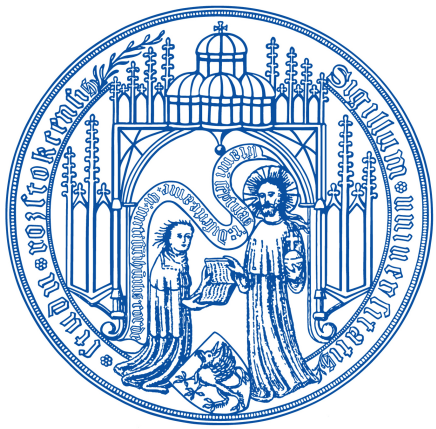
Vergleich paralleler Datenbanksysteme und Big-Data-Umgebungen für Hidden-Markov-Modelle

Bachelorarbeit

Universität Rostock
Fakultät für Informatik und Elektrotechnik
Institut für Informatik



vorgelegt von:	Maximilian Lamster
Matrikelnummer:	214207151
geboren am:	08. Juli 1995 in Waren (Müritz)
Erstgutachter:	Prof. Dr. rer. nat. habil. Andreas Heuer
Zweitgutachter:	Dr.-Ing. Holger Meyer
Betreuer:	Dennis Marten
Abgabedatum:	27. April 2020

Inhaltsverzeichnis

Kapitel 1

Einleitung

Das Erheben und Speichern von Daten wird immer leichter. Technische Geräte weisen immer komplexere Strukturen auf und werden sogar mehrfach mit diversen Sensoren ausgestattet, um einen Service basierend auf den gemessenen Daten liefern zu können. Zum Beispiel befinden sich in einem Kraftfahrzeug je nach Modell etwas mehr als 100 verschiedene Sensoren, welche über ein Dutzend verschiedene physikalische Größen messen [And16]. Bei einer Fahrzeugdichte in Deutschland von 689 Kraftfahrzeugen je 1000 Einwohner kann man die Menge gemessener Daten alleine im Bereich Kraftfahrzeug nur erahnen [Kra19]. Selbsterklärend vervielfältigt sich auch die Nachfrage nach Lösungen für die Analyse, Verarbeitung und Auswertung großer Datensätze ([Sch19]) und das natürlich nicht nur beschränkt auf die Automobilindustrie. Big-Data-Frameworks, so werden eben jene Softwarelösungen genannt, bieten die Möglichkeit der parallelen Analyse und Verarbeitung von großen Datenmengen auf mehreren, dem System zugewiesenen Knoten an. Seit der Einführung von Apache-Hadoop im Jahr 2006, welches auf dem Programmiermodell MapReduce basiert, ist auch die Menge der enwtickelten Big-Data-Frameworks drastisch angestiegen. Die Apache Software Foundation selbst veröffentlichte viele weitere bekannte Big-Data-Frameworks in den darauffolgenden Jahren, angepasst an die jeweiligen Trends: Apache-Spark, als Verbesserung zu Apache-Hadoop und weiterhin Apache-Flink und Apache-Storm für die Verarbeitung von Stromdaten in Echtzeit. Die eben genannten Vertreter der Apache Software Foundation sind außerdem komplett Open-Source und stellen alleine durch die kostenlose Verfügbarkeit eine näher zu betrachtende Alternative zu kommerziellen Systemen dar.
Die steigende Menge an Datensätzen hat allerdings nicht nur Vorteile. Im Bereich des maschinellen Lernens profitiert man zwar von der erhöhten Anzahl an Datensätzen, jedoch ergibt sich für den jeweiligen Verwalter der Daten ein Mehraufwand bezogen auf die Speicherung und Wartung. Doch gerade auch die angestrebten Analysen auf diesen Datensätzen erweisen sich je nach Dimension als schwierig. Relativ üblich ist die Verarbeitung der Daten in einem dafür entwickelten externen System, wie zum Beispiel Apache-Hadoop oder Apache-Spark, doch werden gerade in solchen externen Systemen die Transportwege und somit Ladezeiten in höheren Dimensionen zum Flaschenhals. Als Lösung dafür gilt das Konzept der Verarbeitung nahe an den Daten, oder auch simpler, das Benutzen von Speichersystemen, welche selbst über Analyse- und Auswertungstools verfügen, um besagtem Datentransport zu entgehen. Ein bekannter Vertreter für eine solche Lösung ist die Open-Source-Software Postgres-XL: ein kostenloses, skalierbares

und PostgreSQL-basiertes Datenbank-Cluster [Pos20b].
Eine der Möglichkeiten, maschinelles Lernen [1] zu nutzen, ist das Treffen von Vorhersagen. Beispielsweise das Berechnen konkreter Wahrscheinlichkeiten für das Eintreffen eines Wertes einer Instanz, sofern andere Werte derselben bekannt sind. Dabei ist es oftmals sinnvoll solche Rahmenbedingungen und Werte in einem Modell zusammenzufassen. Für spezielle Vorgänge der realen Welt kann zum Beispiel das Hidden-Markov-Modell (HMM) genutzt werden, welches von L. Rabiner und B. Juang in [Rab89] dokumentiert wurde. Im Grundverständnis handelt es sich dort um zwei verknüpfte stochastische Prozesse, bestehend aus Zuständen, Zustandsübergängen und den mit den Zuständen verknüpften Emissionen. In diesem Modell können dann über verschiedene Algorithmen entsprechende Vorhersagen getroffen werden. Verwendet wird das HMM beispielsweise in der Sprachverarbeitung, Bildverarbeitung, Physik, Bioinformatik und auch in der Statistik [Wen04].
Möchte man das HMM nun im erweiterten Rahmen und auf größeren Datenmengen ansetzen, so stellt sich bei der Vielzahl an Big-Data-Frameworks und anderweitigen Lösungen die Frage nach der Effizienz und Handhabung solcher Software. Ein Vergleich der Umsetzung und Berechnungszeit in diesen verschiedenen Systemen erscheint folglich unabdingbar. In dieser Arbeit wird dementsprechend das System Apache-Spark mit dem verteilten Datenbanksystem Postgres-XL in Bezug auf Umsetzung und Berechnung von Hidden-Markov-Modellen verglichen. Es wird der Forward-Algorithmus als ein spezieller Vertreter der möglichen Algorithmen betrachtet und dabei besonderer Wert auf die dünnbesetzten Matrix-Vektor-Multiplikation, aber auch auf die dichtbesetzte Matrix-Matrix-Multiplikation gelegt, deren Ergebnisse anschließend verglichen und erläutert werden. Die Umsetzung von dünn- und dichtbesetzter linearer Algebra in Postgres-XL wurde bereits in [MMDH19] realisiert und fließt mit seinen Ergebnissen in diese Arbeit mit ein.
In Kapitel 2 werden die mathematischen Grundlagen dieser Arbeit vorgestellt. Es wird dabei im Bereich der linearen Algebra auf Matrizen und deren Multiplikation eingegangen und das Hidden-Markov-Modell mit seinen in [Rab89] beschriebenen Problemklassen und Algorithmen erläutert. Im anschließenden Kapitel 3 werden die Systeme Apache-Spark, Postgres-XL, deren Aufbau und Funktionsweise erläutert. Kapitel 4 gibt dann Aufschluss über das Konzept und die gewählte Methodik zum Vergleich der genannten Systeme. In Kapitel 5 wird dann genauer auf die entsprechende Umsetzung und die daraus resultierenden und genutzten Implementierungen eingegangen. Schlussendlich werden die Ergebnisse dieser Umsetzungen in Kapitel 6 visualisiert und ausgewertet, um dann in Kapitel 7 ein Fazit zu ziehen und einen Ausblick zu geben.

[1] Die in der Einleitung getroffenen Aussagen zu Machine Learning und die darauf bezogenen Begriffe und Erklärungen basieren auf dem Buch [KT18].

Kapitel 2

Grundlagen

Mit diesem Kapitel sollen die nötigen Grundlagen für ein besseres Verständnis dieser Arbeit vermittelt werden. Begonnen wird dabei mit einer kurzen Wiederholung zu Matrizen und deren Darstellungsformen, wie sie auch in der späteren Umsetzung verwendet werden. Zusätzlich wird im Abschnitt der linearen Algebra das Matrix- und Hadamard-Produkt vorgestellt, da diese ebenfalls für spätere Umsetzungen wichtig sind. In Abschnitt 2.2 wird dann das angekündigte Hidden-Markov-Modell inklusive seiner Bestandteile betrachtet und die existierenden Problemklassen erklärt, um dann auf die resultierenden und relevanten Algorithmen eingehen zu können. Danach wird der Forward-Algorithmus als explizites Beispiel näher vorgestellt, in dessen späterer Implementierung auch das Matrix- und Hadamard-Produkt Bestandteile sind. Den Abschluss bilden zwei grundlegende Methoden zur Partitionierung von Datensätzen, da diese im späteren System Postgres-XL verwendet werden und für einen Vergleich mit dem System Apache-Spark relevant sind.

2.1 Lineare Algebra

Der Abschnitt der linearen Algebra bildet das mathematische Grundgerüst dieser Arbeit. Die benötigten Sätze und Definitionen basieren dabei auf [Fis17, TT73, BHWM12].

2.1.1 Matrix

Eine reelle Matrix A vom Format (m,n) ist ein rechteckiges Schema aus reellen Zahlen a_{ij} mit $i = 1, \ldots, m$ und $j = 1, \ldots, n$ und kann auch als Kurzform $A \in \mathbb{R}^{m \times n}$ oder durch Beschreibung der Werte $[a_{ij}]_{mn}$ dargestellt werden. Eine Matrix heißt **dichtbesetzt**, sofern sie nur einen **minimalen Prozentsatz an Nullwerten** aufweist. Ein Beispiel dafür zeigt Abbildung 2.1.

$$A \in \mathbb{R}^{m \times n} \quad = \quad [a_{ij}]_{mn} \quad = \quad \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & & a_{2n} \\ \vdots & & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}$$

Abb. 2.1: dichtbesetzte Beispielmatrix

Im Gegensatz dazu heißt eine Matrix genau dann **dünnbesetzt**, sofern sie nur einen **minimalen Prozentsatz an von Null verschiedenen Werten** aufweist. Eine n-dünnbesetzte Matrix bezeichnet dabei eine reelle Matrix, welche pro Zeile höchstens n von Null verschiedene Werte aufweist [TT73]. Dies ist insofern relevant, da nur 21-dünnbesetzte Matrizen in den später ausgeführten Tests mit dünnbesetzten Matrizen betrachtet werden. In diesen Matrizen gilt zusätzlich die Bedingung, dass eines dieser 21 Elemente pro Zeile das Diagonalelement sein muss. In Abbildung 2.2 ist eine 1-dünnbesetzte Matrix bestehend aus Diagonalelementen zu sehen.

$$A \in \mathbb{R}^{m \times n} \quad = \quad [a_{ij}]_{mn} \quad = \quad \begin{bmatrix} a_{11} & 0 & \cdots & 0 \\ 0 & a_{22} & & \vdots \\ \vdots & & \ddots & 0 \\ 0 & \cdots & 0 & a_{mn} \end{bmatrix}$$

Abb. 2.2: dünnbesetzte Beispielmatrix

Die übliche Speicherform für eine dünnbesetzte Matrix ist die Kombination aus Zeilenindex, Spaltenindex und dem von Null verschiedenen Wert, auch bekannt als i,j,v-Format. Eine Beispiel-Matrix in besagtem Format und ebenfalls eine 1-dünnbesetzte Matrix aus Diagonalelementen gibt die nächste Abbildung an:

(0, 0, a_{00})
(1, 1, a_{11})
(2, 2, a_{22})
(3, 3, a_{33})

Abb. 2.3: dünnbesetzte Diagonalmatrix im i,j,v-Format

Die gewählte Variable a_{ij} bezeichnet den Wert der Matrix in Zeile i und Spalte j und wird im späteren Verlauf ausschließlich mit Werten zwischen Null und Eins gefüllt.

Matrixmultiplikation

Das Matrix-Produkt zweier reeller Matrizen $A \in \mathbb{R}^{m \times p}$ und $B \in \mathbb{R}^{p \times n}$, so wie es in späteren Umsetzungen verwendet wird, ist nur dann definiert, wenn die Spaltenanzahl in Matrix A mit der Zeilenanzahl in Matrix B übereinstimmt. Es wird wie folgt gebildet:

$$AB := [a_{ij}]_{mn} \text{ mit } a_{ij} = \sum_{x=1}^{p} a_{ix} b_{xj} \quad \forall i = 1, \ldots, m\,, \ \forall j = 1, \ldots, n$$

Hadamard-Produkt

Seien $A \in \mathbb{R}^{m \times n}$ und $B \in \mathbb{R}^{m \times n}$ reelle Matrizen, dann ist das Hadamard-Produkt (nach [Mil07]) von A und B wie folgt definiert:

$$A \circ B = [a_{ij}]_{mn},\ a_{ij} = A_{ij} \cdot B_{ij}\ ,\ 1 \leq i \leq m, 1 \leq j \leq n$$

Das Hadamard-Produkt wird später in der Implementierung des Forward-Algorithmus und dort speziell im Rekursionsschritt benötigt.

2.2 Hidden-Markov-Modell

Um maschinelles Lernen zu nutzen, ist es meist notwendig, die gegebenen Parameter in einem Modell zu erfassen. Mit dem HMM lassen sich zum Beispiel Prozesse des realen Lebens modellieren, um diesbezüglich Vorhersagen treffen zu können, die für den Anwender von Vorteil sind. Dieses Modell findet unter anderem in der Spracherkennung, Physik, Bioinformatik oder auch Statistik Verwendung. Das HMM, dessen Problemklassen und Algorithmen werden in den folgenden Abschnitten basierend auf [Wen04, Rab89] definiert. Die primäre Beschreibung als stochastischer Automat soll eine anschaulichere Darstellung und somit einen Einstieg in die Thematik ermöglichen. Danach erfolgt eine alternative und etwas formalere Beschreibung des besagten Modells.

Stochastischer Automat

Ein HMM kann unter anderem als endlicher Automat beschrieben werden, dessen Zustandsübergänge und Ausgaben nicht von den Eingaben abhängen, sondern probabilistisch festgelegt sind. Verwendung findet dieses Modell unter anderem in der Spracherkennung zur Klangmodellierung von Worten. Ein Beispiel eines solchen Modells durch einen stochastischen Automaten ist in Abbildung 2.3 zu sehen.

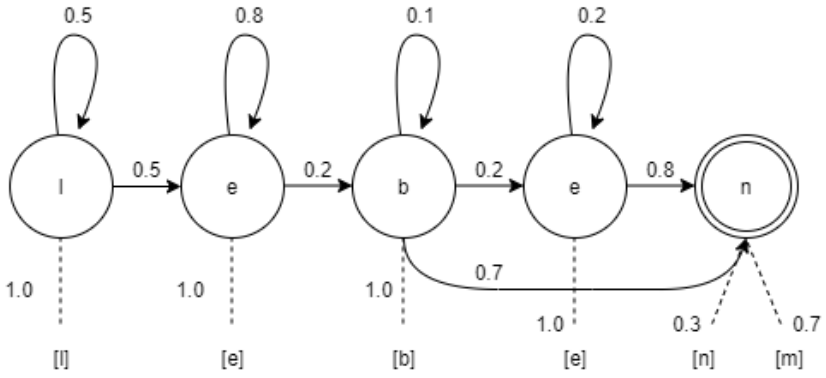


Abb. 2.3: Schematischen Hidden-Markov-Modell für die Klangmodellierung des Wortes *leben*

Durch Zustandsübergänge in denselben Zustand kann eine zeitliche Dehnung des Lautes modelliert werden, auch ist ein Verschlucken des zweiten e und eine undeutliche Aussprache des n modelliert worden.

Folgend lassen sich also verschiedene Arten der Aussprache, z.B. *lleeeben* oder *lebm*, modellieren. Mit diesem Modell ist es nun auch möglich, die Wahrscheinlichkeit zu berechnen, dass ein bestimmtes Wort auch von diesem Automaten erzeugt wurde. Bei dieser Abbildung handelt es sich um ein selbst erstelltes Beispiel. Die Orientierung für die Erstellung erfolgte aan einem ähnlichen Beispiel auf Seite 137 in [Wen04].
Für einen solchen stochastischen Automaten müssen außerdem immer die folgenden Eigenschaften gelten:

- Der Automat kann verschiedene Zustände aus einem endlichen Zustandsalphabet $X = [x_1, x_2, \ldots, x_n]$, $n \in \mathbb{N}^+$ annehmen und befindet sich in jedem diskreten Zeitabschnitt $t \in \mathbb{N}^+$ in einem verborgenen Zustand X_t aus dem Zustandsalphabet X.
- Es besteht für jeden Zustand $x_i \in X$ im Zeitpunkt t eine bestimmte Übergangswahrscheinlichkeit a_{ij}, dass sich der Zustand des HMM's im Zeitpunkt $t+1$ in den Zustand $x_j \in X$ verändern kann.
- Es gilt die **Markoveigenschaft 1. Ordnung**, welche besagt, dass jeder Zustandsübergang statistisch nur vom aktuellen Zustand abhängt.
- Jeder Zustand produziert in jedem Zeitpunkt t eine Ausgabe Y_t. Diese Ausgaben werden Emissionen genannt und entstammen dem endlichen Ausgabealphabet $Y = [y_1, y_2, \ldots, y_m]$, $m \in \mathbb{N}^+$.
- Die Wahrscheinlichkeit, dass ein Zustand x_i in einem Zeitpunkt t die Emission y_j erzeugt, entspricht der Emissionswahrscheinlichkeit b_{ij}. Die Emissionswahrscheinlichkeit ist dabei von t unabhängig.

Formale Beschreibung

Alternativ kann ein Hidden-Markov-Modell auch als 5-Tupel beschrieben werden.

Ein HMM λ ist ein 5-Tupel (X, Y, A, B, π) und es gilt:

- $X = (x_1, x_2, \ldots, x_n)$, endliche Menge von Zuständen
- $Y = (y_1, y_2, \ldots, y_m)$, endliche Menge von Emissionen
- $A \in \mathbb{R}^{n \times n}$, Matrix mit Übergangswahrscheinlichkeiten

$$A = [a_{ij}],\ a_{ij} = P(X_t = x_i | X_{t+1} = x_j)$$

- $B \in \mathbb{R}^{n \times m}$, Matrix mit Emissionswahrscheinlichkeiten

$$B = [b_{ij}],\ b_{ij} = P(X_t = x_i | Y_t = y_j)$$

- $\pi \in \mathbb{R}^n$, Anfangsverteilung

$$\pi = [\pi_i],\ \pi_i = P(X_1 = x_i)$$

Dabei sei $H = (X_1, X_2, \ldots, X_k)$ eine Zustandsfolge der Länge $k \in \mathbb{N}^+$ und $O = (Y_1, Y_2, \ldots, Y_k)$ eine Emissionsfolge der Länge $k \in \mathbb{N}^+$. Man beachte außerdem, dass folgende zwei Annahmen getroffen werden:

- Der aktuelle Zustand ist nur abhängig vom Vorgänger-Zustand:

$$P(X_t|X_1^{t-1}) = (X_t|X_{t-1})$$

- Die Emission zum Zeitpunkt t hängt nur vom Zustand in Zeitpunkt t ab und ist unabhängig von vorherigen Emissionen oder Zuständen:

$$P(Y_t|Y_1^{t-1},\, X_1^t) = (Y_t|X_t)$$

Doppelt stochastischer Zufallsprozess

Ein HMM wird auch als Verknüpfung zweier stochastischer Zufallsprozesse verstanden, bei dem eine Markov-Kette mit unbeobachteten Zuständen und mit bestimmten Wahrscheinlichkeiten zustandsabhängige Emissionen erzeugt. Dieser Prozess wird auch in Abbildung 2.4 noch einmal veranschaulicht.

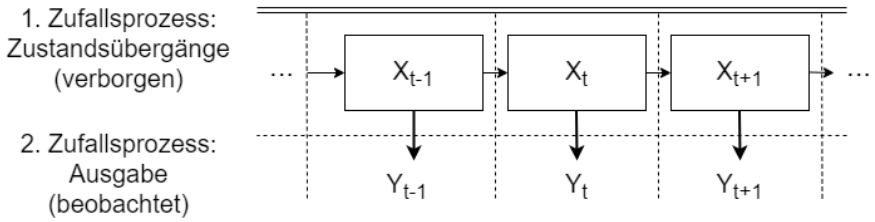


Abb. 2.4: Hidden-Markov-Modell als Kombination zweier stochastischer Zufallsprozesse

2.2.1 Problemklassen

Aus den vorangegangenen Definitionen des Hidden-Markov-Modells ergeben sich drei bekannte Problemklassen, welche in den folgenden Abschnitten mit entsprechende Lösungsmöglichkeiten vorgestellt werden. Da X und Y aus HMM $\lambda = (X, Y, A, B, \pi)$ durch A und B beschrieben werden können, nutzen wir in folgendem Abschnitt die verkürzte Schreibweise HMM $\lambda = (A, B, \pi)$.

Decoding-Problem

Gegeben sei ein HMM $\lambda = (A, B, \pi)$ und eine Beobachtungsfolge von Emissionen $O = (Y_1, Y_2, \ldots, Y_k)$ der Länge k: Wie finden wir eine optimale Zustandsfolge $H = (X_1, X_2, \ldots, X_k)$, welche die Emissionsfolge O am besten erklärt?

Lösung des Decoding-Problems

Ein bekannter Algorithmus zur Lösung eines Decoding-Problems ist der Viterbi-Algorithmus. Dieser gibt die wahrscheinlichste Zustandsfolge bei gegebener Beobachtungsfolge O und gegebenem HMM λ aus. Der

Viterbi-Algorithmus wird in dieser Arbeit wegen des gesetzten Rahmens aber nicht weiter betrachtet. Für Interessierte sei auf [Die18] verwiesen, da der Viterbi-Algorithmus dort nicht nur näher vorgestellt wird, sondern bereits auf einem Cluster-Rechner umgesetzt und getestet wurde.

Learning-Problem

Bei gegebener Beobachtungsfolge $O = (Y_1, Y_2, \ldots, Y_k)$ der Länge k: Wie setzen wir die Parameter des HMM λ so, dass $P(O|\lambda)$, die Wahrscheinlichkeit von O bei gegebenem Modell λ, maximal wird? Das Learning-Problem gehört zu den aufwendigeren Problemstellungen und wird im weiteren Verlauf nicht weiter betrachtet.

Evaluation-Problem

Gegeben sei ein HMM $\lambda = (A, B, \pi)$ und eine Beobachtungsfolge von Emissionen $O = (Y_1, Y_2, \ldots, Y_k)$ der Länge k: Wie berechnen wir nun effizient $P(O|\lambda)$, bei gegebenem Modell λ und bei gegebener Beobachtungsfolge O der Länge $k \in \mathbb{N}^+$?

Lösung des Evaluation-Problems

Es sei ein HMM $\lambda = (A, B, \pi)$ und eine Beobachtungsfolge von Emissionen $O = (Y_1, Y_2, \ldots, Y_k)$ der Länge k gegeben, dann kann $P(O|\lambda)$ für eine bestimmte Zustandsfolge $H = (X_1, X_2, \ldots, X_k)$ wie folgt intuitiv berechnet werden:

$$P(O \mid H, \lambda) = \prod_{t=1}^{k} P(Y_t \mid X_t, \lambda) = b_{X_1}(Y_1) \tag{1}$$

Die Wahrscheinlichkeit der Zustandsfolge H ist dann:

$$P(H \mid \lambda) = \pi_{X_1} a_{X_1 X_2} a_{X_2 X_3} \ldots a_{X_{k-1} X_k} \tag{2}$$

So kann anschließend die Wahrscheinlichkeit von O bei gegebenem Modell λ berechnet werden:

$$P(O \mid \lambda) = \sum_{H} P(O \mid H, \lambda) P(H \mid \lambda) = \sum_{X_1 \ldots X_k} \pi_{X_1} b_{X_1 Y_1} a_{X_1 X_2} b_{X_2 Y_2} \ldots a_{X_{k-1} X_k} b_{X_k Y_k} \tag{3}$$

Diese Form der Berechnung für die Wahrscheinlichkeit von O ist exponentiell zu k und somit ineffizient.

2.2.2 Forward-Algorithmus

Nachfolgend wird eine bessere Lösung in Form des Forward-Algorithmus beschrieben, welcher auch im späteren Verlauf der Arbeit im System Apache-Spark implementiert wird. Der Algorithmus basiert darauf, die vielen redundanten und zur Berechnung von $P(O \mid \lambda)$ nötigen Zwischenergebnisse wiederzuverwenden - ist also ein klassisches Beispiel der dynamischen Programmierung. Dazu wird eine Forward-Variable α eingeführt, welche die Wahrscheinlichkeit einer partiellen Beobachtungsfolge für ein Gitter aus Zuständen im Zeitpunkt t darstellt:

$$\alpha_t(i) = P(Y_1, Y_2, \ldots, Y_t, X_t = x_i | \lambda)$$

Summieren wir nun die Wahrscheinlichkeiten der Zustände des Gitters in der finalen Spalte k, so erhalten wir die gesuchte Wahrscheinlichkeit für die Beobachtungsfolge.
Der Algorithmus wird in folgende drei Abschnitte unterteilt:

[1] Initialisierung $\alpha_1(i) = \pi_i b_{X_i Y_1}, \quad \forall i = 1, \ldots, n$

[2] Rekursion $\alpha_{t+1}(j) = [\sum_{i=1}^{n} \alpha_t(i) a_{X_i X_j}] b_{X_j Y_{t+1}}, \forall t = 1, \ldots, k-1, \forall j = 1, \ldots, n$

[3] Terminierung $P(O \mid \lambda) = \sum_{i=1}^{n} \alpha_k(i)$

2.3 Partitionierung

Vor allem Big-Data-Anwendungen verarbeiten Datensätze von enormer Größe. Um Zugriffe auf entsprechende Daten nicht unnötig zu verlangsamen, sollte eine Ausführung von Analysen und Berechnungen auf dem Knoten erfolgen, auf dem die Daten gespeichert sind. Generell gilt dabei, dass eine größere Partitionierung der Daten eine schnellere Berechnung, aber dafür einen höheren Kommunikationsaufwand in Bezug auf die Zusammenführung der Ergebnisse mit sich bringt. Dementsprechend bedingt eine weniger ausgeprägte Partitionierung zwar eine langsamere Berechnung, dafür aber eine schnellere Zusammenführung der Ergebnisse. Abschließend werden noch zwei Beispiele für die Partitionierung von Datenbeständen vorgestellt, die auch in der Umsetzung von Postgres-XL Verwendung finden:

1) **RoundRobin:** RoundRobin bezeichnet eine Partitionierung der Datensätze im Rundlauf-Verfahren, was bedeutet, dass der erste Datensatz auf den ersten Knoten, der zweite Datensatz auf den zweiten Knoten usw. verteilt wird. Beim letzten Knoten angekommen wird einfach wieder beim ersten Knoten fortgesetzt, deswegen der Name Rundlauf. Die Auslastung der Knoten erfolgt somit gleichmäßig, jedoch liegen die Daten auch sehr verstreut vor, was zu höheren Lesekosten führt.

2) **Range:** Die Range-Partitionierung beschreibt eine Verteilung der Datensätze nach einem gesetzten Attributintervall und somit nach dem Vorhandensein eines bestimmten Attributwertes in einem Datensatz. Ein Vorteil bietet diese Art der Partition vor allem bei Exact-Match-Anfragen, sorgt aber für eine unterschiedliche Besetzung der Knoten mit Datensätzen.

Kapitel 3

Stand der Technik

In diesem Kapitel sollen die verwendeten Systeme und ihre grundlegende Funktionsweise vorgestellt werden. Zuerst wird das Programmierparadigma MapReduce basierend auf [LRU19] vorgestellt, da es die Grundlage vieler Big-Data-Frameworks bildet. Danach erfasst ein kleiner Abschnitt Apache-Hadoop und dessen Dateisystem, weil das Zielsystem Apache-Spark auf einem Hadoop-Cluster installiert wurde und das Hadoop-Speichersystem nutzt. Anschließend wird dann Apache-Spark und dessen Funktionsweise betrachtet, sowie die für die Umsetzung linearer Algebra wichtige Bibliothek MLlib. Zuletzt gibt es einen kleinen Überblick zur Funktionsweise des Vergleichssystems Postgres-XL.

3.1 MapReduce

Eine Anforderung an Big-Data-Frameworks ist die schnelle Verarbeitung großer Datenmengen. Vor allem homogene Datensätze lassen sich dabei in verschiedener Weise parallelisieren. Für eine solche Parallelität der Daten sind mittlerweile nicht mehr Supercomputer, sondern Cluster-Computer zuständig, welche meist aus einfachster Hardware bestehen und über billige Ethernet-Kabel oder Switches verbunden sind. Eine solche Umgebung benötigt demnach auch einen anderen Software-Stack als ein einzelnes System. Eine wichtige Rolle spielt dabei ein verteiltes Dateisystem, welches über mehr Einheiten verfügt als nur einen Plattenblock. Durch gezielte Replikation und Redundanz der Daten auf verschiedenen Knoten kann es dabei Fehlern in den Daten oder der Knoten selbst entgegenwirken. Allerdings führt eine größere Anzahl an Teilnehmern eines Systems auch zu einer höheren Ausfallrate. Der Software-Stack umfasst außerdem eine Implementierung des Programmiermodells namens MapReduce, welches meist in einer high-level Programmiersprache umgesetzt wird und oftmals eine zusätzliche Implementation von SQL anbietet. Die Implementierungen von MapReduce erlauben die Berechnung großer Datenmengen auf Computer-Clustern und sind ebenfalls an der Fehlerbehandlung beteiligt. Um eine solche zu nutzen, ist es allein notwendig, zwei Funktionen zu definieren: **Map** und **Reduce**. Das System verwaltet dann die parallele Verarbeitung und Koordination der Aufgaben, die diese Funktionen ausführen. Hier ein kurzer Überblick über die Ausführung eines MapReduce-Prozesses:

- Jedem Map-Knoten werden ein oder mehrere Teile der Daten des verteilten Dateisystems zuge-

wiesen. Diese Datenblöcke werden dann in Sequenzen aus Key-Value-Paaren umgewandelt. Die Art und Weise, wie diese Paare produziert werden, legt der Anwender im Code der Map-Funktion selber fest.

- Die Schlüssel-Wert-Paare aus den verschiedenen Map-Knoten werden dann von einem Master gesammelt und nach ihrem Schlüssel sortiert. Diese Paare werden anschließend so verteilt, dass all jene mit selbigem Schlüssel auch bei ein und demselben Reduce-Knoten landen.
- Die Reduce-Knoten arbeiten immer nur auf einem Schlüssel zur selben Zeit. Sie kombinieren die mit diesem Schlüssel assoziierten Werte wie es der Nutzer in der Reduce-Funktion festlegt.

MapReduce wurde bereits in vielen Systemen umgesetzt, angefangen mit Google's Implementation mit dem gleichen Namen MapReduce, aber auch in der populären Open-Source Software Apache-Hadoop, bereitgestellt von der Apache Software Foundation. Generell entwickeln sich MapReduce-Systeme vor allem durch die steigende Nutzung von Cluster-Computern und den hohen Bedarf der Analyse großer Datenmengen durch Webanwendungen sehr schnell weiter.

3.2 Apache-Hadoop

Das Framework Hadoop bietet eine kostenlose Möglichkeit, Anwendungen auf einem Cluster mit Standardhardware auszuführen und wird zusammen mit dem Hadoop Distributed File System (HDFS) geliefert. Das HDFS übernimmt dabei die Rolle des verteilten Dateisystems, welches die Daten auf den Rechenknoten speichert, eine hohe Bandbreite im Cluster anbietet sowie für Redundanz und Replikation zuständig ist [Had20]. Auch die in dieser Arbeit beschriebene Realisierung eines Spark-Clusters bezieht die verwendeten Daten aus dem HDFS und ist auf einem Hadoop-Cluster installiert.

3.3 Apache-Spark

Das Big-Data-Framework Apache-Spark ist im Kern ein Workflow-System und stellt eine Weiterentwicklung des Programmiermodells MapReduce unter anderem durch folgende Punkte dar:

- den effizienteren Umgang mit Fehlern
- der effizienten Gruppierung von Aufgaben zwischen Knoten
- der effizienten Planung der Ausführung von Funktionen
- der Integration von Programmiersprachenkonstrukten, z.B. Schleifen, welche es technisch gesehen zu einem azyklischen Workflow-System machen.

In diesem Abschnitt sollen die Bestandteile, Funktionsweise, Bibliotheken und Schnittstellen von Apache-Spark präsentiert werden, welche im Verlauf dieser Arbeit Verwendung finden. Die folgenden Abschnitte basieren dabei auf der offiziellen Dokumentation von Apache-Spark [Spa20a, Spa20b, Spa20c, Spa20d] und zusätzlich auf [LRU19].

3.3.1 Resilient Distributed Dataset

Die zentrale Datenabstraktion von Apache-Spark wird als Resilient Distributed Dataset bzw. RDD bezeichnet. Ein RDD ist eine Datei mit Objekten eines bestimmten Typs. Typbedingte Einschränkungen gibt es dabei nicht. Ein Beispiel für ein RDD stellen unter anderem Dateien mit Key-Value-Paaren dar, welche in MapReduce-Systemen üblich sind. RDD's in Apache-Spark können allerdings auch an Funktionen übergeben und verteilt werden. Resilient (dt.: belastbar) steht dabei für die Eigenschaft, dass Spark bei Verlust eines oder aller Teile eines RDD's diese einfach neu generieren kann, da es die Pläne zur Erstellung für jede RDD und dessen Partitionen aufzeichnet. Eine redundante Speicherung von Zwischenergebnissen wird dadurch unnötig, was in einigen Fällen wertvolle Zeit erspart. Erstellen lässt sich ein RDD unter anderem durch das Parallelisieren einer vorhandenen Datei oder durch das Verweisen auf ein gemeinsam genutztes Speichersystem, z.B. auf den lokalen Speicher, das HDFS oder irgendein anderes auf Hadoop basierendes Dateisystem. Da Apache-Spark eben diese auf Hadoop basierten Speichersysteme verwenden kann, lassen sich Apache-Hadoop und Apache-Spark nicht komplett voneinander trennen. Auf RDD's sind viele Operationen möglich, wobei man grundlegend die folgenden zwei Arten unterscheidet:

- Transformationen: Operationen, die ein neues RDD basierend auf einem vorhandenen RDD erstellen.
- Aktionen: Operationen, die aus einem RDD einen bestimmte Wert berechnen und an das Treiber-Programm zurückliefern.

Map

Die Map-Transformation verwendet einen Parameter in Form einer Funktion und wendet diese an jedem Element einer RDD an, wodurch eine weitere RDD erzeugt wird. Diese Operation erinnert an das *Map* von MapReduce, ist aber nicht genau das Gleiche. Erstens kann in MapReduce eine Map-Funktion nur auf ein Key-Value-Paar angewendet werden. Zweitens erzeugt eine Map-Funktion in MapReduce eine Reihe von Key-Value-Paaren, wobei jedes Paar ein unabhängiges Element der Ausgabe der Map-Funktion ist. In Apache-Spark kann eine Map-Funktion auf jeden Objekttyp angewendet werden. Im Ergebnis produziert eine Map-Funktion auch genau denselben Objekttyp. Um aus einem Objekt eine Menge von Objekten zu erstellen, bietet Apache-Spark sogar die Funktion *FlatMap* an, welche wegen fehlender Verwendung in den späteren Implementierungen nicht weiter beschrieben werden soll.

Reduce

In Apache-Spark ist die Reduce-Funktion eine Aktion. Sie kann zwar auf ein RDD angewendet werden, gibt jedoch nur einen Wert und kein anderes RDD zurück. *Reduce* nimmt als Parameter, ähnlich zum *Map*, eine Funktion, welche dann zwei Elemente eines bestimmten Typs nimmt und ein anderes Element des gleichen Typs zurückliefert. Auf ein RDD mit Elementen desselben Typs angewendet, wird das *Reduce* so oft auf jedem Paar aufeinanderfolgender Elemente wiederholt, bis nur noch ein einzelnes Element übrig bleibt. Dieses einzelne Element bildet dann das Ergebnis der Reduce-Funktion. Wird beispielsweise als Parameter eine Additions-Funktion und ein RDD von Integern an ein *Reduce* übergeben, so liefert dieses

die Summe aller Integer des RDD's als Ergebnis. Es ist außerdem möglich eine beliebige Funktion als Parameter zu verwenden, solange die Kombination von Elementen in beliebiger Reihenfolge keine Probleme bereitet. Um ein Reduce auf gleiche Schlüssel zu beschränken, kann die Operation *reduceByKey*() verwendet werden.

Lazy-Evaluation

Ein großer Vorteil von Apache-Spark gegnüber Apache-Hadoop ist die Nutzung des Prinzips der **Lazy-Evaluation**. Transformationen werden dadurch nicht direkt ausgeführt, sondern es wird lediglich ein Vermerk zur Transformation in dem zugehörigen RDD angelegt. Eine Ausführung der Transformationen erfolgt erst dann, wenn eine Aktion das Ergebnis einer Transformation benötigt, um es anschließend an das Treiber-Programm zurück zu liefern. Aus diesem Grund ist es manchmal erforderlich, eine zusätzliche Aktion, z.B. ein *count*(), durchzuführen, um eine Transformation zu erzwingen. Der Vorteil dieser Strategie beruht vor allem darauf, dass RDD's nicht alle gleichzeitig konstruiert werden. Wenn ein Teil eines RDD's auf einem Knoten erzeugt wird, kann dieser am selben Rechenknoten auch direkt verwendet werden. Laufzeitersparnis entsteht dann unter anderem dadurch, dass ein RDD nicht im lokalen Speicher gespeichert und auch nicht an andere Rechenknoten übertragen werden muss.
Standardmäßig wird eine Transformation auf einem RDD für jede Aktion neu durchgeführt. Eine Lösung zur Zwischenspeicherung dieser Transformationen für weitere Aktionen bietet z.B. die Operation *cache*() an.

Shuffle

Einige Operationen in Apache-Spark erfordern allerdings trotzdem eine Neuverteilung der parallelisierten Daten, die das Kopieren der Daten auf verschiedene Executors und Maschinen und somit auch zusätzlichen Aufwand mit sich bringt. Diese Neuverteilung wird auch als **Shuffle** bezeichnet und kann mit extremen Kosten verbunden sein. Es werden dadurch viele Schreib- und Leseoperationen im Netzwerk beziehungsweise Speicher durchgeführt und zusätzlich noch eine Serialisierung der Daten vorgenommen. Operationen die ein Shuffle hervorrufen sind unter anderem das in der Umsetzung dieser Arbeit verwendete *reduceByKey*().

3.3.2 MLlib

Für die einfache und praktische Anwendung maschinellen Lernens bietet Apache-Spark eine Bibliothek namens MLlib an. Darin enthalten sind übliche Algorithmen wie z.B. Regression und Clustering, aber auch zusätzliche Tools für Featurization oder Pipelines. Es sind außerden Utilities für Statistik und die für diese Arbeit interessante lineare Algebra vorhanden. MLlib unterstützt nicht nur einfache Datenmodelle lokaler Vektoren und Matrizen, sondern auch Datenmodelle verteilter Matrizen. Die Operationen werden dabei von Breeze bereitgestellt. Folgende Datentypen verteilter Matrizen werden via MLlib realisiert:

- RowMatrix
- IndexedRowMatrix

- CoordinateMatrix
- BlockMatrix

Eine **RowMatrix** ist eine zeilenorientierte verteilte Matrix ohne aussagekräftige Zeilenindizes, realisiert durch ein RDD von Zeilenvektoren. Die **IndexedRowMatrix** ähnelt stark einer RowMatrix, verfügt jedoch über Zeilenindizes. Mit diesen können bestimmte Zeilen identifiziert und somit auch Verknüpfungen durchgeführt werden. Realisiert wird diese Form einer verteilten Matrix durch ein RDD von Tupeln, bestehend aus Zeilenindex und Zeilenvektor. Der Datentyp **CoordinateMatrix** ist dagegen eine verteilte Matrix bestehend aus Matrixeinträgen im i,j,v-Format und wird durch ein RDD eben jener Matrixeinträge realisiert. Der letzte Datentyp ist die **BlockMatrix**, eine blockweise verteilte Matrix, die durch ein RDD aus Matrixblöcken realisiert wird. Ein Matrixblock ist dabei ein 3-Tupel der Form (Int, Int, Matrix), wobei das Wort Matrix in diesem Tupel für eine lokale Matrix steht. Auf die lokalen Realisierungen einer Matrix wird deswegen nicht weiter eingegangen, da die Definition einer BlockMatrix aus lokalen Matrizen zu aufwendig ist. Einfacher erfolgt die Erstellung einer BlockMatrix durch die Umwandlung einer CoordinateMatrix mittels $toBlockMatrix()$. Für die Umsetzungen lokaler dicht- und dünnbesetzter Matrizen und Vektoren liefert Apache-Spark ebenfalls Realisierungen, die aber durch die betrachteten Dimensionen der Daten in dieser Arbeit vernachlässigt werden können.

3.3.3 Datasets und DataFrames

Eine Schnittstelle für Datasets wurde mit der Spark-Version 1.6 eingeführt. Ein Dataset beschreibt eine verteilte Sammlung von Daten, welche die Vorteile von RDD's und der optimierten Engine von SparkSQL vereint. Datasets können über Transformationen, die aus der funktionalen Programmierung bekannt sind manipuliert werden.
DataFrames sind darauf aufbauend Datasets mit einer zeilenweise verteilten und in benannten Spalten angeordneten Struktur. Sie können somit als Analogie zu einer relationalen Datenbank oder einem Dataframe in R/Python aufgefasst werden, allerdings mit den zusätzlichen Optimierungen der SparkSQL-Engine. In der späteren Umsetzung werden die Daten aus den csv-Dateien als DataFrames eingelesen und auch als solche weiter verarbeitet.

3.3.4 SparkSQL

Für die strukturierte Datenverarbeitung enthält Apache-Spark das Modul SparkSQL, mit welchem auch eben genannte DataFrames und Datasets bearbeitet werden können. Die von diesem Modul bereitgestellten Schnittstellen bieten im Gegensatz zur herkömmlichen RDD-Schnittstelle zusätzliche Informationen zur Struktur der Daten und den daran ausgeführten Berechnungen an. Darauf basierend werden außerdem interne Optimierungen durchgeführt. Eine weitere Einsatzmöglichkeit von SparkSQL ist das Ausführen von SQL-Anweisungen in den Anwendungen. In dieser Arbeit wird das Modul SparkSQL ausschließlich zur Verarbeitung von DataFrames genutzt.

3.4 Postgres-XL

Postgres-XL ist ein Open-Source Projekt, welches die parallele Verarbeitung und Schreibskalierbarkeit auf dem objektrelationalen Datenbankmanagementsystem PostgreSQL ermöglicht. Es handelt sich dabei um eine Sammlung eng gekoppelter Datenbankkomponenten. Schreibskalierbarkeit beschreibt zu gegebenem Kontext eine mögliche Konfiguration für beliebig viele Datenbankserver und demnach die Umsetzung von mehr Update-Statements als es ein einzelner Datenbankserver umsetzen könnte. Da die Daten je nach Wahl des Nutzers partitioniert und repliziert werden, wird bei Anstoß einer Anfrage zuerst der Aufenthaltsort der benötigten Daten ermittelt und als Plan, inklusive der Daten selbst, zu den Datenbankservern gesendet. Ein Vorteil ist dabei die direkte Sichtbarkeit von Veränderungen auf jeglichen Servern in den angebundenen Anwendungen, da jeder Server seine uniforme Sicht der Daten bereitstellt. Ein weiteres Feature ist die Parallelisierung für Business Intelligence, Data Warehousing und Big Data. Ein Ziel ist die datenbankübergreifende Skalierbarkeit unter Einhaltung des ACID-Prinzips. Die Bestandteile von Postgres-XL lauten dabei wie folgt:

- Global Transaction Manager: Schlüsselkomponente, welche für das Transaktionsmanagement, Tupel-Sichtbarkeit und somit auch für die Konsistenz verantwortlich ist.
- Coordinator: Schnittstelle, die wie ein konventionelles PostgreSQL-Backend agiert, allerdings selber keine Daten speichert.
- Datanode: Ist der eigentlichen Speicherort der Daten. Relationen können dabei auf verschiedenen Datanodes verteilt und repliziert vorkommen und bieten somit keine globale Sicht auf die Datenbank selbst.

Postgres-XL kann außerdem als Erweiterung zu PostgreSQL verstanden werden und erbt somit auch dessen Features und die einfache Erweiterbarkeit. Für eine weitere und genauere Beschreibung von Postgres-XL sei auf die offizielle Dokumentation [Pos20c] verwiesen, auf der auch die hier genannten Grundlagen basieren. Die nachfolgende Abbildung zeigt abschließend einen beispielhaften Aufbau eines Postgres-XL-Clusters.

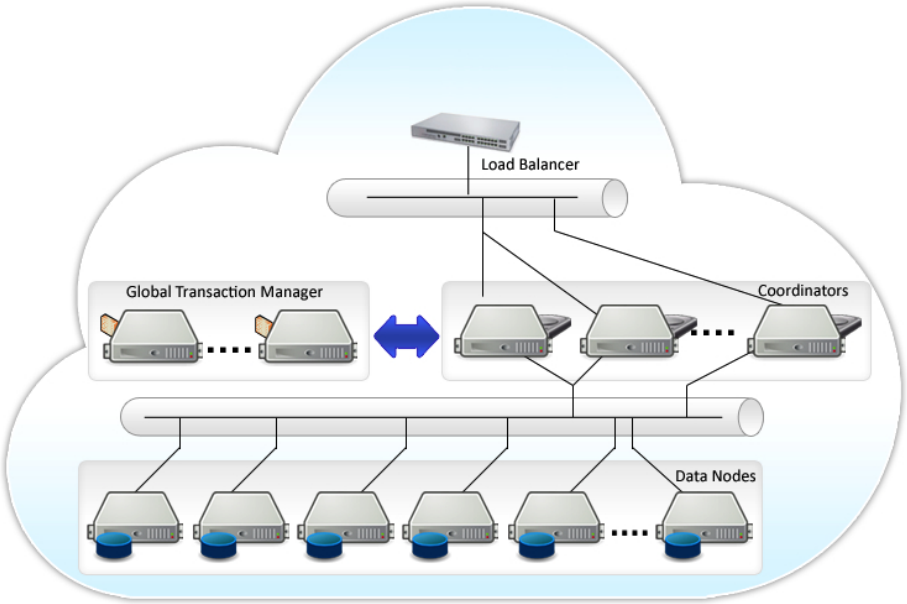


Abb. 3.3: Aufbau eines Postgres-XL-Clusters [Pos20a]

3.5 Stand der Forschung

Die bereits existierenden Lösungen und Herangehensweisen für die Umsetzung von Algorithmen zu Hidden-Markov-Modellen sollen keinesfalls außer Acht gelassen werden. Diese waren maßgebend an der Konzeptbildung beteiligt und haben auch gewisse Erkenntniswege bezüglich der Umsetzung gelenkt. Aus diesem Grund erscheint es allerdings sinnvoll, diese auch erst in den nächsten Kapiteln und im entsprechenden Kontext zu erwähnen.

Kapitel 4

Konzept

Bereits 2010 wurde im Artikel [SAD+10] die prinzipielle Unterlegenheit von Datenbanksystemen gegenüber Big-Data-Frameworks widerlegt. Als Testsysteme fungierten dabei ein kommerzielles zeilenorientiertes Datenbanksystem, Vertica als Vertreter eines spaltenorientierten Datenbanksystems und Hadoop als Big-Data-Framework. Auf dieser Erkenntnis aufbauend wurden in den folgenden Jahren auch durch den Zuwachs an Big-Data-Frameworks und parallelen Datenbanksystemen immer wieder neue Vergleichspartner in verschiedenen Disziplinen gegenübergestellt. Ein Beispiel eines solchen Vergleiches findet sich auch im 2018 veröffentlichten Konferenzbeitrag [DFS+18] wieder. Dort wurden Apache-Spark, Apache-Flink und Postgres-XL in Grep-, Join-Task und im k-Means-Clustering verglichen. Dabei zeigte sich, dass Postgres-XL als paralleles Datenbanksystem deutlich schnellere Ergebnisse in den beiden zuletzt genannten Disziplinen lieferte und in der Ersteren auch nicht deutlich hinter den Big-Data-Systemen zurücklag. Darauf aufbauend stellt sich die Frage der Performanz auch für typische und vor allem häufige Operationen in der stetig wachsenden Thematik des maschinellen Lernens. Als zentrale zu vergleichende Operationen wurden für diese Arbeit die **dünnbesetzte Matrix-Vektor-Multiplikation** mit verschiedenen Partitionierungstrategien und die **dichtbesetzte Matrix-Matrix-Multiplikation** mit blockweiser Partitionierung ausgewählt. Besonderes Interesse gilt dabei vor allem der dünnbesetzten Variante, denn diese stellt den Teil im Rekursionsschritt des Forward-Algorithmus dar, der die Laufzeit am meisten beeinflusst. Der Algorithmus wurde bereits in den Grundlagen vorgestellt und wird in der späteren Umsetzung im System Apache-Spark implementiert. In den folgenden Abschnitten dieses Kapitels werden die für einen Vergleich festgelegten Rahmenbedingungen beschrieben.

4.1 Vergleich

Als absoluter Vergleichsparameter wird die Laufzeit der einzelnen Berechnungen genutzt. Diese wird für Postgres-XL der Veröffentlichung [MMDH19] entnommen und beschreibt die Dauer der Anfrage, welche die Umsetzung der Matrixoperation mit gewissen Parametern darstellt. Für Apache-Spark werden die in [MMDH19] gewählten Matrixeigenschaften übernommen, welche im nächsten Abschnitt noch näher vorgestellt werden. Ebenfalls wird die Laufzeit der Ausführung von Matrixoperationen in diesem System gemessen. Da Apache-Spark auf dem Prinzip der Lazy-Evaluation arbeitet, erweist sich das erfassen

der Laufzeit einer Berechnung allerdings als kompliziert. Berechnungs-, Verteilungs-, Schreib- und Leseschritte laufen parallel ab und sind somit nicht eindeutig trennbar. Folglich muss eine Lösung für einen wissenschaftlichen Vergleich gefunden werden. Eine sinnvolle Lösung ist die Messung der Laufzeit eines vollständigen Spark-Submit, also der benötigten Zeit vom Start der Spark-Umgebung, bis hin zu ihrer Terminierung. Eine Möglichkeit diese kombinierte Laufzeit zu erhalten wäre das Nutzen der mitgelieferten Spark-WebUI. Diese notiert die Ausführung eines Spark-Submits jedoch nur im Sekundenbereich, sofern ein Spark-Job über eine Minute dauert sogar nur im Minutenbereich, was einer gewünschten Genauigkeit widerspricht. Vernünftig erscheint demnach die Laufzeitmessung mit der Shell, wo auch das Spark-Submit ausgeführt wird. Dabei wird vor und nach dem Spark-Submit per Shell-Befehl die Systemzeit abgerufen und deren Differenz als Berechnungszeit in Nanosekunden gespeichert, inklusive Durchlaufnummer, der aktuellen Matrixdimension und dem Namen der Anwendung.

Application ID	Name	Cores	Memory per Executor	Submitted Time	User	State	Duration
	MatrixMultiplication	10	8.0 GB			FINISHED	2.8 min
	MatrixMultiplication	10	8.0 GB			FINISHED	1.9 min
	MatrixMultiplication	10	8.0 GB			FINISHED	2.7 min
	MatrixMultiplication	10	8.0 GB			FINISHED	1.9 min
	MatrixMultiplication	10	8.0 GB			FINISHED	2.7 min
	MatrixMultiplication	10	8.0 GB			FINISHED	2.7 min
	MatrixMultiplication	10	8.0 GB			FINISHED	2.9 min
	MatrixMultiplication	10	8.0 GB			FINISHED	1.9 min

Abb. 4.1: Beispiel für die ungenaue Laufzeitmessung der Spark-WebUI

```
MatrixMultiplication,173555927844,3000000,1
MatrixMultiplication,165885878683,3000000,2
MatrixMultiplication,162819229243,3000000,3
MatrixMultiplication,169645644713,3000000,4
MatrixMultiplication,175007657786,3000000,5
MatrixMultiplication,167175405435,3000000,6
MatrixMultiplication,180896108270,3000000,7
MatrixMultiplication,165002354867,3000000,8
MatrixMultiplication,165883725406,3000000,9
MatrixMultiplication,169485142122,3000000,10
```

Abb. 4.2: Beispieldatei für Ergebnisse der genaueren Laufzeitmessung per Shell (Erste Spalte: Anwendungsname, Zweite Spalte: Laufzeit in Nanosekunden, Dritte Spalte: Dimension der betrachteten Matrix, Vierte Spalte: Versuchsnummer)

4.2 Matrixeigenschaften

Für die **dünnbesetzte Matrix-Vektor-Multiplikation** wird je nach momentaner Vergleichsdimension eine Matrix mit 100000, 500000, 1000000, 2000000 und 3000000 Zeilen und Spalten erstellt. Pro Zeile werden dafür 21 Elemente inklusive Diagonalelement zufällig generiert. Diese Elemente sind dabei mit einer festen Bandbreite zufällig um das Diagonalelement verteilt, welche der dreifachen Dimension der Matrix geteilt durch 40 entspricht. Die Werte der einzelnen Elemente werden zufällig aus einer stetigen Gleichverteilung von Werten zwischen null und eins ausgewählt. Im so genannten i,j,v-Format werden diese Werte dann inklusive Zeilen- und Spaltenindex in einer csv-Datei gespeichert. Zugehörig wird ein dichtbesetzter Spaltenvektor, ebenfalls im i,j,v-Format, in derselben Dimension erstellt, wobei die Werte konstant auf 1 gesetzt sind.

Für die **dichtbesetzte Matrix-Matrix-Multiplikation** wird für jede Kombination aus Zeilen- und Spaltenindex der Zieldimensionen 240, 480, 720, 960, 1200 und 1440 ein zufälliger Wert aus einer stetigen Gleichverteilung zwischen null und eins gewählt und ebenfalls im i,j,v-Format in einer csv-Datei gespeichert. Die zweite Matrix für die Multiplikation wird auf die gleiche Weise erstellt. Die Erstellung erfolgt allerdings auch hier im dünnbesetzten Matrixformat, weil eine Block-Matrix in Apache-Spark am einfachsten aus einer CoordinateMatrix erstellt werden kann [Spa20b], wie es in Kapitel 3 bereits erwähnt wurde.

4.3 Spark-Cluster

Ausgeführt werden die Spark-Jobs auf einem Hadoop-Cluster auf dem Apache-Spark im Standalone-Modus installiert ist. Das Cluster besteht aus sechs virtuellen Maschinen, wobei eine als Gateway bzw. Master und die anderen als Worker agieren. Jeder Knoten des Systems besitzt dabei zwei virtuelle CPU-Kerne mit 2 x 1,9GHz und maximal 8 Gigabyte RAM, sowie einen Sekundärspeicher von einem Terabyte. Zusätzlich ist ein 20 Gigabyte SSD-Speicher für temporäre Dateien konfiguriert worden. Abbildung 4.3 zeigt noch einmal den Aufbau des Clusters:

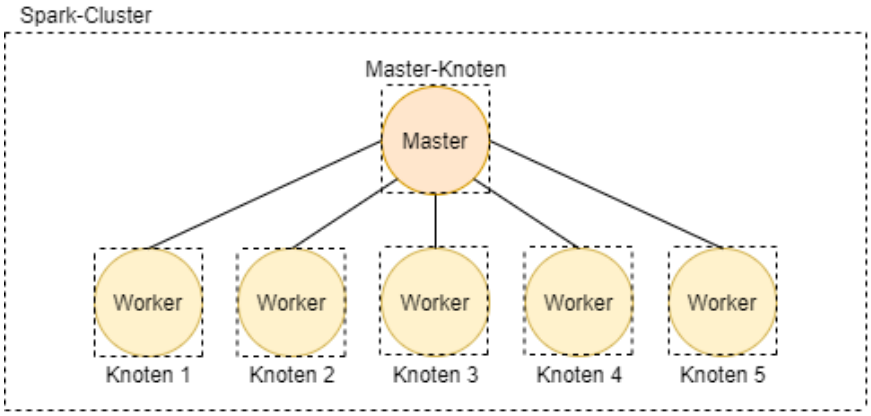


Abb. 4.3: Aufbau des Spark-Clusters

4.4 Postgres-XL-Cluster

Das Postgres-XL-Cluster ist von den Spezifikationen her ähnlich zum Spark-Cluster aufgebaut. Der Master ist in diesem Fall der Global Transaction Manager und die Worker sind die Coordinator mit jeweils 2 zugeordneten Datanodes. Die Systemspezifikationen eines einzelnen Knotens bleiben dabei gleich: ein Knoten des Systems besitzt zwei virtuelle CPU-Kerne mit 2 x 1,9GHz und maximal 8 Gigabyte RAM, sowie einen Sekundärspeicher von einem Terabyte. Ebenfalls sind auch in diesem System 20 Gigabyte SSD-Speicher für temporäre Dateien zur Verfügung gestellt worden. Abbildung 4.4 zeigt abschließend noch den Aufbau des Postgres-XL-Clusters und anschließend wird in Kapitel 5 mit der Umsetzung fortgesetzt:

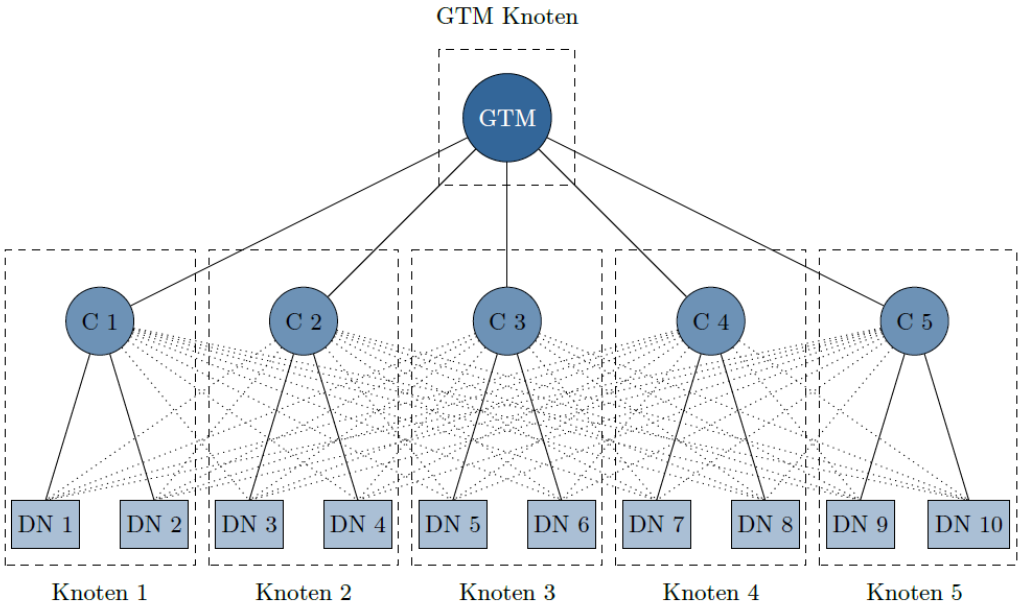


Abb. 4.4: Aufbau des Postgres-XL-Clusters (Abbildung aus [Die18])

Kapitel 5

Umsetzung

In diesem Kapitel soll die Umsetzung der verteilten dicht- und dünnbesetzten linearen Algebra und des Forward-Algorithmus im System Apache-Spark vorgestellt werden. Zusätzlich wird auf notwendige Programmkonstrukte, wie z.B. der Definition einer SparkSession, eingegangen, um für die Implementierungen in Anhang A ein besseres Verständnis zu gewährleisten. Die Umsetzungen in Postgres-XL sind der Veröffentlichung [MMDH19] entnommen und werden deswegen in diesem Kapitel nicht weiter vorgestellt.

5.1 Apache-Spark

Für die Entwicklung und Umsetzung der benötigten Spark-Anwendungen wurde IntelliJ ([Int20]) inklusive eines Build-Tools für die Sprache Scala, siehe [sbt20], mit den in Anhang A.1 gesetzten Dependencies erstellt. Diesen kann man auch die genutzte Scala- und Spark-Version entnehmen.

SparkSession

Für die Erstellung einer Spark-Anwendung ist es primär nötig, ein Scala-Objekt zu erstellen, da dieses die Funktion vergleichbar einer Hauptklasse in Java-Anwendungen und somit auch die main-Methode enthält. Für die Initialisierung von Spark ist es außerdem nötig eine SparkSession zu definieren. In den erstellten Anwendungen wird das wie folgt realisiert:

```
1  val sparkSession = SparkSession.builder()
2  .appName(name)
3  .master(mst)
4  .getOrCreate()
```

Dabei steht die Variable *name* für den Namen der Applikation und *mst* für die URL zum Master des Spark-Clusters.

Einlesen einer CSV-Datei

Apache-Spark liefert mit SparkSQL eine bequeme Möglichkeit zum Einlesen verschiedener Dateiformate mit möglichen Optionen wie z.B. der Übernahme eines Headers der Datei. Dies wird in den folgenden Anwendungen grundlegend wie folgt realisiert:

```
1  val df = sparkSession.read.format("csv")
2    .option("header", "true")
3    .load(path)
```

Die Variable *path* beschreibt dabei den Pfad zur Datei, ausgehend vom Root-Verzeichnis des HDFS.

5.1.1 Dünnbesetzte Matrix-Vektor-Multiplikation

Mit den eben genannten Grundlagen für Apache-Spark wurden zwei Realisierungen für die Matrixmultiplikationen umgesetzt.

Realisierung via MapReduce

Wie in Kapitel 3 schon beschrieben, bietet Apache-Spark für die Repräsentation einer dünnbesetzten verteilten Matrix in entsprechendem i,j,v-Format nur die CoordinateMatrix an. Um eine CoordinateMatrix aus einer csv-Datei im i,j,v-Format zu erstellen, sind die folgenden Schritte notwendig. Zuerst wird die csv-Datei wie im vorher beschriebenen Abschnitt eingelesen. Anschließend werden die Typen der Spalten an die in den MatrixEntrys enthaltenen Datentypen angeglichen. Abschließend werden die nun passenden Datentypen durch ein *map*() in MatrixEntrys umgewandelt. Durch Parallelisierung dieser resultierenden MatrixEntrys wird dann eine entsprechende CoordinateMatrix erstellt.

```
1  val tempDF = df
2    .withColumn("i", df("i").cast("long"))
3    .withColumn("j", df("j").cast("long"))
4    .withColumn("v", df("v").cast("double"))
5  val corMat = new CoordinateMatrix(tempDF.rdd
6    .map(m => MatrixEntry(m.getLong(0), m.getLong(1), m.getDouble(2))))
```

Zwar lässt sich eine CoordinateMatrix sehr einfach in jeden anderen Datentyp einer verteilten Matrix umwandeln, jedoch sind diese ausschließlich für dichtbesetzte Darstellungen einer Matrix geeignet. Eine Multiplikation einer CoordinateMatrix ist in MLlib nicht implementiert. Eine Lösung, wegen mangelnder Umsetzung in MLlib wurde auch schon in [Med16] diskutiert. Die resultierende Implementierung der dünnbesetzten Matrixmultiplikation erfolgt zwischen zwei Matrizen vom Typ CoordinateMatrix und folgt dabei dem Ansatz von MapReduce:

```
1  val M_ = leftMatrix.entries
```

```
2     .map({ case MatrixEntry(i, j, v) => (j, (i, v)) })
3   val N_ = rightMatrix.entries
4     .map({ case MatrixEntry(j, k, w) => (j, (k, w)) })
5   val productEntries = M_
6     .join(N_)
7     .map({ case (_, ((i, v), (k, w))) => ((i, k), (v * w)) })
8     .reduceByKey(_ + _)
9     .map({ case ((i, k), sum) => MatrixEntry(i, k, sum) })
10  new CoordinateMatrix(productEntries)
```

Dabei werden zuerst die Spalten aus dem Datentyp der MatrixEntry extrahiert. Die Zeilen- und Spaltenindexspalten der linken Matrix werden vertauscht, danach werden die zweite und dritte Spalte beider Matrizen zu jeweils einem Tupel umgeformt. Anschließend wird eine Join-Operation auf dem Spaltenindex der ersten Matrix und dem Zeilenindex der zweiten Matrix durchgeführt und im anschließenden *map*() gedroppt. Im selbigen mapping-Prozess werden die Werte der jeweiligen vereinten Einträge multipliziert und anschließend für den Schlüssel, bestehend aus einem Tupel aus Zeilenindex der ersten Matrix und Spaltenindex der zweiten Matrix, per reduce-Funktion aufsummiert. Im letzten mapping-Schritt wird das Schlüsseltupel wieder zu eigenständigen Werten in den Datentyp der MatrixEntry umgeformt und anschließend wieder in eine CoordinateMatrix umgewandelt.
Für diese Implementierung ist es nicht nur nötig, auch den Vektor als CoordinateMatrix darzustellen. Sondern es ist auch eine Umformung der CoordinateMatrix zurück in RDDs notwendig, um auf Operationen wie z.B. reduceByKey() zugreifen zu können. Um nun eine Berechnung dieser Matrixmultiplikation trotz LazyEvaluation zu erzwingen, erfolgt in der entsprechenden Implementierung in Anhang A.2 ein *count*() der MatrixEntrys der resultierenden CoordinateMatrix. Eine Ausgabe des Matrixprodukts ist im Vergleich nämlich nicht vorgesehen.

Realisierung via SparkSQL

Die Realisierung der Matrix-Vektor-Multiplikation per MapReduce erweist sich durch die unnötigen Transformationen, das fehlende Ausnutzen der durch das Einlesen der csv-Dateien schon vorhandenen DataFrames und einer Join-Operation im RDD-Kontext als nicht ganz optimal. Das Einsparen der Transformation in eine CoordinateMatrix, die folgliche Vermeidung einer einhergehenden Realisierung auf RDDs und das schlussendliche Ausnutzen der vorhandenen DataFrames wäre somit eine attraktive Alternative. Nach dem Vorbild der dünnbesetzten Matrix-Vektor-Multiplikation in SQL erfolgt nun die Realisierung auf den DataFrames selbst:

```
1   import org.apache.spark.sql.functions._
2   import sparkSession.implicits._
3
4   val lDf = leftDf
5     .withColumn("t",leftDf("v"))
```

```
6       .select("j","i","t")
7
8     val rDf = rightDf
9       .withColumnRenamed("i","k")
10      .withColumnRenamed("j","l")
11      .withColumnRenamed("v","m")
12
13    val joinedDf = lDf.join(rDf).where(lDf("j") === rDf("k"))
14
15    val tempDf = joinedDf
16      .withColumn("v",joinedDf("t") * joinedDf("m"))
17      .select("i","l", "v")
18
19    val finalDf = tempDf.withColumnRenamed("l","j")
20
21    finalDf.groupBy("i","j").agg(sum($"v"))
22      .withColumnRenamed("sum(v)","v")
```

Dazu erfolgt zuerst ein Import der entsprechenden benötigten Bibliotheken. Analog zur MapReduce-Variante wird auch hier bei der ersten Matrix die Zeilen- und Spaltenindexspalte vertauscht, realisiert durch ein *select*(). Die Wertspalte v wird außerdem in t umbenannt. Um einen eindeutigen *join*() durchführen zu können, werden nun auch die Spalten des zweiten DataFrames umbenannt. Anschließend erfolgt ein *join*() auf der ersten Spalte der neuen DataFrames. Danach wird eine zusätzliche Spalte mit dem Produkt der gejointen Werte-Spalten erstellt und anschließend nur die wichtigen Spalten für den neuen Zeilenindex, Spaltenindex und Wert selektiert. Weiterhin werden die Einträge nach Zeilen- und Spaltenindex gruppiert und die Werte für gleiche Indexkombinationen aufsummiert. Eine letzte Umbenennung der Wertespalte zurück zu v erfolgt deshalb wieder, da Spark diese automatisch und entsprechend der ausgeführten *sum*()-Operation ungewollt umbenennt.

Forward-Algorithmus

Trotz der unterschiedlichen Herangehensweisen der dünnbesetzten Matrix-Vektor-Multiplikation wurden in Anhang A.5 und A.6 Beispielimplementierungen des Forward-Algorithmus für die jeweiligen vorher genannten Realisierungen implementiert, welche aber nicht in großem Maße vorgestellt werden sollen, da es nicht zum ausgiebigen Testen dieser speziellen Implementierungen gekommen ist. In Spark selbst wurde von verschiedenen Personen bis heute zwar das Hidden-Markov-Modell designt[1] oder es wurden sowohl das Hidden-Markov-Modell, als auch Algorithmen für entsprechende Problematiken mit dem Zusatz implementiert, [2] dass diese entweder abgelehnt oder nur für dichtbesetzte und somit Probleme in einer für uns uninteressanten Größe umgesetzt wurden - meistens noch mittels externer Bibliotheken.

[1] https://issues.apache.org/jira/browse/SPARK-17716 aufgerufen am 2020-03-23

[2] https://github.com/skrusche63/spark-intent/tree/master/src/main/scala/de/kp/scala/hmm aufgerufen am 2020-03-23

Die Beispielimplementierungen für den Forward-Algorithmus nutzen die vorher genannten Varianten der dünnbesetzten Matrix-Vektor-Multiplikation. So wird für die jeweilige Umsetzung via MapReduce und SparkSQL auch nur die entsprechende Umsetzung genutzt. Die folgenden, selbst definierten Methoden müssen für ein umfangreiches Verständnis der Implementierungen jedoch noch näher erläutert werden:

- **columnSelector():** Diese Methode übernimmt ein DataFrame im i,j,v-Format sowie eine Zahl für die gewünschte Spalte. Sie gibt nur jene Einträge des DataFrames zurück, welche den gewünschten Spaltenindex haben. Diese Methode wird sowohl in der Initialisierung als auch im Rekursionsschritt des Algorithmus benötigt.
- **elemMultiply():** Die Methode übernimmt zwei DataFrames im i,v-Format, bildet das Hadamard-Produkt und gibt dieses als neues DataFrame zurück. Im i,v-Format deshalb, da in der Umsetzung nur zwei Vektoren elementweise multipliziert werden müssen. Die Methode findet im Rekursionsschritt des Algorithmus Anwendung.
- **cooMatrixTranspose():** Diese Methode übernimmt ein DataFrame im i,j,v-Format, vertauscht die Spalten i und j und benennt diese anschließend wieder um. Damit bleibt der i,j,v-Header erhalten und das neue DataFrame wird zurück gegeben. Diese Methode wird im Rekursionsschritt des Algorithmus benötigt.
- **ivToIjvDf():** Diese Methode wandelt einen Vektor im i,v-Format in einen Vektor im i,j,v-Format durch Hinzufügen einer j-Spalte mit 0-Werten um. Dieses hat keine Auswirkung auf folgende Berechnungen, da dieses DataFrame anschließend im Rekursionsschritt zu einem Zeilenvektor transponiert und dann mit einem weiteren DataFrame multipliziert wird, sodass im Ergebnis sowieso wieder ein Vektor entsteht.

Die weiteren Methoden sind nur für das Einlesen von csv-Dateien zuständig. Dabei unterscheiden sich die Methoden nur im Aufbau der einzulesenden Datei und im resultierenden Format. Die vorgestellten Methoden sind der Implementierung via SparkSQL entnommen. Die Realisierung via MapReduce benötigt noch weitere Methoden zur Umwandlung von DataFrames in den Datentyp der CoordinateMatrix und zurück, ist ansonsten aber ähnlich aufgebaut.

5.1.2 Dichtbesetzte Matrix-Matrix-Multiplikation

Für die dichtbesetzte Matrix-Matrix-Multiplikation stehen von MLlib aus einige Datentypen zur Verfügung. Da die Umsetzung allerdings analog zu [MMDH19] für blockweise verteilte Matrizen mit einem 3x3-Grid durchgeführt werden soll, bietet sich der Datentyp BlockMatrix an. Dabei werden analog zur dünnbesetzten Matrixmultiplikation via MapReduce die csv-Dateien als DataFrame eingelesen und direkt in eine CoordinateMatrix umgewandelt. Abschließend erfolgt die Umwandlung der Matrizen des Typs CoordinateMatrix in den Datentyp BlockMatrix. Sie werden anschließend durch eine von MLlib bereitgestellte Operation multipliziert:

```
1  val bMatP1 = cooMatP1.toBlockMatrix(rowsPerBlock,colsPerBlock)
2  val bMatP2 = cooMatP2.toBlockMatrix(rowsPerBlock,colsPerBlock)
```

```
3
4   val newMat = bMatP1.multiply(bMatP2)
5   newMat.validate()
```

Die Variablen rowsPerBlock und colsPerBlock werden dabei so gewählt, dass ein 3x3-Grid entsteht. Die resultierende Matrix der Multiplikation wird anschließend per *validate*() validiert, um die Berechnung trotz LazyEvaluation durchzuführen. Eine Ausgabe des Matrixprodukts ist im Vergleich nicht vorgesehen. Außerdem ist zu beachten, dass die Standardgröße der Blocks in Apache-Spark 1024x1024 beträgt und der Vergleich somit eher niedrigdimensional für Apache-Spark angesiedelt ist.

Erstellen einer jar-Datei

Um eine für den Spark-Submit notwendige jar-Datei aus dem vorher geschriebenen Programmcode zu erstellen, ist es notwendig, im Projektverzeichnis nach erfolgreicher Installation von [sbt20] den Befehl *sbt package* in der Kommandozeile auszuführen. Dann wird innerhalb des target-Verzeichnisses im Projektordner eine jar-Datei erstellt, welche in das HDFS des Clusters geladen werden muss.

Spark-Submit

Der Spark-Submit Befehl kann dann innerhalb der Kommandozeile wie folgt auf dem entsprechenden Cluster mit der Spark-Installation ausgeführt werden, sofern die entsprechende Umgebungsvariable für einen spark-submit gesetzt ist:

```
1   spark-submit --class NameDesScalaObjekts PfadZurJarDatei.jar
```

Die Master-Adresse kann dabei vernachlässigt werden, da diese bereits în der Initialisierung der Spark-Session gesetzt wurde.

Benchmarking

Das Benchmarken der einzelnen Implementationen erfolgt dann per Shell-Script folgend am Beispiel der dünnbesetzten Matrix-Vektor-Multiplikation inklusive der Datenerstellung der in Anhang B genannten Python-Skripte zur Testdatenerstellung:

```
1   #!/usr/bin/bash
2   dimensionArray=(dim1 dim2 dim3 dim4)
3
4   for i in "${dimensionArray[@]}"
5   do
6     #Generate a Vector for current Dimension
7     python3.6 SparseVectorGenerator.py $i
8
9     #delete current vector in hdfs and put new one in
```

```
10      /usr/local/hadoop/hadoop/bin/hdfs dfs -rm /mmt/m2.csv
11      /usr/local/hadoop/hadoop/bin/hdfs dfs -put m2.csv /mmt/m2.csv
12
13      #for-loop for 10 measurements
14      for j in 1 2 3 4 5 6 7 8 9 10
15      do
16
17        #generate a matrix with 20+diag elem entries per row with dimension
          of i x i
18        python3.6 sparseMatrixGenerator.py 20 $i
19
20        #delete current matrix and put new matrix in
21        /usr/local/hadoop/hadoop/bin/hdfs dfs -rm /mmt/m1.csv
22        /usr/local/hadoop/hadoop/bin/hdfs dfs -put m1.csv /mmt/m1.csv
23
24        #initial time measure point
25        ts=$(date +%s%N)
26
27        #spark-submit
28        /usr/local/spark-2.4.0/bin/spark-submit --class MatrixMult
          MatrixMult.jar
29
30        #write 2nd time measure - 1st time measure with dimension and
          measurement-number to a csv
31        echo "MatrixMultiplication,$(($(date +%s%N) - $ts)),$i,$j" >>
          Benchmark.csv
32      done
33    done
```

Aus diesem Skript ist ersichtlich, dass die Datenerstellung und Zeitmessung für jede Dimension zehnmalig durchgeführt wird. Für die verschiedenen Realisierungen der dünnbesetzten Matrix-Vektor-Multiplikation werden jeweils nur die jar-Dateien ausgetauscht.

Kapitel 6

Ergebnisse

In diesem Abschnitt sollen die Ergebnisse von Apache-Spark und Postgres-XL veranschaulicht und verglichen werden. Die konkreten Ergebnistabellen für Apache-Spark befinden sich in Anhang B, die Ergebnisse von Postgres-XL sind [MMDH19] entnommen.

6.1 Dünnbesetzte Matrix-Vektor-Multiplikation

Im Vordergrund dieser Arbeit steht die Lösung dünnbesetzter Problemstellungen. Deswegen werden die Ergebnisse der dünnbesetzten Matrix-Vektor-Multiplikation auch zuerst betrachtet, siehe Abbildung 6.1. Die Graphen stellen die Mittelwerte der Zeitmessungen der verschiedenen Ansätze in Apache-Spark dar. Die Umsetzung der dünnbesetzten Matrix-Vektor-Multiplikation via CoordinateMatrix, dem von MLlib bereitgestellten Datentyp für dünnbesetzte verteilte Matrizen, ist somit langsamer als die Umsetzung via SparkSQL und auf DataFrames direkt. Begründen könnte man diese Beobachtung durch die zusätzlichen Umwandlungen in eine CoordinateMatrix, RDD's, bzw. wieder zurück und durch die kostenintensive Funktion $reduceByKey()$. Ein $join()$ erfolgt in beiden Realisierungen, einmal im RDD-Kontext und einmal via SparkSQL. Die Funktion $reduceByKey()$ wird in der SQL-Variante durch ein $groupBy()$ und ein $sum()$ realisiert. Ein Blick auf die Spark-WebUI des jeweiligen Spark-Jobs gibt dabei eine bessere Einsicht auf die verschiedenen Stages und widerlegt erstere Annahmen, da die Umbenennungen und Umwandlungen in verschiedene Datentypen nicht ins Gewicht zu fallen scheinen. Der eigentliche Grund für die Zeitdifferenz liegt in der Größe der notwendigen Schreib- und Leseoperationen des Shuffles. So hatte die Implementierung via MapReduce ein fast 50% größeres Shuffle-Write als die Implementierung via SparkSQL (auf DataFrames direkt). Die Dokumentation von Apache-Spark liefert aber eher wenig Aufschluss über selbige Funktionen und deren Umsetzung, was weitere Betrachtungen zur Performanz, sofern Erfahrungen zu System und dessen Funktionsweise fehlen, zur Spekulation werden lassen. Es lässt sich lediglich sagen, dass die Umsetzung auf DataFrames mittels SparkSQL in den getesteten Dimensionen eine bessere Laufzeit aufwies. In Abbildung 6.2 werden nun die Ergebnisse für Postgres-XL visualisiert. Auch in diesen Graphen werden die Mittelwerte mehrerer Zeitmessungen dargestellt. In der Umsetzung in Postgres-XL wurden 3 verschiedene Arten der Partitionierung von Matrizen getestet. Range und RoundRobin wurden bereits in Kapitel 2 erklärt und Multiquery beschreibt letztendlich

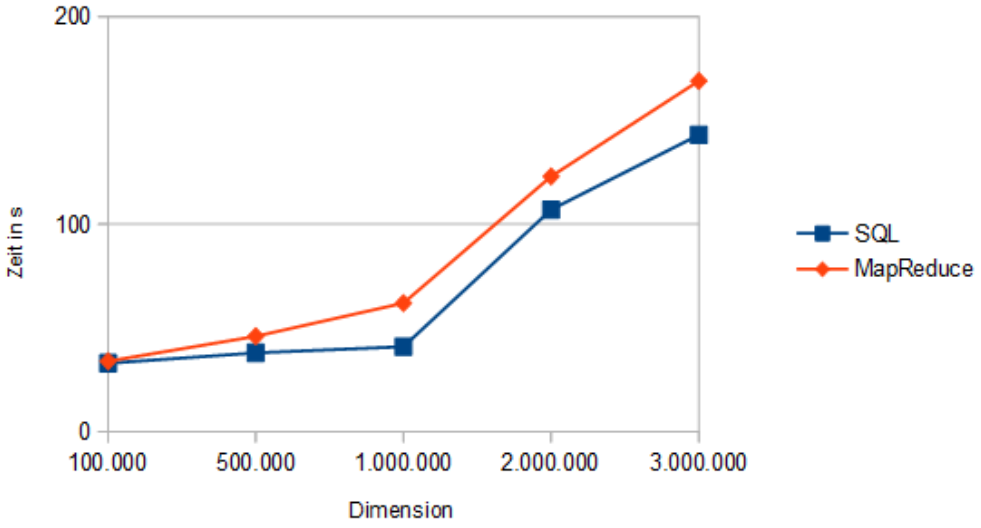


Abbildung 6.1: Dünnbesetzte Matrix-Vektor-Multiplikation in Apache-Spark

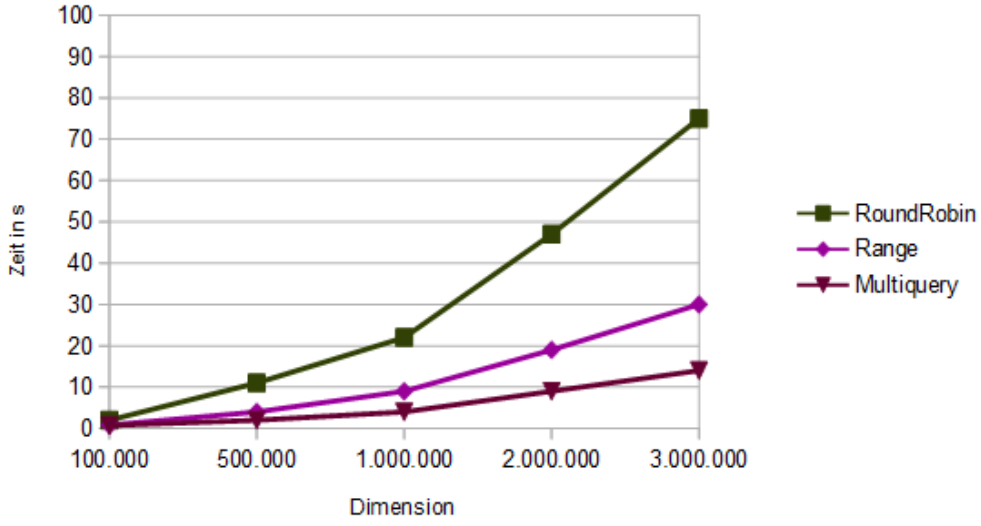


Abbildung 6.2: Dünnbesetzte Matrix-Vektor-Multiplikation in Postgres-XL

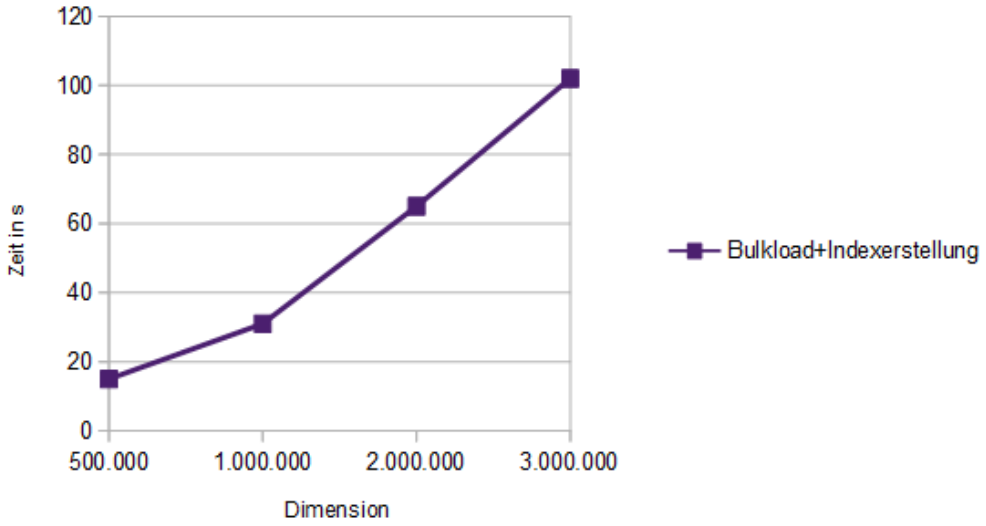


Abbildung 6.3: Zeitmessung für Bulkload + Indexerstellung

ebenfalls eine Range-Partitionierung, die aber durch mehrere simultane Anfragen umgesetzt wird. Zur weiteren Einsicht über die Herkunft der Ergebnisse sei auch nochmal auf [MMDH19] verwiesen.
Bei den Betrachtungen der Abbildungen 6.1 und 6.2 lässt sich deutlich erkennen, dass die Berechnung in Postgres-XL schneller verläuft. Da sich in Apache-Spark aber Lese-, Schreiboperationen und die Berechnung überlappen, ist für einen Vergleich noch keine Fairness gegeben. Um genannte Prozesse aufzuwiegen, ist eine zusätzliche Betrachtung der Ladezeit der Daten in eine Datenbank inklusive der Zeit zur Erstellung eines Index in Postgres-XL sinnvoll. In der Abbildung 6.3 sind die Zeiten für einen Bulkload der Daten in eine Datenbank inklusive der anschließenden Erstellung eines Index in Postgres-XL veranschaulicht. Auch in Postgres-XL nehmen die Lese- und Schreiboperationen einen Großteil der Gesamtzeit ein. Unter Berücksichtigung dieses Fakts und der Addition der Laufzeit dieser Operationen ergeben sich die in Abbildung 6.4 dargestellten Graphen.
Trotz der Hinzunahme der Ladezeiten ist die Berechnung in Postgres-XL immer noch schneller als in Apache-Spark, mit der Ausnahme der Verteilung einer Matrix via RoundRobin. Schlussfolgernd lässt sich sagen, dass ein paralleles Datenbanksystem wie Postgres-XL für die Berechnung von dünnbesetzten Problemen sogar schneller sein kann als Big-Data-Frameworks. Dabei sollte bei der Wahl des Systems jedoch immer auch eine Vorbetrachtung durchgeführt werden, ob die Daten überhaupt in eine Datenbank eingepflegt oder direkt in eine Analysesoftware geladen werden sollen. Liegen die Daten bereits in einem parallelen Datenbanksystem wie Postgres-XL vor, so kann eine Berechnung in selbigem System sogar zu schnelleren Ergebnissen führen (siehe Abbildung 6.2).

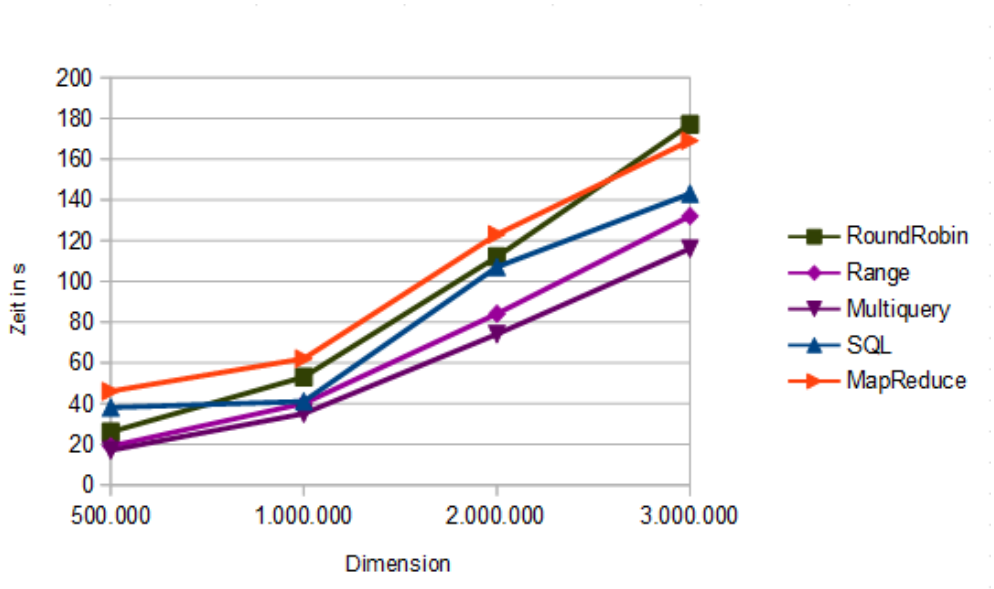


Abbildung 6.4: Dünnnbesetzte Matrix-Vektor-Multiplikation in den Vergleichssystemen

6.2 Dichtbesetzte Matrix-Matrix-Multiplikation

Ein weiterer Vergleichspunkt ist die dichtbesetzte Matrix-Matrix-Multiplikation wegen ihrer Relevanz im Bereich des maschinellen Lernens. Apache-Spark liefert via MLlib eine Umsetzung für die Multiplikation von blockweise verteilten Matrizen mit dem Datentyp der BlockMatrix. Die Mittelwerte wiederholter Zeitmessungen sind in Abbildung 6.5 abgebildet.
Die ähnlichen und schwankenden Ergebnisse lassen sich dadurch begründen, dass die Standard-Blockgröße in Apache-Spark bei 1024x1024 liegt. Der Vergleich ist somit niedrigdimensional in Bezug auf Apache-Spark angesetzt, macht aber einen Vergleich mit Postgres-XL umso interessanter, welches für selbige Dimensionen die Ergebnisse in Abbildung 6.6 liefert. Normal beschreibt dabei eine Blockpartitionierung mit einer SQL-Anfrage, MultiUser9 hingegen eine Blockpartitionierung mittels 9 Subanfragen. Zu sehen ist ein drastischer Zuwachs der Berechnungszeit in höheren Blockdimensionen für beide Ansätze. Auf eine zusätzliche Messung der Zeit für das Laden der Daten inklusive Indexerstellung wurde einmal aus diesem Grund und wegen Abbildung 6.7 verzichtet. Es ist deutlich zu erkennen, dass bereits in relativ geringen Dimensionen die Umsetzung in Apache-Spark eine merklich bessere Performanz liefert als die Umsetzungen in Postgres-XL. Dazu muss man beachten, dass in diesem Test die Daten in Postgres-XL bereits im Datenbanksystem vorliegen und es sich somit nur um die reine Berechnungszeit handelt.

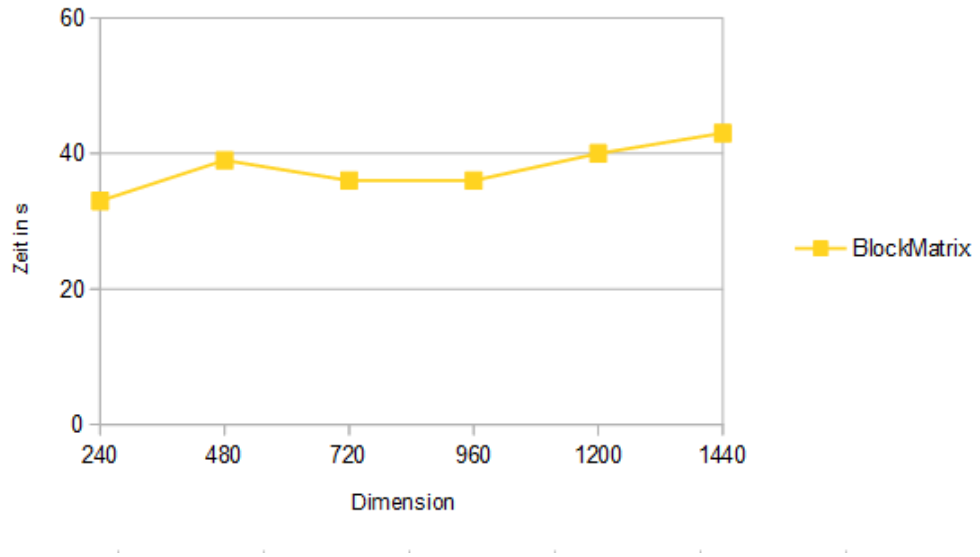


Abbildung 6.5: Dichtbesetzte Matrix-Matrix-Multiplikation in Apache-Spark

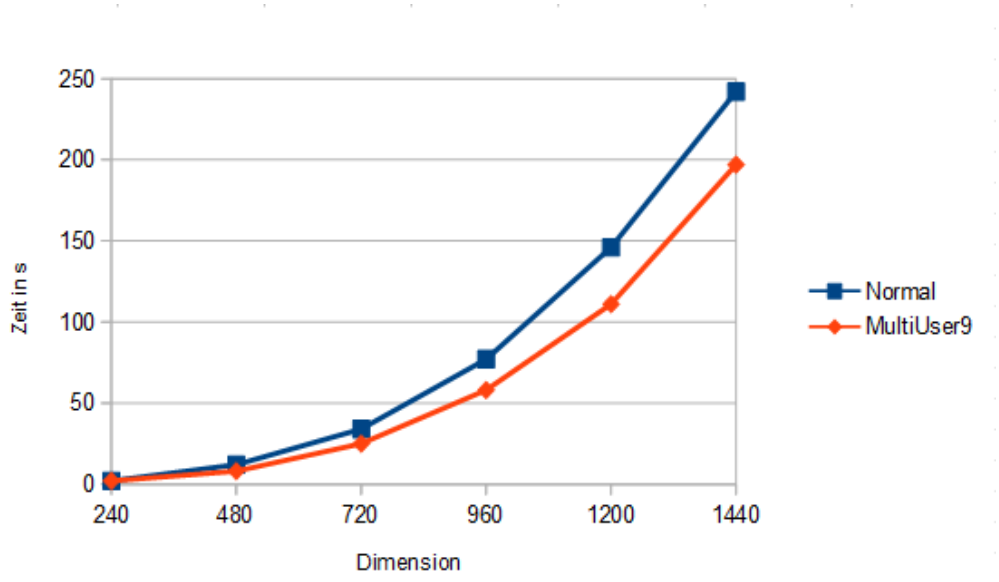


Abbildung 6.6: Dichtbesetzte Matrix-Matrix-Multiplikation in Postgres-XL

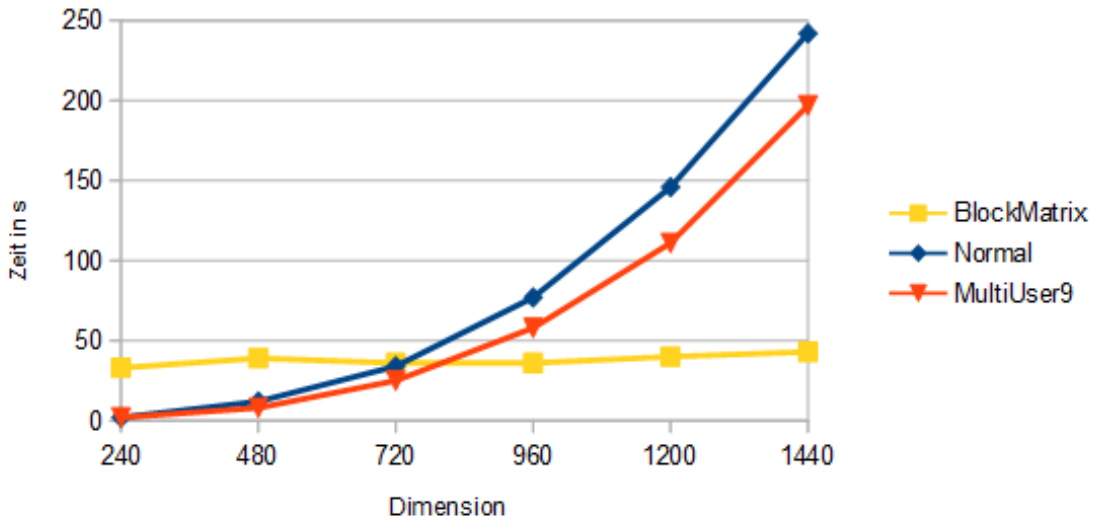


Abbildung 6.7: Dichtbesetzte Matrix-Matrix-Multiplikation in den Vergleichssystemen)

Kapitel 7

Fazit und Ausblick

Aus den Betrachtungen verschiedener Umsetzungen verteilter dicht- und dünnbesetzter Problemstellungen in parallelen Datenbanksystemen und Big-Data-Frameworks lässt sich aus den Ergebnissen und den entsprechenden Visualisierungen ableiten, dass in Bezug auf die dünnbesetzte lineare Algebra das Nutzen eines parallelen Datenbanksystems von Vorteil sein kann. Sollen die Daten jedoch nicht in einer Datenbank manifestiert werden, so könnten entsprechende Umsetzungen mit etwas höheren Laufzeiten auch in Big-Data-Frameworks vorgenommen werden. In Bezug auf die dichtbesetzte lineare Algebra stehen Big-Data-Frameworks an erster Stelle. Es wurde dabei sichtbar, dass in Apache-Spark das Hauptaugenmerk eindeutig auf dichtbesetzten Problemstellungen und deren Lösung liegt, auch zu sehen an den verschiedenen Formaten und Realisierungen dichtbesetzter verteilter Matrizen innerhalb des Systems selbst. Andererseits lässt sich auch für besagtes System ein Mangel an Lösungen für verteilte dünnbesetzte Probleme feststellen, eventuell auch der mangelnden Existenz solcher Probleme im Big-Data-Bereich geschuldet. Abschließend lässt sich sagen, dass sowohl Big-Data-Frameworks als auch parallele Datenbanksysteme absolut ihre Existenzberechtigung und je nach Problemstellung ihre Vorzüge haben. Überraschend ist hingegen, dass ein paralleles Datenbanksystem eine bessere Alternative im Bereich der dünnbesetzten linearen Algebra gegenüber Systemen mit explizitem Fokus auf maschinellem Lernen besitzen kann - und dies trotz zusätzlicher Betrachtung des Bulkloads und der Indexerstellung. Ich denke, dass man in der Zukunft auch noch häufiger auf parallele Datenbanksysteme zurückgreifen wird, da sich diese in einigen Bereichen als sehr vorteilhaft erweisen und generell das attraktive Konzept der Verarbeitung nahe an den Daten ermöglicht.

Im Rahmen dieser Arbeit wurde der Forward-Algorithmus in Apache-Spark in mehreren Versionen umgesetzt. In Bezug auf die dünnbesetzte lineare Algebra scheint eine Nutzung von Postgres-XL zwar von Vorteil zu sein, jedoch konnte die Umsetzung eines Forward-Algorithmus innerhalb dieses verteilten Datenbanksystems aus Zeitgründen nicht mehr realisiert werden. Ein weiterer Vergleich wäre vor allem deshalb interessant, weil der Forward-Algorithmus nach dem Prinzip der dynamischen Programmierung in Apache-Spark umgesetzt wurde. Inwiefern sich dieses Prinzip auch in Postgres-XL umsetzten lässt und ob es wesentlichen Einfluss auf die Laufzeit nimmt, wäre somit eine weitere Betrachtung wert. Allgemein erscheint es auch sinnvoll, dass weitere Algorithmen des maschinellen Lernens in Bezug auf dünnbesetzte Problemstellungen in verschiedenen Umsetzungen verglichen werden. Als explizite Beispiele könnten

dafür unter anderem der Viterbi- und Baum-Welch-Algorithmus gewählt werden, die ebenfalls für die Lösung von Problemklassen von Hidden-Markov-Modellen entwickelt wurden.

Literaturverzeichnis

[And16] Andris, Julia: *Halbleiterlösungen für die Automobil- und Industrieelektronik*, 2016. https://www.elektronikpraxis.vogel.de/hall-sensoren-im-automobilbau-loesungen-fuer-die-entwicklung-a-536711/ aufgerufen am 2020-02-10.

[BHWM12] Burg, Klemens, Herbert Haf, Friedrich Wille und Andreas Meister: *Höhere Mathematik für Ingenieure Band II: Lineare Algebra*. Vieweg+Teubner Verlag, 2012.

[DFS+18] Dietrich, Daniel, Ole Fenske, Stefan Schomacker, Philipp Schweers und Andreas Heuer: *Stonebraker versus Google: 2-0 Scores in Rostock - A Comparison of Big Data Analytics Environments (Stonebraker gegen Google: Das 2: 0 fällt in Rostock)*. In: Klassen, Gerhard und Stefan Conrad (Herausgeber): *Proceedings of the 30th GI-Workshop Grundlagen von Datenbanken, Wuppertal, Germany, May 22-25, 2018*, Band 2126 der Reihe *CEUR Workshop Proceedings*, Seiten 101–107. CEUR-WS.org, 2018.

[Die18] Dietrich, Daniel: *Vergleich zeilen- und spaltenorientierter DBMS als Basis für die Parallelisierung von Vektorraumoperationen auf einem Cluster-Rechner*, 2018. Universität Rostock, Bachelorarbeit.

[Fis17] Fischer, Gerd: *Lernbuch Lineare Algebra und Analytische Geometrie*. Springer Fachmedien Wiesbaden GmbH, 2017.

[Had20] *Hadoop Wikis*, 2020. https://cwiki.apache.org/confluence/display/hadoopl aufgerufen am 2020-03-01.

[Int20] *IntelliJ IDEA - Leistungsfähige und ergonomische IDE für JVM*, 2020. https://www.jetbrains.com/de-de/idea/ aufgerufen am 2020-03-23.

[Kra19] Kraftfahrtbundesamt: *Jahresbilanz des Fahrzeugbestandes am 1. Januar 2019*, 2019. https://www.kba.de/DE/Statistik/Fahrzeuge/Bestand/bestand_node.html aufgerufen am 2020-02-10.

[KT18] Kelleher, John D. und Brendan Tierney: *Data Science*. The MIT Press, 2018.

[LRU19] Leskovec, Jure, Anand Rajaraman und Jeffrey D. Ullman: *Mining of Massive Datasets*. Cambridge University Press, 2019.

[Med16] *Scalable Sparse Matrix Multiplication in Apache Spark*, 2016. https://medium.com/balabit-unsupervised/scalable-sparse-matrix-multiplication-in-apache-spark-c79e9ffc0703 aufgerufen am 2020-03-23.

[Mil07] Million, Elizabeth: *The hadamard product*. Course Notes, 3(6), 2007.

[MMDH19] Marten, Dennis, Holger Meyer, Daniel Dietrich und Andreas Heuer: *Sparse and Dense Linear Algebra for Machine Learning on Parallel-RDBMS Using SQL*. Open Journal of Big Data (OJBD), 5(1):1–34, 2019.

[Pos20a] *Overview: Postgres-XL*, 2020. https://www.postgres-xl.org/overview/ aufgerufen am 2020-03-23.

[Pos20b] *PostgresXL*, 2020. https://www.postgres-xl.org/ aufgerufen am 2020-02-10.

[Pos20c] *What is Postgres-XL?*, 2020. https://www.postgres-xl.org/documentation/intro-whatis-postgres-xl.html aufgerufen am 2020-03-01.

[Rab89] Rabiner, Lawrence R: *A tutorial on hidden Markov models and selected applications in speech recognition*. Proceedings of the IEEE, 77(2):257–286, 1989.

[SAD+10] Stonebraker, Michael, Daniel Abadi, David J. DeWitt, Sam Madden, Erik Paulson, Andrew Pavlo und Alexander Rasin: *MapReduce and Parallel DBMSs: Friends or Foes?* Communications of the ACM, 53(1):64–71, 2010.

[sbt20] *The interactive build tool*, 2020. https://www.scala-sbt.org/ aufgerufen am 2020-03-23.

[Sch19] Schahinian, David: *Big Data nimmt in der Autoindustrie Fahrt auf*, 2019. https://www.hannovermesse.de/de/news/news-fachartikel/big-data-nimmt-in-der-autoindustrie-fahrt-aufl aufgerufen am 2020-02-10.

[Spa20a] *Apache Spark*, 2020. https://spark.apache.org/ aufgerufen am 2020-02-10.

[Spa20b] *Machine Learning Library Guide*, 2020. https://spark.apache.org/docs/latest/ml-guide.html aufgerufen am 2020-03-01.

[Spa20c] *RDD Programming Guide*, 2020. https://spark.apache.org/docs/latest/rdd-programming-guide.html aufgerufen am 2020-03-01.

[Spa20d] *Spark SQL, DataFrames and Datasets Guide*, 2020. https://spark.apache.org/docs/latest/sql-programming-guide.html aufgerufen am 2020-03-01.

[TT73] Tewarson, Reginald P und Reginald P Tewarson: *Sparse matrices*, Band 69. Academic Press New York, 1973.

[Wen04] Wendemuth, Andreas: *Grundlagen der stochastischen Sprachverarbeitung*. Oldenbourg Wissenschaftsverlag GmbH, 2004.

Anhang A

Realisierungen in Apache-Spark

Alle nachfolgenden Anwendunge die für Apache-Spark geschrieben wurden, sind in der Programmiersprache Scala verfasst worden.

A.1 Spark-Dependencys

Die in den Spark-Anwendungen gesetzten Abhängigkeiten sind in allen Projekten gleich und lauten:

```
1  name := "SparseMatrixMultiplication"
2
3  version := "0.1"
4
5  scalaVersion := "2.11.12"
6
7  // https://mvnrepository.com/artifact/org.apache.spark/spark-core
8  libraryDependencies += "org.apache.spark"  "spark-core" % "2.4.0" %
   "provided"
9  //remove provided for local use only
10
11 // https://mvnrepository.com/artifact/org.apache.spark/spark-sql
12 libraryDependencies += "org.apache.spark"  "spark-sql" % "2.4.0"
13
14 // https://mvnrepository.com/artifact/org.apache.spark/spark-mllib
15 libraryDependencies += "org.apache.spark"  "spark-mllib" % "2.4.0"
```

A.2 Dünnbesetzte Matrixmultiplikation via Map-Reduce

```
1  import org.apache.spark.mllib.linalg.distributed.{CoordinateMatrix,
   MatrixEntry}
2  import org.apache.spark.sql
3  import org.apache.spark.sql.SparkSession
4
5  //memorize the name for it, because it is necessary in the spark-submit
   step
6  object MatrixMult {
7  //variables for master-URL and AppName
8  val mst = "spark://pgxlgtm:7077"
9  val name = "MatrixMultiplication"
10
11 //initialization of SparkSession
12 val sparkSession = SparkSession.builder()
13 .appName(name)
14 .master(mst)
15 .getOrCreate()
16
17
18 //main method:
19 def main(args: Array[String]): Unit = {
20
21   //paths for the csv files
22   val path1 = "/m1.csv"
23   val path2 = "/m2.csv"
24   //converts csv-data to coo-matrices
25   val cooMatP1 = csvToCooMatrix(path1)
26   val cooMatP2 = csvToCooMatrix(path2)
27
28   //matrix multiplication
29   val newMat = coordinateMatrixMultiply(cooMatP1, cooMatP2)
30   newMat.entries.count()
31 }
32
33
34
35 //definition of multiply for coo-matrices
36 def coordinateMatrixMultiply(leftMatrix: CoordinateMatrix,
37 rightMatrix: CoordinateMatrix): CoordinateMatrix = {
```

```
38      val M_ = leftMatrix.entries
39      .map({ case MatrixEntry(i, j, v) => (j, (i, v)) })
40      val N_ = rightMatrix.entries
41      .map({ case MatrixEntry(j, k, w) => (j, (k, w)) })
42      val productEntries = M_
43      .join(N_)
44      .map({ case (_, ((i, v), (k, w))) => ((i, k), (v * w)) })
45      .reduceByKey(_ + _)
46      .map({ case ((i, k), sum) => MatrixEntry(i, k, sum) })
47      new CoordinateMatrix(productEntries)
48    }
49
50
51
52    //convert csv files (with header) containing matrix-entrys to a
      coo-matrix
53    def csvToCooMatrix(path: String): CoordinateMatrix = {
54      val df = sparkSession.read.format("csv")
55      .option("header", "true")
56      .load(path)
57      val tempDF = df
58      .withColumn("i", df("i").cast("long"))
59      .withColumn("j", df("j").cast("long"))
60      .withColumn("v", df("v").cast("double"))
61      val corMat = new CoordinateMatrix(tempDF.rdd
62      .map(m => MatrixEntry(m.getLong(0), m.getLong(1), m.getDouble(2))))
63      corMat //return unnecessary, because last statement gets returned
64    }
65
66
67
68    def cooMatToDf(cooMat: CoordinateMatrix) : sql.DataFrame = {
69      val ivRDD = cooMat.entries.map({ case MatrixEntry(i, j, v) => (i, v) })
70      val newDf = sparkSession.createDataFrame(ivRDD).toDF("i","v")
71      newDf
72      }
73    }
```

A.3 Dünnbesetzte Matrixmultiplikation via SQL

```
1  import org.apache.spark.sql.{DataFrame, SparkSession}
2
3  //memorize the name, for it is necessary in the spark-submit step
4  object MatrixMult {
5
6  //variables for master-URL and AppName
7  val mst = "spark://pgxlgtm:7077"
8  val name = "MatrixMultiplication"
9
10 //initialization of SparkSession
11 val sparkSession = SparkSession.builder()
12 .appName(name)
13 .master(mst)
14 .getOrCreate()
15
16
17 //main method:
18 def main(args: Array[String]): Unit = {
19
20   //paths for the csv files
21   val path1 = "/m1.csv"
22   val path2 = "/m2.csv"
23
24   //converts csv-data to coo-DataFrames
25   val df1 = csvToDf(path1)
26   val df2 = csvToDf(path2)
27
28   //matrix multiplication
29   val newMat = coordinateMatrixMultiply(df1, df2)
30   newMat.count()
31   }
32
33
34 //definition of multiply for coo-matrices
35 def coordinateMatrixMultiply(leftDf: DataFrame,
36 rightDf: DataFrame): DataFrame = {
37   import org.apache.spark.sql.functions._
38   import sparkSession.implicits._
39
```

```
40    val lDf = leftDf
41      .withColumn("t",leftDf("v"))
42      .select("j","i","t")
43
44    val rDf = rightDf
45      .withColumnRenamed("i","k")
46      .withColumnRenamed("j","l")
47      .withColumnRenamed("v","m")
48
49    val joinedDf = lDf.join(rDf).where(lDf("j") === rDf("k"))
50
51    val tempDf = joinedDf
52      .withColumn("v",joinedDf("t") * joinedDf("m"))
53      .select("i","l", "v")
54
55    val finalDf = tempDf.withColumnRenamed("l","j")
56
57    finalDf.groupBy("i","j").agg(sum($"v"))
58      .withColumnRenamed("sum(v)","v")
59    }
60
61  //convert csv files (with header) containing matrix-entrys to a coo-matrix
62  def csvToDf(path: String): DataFrame = {
63    val df = sparkSession.read.format("csv")
64    .option("header", "true")
65    .load(path)
66    df
67    }
68
69  }
```

A.4 Dichtbesetzte Matrixmultiplikation via BlockMatrizen

Im nachfolgenden Code wurde die Blockgröße nicht weiter spezifiziert, sondern durch (rowsPerBlock,colsPerBlock) ersetzt. Die entsprechenden Werte sind den jeweiligen Tabellen und Plots zu entnehmen.

```
1  import org.apache.spark.mllib.linalg.distributed.{CoordinateMatrix,
   MatrixEntry}
2  import org.apache.spark.sql.SparkSession
3
4  //memorize the name for it is necessary in the spark-submit step
5  object MatrixMult {
6  //variables for master-URL and AppName
7  val mst = "spark://pgxlgtm:7077"
8  val name = "MatrixMultiplication"
9
10 //initialization of SparkSession
11 val sparkSession = SparkSession.builder()
12   .appName(name)
13   .master(mst)
14   .getOrCreate()
15
16
17 //main method:
18 def main(args: Array[String]): Unit = {
19
20   //HDFS-paths for the csv files
21   val path1 = "/m1.csv"
22   val path2 = "/m2.csv"
23   //converts csv-data to coo-matrices
24   val cooMatP1 = csvToCooMatrix(path1)
25   val cooMatP2 = csvToCooMatrix(path2)
26
27   val bMatP1 = cooMatP1.toBlockMatrix(rowsPerBlock,colsPerBlock)
28   val bMatP2 = cooMatP2.toBlockMatrix(rowsPerBlock,colsPerBlock)
29
30   val newMat = bMatP1.multiply(bMatP2)
31   newMat.validate()
32 }
33
34
```

```
35 //convert csv files (with header) containing matrix-entrys to a coo-matrix
36 def csvToCooMatrix(path: String): CoordinateMatrix = {
37   val df = sparkSession.read.format("csv")
38     .option("header", "true")
39     .load(path)
40   val tempDF = df
41     .withColumn("i", df("i").cast("long"))
42     .withColumn("j", df("j").cast("long"))
43     .withColumn("v", df("v").cast("double"))
44   val corMat = new CoordinateMatrix(tempDF.rdd
45     .map(m => MatrixEntry(m.getLong(0), m.getLong(1), m.getDouble(2))))
46   corMat
47   }
48 }
```

A.5 Forward-Algorithmus via MapReduce-Matrixmultiplikation

Die Dateien für die Startverteilung pi und Observationsfolge o müssen jeweils in zweispaltiger Form vorliegen - in diesem Fall als i,v-Form bezeichnet. Index i beschreibt für o die Reihenfolge der Observationen und v den beobachteten Zustand. Für pi beschreibt i die möglichen Zustände und v die Wahrscheinlichkeit, dass sich das System zum Startzeitpunkt in Zustand i befindet. A und B sind jeweils Matrizen im i,j,v-Format. Alle diese csv-Dateien müssen im HDFS vorliegen und sind mit entsprechenden Headern versehen.

```
1
2  import org.apache.spark
3  import org.apache.spark.mllib.linalg.distributed.{CoordinateMatrix,
   MatrixEntry}
4  import org.apache.spark.sql
5  import org.apache.spark.sql.functions.lit
6  import org.apache.spark.sql.{DataFrame, SparkSession}
7
8  //memorize the name for it because it is necessary in the spark-submit step
9  object HMM {
10
11   //variables for master-URL and AppName
12   val mst = "spark://pgxlgtm:7077"
13   val name = "HMM"
14   //for local use only
15   //val mst = "local"
16
17   //initialization of SparkSession
18   val sparkSession = SparkSession.builder()
19     .appName(name)
20     .master(mst)
21     .getOrCreate()
22
23   //main method:
24   def main(args: Array[String]): Unit = {
25
26     //csv-paths on the server
27     val o = "/o.csv"
28     val A = "/A.csv"
29     val B = "/B.csv"
30     val pi ="/pi.csv"
31
```

```
32      println(forwardMethod(o, A, B, pi))
33
34    }
35
36
37    def forwardMethod(oPath: String, APath : String, BPath : String,
      piPath: String) : Double = {
38
39      val o = ivFileToArray(oPath)
40      val pi = ivFileToDataFrame(piPath)
41      val B = ijvFileToDataFrame(BPath)
42      val A = ijvFileToCooMatrix(APath)
43
44
45      val oZero = o(0)
46      val B0 = columnSelector(B, oZero)
47      val alp = elemMultiply(B0, pi)
48      val alpList = Array(alp, alp)
49      for (k <- 1 to (o.length - 1)) {
50
51
52        alpList(0) =
        elemMultiply(cooMatToDf(cooMatrixTranspose(cooMatrixMultiply(
53          cooMatrixTranspose(ijvDfToCooMatrix(ivToIjvDf(alpList(1)))), A))),
54          columnSelector(B, o(k)))
55
56        alpList(1) = alpList(0)
57      }
58
59      val newAlpSum = alpList(0).select("v").rdd.map(_
        (0).asInstanceOf[Double]).reduce(_ + _)
60      newAlpSum
61
62    }
63
64
65
66
67    def ivToIjvDf (alp : DataFrame) : sql.DataFrame = {
68      val alpNew = alp.withColumn("j", lit(0))
69        .select("i", "j", "v")
```

```
70      alpNew
71    }
72
73    def cooMatToDf(cooMat: CoordinateMatrix) : sql.DataFrame = {
74      val ivRDD = cooMat.entries.map({ case MatrixEntry(i, j, v) => (i, v) })
75      val newDf = sparkSession.createDataFrame(ivRDD).toDF("i","v")
76      newDf
77    }
78
79    def ijvDfToCooMatrix (df: DataFrame) : CoordinateMatrix = {
80      val tempDF = df
81        .withColumn("i", df("i").cast("long"))
82        .withColumn("j",df("j").cast("long"))
83        .withColumn("v",df("v").cast("double"))
84      new CoordinateMatrix(tempDF.rdd
85        .map(m => MatrixEntry(m.getLong(0),m.getLong(1),m.getDouble(2))))
86    }
87
88    def columnSelector (df: DataFrame, colNumber: String) : sql.DataFrame = {
89      df.select("i","v").filter(df("j").contains(colNumber))
90    }
91
92
93    def elemMultiply(df1 : DataFrame, df2 : DataFrame): sql.DataFrame  = {
94
95      val df1x = df1.withColumnRenamed("v","v1")
96      val df2x = df2.withColumnRenamed("v","v2")
97      val joinedDf = df1x.join(df2x, Seq("i"))
98      val productDf = joinedDf
99        .withColumn("v",joinedDf("v1") * joinedDf("v2"))
100       .select("i","v")
101     productDf
102
103   }
104
105   def cooMatrixTranspose (cooMat : CoordinateMatrix) : CoordinateMatrix = {
106     val cooMatEntriesTransposed = cooMat
107       .entries.map({ case MatrixEntry(i, j, v) => MatrixEntry(j, i, v) })
108     new CoordinateMatrix(cooMatEntriesTransposed)
109   }
110
```

```
111  //definition of multiply for coo-matrices
112  def cooMatrixMultiply(leftMatrix: CoordinateMatrix,
113  rightMatrix: CoordinateMatrix): CoordinateMatrix = {
114    val M_ = leftMatrix.entries
115      .map({ case MatrixEntry(i, j, v) => (j, (i, v)) })
116    val N_ = rightMatrix.entries
117      .map({ case MatrixEntry(j, k, w) => (j, (k, w)) })
118    val productEntries = M_
119      .join(N_)
120      .map({ case (_, ((i, v), (k, w))) => ((i, k), (v * w)) })
121      .reduceByKey(_ + _)
122      .map({ case ((i, k), sum) => MatrixEntry(i, k, sum) })
123    new CoordinateMatrix(productEntries)
124  }
125
126  //convert csv files with header containing matrix-entrys to a coo-matrix
127  def ijvFileToCooMatrix(path: String): CoordinateMatrix = {
128    val df = sparkSession.read.format("csv")
129      .option("header", "true")
130      .load(path)
131    val tempDF = df
132      .withColumn("i", df("i").cast("long"))
133      .withColumn("j",df("j").cast("long"))
134      .withColumn("v",df("v").cast("double"))
135    new CoordinateMatrix(tempDF.rdd
136      .map(m => MatrixEntry(m.getLong(0),m.getLong(1),m.getDouble(2))))
137  }
138  def ijvFileToDataFrame(path: String): spark.sql.DataFrame = {
139    val df = sparkSession.read.format("csv")
140      .option("header", "true")
141      .load(path)
142    val newDF = df
143      .withColumn("i", df("i").cast("long"))
144      .withColumn("j",df("j").cast("long"))
145      .withColumn("v",df("v").cast("double"))
146    newDF
147  }
148  def ivFileToDataFrame(path: String): spark.sql.DataFrame = {
149    val df = sparkSession.read.format("csv")
150      .option("header", "true")
151      .load(path)
```

```
152     val newDF = df
153       .withColumn("i", df("i").cast("long"))
154       .withColumn("v",df("v").cast("double"))
155     newDF
156   }
157
158   def ivFileToArray(path: String): Array[String] = {
159     val df = sparkSession.read.format("csv")
160       .option("header", "true")
161       .load(path)
162     val newDF = df
163       .withColumn("i", df("i").cast("Int"))
164       .withColumn("v",df("v").cast("String"))
165     val oCount = newDF.count().toInt
166     val oArray = Array.ofDim[String](oCount)
167     for (k <- 0 to oCount - 1){
168       val dfArray = newDF.select("v").filter(newDF("i")
169         .contains(k)).take(1)
170       val dfRow = dfArray(0)
171       oArray(k) = dfRow.getString(0)
172     }
173
174     oArray
175   }
176
177 }
```

A.6 Forward-Algorithmus via SparkSQL-Matrixmultiplikation

```
1  import org.apache.spark
2  import org.apache.spark.sql
3  import org.apache.spark.sql.functions.lit
4  import org.apache.spark.sql.{DataFrame, SparkSession}
5
6  //memorize the name for it because it is necessary in the spark-submit step
7  object HMM {
8
9    //variables for master-URL and AppName
10   val mst = "spark://pgxlgtm:7077"
11   val name = "HMM"
12
13   //initialization of SparkSession
14   val sparkSession = SparkSession.builder()
15   .appName(name)
16   .master(mst)
17   .getOrCreate()
18
19   //main method:
20   def main(args: Array[String]): Unit = {
21
22
23     //csv-paths on the server
24     val o = "/o.csv"
25     val A = "/A.csv"
26     val B = "/B.csv"
27     val pi ="/pi.csv"
28
29
30     println(forwardMethod(o, A, B, pi))
31
32   }
33
34
35   def forwardMethod(oPath: String, APath : String, BPath : String,
     piPath: String) : Double = {
36
37     val o = ivFileToArray(oPath)
38     val pi = ivFileToDataFrame(piPath)
```

```
39      val B = ijvFileToDataFrame(BPath)
40      val A = ijvFileToDataFrame(APath)
41
42
43      val oZero = o(0)
44      val B0 = columnSelector(B, oZero)
45      val alp = elemMultiply(B0, pi)
46      val alpList = Array(alp, alp)
47      for (k <- 1 to (o.length - 1)) {
48
49
50        alpList(0) =
        elemMultiply(cooMatrixTranspose(coordinateMatrixMultiply(
51          cooMatrixTranspose(ivToIjvDf(alpList(1))), A)), columnSelector(B,
          o(k)))
52        alpList(1) = alpList(0)
53      }
54
55      val newAlpSum = alpList(0).select("v").rdd.map(_
        (0).asInstanceOf[Double]).reduce(_ + _)
56      newAlpSum
57
58    }
59
60
61
62
63    def ivToIjvDf (alp : DataFrame) : sql.DataFrame = {
64      val alpNew = alp.withColumn("j", lit(0))
65      .select("i", "j", "v")
66      alpNew
67    }
68
69    //definition of multiply for coo-matrices
70    def coordinateMatrixMultiply(leftDf: DataFrame,
71    rightDf: DataFrame): DataFrame = {
72      import org.apache.spark.sql.functions._
73      import sparkSession.implicits._
74
75      val lDf = leftDf
76        .withColumn("t",leftDf("v"))
```

```
77        .select("j","i","t")
78
79      val rDf = rightDf
80        .withColumnRenamed("i","k")
81        .withColumnRenamed("j","l")
82        .withColumnRenamed("v","m")
83
84      val joinedDf = lDf.join(rDf).where(lDf("j") === rDf("k"))
85
86      val tempDf = joinedDf
87        .withColumn("v",joinedDf("t") * joinedDf("m"))
88        .select("i","l", "v")
89
90      val finalDf = tempDf.withColumnRenamed("l","j")
91
92      finalDf.groupBy("i","j").agg(sum($"v"))
93        .withColumnRenamed("sum(v)","v")
94    }
95
96    def columnSelector (df: DataFrame, colNumber: String) : sql.DataFrame = {
97      df.select("i","v").filter(df("j").contains(colNumber))
98    }
99
100   def elemMultiply(df1 : DataFrame, df2 : DataFrame): sql.DataFrame  = {
101     val df1x = df1.withColumnRenamed("v","v1")
102     val df2x = df2.withColumnRenamed("v","v2")
103     val joinedDf = df1x.join(df2x, Seq("i"))
104     val productDf = joinedDf
105       .withColumn("v",joinedDf("v1") * joinedDf("v2"))
106       .select("i","v")
107     productDf
108   }
109
110   def cooMatrixTranspose (cooMat : DataFrame) : DataFrame = {
111     val cooMatTransposed = cooMat.select("j","i","v")
112       .withColumnRenamed("j","k")
113       .withColumnRenamed("i","l")
114     val cooMatTransposedRenamed = cooMatTransposed
115       .withColumnRenamed("k","i")
116       .withColumnRenamed("l","j")
117     cooMatTransposedRenamed
```

```
118   }
119
120   def ijvFileToDataFrame(path: String): spark.sql.DataFrame = {
121     val df = sparkSession.read.format("csv")
122       .option("header", "true")
123       .load(path)
124     df
125   }
126
127   def ivFileToDataFrame(path: String): spark.sql.DataFrame = {
128     val df = sparkSession.read.format("csv")
129       .option("header", "true")
130       .load(path)
131     df
132   }
133
134   def ivFileToArray(path: String): Array[String] = {
135     val df = sparkSession.read.format("csv")
136       .option("header", "true")
137       .load(path)
138     val newDF = df
139       .withColumn("i", df("i").cast("Int"))
140       .withColumn("v",df("v").cast("String"))
141     val oCount = newDF.count().toInt
142     val oArray = Array.ofDim[String](oCount)
143     for (k <- 0 to oCount - 1){
144       val dfArray = newDF.select("v").filter(newDF("i")
145         .contains(k)).take(1)
146       val dfRow = dfArray(0)
147       oArray(k) = dfRow.getString(0)
148     }
149     oArray
150   }
151
152 }
```

Anhang B

Python-Skripte

B.1 Skript für dünnbesetzte Matrixerstellung

```
1
2  """
3  Module for generating large Matrices with set no_of_elements, bandwidth
   and size.
4  """
5
6  import random
7  import sys
8  import argparse
9  import math
10
11 __author__ = 'Maximilian Lamster'
12 __version__ = '0.1'
13
14 def parse_arguments():
15   """
16   Parse arguments from command line.
17   :return: parsed arguments object.
18   """
19   parser = argparse.ArgumentParser(
20     description='Generate large Matrices for testing matrix-multiplication
     in distributed systems.',
21   epilog='If something is wrong send me a mail.')
22   parser.add_argument('--version', action='version', version=__version__)
23   parser.add_argument('n', type=int, help='number of elements per row')
24   parser.add_argument('s', type=int, help='size of the s x s Matrix')
```

```
25    return parser.parse_args()
26
27
28  def generate_matrix(n, s):
29    """
30    Generate a s x s Matrix with n elements per row and a bandwidth of
      n*3/40 around the diag elem.
31    """
32
33    dataname = (str(n) + ' ' + str(s))
34    print(dataname)
35
36    #csv creation block
37    outFile = open(('m1.csv'),'w', encoding='utf8')
38    header = 'i,j,v\n'
39    outFile.writelines(header)
40
41    max_no_row_el = n
42    bandwitdh = math.floor(3*s/40)
43    a = 0.0
44    b = 1.0
45
46    #generates the entrys for every row and writes em directly to the file
      one by one
47    for i in range(s):
48      __no_row_el_i = max_no_row_el
49
50      #generating diag element
51      __ith_row = (i, i, random.uniform(a, b))
52
53      #make a list containing every index
54      __col_i = list(range(max(0, i - bandwitdh), min(s, i + 1 + bandwitdh)))
55      #remove diag index i
56      __col_i.remove(i)
57      #take a sample of remaining indices with the previous generated number
58      __col_i = random.sample(__col_i, min(__no_row_el_i,len(__col_i)))
59
60      #write diag elem to file
61      outFile.write('%s,%s,%s' % __ith_row + '\n')
62
63      #write a matrix entry for every remaining index to file
```

```
64      for j in __col_i:
65        elem = (i, j, random.uniform(a, b))
66        outFile.writelines('%s,%s,%s' % elem + '\n')
67
68    outFile.close()
69    print(dataname + ' done...')
70
71  if __name__ == '__main__':
72    args = parse_arguments()
73    generate_matrix(args.n, args.s)
```

B.2 Skript für Einservektorerstellung

```
"""
Module for generating vectors with the given size s and with entrys in
format i,j,v.
"""

import random
import sys
import argparse

__author__ = 'Maximilian Lamster'
__version__ = '0.1'

def parse_arguments():
  """
  Parse arguments from command line.
  :return: parsed arguments object.
  """
  parser = argparse.ArgumentParser(
  description='Generate a vector with entrys in the format i,j,v.',
  epilog='If something is wrong send me a mail.')
  parser.add_argument('--version', action='version', version=__version__)
  parser.add_argument('s', type=int, help='size of the vector')
  return parser.parse_args()

def generate_vector(s):
  """
  Generate an s-sized vector in the format i,j,v.
  """

  dataname = (str(s))
  print(dataname)

  #csv creation block
  outFile = open(('m2.csv'),'w', encoding='utf8')
  header = 'i,j,v\n'
  outFile.writelines(header)

  #generates the entrys for every row and writes those in the file one by
  one
```

```
38    for i in range(s):
39      elem = (i, '0', '1')
40      outFile.writelines('%s,%s,%s' % elem + '\n')
41
42    outFile.close
43    print(dataname + ' done...')
44
45  if __name__ == '__main__':
46    args = parse_arguments()
47    generate_vector(args.s)
```

B.3 Skript für dichtbesetzte Matrizenerstellung

```
1  """
2  Module for generating large Matrices with size s.
3  """
4
5  import random
6  import sys
7  import argparse
8  import math
9
10 __author__ = 'Maximilian Lamster'
11 __version__ = '0.1'
12
13 def parse_arguments():
14   """
15   Parse arguments from command line.
16   :return: parsed arguments object.
17   """
18   parser = argparse.ArgumentParser(
19   description='Generate large Matrices for testing matrix-multiplication
     in distributed systems.',
20   epilog='If something is wrong send me a mail.')
21   parser.add_argument('--version', action='version', version=__version__)
22   parser.add_argument('s', type=int, help='size of the s x s Matrix')
23   return parser.parse_args()
24
25
26 def generate_matrix(s):
27   """
28   Generate 2 dense s x s Matrices m1.csv and m2.csv with.
29   """
30
31   dataname = ('dense ' + str(s))
32   print(dataname)
33
34   #csv creation block
35   outFile = open(('m1.csv'),'w', encoding='utf8')
36   header = 'i,j,v\n'
37   outFile.writelines(header)
38
```

```
39    a = 0.0
40    b = 1.0
41
42    #generates every entry and writes em to the file 1
43    for i in range(s):
44      for j in range(s):
45        elem = (i, j, random.uniform(a, b))
46        outFile.writelines('%s,%s,%s' % elem + '\n')
47
48    outFile.close()
49    print(dataname + ' 1 done...')
50
51    #csv creation block2
52    outFile2 = open(('m2.csv'),'w', encoding='utf8')
53    header = 'i,j,v\n'
54    outFile2.writelines(header)
55
56    a = 0.0
57    b = 1.0
58
59    #generates every entry and writes em to the file 2
60    for x in range(s):
61      for y in range(s):
62        elem2 = (x, y, random.uniform(a, b))
63        outFile2.writelines('%s,%s,%s' % elem2 + '\n')
64
65    outFile2.close()
66    print(dataname + ' 2 done...')
67
68
69
70  if __name__ == '__main__':
71    args = parse_arguments()
72    generate_matrix(args.s)
```

Anhang C

Messwerttabellen

C.1 dünnbesetzte Matrixmultiplikation via Map-Reduce

Zeit in ns	Dimension	Durchlauf
34374619189	100000	1
34074141210	100000	2
35517339819	100000	3
32719212913	100000	4
33489604492	100000	5
34067842874	100000	6
34499288422	100000	7
34231662407	100000	8
32709581176	100000	9
33178734986	100000	10
47651166575	500000	1
44825864799	500000	2
47331728790	500000	3
47778173451	500000	4
46424299367	500000	5
45991115079	500000	6
46077823707	500000	7
44632958050	500000	8
46009383319	500000	9
42956313321	500000	10
61614434961	1000000	1
63928126815	1000000	2
63402837639	1000000	3
62386019175	1000000	4
62549744427	1000000	5

Zeit in ns	Dimension	Durchlauf
61960880098	1000000	6
61010660776	1000000	7
62779665445	1000000	8
61203587120	1000000	9
63739443078	1000000	10
124003693064	2000000	1
124873917419	2000000	2
125075356050	2000000	3
126649691076	2000000	4
121657478321	2000000	5
120368383465	2000000	6
122396834448	2000000	7
123575132337	2000000	8
123947157114	2000000	9
124753344903	2000000	10
173555927844	3000000	1
165885878683	3000000	2
162819229243	3000000	3
169645644713	3000000	4
175007657786	3000000	5
167175405435	3000000	6
180896108270	3000000	7
165002354867	3000000	8
165883725406	3000000	9
169485142122	3000000	10

C.2 dünnbesetzte Matrixmultiplikation via SQL

Zeit in ns	Dimension	Durchlauf
33765206767	100000	1
31924126048	100000	2
34667324618	100000	3
32627385896	100000	4
32088292727	100000	5
31639636622	100000	6
34709446337	100000	7
31426949888	100000	8
33231425976	100000	9
34209114574	100000	10
37788129421	500000	1
39037362634	500000	2
38313535636	500000	3
40415487510	500000	4
35214512584	500000	5
37537748370	500000	6
38133998103	500000	7
37992300498	500000	8
36579976172	500000	9
37124954860	500000	10
39258556142	1000000	1
42809357914	1000000	2
37946354924	1000000	3
43160512641	1000000	4
43068590209	1000000	5

Zeit in ns	Dimension	Durchlauf
41884399142	1000000	6
41511485066	1000000	7
42391037529	1000000	8
43927349179	1000000	9
38998977406	1000000	10
106871406898	2000000	1
125586020178	2000000	2
124800615952	2000000	3
90309067752	2000000	4
107009331942	2000000	5
97434218057	2000000	6
115362784782	2000000	7
109239478922	2000000	8
103859002348	2000000	9
95805773460	2000000	10
161994366305	3000000	1
159791843289	3000000	2
115421500661	3000000	3
163521927907	3000000	4
158760915021	3000000	5
164247870479	3000000	6
118888702641	3000000	7
167769042987	3000000	8
117391821351	3000000	9
116037562777	3000000	10

C.3 dichtbesetzte Matrixmultiplikation via BlockMatrizen

Zeit in ns	Dimension	Durchlauf
34898471349	240	1
36209383773	240	2
30937203881	240	3
35761655454	240	4
31161428087	240	5
35287245454	240	6
27791143150	240	7
38838822313	240	8
34790273021	240	9
32393863419	240	10
40079510247	480	1
36914158519	480	2
39758098621	480	3
40659176862	480	4
39175792108	480	5
39481044092	480	6
39666255960	480	7
39312455787	480	8
35611794723	480	9
41066143516	480	10
39293946809	720	1
38132902062	720	2
36390411035	720	3
36630465497	720	4
39334688738	720	5
33800995319	720	6
39502708002	720	7
35397607182	720	8
38956784636	720	9
30963217407	720	10

Zeit in ns	Dimension	Durchlauf
37214263516	960	1
37682443649	960	2
34102139303	960	3
36597389210	960	4
37244086504	960	5
34231951487	960	6
37437136797	960	7
38092821223	960	8
36326657515	960	9
34337951255	960	10
41067165303	1200	1
40543246586	1200	2
40216831113	1200	3
39624531891	1200	4
40199636715	1200	5
39019440780	1200	6
39694972172	1200	7
36472992055	1200	8
40615744669	1200	9
39354855292	1200	10
43837733283	1440	1
44187783275	1440	2
41512793692	1440	3
41826304644	1440	4
42582278310	1440	5
43694731297	1440	6
42495810909	1440	7
44243052061	1440	8
43521462573	1440	9
45518707468	1440	10

Selbständigkeitserklärung

Ich erkläre, dass ich die vorliegende Arbeit selbständig und nur unter Vorlage der angegebenen Literatur und Hilfsmittel angefertigt habe.

Rostock, den