (Sep. 2015), S. 2110–2121. ISSN: 2150-8097. DOI: 10.14778/2831360.2831365. URL: http://dx.doi.org/10.14778/2831360.2831365 (siehe S. 14).

[39] Lei Gu und Huan Li. „Memory or Time: Performance Evaluation for Iterative Operation on Hadoop and Spark“. In: *10th IEEE International Conference on High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing, HPCC/EUC 2013, Zhangjiajie, China, November 13-15, 2013*. 2013, S. 721–727. DOI: 10.1109/HPCC.and.EUC.2013.106. URL: http://dx.doi.org/10.1109/HPCC.and.EUC.2013.106 (siehe S. 14, 15).

[40] Holden Karau u. a. *Learning Spark: Lightning-Fast Big Data Analytics*. 1st. O'Reilly Media, Inc., 2015. ISBN: 1449358624, 9781449358624 (siehe S. 14, 15).

[41] Xiangrui Meng u. a. *ML Pipelines: A New High-Level API for MLlib*. 2015. URL: https://databricks.com/blog/2015/01/07/ml-pipelines-a-new-high-level-api-for-mllib.html (siehe S. 15).

[42] DataBricks. *CSV data source for Spark SQL and DataFrames*. 2016. URL: https://github.com/databricks/spark-csv (siehe S. 15).

[43] Chi Yau. *R Tutorial, Data Frame*. 2016. URL: http://www.r-tutor.com/r-introduction/data-frame (siehe S. 16).

[44] The R Foundation. *R project for statistical computing*. 2016. URL: http://www.r-project.org (siehe S. 16).

[45] Goetz Graefe. „The Cascades Framework for Query Optimization“. In: *IEEE Data Eng. Bull.* 18.3 (1995), S. 19–29. URL: http://sites.computer.org/debull/95SEP-CD.pdf (siehe S. 17).

[46] Goetz Graefe und David J. DeWitt. „The EXODUS Optimizer Generator“. In: *Proceedings of the Association for Computing Machinery Special Interest Group on Management of Data 1987 Annual Conference, San Francisco, California, May 27-29, 1987*. 1987, S. 160–172. DOI: 10.1145/38713.38734. URL: http://doi.acm.org/10.1145/38713.38734 (siehe S. 17).

[47] Philip Wadler. „Monads for Functional Programming“. In: *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques, Baastad, Sweden, May 24-30, 1995, Tutorial Text*. 1995, S. 24–52. DOI: 10.1007/3-540-59451-5_2. URL: http://dx.doi.org/10.1007/3-540-59451-5_2 (siehe S. 17).

[48] Denys Shabalin, Eugene Burmako und Martin Odersky. *Quasiquotes for scala*. Techn. Ber. 2013 (siehe S. 18).

[49] Nico Braunisch. *DataLog: Datenbank-Programmiersprache fuer deduktive Datenbanken, Proseminar „Programmierparadigmen und Sprachen“*. 2009. URL: http://st.inf.tu-dresden.de/files/teaching/ss09/PS09/braunisch_folien.pdf (siehe S. 19).

[50] Frank Pfenning und Carsten Schuermann. *Twelf user's guide*. Techn. Ber. version 1.2. Technical Report CMU-CS-98-173, Carnegie Mellon University, 1998. URL: http://www.cs.cmu.edu/~twelf/guide-1-4/twelf_5.html#SEC31 (siehe S. 20).

[51] Klaus Manhart. *BI-Methoden (Teil 1): Ad-hoc Analysen mit OLAP*. 2008. URL: http://www.tecchannel.de/a/bi-methoden-teil-1-ad-hoc-analysen-mit-olap,1751285,2 (siehe S. 20).

[52] Jags Ramnarayan u. a. „SnappyData: A Hybrid Transactional Analytical Store Built On Spark“. In: *Proceedings of the 2016 International Conference on Management of Data*. SIGMOD '16. San Francisco, California, USA: ACM, 2016, S. 2153–2156. ISBN: 978-1-4503-3531-7. DOI: 10.1145/2882903.2899408. URL: http://doi.acm.org/10.1145/2882903.2899408 (siehe S. 21, 22).

[53] Lucas Braun u. a. „Analytics in Motion: High Performance Event-Processing AND Real-Time Analytics in the Same Database“. In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. SIGMOD '15. Melbourne, Victoria, Australia: ACM, 2015, S. 251–264. ISBN: 978-1-4503-2758-9. DOI: 10.1145/2723372.2742783. URL: http://doi.acm.org/10.1145/2723372.2742783 (siehe S. 21).

[54] Erietta Liarou u. a. „MonetDB/DataCell: Online Analytics in a Streaming Column-store“. In: *Proc. VLDB Endow.* 5.12 (Aug. 2012), S. 1910–1913. ISSN: 2150-8097. DOI: 10.14778/2367502.2367535. URL: http://dx.doi.org/10.14778/2367502.2367535 (siehe S. 21).

[55] SnappyData Inc. *SnappyStore on GitHub*. 2016. URL: `https://github.com/SnappyDataInc/snappy-store` (siehe S. 22).

[56] SnappyData Inc. *About SnappyData*. 2016. URL: `http://stackoverflow.com/tags/snappydata/info` (siehe S. 22).

[57] Barzan Mozafari und Ning Niu. „A Handbook for Building an Approximate Query Engine“. In: *IEEE Data Eng. Bull.* 38.3 (2015), S. 3–29. URL: `http://sites.computer.org/debull/A15sept/p3.pdf` (siehe S. 22).

[58] Oracle. *MySQL Reference Manual*. 2016. URL: `http://dev.mysql.com/doc/refman/5.7/en/group-by-optimization.html` (siehe S. 23).

[59] The R Foundation. *Documentation for the package mallet.topic.hclust*. 2016. URL: `http://www.rdocumentation.org/packages/mallet/versions/1.0/topics/mallet.topic.hclust` (siehe S. 23).

[60] Database und Artificial Intelligence Group TU Wien. *Anfragebearbeitung 2, Vorlesung Datenbanksysteme vom 18.11.2015*. 2015. URL: `http://www.dbai.tuwien.ac.at/education/dbs/current/folien/Kapitel8b.pdf` (siehe S. 25).

[61] Apache Software Foundation. *Spark SQL, DataFrames and Datasets Guide*. 2016. URL: `http://spark.apache.org/docs/1.6.1/sql-programming-guide.html#performance-tuning` (siehe S. 31).

[62] Reynold Xin und Josh Rosen. *Project Tungsten: Bringing Apache Spark Closer to Bare Metal*. 2015. URL: `https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html` (siehe S. 31).

[63] LANCE CO TING KEH. *Apache Spark: Caching and Checkpointing Under the Hood*. 2015. URL: `https://blog.box.com/blog/apache-spark-caching-and-checkpointing-under-hood/` (siehe S. 33).

[64] Apache Software Foundation. *Spark SQL Programming Guide, RDD Persistence*. 2016. URL: `http://spark.apache.org/docs/latest/programming-guide.html#rdd-persistence` (siehe S. 33).

[65] Alexander Hinneburg u. a. „Exploring Document Collections with Topic Frames“. In: *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management*. CIKM '14. Shanghai, China: ACM, 2014, S. 2084–2086. ISBN: 978-1-4503-2598-1. DOI: `10.1145/2661829.2661857`. URL: `http://doi.acm.org/10.1145/2661829.2661857` (siehe S. 37).

**Eidesstattliche Erklärung**

Hiermit versichere ich, dass ich die Abschlussarbeit bzw. den entsprechend gekennzeichneten Anteil der Abschlussarbeit selbstständig verfasst, erstmalig eingereicht und keine anderen als die angegebenen Quellen und Hilfsmittel einschließlich der angegebenen oder beschriebenen Software benutzt habe. Die den benutzten Werken bzw. Quellen wörtlich oder sinngemäß entnommenen Stellen habe ich als solche kenntlich gemacht.

Ort, Datum

Unterschrift Dmitry Trofimov