

Institut für Parallele und Verteilte Systeme

Abteilung Anwendersoftware

Universität Stuttgart
Universitätsstraße 38
D - 70569 Stuttgart

Masterarbeit

**Migration des Datenmanagements einer
datenintensiven Anwendung in ein
Big-Data-Framework**

Dominik Bäßler

Studiengang:	Informatik
Prüfer:	PD Dr. Holger Schwarz
Betreuer:	PD Dr. Holger Schwarz
begonnen am:	15.11.2016
beendet am:	15.05.2017
CR-Klassifikation:	H.4.2

KURZFASSUNG

Das Ziel der vorliegenden Arbeit ist es, die Migration des Datenmanagements einer datenintensiven Anwendungen auf Basis relationaler Daten anhand einer entworfenen generischen Vorgehensweise in ein Big-Data-Framework zu migrieren und die Ergebnisse zu evaluieren. Nach einer Ist-Analyse der zu migrierenden Anwendung wurde Spark als potentiell Framework für die Umsetzung durch ausgewählte gewichtete Kriterien ermittelt. Nach der Konzeption und problembehafteten Implementierung der Anwendung in Sparks Java API wurden die Vorgehensweise und die erzielten Ergebnisse evaluiert. Die resultierende Hinterfragung der Tauglichkeit von bestimmten Datenmanagements und deren Umsetzung in einem horizontal skalierten System spiegelt sich in der Vorgehensweise als Reflexion wider. Entwickler profitieren von der Möglichkeit im Vorfeld abschätzen zu können, ob der Versuch der Migration zielführend wäre. Es werden drei Indikatoren bereitgestellt, mit denen die Daten anhand von Abhängigkeiten untereinander und der Art der Verarbeitung beurteilt werden können. Die Probleme, die durch Ignorieren der Reflexion und ihres Ergebnisses entstehen können, werden in dieser Arbeit identifiziert und evaluiert.

ABSTRACT

This work provides a generic approach to migrate a data intensive application based on relational data to a big data framework. The generic approach is characterized by the exemplary implementation of the approach itself and the results of this implementation. During the course of this work challenges and problems of the migration of a typical data management are exposed. The underlying source of this data management is the data of multi-variant complex products from serial production. The interdependency of these data and their processing leads to several difficulties. Based on these characteristics the implementation in Spark undergoes performance and memory issues, because the data processing is not suitable for parallelization. These problems are identified and discussed, resulting in indicators for the preliminary decision making about the benefit of the migration. This step, called reflection, is integrated into the generic approach.

INHALTSVERZEICHNIS

1	Einleitung	1
2	Grundlagen	2
2.1	Big Data und Big-Data-Frameworks	2
2.2	Motivation zum Umstieg auf ein Big-Data-Framework	4
2.3	Stand der Technik	5
3	Generische Vorgehensweise zum Umstieg auf ein Big-Data-Framework	6
3.1	Ist-Analyse	6
3.2	Reflexion	8
3.3	Auswahl des Big-Data-Frameworks	9
3.4	Konzeption und Implementierung	9
3.5	Evaluierung	10
4	Kernproblem	11
4.1	Daten	12
4.2	Datenmanagement	13
4.3	Grenzen der traditionellen relationalen Lösung	16
5	Apache Spark	19
5.1	Architektur	19
5.2	Spark Core	21
5.2.1	RDD – Resilient Distributed Datasets	21
5.2.2	Speicher-Management	24
5.2.3	Fehlertoleranz	25
5.3	Spark SQL	26
5.3.1	DataFrame API	27
5.3.2	Catalyst	27
5.3.3	Evaluation	29
5.4	Erläuterung der Wahl von Spark	29
5.4.1	Kriterien	30
5.4.2	Evaluation	32
6	Konzept für den Prototyp	38
6.1	Daten und Workflow	38
6.1.1	Aufträge	38
6.1.2	Aktionen	39

6.1.3	Workflow	42
6.2	Grobentwurf	43
6.3	Feinentwurf	44
7	Implementierung des Konzepts in Spark	46
7.1	Driver	46
7.2	Datenextraktion	46
7.3	Datenverarbeitung	48
7.4	Datenausgabe	51
7.5	Deployment	52
8	Evaluierung	54
8.1	Evaluierung der Vorgehensweise	54
8.2	Evaluierung der implementierten Lösung	54
8.2.1	Testaufbau	55
8.2.2	Evaluierung der Datenextraktion	55
8.2.3	Evaluierung der Datenverarbeitung	58
8.2.4	Verbesserungsansätze	61
9	Zusammenfassung und Ausblick	62
I.	Abbildungsverzeichnis	I
II.	Tabellenverzeichnis	II
III.	Quellenverzeichnis	II

1 Einleitung

Die Welt der Unternehmen befindet sich seit mehr als einer Dekade im Wandel. Viele Unternehmen wollen dem Trend folgen und mit Bereichen wie Digitalisierung und Big Data das Sortiment ihres Angebotes erweitern. Dieser Trend resultiert aus den immer schneller wachsenden und immer größer werdenden Datenmengen, die durch die stetig wachsende Anzahl an Produzenten dieser erzeugt werden. Um die produzierten Datenmengen, die oft gar nicht oder nur semistrukturiert sind, zu nutzen brauchen die Unternehmen Technologien wie NOSQL-Datenbanken und Big-Data-Frameworks, die auf Grund ihrer horizontalen Skalierbarkeit mit den großen Datenmengen und Datentypen wie Streams und Graphen umgehen können. Durch die Verarbeitung der Daten sind die Unternehmen in der Lage Wissen zu extrahieren.

Durch diesen Trend werden vielen Unternehmen neue Möglichkeiten und Wege eröffnet den Profit zu steigern oder den Kunden zufrieden zu stellen. Doch das Tagesgeschäft der etablierten Unternehmen basiert immer noch auf relationalen Daten und den Technologien diese zu verarbeiten. Den Entwicklern von Technologien wie Big-Data-Frameworks ist diese Tatsache bewusst, weshalb in den letzten Jahren die Unterstützung der Verarbeitung von relationalen Daten weitestgehend nachgerüstet wurde. Fraglich ist jedoch inwiefern diese Technologien für die Verarbeitung der Unternehmensdaten genutzt werden können. Bei Betrachtung von typischen Unternehmensdaten wie Produktdaten wird ersichtlich, dass durch Individualisierung und immer komplexere Produkte auch hier ein stetiges Wachsen der Datenmenge zu beobachten ist. Unternehmen wollen diese Daten nutzen um Wissen zu generieren und so zum Beispiel Bedarfsprognosen für die Zukunft aufstellen zu können. Inwieweit Big-Data-Frameworks für diese Daten genutzt werden können soll in dieser Arbeit an der Primärbedarfsprognose der Automobilindustrie beispielhaft erörter werden.

Um dem Leser das Verständnis für Big Data und Frameworks näher zu bringen sind in Kapitel 2 die Grundlagen erklärt. In Kapitel 3 wird eine generische Vorgehensweise zur Migration des Datenmanagements datenintensiver Anwendungen auf ein Big-Data-Framework eingeführt. Die Vorgehensweise umfasst dabei notwendige Schritte und Überlegungen von der Ist-Analyse bis zur Evaluierung der neuen Anwendung. Kapitel 4 behandelt das Kernproblem bei der Erstellung von Primärbedarfsprognosen variantenreicher Produkte und zeigt Grenzen von traditionellen relationalen Systemen auf, die durch den Einsatz eines Big-Data-Frameworks beseitigt werden könnten. In Kapitel 5 wird das Big-Data-Framework Spark inklusive des Moduls SparkSQL vorgestellt, welches in dieser Arbeit bei der Umsetzung eines Prototypen Verwendung findet. Zusätzlich behandelt das letzte Unterkapitel die Frage, warum genau Spark verwendet wird und begründet die Wahl anhand von ausgesuchten Kriterien. Das Konzept für den Prototyp und die Implementierung in Spark werden in Kapitel 6 und 7 vorgestellt. Nach einer Evaluierung der in dieser Arbeit angewendeten Vorgehensweise aus Kapitel 3 und der Ergebnisse des Prototyps und seiner Vor- und Nachteile schließt die Arbeit in Kapitel 9 mit einer Zusammenfassung und einem Ausblick auf zukünftige Arbeit ab.

2 Grundlagen

Im diesem Kapitel sollen neben den Grundlagen zu Big Data und Big-Data-Frameworks auch die Motivation zum Umstieg auf ein Big-Data-Framework verdeutlicht und vergleichbare Arbeiten vorgestellt werden.

2.1 Big Data und Big-Data-Frameworks

In der heutigen Zeit werden Sachverhalte gerne mit Modewörtern erklärt, deren Bedeutung vielen nicht genau klar ist. Auch Big Data gehört zu diesen Modewörtern und verleitet oft zu der Annahme, dass es sich einfach um große Datenmengen handelt. Die Bedeutung von Big Data geht jedoch weit über die reine Definition der Daten durch die Datenmenge hinaus.

„The term Big Data describes a data environment in which scalable architectures support the requirements of analytical and other applications which process, with high velocity, high volume data which may have a variety of data formats and which may include high velocity data acquisition.” [1]

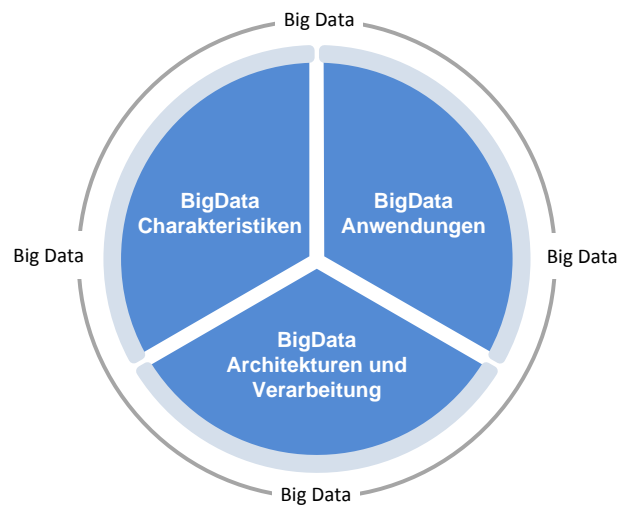


Abbildung 1 - Ansatz zur Definition von Big Data (nach [1])

Emmanuel und Stanier diskutieren die Menge an unterschiedlichen Definitionen für Big Data [1].

Sie betrachten dabei verschiedene Perspektiven, durch die die Definitionen entstanden. Aus einer vergleichenden Sicht wird Big Data im Vergleich zu bestehenden Technologien betrachtet, wodurch die Möglichkeiten durch Big Data aufgezeigt werden [2][3]. Die Definitionen von Big Data über die Daten trifft Aussagen über die Eigenschaften, über die Daten verfügen müssen, um unter den Begriff Big Data zu fallen [4][6]. Die dritte Perspektive bezieht sich auf die Umgebung, bei deren Definitionen über die Betrachtung der Eigenschaften hinausgeblickt wird und die Verarbeitungs- und Ausführungsumgebung, sowie die Anwendungen mit einbezogen werden [7].

Aus diesen Perspektiven erstellten Emmanuel und Stanier die hier dargestellte umfassende Definition. Abbildung 1 zeigt, dass die Definition neben den Charakteristiken der Daten sowohl Anwendungen als auch Architektur und Verarbeitung der Daten mit einbezieht.

Die typischen Eigenschaften der Daten von Big Data werden die drei V's genannt, die von Gartner [4] zur Erklärung der Auswirkungen von neu aufkommenden Datenmanagementstrategien aufgestellt wurden [1]:

- *Volume* beschreibt die Anforderungen an Speicher und Rechenleistung für die Verarbeitung einer großen Menge an Daten.
- *Velocity* ist die Anforderung an Big-Data-Systeme zur Unterstützung von Daten, die in hoher Geschwindigkeit generiert und verarbeitet werden.
- *Variety* stellt die Vielfalt der Datenformate im Big-Data-Umfeld dar und fordert die Unterstützung dieser Vielfalt von den Systemen.

Bereits hier wird ersichtlich, dass die Teilbereiche nicht voneinander unabhängig sind, denn die Charakteristik der Daten stellt Anforderungen an die Architektur und die Verarbeitung der Daten. Für die Unterstützung der Verarbeitung von Big Data bedarf es einer skalierbaren Architektur, welche die Anforderungen durch die Datencharakteristiken erfüllt. Zu guter Letzt beeinflusst das Warum, also die Anwendungen von Big Data, ebenso wie die Charakteristiken der Daten, die Architektur und die Verarbeitung. [1]

Auf der Grundlage der hier gegebenen Definition und Beschreibung von Big Data kommt die Frage auf, wie ein Big-Data-Framework beschrieben werden kann. Tekiner und Keane identifizieren dieses als die Basis zur Entwicklung und dem Management von Big Data Anwendungen, das die folgenden drei Phasen unterstützen muss [8]:

- Phase 1 umschreibt die Erfassung und das Anpassen der Daten. Bei der Anpassung müssen die Daten integriert und mittels Transformationen die Bedeutung der Daten hervorgehoben werden.
- Phase 2 umfasst die Verarbeitung der in Phase 1 vorbereiteten Daten, wodurch neu gewonnene Informationen entstehen.
- Diese Informationen werden in Phase 3 aufbereitet indem die Daten organisiert und interpretiert werden. Durch die letzte Phase entsteht der „Gewinn“ für das Unternehmen.

Neben der Unterstützung dieser Phasen muss ein Big-Data-Framework noch weitere Merkmale aufweisen, um Daten mit den Eigenschaften von Big Data effizient verarbeiten zu können. Der Bezug der Datencharakteristika zur Verarbeitungsumgebung ist die Grundlage der Umgebungsperspektive bei der Definition von Big Data. Durch *horizontale Skalierung* wird ein Parallelisieren der Verarbeitung der Datenmenge erreicht. So wird gewährleistet, dass ausreichend Speicher und Rechenleistung zur effizienten Verarbeitung der Daten mit der Eigenschaft *Volume* bereitgestellt werden kann. Dies wird oft durch die Anwendung in der Cloud und somit dem Pay-as-you-go Prinzip folgend erreicht.

Die Beschreibung ist sehr abstrakt gefasst, da Big Data ein sich änderndes Umfeld darstellt und nicht, wie zur Zeit der Prägung des Begriffs von Big Data noch üblich, auf ein Konzept wie MapReduce heruntergebrochen werden kann.

2.2 Motivation zum Umstieg auf ein Big-Data-Framework

Die Erläuterungen im vorherigen Abschnitt werfen weitergehend die Frage auf, warum sich der Umstieg auf ein Big-Data-Framework lohnt. Im folgenden Abschnitt soll diese Frage näher erläutert werden und damit auch die Motivation des Umstiegs auf ein solches im konkreten Fall dieser Arbeit dargestellt werden.

Ökonomischer Aspekt

Der Umstieg auf ein Big-Data-Framework kann für ein Unternehmen sowohl die Einsparung von finanziellen Mitteln, als auch die schonende Verwendung von Ressourcen bedeuten. Die Frameworks sind auf eine Ausführung auf handelsüblicher Hardware spezialisiert. Das dadurch ermöglichte Outsourcen der Verarbeitung der Daten in die Cloud führt zu einer Flexibilität, die mit On-premise-Hardware, einem eigenen Datenzentrum, nicht erreicht werden kann. Durch das sogenannte Pay-as-you-go-Konzept der Cloud-Anbieter kann die benötigte Hardware exakt für den gebrauchten Zeitraum allokiert werden. Zudem bietet die Cloud eine dynamische Skalierbarkeit, die auch Spitzen in der Daten- und Verarbeitungslast ohne Probleme abfangen kann. Je nach der im Unternehmen umgesetzten Lösung können somit auch finanzielle Mittel eingespart werden, wobei in Diskussionen über den Kostengewinn durch die Cloud oft betont wird, dass die Cloud nicht immer günstiger ist [9][10].

Heterogene Datenformate und Kombination verschiedener Anwendungen

In der Ära der Informationen liegen die Daten, welche verarbeitet werden sollen, nicht mehr nur in strukturierter oder relationaler Form vor, vielmehr sind semi- und unstrukturierte Datenformate in der Datenverarbeitung heute etabliert. Für alle Datenformate gibt es Tools und Software, welche eine performante Verarbeitung des entsprechenden Datenformats ermöglichen. Die Wertgewinnung durch Wissensextraktion aus Daten basiert jedoch auf der Kombination der Daten aus verschiedensten Datenquellen und daher rührenden heterogenen Datenformaten. Big-Data-Frameworks sollen deshalb den Zugriff auf unterschiedlichste Datenformate unterstützen und eine integrierte Verarbeitung der gesamten Datenmenge ermöglichen, welche aus graphischen Daten, Streams, strukturierten Daten und anderen Daten bestehen kann.

Datenmenge und Performance

Die traditionell eingesetzten Systeme zur „Datenanalyse“ sind zwar meist vertikal skalierbar, können dabei aber durch unterschiedliche Engpässe eingeschränkt werden. Dies gilt im Speziellen, wenn mehrere Anwendungen auf derselben Recheneinheit laufen. Diese Engpässe entstehen zum Beispiel durch unzureichende Hardware, Beschränkungen beim Ressourcenzugriff und den Overhead durch die Kontrolle der Nebenläufigkeit. Dies kann mit dem Umstieg auf ein Big-Data-Framework und damit einhergehender horizontaler Skalierung umgangen werden, wobei auch diese nicht ohne Engpässe existiert. Big-Data-Frameworks sind durch die Kommunikationsbandbreite zwischen den einzelnen Recheneinheiten limitiert, welche allerdings durch die entsprechende Hardware, die geographische Nähe der Recheneinheit oder eine Netzauslastungsschonende Ausführung beeinflusst werden kann.

2.3 Stand der Technik

Mit dem Bekanntwerden von Hadoop und Big Data wurde viel Arbeit von Wissenschaftlern geleistet, um das Gebiet, die Möglichkeiten und die Grenzen der Big-Data-Welt zu erforschen. Die meisten Arbeiten richten sich dabei auf die neuen Möglichkeiten, die durch Frameworks wie Hadoop und Spark entstehen. Darunter finden sich viele Arbeiten, die Möglichkeiten aufzeigen mit den Frameworks neues Wissen aus bestehenden Daten zu generieren oder bestehende Lösungen zur Wissensextraktion zu erweitern und zu verbessern. Auf dem Gebiet des Umstiegs vom Datenmanagement relationaler Daten mit SQL auf die Verwendung eines Big-Data-Frameworks wurden bei den Nachforschungen im Laufe dieser Arbeit jedoch keine vergleichbaren Ansätze gefunden.

Es existieren jedoch Arbeiten, die einen Teilbereich dieser Arbeit bereits behandeln oder von einem anderen Standpunkt aus betrachten.

Kobielus und Marcus [5] zum Beispiel stellen in ihrer Arbeit eine Roadmap für das Deployment von Big-Data-Anwendungen auf. Sie gehen dabei zwar nicht auf die Entwicklung der Anwendung ein, weisen jedoch auf wichtige Schritte hin, die vor und nach der Entwicklung notwendig sind.

Einige der Arbeiten beschäftigen sich mit der Migration von Daten in den Big-Data-Bereich, also in NoSQL, NewSQL und verteilten Datenspeicher. Der Fokus liegt hierbei jedoch auf der Datenmigration, die in dieser Arbeit nicht erforderlich ist, und nicht auf dem Datenmanagement einer datenintensiven Anwendung. [12][14]

Stonebraker und Cattell [15] stellen 10 Regeln für skalierbare Performance in Datenspeichern für einfache Operationen, wenige Lese- und/oder Schreibeoperationen, vor. Dabei gehen sie jedoch eher auf generell zu beachtende Themen wie zum Beispiel eine Shared-nothing Skalierbarkeit ein. Diese

Cannataro et al. [16] geben allgemein einen Überblick über Probleme der parallelen Verarbeitung großer Datenmengen. Weitere Arbeiten [17][18] vergleichen Ansätze zur Analyse sehr großer Datensätze und betrachten dabei Hadoop, sowie Systeme zur verteilten Verarbeitung von Daten und parallele Datenbankmanagementsysteme (DBMS). Dabei werden die Ansätze jeweils evaluiert und die Performanceunterschiede diskutiert. In dieser Arbeit ist der Vergleich von Frameworks zur verteilten Verarbeitung von Daten jedoch nur ein kleiner Teil der Vorgehensweise, die den kompletten Umstieg auf ein Big-Data-Framework betrachtet. Auch Liu [19] vergleicht in seiner Arbeit verschiedene Ansätze zur Datenverarbeitung aus dem Big-Data-Bereich anhand diverser Big-Data-Benchmarks.

Obwohl bereits viel Arbeit auf dem Gebiet Big Data geleistet wurde, konnte keine vergleichbare Arbeit gefunden werden, die einen Ansatz zur Migration des Datenmanagements einer datenintensiven Anwendung in ein Big-Data-Framework aufstellt, exemplarisch ausführt und evaluiert.

3 Generische Vorgehensweise zum Umstieg auf ein Big-Data-Framework

In dieser Arbeit soll neben der Migration des Datenmanagements einer datenintensiven Anwendung in ein Big-Data-Framework eine generische Vorgehensweise zum Umstieg auf Big-Data-Frameworks vorgestellt werden. Dabei wird von einer strukturierten Datenbasis als Ausgangspunkt des Umstiegs ausgegangen.

Die Vorgehensweise zur Migration des Datenmanagements auf ein Big-Data-Framework umfasst folgende Schritte:

1. Ist-Analyse
2. Reflexion
3. Auswahl des Frameworks
4. Konzeption und Implementierung der neuen Software
5. Evaluierung
6. Planung des Umstiegs (parallele Entwicklung, Downtime, etc.)

Die Schritte umfassen neben einer ausführlichen Analyse der Ist-Situation eine Reflexion der Situation und des Sinns eines Umstiegs, die eigentliche Umsetzung mit Konzeption, Implementierung und Evaluierung, und die Planung des Umstiegs. Die einzelnen Schritte werden in den folgenden Abschnitten erklärt.

3.1 Ist-Analyse

Der erste Schritt in der Vorgehensweise beim Umstieg auf ein Big-Data-Framework ist die Analyse des Ist-Zustandes. Bei der Ist-Analyse müssen zwei Teilbereiche betrachtet werden.

- Daten und Workflows
- Technische und betriebswirtschaftliche Anforderungen

Daten und Workflows

Die wichtigste Grundlage für die Migration des Datenmanagements einer datenintensiven Anwendung ist das Verständnis der Daten und wie diese verarbeitet werden sollen. Big-Data-Frameworks basieren auf horizontaler Skalierung, also neben der verteilten Verarbeitung auch auf der verteilten Speicherung, der Partitionierung, der Daten. Aus diesem Grund muss neben dem Schema insbesondere die Bedeutung der Daten in die Überlegungen mit einbezogen werden.

Die verteilte Verarbeitung von Daten erreicht eine höhere Performance bei der Arbeit mit den Daten, wenn die Datenmenge partitioniert ist. Mit Hilfe der richtigen Partitionierung kann also dafür gesorgt werden, dass Anfragen an die Daten nur auf wenigen oder auch nur einer einzigen Partition ausgeführt werden müssen.

Personen		
Name	Vorname	Alter
Albert	Melanie	18
Arjim	Ali	22
Austermann	Karl	100
Bauer	Annika	15
Braun	Simon	98
Manni	Manfred	19
Mozart	Hannes	47
Mueller	Mark	29

Personen1 (Alter < 20)		
Name	Vorname	Alter
Bauer	Annika	15
Albert	Melanie	18
Manni	Manfred	19

Personen1 (20 > Alter < 30)		
Name	Vorname	Alter
Arjim	Ali	22
Mueller	Mark	29

Personen2 (Alter > 30)		
Name	Vorname	Alter
Mozart	Hannes	47
Braun	Simon	98
Austermann	Karl	100

Abbildung 2 - Beispiel für die Partitionierung einer Tabelle (Spalte Alter)

Bei Betrachtung der in Abbildung 2 dargestellten Partitionierung und einer Suche nach allen Personen mit

```
SELECT *
FROM Personen
WHERE Alter BETWEEN 20 AND 30
```

muss dementsprechend nicht die komplette Tabelle *Personen* durchsucht werden, sondern lediglich die Partitionen, die alle Einträge für Personen mit Alter 20 bis 30 beinhalten. Zudem können auch nicht-lesende Anfragen durch die Partitionierung der Daten parallelisiert werden. So kann gleichzeitig auf allen Partitionen gearbeitet werden.

Zur Wahl eines passenden Partitionierungsschemas ist die Auseinandersetzung mit der Frage wie auf die Daten zugegriffen wird, also der Aufgaben, die auf den Daten ausgeführt werden sollen, notwendig. Erst mit dem Wissen, welche Aktionen auf den Daten ausgeführt werden, kann bestimmt werden, wie eine Datenmenge partitioniert wird. Die in Abbildung 2 dargestellte Partitionierung erzielt bei der Suche nach Personen, deren Nachname mit dem Buchstaben A anfangen keine besondere Wirkung, da trotzdem alle Partitionen durchsucht werden müssen. Dementsprechend ist bei der Partitionierung von Daten darauf zu achten welche Spalten als qualifizierende Merkmale der Anfragen verwendet werden.

Ein besonderer Fall existiert dabei, wenn auch die auf den Daten auszuführenden Aktionen, also die Verarbeitung der Daten selbst, als Daten vorliegen. Ist dies der Fall, muss hier ebenfalls überlegt werden, inwiefern diese Daten partitioniert werden können. Darüber hinaus muss allerdings auch darauf geachtet werden, dass die Aktionsdaten bestenfalls im gleichen Speicher liegen wie die Partition der Daten, auf welche die Aktionen dieser Partition zugreifen.

Technische und betriebswirtschaftliche Anforderungen

Bei der Überlegung die Datenverarbeitung einer Anwendung in ein Big-Data-Framework zu migrieren muss auch auf technische und betriebswirtschaftliche Anforderungen eingegangen werden. Einige Beispiele dazu sind:

- Finanzielle Ressourcen
- Technikvorgaben und Unternehmenslizenzen
- Standortvorgaben wie z.B. on-premise

Durch diese nicht fachlichen Vorgaben kann der Umstieg bereits erschwert und sogar behindert werden. Da die bekannten und weit verbreiteten Big-Data-Frameworks jedoch unter Apache-Lizenz stehen, oder auf den Apache-Entwicklungen basieren, und in der Cloud laufen, ist hier nicht mit großen Hindernissen zu rechnen, solange die entsprechenden finanziellen Ressourcen vom Unternehmen bereitgestellt werden.

3.2 Reflexion

An dieser Stelle ist es sinnvoll, die Ist-Analyse noch einmal durchzugehen und zu überlegen, ob es wirklich notwendig ist auf ein Big-Data-Framework umzusteigen. Den Umstieg durchzuführen nur um dem Trend zu folgen und auch „Big Data zu machen“, obwohl die Auseinandersetzung mit vorhandenen Lösungen und Technologien ausreichend wäre, ist selten sinnvoll. Indikatoren für bzw. gegen einen Umstieg sind:

- Die sequentielle Verarbeitung der Daten wie zum Beispiel die Aggregation eines Wertes lässt sich gut in einem Big-Data-Framework umsetzen.
- Besteht die Aufgabe der Verarbeitung der Daten aus mehreren Schritten und die Schritte bauen aufeinander auf, so ist dies ein Indikator dafür, dass sich der Umstieg nicht lohnen könnte, da sich nur die Zwischenschritte parallelisieren lassen.
- Auch eine hohe Abhängigkeit der Daten untereinander spricht gegen die Umsetzung der Verarbeitung in einem Big-Data-Framework, da dabei die Parallelisierung von den Datenzugriffen über das Netzwerk beeinträchtigt wäre.

Die seltenen Fälle in denen es sinnvoll ist den Umstieg entgegen der Anzeichen umzusetzen beruhen auf ein in die Zukunft gerichtetes Denken. Beispiele für diese Fälle sind der Aufbau von Kompetenz im Big-Data-Bereich oder das Öffnen einer Anwendung für Erweiterungen im Big-Data-Bereich.

Hat man sich nun in der Ist-Analyse mit den Workflows, den Daten und den nicht fachlichen Vorgaben auseinandergesetzt und ist nach der Reflexion zu dem Schluss gekommen, dass der Umstieg tatsächlich durchgeführt werden soll, folgt als nächster Schritt die Wahl des passenden Big-Data-Frameworks.

3.3 Auswahl des Big-Data-Frameworks

Die Auswahl des passenden Frameworks ist abhängig von der umzusetzenden Anwendung, also den fachlichen, technischen und betriebswirtschaftlichen Vorgaben. Diese werden in entsprechenden Auswahlkriterien widergespiegelt und können nicht verallgemeinert werden. Es existieren jedoch einige Kriterien, die in den meisten Unternehmen von Bedeutung sind:

- Lizenzkosten und benötigte Hardware
- Verfügbare/unterstützte Datenformate
- Recovery und Fehlerverhalten
- Performance

Durch eine Kombination aus allgemein gültigen und auf Vorgaben abgestimmte Kriterien, wie zum Beispiel der Unterstützung von relationalen Daten und SQL-Standards, kann das optimale Framework für den Umstieg gefunden werden.

Zur präziseren Bewertung der Frameworks ist es empfehlenswert die Kriterien zusätzlich mit Hilfe eines paarweisen Vergleichs [13] zu gewichten.

	K1	K2	K3	Summe	Wert
Kriterium 1	X	0	2	2	0,50
Kriterium 2	2	X	2	4	1,00
Kriterium 3	0	0	X	0	0,00
				4	1,00

BEWERTUNG	0	weniger wichtig
	1	gleichwertig
	2	wichtiger

Abbildung 3 - Beispiel paarweiser Vergleich von Kriterien

Dabei werden jeweils zwei Kriterien als Zeile und Spalte einer Tabelle gegenübergestellt und festgelegt, welches der Kriterien wichtiger ist. Als Bewertung wird dabei die in Abbildung 3 dargestellte Methode verwendet. An diesem Beispiel wird ersichtlich, dass die Reihenfolge der Gewichtung $K3 < K1 < K2$ entsteht und den Kriterien dementsprechende Modifikatoren zwischen 0 und 1 zugewiesen werden, der sich aus der Summe der Punkte dividiert durch den Maximalwert ergibt.

Die so ermittelten Modifikatoren werden auf die Bewertung der Frameworks anhand der entsprechenden Kriterien angewendet.

3.4 Konzeption und Implementierung

Nachdem ein Framework gefunden ist, folgt die Planung und Umsetzung für die Anwendung. Abhängig vom gewählten Framework treten dabei Änderungen gegenüber dem alten Konzept auf, wodurch leichte Änderungen an der Implementierung der Geschäftslogik bedingt werden können. Generell gibt es dabei drei Möglichkeiten:

- Die Software kann direkt portiert werden
- Die Software muss angepasst werden
- Die Software muss von Grund auf neu implementiert werden

Zum Beispiel muss beim Umstieg auf Spark darauf geachtet werden, dass die Daten im Vorfeld in den Arbeitsspeicher geladen werden und dort als unveränderbare Datensets vorliegen. Bei der Konzeption und der Umsetzung der neuen Software müssen dementsprechende Anpassungen bedacht werden.

3.5 Evaluierung

Sobald die Anwendung in dem gewählten Framework umgesetzt ist, sollte der geleistete Aufwand und dessen Erfolg gegen das Originalsystem gestellt werden. Dazu wird zum Beispiel die Performance der neuen Software evaluiert und mit der Performance der alten Software verglichen, wobei bedacht werden sollte, dass Unterschiede im Optimierungsgrad der Beiden bestehen können. Neben einer potentiellen Steigerung der Performance und finanziellen Aspekten kann der Nutzen auch in nicht messbaren Bereichen wie der Ermöglichung von Erweiterungen der Software im Big-Data-Bereich und dem Aufbau von Big-Data-Kompetenz zu finden sein.

Abhängig von solchen Erwägungen kann der Entschluss zum Umstieg auf die neue Software und damit das Datenmanagement mit einem Big-Data-Framework gefasst werden und die Umstellung auf die neue Software erfolgen.

4 Kernproblem

Bei der kundenauftragsorientierten Fertigung von variantenreichen Produkten stellt die präzise Planung durch Primärbedarfsprognose und Materialbedarfsplanung einen wichtigen Einflussfaktor auf die Profitabilität von Unternehmen dar. Dies ergibt sich daraus, dass beispielsweise in der Autoindustrie 60-70% der Kosten auf die Beschaffung von Komponenten und Teilen zur Herstellung der Autos entfällt [21]. Dabei entstehen durch die kundenauftragsbezogenen Serienproduktion und die Variantenvielfalt Produktvarianten mit bis zu 10000 Komponenten, die von mehreren hundert Lieferanten bezogen werden [20]. Die möglichen Komponenten und Varianten sind in der Produktdokumentation niedergeschrieben, die jedoch häufig fortlaufend geändert wird. Etwa 100.000 Änderungen pro Jahr machen eine einmalige Prognose unmöglich, weshalb diese permanent wiederholt und entsprechend angepasst werden muss [20].

Die Lieferanten fremdbezogener Komponenten benötigen Vorlaufzeit für die Herstellung, um die Komponenten zum Fertigungstermin bereitstellen zu können. Wie in Abbildung 4 ersichtlich liegen die von Kunden spezifizierten Aufträge meist jedoch nur bis 12 Wochen im Voraus vor, daher müssen für den weiteren Bedarf Prognosen erstellt werden, um kurze Lieferzeiten und Liefertermintreue gewährleisten zu können.

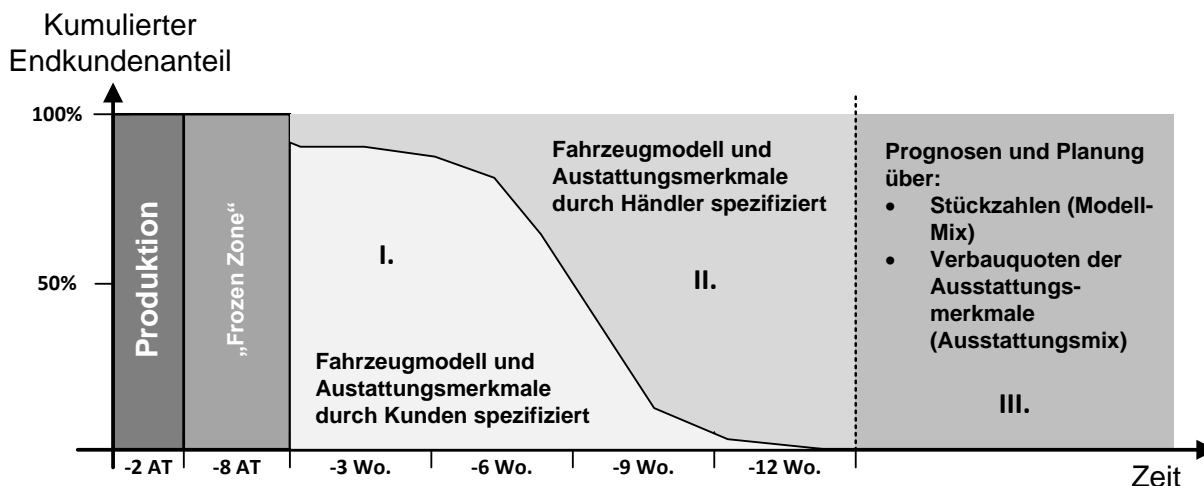


Abbildung 4 - Bezug der Aufträge während der Prognose (nach [20])

Bei der Bedarfsprognose wird zwischen der Primärbedarfsprognose und dem Sekundärbedarfsplan unterschieden. Die Primärbedarfsprognose soll den Bedarf in drei verschiedenen Dimensionen klären. Die volumenseitige Dimension gibt die absetzbaren Fahrzeugstückzahlen an, die ausstattungsseitige Dimension gibt Auskunft über die Quoten der Verbauung von Ausstattungsmerkmalen und zuletzt gibt der zeitliche Bezug an, zu welchem Zeitpunkt mit dem Bedarf zu rechnen ist. Aus dieser Prognose wird dann der Sekundärbedarfsplan aufgestellt, welcher den Zeitpunkt, die zu produzierenden Mengen und den Bereitstellungstermin der einzelnen fremdbezogenen Komponenten angibt. [20]

Zur Erstellung der Primärbedarfsprognose, auf deren Datenmanagement in dieser Arbeit der Fokus liegt, bedarf es einiger eingehender Informationen. Als Grundlage werden unter ande-

rem Informationen wie die Entwicklungserwartung der Märkte, unternehmerische Entscheidungen und Angaben zur Unternehmensleistung benötigt. Da in dieser Arbeit das Hauptaugenmerk allerdings auf dem Datenmanagement bei der Prognostizierung liegt, werden die vertraulichen betriebswirtschaftlichen Hintergründe hier nicht weiter ausgeführt. [20]

4.1 Daten

Um das Datenmanagement in Systemen für Primärbedarfsprognosen nachvollziehen zu können, ist eine genauere Betrachtung der Darstellung von komplexen und variantenreichen Serienprodukten erforderlich. Die laut Stäblein in der Automobilindustrie vorherrschende code-regelbasierte Produktdokumentation, beschreibt die Produktvarianten und deren Zusammenbaustruktur und besteht aus der **Baureihe**, den **Codes** und dem **Teil** [20].

Eine **Baureihe** stellt dabei einen Überbegriff zur Gliederung der Gesamterzeugnismenge dar und fasst verschiedene Typklassen, auch Modelle genannt, zusammen, welche wiederum in verschiedene Baumuster unterteilt werden. Die **Codes** beschreiben die Merkmale und Ausstattungen eines Produktes, wobei zwischen zwei Arten von Codes, definiert durch codeArt in Abbildung 5, unterschieden wird. Kundencodes stellen Auswahlmöglichkeiten und Konfigurationsoptionen für den Kunden dar, während Steuerungs-codes hingegen vertriebsseitige Merkmale wie länderspezifische oder Ausstattungslinien betreffende Merkmale und produktionsseitige Merkmale wie gesetzliche Vorschriften oder Werksbesonderheiten abbilden. Für die Kombination von Codes bestehen sogenannte Zusteuerungs- und Baubarkeitsregeln, die gesetzliche, technische und kaufmännische Restriktionen darstellen. Die verbaubaren **Teile** sind in sogenannten Positionsvarianten-Stücklisten enthalten, welche für eine Position alle möglichen Varianten an Teilen enthält. Dabei gibt es für diese Varianten Einbauregeln, die beschreiben unter welchen Voraussetzungen ein Teil eingebaut werden kann. Die kurze Coderegeln (KCR) beschreibt direkt die Code-Konstellation des zu verbauenden Teils und schließt die Positionsvarianten damit implizit aus. Die lange Coderegeln (LCR) hingegen schließt diese explizit aus, indem alle negierten KCR der Positionsvarianten angehängt werden. [20]

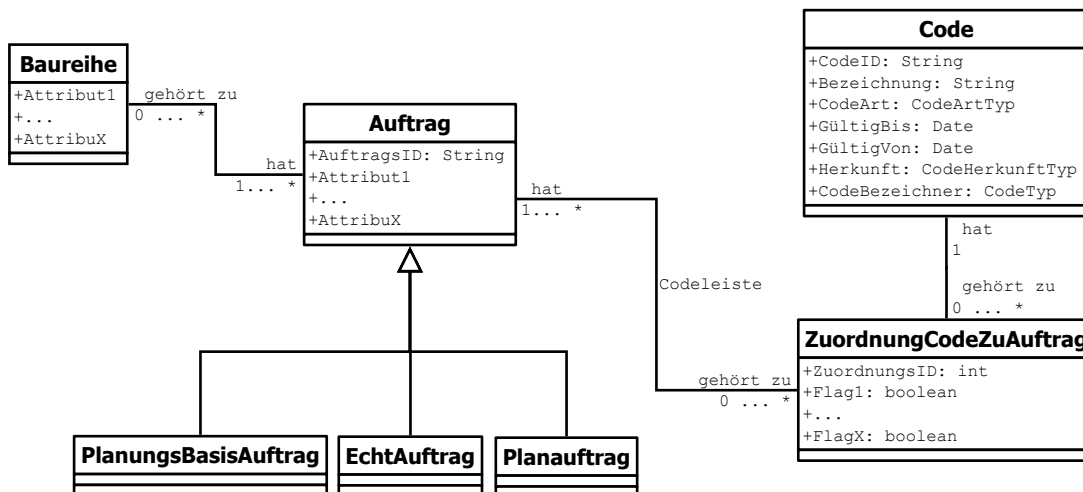


Abbildung 5 - abstraktes Datenmodell zur Darstellung von Aufträgen variantenreicher Produkte

In Abbildung 5 ein abstrahiertes Datenmodell der Auftragsdaten komplexer hochvarianter Produkte für die Erstellung der Prognose abgebildet.

Bei den Auftragsdaten werden die verbauten Komponenten durch Codes dargestellt. Dabei kann die Code-Tabelle bis zu 25.000 Einträge enthalten, was bei größeren Mengen von Aufträgen zu dementsprechend vielen Einträgen in der Zuordnungstabelle führt [20]. Über die Baureihen werden die Aufträge weiter spezifiziert.

4.2 Datenmanagement

Für die Primärbedarfsprognose (nachfolgend vereinfacht als Prognose bezeichnet) existieren unterschiedliche Verfahren, welche die Prognostizierung auf verschiedene Weise umsetzen [20]. Diese nachfolgende Beschreibung basiert auf dem Prognose-System eines großen Automobilherstellers und stellt eine auftragsbasierte Prognose dar. Diese basiert auf den Echtaufträgen aus der Vergangenheit, Kundenspezifizierte und geordnete Produkten. Mittels dieser Echtaufträge wird eine sogenannte Planungsbasis erstellt, die durch das Aussortieren ungeeigneter Datensätze unter Anwendungen von Restriktionen wie Länderbeschränkungen gebildet wird.

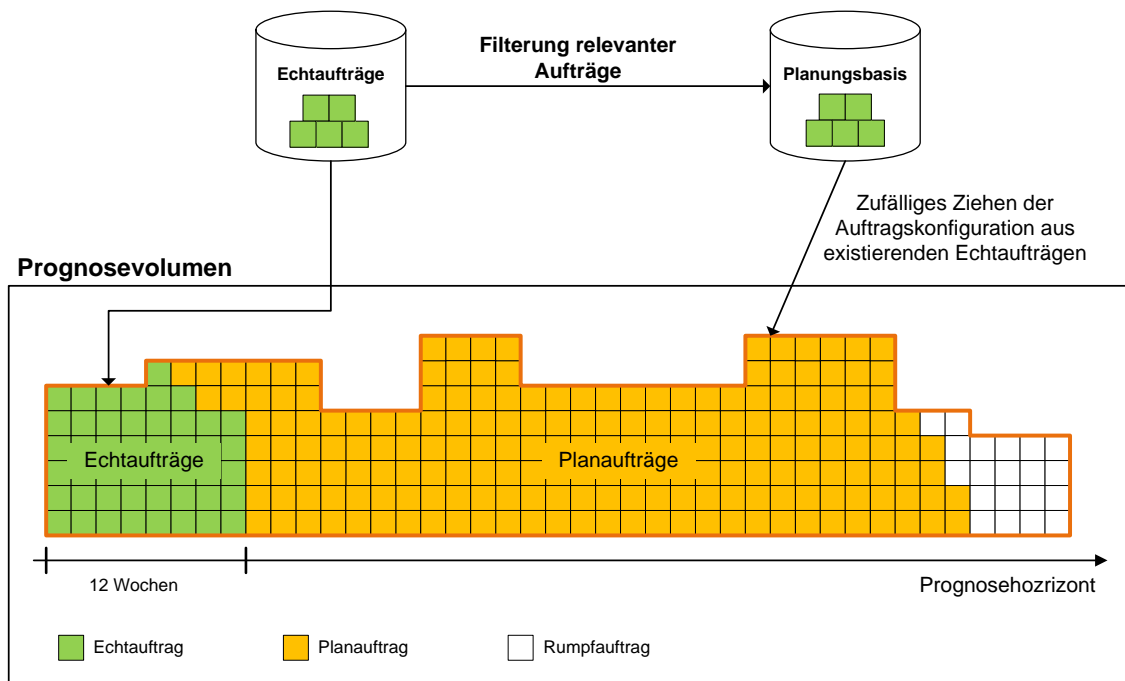


Abbildung 6 - Erstellung des Auftragsvolumens für die Prognose

Abbildung 6 veranschaulicht dabei, dass auf Grundlage dieser Planungsbasis anschließend die erwarteten Aufträge oder auch Planaufträge prognostiziert werden. Hierzu wird eine durch die Produktionsplanung vorgegebene Menge an Planaufträgen als Skelett, sogenannte Rumpfaufträge, angelegt und ihre Konfiguration durch die von Echtaufträgen der Vergangenheit ersetzt. Diese Ersetzung erfolgt dabei mittels zufälligen Ziehens ohne Zurücklegen. Sollten dafür nicht

genügend Echtaufträge vorhanden sein, so wird wieder auf die Gesamtmenge der Echtaufträge zurückgegriffen. Zusätzlich wird eine Validierung der Baubarkeit und die Beschränkung der Werksvolumen mittels Coderestriktionen für die erzeugte Auftragsmenge durchgeführt, wie bei der coderegelbasierten Produktdokumentation in Abschnitt 4.1 bereits beschrieben. Auf diese Beschränkung des Werksvolumens wird in dieser Arbeit jedoch nicht weiter eingegangen.

Durch das in Abbildung 6 dargestellte Vorgehen entsteht eine repräsentative Auftragsmenge für die Zukunft deren Prognosequalität jedoch nicht besonders hoch ist, da Aspekte, die nicht von der Vergangenheit abgeleitet werden können, unberücksichtigt bleiben. Hierfür existiert ein nachgelagerter Schritt, um die Auftragsmenge anzupassen und so Aspekte wie die Bedarfsaussicht der Märkte, saisonale Schwankungen oder an- bzw. auslaufende Typklassen berücksichtigen zu können.

Diese Anpassungen erfolgen mithilfe sogenannter Aktionen, die in Alternativen genannte Sets gruppiert sind und eine Zielverteilung für bestimmte Aufträge auf Codebasis enthalten. Die Aufträge werden dazu mit Hilfe von Qualifikationsmerkmalen wie einem Zeitraum oder einer bestimmten Vertriebseinheit selektiert. Auf der dadurch entstandenen Datenmenge wird die Zielverteilung auf Codebasis anhand einer relativ oder absolut vorgegebenen Verteilung vorgenommen.

Aktions-ID	1045		Beschreibung:	
Code	430 (blauer Lack)		Anpassung der Lackfarbe bei Kombis im Zeitraum 03/2015 der Vertriebseinheit X45.	
	Zeitraum	032015	Vertriebseinheit	X45
	Relative Verteilung	30	Absolute Verteilung	-

Tabelle 1 - Beispielhafte Aktion für die Anpassung von Codes mittels Zielverteilung

Das Beispiel in Tabelle 1 zeigt eine Aktion, die die Verteilung des Codes für blauen Lack anpasst. Die Zielverteilung ist dabei relativ angegeben und bewirkt daher eine Anpassung des Anteils von Kombis mit blauem Lack, die im März 2015 in Vertriebseinheit X45 produziert wurden, auf 30% aller in diesem Zeitraum und der Vertriebseinheit produzierten Kombis.

Die Aktionen werden dann durch Modifikationen an den Codes der selektierten Aufträge durchgeführt. Die Modifikation besteht entweder aus einem Insert, falls der Anteil des Codes in der Auftragsmenge erhöht werden soll, einem Delete, wenn er gesenkt werden soll, oder bei einem Ersetzen von Codes aus einer Kombination aus Insert und Delete. Das obige Beispiel bedingt den Insert des Codes 430 oder die Ersetzung von Codes anderer Lackfarben mit dem Code für den blauen Lack, falls in der selektierten Auftragsmenge weniger als 30% mit dem Code für blauen Lack existieren. Gibt es jedoch mehr als den vorgegebene relativen oder absoluten Anteil in der Auftragsmenge, werden als Modifikation für diese Aktion entweder Deletes oder wiederum Codeersetzungen durchgeführt.

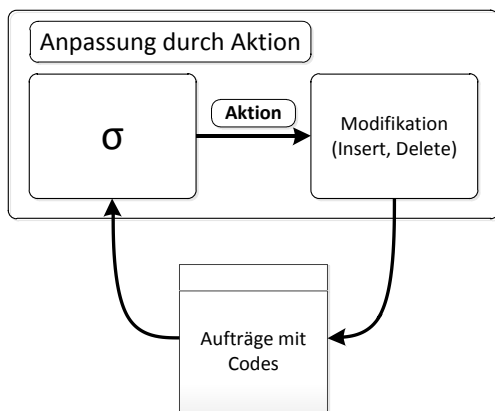


Abbildung 7 - Schematische Darstellung der Anpassung durch Aktionen

Abbildung 7 zeigt eine schematische Darstellung der Ausführung von Aktionen auf den Aufträgen mit den zugeordneten Codes. Dabei wird aus der kompletten Datenmenge der Aufträge der für die Aktion relevante Teil anhand eines oder mehrerer Qualifikationsmerkmale selektiert. Anschließend werden die für die Durchführung der Aktion erforderlichen Anpassungen vorgenommen. Dabei werden die Aktionen in einem Set in beliebiger Reihenfolge ausgeführt, wobei das Ergebnis nicht sofort in die Daten eingepflegt wird, sondern alle Ergebnisse am Ende zusammengefasst in die Datenbank geschrieben werden.

Da diese Vorgehensweise für jede Aktion durchgeführt werden muss, entsteht durch die Anpassung der Aufträge mit Hilfe von Aktionen ein enormer Aufwand. In dem vorliegenden System werden so in jedem Durchlauf ca. 25.000 Aktionen ausgeführt, die aus einer Datenmenge von bis zu drei Millionen Aufträgen selektieren müssen. Dabei gilt zu beachten, dass jeder Auftrag wiederum über durchschnittlich 96 Codes verfügt, wodurch die zu überprüfende Datenmenge enorm ansteigt. Das Hauptaugenmerk in dieser Arbeit liegt auf dieser hier dargestellten Problematik des Datenmanagements bei der Ausführung der Anpassungen.

Die für das betrachtete System verwendete relationale Datenbank ist mittels verschiedener Mechanismen wie Partitionierung und Indexierung weitestgehend optimiert, wodurch die Performancevorgaben des Kunden bei aktuellem Marktverhalten eingehalten werden können. Die Entwicklung der Märkte kann jedoch eine Steigerung des Auftragsvolumens oder eine höhere Anzahl an Komponenten pro Auftrag bedingen, was einen direkten Einfluss auf die zu verarbeitende Datenmenge hat. Durch ein höheres Auftragsvolumen oder eine höhere Anzahl an Codes pro Auftrag steigt die Datenmenge aus der die Aufträge für die Aktionen selektiert werden, was zu einem Performanceproblem im derzeitigen System führen kann. Daher soll in dieser Arbeit eine performante skalierbare Lösung aus dem Big Data Bereich erörtert werden, wodurch die im nächsten Abschnitt beschriebenen Beeinträchtigungen der relationalen Lösung minimiert oder gar eliminiert werden können.

4.3 Grenzen der traditionellen relationalen Lösung

Wenn in dieser Arbeit von einer traditionellen relationalen Lösung gesprochen wird, ist damit ein klassisches relationales Datenbanksystem gemeint, das auf einer einzelnen Maschine läuft. Solche Maschinen sind bei großen Firmen wie Autoherstellern meist Mainframes.

Um den Umstieg auf ein Big-Data-Framework wie Spark für das im vorherigen Abschnitt beschriebene Problem zu rechtfertigen muss betrachtet werden, in welcher Hinsicht die bisherige Lösung unzureichend ist. Da die gängigen relationalen Datenbanksysteme auf das System R zurückzuführen sind, müssen dafür auch architektonische Entscheidungen des System R betrachtet werden, um so die Grenzen von traditionellen relationalen Lösungen zu erfassen [20]. Im Folgenden werden einige von Stonebraker et. al identifizierten Design-Entscheidungen, welche einen Performance-Overhead produzieren können, näher betrachtet [20][25]. Dabei wird hier vorrangig auf die für das Kernproblem relevanten Aspekte eingegangen.

Recovery/ Wiederherstellungs-Strategie

Nach dem Auftreten eines Ausfalls, sowohl Hardware- als auch Softwareseitig, muss die Datenbank in einen konsistenten Zustand zurückgeführt werden. Dadurch wird sichergestellt, dass keine Schreiboperation unvollendet bleibt. In typischen traditionellen relationalen Lösungen wurde dies über ein Log-basiertes Recovery-Subsystem realisiert [24]. Dazu müssen die Logs gesammelt und auf einen stabilen Speicher geschrieben sowie verwaltet werden. Durch die dafür notwendige Überwachung der ausgeführten Aktionen und die I/O-Zugriffe auf die Festplatte des Systems entsteht maßgeblicher Overhead, der heutzutage durch andere Techniken wie zum Beispiel Replikation in einem Cluster-System überflüssig werden kann. [25]

Plattenorientierte Speicherung

In den 1970er Jahren hatten starke Rechner etwa einen Megabyte an Arbeitsspeicher, also nicht ausreichend Volumen für eine relationale Datenbank, was zur Plattenorientierten Speicherung führte. In der heutigen Zeit sind an die 100 GB und mehr jedoch gängig Arbeitsspeichergrößen bei starken Rechnern. [20]

Eine Verlagerung der Datenspeicherung in den Arbeitsspeicher macht den Puffer-Manager, der für die Bereitstellung der vom DBMS benötigten Speicherseiten im Arbeitsspeicher zuständig ist, überflüssig. Dadurch entfällt ein Level von Indirektion bei Zugriffen auf den Speicher und die Performance kann verbessert werden. [25]

Skalierung

Da die meisten traditionellen relationalen Datenbanksysteme in der Produktion auf Mainframes laufen, also auf einzelnen Maschinen mit sehr hoher Rechenleistung und viel Speicher, werden diese über eine so genannte *scale-up*-Strategie skaliert, auch vertikale Skalierung genannt. Dabei werden dem System weitere Hardware-Ressourcen hinzugefügt um so mehr Daten speichern und verarbeiten zu können. Dank der Preisentwicklung bei Hardwarekomponenten wird heute jedoch eine *scale-out* Strategie bevorzugt, bei der die Daten über mehrere Rechner in einem Cluster hinweg partitioniert werden, das sogenannte horizontale Skalieren. [20]

Durch die Datenhaltung und die Verarbeitung auf mehreren Rechnern lässt sich Multithreading, also die gleichzeitige Abarbeitung von Aufgaben in einem Prozess, verhindern, wodurch das sogenannte *Latching* nicht benötigt wird. Das *Latching* ist eine Art leichtgewichtige Sperre um gemeinsam verwendete Datenstrukturen, wie zum Beispiel den Puffer-Pool, zu managen [26]. [25]

Verwaltung und Optimierung

Zudem soll hier noch auf einen wirtschaftlichen Aspekt eingegangen werden. In der Zeit, in der die heutigen relationalen Systeme ihren Ursprung hatten, war Hardware noch sehr teuer und Arbeitsstunden waren günstig. Aus dieser Situation entstanden Systeme, die zur Verwaltung und Optimierung die Arbeit einer Reihe von Datenbankspezialisten benötigten. Heute ist es in etwa andersherum, gewöhnliche Hardware ist günstig zu erstehen, wohingegen die Arbeitsstunden teuer geworden sind. Dementsprechend haben sich die Anforderungen an Datenbanksysteme geändert und Systeme sollen heute „self-everything“ sein, sich also um alles selbst kümmern [20]. Moderne RDBMS sind dazu ebenso in der Lage wie neu entstandene Datenverarbeitungssysteme aus dem Big Data Bereich, wodurch die Kosten von den Datenbankspezialisten hin zu Lizenzkosten verschoben werden.

RDBMS-Overhead im Überblick

Um das Ausmaß des Overheads in relationalen Datenbanksystemen aufzuzeigen erstellten Harizopoulos et al. eine Versuchsreihe, bei der ein bestehendes System iterativ verändert wurde um so die Auswirkungen zu zeigen, die mit der Eliminierung eines der Overhead-Erzeuger einhergehen [25].

Die Ergebnisse des Versuches sind in Abbildung 8 dargestellt und anhand der Ausführung eines Benchmarks für OLTP Systeme ermittelt [27]. Dabei zeigt die Grafik deutlich, dass die hier angesprochenen Aspekte mit etwa 60%, also *buffer manager*, *latching*, und *logging* zusammen, viel Verwaltungsarbeit für eigentlich wenig produktive Arbeit, etwa 2-3% der insgesamt gebrauchten Instruktionen, bedeuten. Im Umkehrschluss heißt das, dass durch die Reduzierung oder gar Eliminierung des Bedarfs von *buffer manager*, *latching* und *logging*, zum Beispiel durch die Speicherung im Arbeitsspeicher, bereits etwa 60% der benötigten Instruktionen eingespart werden können. [25]

Abbildung 8 zeigt, dass neben den hier beschriebenen Aspekten noch weitere Bereiche existieren, welche die Performance verringern können. Dazu gehören unter anderem der Mehrbenutzerbetrieb und die dafür eingesetzten Locking-Mechanismen zur Sicherstellung der Konsistenz, sowie Mechanismen zur Sicherstellung der Verfügbarkeit, zum Beispiel durch Replikation. [25]

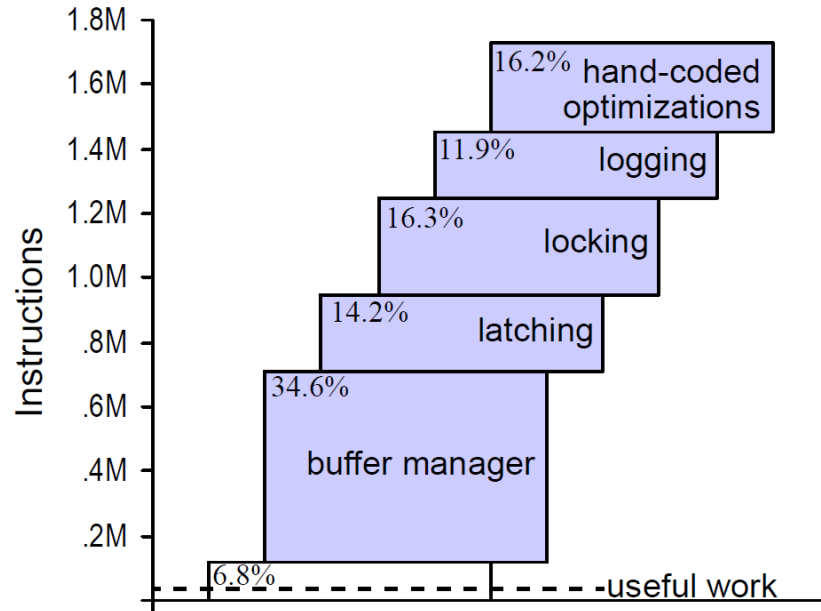


Abbildung 8 – aufgeschlüsselter Overhead von RDBMS [25]

In dieser Arbeit soll an einem konkreten Beispiel evaluiert werden, ob der Einsatz eines Big-Data-Frameworks einen Performancevorteil gegenüber dem verwendeten traditionellen relationalen System bringt. Dabei wird im nächsten Kapitel ein Framework vorgestellt, welches anhand problembezogener Kriterien fundiert ausgewählt wurde. Der mit diesem Framework entwickelte Prototyp wird abschließend evaluiert und so potentielle Performanceverbesserungen aufgezeigt.

5 Apache Spark

Spark entstand an der University of California, Berkley als Forschungsprojekt des AMPLab¹ [28]. Ziel war es, eine einheitliche Engine für die Verarbeitung von großen Datenmengen zu entwerfen, welche die Entwicklung von Anwendungen durch den Einsatz von high level APIs vereinfacht. Spark wurde in Scala geschrieben und basiert auf dem Akka² Toolkit.

Nach einem Open Source Release im Jahr 2010 und einer aktiven Weiterentwicklung wurde Spark 2013 in den Apache Incubator aufgenommen und ist seit Anfang 2014 als Apache Top-Level-Projekt deklariert [28][30]. Mit über 400 Entwicklern aus über 100 Unternehmen weltweit wird Apache Spark als eines der größten Top-Level-Projekte immer noch weiter entwickelt [28]. Mit Spark selber oder auf Grundlage von Spark wurden seitdem Rekorde für zwei Benchmarks aufgestellt. Dabei wurde 2014 der Rekord im GraySort Benchmark, bei dem der reine Durchsatz beim Sortieren großer Datenmengen in TB/min gemessen wird, und 2016 der Rekord im CloudSort Benchmark, bei dem der Aufwand von Cloud-Ressourcen, genauer gesagt die Kosten, für eine bestimmte Menge an zu sortierenden Daten gemessen wird, unterboten [29].

In diesem Kapitel werden die Grundlagen von Spark erläutert. Dabei wird neben einem Überblick und den grundsätzlichen Konzepten von Spark in 5.1 und 5.2 auch auf die in Spark fest integrierten Module eingegangen. Abschließend wird Spark mit anderen Frameworks verglichen und geschildert, warum Spark als Framework für die Bearbeitung des in Kapitel 4 beschriebenen Kernproblems herangezogen wird.

5.1 Architektur

Abbildung 9 zeigt, dass das Spark Framework aus mehreren Komponenten bzw. Modulen besteht. Neben den Modulen benötigt Spark zudem einen Cluster-Manager, welcher für die Bereitstellung und das Scheduling von Ressourcen unter verschiedenen Spark-Programmen zuständig ist. Spark bietet dafür mit dem Standalone Scheduler einen eigenen Cluster-Manager, der in Abbildung 9, wie alle zu Spark zugehörigen Bestandteile, in dunkelblau dargestellt ist. Zusätzlich ist es mit Spark jedoch auch möglich anderen Cluster-Managern, Apache Mesos oder Hadoop YARN, diesen Job zu überlassen. [33]

¹ Algorithms, Machines and People Lab: <https://amplab.cs.berkeley.edu>

² Ein Toolkit zum Erstellen stark verteilter, paralleler Systeme: <http://akka.io/>

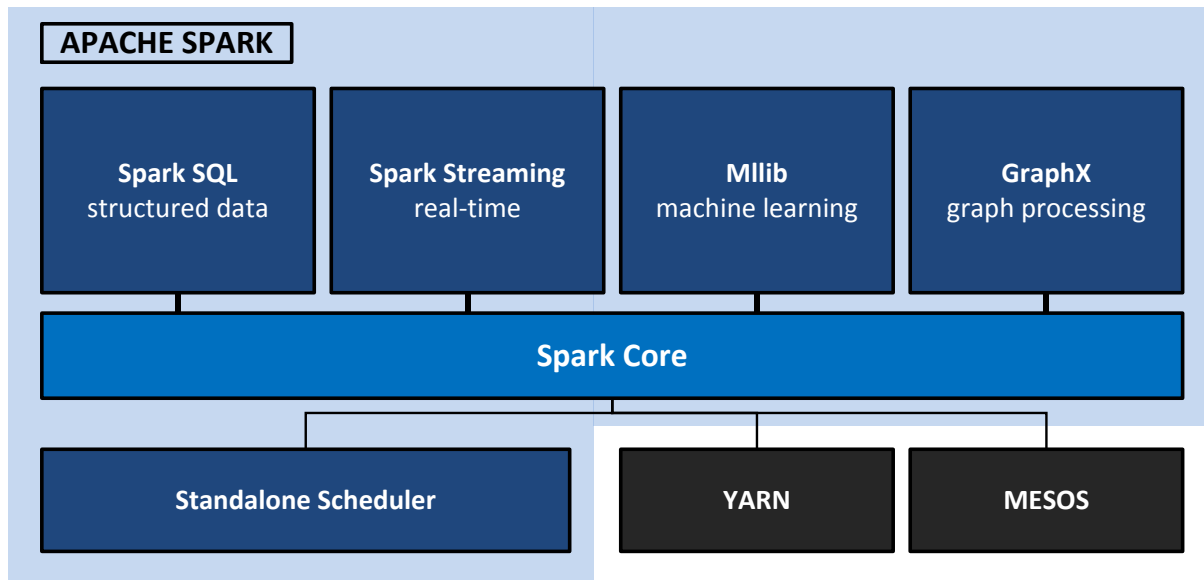


Abbildung 9 - Spark Komponenten (nach [31])

Spark verfügt neben dem Spark Core über einige Module, welche auf die Datenverarbeitung von Use-Cases spezialisiert sind. Darunter befinden sich neben Modulen für die Verarbeitung von Graphen und gestreamten Daten, sowie einem Modul für Machine Learning auch das Modul Spark SQL, welches für den Zugriff auf relationale Daten zuständig ist und in Abschnitt 5.3 näher erläutert wird. Dabei wurde mit Spark jedoch nicht versucht diese Frameworks in ihrem Spezialgebiet zu verbessern, vielmehr war es das Ziel, eine einheitliche Plattform für die verteilte Verarbeitung großer Datenmengen zu schaffen. Einer der Erfinder von Spark brachte als Beispiel den Vergleich zwischen Smartphones und den vorher existierenden portablen Geräten, wie Kameras, Walkmans und Mobiltelefonen. Dabei übernahm das Smartphone die Rolle der einheitlichen Plattform, die alle Funktionen der bisherigen Geräte vereinte und weitere neue Anwendungen, wie zum Beispiel die Kombination bisher nur auf unterschiedlichen Geräten verfügbarer Funktionen, anbot. [34]

Abbildung 10 zeigt die Ausführungsumgebung eines Spark-Programms, welches über einen SparkContext, das eigentliche Benutzerprogramm, verfügt, das die Abgrenzung zu anderen Spark-Programmen darstellt. Das Programm läuft auf dem Master-Knoten im sogenannten Driver-Prozess, welcher für die allgemeine Programmausführung zuständig ist und eine Menge sogenannter Executor-Prozesse auf den Worker-Knoten erstellt, welche ausschließlich für die Bearbeitung von Aufgaben des erstellenden Programms zuständig sind. Dabei werden vom Driver-Prozess Aufgaben, respektive Tasks, an die Executor-Prozesse verteilt. Die Ressourcen-Bereitstellung wird dabei vom Cluster-Manager übernommen. [32][35]

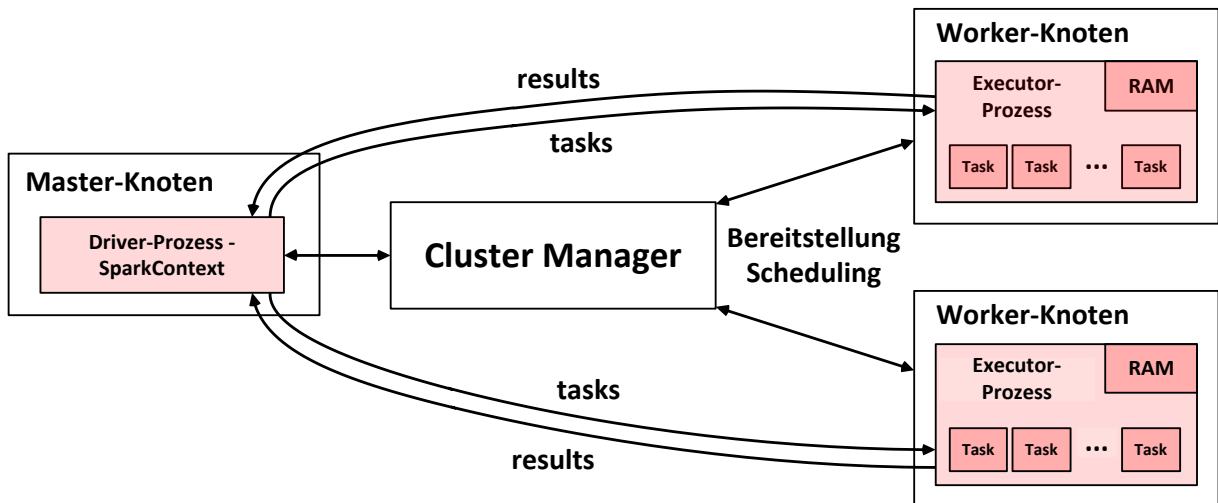


Abbildung 10 - Ausführungsumgebung Spark (nach [35] und [32])

Ein Spark-Programm führt sogenannte Jobs aus, welche Aktionen auf der gesamten Datenmenge darstellen. Diese Jobs werden bei der Ausführung in Aufgaben unterteilt, die vom Driver an die Executor-Prozesse auf den Worker-Knoten verteilt werden. Die Steuerung der Programmausführung vom Driver umfasst dabei auch das Scheduling dieser Jobs, weshalb der Master- und die Worker-Knoten idealerweise im selben LAN sein sollten um dadurch eine geringe Latenz zu gewährleisten [31]. Standardmäßig wird ein FIFO-Scheduling angewandt, was dazu führen kann, dass hinter Jobs mit langen Ausführungszeiten platzierte Jobs warten müssen und deswegen eine hohe Latenz aufweisen. Dies kann in Spark³ je nach Situation jedoch durch das Umstellen auf ein Fair-Scheduling verhindert werden, bei dem jedem Job etwa gleich viele Ressourcen des Clusters zur Verfügung gestellt werden. [33]

5.2 Spark Core

Die Core-Komponente umfasst die wesentlichen Konzepte von Spark. Im folgenden Abschnitt wird dabei neben einer ausführlichen Darstellung der Resilient Distributed Datasets, nachfolgend RDD, eine Erläuterung des Speichermanagements und der Fehlertoleranz von Spark gegeben.

5.2.1 RDD – Resilient Distributed Datasets

Das wichtigste Konzept in Spark sind die Resilient Distributed Datasets, welche eine Abstraktion einer Datenmenge darstellen [35]. RDDs sind schreibgeschützte und verteilte Zusammenstellungen von Dateneinträgen mit einem Datentypen, welche durch deterministische Operationen, sogenannte *Transformationen*, auf Daten im persistenten Speicher oder anderen RDDs erzeugt werden. Auf den RDDs können nach der Erstellung weitere Operationen ausgeführt

³ Ab Version 0.8

werden, welche einen Wert berechnen und an die Programmausführung zurückgeben oder Daten in den persistenten Speicher schreiben. Diese Operationen werden *Aktionen* genannt. [35]

Ein RDD wird als logischer Plan, einer Folge von Transformationen angelegt. Die Abstammungslinie wird vom Driver-Prozess verwaltet und immer bei der Ausführung einer Aktion auf dem RDD für die Berechnung bzw. Neuberechnung herangezogen. Diese Vorgehensweise hat zwei essentielle Vorteile. Zum einen kann die Erstellung des RDDs durch Transformationen von Spark optimiert werden, da ein RDD bis zur Ausführung der ersten Aktion lediglich als logischer Plan vorliegt. Zum anderen wird dadurch der Datenaustausch zwischen verschiedenen Berechnungen auf den Daten ermöglicht, da das RDD laufend neu berechnet wird. [34]

5.2.1.1 Operationen auf RDDs

Es gibt zwei unterschiedliche Arten von Operationen, die auf RDDs ausgeführt werden können. Zum einen die *Transformationen*, mit denen ein neues RDD erstellt werden kann, zum anderen *Aktionen*, welche entweder einen Wert berechnen und an das Programm zurückgeben, oder Daten in den Speicher schreiben. Tabelle 2 gibt einen Überblick über die in Spark verfügbaren Operationen.

Transformationen	$map(f: T \Rightarrow U)$: $RDD[T] \Rightarrow RDD[U]$
	$filter(f: T \Rightarrow Bool)$: $RDD[T] \Rightarrow RDD[T]$
	$flatMap(f: T \Rightarrow Seq[U])$: $RDD[T] \Rightarrow RDD[U]$
	$sample(fraction: Float)$: $RDD[T] \Rightarrow RDD[T]$ (deterministic sampling)
	$groupByKey()$: $RDD[(K, V)] \Rightarrow RDD[(K, Seq[V])]$
	$reduceByKey(f: (V, V) \Rightarrow V)$: $RDD[(K, V)] \Rightarrow RDD[(K, V)]$
	$union()$: $(RDD[T], RDD[T]) \Rightarrow RDD[T]$
	$join()$: $(RDD[(K, V)], RDD[(K, W)]) \Rightarrow RDD[(K, (V, W))]$
	$cogroup()$: $(RDD[(K, V)], RDD[(K, W)]) \Rightarrow RDD[(K, (Seq[V], Seq[W]))]$
	$crossProduct()$: $(RDD[T], RDD[U]) \Rightarrow RDD[(T, U)]$
	$mapValues(f: V \Rightarrow W)$: $RDD[(K, V)] \Rightarrow RDD[(K, W)]$ (preserve partitioning)
	$sort(c: Comparator[K])$: $RDD[(K, V)] \Rightarrow RDD[(K, V)]$
	$partitionBy(p: Partitioner[K])$: $RDD[(K, V)] \Rightarrow RDD[(K, V)]$
Aktionen	$count()$: $RDD[T] \Rightarrow Long$
	$collect()$: $RDD[T] \Rightarrow Seq[T]$
	$reduce(f: (T, T) \Rightarrow T)$: $RDD[T] \Rightarrow T$
	$lookup(k: K)$: $RDD[(K, V)] \Rightarrow Seq[V]$ (On hash/range partitioned RDDs)
	$save(path: String)$: Outputs <i>RDD</i> to a storage system, e.g., HDFS

Tabelle 2 - Verfügbare Operationen auf RDDs (nach [35])

Matei et al. orientierten sich bei der Benennung der Operationen an anderen funktionalen Sprachen, wodurch eine Vereinfachung der Handhabung bei Entwicklern erzielt werden konnte, die sich bereits mit gleichen Funktionen einer funktionalen Sprache auseinandergesetzt haben [35].

Zusätzlich zu den hier aufgeführten Operationen können Entwickler zwei weitere Funktionen von RDDs nutzen. So kann über die entsprechenden Aufrufe die Persistenz und die Partitionierung eines RDD gesteuert werden. Einerseits kann durch das Setzen des Flags *persist* die Wahl einer Speicherstrategie festgelegt werden, wodurch z.B. ein RDD persistent im in-memory Speicher gehalten, oder ausschließlich im beständigen Speicher abgelegt wird, andererseits können durch die Definition einer Partitionierungsordnung mittels der *Partitioner*-Klasse mehrere RDDs anhand dieser partitioniert werden. [35]

5.2.1.2 Darstellung von RDDs

Um ein RDD zu beschreiben bedarf es einer Technik, welche auf ein breites Spektrum an Transformationen, also auch auf beliebige Zusammenstellungen von Transformations-Operationen, anwendbar ist. Matei et al. stellen einen simplen graphenbasierten Ansatz zur Darstellung von RDDs vor [35].

In diesem werden RDDs durch ein Interface dargestellt, welches über die folgenden fünf Informationen verfügt:

- Einen Satz Partitionen
- Einen Satz Abhängigkeiten (*dependencies*) von den Eltern-RDDs, aus denen das jeweilige RDD erzeugt wurde
- Eine Funktion zur Berechnung des Datensets auf Grundlage der Abhängigkeiten
- Metadaten über das Partitionierungs-Schema
- Metadaten über die Daten-Platzierung

Abhängigkeiten von den Eltern-RDDs, also die auf den Eltern-RDD ausgeführte Transformationsoperation, werden bei Matei et al. in eingeschränkte und umfangreiche Abhängigkeiten unterteilt. Die Unterscheidung liegt hierbei ausschließlich in der Häufigkeit der Nutzung der Partitionen des Eltern-RDDs, von denen die Partition des Kind-RDDs abhängig ist.

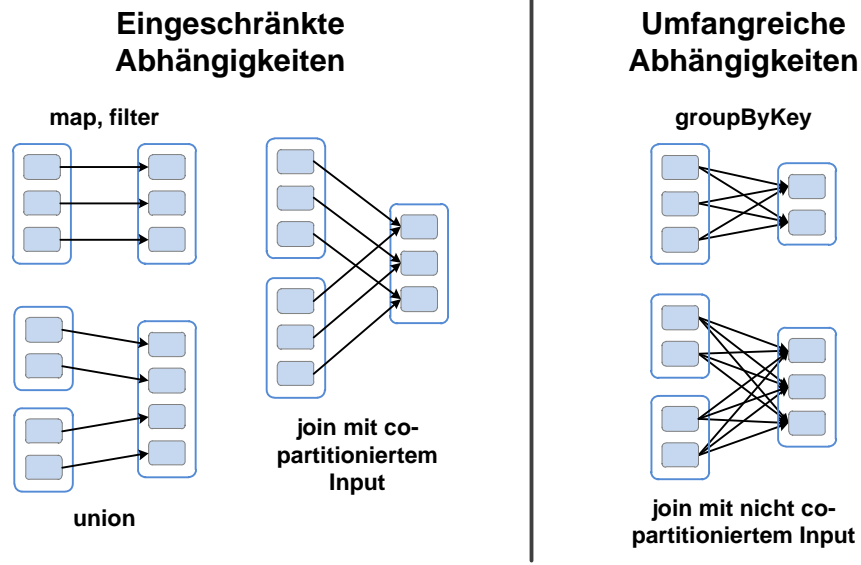


Abbildung 11 - RDD-Abhängigkeiten (nach [35])

Wie in Abbildung 11 zu erkennen, werden die Partitionen des Eltern-RDDs bei eingeschränkter Abhängigkeit maximal von einer Partition des Kind-RDDs genutzt, wohingegen bei umfangreichen Abhängigkeiten für mehrere Partitionen dieselbe Partition des Eltern-RDDs genutzt werden.

Diese Unterscheidung macht sich auch bei Erstellung und Wiederherstellung von RDDs bemerkbar. So können die Transformationsoperationen zur Erstellung eines RDDs, das ausschließlich durch eingeschränkte Abhängigkeiten gebildet wird, auf einem einzigen Knoten im Cluster aneinandergereiht ausgeführt werden ohne dabei Datentransfer zwischen verschiedenen Knoten zu benötigen. Bei der Wiederherstellung nach einem Knotenausfall ersparen eingeschränkte Abhängigkeiten Berechnungsarbeit, da nur die verlorengegangenen Partitionen des Eltern-RDDs neu berechnet werden müssen. Bei umfangreichen Abhängigkeiten kann sich dies auf die Neuberechnung kompletter RDDs ausweiten.

5.2.2 Speicher-Management

Spark verfügt nach Matei et al. [35] über die folgenden drei Speicheroptionen:

- Im Arbeitsspeicher als deserialisiertes Java-Objekt
- Im Arbeitsspeicher als serialisierte Daten
- Im beständigen Speicher auf der Festplatte

Während im Arbeitsspeicher die Speicherung als deserialisiertes Java-Objekt Performancevorteile bringt, da die JVM von Spark nativ auf die Daten zugreifen kann, ist die Speicherung serialisierter Daten speichereffizienter, was jedoch auf Grund der notwendigen Deserialisierung bei der Performance negative Auswirkungen zeigt. Für das Management des vorhandenen Speichers wird die Least-recently-used Methodik auf Ebene der RDDs herange-

zogen. Ist der vorhandene Speicher ausgenutzt und es soll eine weitere Partition eines RDDs in den Speicher geladen, bzw. berechnet werden, so wird eine Partition des am wenigsten genutzten RDDs entfernt. Eine Ausnahme besteht hier allerdings. Sobald eine neue Partition dem seltenst genutzten RDD zugeordnet wird, bleibt dieses intakt und eine Lösungsalternative wird gesucht. Diese trifft stattdessen das am zweit-wenigsten genutzte RDD um einen Zyklus des in und aus dem Speicher Nehmens von Partitionen desselben RDDs zu verhindern. [35]

Die Option der Speicherung im beständigen Speicher hingegen ist nur für RDDs bzw. Partitionen sinnvoll, die entweder zu groß für den Arbeitsspeicher sind oder deren ständige Neuberechnung zu teuer ist. Matei et al. verdeutlichten den dadurch entstehenden Performanceverlust in einer Messung der Berechnung einer logistischen Regression auf 100GB Datenvolumen [35].

Abbildung 12 zeigt die gleichmäßige Abnahme der Performance bei Reduzierung des vorhandenen Speichers. Dabei wurde Spark vorgegeben, wie viel des vorhandenen Speichers genutzt werden darf.

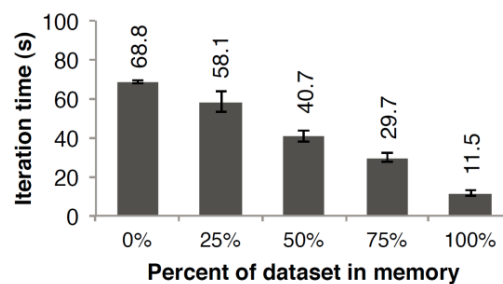


Abbildung 12 - Performance bei verschiedenen Speichergrößen [35]

5.2.3 Fehlertoleranz

Bei Sparks Toleranz gegen Fehler wird an dieser Stelle auf die Erholung nach einem Knotenausfall eingegangen. Durch die Verwaltung der Abstammungslinie der RDDs können komplette RDDs oder auch einzelne Partitionen eines RDDs bei einem Ausfall schnell wiederhergestellt werden. Diese Art der Wiederherstellung hat zwei Vorteile:

- Performance – Die Daten müssen nicht über das Netzwerk gesendet werden, sondern einfach nur in den Arbeitsspeicher geschrieben werden.
- Speichereffizienz – Es müssen keine Kopien der Daten an sich angelegt, sondern lediglich die Abstammungslinien der RDDs gespeichert werden

Dabei muss aufgrund des Schreibschutzes kein Aufwand zur Sicherstellung der Konsistenz betrieben werden und die verlorengangenen Aufgaben können parallel auf verschiedenen Maschinen nachgearbeitet werden. Auch der Datenaustausch zwischen den Knoten kann hierbei rekonstruiert werden, da der Sender die Daten lokal im Speicher hält. Matei et al. zeigen anhand eines Versuches, dass sich Spark nach einem Knotenausfall schnell wieder erholt. [35]

Der Versuch umfasst dabei die Berechnung des k-means Algorithmus auf 75 Knoten für 10 Iterationen, bestehend aus 400 Aufgaben auf einer 100 GB großen Datenmenge. Abbildung 13 vergleicht die Ergebnisse von Ausführungen mit Knotenausfall am Anfang der sechsten Iteration und ohne Knotenausfall.

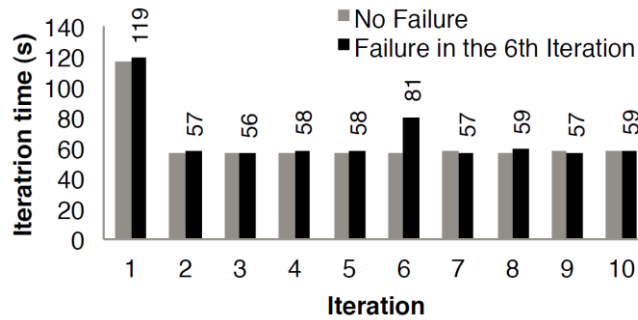


Abbildung 13 - Ausführungszeit der Iterationen mit und ohne Knotenausfall [35]

Es ist deutlich zu erkennen, dass Spark in der Iteration mit dem Knotenfehler eine erhöhte Ausführungszeit hat, da hier die verlorengegangenen Partitionen und Aufgaben neu berechnet werden müssen. Nach der Wiederherstellung springt die Ausführungszeit wieder auf das vorherige Niveau ohne einen Neustart zu benötigen.

5.3 Spark SQL

Mit dem Release von Spark SQL im Jahr 2014 versuchten die Entwickler den Nutzen der funktionalen und der prozeduralen Programmierung zu kombinieren. Anhand der weiten Verbreitung von relationalen Systemen wurde dabei angenommen, dass Entwickler deklarative Anfragen vorziehen. Da jedoch sowohl der Zugriff auf semi- und unstrukturierte Daten, wie auch die Berechnung komplexer Analysen mit relationalen Systemen eine Herausforderung darstellen, wurde versucht dies durch die Kombination mit der prozeduralen Programmierung zu vereinfachen. [36]

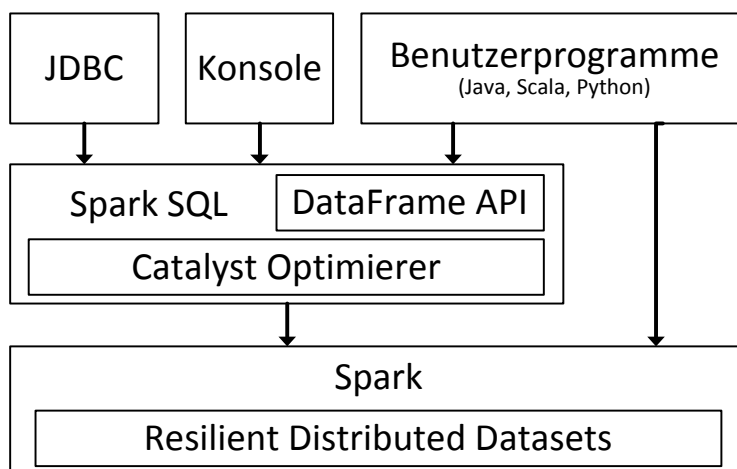


Abbildung 14 - Spark SQL Architektur (nach [36])

In Abbildung 14 ist zu sehen, dass Spark SQL als Bibliothek auf Spark läuft und zum einen die *DataFrame* API und zum anderen den *Catalyst*-Optimierer bietet. Der Zugriff erfolgt dabei über verschiedenste Technologien, wie JDBC/ODBC, die Konsole oder Benutzerprogramme.

5.3.1 DataFrame API

Ein *DataFrame* ist die Abstraktion einer verteilten Kollektion von Daten, die aus Datensätzen mit einem homogenen Schema besteht. Dadurch wurde in Spark SQL eine Darstellung von Daten geschaffen, die sowohl gleichwertig zu einer Tabelle in einem relationalen Datenbanksystem ist, aber auch als RDD von Datensätzen gesehen werden kann. Ein *DataFrame* verbindet somit die Verarbeitungs- und Optimierungsmöglichkeiten eines RDD mit der Unterstützung von Datenstrukturen und relationalen Operationen. Zur Erstellung eines *DataFrame*s können verschiedenste Quellen, wie strukturierte Daten, Dateien und existierende RDDs herangezogen werden [37]. Diese werden, wie aus dem Spark Core bereits bekannt, als logischer Plan angelegt um eine Optimierung vor der Ausführung zu ermöglichen. [36]

Das Datenmodell von *DataFrames* basiert dabei auf Betrachtungen von Hive [38] und unterstützt die wichtigsten SQL-Datentypen und komplexe Datentypen wie Structs, Arrays, Maps und Unions. Mittels einer Domain-spezifischen Sprache können alle gängigen relationalen Operationen, wie Projektionen, Selektionen, Filter, Vereinigungen und Aggregationen ausgeführt werden. Der dadurch entstehende Ausdruck wird als abstrakter Syntaxbaum zur Optimierung weitergegeben, was im folgenden Abschnitt noch näher betrachtet wird. *DataFrames* können alternativ auch als temporäre Tabellen im Systemkatalog abgelegt werden und ermöglichen so Anfragen mittels SQL. [36]

Die API bietet zudem ein *In-Memory Caching* und *inline User Defined Functions* (UDF). Das Caching ermöglicht eine spaltenbasierte Materialisierung von *DataFrames* im Arbeitsspeicher und verbraucht dank spaltenbasierter Kompression - durch dictionary encoding und run-length encoding - ein vielfaches weniger Speicher als Sparks eingebautes Caching. Dabei resultiert der geringere Speicherbedarf hier aus der Tatsache, dass bei spaltenorientierter Speicherung mit höherer Wahrscheinlichkeit gleiche Werte hintereinander gespeichert werden, die zum Beispiel mit einer run-length Komprimierung zusammengefasst werden. Inline UDFs bieten den Vorteil, dass hier Funktionen der jeweils gewählten Programmiersprache und -umgebung genutzt werden können, welche wiederum auf den vollen Funktionsumfang der Spark API zurückgreifen können. [36]

5.3.2 Catalyst

Catalyst ist ein auf Scala basierender erweiterbarer Optimierer, der sowohl regelbasierte als auch kostenbasierte Optimierung unterstützt. Der Kern des Optimierers ist eine Bibliothek zur Darstellung von Bäumen und der Anwendung von Regeln auf diese Bäume. Bereits zur *DataFrame* API erwähnt, werden aus den Ausdrücken sogenannte abstrakte Syntaxbäume erstellt, welche an Catalyst zur Optimierung übergeben werden. [36]

Diese Catalyst-Bäume bestehen aus Knoten mit einem Knotentyp und null oder mehr Kind-Knoten. Abbildung 15 zeigt einen Baum, der aus den folgenden Knotentypen besteht:

- Literal (value: Int)
- Attribute (name: String)
- Add (left: TreeNode, right: TreeNode)

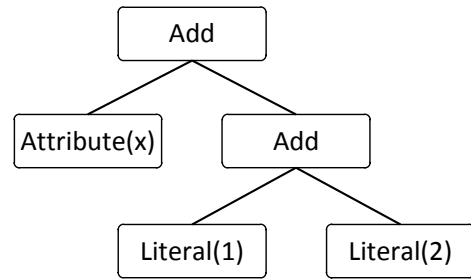


Abbildung 15 - Catalyst-Baum für den Ausdruck $x + (1 + 2)$ (nach [36])

Weitere Knotentypen können dabei als Subklasse von `TreeNode` erstellt werden.

Für die Regel-basierte Optimierung stehen in Catalyst Funktionen zur Verfügung, die einen Baum in einen anderen Baum überführen. Diese Funktionen werden rekursiv auf den Catalyst-Baum angewandt bis keine weiteren Veränderungen eingesteuert werden. Die Bedingung zur Überführung in einen neuen Baum kann beliebigen Scala-Code beinhalten, wird aber meistens durch einen Musterabgleich realisiert. [36]

Für den Einsatz in Spark SQL existieren bereits Bibliotheken und Regeln, die für die Verarbeitung relationaler Anfragen entworfen wurden. Diese werden bei der Ausführung einer Anfrage in den in Abbildung 16 ersichtlichen Phasen genutzt um die Anfrage zu optimieren.

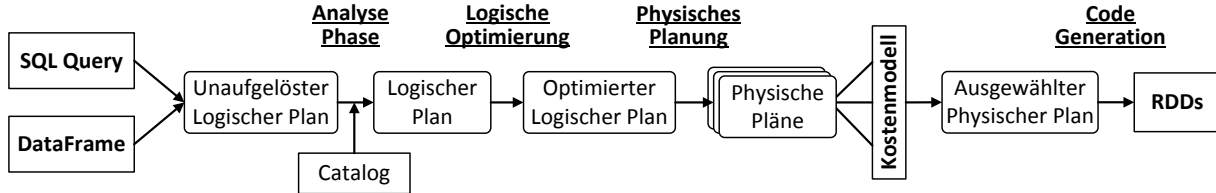


Abbildung 16 - Anfrage-Planung in Spark SQL - abgerundete Rechtecke stellen Catalyst-Bäume dar (nach [36])

In der *Analyse-Phase* werden, falls vorhanden, ungeklärte Attribut-Referenzen und Relationen aufgelöst. Dabei werden zum Beispiel Namen im Katalog überprüft und Zweideutigkeiten aufgelöst. Der durch die Analyse entstehende logische Plan wird in der nächsten Phase, der *Logischen Optimierung*, mithilfe von Regeln weiter optimiert. Diese umfassen aus relationalen Systemen bereits bekannte Regeln, wie predicate pushdown, constant folding und projection pruning. Während der *physischen Planung* werden ein oder mehrere physische Pläne aus dem optimierten logischen Plan erzeugt, wofür das Aneinanderreihen von Projektionen und verwandte Regeln angewendet werden. Aus den so erzeugten Plänen wird anhand eines Kosten-Modells ein Plan ausgewählt, welcher in der letzten Phase, der *Code Generierung*, in ausführbaren Code umgewandelt wird. [36]

5.3.3 Evaluation

Armbrust et al. führen in [36] einige Versuchsreihen durch um die Performance von Spark SQL zu beurteilen. Die nachfolgend vorgestellten Ergebnisse der Versuche sollen dabei nur einen Einblick in die mögliche Leistung von Spark SQL geben und sind keinesfalls eine allgemeingültige Aussage über die Performance, da diese stark von der Umgebung und den zu bewältigenden Aufgaben abhängt.

Der erste Versuch stellte einen Vergleich von Spark SQL und vergleichbaren Systemen anhand des Big Data Benchmarks des AMPLabs [41] auf, welcher verschiedenste relationale Anfragen auf verschieden großen Datenmengen testet. Dabei wurde festgestellt, dass Spark SQL hinsichtlich der Performance im Vergleich zu seinem Vorgänger Shark [39] stark besser und im Vergleich zu Impala [40], einer SQL Engine für Hadoop, gleichwertig ist. [36]

Bei einem Vergleich der DataFrame API und dem nativen Spark Code, dargestellt in Abbildung 17, konnte dabei gegenüber dem mit der Python API geschriebenen Test-Code eine starke Verbesserung und gegenüber der Scala API eine leichte Verbesserung festgestellt werden. [36]

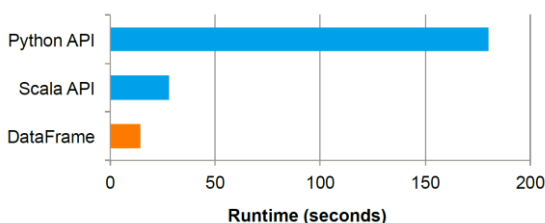


Abbildung 17 - Evaluation DataFrame API [36]

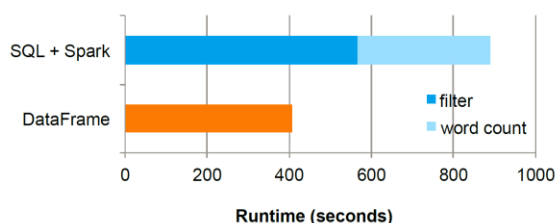


Abbildung 18 - Evaluation Pipeline Performance [36]

Bei dem dritten Versuch wurde aufgezeigt, dass die DataFrame API durch die integrierte Verarbeitung von relationalem und prozeduralem Code die Performance steigert. Dabei wurde eine SQL Anfrage, die Filterung eines Textes, mit einem Scala basiertem Spark Job zum Zählen der Wörter kombiniert und mit der entsprechenden Ausführung in der DataFrame API verglichen. Wie in Abbildung 18 erkennbar braucht die kombinierte Lösung der relationalen Anfrage mit der prozeduralen Berechnung der DataFrame API in etwa die Hälfte der Zeit. Dies rührt von der Tatsache, dass keine Zwischenspeicherung des Ergebnisses der relationalen Anfrage erforderlich ist. [36]

5.4 Erläuterung der Wahl von Spark

Im folgenden Abschnitt wird dargestellt, warum für das vorliegende Problem Spark als Big-Data-Framework herangezogen wird. Dabei werden unterschiedliche Frameworks hinsichtlich bestimmter Kriterien verglichen. Für den Vergleich werden die Frameworks Spark, Hadoop und Flink herangezogen, da diese fähig sind Daten in Stapelverarbeitung zu verarbeiten. Zudem ist insbesondere Hadoop als Vergleichs-Framework aussagekräftig, da viele bestehende

Lösungen in der verteilten Verarbeitung von Daten auf Hadoop basieren oder eine Distribution von Hadoop darstellen.

Der hier vorgestellte Vergleich ist nicht als genereller Vergleich der gewählten Frameworks zu betrachten, da die Kriterien auf das in dieser Arbeit behandelte Problem bezogen ausgewählt sind. Zudem ist die Gewichtung der Kriterien auch auf das hier vorliegende Problem bezogen.

5.4.1 Kriterien

Im Folgenden werden die Kriterien zur Bewertung der einzelnen Frameworks näher erläutert, wobei diese im Anschluss durch einen paarweisen Vergleich gewichtet werden.

Kriterium 1

Lizenzkosten und benötigte Hardware

Dieses Kriterium beschreibt, in wieweit Lizenzkosten für das Framework aufzubringen sind und wie viel finanzieller Aufwand für die Bereitstellung der Hardware betrieben werden muss. Hierbei ist ein niedriger finanzieller Aufwand eine gute Bewertung.

Kriterium 2

Anwendungs-Erweiterbarkeit und Einheitlichkeit des Frameworks

Um den größtmöglichen Nutzen aus den Daten, gleich welcher Art, zu ziehen, muss der Anwender auf ein größtmögliches Repertoire von Ver- und Bearbeitungsfunktionalität zurückgreifen können. Das Kriterium beschreibt, inwiefern es dem Anwender vom Framework ermöglicht wird auch unterschiedlichste Verarbeitungsprozesse auf denselben Daten durchführen zu können. Dies wird zum Beispiel durch die Einheitlichkeit des Datenformats der unterschiedlichen Tools des Frameworks bestimmt. Bietet ein Framework die Möglichkeit die Daten an andere Verarbeitungsprozesse weiterzugeben oder verfügt es über ein sehr großes Repertoire wird es gut bewertet.

Kriterium 3

Verfügbare/Unterstützte Datenformate

Das Datenformat begrenzt oft die Anwendbarkeit eines Frameworks, da die vorliegenden Daten, die bearbeitet werden sollen, entweder nicht in diesem Format vorliegen oder nur unter großem Aufwand in dieses Format gebracht werden können. Das Kriterium beschreibt auf welche Datenformate das Framework zugreifen kann, bzw. für welche Datenformate Wrapper bereitgestellt werden. Werden die wichtigsten Datentypen wie Text-, csv- und JSON-Dateien unterstützt erhält das Framework eine gute Bewertung, die durch Unterstützung weiterer Datentypen noch verbessert werden kann. Werden die wichtigsten Datentypen jedoch nicht unterstützt wird eine schlechte Bewertung vergeben.

Kriterium 4**Anwendungsprogrammierung API**

Die verfügbaren Schnittstellen für die Anwendungsprogrammierer können auch für die Wahl eines Frameworks entscheidend sein. Zum einen liegt dies daran, dass nicht jeder der ein Big-Data-Framework anwenden möchte auch über die entsprechenden Programmierkenntnisse verfügt, die von dem Framework als Möglichkeiten angeboten werden. Zum anderen erschweren unterschiedliche Programmierschnittstellen die Kombination von unterschiedlichen Workflows in einem Programm. Für die Bereitstellung von APIs für weit verbreitete Sprachen wie Java und Python gibt es gute Bewertungen, die durch Bereitstellen weiterer APIs noch verbessert wird.

Kriterium 5**Unterstützung relationaler Daten und Operationen**

Speziell bei dem hier vorliegenden Problem ist es wichtig, dass das gewählte Framework den Zugriff auf relationale Daten und den Zugriff auf die Daten mit SQL so einfach wie nur möglich gestaltet. Ist der Zugriff auf relationale Daten unproblematisch und das Framework unterstützt den SQL-92 Standard wird hier eine gute Bewertung vergeben. Bei Unterstützung eines neueren Standards verbessert sich diese.

Kriterium 6**Recovery und Fehlerverhalten**

Das Framework sollte dem Anwendungsentwickler nicht mit Überlegungen zum Fehlerverhalten und Recoveryansätzen der Ausführungsumgebung belasten. Darüber hinaus sollte die Fehlerbehandlung des Frameworks verlustfrei und performant sein. Muss sich der Anwender nicht um ein performantes Recovery und Fehlerverhalten kümmern gibt es hier eine gute Bewertung.

Kriterium 7**Art des Speichers und Performance**

Die Art des Speichers, der für die zu verarbeitenden Daten verwendet wird, wirkt sich auf die gesamte Performance der Anwendung aus, da dabei Daten gelesen und geschrieben werden. Die Frage, ob das Framework Daten im Arbeitsspeicher oder im persistenten Speicher hält ist dabei essentiell. Neben der Art des Speichers gibt es weitere Einflussfaktoren auf die Performance des Frameworks. Bei Datenspeicherung im Arbeitsspeicher und einer guten Performance gibt es eine gute Bewertung.

Beim folgenden paarweisen Vergleich der Kriterien werden immer zwei Kriterien gegenübergestellt. Dabei erhält das vordere Kriterium, also das in der Zeile, immer eine 2, eine 1 oder eine 0. Eine 2 bedeutet dabei, dass das Kriterium in der Zeile wichtiger ist, als sein Gegenüber, eine 1 bedeutet Gleichwertigkeit und eine 0, dass es weniger wichtig ist. Aus der Summe der für ein Kriterium vergebenen Punkte wird dann durch die Division durch die Anzahl der insgesamt vergebenen Punkte ein Modifikator für die einzelnen Kriterien ermittelt. Dabei ist der Modifikator hier zur besseren Veranschaulichung eine Zehnerpotenz höher dargestellt.

Kriterium	1	2	3	4	5	6	7	Summe	Modifikator
1		0	0	1	0	0	0	1	0,238
2	2		1	2	0	1	0	6	1,429
3	2	1		2	0	1	0	6	1,429
4	1	0	0		0	0	0	1	0,238
5	2	2	2	2		2	1	11	2,619
6	2	1	1	2	0		1	7	1,667
7	2	2	2	2	1	1		10	2,381

Tabelle 3 - Gewichtung der Kriterien mit Hilfe eines paarweisen Vergleichs

5.4.2 Evaluation

Nachfolgend werden die Frameworks anhand der im vorherigen Abschnitt beschriebenen und gewichteten Kriterien bewertet. Dabei wird pro Kriterium für jedes der Frameworks eine Note zwischen 0 und 10 vergeben, wobei 0 die schlechteste Note und 10 die Bestnote darstellt.

Kriterium 1	Lizenzkosten und benötigte Hardware	
Spark	Spark, Hadoop und Flink werden alle drei von der Apache Software Foundation als Top-Level-Projekte entwickelt und stehen daher unter der kostenlosen Apache-Lizenz zur Verfügung [42]. Da die Frameworks allerdings mit verteilter Datenverarbeitung funktionieren und somit nicht auf einem Großrechner laufen, kann durch dynamische Bereitstellung von gängiger Hardware der finanzielle Bedarf eingeschränkt werden.	
	Spark hält die zu verarbeitenden Daten im Arbeitsspeicher, wodurch der für die Ausführung des Programmes verwendete Rechen-Cluster über die der zu verarbeitenden Datenmenge entsprechende Menge an Arbeitsspeicher verfügen muss. Da die Angebote in der Cloud mit steigendem Arbeitsspeicher teurer werden, muss an dieser Stelle auf einen erhöhten finanziellen Bedarf geachtet werden. [43]	7
Hadoop	Hadoop hat im Gegensatz zu Spark keine besonderen Anforderungen an den Arbeitsspeicher des ausführende Rechen-Cluster, weshalb hier keine besondere Rücksicht auf den Arbeitsspeicher genommen werden muss. Da Hadoop jedoch auf einem verteilten Dateisystem	9

	basiert, sollte die Übertragungsrate im Cluster-Netz berücksichtigt werden [44].	
Flink	Ebenso wie Hadoop benötigt auch Flink nur die sogenannte gängige Hardware. Da jedoch auch hier viele Lese- und Schreiboperationen stattfinden, muss auch auf die Übertragungsraten im Cluster-Netz geachtet werden [45].	9
Kriterium 2	Anwendungs-Erweiterbarkeit und Einheitlichkeit des Frameworks	
Spark	Aufgrund der Einheitlichkeit von Spark und des durchgehend im Core und allen Modulen umgesetzten Konzeptes der Resilient Distributed Datasets (siehe 5.1 und 5.2) kann ein mit Spark umgesetztes Programm vielfältig erweitert und in bereits bestehende Anwendungsumgebungen eingebunden werden. So können ohne weiteren Aufwand unterschiedlichste Verarbeitungsprozesse auf denselben Daten ausgeführt werden, ohne die Daten aus dem Arbeitsspeicher entfernen zu müssen.	10
Hadoop	Hadoop basiert auf dem MapReduce Konzept und dem Hadoop Distributed File System (HDFS) [46]. Das heißt, dass Hadoop sogenannte MapReduce-Jobs auf Dateien im HDFS ausführt. Diese können als abgeschlossene Einheiten betrachtet werden, da aus einem Input, den Dateien aus HDFS, immer ein Output, wiederum Dateien im HDFS, erzeugt wird. Werden in einer Anwendung mehrere solcher Jobs ausgeführt, die voneinander abhängen, wird das Ergebnis der einzelnen Jobs also immer im HDFS zwischengespeichert und als Input für den nächsten Job verwendet. [47] Zum Kombinieren verschiedener Jobs existiert in Hadoop ein Workflow-Manager [48], mit dem die Ausführung eines Workflows, also mehrerer Jobs automatisiert werden kann.	7
Flink	Flink erweitert Hadoops MapReduce-Ansatz mit dem Dataflow-Programmiermodell um weitere Funktionalität und eine aneinandergereihete Ausführung der Operationen, wodurch verschiedenste Arbeitsschritte kombiniert bzw. erweitert werden können. [50][51]	8
Kriterium 3	Verfügbare/Unterstützte Datenformate	
Spark	Unter die von Spark unterstützten diverse fallen zum Beispiel CSV, reine Text und JSON Dateien [37]. Aber auch der Zugriff über	9

	JDBC/ODBC auf relationale Datenbanken und der Zugriff auf spaltenbasierte parquet [58] und orc [60] Dateien, sowie Vektordarstellungen über die libsvm Bibliothek [59] werden von Spark ermöglicht.	
Hadoop	Obwohl das verteilte Dateisystem, das verwendet wird, nicht auf bestimmte Datenformate beschränkt ist, ist es für die MapReduce-Jobs notwendig, dass die Dateien teilbar sind, da diese verteilt parallel bearbeitet werden sollen [47]. Als gängige Formate beschreibt Chris Deptula [49] CSV, Textdateien und JSON Records, sowie mit Avro serialisierte Dateien [49] und spaltenbasierte Dateiformate wie RC- [57], ORC- [60] und Parquet-Dateien [58]. Je nach verwendeter Distribution von Hadoop können noch weitere Datenformate unterstützt werden.	8
Flink	Flink unterstützt den Zugriff auf Dateisysteme wie Amazon S3 und HDFS, diversen Dateitypen, u.a. auch komprimierte, und dem Zugriff auf NoSQL Datenbanken, wie MongoDB. Zudem ist Flink hoch kompatibel zu Hadoop und unterstützt so alle in Hadoop verfügbaren Dateisysteme und Dateien durch Wrapper. [52]	9
Kriterium 4	Anwendungsprogrammierung API Die Bewertung der Möglichkeiten der Anwendungsprogrammierung ist aufgrund der personellen Ressourcen und den subjektiven Vorlieben und Kenntnissen des jeweiligen Programmierers schwer zu bewerten und selten objektiv.	
Spark	Spark unterstützt für den Core und alle Module eine einheitliche Programmierschnittstelle für Java, Scala, Python und R.	9
Hadoop	Grundsätzlich lassen sich dank Hadoop Streaming MapReduce-Jobs in jeder erdenklichen Programmiersprache entwickeln, solange diese über eine Standardein- und -ausgabe verfügen [61]. Bei der Verwendung von Tools aus dem Hadoop Ökosystem wird allerdings auf die entsprechende API des Tools, wie zum Beispiel die Java-API von Hive, vorausgesetzt.	7
Flink	Flink bietet für die Stapelverarbeitung von Daten eine abstrahierte Programmierschnittstelle in Java und Scala [51]. Doch auch hier ist der Programmierer auf die Schnittstellen der zusätzlichen Bibliotheken angewiesen.	7

Kriterium 5	Unterstützung relationaler Daten und relationaler Algebra	
Spark	Durch den Zugriff auf relationale Daten über die entsprechenden Treiber und die mögliche schematische Darstellung durch das DataFrame Konzept [37], wird die Bearbeitung relationaler Daten in Spark ermöglicht. Durch das Modul Spark SQL (s.a. 5.3) wird die Verarbeitung der Daten mit SQL-92 Standard und die Optimierung der Abfragen unterstützt.	9
Hadoop	Im Hadoop Ökosystem existieren sowohl für den Zugriff auf relationale Daten, als auch für die Abfragen mittels relationaler Algebra eigene Tools. Dabei ermöglicht Apache Sqoop den Zugriff auf Datenquellen mit strukturierten Daten [63] und mit Apache Hive können SQL-ähnliche Anfragen erstellt werden, wobei auch hier die Anfragen optimiert werden [64].	9
Flink	Flink unterstützt relationale Daten und die deklarative Bearbeitung dieser durch die, momentan noch in der Beta Version befindlichen, Table API und SQL Bibliothek im vollen Umfang [53].	7
Kriterium 6	Recovery und Fehlerverhalten	
Spark	Spark verfügt über ein performantes und speichereffizientes Wiederherstellungs-Konzept, durch das im Fall eines Knoten- oder Kommunikationsfehlers die fehlenden Daten über die Abstammungsinformationen schnell wieder hergestellt werden können (siehe 5.2.3).	10
Hadoop	Fehler werden in Hadoops Dateisystem über Replikation abgefangen. Fällt jedoch ein Knoten aus, auf dem ein Map oder Reduce ausgeführt wird, so werden dessen Aufgaben die nur zu lokalen Ergebnissen geführt haben und seine Aufgaben, die noch nicht beendet waren, für andere Knoten freigeschaltet [47]. Dass dabei jedoch die Daten repliziert werden müssen, wirkt sich sowohl auf den Speicherbedarf, als auch auf die Netzwerklast aus.	8
Flink	Da Flink ein Stream-orientiertes Framework ist und auch die Stapelverarbeitung intern als Stream gehandhabt wird [51], wird mit dem einfachen Ansatz gearbeitet, dass wenn ein Fehler auftritt, dass die komplette Ausführung wiederholt wird [54].	6

Kriterium 7	Art des Speichers und Performance	
Spark	Durch das Halten der zu verarbeitenden Daten im Arbeitsspeicher entfällt einiger Verwaltungsoverhead wie der buffer manager und das latching (siehe 4.3). Das wirkt sich auf die Geschwindigkeit der Lese- und Schreibeoperationen und damit auch direkt auf die Performance der Anwendung aus. Neben dem Arbeitsspeicher stellt die geringere Netzwerkauslastung einen weiteren Faktor für die Performance dar. Matei et al. stellen an verschiedenen Beispielen den Performancegewinn durch Spark dar [34].	10
Hadoop	Das verteilte Dateisystem von Hadoop, wofür entweder das bereitgestellte HDFS oder aber Speicher von Drittanbietern verwendet werden kann, speichert die Daten auf der Festplatte [46]. Neben der langsameren Geschwindigkeit bei Lese- und Schreibzugriffen erzeugt Hadoop durch die MapReduce-Jobs und die Replikation höhere Netzwerklast, wodurch die Performance weiter eingeschränkt wird.	8
Flink	Dank der Ausführung der Datenverarbeitung in einer Pipeline müssen in Flink keine Zwischenergebnisse gespeichert werden, womit nur das initiale Lesen und das Schreiben des Endergebnisses in Form von Zugriffen auf die Festplatte durchgeführt werden. Die Verarbeitung der Daten funktioniert wie bei Spark im Arbeitsspeicher, wobei in Spark die Speicher-Verwaltung von Flink übernommen wurde. [55]	10

Nach der Bewertung der einzelnen Kriterien für die Alternativen der Big-Data-Frameworks werden diese in der folgenden Tabelle mit dem Modifikator aus 5.4.1 verrechnet und somit eine Gesamtbewertung für die einzelnen Frameworks ermittelt:

- [Kriterium 1] Lizenzkosten und benötigte Hardware
- [Kriterium 2] Anwendungs-Erweiterbarkeit und Einheitlichkeit des Frameworks
- [Kriterium 3] Verfügbare/Unterstützte Datenformate
- [Kriterium 4] Anwendungsprogrammierung API
- [Kriterium 5] Unterstützung relationaler Daten und relationaler Algebra
- [Kriterium 6] Recovery und Fehlverhalten
- [Kriterium 7] Art des Speichers und Performance

Kriterium	1	2	3	4	5	6	7	Bewertung
Flink	9	8	9	6	7	6	10	8,00
Hadoop	9	7	8	6	9	8	8	8,10
Spark	7	10	9	9	9	10	10	9,50
Modifikator	0,238	1,429	1,429	0,238	2,619	1,667	2,381	Spark

Tabelle 4 - Bewertung der Frameworks mit Hilfe der gewichteten Kriterien

Anhand des Vergleichs in Tabelle 4 wird ersichtlich, dass Spark in Bezug auf die festgelegten gewichteten Kriterien die beste Bewertung erhält. Spark übertrifft dabei die Vergleichsframeworks in fast allen Kriterien. Sollte die Unterstützung der relationalen Daten in Flink die Testphase erfolgreich beenden ist Flink eine ernstzunehmende Alternative.

6 Konzept für den Prototyp

In diesem Kapitel wird die Umsetzung des Prototyps aus der konzeptionellen Sicht dargestellt. Das Konzept betrachtet dabei allgemein die Planung der Anwendung und ist nicht spezifisch für Spark. Für Spark relevante Design- und Implementierungsentscheidungen sind in Kapitel 7 dargestellt.

Dieses Kapitel stellt den ersten Teil des Schritts Konzeption und Implementierung der in Kapitel 3 vorgestellten generellen Vorgehensweise zum Umstieg auf ein Big-Data-Framework um. Die Gliederung ist in die Betrachtung der notwendigen Daten und des Workflows sowie in Grob- und Feinentwurf, welche den Aufbau der Software und die Umsetzung des Workflows darstellen, unterteilt. Abschließend wird ein grober Plan für das Deployment der neuen Software aufgezeigt.

6.1 Daten und Workflow

Bei der nachfolgenden Betrachtung liegt der Fokus auf den für die hier umgesetzten Verarbeitungen relevanten Daten. Von einer ausführlichen Erklärung der kompletten Daten des Systems wird aus Gründen der Komplexität und Relevanz für diese Arbeit abgesehen.

Die Benennung der Daten erfolgte im Originalsystem unter Konventionsvorgabe des Kunden. Zum besseren Verständnis ist diese daher in eine konkrete Benennung der Entitäten überführt worden. Zur Vereinfachung der Erklärung und aus Gründen der Übersichtlichkeit ist das Datenmodell zudem aufgeteilt worden.

Die in Kapitel 4 auf fachlicher Ebene dargestellten Daten und Workflows umfassen im Wesentlichen zwei Bereiche der Daten. Auf der einen Seite stehen dabei die Aufträge, welche durch die Primärbedarfsprognose erstellt und konfiguriert werden, auf der anderen Seite die Aktionen, welche von Systembenutzern eingetragen werden und auf den Aufträgen ausgeführt werden sollen.

6.1.1 Aufträge

Ein Auftrag stellt den Produktauftrag eines Kunden dar und ist daher neben den zugehörigen Codes, oder auch Komponenten des Produktes, über mehrere Daten zu konkretisieren. Darunter fallen die Daten der folgenden Tabellen in Abbildung 19 dargestellten:

- Vertriebseinheit
- Zeitraum
- Werk
- Baumustergruppe
- Lauf

Die Vertriebseinheit, das Sammelland und der Zeitraum definieren wo und wann der Kunde den Auftrag erteilt hat und sind eindeutig für einen Auftrag. Ein Sammelland stellt eine Zusammenfassung mehrerer Vertriebseinheiten dar. Werk und Baumustergruppe geben zusätzliche Informationen über die Herstellung, wobei das Werk wiederum eindeutig für den Auftrag

ist. Die Baumsterngruppe ist aufgrund der internen Darstellung als Hierarchie mehrfach vertreten, wodurch auch Untergruppen dargestellt werden können, und ist für bestimmte Sammelländer gültig. Der Lauf definiert die Durchführung der Prognose, in welcher der Auftrag verarbeitet wird.

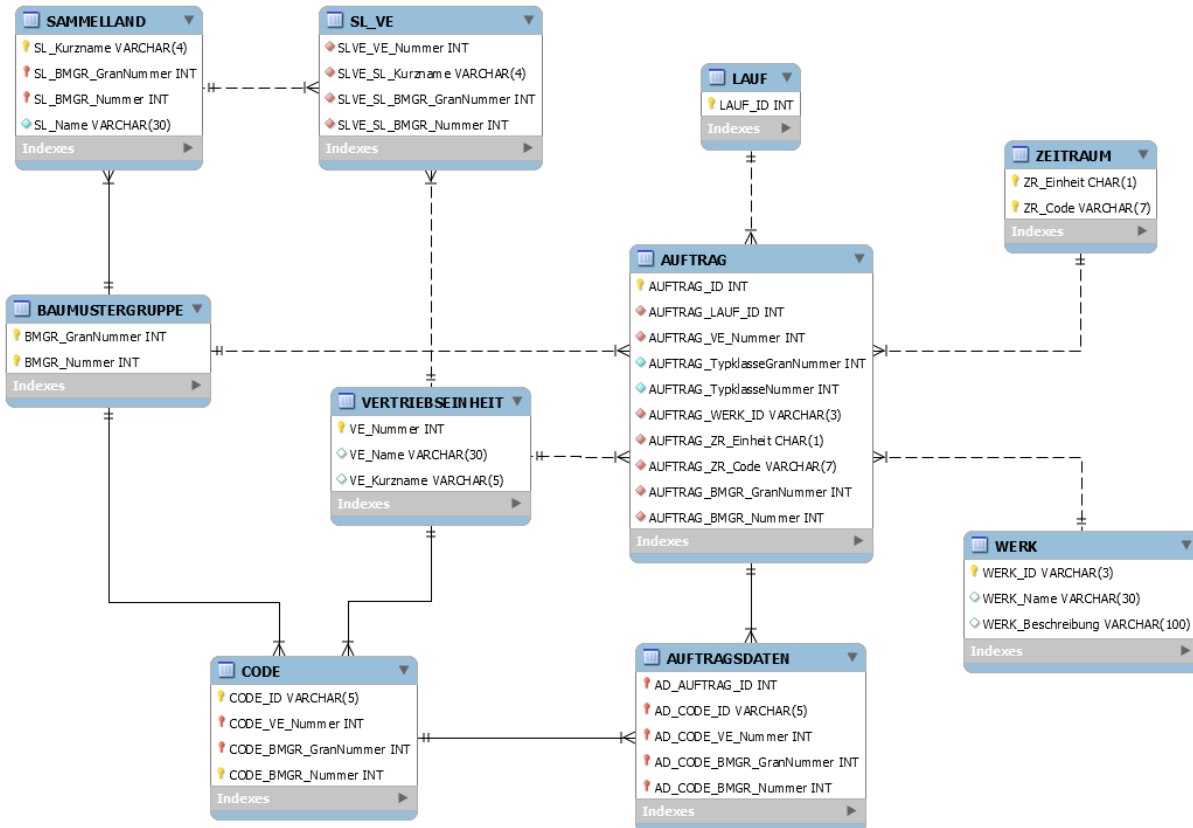


Abbildung 19 – ER-Modell für Aufträge

6.1.2 Aktionen

Im Gegensatz zu den Aufträgen umfassen Aktionen wesentlich mehr Informationen, weshalb diese zur Veranschaulichung in diesem Abschnitt von zwei Seiten betrachtet werden. Zuerst werden Aktionen hinsichtlich der Restriktionen des Geltungsbereichs betrachtet, also hinsichtlich der Daten, die einschränken auf welche Aufträge Aktionen ausgeführt werden sollen. Anschließend folgt eine Betrachtung der Aktionsdaten, die zur eigentlichen Ausführung benötigt werden.

In Abbildung 20 sind Aktionen und deren restriktive Daten dargestellt, also die Daten, die einschränken, auf welchen Aufträgen Aktionen ausgeführt werden. Bei der Betrachtung wird ersichtlich, dass die Daten derer der Aufträgen entsprechen, was erforderlich ist, da diese Daten bei der Ausführung einer Aktion herangezogen werden um die entsprechenden Aufträge zu selektieren.

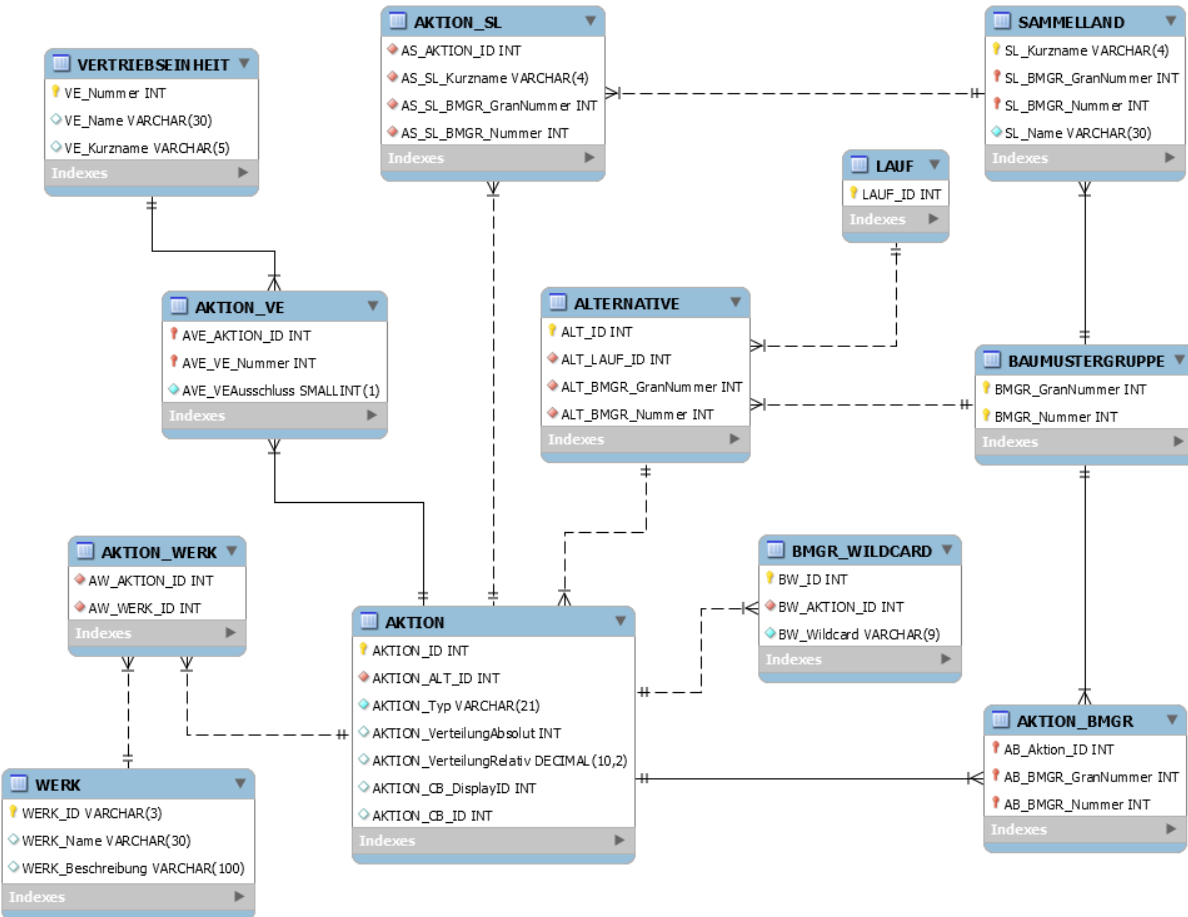


Abbildung 20 - ER-Modell der Aktionen mit restriktiven Daten

Die Aktion verfügt dabei über die Zuordnung zu Werk, Vertriebseinheiten und verschiedenen Baumustergruppen, die entweder aus der direkten Zuordnung entnommen oder über eine Wildcard aus der Tabelle BMGR_WILDCARD berechnet werden können. Neben diesen qualifizierenden Daten sind Aktionen, wie in der fachlichen Beschreibung in Abschnitt 4.2 bereits erwähnt, in Alternativen eingeordnet. In Abbildung 20 wird ersichtlich, dass einige der Daten nicht direkt mit den Aktionen in Beziehung stehen, sondern über die Alternative angebunden werden und somit für mehrere Aktionen gelten.

Zudem existieren die in Abbildung 21 dargestellten Ausführungsdaten der Aktionen, mit denen festgelegt wird, welche Aufgabe durch die Aktion durchgeführt wird.

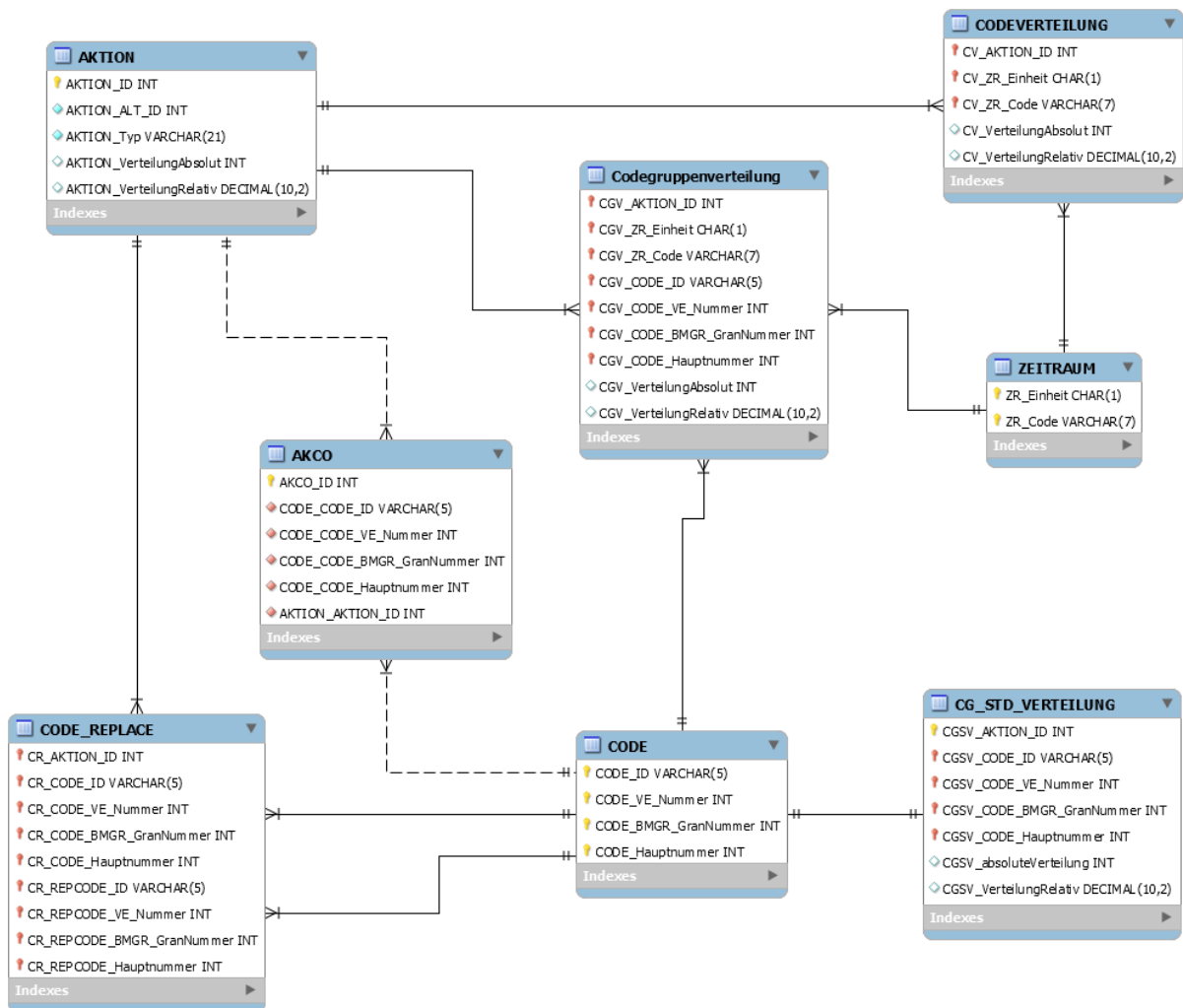


Abbildung 21 - ER-Modell mit Ausführungsdaten

Eine AKTION hat einen den Typ Codeverteilung, Codegruppenverteilung oder Codereplacement. Entsprechend dem Typ sind der Aktion Einträge aus den Tabellen CODE-DISTRIBUTION, CODEGROUPDISTRIBUTION oder CODE_REPLACE zugeordnet, die für den Aktionstyp benötigte Daten wie Verteilungswerte, Codes und Zeiträume definieren.

- **Codeverteilung:**
Für die Codeverteilung werden die Verteilungswerte für Zeiträume benötigt in denen der Code angepasst werden soll. Der Code selber ist dabei über die Tabelle AKKO direkt von der Aktion referenziert

- **Codegruppenverteilung:**
Die Codegruppenverteilung benötigt Wertungswerte pro Code und Zeitraum, weshalb die Codes hier von der Tabelle CODEGROUPDISTRIBUTION referenziert wird.
- **Codeersetzung:**
Für die Codeersetzung werden nur der Code der ersetzt werden soll und der neue Code benötigt, die beide direkt von der Tabelle CODEREPLACEMENT referenziert werden.

6.1.3 Workflow

Zur Anpassung der Datenmenge der Aufträge wird immer einer der vorhergehend erklärten Aktionstypen ausgeführt.

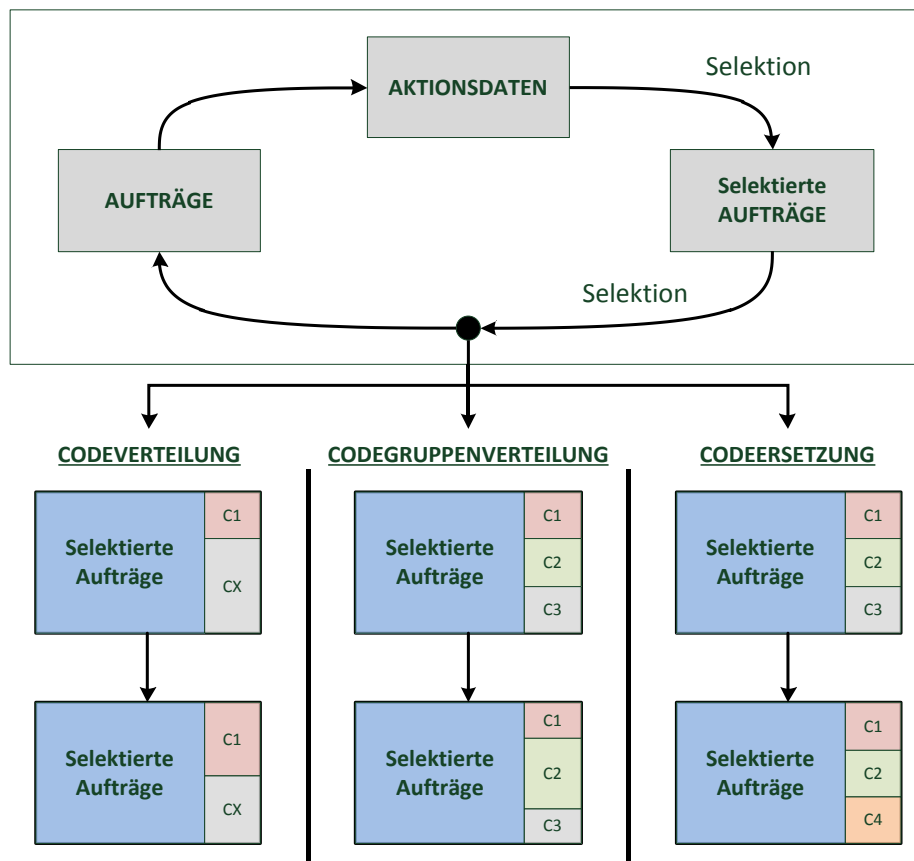


Abbildung 22 - abstrakte Darstellung der Anpassung von Aufträgen durch Aktionen

Abbildung 22 veranschaulicht die Vorgänge bei der Anpassung der Aufträge durch Aktionen mit Beispielen. Dabei wird die Menge der Aufträge anhand der im vorherigen Abschnitt beschriebenen restriktiven Daten der Aktion selektiert und die dem Typ der Aktion entsprechenden Anpassungen vorgenommen.

Bei der **Codeverteilung** wird die Anzahl der Aufträge mit dem entsprechenden Code einem Verteilungswert angepasst. Die zu verändernden Aufträge werden dabei zufällig aus den Aufträgen mit bzw. ohne entsprechenden Code ausgewählt. Am Beispiel in der Abbildung wird der Verteilungswert des Codes C1 von einem Drittel auf die Hälfte erhöht, wobei der Code zufällig bei Aufträgen hinzugefügt wird, die noch nicht über diesen verfügen, also bei CX.

Bei der **Codegruppenverteilung** hingegen werden die Aufträge gleichzeitig für eine Verteilungsvorgabe von mehreren Codes angepasst. Dafür darf für die zu verändernden Aufträge eines der Codes nicht auf die Menge der Aufträge zurückgegriffen werden, die für die Zielverteilung eines der anderen Codes bestimmt wurden. Dadurch wird sichergestellt, dass das Ergebnis der Codegruppenverteilung tatsächlich der Vorgabe entspricht. Im vorliegenden Beispiel wird die Verteilung der Codes C1, C2 und C3 auf die relative Verteilung 25 – 50 – 25 angepasst.

Letztlich werden bei der **Codeersetzung** lediglich alle Aufträge mit dem entsprechenden Code entsprechend angepasst, indem der Code durch den neuen Code ersetzt wird. Das Beispiel zeigt eine Ersetzung des Codes C3 bei allen entsprechenden Aufträgen durch den Code C4.

Die Daten einer Aktion entsprechen dabei mehreren Zeilen, in denen die Informationen für verschiedene Zeiträume bzw. Codes gespeichert sind, auf die sich die Aktion auswirkt. Der Schritt der Selektion der Daten ist also zweigeteilt. Am Anfang werden die Aufträge nach den für die Aktion allgemeingültigen Werten Baumuster, Vertriebseinheit und Werk gefiltert und erst anschließend für jede Zeile der Aktion entweder nach Zeitraum oder Code weiter reduziert. Die Anpassungen an den Aufträgen werden dann anhand des Verteilungswertes der Zeile auf dieser Datenmenge vorgenommen.

Das Ergebnis der Anpassung der Aufträge durch die Aktion wird zwischengespeichert und eine zufällig ausgewählte nächste Aktion der Alternative ausgeführt. Sobald die Alternative über keine weiteren Aktionen mehr verfügt wird eine neue Alternative abgearbeitet.

6.2 Grobentwurf

Der grobe Entwurf der Software umfasst neben dem Driver, also der Klasse die für die Koordination des Anwendungsablaufs zuständig ist, drei Bestandteile, die jeweils die Verantwortung für einen in sich geschlossenen Vorgang übernehmen. Diese sind in Abbildung 23 abstrakt dargestellt.

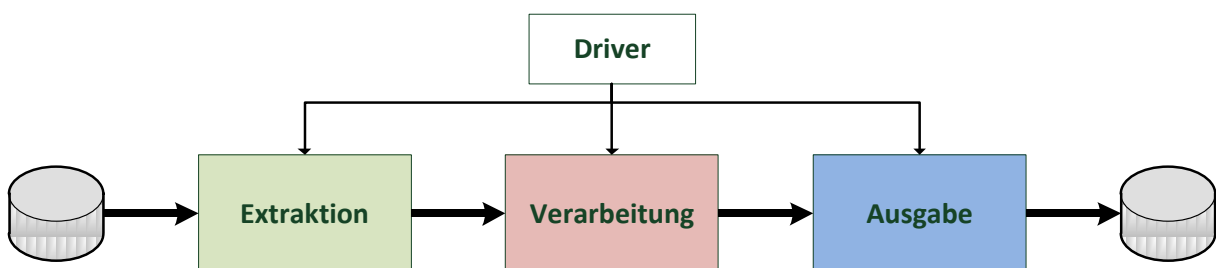


Abbildung 23 - Bestandteile und Datenfluss des Prototyps

Bei der **Extraktion** werden die relevanten Daten separiert mittels JDBC aus der Datenbank ausgelesen und als Dataframes im Arbeitsspeicher zur Verfügung gestellt. Dadurch wird sichergestellt, dass während der Verarbeitung der Aktionen nur wenige Zugriffe auf die Datenbank benötigt werden und der Arbeitsaufwand somit von der Datenbank in den Arbeitsspeicher verlagert wird.

Die Daten der einzelnen Aktionen werden während der **Verarbeitung** zuerst mit den relevanten Daten der referenzierten Tabellen verknüpft und die Aktion anschließend ausgeführt. Durch diese Vorgehensweise müssen die entsprechenden Daten erst kurz vor der Ausführung einer Aktion aus dem Arbeitsspeicher gelesen werden, anstatt mittels aufwendiger JOINS auf die Datenbank zugreifen zu müssen. Letztendlich sollen die Ergebnisse in der **Ausgabe** zurück in die Datenbank geschrieben und für Debugging und die Evaluierung in Logdateien abgelegt werden.

6.3 Feinentwurf

Der Grobentwurf der Software aus dem vorherigen Abschnitt spiegelt das Design des Prototyps wieder. Für jeden Bestandteil existiert ein Softwarepaket, das die entsprechenden Verantwortungen übernimmt. Das entsprechende Klassendiagramm ist in Abbildung 24 dargestellt.

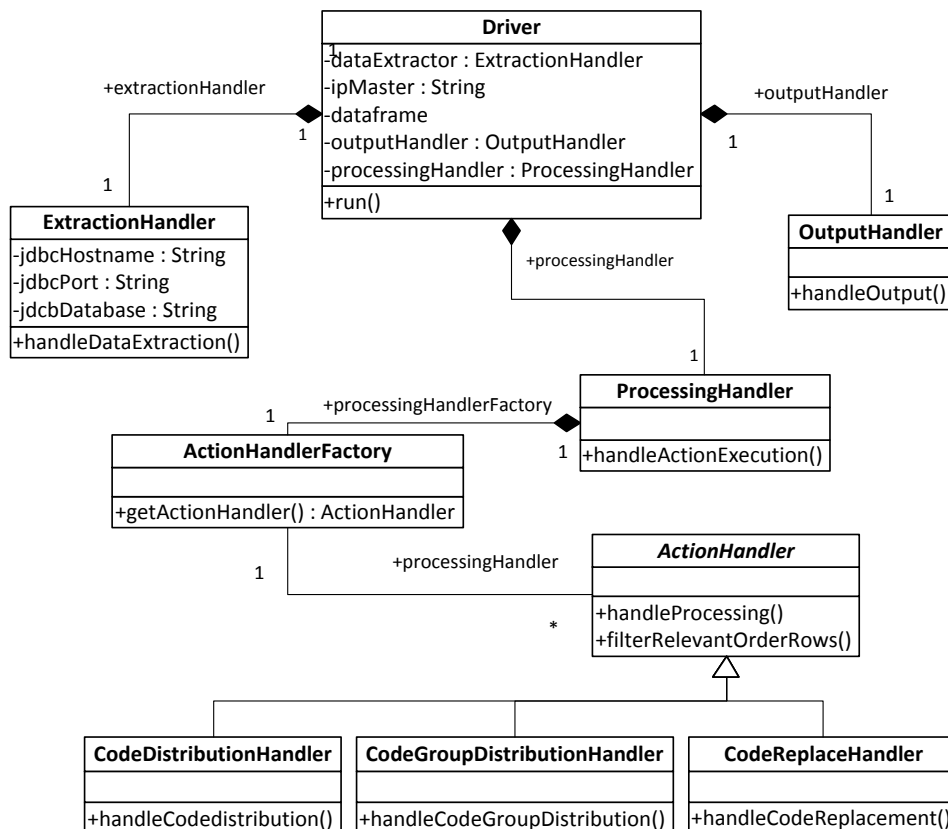


Abbildung 24 - Klassendiagramm des Prototyps

Die Driver Klasse verfügt über Instanzen der jeweiligen „Vertreterklasse“ für die Pakete, also von den Klassen `ExtractionHandler`, `ProcessingHandler` und `OutputHandler`, und kann so den Ablauf der Anwendung koordinieren.

Die Pakete `Extraktion` und `Ausgabe` haben jeweils nur ein Aufgabengebiet, nämlich den Import der Daten in die Anwendung und den abschließenden Export der Daten. Dementsprechend ist jeweils eine Klasse ausreichend um sowohl eine hohe Kohäsion, als auch eine lose Kopplung zu erreichen. Dadurch wird erreicht, dass der Prototyp für einen zukünftigen Einsatz leicht verändert bzw. erweitert und gewartet werden kann.

Der eigentliche Workflow der Anwendung wird vom Paket `Verarbeitung` verantwortet und in der Klasse `ProcessingHandler` koordiniert. Die Verarbeitung einer Aktion wird dabei von einem `ActionHandler` übernommen, der je nach Aktionstyp über unterschiedliche Implementierungen verfügt. Die Erzeugung des entsprechenden Handlers wird durch ein Factory Design [62] von der eigentlichen Verarbeitung der Aktionen abgekoppelt, wodurch gute Erweiterbarkeit und Wartbarkeit der Software gewährleistet wird.

7 Implementierung des Konzepts in Spark

In den folgenden Abschnitten wird die Umsetzung des Konzepts aus Kapitel 6 beschrieben. Die Implementierung erfolgte dabei unter Verwendung von Java 8 und der Java API von Apache Spark in der Version 2.0.

7.1 Driver

Der **Driver** ist der Einstieg in die Anwendung und ist für die Konfiguration von Spark und die Koordination der Anwendung verantwortlich. Zum Aufruf der Anwendung nimmt der **Driver** die folgenden Parameter entgegen:

- `databaseIP` – IPv4 des Datenbankhosts
- `databasePort` (optional) – Port der Datenbank (falls abweichend von 50000)
- `database` (optional) – Name der Datenbank (falls abweichend von SPDATA)

Der Ausgangspunkt einer Sparkanwendung ist der `SparkContext`, der mit einer entsprechenden Konfiguration erstellt wird.

```
SparkConf config = new SparkConf()
    .setAppName("Datamanagement Prototype Spark")
    .setMaster("spark://" + ipMaster + ":" + standardSparkPort)
    .setJars(jdbcJars);
sparkSession = SparkSession.builder().config(config).getOrCreate();
```

Die für die Ausführung der Anwendung wichtigen Konfigurationen sind der Verweis auf den Master mit `setMaster` und der zusätzliche Bezug zu externen Archiven über `setJars`, der für die Verfügbarkeit der Archive auf den Workern und somit in den Executor-Prozessen sorgt. Zudem muss für die Verwendung von SparkSQL noch eine `SparkSession` bereitgestellt werden.

```
dataExtractor.handleDataExtraction(_laufID);
processingHandler.handleActionExecution();
outputHandler.handleOutput();
```

Für die Koordination des Programflusses greift der Driver auf die Vertreterklassen der jeweiligen Pakete zurück und sichert so die korrekte Ausführung der Anwendung.

7.2 Datenextraktion

Mit der Klasse `ExtractionHandler` wird die Funktionalität bereitgestellt über SQL-Statements Dataframes zu erzeugen. Dabei werden die Statements auf der Datenbank ausgeführt und das Ergebnis vorerst als logischer Plan gespeichert. Der Zugriff auf die Datenbank wird dabei über die `ConnectionProperties` gesteuert.

```

connectionProperties.put("driver", "class.of.Driver");
connectionProperties.put("user", jdbcUsername);
connectionProperties.put("password", jdbcPassword);

connectionProperties.put("partitionColumn", "ORDR_GOPDATE_TMPR_CODE");
connectionProperties.put("lowerBound", "2016");
connectionProperties.put("upperBound", "2017123");
connectionProperties.put("numPartitions", "4");

```

In den ersten drei Zeilen werden allgemeine Eigenschaften definiert, die für alle Zugriff gleich sind. Im zweiten Teil wird die Partitionierung einer einzelnen Tabelle definiert. Die Werte der für die Partitionierung verwendete Spalte `partitionColumn` werden mithilfe von `lowerBound`, `upperBound` und `numPartitions` berechnet. Dadurch kann die Tabelle parallel auf die Workerknoten von Spark eingelesen werden.

Die Definition der eigentlichen SQL-Anfrage ist mit `jojo` [65], einer API zum typsicheren Entwurf von SQL-Anfragen, umgesetzt und kann daher auch für zukünftigen Bedarf leicht anpasst oder verändert werden.

```

Field<String> aktionID = DSL.field(DSL.name("BMW.BMW_ADJS_ID"), String.class);
Field<String> isExclusion = DSL.field(DSL.name("BMW.BMW_IS_EXCLUSION"),
    String.class);
Field<String> bmgrWildcard = DSL.field(DSL.name("BMW.BMW_WILDCARD"), String.class);

String jdbcQueryBmgrWildcards = ctx.select(aktionID.as("aktionID"),
    isExclusion.as("isExclusion"), bmgrWildcard.as("bmgrWildcard"))
    .from(DSL.table("\SPDA\."BMW").as("BMW"))
    .getSQL();
jdbcQueryBmgrWildcards = addingQueryAlias(jdbcQueryBmgrWildcards, "BmgrWildcards");

```

Die benötigten Spalten werden dabei über den Typ `Field` definiert und können in der `Select`- und `Where`-Klausel referenziert wird. Über den Aufruf von `getSQL()` wird die entsprechende SQL-Anfrage als `String` zurückgegeben. In `SparkSQL` muss bei der Erzeugung eines `Dataframe`s von externen Quellen immer ein `Alias` für die Anfrage mit angegeben werden, was mit der internen Funktion `addingQueryAlias()` umgesetzt wird.

Die eigentliche Ausführung der Anfrage und damit auch die Erstellung des logischen Plans des `Dataframe`s verbindet die SQL-Anfrage mit den `connectionProperties`.

```

Dataset<Row> dataframe = spark.read().jdbc(jdbcUrl, jdbcQueryBmgrWildcards,
    connectionProperties);

```

Die Benennung `Dataframe` ist nur ein `Alias`, da ein `Dataframe` auf Grund der internen Umsetzung nichts anderes als ein `Dataset<Row>`, also eine Datenmenge von Zeilen, ist. Dieses kann über den `DataFrameReader`, aufgerufen durch `read().jdbc()` und den entsprechenden Parametern eingelesen werden.

Über diese Vorgehensweise werden die folgenden relevanten Daten für die Anwendung eingelesen und dann an die Datenverarbeitung weitergegeben:

- `codeDistributionDF` ist eine Kombination für die Daten für eine Codeverteilung, bestehend aus der Aktion, den Verteilungswerten pro Zeitraum und dem Code der verändert werden soll.
- `codeGroupDistributionDF` ist eine Kombination für die Daten einer Codegruppenverteilung, bestehend aus der Aktion und den Verteilungswerten pro Zeitraum und Code, mit dem jeweils entsprechendem Code.
- `codeReplacementDF` ist eine Kombination für die Daten für eine Codeersetzung, bestehend aus der Aktion und den beiden benötigten Codes.
- `codeConditionDF` enthält Konditionen zum Filtern der Daten pro Aktion, bestehend aus AktionsID, dem Code für die Kondition und einem Indikator, der beschreibt ob dieser Code im Auftrag existieren muss (0) oder nicht existieren darf (1).
- `bmgrDF` enthält die Baumustergruppen.
- `bmgrWildDF` enthält Wildcard-Ausdrücke zum Bestimmen der Baumustergruppen, bestehend aus AktionsID, dem Wildcard-Ausdruck und einem Ausschluss-Indikator wie bei den Codekonditionen.
- `veDF` stellt die Zuordnung der Vertriebseinheiten zu den Aktionen dar, bestehend aus AktionsID und der ID der Vertriebseinheit.
- `codeGroupDistStdDF` enthält die Standardverteilungswerte für Codes, welche bei der Codegruppenverteilung verwendet werden.
- `jointCountryActionDF` bildet die Zuordnung der Sammelländer zu den Aktionen, bestehend aus AktionsID und identifizierenden Werten der Sammelländer.
- `jointCountrySalesUnitDF` ist die Zuordnung der Sammelländer zu den Vertriebseinheiten, bestehend aus identifizierenden Werten der Sammelländer und der ID der Vertriebseinheiten.
- `actionPlantDF` enthält die Zuordnung der Werke zu den Aktionen, bestehend aus AktionsID und der ID der Werke.

Die Auftragsdaten werden bei Bedarf gefiltert ausgelesen, da der zusätzliche Speicherbedarf der kompletten Daten sonst die Verarbeitung der Daten beeinträchtigt.

7.3 Datenverarbeitung

Da die Daten vom `extractionHandler` separiert ausgelesen werden, müssen vor der Ausführung einer Aktion erst die entsprechenden Daten zusammengestellt werden. Dafür werden aus den jeweiligen Dataframes die entsprechenden Zeilen für die Aktion extrahiert:

- Die **actionRows** sind die Zeilen der ausgelesenen Aktionsdaten für die entsprechende Aktion. Dies können auch mehrere Zeilen sein, da die Datenmenge zum Beispiel bei der Codegruppenverteilung für jeden Code einen Eintrag enthält.
- Hinter den **codeConditionData** verstecken sich Konditionen in DNF, die wiederum qualifizierend für die anzupassenden Aufträge wirken. Hier werden Codes definiert,

die der Auftrag enthalten muss oder nicht enthalten darf um relevant für die Aktionsausführung zu sein.

- Die Liste **vePerAction** stellt die Vertriebseinheiten dar, auf welche die Aktion angewendet werden soll.
- In **bmgrPerAction** werden alle Baumustergruppen angegeben, auf welche die Aktion angewendet werden soll. Diese werden mit Hilfe der Baumustergruppen-Wildcard, die für Aktionen definiert sind, aus allen Baumustergruppen ausgelesen.
- In der Liste **plantsPerAction** sind die Werke enthalten, auf welche die Aktion angewendet werden soll.
- In **stdDistPerCGV** werden die Zeilen mit den Standardverteilungswerten für die Codes bei einer Codegruppenverteilung bereitgestellt.

Damit die zusammengestellten Daten richtig verarbeitet werden können, wird im nächsten Schritt der passende `ActionHandler` über die Fabrik erstellt und die Daten zusammen mit der `sparkSession` und dem Datenbankhost für den Zugriff auf die Datenbank an diesen übergeben.

```
ActionHandler handler = factory.getProcessingHandler(actionType);
if (handler instanceof CodeGroupDistributionHandler) {
    handler.handleProcessing(actionRows, codeConditionData, vePerAction,
        bmgrPerAction, plantsPerAction, stdDistPerCGV,
        sparkSession, databaseIP);
} else if (handler instanceof CodeDistributionHandler || handler instanceof
CodeReplaceHandler) {
    handler.handleProcessing(actionRows, codeConditionData, vePerAction,
        bmgrPerAction, plantsPerAction, sparkSession, databaseIP);
}
```

Die jeweiligen Handler für die unterschiedlichen Aktionstypen implementieren dabei die Methode `handleProcessing()` für die Verarbeitung der Daten. Dazu sind zwei weitere vorgelegte Schritte notwendig. Einerseits müssen für das Berechnen des Verteilungswertes, das bei der Ersetzung von Codes entfällt, mehrere Spalten der Daten betrachtet werden, andererseits muss die Datenmenge der Aufträge gefiltert werden um die Einträge zu finden, auf die die Aktion angewendet werden soll.

Die Verteilungswerte sind in vier Tabellen definiert, in `AKTION`, `CODEVERTEILUNG`, `CODEGRUPPENVERTEILUNG` und `CG_STD_VERTEILUNG`. Je nach Aktionstyp erfolgt die korrekte Bestimmung des Verteilungswertes auf den Daten dieser vier Tabellen wie in Abbildung 25 dargestellt.

Für eine Codeverteilung gilt immer der Verteilungswert der in der Tabelle `CODEVERTEILUNG` für jeden Zeitraum definiert ist. Existiert dieser nicht wird der Standardwert der Aktion herangezogen. Diese Werte werden auch für die Codegruppenverteilung verwendet, sagen dabei aber nur etwas über die Gültigkeit der Aktion aus. Sofern einer der Werte vorhanden ist, die Aktion also gültig ist, wird der Verteilungswert für die Codes der Codegruppenverteilung auf die gleiche Art wie bei der Codeverteilung bestimmt. Gibt es einen definierten Wert in der Tabelle `CODEGRUPPENVERTEILUNG` für die Codes, so wird dieser verwendet, existiert

der Wert jedoch nicht, wird der Standardwert für den Code aus der Tabelle CG_STD_VERTEILUNG ausgelesen. Für den Fall, dass auch dieser Wert nicht vorliegt ist die Aktion wiederum ungültig.

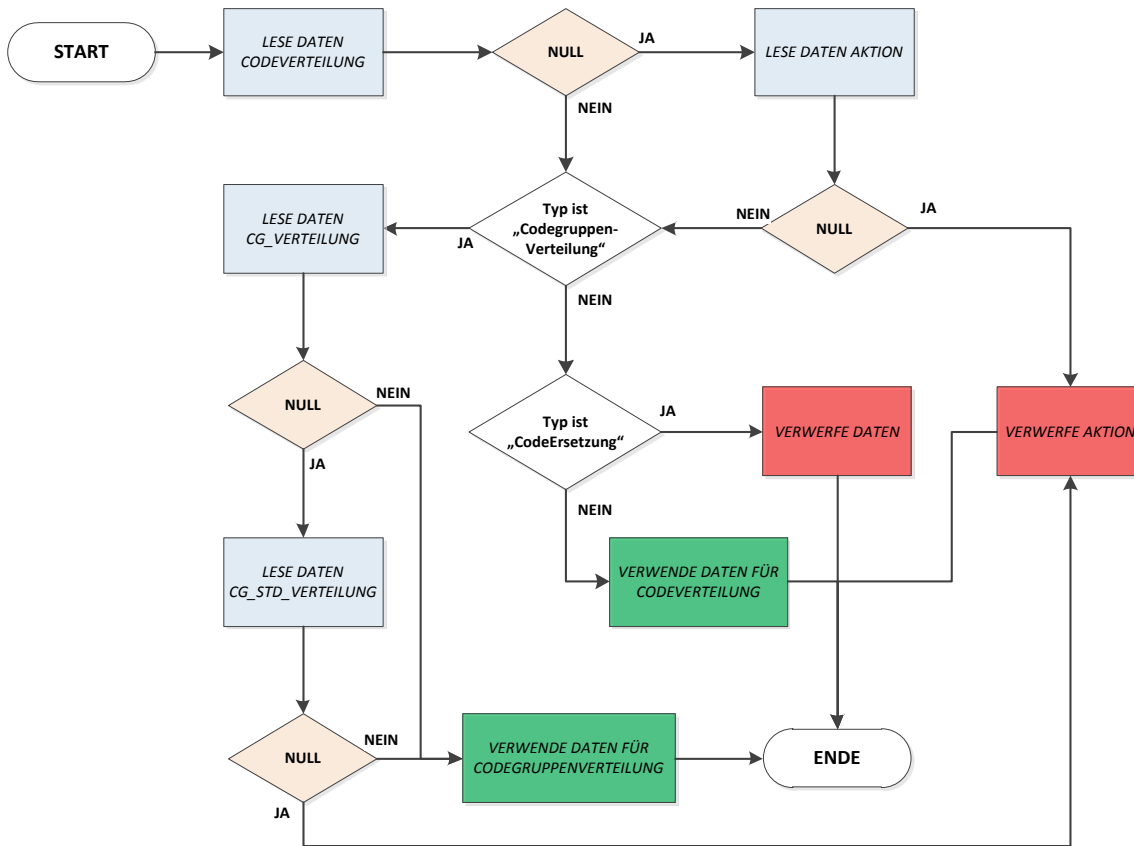


Abbildung 25 - Bestimmung des Verteilungswertes für Aktionen

Um die Aktion nur auf die richtigen Aufträge auszuführen, muss die Datenmenge der Aufträge selektiert werden. Wie im Konzept in Abschnitt 6.1.3 bereits beschrieben werden die Auftragsdaten in zwei Schritten selektiert. Der erste ist das Auslesen der allgemein für die Aktionswerte gültigen Aufträge über JDBC wie bei der Datenextraktion in Abschnitt 7.2 Dafür werden die Baumsterngruppen, Vertriebseinheiten, Werke und Codekonditionen der Aktion herangezogen und das entsprechenden Dataframe der Auftragsdaten erstellt. Anschließend wird dieses nach dem Zeitraum, bzw. den Codes der Aufträge partitioniert, da im zweiten Schritt bei Codeverteilung und Codegruppenverteilung nach Zeitraum und bei Codeersetzung nach Codes gefiltert wird.

```

filteredOrderData.repartition(filteredOrderData.col("ZRCODE"));
relevantOrderData = filteredOrderData
    .filter(filteredOrderData.col("ZRCODE").contains(timeCode));
filteredOrderData.unpersist();
  
```

Nach Ausführung des Verarbeitens der Daten wird das erstellte Dataframe mit der Methode

`unpersist()` wieder aus dem Speicher genommen um den Platz für das Dataframe des nächsten Zeitraums frei zu machen.

Die übrig gebliebenen Aufträge stellen die Menge dar, auf welche die Aktion ausgeführt werden soll.

```
handleDistributionPerTimecode(distribution, filteredOrderData);
```

Dabei wird berechnet, welche der selektierten Aufträge verändert werden müssen um den vorgegebenen Verteilungswert zu erreichen, oder einen Code in allen selektierten Aufträgen zu ersetzen. Die dadurch berechneten Anpassungen werden abschließend in einem `AdjustmentContainer` zwischengespeichert, damit sie am Ende ausgegeben werden können.

```
ordersToAdapt = new AdjustmentContainer(aktionID, alternativeID);
ordersToAdapt.setContainerType(EContainerType.distribution);

while (orderIDs.toLocalIterator().hasNext()) {
    Row row = orderIDs.toLocalIterator().next();
    // store rows for adaption with this adjustmentID
    ordersToAdapt.addAdjustment(row.getInt(0), code, adjsType);
}
```

Der Container kann den Typ `distribution`, `group` oder `replace` annehmen und kann dadurch auf einen Aktionstyp ausgerichtet werden. Die Anpassungswerte besteht aus der ID des Auftrags, dem Code der verändert wird und dem Anpassungstyp `adjType`, der `insert`, `delete` oder `update` entspricht.

7.4 Datenausgabe

Die Datenausgabe erfolgt aus verschiedenen Gründen nur als reines Logging der Anpassungen. Zum einen bleibt die Datenmenge so unverändert und die Anwendung kann beliebig oft laufen, ohne verfälschte Ergebnisse durch geänderte Datensätze zu erhalten. Zum anderen ist das Schreiben der Änderungen in die Datenbank nicht die relevante Größe um die es bei dieser Arbeit und der damit verbundenen Evaluierung geht. Natürlich spielt der Zugriff auf die Datenbank durch das Schreiben der Änderungen eine Rolle in der Auswertung der Performance, was jedoch aufgrund der erreichten Performanceergebnisse irrelevant wird.

```
<Alternative><Anzahl Aktionen>(<Anzahl Inserts><Anzahl Deletes><Anzahl Updates>)
<Ausführungszeit><Vorbereitungszeit>
```

Der `OutputHandler` schreibt die Daten nach dem hier dargestellten Schema auf die Konsole.

7.5 Deployment

Der Prototyp ist im Zuge der Tests und der Evaluierung auf einem Amazon-Webservice-Cluster bereitgestellt worden. Für die Ausführung der Anwendung wurden Spark 2.0 und Java 8 verwendet.

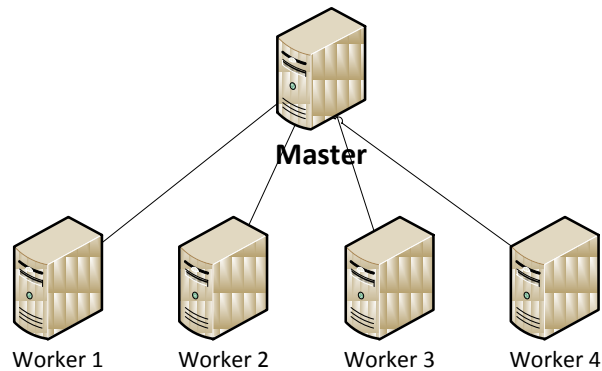


Abbildung 26 - AWS-Cluster für die Sparkanwendung

Der in Abbildung 26 dargestellte Cluster besteht aus fünf m1.xlarge ec2 Instanz für den Master und vier Worker. Die Eigenschaften des Instanztypen sind in Tabelle 1 dargestellt. Die Bereitstellung des Clusters erfolgte dabei über das spark-ec2 Skript [67], das ebenso wie Spark ursprünglich an der Universität Berkeley entwickelt wurde.

Instanztyp	vCPU	Arbeitsspeicher	Instanzspeicher	Netzwerkleistung
m1.xlarge	4	15	4 x 420	Hoch

Tabelle 5 - Eigenschaften ec2-Instanz m1.xlarge [66]

Das Skript sorgt für das Aufsetzen der Maschinen mit erforderlichen Installationen und richtet das Cluster für die Verwendung als Sparkcluster ein. Die Konfigurationen werden in der spark-defaults.conf und spark-env.sh definiert.

- spark.executor.memory definiert den verfügbaren Arbeitsspeicher der Worker
- spark.executor.extraClassPath definiert das Verzeichnis der Archive für die Worker
- SPARK_WORKER_CORES definiert die verfügbaren CPU-Kerne der Worker
- SPARK_MASTER_IP definiert die IP des Masters, mit denen sich die Worker verbinden sollen

Abhängig davon müssen jedoch noch einige Einstellungen vorgenommen werden, die bei Spark durch Parameter beim Aufruf der Anwendung über die Klasse spark-submit festgelegt werden. Hierbei übergebene Konfigurationen werden bevorzugt behandelt und überschreiben damit Werte aus den Konfigurationsdateien.

```
~/spark/bin/spark-submit
--class=corp.capgemini.masterthesis.baessler.Driver
--driver-class-path "root/spark-prototype/driver/*"
--driver-memory=13g --conf spark.executor-memory=13g
--conf spark.sql.warehouse.dir=file:///root/spark-prototype/spark-warehouse/
~/spark-prototype/SparkPrototype-1.0-SNAPSHOT-jar-with-dependencies.jar "ip"
```

Die erste Konfiguration gibt an, welche Klasse die main-Methode enthält. In der zweiten wird der Pfad zu zusätzlichen Archiven definiert, bei dem hier die JDBC-Treiber angegeben sind. In der dritten Zeile werden Speicherressourcen für Driver- und Executorprozesse festgelegt, während die vierte Konfiguration den Pfad übergibt, wohin die Daten gespeichert werden sollen, falls Spark Daten auf die Festplatte persistieren muss. Zuletzt wird das ausführbare Java-Archiv mit den entsprechenden Parametern übergeben.

8 Evaluierung

8.1 Evaluierung der Vorgehensweise

Die in Kapitel 3 vorgestellte Vorgehensweise zur Migration des Datenmanagements in ein Big-Data-Framework ist während der Entwicklung der in dieser Arbeit vorgestellten Lösung Schritt für Schritt umgesetzt worden. Dabei wurde ausschließlich der zweite Schritt weggelassen, da mit dieser Arbeit der mögliche Umstieg für die vorliegende Problematik evaluiert und nicht der eigentliche Umstieg durchgeführt wurde.

In dieser Arbeit hat die Vorgehensweise gut funktioniert, obwohl ein Schritt ausgelassen wurde. Die in Kapitel 4 und 6 umgesetzte Ist-Analyse bestätigt sich dabei als wichtige Grundlage für die nachfolgenden Schritte, stellte sich aber bei einem komplexen Datenmodell der zu migrierenden Anwendung als anspruchsvolle Aufgabe heraus.

Der dritte Schritt ist in dieser Arbeit in Abschnitt 5.4 umgesetzt. Die Auswahl des Big-Data-Frameworks durch das Aufstellen und das Gewichten von Kriterien mit der Methode des Paarweisen Vergleichs funktionierte sehr gut um eine fundierte Entscheidung über das passende Framework treffen zu können.

Der konzeptionelle Entwurf der Anwendung in Kapitel 6 war schnell erarbeitet, hätte aber bei vorliegendem Konzept der zu migrierenden Anwendung entfallen können. Die anschließende Implementierung stellte sich als Herausforderung dar, was an der Anwendung selbst liegt.

Die im folgenden Abschnitt dargestellte Evaluierung der Software entsprach nicht den Erwartungen, da durch das Ausfallen des Schrittes der Reflexion nicht überprüft wurde, inwieweit die Migration des Datenmanagements der Anwendung auf ein Big-Data-Framework sinnvoll ist. Die Ergebnisse der Evaluierung sind allerdings wiederum in die Vorgehensweise mit eingeflossen und haben durch identifizierten Probleme die Indikatoren mit geprägt.

8.2 Evaluierung der implementierten Lösung

In den folgenden Abschnitten soll der in dieser Arbeit entworfene Prototyp evaluiert werden und mit der bisherigen Software verglichen werden. Dabei soll diese Evaluierung nur eine Tendenz darstellen zum Verhalten der in Spark implementierten Lösung gegenüber dem Originalsystem hinsichtlich der Performance darstellen. Da das Originalsystem stark optimiert ist und der Prototyp eben nur ein Prototyp und nicht eine optimierte Umsetzung in Spark darstellt, wird hier nur eine Tendenz des Verhaltens ermittelt und keine endgültige Aussage über das Performanceverhalten von der in Spark implementierten Lösung gegenüber dem Originalsystem getroffen.

Die Performance des Prototyps soll dabei in den folgenden Bereichen evaluiert werden:

- **Datenextraktion**

Für die Datenextraktion des Prototyps sind die Messwerte für alle extrahierten relevanten Daten aufgeführt.

- **Laufzeit einzelner Alternativen**

Für die Messwerte sind hier beispielhaft 3 Alternativen ausgewählt. Dabei werden in beiden Systemen, also im Prototyp und im originalen System die Messwerte für diese 3 Alternativen ermittelt.

Beim Originalsystem wird auch die Laufzeit einzelner Alternativen gemessen. Die Datenextraktion entfällt hier, da diese nicht im Vorfeld sondern während der Abarbeitung der einzelnen Aktionen in den Alternativen stattfindet.

8.2.1 Testaufbau

Der Testaufbau folgt dem Plan zum Deployment aus Kapitel 0. Als Datenbank kann leider nicht die Datenbank des Originalsystems herangezogen werden, weil der Zugriff aus der Cloud auf das Unternehmensnetzwerk hier eine unüberwindbare Hürde darstellt. Obwohl das die Vergleichbarkeit schmälert muss also eine lokale Datenbank auf einem Laptop eingerichtet werden, auf die der Prototyp zugreift.

Für die Performancemessung des Prototyps wurden Mechanismen zur Zeitmessung in den Quellcode mit eingebaut, von denen nachfolgend die Datenextraktion beispielhaft dargestellt ist:

```
StopWatch dataExtractorTimer = new Stopwatch();
dataExtractorTimer.start();
// extract the relevant data here ...
dataExtractorTimer.stop();
Logger.log(Level.DEBUG, "Execution time for extraction of relevant data: " +
    dataExtractorTimer.getTime() + " ms\n");
```

Für die Messwerte des Originalsystems sind die entsprechenden Logdateien und in der Datenbank gespeicherte Laufzeitwerte der Integrationsumgebung ausgewertet worden.

8.2.2 Evaluierung der Datenextraktion

Die Datenextraktion stellt im Prototyp einen vorgelagerten Schritt dar um die Zugriffe auf die Datenbank während der Ausführung der Aktionen auf ein Minimum zu reduzieren. Abbildung 27 zeigt die Messwerte der Datenextraktion des Prototyps.

Die Messung umfasst die Zeit, die der Prototyp braucht, um die Dataframes anhand der SQL-Anfragen zu erstellen. In der ersten Messung wurde das Dataframe nach der Erstellung des logischen Plans gecached, bei der zweiten Messung mit show() die ersten 20 Zeilen des Dataframes angezeigt.

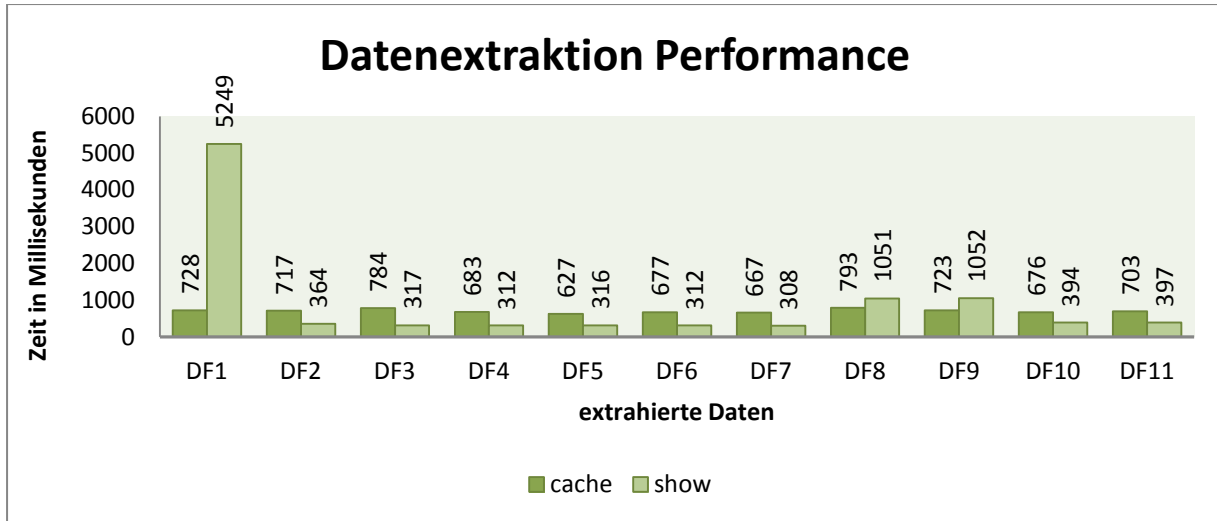


Abbildung 27 - Performanceübersicht Datenextraktion Prototyp

Durch beide Methoden wird sichergestellt, dass die Daten im Speicher des Clusters vorliegen. Um die Daten mit der Methode `show()` anzeigen zu können muss Spark das Dataframe einlesen und die Daten ausgeben. Die Methode `cache()` hingegen sorgt dafür, dass das Dataframe komprimiert im Speicher abgelegt wird. Der größte Unterschied zwischen den beiden Methoden wird bei Dataframe DF1 in Abbildung 27 ersichtlich. Das Auslesen von DF1 ist die datenintensivste Anwendung bei dieser Aufstellung, weshalb das Anzeigen hier wesentlich mehr Zeit ein Anspruch nimmt. Dies gilt speziell dann, wenn Spark das Dataframe parallelisiert auf die Worker eingelesen hat, da für die Methode `show()` die Daten auf einem Knoten gesammelt werden.

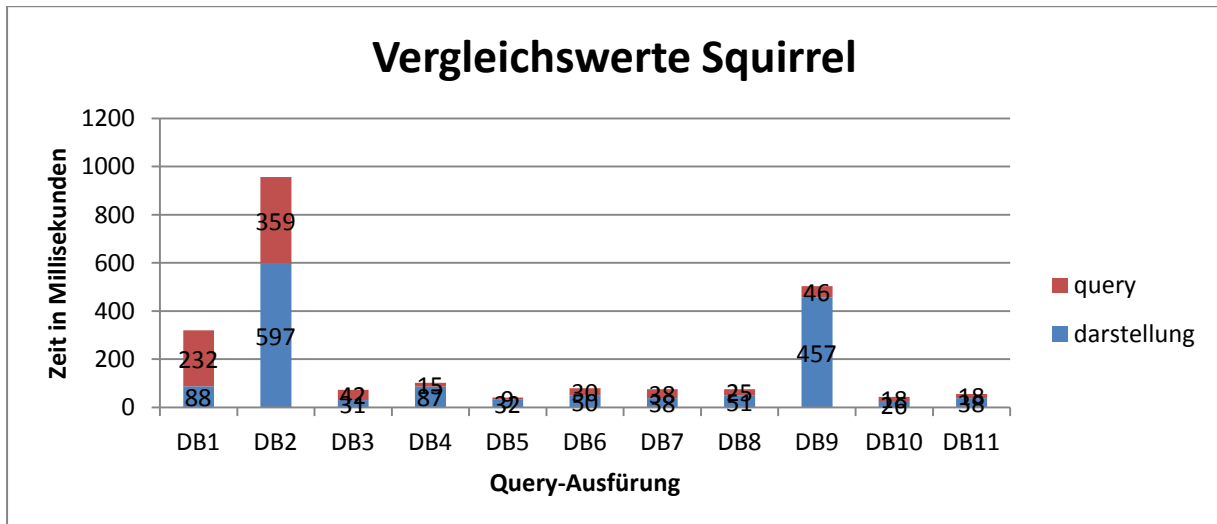


Abbildung 28 - Datenextraktion - Vergleichswerte Squirrel

Um die Performance des Auslesens der Daten interpretieren zu können sind in Abbildung 28 Vergleichswerte angegeben, die für dieselben SQL-Anfragen mit dem SQL-Client Squirrel auf

dem gleichen Rechner wie die Datenbank ausgeführt wurden. Unter Beachtung des Testaufbaus und der unterschiedlichen Ausführungsumgebungen fällt beim Vergleich der Daten des Prototyps und der Daten von Squirrel auf, dass die Vergleichswerte der lokalen Ausführung für die weniger datenintensiven und somit auch einfacheren Anfragen fast durchgehend um ein vielfaches niedriger sind. Eine Komponente die Einfluss auf diesen großen Unterschied hat ist ohne Zweifel die des Netzwerkzugriffs. Da der Laptop mit der Datenbank über eine handelsübliche Internetleitung angebunden ist und mit 4 Gigabyte Arbeitsspeicher und einem i5 Dualcore auch nicht hochklassig ist, bleibt die Verwunderung über diesen Unterschied aus.

Zur besseren Darstellung sind die Messungen des Auslesens der Dataframes von Aufträgen und Auftragsdaten hier separat in Abbildung 29 und Abbildung 30 dargestellt. Das Auslesen dieser beiden Dataframes ist mit 2 Millionen und über 200 Millionen Einträgen mit Abstand die datenintensivste Aufgabe. Zudem sind die entsprechenden Anfragen verschachtelt und extrahieren mehr Spalten gegenüber den zuvor dargestellten Dataframes.

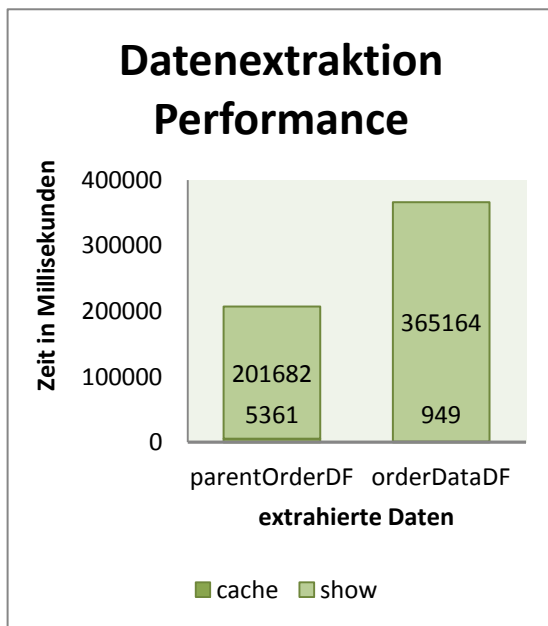


Abbildung 29 - Performance Datenextraktion Auftragsdaten

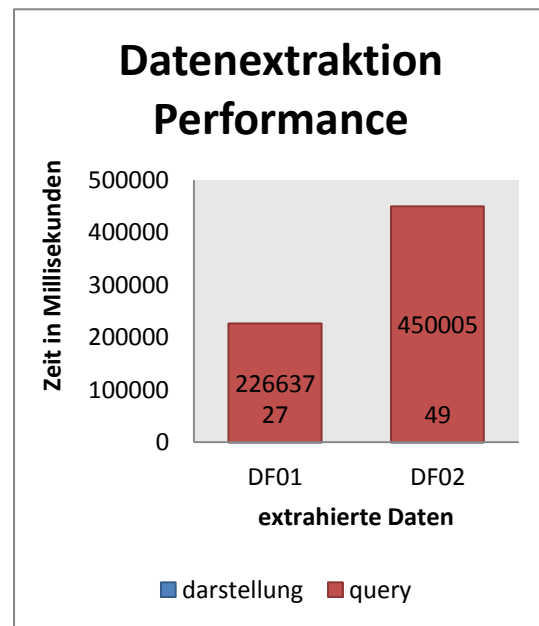


Abbildung 30 - Datenextraktion - Vergleichswerte Squirrel

Beim Vergleich der Messwerte fällt auf, dass bei diesen Anfragen der Prototyp schneller war. Dies erklärt sich dadurch, dass die Anfragen hier durch paralleles Einlesen von mehreren Workerknoten ausgeführt wurden, wodurch die Zeit für das reine Auslesen und Ablegen in den Arbeitsspeicher mit der Methode cache() in der Darstellung fast nicht mehr zu erkennen ist.

Da die Messung der Datenextraktion nur im Prototyp repräsentiert ist können diese Werte nicht direkt verglichen werden, sondern fließen mit in die Evaluierung der Laufzeit einzelner Alternativen ein. Die Extraktion der relevanten Daten stellt im Prototyp einen vorgelagerten Schritt

dar, der die direkten Zugriffe auf die Datenbank während der Ausführung von Aktionen zusammenfasst. Durch die Speicherung der Daten im Arbeitsspeicher können Anfragen somit im wesentlich schnelleren Arbeitsspeicher bearbeitet und permanente Zugriffe auf die Datenbank verhindert werden.

$$\text{Mehraufwand pro Alternative} = \frac{\text{Laufzeit Datenextraktion}}{\text{Anzahl an Alternativen}}$$

Die Laufzeit der Datenextraktion stellt damit einen Overhead dar, der zum direkten Vergleich der Laufzeit der Alternativen beider Systeme auf die Messwerte des Prototyps aufgerechnet werden muss. Dabei wird ein durchschnittlicher Wert, wie oberhalb berechnet, auf die Laufzeit der einzelnen Alternativen summiert. Wird man jedoch die Anzahl der Alternativen beachtet, so liegt dieser Overhead im niedrigen Millisekundenbereich

8.2.3 Evaluierung der Datenverarbeitung

Für die Evaluierung der Datenverarbeitung wird die Laufzeit einzelner Alternativen gemessen und mit Werten des Originalsystems verglichen. Durch die Messung der Laufzeit einzelner Alternativen lässt sich so ermitteln, ob die Verarbeitung der Datenmenge im Arbeitsspeicher einen Unterschied zur Verarbeitung mit Zugriff auf die Datenbank bedeutet.

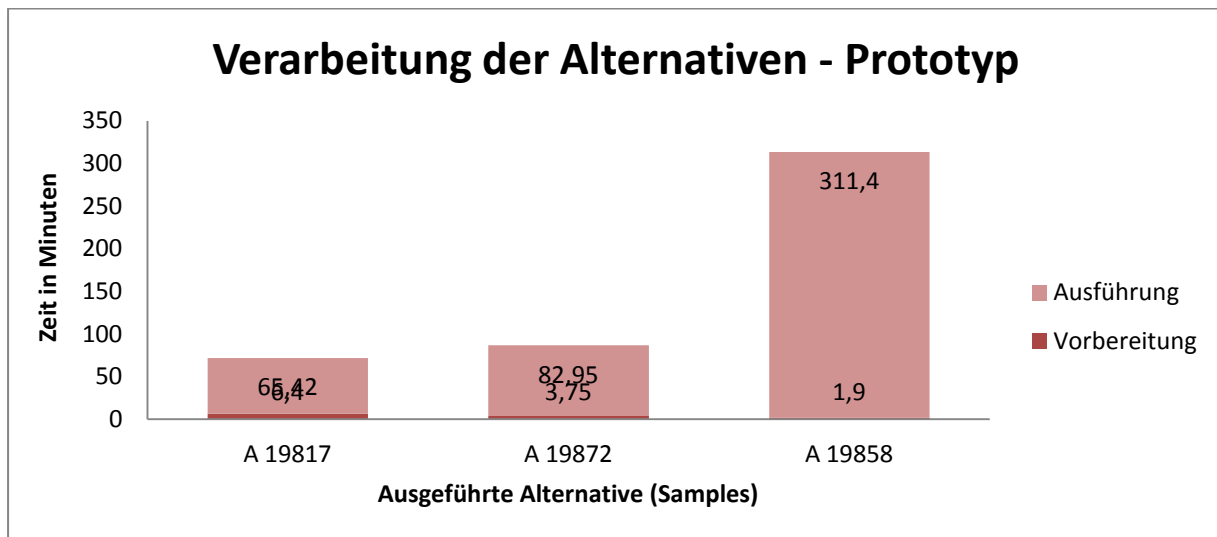


Abbildung 31 – Performanceergebnisse der Alternativen-Verarbeitung Prototyp

Die Messung in Abbildung 31 wurde an 3 Alternativen ausgeführt, die bei der Ausführung vom Prototyp ohne Unterbrechung durchgeführt wurden. Dabei sind in den Alternativen die Aktionstypen Codeverteilung und Codegruppenverteilung enthalten. Die Codeersetzung kam leider in keiner dieser Alternativen vor. Die Werte verdeutlichen die Performanceprobleme der Alternativenausführung. Etwa 80% der Zeit der Ausführung wird vom Selektieren der Auftragsdaten in Anspruch genommen.

Die Messung der einzelnen Alternativen des Originalsystem in Abbildung 32 ist nicht direkt mit der Messung des Prototyps vergleichbar, da in diesem Fall die Zugriffe auf die Datenbank während der Verarbeitung der Aktionen durchgeführt werden.

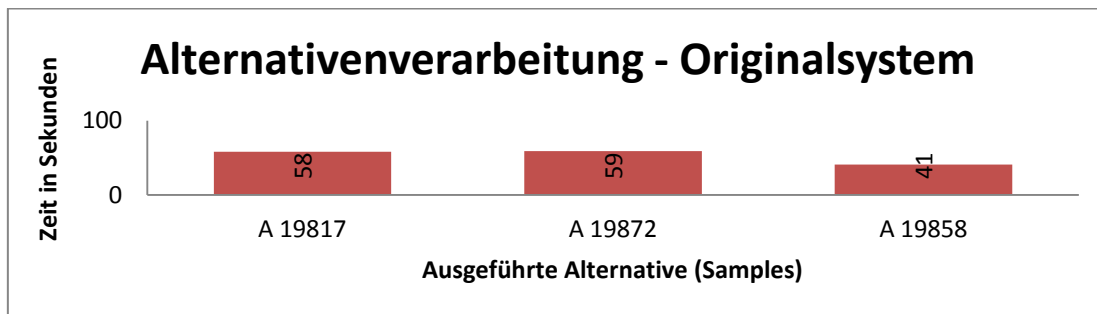


Abbildung 32 – Performanceergebnisse der Alternativen-Verarbeitung Originalsystem

Bei Betrachtung der Ausführungswerte des Prototyps in Minuten und des Originalsystems in Sekunden wird klar, dass der Prototyp der in der Zeit dieser Arbeit entwickelt wurde hinsichtlich der Performance klar unterliegt. Eine um ein Vielfaches höhere Bearbeitungszeit der Alternativen wurde zu Beginn der Arbeit wahrlich nicht erwartet. In diesem Fall ist jedoch der Vergleich der Laufzeiten für die Bearbeitung der Alternativen nicht aussagekräftig. Vielmehr liegt hier der Fokus auf der Frage, warum die Performance der Anwendung in Spark um so ein Vielfaches schlechter ist als im Originalsystem. Um diese Frage zu beantworten werden nachfolgend Aspekte mit Auswirkung auf die Performance des Prototyps bzw. den Performanceunterschied betrachtet.

Ablauf der Datenzugriffe, Datenabhängigkeiten und fehlende Parallelisierung

Wie bereits im Schritt der Reflexion in Abschnitt 3.2 erwähnt gibt es einige Indikatoren dafür, dass die Migration des Datenmanagements einer Anwendung in ein Big-Data-Framework nicht sinnvoll sein kann. Einer der Indikatoren besagt, dass eine hohe Abhängigkeit der Daten untereinander die Parallelisierung beeinträchtigt.

Der in Spark umgesetzte Prototyp verfügt über wenig Parallelisierung, die hauptsächlich im Bereich der Speicherung der Daten stattfindet. Im Verlauf der Entwicklung wurde immer wieder versucht auch im Bereich der Datenzugriffe und der Verarbeitung der Daten Parallelisierung zu schaffen, was jedoch nicht gelang. Spark stellt zur parallelisierten Verarbeitung von Daten zwei Methoden zur Verfügung.

```
dataframe.foreach(new ForeachFunction<Row>() {  
    @Override  
    public void call(Row arg0) throws Exception {  
        // TODO Auto-generated method stub  
    }  
});
```

in der `foreach()` Methode können Daten ausgezeichnet sequentiell verarbeitet werden, da über die zu implementierende Methode `call()` der Zugriff auf alle Zeilen des Dataframes

ermöglicht wird. Im Prototyp wurde versucht die Methode zu nutzen um für alle Zeilen einer Aktion die Verarbeitung der Auftragsdaten umzusetzen. Dies ist jedoch in Spark nicht möglich, da die `ForEachFunction` keine verschachtelten Dataframes zulässt.

In Folge dessen wurde versucht die einzelnen Aktionsreihen auf die nach Zeitraum partitionierten Auftragsdaten abzustimmen und so jeweils die auszuführende Aktionszeile auf den Knoten der benötigten Auftragsdatenpartition abzulegen. Diese Vorgehensweise lässt sich durch die Partitionierung über den gleichen Schlüssel realisieren. Da die `ForEachPartitionFunction` den Zugriff auf die Partition aber nur über einen Zeileniterator ermöglicht, müsste die komplette Partition durch sequentielles Filtern der Zeilen stattfinden.

```
dataframe.foreachPartition(new ForEachPartitionFunction<Row>() {  
    @Override  
    public void call(Iterator<Row> arg0) throws Exception {  
        // TODO Auto-generated method stub  
    }  
});
```

Hier kommt ein weiterer Indikator ins Spiel, der die Abhängigkeit der einzelnen Schritte der Datenverarbeitung untereinander betrifft. Durch die Notwendigkeit des Resultats der Filterung als Datenset, welches in dieser Funktion nicht erzeugt werden kann, ist auch die `ForEachPartitionFunction` für die Parallelisierung der Datenverarbeitung im Prototyp unbrauchbar.

An diesen zwei Beispielen wird ersichtlich, dass die Daten und die Art, wie auf diese zugegriffen wird, die Parallelisierung nicht nur wie im Indikator bereits beschrieben beeinträchtigen, sondern sogar verhindern kann. Ohne Parallelisierung der Datenverarbeitung kann die Performance allerdings nicht auf den theoretisch möglichen Wert von

$$\frac{1}{\text{Anzahl Worker}} * \text{Performance vor Parallelisierung}$$

verbessert werden.

Speichermanagement

Beim Datenmanagement einer datenintensiven Anwendung im Arbeitsspeicher muss besonders auf das Speichermanagement der Anwendung geachtet werden. Sobald der verfügbare Speicher für die gespeicherten Daten nicht mehr ausreicht gibt es zwei Möglichkeiten wie die darunterliegende Ausführungsumgebung damit umgeht. Die für den Entwickler und Anwender bessere Möglichkeit ist das Persistieren der Daten, die nicht mehr in den Arbeitsspeicher passen, auf die Festplatte. In diesem Fall wird ein Teil der Performance für die Verfügbarkeit der Anwendung geopfert. Sobald wieder genug Platz im Arbeitsspeicher vorhanden ist kann dieser Zustand jedoch wieder aufgehoben werden. Bei der zweiten Möglichkeit wirft die Anwendung

eine Out-Of-Memory-Exception, die entweder vom Entwickler in Betracht gezogen wurde und behandelt wird, oder aber als Worst-Case-Szenario die Ausführung der Anwendung beendet.

Um dies zu verhindern stellt Spark die Funktionen `uncache()` und `unpersist()` zur Verfügung, mit denen nicht mehr benötigte Objekte aus dem Arbeitsspeicher, bzw. auch von der Festplatte entfernen lassen. Aufgrund der Eigenschaften von RDDs, die prinzipiell als logischer Plan vorliegen, sind die Daten nach dem Aufrufen dieser Methoden jedoch nicht verloren.

In der Umsetzung des Prototyps werden alle Dataframes mit der Methode `unpersist()` aus dem Speicher entfernt, sobald sie nicht mehr benötigt werden. Trotzdem werden teilweise sehr große Datenmengen im Arbeitsspeicher gehalten, da die entsprechenden Dataframes als Input für den nächsten Schritt noch gebraucht werden. Für die Codeverteilung werden zum Beispiel zwei Dataframes von Aufträgen mit und ohne den Code, dessen Häufigkeit angepasst werden soll, benötigt. Dazu kommt noch das Dataframe der vorgefilterten Aufträge und weitere Dataframes, die nicht direkt in der Verarbeitung verwendet werden. Diese Problematik wird zudem durch die fehlende Parallelisierung verschärft, wodurch alle Berechnungen auf einem einzigen Worker ausgeführt werden. Dementsprechend werden alle persistierten Daten für die Anwendung auf diesem Knoten gehalten.

Neben einem schwierigen Speichermanagement ist die Ausführung des Prototyps durch häufig und sporadisch auftretende Memory Leaks gekennzeichnet, was insbesondere bei langen Alternativen zu Fehlverhalten oder Abbruch der Anwendung führt. In den vorherigen Versionen von Spark wurde das Auftreten der Memory Leaks bereits festgestellt, als kritischer Fehler von Spark identifiziert und in Version 1.6 behoben [68]. Ob der Verursacher die Implementierung von Spark oder vom Prototyp ist, konnte im Lauf der Arbeit nicht festgestellt werden.

8.2.4 Verbesserungsansätze

Aufgrund der in diesem Kapitel vorgestellten Evaluierung des Prototyps lassen sich folgende wichtige Ansätze zusammenfassen, die zur Verbesserung der Performance des Prototyps beitragen können:

- Optimieren der Datenverarbeitung unter dem Gesichtspunkt des Speichermanagements
- Evaluieren und Implementieren weiterer Möglichkeiten zur Parallelisierung
- Bei unzureichenden Hardwareressourcen vertikal oder horizontal skalieren
- Portieren der Datenbank in den Cluster um die Latenz zu verringern und die Datenbank auf einem leistungsstärkeren dedizierten Rechner platzieren zu können

Nach der Durchführung dieser Ansätze lässt sich eine deutlich bessere Performance des Prototyps erwarten. Fraglich ist jedoch ob sich dieser massive Unterschied zum Originalsystem durch diese Verbesserungen beseitigen lässt oder gar bessere Performancewerte erreicht werden können.

9 Zusammenfassung und Ausblick

Zusammenfassend kann gesagt werden, dass bei der Migration des Datenmanagements einer Anwendung in ein Big-Data-Framework sorgfältig recherchiert werden muss, ob und inwieweit der Umstieg sinnvoll ist. Die Vorgehensweise in dieser Arbeit ist ein guter Anhaltspunkt für die benötigten Schritte einer erfolgreichen Migration. Unter Beachtung aller Schritte können finanzielle Mittel und Arbeitskraft eingespart werden, wenn wie im Fall dieser Arbeit die Daten und die Zugriffsmuster nicht für den Umstieg auf ein Big-Data-Framework geeignet sind.

Die Eignung kann durch die Indikatoren der Reflexion ermittelt werden, die im Nachhinein betrachtet für das vorliegende Problem den Umstieg kontraindizieren. Aus Evaluierungsgründen des Ansatzes der Durchführung von Primärbedarfsprognosen für variantenreiche Produkte in horizontaler Skalierung ist dieser Schritt allerdings übersprungen worden. Abgesehen davon sind die Indikatoren für die Reflexion erst durch die Herausforderungen und Probleme bei der Implementierung des Prototypen in Spark entstanden. Aufgrund einer zu hohen Abhängigkeit der Daten untereinander und der inkrementellen Verarbeitung konnte die horizontale Skalierbarkeit in Form von Parallelisierung nicht genutzt werden. Aufeinander aufbauende Teilverarbeitungen und die Abhängigkeit der Verarbeitungsdaten von den zu verarbeitenden Daten stehen in Konflikt mit der Parallelisierung. Durch diesen Konflikt entstanden Probleme des Speicher-Managements, ausgelöst durch die Speicherung aller verarbeitungsrelevanten Daten auf einer Recheneinheit anstatt einem Rechencluster. Die zusätzlich durch sporadisch auftretende Memory Leaks stark erschwerten Bedingungen bei der Entwicklung resultierten in einem Prototypen der die zu bewältigende Aufgabe um ein Vielfaches langsamer bearbeitet als die zu migrierende Anwendung auf dem Mainframe. Dabei ist der Großteil des Verlustes an Performance auf das Filtern großer Datenmengen anhand einer anderen Datenmenge entfallen. Durch die Definition eines Verarbeitungsschrittes als Datenmenge in den Daten selbst konnten die Parallelisierungsmechanismen von Spark nicht auf das Filtern der zu bearbeitenden Daten angewendet werden. Dadurch ist der Prototyp gezwungen sequentiell zu arbeiten und kann die Vorteile der horizontalen Skalierung nicht auf das Problem anwenden.

Entgegen den Erwartungen zu Beginn der Arbeit diskutiert die Evaluierung nicht den Performanceunterschied zwischen den Systemen, sondern erläutert die Problematik des vorliegenden Datenverarbeitungsproblems in Spark. Dabei werden performancemindernde Aspekte identifiziert, für das vorliegende Problem diskutiert und Ansätze zur Verbesserung der Performance definiert. Diese Ansätze sind der nächste Schritt in der Evaluierung der Verarbeitung von Unternehmensdaten, insbesondere Daten variantenreicher komplexer Serienprodukte, in einem Big-Data-Framework. Durch weiterführende Arbeit an dieser Problematik kann die Möglichkeit der Parallelisierung dieser Daten weiter erforscht werden.

I. Abbildungsverzeichnis

Abbildung 1 - Ansatz zur Definition von Big Data (nach [1])-----	2
Abbildung 2 - Beispiel für die Partitionierung einer Tabelle (Spalte Alter)-----	7
Abbildung 3 - Beispiel paarweiser Vergleich von Kriterien-----	9
Abbildung 4 - Bezug der Aufträge während der Prognose (nach [20]) -----	11
Abbildung 5 - abstraktes Datenmodell zur Darstellung von Aufträgen variantenreicher Produkte -----	12
Abbildung 6 - Erstellung des Auftragsvolumens für die Prognose -----	13
Abbildung 7 - Schematische Darstellung der Anpassung durch Aktionen -----	15
Abbildung 8 – aufgeschlüsselter Overhead von RDBMS [25]-----	18
Abbildung 9 - Spark Komponenten (nach [31])-----	20
Abbildung 10 - Ausführungsumgebung Spark (nach [35] und [32]) -----	21
Abbildung 11 - RDD-Abhängigkeiten (nach [35])-----	24
Abbildung 12 - Performance bei verschiedenen Speichergrößen [35] -----	25
Abbildung 13 - Ausführungszeit der Iterationen mit und ohne Knotenausfall [35] -----	26
Abbildung 14 - Spark SQL Architektur (nach [36]) -----	26
Abbildung 15 - Catalyst-Baum für den Ausdruck $x + (1 + 2)$ (nach [36])-----	28
Abbildung 16 - Anfrage-Planung in Spark SQL - abgerundete Rechtecke stellen Catalyst-Bäume dar (nach [36])-----	28
Abbildung 17 - Evaluation DataFrame API [36]-----	29
Abbildung 18 - Evaluation Pipeline Performance [36]-----	29
Abbildung 19 – ER-Modell für Aufträge-----	39
Abbildung 20 - ER-Modell der Aktionen mit restriktiven Daten -----	40
Abbildung 21 - ER-Modell mit Ausführungsdaten -----	41
Abbildung 22 - abstrakte Darstellung der Anpassung von Aufträgen durch Aktionen -----	42
Abbildung 23 - Bestandteile und Datenfluss des Prototyps-----	43
Abbildung 24 - Klassendiagramm des Prototyps-----	44
Abbildung 25 - Bestimmung des Verteilungswertes für Aktionen -----	50
Abbildung 26 - AWS-Cluster für die Sparkanwendung-----	52

Abbildung 27 - Performanceübersicht Datenextraktion Prototyp -----	56
Abbildung 28 - Datenextraktion - Vergleichswerte Squirrel -----	56
Abbildung 29 - Performance Datenextraktion Auftragsdaten -----	57
Abbildung 30 - Datenextraktion - Vergleichswerte Squirrel -----	57
Abbildung 31 – Performanceergebnisse der Alternativen-Verarbeitung Prototyp -----	58
Abbildung 32 – Performanceergebnisse der Alternativen-Verarbeitung Originalsystem -----	59

II. Tabellenverzeichnis

Tabelle 1 - Beispielhafte Aktion für die Anpassung von Codes mittels Zielverteilung -----	14
Tabelle 2 - Verfügbare Operationen auf RDDs (nach [35]) -----	22
Tabelle 3 - Gewichtung der Kriterien mit Hilfe eines paarweisen Vergleichs -----	32
Tabelle 4 - Bewertung der Frameworks mit Hilfe der gewichteten Kriterien -----	37
Tabelle 5 - Eigenschaften ec2-Instanz m1.xlarge [66] -----	52

III. Quellenverzeichnis

- [1] Emmanuel, Isitor, und Clare Stanier. „Defining Big Data“. In *Proceedings of the International Conference on Big Data and Advanced Wireless Technologies*, 5. ACM, 2016. <http://dl.acm.org/citation.cfm?id=3010090>.
- [2] Manyika, James, Michael Chui, Brad Brown, Jacques Bughin, Richard Dobbs, Charles Roxburgh, und Angela Hung Byers. „Big data: The next frontier for innovation, competition, and productivity | McKinsey & Company“. Zugegriffen 10. Mai 2017. <http://www.mckinsey.com/business-functions/digital-mckinsey/our-insights/big-data-the-next-frontier-for-innovation>.
- [3] Jacobs, Adam. „The Pathologies of Big Data“. *Communications of the ACM* 52, Nr. 8 (1. August 2009): 36.
- [4] „What Is Big Data? - Gartner IT Glossary - Big Data“. Zugegriffen 10. Mai 2017. <http://www.gartner.com/it-glossary/big-data/>.
- [5] Kobielus, James, und Marcus, Bob. „CSCC-Deploying-Big-Data-Analytics-Applications-to-the-Cloud-Roadmap-for-Success(1).pdf“. Cloud Standards Customer Council, 2014. <http://www.cloud-council.org/deliverables/CSCC-Deploying-Big-Data-Analytics-Applications-to-the-Cloud-Roadmap-for-Success.pdf>.
- [6] Demchenko, Yuri, Paola Grosso, Cees de Laat, und Peter Membrey. „Addressing big data issues in Scientific Data Infrastructure“. 2013 International Conference on Collaboration Technologies and Systems (CTS). Zugegriffen 10. Mai 2017.

http://www.academia.edu/4307681/Addressing_Big_Data_Issues_in_Scientific_Data_Infrastructure.

- [7] „Research paper: Instructional Model for Building Effective Big Data Curricula for Online and Campus Education“. ResearchGate. Zugegriffen 10. Mai 2017. https://www.researchgate.net/publication/273945502_Instructional_Model_for_Building_Effective_Big_Data_Curricula_for_Online_and_Campus_Education.
- [8] Tekiner, Firat, und John A. Keane. „Big Data Framework“, 1494–99. IEEE, 2013.
- [9] Gillis, Tom. „Cost Wars: Data Center vs. Public Cloud“. Forbes. Zugegriffen 9. März 2017. <http://www.forbes.com/sites/tomgillis/2015/09/02/cost-wars-data-center-vs-public-cloud/>.
- [10] Prigge, Matt. „The cloud isn’t always cheaper -- and that’s OK“. InfoWorld, 17. Dezember 2012. <http://www.infoworld.com/article/2616379/cloud-computing/the-cloud-isn-t-always-cheaper----and-that-s-ok.html>.
- [11] „IBM CSCC Deploying Big Data Analytics Applications to the Cloud Roadmap for Success | Big Data | Cloud Computing“. Scribd. Zugegriffen 5. Mai 2017. <https://de.scribd.com/document/307541118/IBM-CSCC-Deploying-Big-Data-Analytics-Applications-to-the-Cloud-Roadmap-for-Success>.
- [12] Pippal, Sanjeev, Shiv Pratap Singh, und Dharmender Singh Kushwaha. „Data Transfer From MySQL To Hadoop: Implementers’ Perspective“. In *Proceedings of the 2014 International Conference on Information and Communication Technology for Competitive Strategies*, 79:1–79:5. ICTCS ’14. New York, NY, USA: ACM, 2014.
- [13] „Instrument: Paarweiser Vergleich“. Promidis, 2015. <https://www.inf.uni-hamburg.de/de/inst/ab/itmc/research/completed/promidis/instrumente/paarweiser-vergleich>.
- [14] „Model Transformation and Data Migration from Relational Database to MongoDB - Semantic Scholar“. Zugegriffen 5. Mai 2017. </paper/Model-Transformation-and-Data-Migration-from-Jia-Zhao/0baf563b8ead7de96236f0abfa8c5e97ef9ac32b>.
- [15] Stonebraker, Michael, und Rick Cattell. „10 Rules for Scalable Performance in ‘Simple Operation’ Datastores“. *Communications of the ACM* 54, Nr. 6 (1. Juni 2011): 72.
- [16] „Parallel data intensive computing in scientific and commercial applications“. Zugegriffen 5. Mai 2017. <http://www.sciencedirect.com/science/article/pii/S0167819102000911>.
- [17] Pavlo, Andrew, Erik Paulson, Alexander Rasin, Daniel J. Abadi, David J. DeWitt, Samuel Madden, und Michael Stonebraker. „A comparison of approaches to large-scale data analysis“. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, 165–178. ACM, 2009. <http://dl.acm.org/citation.cfm?id=1559865>.
- [18] Gunarathne, Thilina, Tak-Lon Wu, Judy Qiu, und Geoffrey Fox. „Cloud Computing Paradigms for Pleasingly Parallel Biomedical Applications“. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, 460–469. HPDC ’10. New York, NY, USA: ACM, 2010.
- [19] Liu, Lu. „Performance comparison by running benchmarks on Hadoop, Spark, and HAMR“. University of Delaware, 2015. <https://dspace.udel.edu/handle/19716/17628>.

- [20] Stäblein, Thomas. *Integrierte Planung des Materialbedarfs bei kundenauftragsorientierter Fertigung von komplexen und variantenreichen Serienprodukten*. Innovationen der Fabrikplanung und -organisation 18. Aachen: Shaker, 2008.
- [21] Holweg, Matthias, und Frits K. Pil. *The Second Century: Reconnecting Customer and Value Chain through Build-to-Order ; Moving beyond Mass and Lean Production in the Auto Industry*. Cambridge, Mass.: MIT Press, 2004.
- [22] Kappler, Jochen, Andreas Schütte, Heiko Jung, Dennis Arnhold, und Uwe Bracht. „Robuste Primär- und Sekundärbedarfsplanung komplexer und variantenreicher Serienprodukte“. *Integrationsaspekte der Simulation: Technik, Organisation und Personal*. Karlsruhe: KIT Scientific Publishing, 2010, 69–76.
- [23] Stonebraker, Michael, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, und Pat Helland. „The end of an architectural Era (it’s time for a complete rewrite)“. In *Proceedings of the 33rd international conference on Very large data bases*, 1150–1160. VLDB Endowment, 2007. <http://dl.acm.org/citation.cfm?id=1325981>.
- [24] Chamberlin, Donald D., Morton M. Astrahan, Michael W. Blasgen, James N. Gray, W. Frank King, Bruce G. Lindsay, Raymond Lorie, u. a. „A history and evaluation of System R“. *Communications of the ACM* 24, Nr. 10 (1981): 632–646.
- [25] Harizopoulos, Stavros, Daniel J. Abadi, Samuel Madden, und Michael Stonebraker. „OLTP through the looking glass, and what we found there“. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, 981–992. ACM, 2008. <http://dl.acm.org/citation.cfm?id=1376713>.
- [26] Gottemukkala, Vibby, und Tobin J. Lehman. „Locking and latching in a memory-resident database system“. In *VLDB*, 533–544, 1992. <http://www.vldb.org/conf/1992/P533.PDF>.
- [27] „TPC-C - Homepage“. Zugegriffen 7. Januar 2017. <http://www.tpc.org/tpcc/>.
- [28] „Community | Apache Spark“. Zugegriffen 21. Dezember 2016. <https://spark.apache.org/community.html#history>.
- [29] „Sort Benchmark Home Page“. Zugegriffen 21. Dezember 2016. <http://sortbenchmark.org/>.
- [30] „The Apache Software Foundation Announces Apache™ Spark™ as a Top-Level Project : The Apache Software Foundation Blog“. Zugegriffen 21. Dezember 2016. https://blogs.apache.org/foundation/entry/the_apache_software_foundation_announces50.
- [31] Karau, Holden, Andy Konwinski, Patrick Wendell, und Matei Zaharia, Hrsg. *Learning Spark: [Lightning-Fast Data Analysis]*. 1. ed. Beijing: O’Reilly, 2015.
- [32] „Cluster Mode Overview - Spark 2.0.2 Documentation“. Zugegriffen 19. Dezember 2016. <http://spark.apache.org/docs/latest/cluster-overview.html>.
- [33] „Job Scheduling - Spark 2.0.2 Documentation“. Zugegriffen 22. Dezember 2016. <http://spark.apache.org/docs/latest/job-scheduling.html>.

- [34] Zaharia, Matei, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, u. a. „Apache Spark: A Unified Engine for Big Data Processing“. *Commun. ACM* 59, Nr. 11 (Oktober 2016): 56–65.
- [35] Zaharia, Matei, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, und Ion Stoica. „Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing“. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, 2–2. USENIX Association, 2012. <http://dl.acm.org/citation.cfm?id=2228301>.
- [36] Armbrust, Michael, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, u. a. „Spark SQL: Relational Data Processing in Spark“. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, 1383–1394. SIGMOD ’15. New York, NY, USA: ACM, 2015.
- [37] „Spark SQL and DataFrames - Spark 2.1.0 Documentation“. Zugegriffen 23. Februar 2017. <http://spark.apache.org/docs/latest/sql-programming-guide.html>.
- [38] „LanguageManual DDL - Apache Hive - Apache Software Foundation“. Zugegriffen 4. Januar 2017. <https://cwiki.apache.org/confluence/display/Hive/LanguageManual+DDL>.
- [39] Xin, Reynold S., Josh Rosen, Matei Zaharia, Michael J. Franklin, Scott Shenker, und Ion Stoica. „Shark: SQL and rich analytics at scale“. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of data*, 13–24. ACM, 2013. <http://dl.acm.org/citation.cfm?id=2465288>.
- [40] Bittorf, Marcel Kornacker Alexander Behm Victor, Taras Bobrovitsky, Casey Ching Alan Choi Justin Erickson, Martin Grund Daniel Hecht, Matthew Jacobs Ishaan Joshi Lenni Kuff, Dileep Kumar Alex Leblang, Nong Li Ippokratis Pandis Henry Robinson, David Rorke Silvius Rus, John Russell Dimitris Tsirogiannis Skye Wanderman, und Milne Michael Yoder. „Impala: A Modern, Open-Source SQL Engine for Hadoop“. CIDR, 2015. <http://web.eecs.umich.edu/~mozafari/fall2015/eecs584/papers/impala.pdf>.
- [41] „Big Data Benchmark“. Zugegriffen 4. Januar 2017. <https://amplab.cs.berkeley.edu/benchmark/>.
- [42] „Apache Projects List“. Zugegriffen 23. Februar 2017. <https://projects.apache.org/projects.html?name>.
- [43] „EC2-Instance-Preise – Amazon Web Services (AWS)“. *Amazon Web Services, Inc.* Zugegriffen 23. Februar 2017. <https://aws.amazon.com/de/ec2/pricing/on-demand/>.
- [44] „Apache Hadoop 2.7.2 – HDFS Architecture“. Zugegriffen 23. Februar 2017. <https://hadoop.apache.org/docs/r2.7.2/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html>.
- [45] „Apache Flink 1.2.0 Documentation: Connectors“. Zugegriffen 23. Februar 2017. <https://ci.apache.org/projects/flink/flink-docs-release-1.2/dev/batch/connectors.html>.
- [46] „Welcome to Apache™ Hadoop®!“ Zugegriffen 24. Februar 2017. <http://hadoop.apache.org/#What+Is+Apache+Hadoop%3F>.
- [47] Dean, Jeffrey, und Sanjay Ghemawat. „MapReduce: Simplified Data Processing on Large Clusters“. *Communications of the ACM* 51, Nr. 1 (1. Januar 2008): 107.

- [48] „Oozie -“. Zugegriffen 24. Februar 2017. http://oozie.apache.org/docs/4.3.0/DG_Overview.html.
- [49] „Hadoop File Formats: It’s not just CSV anymore“. Text. *Inquidia*. Zugegriffen 24. Februar 2017. <http://www.inquidia.com/news-and-info/hadoop-file-formats-its-not-just-csv-anymore>.
- [50] „Apache Flink 1.2.0 Documentation: DataSet Transformations“. Zugegriffen 24. Februar 2017. https://ci.apache.org/projects/flink/flink-docs-release-1.2/dev/batch/dataset_transformations.html.
- [51] „Apache Flink 1.2.0 Documentation: Dataflow Programming Model“. Zugegriffen 24. Februar 2017. <https://ci.apache.org/projects/flink/flink-docs-release-1.2/concepts/programming-model.html>.
- [52] „Apache Flink 1.2.0 Documentation: Connectors“. Zugegriffen 24. Februar 2017. <https://ci.apache.org/projects/flink/flink-docs-release-1.2/dev/batch/connectors.html>.
- [53] „Apache Flink 1.2.0 Documentation: Table and SQL“. Zugegriffen 24. Februar 2017. https://ci.apache.org/projects/flink/flink-docs-release-1.2/dev/table_api.html.
- [54] „Apache Flink 1.2.0 Documentation: Fault Tolerance“. Zugegriffen 24. Februar 2017. https://ci.apache.org/projects/flink/flink-docs-release-1.2/dev/batch/fault_tolerance.html.
- [55] „Memory Management (Batch API) - Apache Flink - Apache Software Foundation“. Zugegriffen 24. Februar 2017. <https://cwiki.apache.org/confluence/pages/viewpage.action?pageId=53741525>.
- [56] „Apache Avro™ 1.8.1 Documentation“. Zugegriffen 24. Februar 2017. <http://avro.apache.org/docs/current/>.
- [57] He, Yongqiang, Rubao Lee, Yin Huai, Zheng Shao, Namit Jain, Xiaodong Zhang, und Zhiwei Xu. „RCFile: A fast and space-efficient data placement structure in MapReduce-based warehouse systems“. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, 1199–1208. IEEE, 2011. <http://ieeexplore.ieee.org/abstract/document/5767933/>.
- [58] „Apache Parquet“. Zugegriffen 23. Februar 2017. <https://parquet.apache.org/>.
- [59] „LIBSVM -- A Library for Support Vector Machines“. Zugegriffen 23. Februar 2017. <https://www.csie.ntu.edu.tw/~cjlin/libsvm/>.
- [60] „Apache ORC - Documentation - Background“. Zugegriffen 22. Februar 2017. <https://orc.apache.org/docs/>.
- [61] „Hadoop Streaming“. Zugegriffen 24. Februar 2017. <https://hadoop.apache.org/docs/r1.2.1/streaming.html>.
- [62] „Factory Pattern | Object Oriented Design“. Zugegriffen 11. Mai 2017. <http://www.oodesign.com/factory-pattern.html>.
- [63] „Sqoop“. Zugegriffen 24. Februar 2017. <http://sqoop.apache.org/>.
- [64] „Apache Hive™“. Zugegriffen 24. Februar 2017. <https://hive.apache.org/>.
- [65] GmbH, Data Geekery. „jOOQ“. jOOQ. Zugegriffen 8. Mai 2017. <http://www.jooq.org>.

- [66] „Instances der vorherigen Generation“. Amazon Web Services, Inc. Zugegriffen 9. Mai 2017. [//aws.amazon.com/de/ec2/previous-generation/](https://aws.amazon.com/de/ec2/previous-generation/).
- [67] „amplab/spark-ec2“. GitHub. Zugegriffen 9. Mai 2017. <https://github.com/amplab/spark-ec2>.
- [68] „[SPARK-11293] ExternalSorter and ExternalAppendOnlyMap should free shuffle memory in their stop() methods - ASF JIRA“. Zugegriffen 12. Mai 2017. <https://issues.apache.org/jira/browse/SPARK-11293>.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben.

Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet.

Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht.

Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Stuttgart, den 1. Mai 2017 _____

Declaration

I hereby declare that the work presented in this thesis is entirely my own.

I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations.

Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before.

The electronic copy is consistent with all submitted copies

Stuttgart, 1. Mai 2017 _____