Universität Rostock Traditio et Innovatio

Bachelorarbeit

Effiziente Datenvorbereitung für Analysen im Automotive-Bereich

eingereicht von: Adrian Lutsch

eingereicht am: 25.03.2019

Gutachter: Prof. Dr. rer. nat. habil. Andreas Heuer
Dr.-Ing. Holger Meyer

Zusammenfassung

In dieser Arbeit wird ein Konzept zum Einsatz von Apache Spark für die Analyse von Fahrzeugmessdaten vorgestellt und evaluiert. Fahrzeugmessdaten sind multivariate Zeitreihen mit beliebiger, möglicherweise sehr großer Anzahl von Messpunkten und beliebigem, nicht in jedem Fall äquidistanten Zeitraster. Ziel ist ein System zur zentralisierten und parallelisierten Auswertung von Messungen, die im Rahmen der Fahrzeugentwicklung durchgeführt werden.

Die Arbeit gibt einen Überblick über Systeme zur Analyse und Speicherung von Messdaten. Es wird ein Konzept für einen Softwarestack zur zentralen Messdatenanalyse entwickelt, in dessen Rahmen unterschiedliche Systeme für die Speicherung verwendet werden können. Existierende Erweiterungen zur Zeitreihenanalyse in Apache Spark werden evaluiert und neue Schemata und Transformationen zwischen diesen konzipiert.

Basierend auf den Konzepten wird eine Bibliothek mit Funktionen für das Laden der Messdaten, Transformationen zwischen Schemata und einfachen Analysen vorgestellt. Als Speichertechnologien werden Postgres-XL und Apache Parquet evaluiert und verglichen. Weiterer Systeme, die genutzt werden könnten, werden vorgeschlagen. Die Verwendbarkeit des Systems wird anhand von Beispielanalysen gezeigt.

Abstract

This thesis proposes and evaluates a concept for the analysis of vehicle measurement data with Apache Spark. Vehicle measurement data is composed of multivariate time series with a variable, possibly great number of data points and not necessarily equally spaced time stamps. The goal is to create a system for centralized and parallelized analysis of measurements carried out during the vehicle development.

After an overview of systems for analysis and storage of measurement data is given, a concept for a software stack for centralized measurement data analysis is developed. The stack supports the use of different systems for data storage. Existing extensions for time series analysis in Apache Spark are evaluated and new schemas and transformations between these schemas are designed.

Based on these concepts a library containing functions for data loading, transformations between schemas and basic data analysis is implemented. Postgres-XL and Apache Parquet are used to store the measurement data and are compared to each other. Additional systems that could be tested or used are proposed. The usability of the system is demonstrated by the implementation of selected use cases.

Inhaltsverzeichnis

1. Einleitung

1.1. Problemstellung

Seit Jahren findet in der Automobilindustrie eine Digitalisierung auf allen Ebenen statt. Immer mehr Teile von Fahrzeugen werden heute mittels digitaler Technik überwacht und gesteuert. Bei der Weiterentwicklung von Fahrzeugen spielt das Aufzeichnen von Messdaten und deren Auswertung eine wachsende Rolle. Durch Verbesserung der Sensoren und Aufzeichnungslösungen sind die Datenmengen aus Prüfständen und Testfahrten gestiegen. Zusätzlich werden immer ausgefeiltere Datenanalysen über längere gemessene Zeitabschnitte durchgeführt. Die Speicherung und performante Bereitstellung dieser Daten ist eine wachsende Herausforderung, die mit neuen, verteilten Speichertechnologien angegangen wird, weil die Zugriffs- und Analysegeschwindigkeiten einzelner Server in Zeiten von Big Data und Big-Data-Analysen nicht mehr ausreichend sind.

Die Forschungsgruppe Datenbank- und Informationssysteme an der Universität Rostock beschäftigt sich im Rahmen eines Drittmittelprojekts mit der effizienten Speicherung von Fahrzeugmessdaten in verschiedenen (No)SQL-Datenbankmanagementsystemen. Ziel ist es, existierende Messdaten in einem System zu speichern und mittels Index die Performance von Auswertungen zu verbessern.

Bisher werden die Messdaten im Loggerformat Measurement Data Format (MDF) in einer Verzeichnisstruktur auf einem Server abgelegt und von Analysten zur Arbeit auf den lokalen PC kopiert. Diese Lösung ist zwar schnell eingerichtet und intuitiv benutzbar, hat aber viele Nachteile. So besteht ein hoher Verwaltungsaufwand und die Analyse ist immer an die Ressourcen des PCs und Parallelisierungsmöglichkeiten der genutzten Programmiersprache gebunden. Die Speicherung erfolgt zudem messungsweise, was bei messungsübergreifenden Auswertungen zu wiederkehrendem Aufwand führt. Außerdem ist die Indexierung über Ordnerstrukturen und Dateinamen fehleranfällig und unflexibel.

Diese Probleme sollen im Projekt durch die Speicherung in einem Datenbanksystem gelöst werden. Dadurch wird zentraler Zugriff auf Messungen ohne vorheriges Kopieren ermöglicht. Zudem wird übergreifende Indexierung und Metadatenspeicherung das Auffinden und gemeinsame Auswerten der Daten vereinfachen. Die Möglichkeit Selektionen, Projektionen, Aggregationen und Joins zentral und effizient vom Datenbankserver ausführen zu lassen, wird als größter Vorteil dieser Herangehensweise vermutet.

Darauf aufbauend wird in dieser Arbeit die Nutzung einer integrierten Analyseumgebung, die das Auswerten der effizient abgelegten Daten mit massiver Parallelisierung unterstützt, evaluiert.

1.2. Thema der Arbeit

In dieser Arbeit wird evaluiert, wie Auswertungen vorgenommener Experimente, Beprobungen und Testläufe mit Apache Spark unterstützt werden können und mit welchem zugrundeliegenden Speichersystem die beste Performance bezüglich der im Projekt festgelegten Benchmarks erreicht wird. Es wird ein Analyse-Stack von Datenspeicherung bis Nutzerschnittstelle vorgestellt, der die Programmierung

in Python unterstützt und Datenauswertungen auf einem Apache-Spark-Cluster ermöglicht. Dazu werden die Systeme MySQL, Postgres-XL und Parquet in HDFS zur Datenhaltung evaluiert und Ansätze für die Verwendung von HBase und VectorH vorgestellt. Das User Interface wird durch Jupyter Notebooks bereitgestellt. Als Schnittstelle zu den Datenbanken werden Bibliotheken implementiert und vorgestellt, die das einfache Importieren, Laden, Explorieren und Auswerten der gegebenen Messdaten unterstützen. Außerdem werden sowohl kommerziell vertriebene, als auch in der Literatur vorgeschlagene Software-Stacks um Apache Spark vorgestellt und mit dem hier entwickelten Ansatz verglichen.

In Abgrenzung zum Titel ist der Kern der Arbeit nicht die Datenvorbereitung, sondern ein Konzept zur Datenanalyse in Apache Spark mit Anbindung zu unterschiedlichen Datenquellen. Die Änderung des Themas hat sich durch verschiedene Schwierigkeiten beim Betrieb der Speichersysteme ergeben. Zudem ist die Bereitstellung eines funktionierenden Analysesystems ein zusätzlicher Nutzen im Rahmen des Projekts der Arbeitsgruppe.

Andere Analyseumgebungen in Konkurrenz zu Apache Spark werden in dieser Arbeit nicht betrachtet und es wird nur Batch-Verarbeitung der Daten untersucht. Streaming ist im Usecase nicht vorgesehen, weil es sich um Messdaten von durchgeführten Testläufen handelt.

Die Arbeit gliedert sich von hier an wie folgt: In Kapitel 2 werden Grundlagen für die Analyse von großen Mengen Messdaten in verteilten Systemen vorgestellt. Anschließend wird der Stand der Technik in der Messdatenspeicherung und -analyse in Kapitel 3 näher beleuchtet. Kapitel 4 erläutert die Konzepte der implementierten Funktionsbibliotheken und des entwickelten Softwarestacks und stellt Kriterien zur Auswahl des besten Speichersystems für Messdatenanalyse in Apache Spark vor. Anschließend werden in Kapitel 5 die Einrichtung des zum Testen verwendeten Clusters und die implementierten Funktionsbibliotheken beschrieben, die in Kapitel 6 mit Beispielanwendungen evaluiert und diskutiert werden. In Kapitel 7 wird die Arbeit zusammengefasst und Ansätze für weiterführende Forschung vorgestellt.

2. Grundlagen

Im folgenden Kapitel werden wichtige Grundlagen für die Analyse von großen Mengen Messdaten in verteilten Systemen vorgestellt. Zunächst wird auf grundsätzliche Strategien zur parallelisierten Ausführung von Algorithmen eingegangen (2.1), anschließend werden Möglichkeiten zur Partitionierung und Verteilung von Datenbeständen vorgestellt (2.2). In Abschnitt 2.3 wird zusätzlich erläutert, wie sich die Art der Speicherung von Tupeln auf die Performance unterschiedlicher Anfragen auswirkt. Abschnitt 2.4 und 2.5 gehen näher auf wichtige Methoden zur Programmierung der genutzten Analysesysteme ein: Funktionen höherer Ordnung und User Defined Functions.

2.1. Parallelisierung

Bei der Parallelisierung von Anwendungen ist das Ziel, Nebenläufigkeit von Programmteilen auszunutzen, um mehrere Prozessorkerne zur gleichen Zeit mit Aufgaben zu versorgen und durch die gleichzeitige Berechnung einen Speedup zu erhalten. An dieser Stelle werden die in verschiedenen Systemen genutzten Parallelisierungsansätze vorgestellt.

2.1.1. Grundsätzliche Parallelisierungsstrategien

Eine Standardeinteilung für parallele Rechnerarchitekturen ist die Flynn'sche Taxonomie [Fly66]. Sie teilt Rechenarchitekturen in zwei Dimensionen ein: Parallelisierung von Daten und Parallelisierung von Instruktionen. So ergeben sich die 4 Kategorien Single Instruction Single Data (SISD), Multiple Instruction Single Data (MISD), Single Instruction Multiple Data (SIMD) und Multiple Instruction Multiple Data (MIMD), die in Abbildung 2.1 tabellarisch eingeordnet werden.

SISD ist keine parallelisierende Architektur. Es wird pro Schritt eine Instruktion mit ihren Parametern (Daten) ausgeführt. SIMD-Architekturen wenden eine Instruktion auf mehrere Eingabedaten parallel an. Man spricht von vektorisierter Berechnung und Vektorrechnern, die diese Architektur implementieren. Ein Beispiel für SIMD-Architekturen sind die AVX-Befehlssatzerweiterungen für x86-Prozessoren von Intel [Mar17]. Die in dieser Arbeit verwendeten Systeme wenden auf einen partitionierten Datenbestand parallel die gleichen Instruktionen an und sind als SIMD zu klassifizieren.

MIMD-Architekturen parallelisieren neben den Daten auch die Instruktionen. MIMD ist Standard bei Prozessoren mit mehreren Kernen. Die meisten aktuellen Prozessoren nutzen die MIMD-Architektur [Unb08].

	Single Instruction	Multiple Instruction
Single Data	SISD	MISD
Multiple Data	SIMD	MIMD

Abbildung 2.1. Flynn'sche Taxonomie

Eine andere Kategorisierung von Parallelisierung kann anhand des Gegenstands der Parallelisierung vorgenommen werden. Hierbei werden vor allem Task-Parallelism und Data-Parallelism unterschieden [Rei07]. Während bei Task-Parallelism mehrere unterschiedliche Aufgaben gleichzeitig ausgeführt werden, wird bei Data-Parallelism durch partitionierte oder replizierte Datenhaltung und dadurch schnelleren Datenzugriff und parallele Ausführung von Instruktionen auf unterschiedlichen Daten ein Speedup erreicht [Rei07]. In diesem Sinne ist Data-Parallelism vergleichbar mit SIMD, wobei er sich nicht auf die Rechnerarchitektur beschränkt, sondern abstrakter den Algorithmus und die Datenstruktur in den Mittelpunkt der Betrachtung stellt. Übliche Anwendungsfelder von Data-Parallelism sind die Berechnung von Operationen auf sehr großen Matrizen oder Arrays [Wik19a].

2.1.2. Parallelisierung in Shared-Nothing-Architekturen

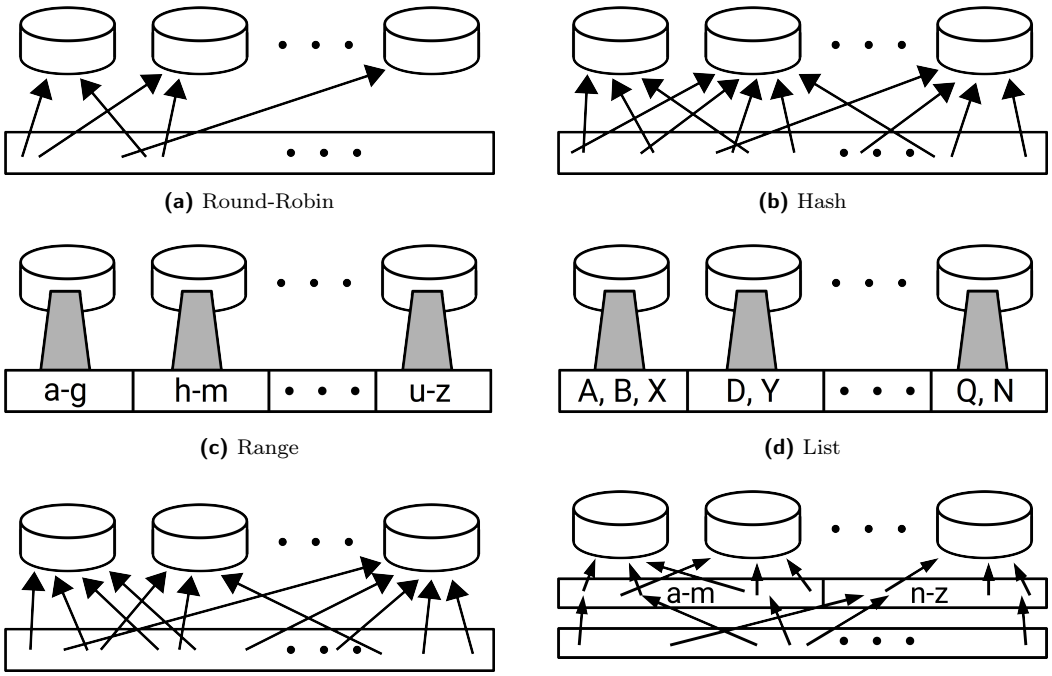
Die in dieser Arbeit verwendeten verteilten Systeme basieren auf der Shared-Nothing-Architektur. Beim Shared-Nothing-Ansatz hat jeder Prozessor exklusiven Zugriff auf den angeschlossenen Haupt- und Sekundärspeicher. Entsprechend läuft auf jedem Prozessor ein von den anderen unabhängiges Betriebssystem, auf dem eine lokale Instanz des Datenbank- oder Analysesystems ausgeführt wird. Die Shared-Nothing-Architektur steht im Gegensatz zur Shared-Disk- und der Shared-Memory-Architektur, bei denen Sekundärspeicher (Shared-Disk) oder Hauptspeicher (Shared-Memory) unter den Prozessoren aufgeteilt werden. Wichtig für die Skalierbarkeit der Shared-Nothing-Architektur ist eine schnelle Verbindung zwischen den einzelnen beteiligten Systemen. Ist diese schnelle Verbindung vorhanden, können potentiell tausende Knoten an einem System beteiligt sein [ÖV11].

Die Shared-Nothing-Architektur hat drei zentrale Vorteile gegenüber Shared-Memory- und Shared-Disk-Architekturen: geringe Kosten, hohe Skalierbarkeit und hohe Ausfallsicherheit. Die geringeren Kosten ergeben sich aus der Nutzung von Standardhardware. Spezielle Systeme für das Teilen von Haupt- oder Sekundärspeicher werden nicht benötigt. Die hohe Erweiterbarkeit kann durch passende Implementierung erreicht werden, die einfaches Hinzufügen von Knoten zum System ermöglicht. Zuletzt kann hohe Ausfallsicherheit erreicht werden, indem Daten auf mehreren Knoten redundant gespeichert werden [ÖV11].

2.2. Partitionierung von Datenbeständen

Weil die Daten einer Anwendung – vor allem bei Big-Data-Anwendungen – viel größer sind als die Programme, sollte die Ausführung auf dem Knoten stattfinden, auf dem die Daten abgespeichert sind, um die Zugriffslücke nicht durch Netzwerkübertragung zu vergrößern. Zur Beschleunigung der Ausführung durch Parallelisierung werden die Daten im Speichersystem partitioniert, sodass sie über viele Knoten verteilt sind und diese Knoten an der Ausführung beteiligt sind. Dabei ist zu beachten, dass bei der Verteilung auf viele Knoten der Kommunikationsaufwand zwischen den einzelnen Maschinen steigt. Es gilt eine Abwägung zu treffen: je mehr Knoten sich die Daten teilen, desto schneller wird die Ausführung von parallelen Berechnungen, desto größer wird der Kommunikationsaufwand zum Zusammenfassen der Daten. Je weniger Knoten relevante Daten enthalten, desto kleiner der Kommunikationsaufwand und desto weniger Parallelisierung findet statt, wenn Daten lokal verarbeitet werden sollen.

Zur Partitionierung von Datenbeständen gibt es verschiedene grundsätzliche Strategien. In [ÖV11] werden Round-Robin, Hash- und Range-Partitionierung vorgestellt, zusätzlich bieten verschiedene Systeme die List-, Random- und Composite-Partitionierung an. Die Partitionierungsarten werden im Folgenden kurz erklärt und sind in Abbildung 2.2 dargestellt.



(a) Round-Robin

(b) Hash

(c) Range

(d) List

(e) Random

(f) Composite: Erst Range-Partitionierung, dann Hash-Partitionierung

Abbildung 2.2. Arten von Partitionierungen

Round-Robin Bei der Round-Robin-Partitionierung werden die Datensätze nach der Reihenfolge des Hinzufügens auf die beteiligten Knoten verteilt. Der erste Datensatz wird dem ersten Knoten zugeteilt, der zweite Datensatz dem zweiten Knoten, nach diesem Verfahren wird fortgefahren, bis der letzte Knoten erreicht ist. Anschließend wird wieder dem ersten Knoten ein Datensatz zugeteilt. Das Round-Robin-Verfahren sorgt für eine gleiche Anzahl von Datensätzen auf allen Knoten und nimmt keine Rücksicht auf Zusammenhänge zwischen den einzelnen Datensätzen. Vorteil dieses Verfahrens ist die gleichmäßige Auslastung aller Knoten. Nachteil ist, dass Lesevorgänge oder Auswertungen, die auf einen Teil der Daten beschränkt sind, nicht auf einen Teil der beteiligten Knoten eingeschränkt werden.

Hash Hash-Partitionierung nutzt eine Hashfunktion auf mindestens einem Attribut eines Datensatzes, um den Datensatz einer Partition zuzuweisen. Bei Hash-Partitionierung ist es von der verwendeten Hash-Funktion und von der Verteilung der Attributwerte abhängig, wie gleichmäßig die Tupel auf die Knoten verteilt werden. Partitionierung mittels Hash ermöglicht Exact-Match-Anfragen auf dem Hash-Attribut mit konstanter Komplexität, weil diese von einem einzelnen Knoten bearbeitet werden. Bei anderen Anfragen sind alle Knoten beteiligt.

Range Range-Partitionierung verteilt Datensätze anhand des Enthalten-Seins eines Attributwerts des Datensatzes in einem Intervall der Domain des Attributs. Range-Partitionierung sorgt sowohl bei Exact-Match-Anfragen, als auch bei Anfragen der Form „alle Datensätze bei denen Attribut A in $[A_1, A_2]$“ für geringen Kommunikationsaufwand, weil die Datensätze zusammen auf einem Knoten gespeichert werden. Nachteil dieser Verteilung ist, dass die Partitionen bei vielen Arten von Attributen[1] sehr unterschiedliche Größen annehmen.

List Bei der Partitionierung anhand von Listen werden die Datensätze über die Zugehörigkeit eines Attributwerts zu einer endlichen Liste von Werten verteilt. List-Partitionierung ist sehr ähnlich zur Range-Partitionierung und teilt sich deshalb Vor- und Nachteile mit dieser, erfordert aber eine endliche Domain [Ora19b].

[1] z.B. Nachnamen oder Wörter einer Sprache aufgeteilt nach den Anfangsbuchstaben

Random Bei dieser Art der Verteilung werden die Datensätze zufällig den einzelnen Knoten zugewiesen. Dadurch enthalten die Partitionen ähnliche Mengen Datensätzen, aber für alle Anfragen müssen wie beim Round-Robin-Verfahren alle Knoten genutzt werden [IBM14b].

Composite Composite-Partitionierung beschreibt die aufeinanderfolgende Anwendung mehrerer hier genannter Verfahren, um die Partition für einen Datensatz zu bestimmen. Zum Beispiel kann eine Range-Partitionierung der Datensätze nach Attribut A vorgenommen werden und Unterpartitionen durch List-Partitionierung auf Attribut B erzeugt werden. Üblicherweise werden Hash-, Range- und List-Partitionierung kombiniert [Ora19a].

2.3. Row Stores und Column Stores

Zur Speicherung von tabellarischen Daten gibt es zwei grundsätzliche und einen kombinierten Ansatz, die hier erläutert werden. Ursprünglich haben nahezu alle relationalen Datenbanken die zeilenorientierte Speicherung verwendet, bei der die Einträge einer Zeile hintereinander auf dem Sekundärspeicher abgelegt werden und die Zeilen aufeinander folgen. Systeme, die diesen Ansatz verwenden, werden Row Stores genannt. Im Gegensatz dazu speichern Column Stores alle Werte einer Spalte hintereinander im Speicher und erst dann folgt die nächste Spalte. Dieser Ansatz wird auch als spaltenorientierte Speicherung bezeichnet. Es zeigt sich, dass Column Stores Vorteile beim Lesen wenigen Spalten und beim Bilden von Aggregaten haben und es ermöglichen, Caching und Pipelining-Techniken moderner CPUs besser auszunutzen, während Row Stores besser geeignet sind für gemischte Aufgaben und Transaktionsverarbeitung mit Lösch- und Änderungsoperationen [SSH11].

Um die Vorteile beider Ansätze zu nutzen und die Nachteile auszugleichen, gibt es einen kombinierten Ansatz, bei dem die Tabelle zeilenweise in Blöcke zerlegt wird, die spaltenorientiert gespeichert werden. Je mehr Zeilen in einem Block gespeichert werden, desto mehr wird dieser Ansatz zum Column Store und hat dessen Vor- und Nachteile. Je weniger Zeilen in einem Block, desto mehr ähnelt die Struktur einer zeilenorientierten Speicherung [SSH11].

2.4. Funktionen höherer Ordnung

Funktionen höherer Ordnung sind Funktionen, die andere Funktionen als Parameter entgegennehmen, oder Funktionen als Ergebnis zurückgeben. Der Ansatz kommt aus der Funktionalen Programmierung. Für diese Arbeit relevant sind Funktionen, die andere Funktionen als Parameter akzeptieren und dabei vor allem `map`, `filter` und `fold` bzw. `reduce`.

map Die `map`-Funktion $map :: (a \to b) \to [a] \to [b]$ wendet eine übergebene einstellige Funktion auf jedes Element einer Liste an und erzeugt eine Liste mit den Ergebnissen der Anwendung.

filter Die `filter`-Funktion $filter :: (a \to bool) \to [a] \to [a]$ hat als Argument eine Funktion mit boolschem Ergebnis und eine Liste. Es erzeugt die Liste aller Elemente der ursprünglichen Liste, die das Prädikat erfüllen.

fold/reduce Die `fold`-Funktion $fold :: (b \to a \to b) \to b \to [a] \to b$ (je nach Programmiersprache `reduce`) wendet eine zweistellige Funktion auf die Elemente einer Liste an, bis nur noch ein Ergebniswert übrig ist. Es benötigt zusätzlich einen Startwert und geht entweder vorwärts oder rückwärts durch die Liste.

```
>>> map(lambda x: x * x, [1,2,3,4])
[1,3,9,16]
>>> filter(lambda x: x % 2 == 0, [1,2,3,4])
[2,4]
>>> reduce(lambda a, b: a * b, [1,2,3,4], 1)
24
```

Programmcode 2.1 Wichtige Funktionen höherer Ordnung im Beispiel. Das erste Argument der Funktionen ist jeweils die übergebene anzuwendende Funktion, hier in Lambda-Notation. Das zweite Argument ist jeweils eine Eingabeliste. Das dritte Argument der `reduce`-Funktion ist der Startwert. Die Anwendung von `map` erzeugt die Liste der Quadrate, `filter` selektiert im Beispiel die geraden Zahlen, `reduce` bildet das Produkt aller Listenelemente.

Alle drei Standardfunktionen werden in Programmcode 2.1 am Beispiel von Python dargestellt. Das erste Argument der Funktionen ist jeweils die übergebene anzuwendende Funktion, hier in Lambda-Notation. Das zweite Argument ist jeweils eine Eingabeliste. Das dritte Argument der `reduce`-Funktion ist der Startwert. Die Anwendung von `map` erzeugt die Liste der Quadrate, `filter` selektiert im Beispiel die geraden Zahlen, `reduce` bildet das Produkt aller Listenelemente.

2.5. User Defined Functions

Der Begriff User Defined Function (UDF) kommt aus dem Bereich der relationalen Datenbanken und bezeichnet vom Nutzer implementierte Funktionen, die innerhalb von SQL-Anfragen verwendet werden können. UDFs wurden 1999 zuerst in den SQL-Standard aufgenommen und 2003 erweitert. Grundsätzlich können UDFs in zwei Dimensionen kategorisiert werden: nach Implementierungsart und -sprache und nach den Ein- und Ausgabewerten. Weil sich diese Arbeit nicht mit User Defined Functions bei relationalen Datenbanken beschäftigt, ist die erste Dimension nicht relevant. In der zweiten Dimension unterscheidet man skalare Funktionen, Aggregatfunktionen, Zeilenfunktionen und Tabellenfunktionen [Mey18; IBM14a; Ora19c], deren Eigenschaften in Tabelle 2.1 verglichen werden.

Bei der Verwendung von relationalen Datenbanken bieten sich UDFs an, um Anwendungslogik nahe bei den Daten umzusetzen. So kann der Netzwerkverkehr reduziert werden und der Transaktionsschutz vom Datenbanksystem ausgenutzt werden. Außerdem ermöglichen UDFs das Nachrüsten von Funktionalität in einem Datenbanksystem, was den Wechsel zwischen unterschiedlichen Systemen funktionell vereinfacht [SSH18].

	Eingabe	Ausgabe	Beispiele
skalare Funktionen	feste Anzahl von Werten	einzelner Wert	lower, upper, *
Aggregatfunktionen	eine order mehrere Spalten	ein einzelnes Tupel	min, max, corr
Zeilenfunktionen	feste Anzahl von Strukturierten Werten	ein einzelnes Tupel	-
Tabellenfunktionen	beliebig	eine Tabelle	-

Tabelle 2.1. Arten von UDFs verglichen, Quelle: [Mey18]

2.6. Nutzung der Grundlagen

Die bis hierhin vorgestellten Methoden sind wichtige Grundlagen für die in der Arbeit verwendeten Systeme. Apache Spark, das in Abschnitt 3.3 vorgestellt wird, nutzt Datenparallelität und SIMD-Instruktionen zur Parallelisierung und Beschleunigung von Berechnungen. Die genutzten Datenbanksysteme (Abschnitt 3.4)

stellen verschiedene Partitionierungsarten bereit, um Daten über mehrere Systeme zu verteilen. Alle nutzen die Shared-Nothing-Architektur. Das Verständnis von Row- und Column-Stores ist relevant für die Einschätzung der Unterschiede zwischen den vorgestellten Speichersystemen (Abschnitt 3.4 und Abschnitt 3.5). Funktionen höherer Ordnung und insbesondere User Defined Functions sind wichtig für die Programmierung von Apache Spark und werden in Kapitel 4 und Kapitel 5 genutzt.

Im folgenden Kapitel werden die Systeme zur Speicherung und Analyse von Daten genauer betrachtet. Sie bauen auf die hier vorgestellten Grundlagen auf.

3. Mittel und Methoden der Messdatenspeicherung und -Analyse

In diesem Kapitel wird zunächst untersucht, wie die wissenschaftliche Datenanalyse aktuell vorgenommen wird. Im Anschluss wird die Struktur der Messdaten und des Speicherformats vorgestellt (3.2), die in dieser Arbeit verarbeitet werden. Die Möglichkeiten von Apache Spark, das als parallelisierte Analysesoftware das Kernstück des vorgestellten Softwarestacks bildet, werden in Abschnitt 3.3 beschrieben. Weil die Verbindung von Spark mit verschiedenen Speichersystemen getestet werden soll, werden Datenbanksysteme (3.4) und unterschiedliche Dateiformate (3.5), die zur Speicherung von Messdaten genutzt werden könnten, vorgestellt. Zuletzt gibt Abschnitt 3.6 einen kurzer Überblick über Architekturen, die zur Analyse von Fahrzeugdaten in der Literatur vorgeschlagen wurden, und Abschnitt 3.7 stellt kommerziellen Angebote für Analyseumgebungen mit Apache Spark vor.

3.1. Datenauswertung

Datenanalyse ist in der Anwenderliteratur in den letzten Jahren ein wichtiges Thema. Beispielsweise listet ein internationaler Fachbuchverlag in seiner Suche ab 2015 mehr als 15 Bücher über die Statistik- und Data-Analytics-Sprache R [ORe19a] und mehr als 10 Bücher über die Nutzung von Python für Datenanalyse oder Machine Learning [ORe19b]. In diesem Abschnitt sollen häufig zur Datenanalyse genutzte Tools kurz vorgestellt werden und das gewöhnlich durch Literatur gegebene Vorgehen untersucht werden.

Häufig eingesetzte Software für Datenanalyse auf Notebooks oder Workstations ist zum einen Python [Pyt] mit den Bibliotheken Numpy, SciPy [Sci19] und Pandas [Unb19] für wissenschaftliches Rechnen und zum anderen R [The19b], das von Statistikern entwickelt wurde und eine Vielfalt von Bibliotheken für verschiedenste Analyseaufgaben bietet. Von Ingenieuren und Naturwissenschaftlern wird außerdem Matlab [Mat19] verwendet, das durch eine integrierte Oberfläche, viele kuratierte Erweiterungen und durch den hohen Preis heraussticht. In der Finanzwelt ist SPSS [IBM19] ein häufig verwendetes Tool zur Datenauswertung.

Das aus verschiedenen Büchern [Gru16; McK15] und aus der Erfahrung des Autors bekannte Vorgehen zur Datenanalyse mit diesen Tools ist folgendermaßen. Zunächst wird angenommen, dass die Daten auf dem Sekundärspeicher des verwendeten PCs vorliegen. Es wird weiterhin angenommen, dass die zu verarbeitenden Daten simultan in den Hauptspeicher des verwendeten Gerätes passen. Reicht der RAM nicht aus, wird dies durch wiederholtes Ausführen der gleichen Algorithmen auf unterschiedlichen Daten umgangen. Bei mehr Daten ist zunehmend die Lesegeschwindigkeit des Sekundärspeichers der begrenzende Faktor. Ein Terrabyte Daten von einer schnellen Festplatte mit 200 $\frac{MB}{s}$ zu laden, dauert ungefähr 1,5 Stunden. Im Rahmen dieser Arbeit soll die Analyse größerer Datenmengen in kurzer Zeit ermöglicht werden, indem die Ausführung der Analyse und die Speicherung der Daten mithilfe eines Clusters parallelisiert werden.

3.2. Fahrzeugmessdaten

In dieser Arbeit soll die Nutzbarkeit von Apache Spark für die Analyse von Messdaten von Fahrzeugen evaluiert werden. Die zu analysierenden Daten liegen als multivariate Zeitreihen vor, die mithilfe von Spezialhardware im laufenden Betrieb aufgezeichnet und als MDF-Dateien abgespeichert werden. Multivariate Zeitreihen sind Zeitreihen mit vektorwertigen Messungen zu jedem Zeitpunkt, der Begriff steht im Gegensatz zu univariaten Zeitreihen, die nur eine Messgröße enthalten. In dieser Arbeit werden zusätzlich folgende Bezeichnungen verwendet: Zeitstempel bezeichnet einen oder mehrere Zeitmesswerte zu den Messungen in einer Zeitreihe, Wertereihe bezeichnet die Reihe der Messwerte einer univariaten Zeitreihe oder eine einzelne Reihe von Messwerten aus einer multivariaten Zeitreihe. Das zum Aufzeichnen der Daten genutzte Format wird in in Unterabschnitt 3.2.1 näher erläutert. An dieser Stelle soll zunächst allgemeiner auf die Form der Daten eingegangen werden.

Die Zeitstempel sind Fließkommazahlen mit Einheit Sekunde. Sie können äquidistant sein, müssen aber nicht. Übliche Abstände zwischen Messewerten sind 10 und 100 ms. Messungen und erzeugte Zeitreihen können Längen von wenigen Sekunden bis zu mehreren Tagen haben. Typischerweise sind Messungen unterschiedlich lang. Zu beachten ist, dass bei einer Messung, die Daten von einer Fahrt enthält, mehrere multivariate Zeitreihen mit unterschiedlichen Zeitstempeln parallel aufgezeichnet werden. Es besteht eine 1:n-Beziehung zwischen Messung und Zeitreihe. Die Beziehungen zwischen den Entitäten Fahrzeug, Messung, Zeitreihe und Wertereihe werden in Abbildung 3.1 in Form eines ER-Diagramms dargestellt. Die Werte der Zeitreihen können in zwei grundsätzliche Kategorien aufgeteilt werden: zum einen werden reelle Größen als Festkommazahlen gemessen, zum anderen Statuswerte als Integer mit beschränktem Wertebereich, im Extremfall 0 und 1.

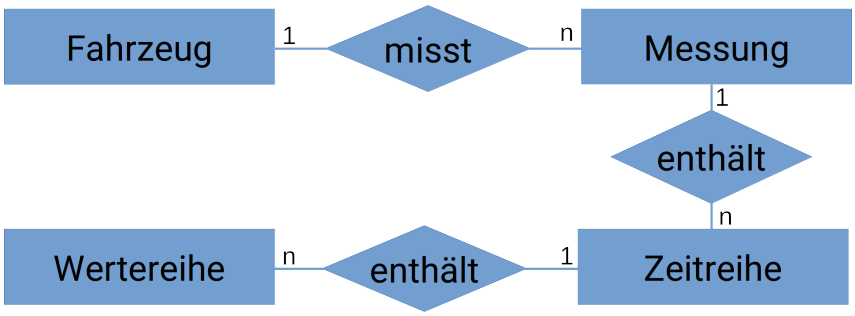


Abbildung 3.1. ER-Diagramm zu Messdaten

3.2.1. MDF

Zum genaueren Verständnis der zu verarbeitenden Daten wird die Spezifikation des verwendeten Messdatenformats zusammengefasst.

„Measurement Data Format (MDF) ist ein binäres Dateiformat für Messdaten, das 1991 von Vector in Zusammenarbeit mit der Robert Bosch GmbH entwickelt wurde. Nachdem das MDF-Format sich schnell zum Defacto-Standard in der Automobilindustrie entwickelte, wurde die überarbeitete Version 4.0 im Jahr 2009 schließlich als offizieller ASAM-Standard veröffentlicht. Das letzte Update des Formats erfolgte 2012 mit ASAM MDF 4.1.“ [Vec19].

MDF-Dateien bestehen aus durch Zeiger verbundenen Blöcken. So bildet sich eine Baumstruktur wie in Abbildung 3.2. Der Header-Block (HD) bildet die Wurzel des Baums. Von ihm aus führen Zeiger zu mindestens einem Data-Group-Block (DG). Jede Data-Group enthält genau einen Data-Block (DT) mit Nutzdaten und ein oder mehrere Channel-Groups (CG). Die Channel-Groups definieren, wie die

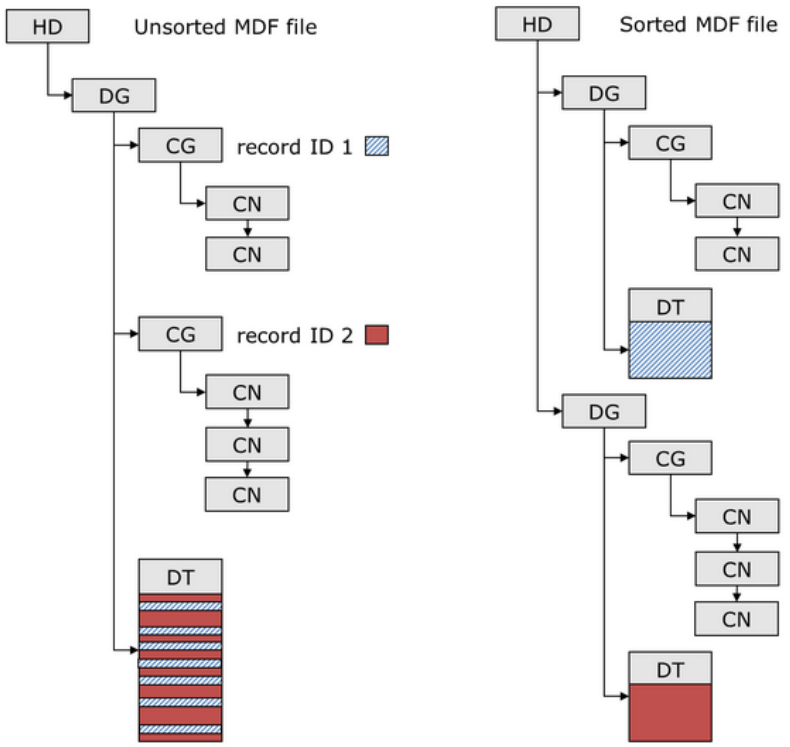


Abbildung 3.2. Block-Hierarchie einer MDF-Datei. Sortiert (links) und unsortiert (rechts). Quelle: [ASA19]

Daten im Data-Block zu interpretieren sind und geben die Zeitachse vor. Alle Channels (CN) in einer Channel-Group werden immer gleichzeitig aufgezeichnet und haben deshalb die gleichen Zeitstempel. Channels sind dabei Interpretierungsvorschriften für einen Teil der Daten in einer Channel-Group. Eine Channel-Group beschreibt somit die Speicherung einer multivariate Zeitreihe mit den Channels als Messgrößen.

Im Channel-Block (CN) werden Informationen wie Datentyp, Codierung, Bitlänge und Umwandlungsvorschrift gespeichert. Auch Namen und zusätzliche Informationen in Form von Strings können in Channels gespeichert werden. Mögliche Datentypen in MDF umfassen unter anderem Integer (1-64 bit), Float (32 oder 64 bit), String (beliebig viele Bytes, UTF-8, Null-terminiert) und komplexe Datentypen (z.B. Byte-Array). Umwandlungsvorschriften können zum Beispiel lineare Umrechnungen (Faktor * X + Offset), oder eine Übersetzung mit einer Wertetabelle (Statuswert 0 zu String „OK“) sein. Der gebräuchliche Anwendungsfall ist, dass mit einer Channel-Group ein Typ von CAN-Nachricht beschrieben wird. In CAN-Nachrichten von 8 Byte werden auf Bitebene viele einzelne Messwerte Codiert, wobei reelle gemessene Werte mit einem Wertebereich und einem Raster mittels einer linearen Funktion auf wenigen Bit gespeichert werden[1]. Diese CAN-Nachrichten können unverändert in MDF gespeichert werden und mithilfe der Regeln aus den Channel-Blocks gelesen und in menschenlesbare, physische Werte umgerechnet werden.

Der Data-Block enthält alle gemessenen Records in der Reihenfolge der Messung. Records sind die kleinste Dateneinheit in MDF und bestehen aus der ID der zugehörigen Channel-Group, einem Zeitstempel und den pro Channel gemessenen Datenwerten. Mithilfe der in Channel-Group und Channel gespeicherten Metadaten können die im Data-Block enthaltenen Records mit gleicher ID als multivariate Zeitreihe interpretiert werden. Ein Auslesen von durch einzelne Channels beschriebenen Teilen der Daten als univariate Zeitreihe ist zusätzlich möglich. Üblich sind Records mit fester Länge, Records mit

[1] Beispiel: Außentemperatur wird zwischen +100 und -100 °C angenommen. 0,5 °C ist ausreichend genau. Dann müssen 401 Werte unterschieden werden. Dafür benötigt man 9 Bit ($2^9 = 512 > 401$)

variabler Länge sind aber auch im Standard enthalten. Detailliertere Informationen sind in [ASA19] zu finden.

Zur deutlichen Beschleunigung des Lesens von MDF-Dateien sind im Standard die sortierten MDF-Dateien (siehe Abbildung 3.2 rechte Seite) definiert. Eine MDF-Datei ist sortiert, wenn in jedem Data-Block nur Records einer Data-Group vorkommen und deshalb alle Data-Blocks zeitlich sortiert sind. Die Sortierung ist erreichbar, indem pro Channel-Group eine Data-Group mit Data-Block erzeugt wird. In sortierten MDF-Dateien ist das Suchen nach Einträgen anhand der Zeit mit logarithmischer Laufzeit und das schnelle Auslesen von Channels und Channel-Groups möglich. Ebenfalls vorteilhaft ist, dass im Data-Block die Angabe der Channel-Group-ID entfällt, was Speicher spart.

Aufgrund der Speicherung von ganzen Records mit mehreren Channels pro Channel-Group ist MDF ein zeilenorientiertes Speicherformat. Es hat Vorteile von spaltenorientierter Speicherung, wenn sortierte MDF-Dateien verwendet werden, weil in einer sortierten MDF-Datei mehrere Tabellen gespeichert sind, auf die unabhängig voneinander zugegriffen werden kann.

Zusätzlich zu den genannten grundsätzlichen Möglichkeiten bietet die MDF4-Spezifikation noch weitere Elemente wie virtuelle oder zusammengesetzte Channels und unterschiedliche komplexe Datentypen. Detailliertere Informationen enthält die Website der ASAM [ASA19].

Channels werden im Folgenden als Kanal oder Messkanal bezeichnet.

3.3. Apache Spark

Apache Spark ist ein programmierbares Cluster-Computing-System, das auf MapReduce aufbaut und es erweitert. Zentrale Schnittstelle sind dabei die Resilient Distributed Dataset (RDD). Spark bietet APIs für Scala, Java, Python und R und zusätzlich interaktive Konsolen für Scala, Python, R und SQL. Neben „Spark Core“ gehören zu Apache Spark mittlerweile vier große Funktionsbibliotheken: Spark SQL, Spark Streaming, MLlib und GraphX [The19a; Zah+10]. Im Folgenden werden die RDDs (3.3.1) und die für die Arbeit wichtige integrierte Erweiterung Spark SQL (3.3.2) näher vorgestellt. Auf MLlib, Spark Streaming und GraphX wird hier nicht weiter eingegangen. Diese Erweiterungen sind für den Anwendungsfall nicht relevant, weil bereits abgeschlossene Messungen, die keine Graphen enthalten, analysiert werden und zunächst kein maschinelles Lernen verwendet wird. Abschließend wird in Unterabschnitt 3.3.3 auf Erweiterungen für Zeitreihenanalyse in Apache Spark eingegangen.

Spark-Anwendungen werden als Menge von unabhängigen Prozessen auf einem Cluster ausgeführt (Shared-Nothing-Architektur). Die Prozesse werden vom `SparkContext` des `Driver Program` gesteuert. Der `SparkContext` kann verschiedene Cluster-Manager nutzen, um auf den Worker Nodes im Cluster `Executors` zu starten, bei denen es sich um Prozesse handelt, die Daten für die Anwendung laden, verarbeiten und speichern. Wenn die `Executors` erstellt sind, sendet der `SparkContext` den Anwendungscode in Form von JAR- oder Python-Dateien an die Worker Nodes und versorgt die `Executors` mit Tasks in Form von Bytecode. Jeder `Executor` kann mehrere CPU-Kerne gleichzeitig nutzen um Tasks parallel auszuführen. Die Architektur eines Spark-Clusters ist in Abbildung 3.3 schematisch dargestellt.

Bei der Verwendung von Python als Programmiersprache für Apache Spark kommen zur Architektur die Python-Executors hinzu. Für jeden Executor startet PySpark einen Python-Interpreter pro zugeordnetem CPU-Kern. Zur Kommunikation zwischen dem Python-Driver, Spark und den Python-Executors wird `Py4J` [Dag18] genutzt und die Daten werden zur Übertragung zwischen den Prozessen serialisiert.

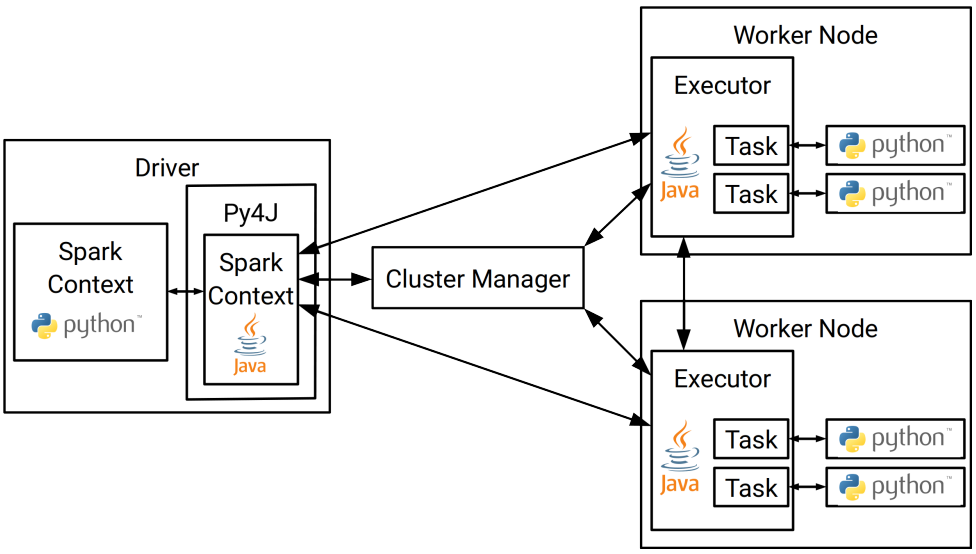


Abbildung 3.3. Überblick über ein Apache Spark Cluster: Das `Driver Program` steuert die Ausführung. Dazu nutzt es den `Cluster Manager`, um `Executors` zu starten, die den Analysecode in Form von `Tasks` ausführen. Bei Nutzung von PySpark muss zwischen Python und der JVM kommuniziert werden. Grafik angelehnt an: [The18b; RXO16] Teilbildquellen: [Afr11; www08]

3.3.1. Resilient Distributed Datasets

Resilient Distributed Dataset (RDD) sind partitionierte, schreibgeschützte Kollektionen von Einträgen [Zah+12] und die zentrale, grundlegende Schnittstelle von Apache Spark. Sie können aus Daten im Sekundärspeicher, oder aus anderern RDDs erstellt werden. Zur Programmierung werden auf die Kollektionen Transformationen und Aktionen angewendet, die elementweise arbeiten und aus der funktionalen Programmierung bekannt sind. Beispiele für Transformationen sind `map`, `filter` und `join`. Mögliche Aktionen sind unter anderem `reduce` und `count`. Die im Code angewendeten Transformationen werden erst ausgeführt, wenn eine Aktion die Auswertung erzwingt (lazy evaluation). Zur Umsetzung dieser Herangehensweise bildet Apache Spark Abstammungsgraphen, in denen gespeichert wird, wie das RDD aus der Datenquelle berechnet werden kann, bzw. welche Transformationen nacheinander angewendet werden [Zah+12].

Zur Ausführung wird das RDD in Partitionen unterteilt. Diese Partitionen ergeben sich aus der Datenquelle oder werden von Spark festgelegt. Sie werden auf die Knoten im Cluster verteilt, wobei auf Datenlokalität geachtet wird. Anschließend wird der Abstammungsgraph in Phasen unterteilt, deren Ausführungscode in JVM-Bytecode übersetzt und zur Ausführung an die Knoten versendet wird [Zah+12].

Bei der Bildung der Phasen werden so viele Transformationen wie möglich, die keinen Austausch von Daten zwischen den Partitionen erfordern, zusammengefasst, um sie nacheinander oder gleichzeitig auf die Zeilen des RDDs anzuwenden. Beispielsweise können konsekutive `map`- und `filter`-Operationen zusammengefasst werden [Zah+12].

Spark als System nutzt Datenparallelität, aber keine Aufgabenparallelität, weil unterschiedliche Datenströme auf den einzelnen Knoten gleichzeitig mit dem gleichen JVM-Bytecode verarbeitet werden, bis eine Phase abgeschlossen ist. Nach einem Shuffle beginnt die nächste Phase, in der die Knoten neuen Bytecode auf den umverteilten Daten ausführen.

Der Nutzer kann sowohl die Partitionierung als auch das Caching der RDDs beeinflussen. Wenn zwei RDDs passend partitioniert sind, kann zum Beispiel der Join dieser RDDs stark beschleunigt werden. Durch das explizite Caching eines mehrmals genutzten RDDs im RAM kann eine Analyse beschleunigt werden, weil das RDD nicht wiederholt berechnet werden muss und Daten nicht erneut gela-

```
1 lines = sc.textFile("data.txt")
2 lineLengths = lines.map(lambda s: len(s))
3 totalLength = lineLengths.reduce(lambda a, b: a + b)
```

Programmcode 3.1 Beispielcode für Apache Spark

den werden. Wenn der RAM nicht ausreicht, nutzt Spark automatisch zusätzlich den Sekundärspeicher [Zah+12].

Die Erklärungen zu RDDs sollen mit einem Programmierbeispiel abgeschlossen werden, das inklusive Erläuterung aus der Dokumentation von Apache Spark entnommen ist [The18e]. Der in Programmcode 3.1 gegebene Code erzeugt in der ersten Zeile über den Spark Context zeilenweise ein RDD aus der Datei `data.txt`. In Zeile zwei wird das RDD `lineLengths` erzeugt, indem auf jede Zeile des ursprünglichen RDDs die Funktion `lambda s: len(s)` angewendet wird. Anschließend werden alle Zeilenlängen mit `reduce` zur Gesamtlänge addiert. Dabei ist zu beachten, dass die Zeilen 1 und 2 aufgrund der Lazy Evaluation keine Berechnung in Spark auslösen. Diese wird durch die Aktion `reduce` in Zeile 3 ausgelöst.

3.3.2. Spark SQL

Spark SQL erweitert Apache Spark um die deklarative DataFrame API und die Möglichkeit, strukturierte Daten mit SQL zu analysieren [Arm+15]. Außerdem enthält Spark SQL den Anfrageoptimierer Catalyst. Die Entwickler sehen Spark SQL sowohl als Erweiterung für „SQL-on-Spark“, als auch als Erweiterung für Spark als ganzes, weil es neue, funktionsreichere APIs und weitreichendere Optimierungen einführt. Die zusätzlichen Bibliotheken erleichtern die Arbeit mit strukturierten, tabellarischen Daten, verbessern die Performance durch RDBMS-Techniken und vereinfachen die Anbindung neuer Datenquellen [Arm+15].

Im Folgenden wird näher auf die DataFrame API, den Optimierer Catalyst und die Data Source API zur Implementierung neuer Datenquellen eingegangen.

DataFrame API

„Die wichtigste Abstraktion in der API von Spark SQL ist das DataFrame, eine verteile Kollektion von Zeilen mit dem gleichen Schema. Ein DataFrame ist äquivalent zu einer Tabelle in einer relationalen Datenbank und kann wie RDDs manipuliert werden. Im Gegensatz zu RDDs verfolgen DataFrames Änderungen an ihrem Schema und unterstützen verschiedene relationale Operatoren, die für eine optimiertere Ausführung sorgen“ [Arm+15]. DataFrames könne sowohl aus externen Datenquellen wie relationalen Datenbanken, JSON-Dateien oder CSVs, als auch aus anderen RDDs erzeugt werden. Im Gegensatz zu den klassischen DataFrame-Implementierungen in R oder Pandas ist ein DataFrame-Objekte in Spark ein logischer Plan, der durch das Anwenden von verschieden Operatoren erzeugt wird. Unterstütze Operatoren sind zum Beispiel `select`, `where`, `groupBy` und `agg`. Die Auswertung des logischen Plans wird durch spezielle Aktionen wie `save` ausgelöst [Arm+15]. Neben der DataFrame-DSL gibt es durch Spark SQL die Möglichkeit, Anfragen an DataFrames in SQL zu schreiben. Alle wichtigen SQL-Typen werden von Spark SQL unterstützt. Zusätzlich gibt es Unterstützung für UDTs und UDFs [Arm+15].

Seit Spark 1.6.0 gibt es neben der untypisierten DataFrame-API für Scala und Java die Typisierte Dataset-API, die Typüberprüfung durch den Compiler und weitere Performanceoptimierungen ermöglicht. Seit

Spark 2.0 ist das DataFrame in Scala und Spark ein Typalias für `Dataset[Row]`. Da Python eine untypisierte Sprache ist und als solche keine Datasets unterstützt [Dam16], ist dies für die Arbeit nicht relevant. DataFrames werden spaltenweise gespeichert, um bessere Komprimierung und Analyseperformance zu erreichen. Trotzdem können sie als `RDD[Row]` zeilenweise genutzt werden, was Spark durch Ad-Hoc-Umwandlung in Zeilenform umsetzt [Arm+15].

Catalyst Optimierer

Catalyst ist der Optimierer von Spark SQL. Kern von Catalyst sind die Anfragepläne in Form von abstrakten Syntaxbäumen der DataFrames, die durch Regeln umgeformt werden. Diese Regeln sind als Scala-Funktionen mit Pattern Matching implementiert. Sie sind partiell und werden wiederholt angewandt, bis ein Fixpunkt erreicht ist. Wenn alle Regeln angewendet wurden, wird eine physische Planung mit verschiedenen Operatorbäumen und einfachen Statistiken wie Minimum, Maximum und Menge der unterschiedlichen Werte durchgeführt. Nachdem die Planung abgeschlossen ist, wird wie bei den RDDs JVM-Bytecode erzeugt und an die Worker geschickt [Arm+15]. Dabei werden seit einiger Zeit zunehmend SIMD-Vektoroperationen von aktuellen Prozessoren ausgenutzt, indem Berechnungen möglichst spaltenweise durchgeführt werden [XR15]. Bei der Optimierung von Anfrageplänen kann Catalyst auf die Datenquellen zurückgreifen und zum Beispiel Selektionen von der Datenquelle ausführen lassen. Zur einfachen Implementierung von Datenquellen gibt es die Data-Source-API.

Data-Source-API

Die Data-Source-API definiert vier unterschiedliche Interfaces, die von Datenquellen implementiert werden können: TableScan, PrunedScan, PrunedFilteredScan und CatalystScan. Datenquellen sind im Kontext der Data-Source-API Klassen, die über ihre Methoden Daten in Form von DataFrames bereitstellen. TableScan ist das einfachste Interface und schreibt zwei Funktionen vor: Die Möglichkeit, ganze Tabellen einzulesen und deren Größe in Bytes abzuschätzen. Ist ein Speichersystem zu Projektionen fähig, implementiert die zugehörige Datenquelle die PrunedScan-Schnittstelle, werden zusätzlich Filter unterstützt, wird das PrunedFilteredScan-Interface Implementiert. Beim CatalystScan werden ganze Teilbäume des Anfrageplans an die Datenquelle abgegeben [Arm+15].

Beispiele für in Spark SQL vorhandene Datenquellen sind der CSV-Reader, der die TableScan-Schnittstelle implementiert, der Parquet-Reader, der Projektion und Selektion ermöglicht, und JDBC-Datenquellen, die neben Projektion und Selektion das gleichzeitige Scannen von Abschnitten der gleichen Tabelle ermöglichen [Arm+15].

3.3.3. Zeitreihen in Apache Spark

Weil in dieser Arbeit die Analyse von Fahrzeug-Messreihen mit Apache Spark evaluiert werden soll, wurden die Bibliotheken „Spark-Timeseries“ und „flint“ untersucht. Sie erweitern Spark um Datenstrukturen und Algorithmen für die Verarbeitung von Zeitreihen. Beide nutzen ein unterschiedliches Datenmodell und sind für unterschiedliche Anwendungsfälle geeignet.

times:	[1970.01.01 01:00:00.1, 1970.01.01 01:00:00.2 ...]
0	[0.4, 0.4, ...]
1	[1.0, 1.1, ...]
2	[6.0, 5.0, ...]
3	[0.3, 0.4, ...]
4	[0.1, 0.0, ...]
...	...

Abbildung 3.4. Beispiel für TimeSeriesRDD von Spark-TS

Spark-Timeseries

Das Projekt Spark-Timeseries [Ryz19] wurde Anfang 2015 auf Github begonnen und wird seit April 2017 nicht mehr gepflegt. Die Bibliothek ist für die Verarbeitung von Finanzdaten entwickelt worden und richtet sich nach diesem Anwendungsgebiet aus. Spark-Timeseries erweitert Apache Spark um das TimeSeriesRDD. Das TimeSeriesRDD, beispielhaft in Abbildung 3.4, ist ein RDD, welches in jeder Zeile eine Reihe von Werten enthält. Die Reihe von Zeistempeln ist getrennt von den Werten und gilt für alle Zeitreihen. Das TimeSeriesRDD ist optimiert für die Arbeit mit sehr vielen, relativ kurzen Zeitreihen, die gleich lang sind und die gleichen Zeitstempel haben.

Neben TimeSeriesRDDs und Funktionen zum Erstellen und Manipulieren von solchen, bringt Spark-Timeseries eine Reihe von Machine-Learning-Modellen für Zeitreihen mit.

flint

Die Bibliothek flint [Two19] ergänzt Spark um ein TimeSeriesRDD und ein TimeSeriesDataFrame. Dabei verarbeitet flint Zeitreihen im Vergleich zu Spark-Timeseries transponiert, d.h. in jeder Zeile des TimeSeriesRDDs oder des TimeSeriesDataFrame befindet sich ein Zeitwert und ein oder mehrere Messwerte. Dadurch eignet sich flint für die Verarbeitung von extrem langen Zeitreihen, die durch Spark über mehrere Clusterrechner verteilt werden. Abbildung 3.5 stellt ein TimeSeriesDataFrame dar. Neben der Datenstruktur stellt flint eine Reihe von Higher-Order-Functions zur Arbeit mit TS-RDDs und TS-DFs bereit. Darunter sind Funktionen für Gruppierung, Window-Funktionen, temporale Joins und Funktionen für zusammenfassende Statistiken (summarizers).

time	value	Weitere Spalten...
1970.01.01 01:00:00.1	0.3	...
1970.01.01 01:00:00.2	0.5	...
1970.01.01 01:00:00.3	0.4	...
1970.01.01 01:00:00.4	0.7	...
1970.01.01 01:00:00.5	0.1	...
...	...	...

Abbildung 3.5. Beispiel für TimeSeriesDataFrame von flint

3.4. Datenbanksysteme

In den folgenden Abschnitten werden die in dieser Arbeit verwendeten Datenbanksysteme mit ihren zentralen Ideen vorgestellt. Bei der Auswahl wurde auf die Abbildung verschiedener Klassen von Systemen geachtet. Es werden klassisch relationale Datenbanken, spaltenorientierten Datenbanken und NoSQL-Systemen betrachtet. Einen Überblick über die Systeme gibt Tabelle 3.1.

System	Zentrale Eigenschaften
MySQL	relational, nicht verteilt
Postgres-XL	relational, verteilt
Vector	relational, spaltenorientiert, verteilt
HBase	NoSQL, verteilt

Tabelle 3.1. Übersicht über Verwendete Datenbanksysteme und deren zentrale Eigenschaften

3.4.1. MySQL

MySQL ist das am weitesten verbreitete Open Source-RDBMS. Vor allem in der Webentwicklung ist es eine Standardanwendung und wird im „LAMP“-Stack (Linux, Apache, MySQL, PHP) mit aufgezählt. MySQL wird seit 1994 entwickelt und gehört seit 2010 der Firma Oracle. Es gibt MySQL unter zwei Lizenzen. Neben dem freien MySQL unter GPL gibt es das durch Oracle vertriebene MySQL Enterprise mit geschlossenem Quellcode. Im April 2018 ist nach zweieinhalbjähriger Entwicklungszeit die aktuelle Version 8.0 erscheinen, die unter Anderem Window Functions und JSON-Verarbeitung hinzugefügt hat. Eine Besonderheit an MySQL ist, dass die Storage Engine austauschbar ist und unterschiedliche Storage Engines verschiedene Vor- und Nachteile mit sich bringen [Wik19b].

Im Rahmen der Arbeit wird MySQL verwendet, um ein weit verbreitetes, einfaches, spaltenorientiertes, relationales Datenbanksystem mit im Vergleich zu haben.

3.4.2. Postgres-XL

Postgres-XL ist ein Fork von PostgreSQL, der das Open Source-RDBMS zu einem verteilten System erweitert. Dazu wurden die drei Komponenten Global Transaction Manager, Koordinator und Data Node eingeführt, die über mehrere Maschinen verteilt werden können. Der Global Transaction Manager verwaltet die Transaktionen des Systems über die Vergabe von Transaktions-IDs und „Multi-version Concurrency Control“ [The18h]. Die Transaktionen werden von den Koordinatoren durchgeführt. Die Koordinatoren nehmen Anfragen von Client-Anwendungen entgegen, lassen den Global Transaction Manager eine Transaktions-ID erstellen, verteilen die Anfrage auf die Data Nodes, sichern per Two-Phase-Commit ab, dass alle beteiligten DataNodes die Anfrage erfolgreich abgearbeitet haben und fassen die Ergebnisse der Data Nodes zusammen. Auf den Data Nodes liegen die eigentlichen Daten [The18h; The18f].

Tabellen können in Postgres-XL partitioniert oder repliziert werden. Dabei können beliebig viele vom Administrator bestimmte Data Nodes beteiligt sein. Als Partitionierungsarten stehen Round Robin, Hash, Modulo und Random zur Verfügung [The18g].

Für Anwendungsprogramme sind die Änderungen von Postgres-XL im Vergleich zu PostgreSQL transparent. Die Koordniatoren haben die gleiche Benutzer-Schnittstelle wie ein gewöhnliches PostgreSQL und zerlegen die Anfrage intern in mehrere Anfragen für die Data Nodes. Dabei werden unter anderem Projektionen, Selektionen, Joins und Sortierungen verteilt abgearbeitet [The18f].

3.4.3. Vector in Hadoop

Actian Vector in Hadoop ist eine mit dem HDFS verteilte Version von Actian Vector, einer spaltenorientierten, relationalen SQL-Datenbank, die auf OLAP-Anwendungen spezialisiert ist [Act19a]. VectorWise (der ursprüngliche Name von Actian Vector) ist im Rahmen des X100-Projekts entstanden, und wurde nach dem Kauf durch Ingres um das Ingres-SQL-Frontend erweitert [ZB12]. Durch das

spaltenorientierte Verarbeiten und die dadurch möglichen Optimierungen wie die Nutzung von SIMD wird eine sehr hohe Performance bei analytischen Anfragen ermöglicht. Actian Vector stellt eine JDBC-Schnittstelle bereit, über die Daten von Spark geladen werden können. Zusätzlich gibt es den `Spark-Vector connector` [Act19b; Bar16], der das parallele Lesen aus und Schreiben nach Vector ermöglicht.

Im Vergleich der Datenbanksysteme in dieser Arbeit ist Actian Vector wegen der spaltenorientierten Speicherung und Anfrageverarbeitung vertreten. Die Daten werden in HDFS verteilt gespeichert und parallel verarbeitet. Actian Vector ist das einzige ausschließlich kommerzielle, getestete Produkt.

3.4.4. HBase

Apache HBase ist ein nach Google Bigtable modellierter Wide-Column-Store. Es zählt zu den NoSQL-Systemen im ursprünglichen Sinne. Eine Tabelle in HBase ist eine dreidimensionale Map der Form

$$\{row, column, timestamp\} \rightarrow string$$

Die Daten werden in Zeilen gespeichert die einige Spaltenfamilien enthalten, die beliebig viele Spalten enthalten dürfen. Zu jedem $(row, column)$-Paar können mehrere Werte mit unterschiedlichen Zeitstempeln gespeichert werden, wobei der neuste Wert in der Ordnung an erster Stelle steht und standardmäßig verwendet wird [Apa19]. Die Verteilung von Tabellen auf mehrere Maschinen erfolgt in HBase nach dem Range-Verfahren auf Basis der Zeilenschlüssel. Dabei sind die Zeilenschlüssel immer lexikographisch sortiert. Eine der Wichtigsten Optimierungen ist deshalb, oft gemeinsam verwendeten Zeilen Schlüssel zu geben, die lexikographisch nah beieinander liegen. Es ist zu beachten, dass die Speicherung der Tabellen sich nach den Spaltenfamilien richtet. Alle Spalten einer Familie werden gleich codiert, gemeinsam komprimiert und gespeichert [Apa19].

Im Gegensatz zu MySQL, Postgres-XL und Vector in Hadoop bietet HBase keine deskriptive SQL-Schnittstelle an. Der Zugriff auf die Daten erfolgt stattdessen über eine prozedurale Schnittstelle mit den den Low-Level-Operationen `Get`, `Put`, `Scan` und `Delete` [Apa19]. Es gibt Bibliotheken zum Laden von Daten aus HBase in Apache Spark, welche diese Operationen parallelisieren.

3.5. Speicherformate

Weil Apache Spark als „analytics engine“ [The19a] in der Lage ist, unterschiedliche Dateitypen einzulesen und auszuwerten, werden im Folgenden verschiedene Dateiformate für Messdaten vorgestellt.

3.5.1. CSV

Comma Seperated Values (CSV) ist ein textbasiertes, nicht binäres, zeilenorientiertes Speicherformat für tabellarische Daten. Es ist sehr variabel und es gibt keinen einheitlichen Standard [Sha05]. In CSV-Dateien werden Zeilen durch CRLF getrennt und die Spaltenwerte in den Zeilen werden durch Komma getrennt. Zusätzlich ist es möglich, LF zur Zeilentrennung und Semikolon oder Doppelpunkt für die Spaltentrennung zu nutzen. Alle Zeilen sollten die gleiche Anzahl an Werten enthalten, dies ist aber optional. Die erste Zeile kann ein Header sein, der die Spaltennamen angibt. Werte können optional in doppelten Anführungszeichen eingeschlossen werden. Wenn Werte Zeilenumbrüche oder Spaltentrenner (normalerweise Kommata) enthalten sollen, müssen sie in Anführungszeichen stehen [Sha05].

```
Stunde,Montag,Dienstag,Mittwoch,Donnerstag,Freitag
1,Mathematik,Deutsch,Englisch,Mathematik,Kunst
2,Sport,Französisch,Geschichte,Sport,Geschichte
3,Sport,"Religion (ev, kath)",Kunst,,Kunst
```

Stunde	Montag	Dienstag	Mittwoch	Donnerstag	Freitag
1	Mathematik	Deutsch	Englisch	Mathematik	Kunst
2	Sport	Französisch	Geschichte	Sport	Geschichte
3	Sport	Religion (ev & kath)	Kunst		Kunst

Abbildung 3.6. Beispiel für eine CSV-Datei als Quellcode und in tabellarischer Darstellung. Quelle: [Wik18]

Großer Vorteil von CSV-Dateien ist die Lesbarkeit für Menschen und die verbreitete Nutzung. Nachteile für den Anwendungsfall sind die fehlende Kompression, die ungeeignete Codierung von Zahlen als Zeichen und das Fehlen von spaltenweisem Lesen. Des Weiteren ist es kompliziert, die unterschiedlichen Zeitreihen einer Messung in eine CSV-Datei zu schreiben. Eine Messung müsste aus vielen einzelnen Dateien bestehen, was die Handhabbarkeit deutlich verschlechtert.

3.5.2. Apache Parquet

„Apache Parquet ist ein spaltenorientiertes Speicherformat [...]“ [The18a], das von Spark SQL gelesen und geschrieben werden kann [The18d]. Es ist ein offenes Format, das von vielen Frameworks im Hadoop Ökosystem genutzt und unterstützt wird und baut auf Algorithmen auf, die im Rahmen von Dremel [Mel+10] entwickelt wurden.

Für Spark werden die Daten in Parquet-Dateien im HDFS abgelegt. Zusätzlich ist es möglich, die Daten in einer Ordnerstruktur anhand von einzelnen Spalten zu partitionieren [The18d]. Diese Partitionierung wird von Spark als Index verwendet. In dieser Arbeit soll Parquet als Basiswert für die Performance von Apache Spark verwendet werden, weil Spark die Dateien von sich aus, ohne Erweiterungen oder ein Datenbanksystem lesen und auswerten kann. Es stellt sich die Frage, welchen Vorteil die Verwendung der anderen getesteten DBMS gegenüber dem Einsatz von Parquet-Dateien hat.

3.5.3. MDF

MDF, das in Unterabschnitt 3.2.1 vorgestellt wurde, ist ein erprobtes und ausgereiftes Format für das Speichern von Messdaten, vor allem im Automotive-Bereich. Es gibt keine öffentlich verfügbare Unterstützung von MDF in Apache Spark. Weil das Schreiben eines parallelisierten Readers für MDF den Rahmen der Arbeit übersteigt, wird auf die Nutzung des Formats verzichtet. Im Projekt werden die MDF-Dateien über einen Python-basierten Reader ausgelesen und in andere Formate umgewandelt, oder in ein Datenbanksystem importiert.

3.6. Vorgeschlagene Architekturen

In der Literatur wurden verschiedene Architekturen zur Datenauswertung in der Automobilindustrie vorgeschlagen. Einige sollen an dieser Stelle kurz vorgestellt werden.

3.6.1. Automotive Big Data

A. Luckow et al schlagen in ihrem Paper „Automotive Big Data“ [Luc+15] die Data-Lake-Architektur (siehe Abbildung 3.7) vor, die verschiedene Big-Data-Anwendungen in der Automobilindustrie unterstützen soll. Die Architektur besteht aus fünf Schichten, von denen die oberen vier für diese Arbeit von Interesse sind. Die Autoren gehen davon aus, dass Anwendungen normalerweise über SQL oder die DataFrame-Schnittstelle auf den Data Lake zugreifen. Deshalb untersuchen sie, wie in der Abbildung dargestellt Systeme eine SQL- oder DataFrame-Schnittstelle für in Hadoop gespeicherte Daten bereitstellen können. Dabei wird Apache Spark genauer untersucht, weil es beide Schnittstellen für Anwendungen bereitstellen kann und die Programmiersprachen Python und R unterstützt. Zudem fällt auf, dass die Luckow et al. klassische Relationale Datenbanksysteme in den Schichten neben und nicht als Datenquelle für Apache Spark sehen.

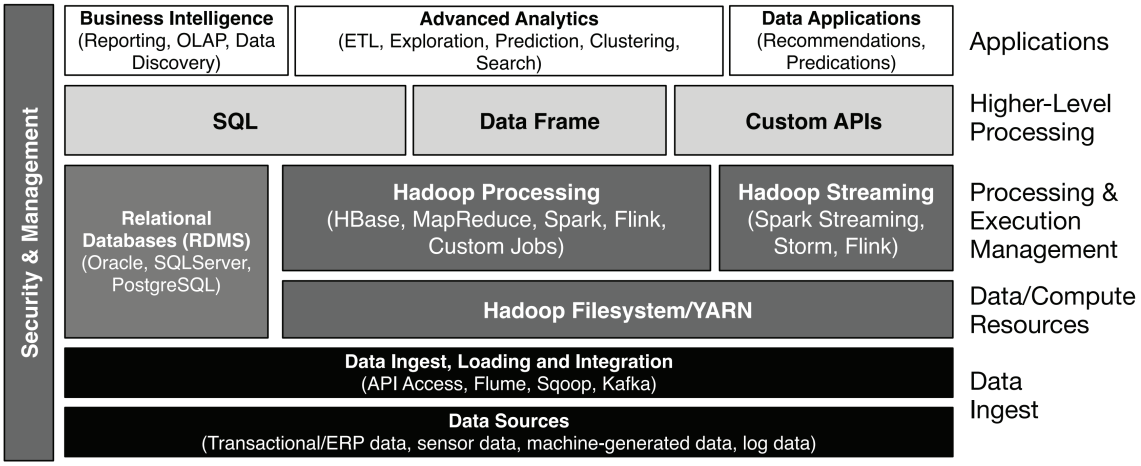


Abbildung 3.7. Data Lake nach A. Luckow et al [Luc+15].

3.6.2. A Big Data Architecture for Automotive Applications

Haroun et al. [HMD17] stellen eine Architektur vor, deren Fokus auf Streaming-Daten liegt, und die einen Batch-Layer enthält (siehe Abbildung 3.8). Dieser nutzt zur Speicherung das verteilte Dateisystem GPFS mit einem Connector, der es Hadoop-kompatibel macht. Darauf aufbauend wird „BigInsights“ von IBM mit dem Produkt „Big R“ verwendet. Big R ermöglicht es, in R geschriebene Funktionen als Map-Reduce-Jobs auszuführen und ähnelt somit Apache Spark.

3.7. Kommerzielle Angebote für Analyse-Umgebungen

Es existieren eine Reihe von Platform as a Service-Angeboten, die Big-Data-Stacks anbieten. An dieser Stelle sollen Angebote für Apache Hadoop und Apache Spark vorgestellt werden.

Hortonworks bietet mit der „Hortonworks Data Platform“ ein umfangreiches Portfolio von Open Source Software für den Einsatz im Bereich Big Data und Maschine Learning an. Zentral ist dabei Apache Hadoop, als Software für den Bereich Data Science werden Apache Spark und Apache Zeppelin angeboten, HBase dient als „Operational Data Store“ [Hor18].

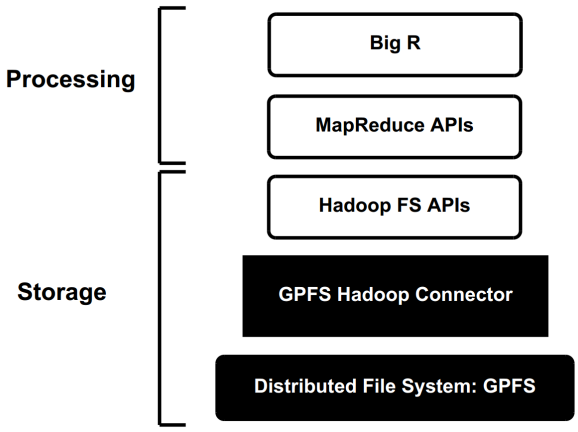


Abbildung 3.8. Batch Layer nach A. Haroun et al.
Quelle: [HMD17]

Cloudera hat ein ähnliches Angebot, auch hier bildet Hadoop mit einigen Zusätzen die Basis, welche um Apache Spark mit seinen Analysefunktionen und ML-Features ergänzt wird. Apache HBase ist eine zusätzlich konfigurierbare Option.

Die Entwickler von Apache Spark arbeiten heutzutage zum großen Teil in der von ihnen gegründeten Firma Databricks, die Apache Spark as a Service unter dem Namen „Databricks Unified Analytics Platform“ anbietet [Dat19a]. Die Software läuft dabei entweder auf Amazon Web Services oder Microsoft Azure. Zentrale Entwicklerschnittstelle sind Notebooks ähnlich zu Apache Zeppelin oder Jupyter. Das erweiterte und um zusätzliche Software angereicherte Spark wird hier unter dem Namen „Databricks Runtime“ als Service zur Nutzung vermietet.

Auf den Vertrieb von Dienstleistungen zu Apache Cassandra hat sich das Unternehmen DataStax spezialisiert. Hier wird Apache Spark als Analyseumgebung verwendet [Dat19b].

Mit „IBM Cloud SQL Query“ [IBM] gibt es auch ein Angebot für Apache Spark von einem traditionellen RDBMS-Anbieter. Hier wird Spark in Kombination mit IBMs „Cloud Object Storage“ und „Watson Studio“ beworben.

Zusammenfassung

In diesem Kapitel wurden die Struktur der zu analysierenden Daten und alle für die Arbeit relevanten Systeme vorgestellt. Zudem wurden Ansätze aus der Forschung und kommerziell vertriebene verteilte Analyseumgebungen vorgestellt. Im nächsten Kapitel wird auf Basis der Data-Lake-Architektur ein Softwarestack für die Analyse von Fahrzeugmessdaten in Apache Spark mit unterschiedlichen Speichersystemen vorgestellt. Anschließend wird der Fokus auf Schemata zur Verarbeitung der vorgestellten Daten in Apache Spark gelegt (4.2.2). Für Postgres-XL und Apache Parquet werden Funktionen implementiert, die gespeicherte Messdaten abrufen. Die Implementierung wird in Unterabschnitt 4.2.3 konzipiert und in Unterabschnitt 5.2.3 vorgestellt. Die vorgestellten kommerziellen Produkte werden nicht verwendet. Die Systeme werden Lokal mit den frei verfügbaren Versionen betrieben.

4. Konzepte

In diesem Kapitel werden Anforderungen und Konzepte für die Analyse von Fahrzeugmessdaten mit Apache Spark gesammelt und entwickelt. Zunächst wird als Überblick der verwendete Software-Stack vorgestellt, der Analyse von Daten aus unterschiedlichen Speichersystemen mit Apache Spark unterstützt. Anschließend werden in Abschnitt 4.2 Anforderungen für die Messdatenanalyse mit Apache Spark zusammengefasst, Schemata zur Verarbeitung von Messdaten mit Apache Spark entwickelt, verglichen und evaluiert und Transformationen zwischen verschiedenen Schemata vorgestellt. In Abschnitt 4.3 werden Kriterien für die Auswahl des optimalen Speichersystems entwickelt und in Abschnitt 4.4 Benchmarks zur Bewertung des Systems konzeptioniert.

4.1. Software-Stack

Um die zentralisierte, parallelisierte Analyse von Messdaten zu ermöglichen, wurde die in Abbildung 4.1 dargestellte, fünfschichtige Architektur entwickelt, die sich an der in Unterabschnitt 3.6.1 vorgestellten Architektur orientiert. Die unterste Schicht „Datenintegration“ ist für diese Arbeit nicht relevant. Es wird angenommen, dass die Integration der Daten in den Speichersystemen abgeschlossen ist. Die darauf aufbauende Schicht der Datenspeicherung betrifft diese Arbeit. Es sollen Schnittstellen entwickelt werden, die das Abrufen der Daten aus dem Speicher mit Apache Spark ermöglichen, sodass Spark DataFrames auf Basis der gespeicherten Messdaten erstellen kann. Die in Abschnitt 4.2 vorgestellten Datenstrukturen, Algorithmen und Funktionen werden als Funktionsbibliothek für Apache Spark mit der DataFrame-API implementiert. Die Möglichkeit, mithilfe von Spark SQL-basierte Auswertungen unabhängig von der Datenspeicherung anzubieten wird hier nicht evaluiert. Auf der Anwendungsebene befinden sich im Forschungskontext die von Fahrzeugingenieuren entwickelten Analysealgorithmen. Dabei nutzen sie eine Notebook-Weboberfläche für Python-Programmierung, die im Hintergrund die PySpark Shell ausführt. So ergibt sich ein System, das es ermöglicht, im Browser Analysen zu entwickeln und diese auf einem Apache-Spark-Cluster auszuführen.

Die Speichersysteme und deren Schemata aus Schicht zwei wurden nicht im Rahmen dieser Arbeit entwickelt. Kern sind die Funktionsbibliotheken und die Anbindung der Speichersysteme an Spark. Die Benutzbarkeit über eine Weboberfläche mit Codevervollständigung und Möglichkeiten zur grafischen Darstellung von Ergebnissen ist ein Teilaspekt, der durch die Verwendung von Jupyter Notebooks hinreichend gelöst werden kann und nicht näher betrachtet wird.

Als Speichersysteme sind die Systeme aus Abschnitt 3.4 und Apache Parquet angedacht. So wird ein Querschnitt über unterschiedliche Ansätze zur Datenspeicherung abgebildet. Mit MySQL, Postgres-XL und VectorH sind drei Systeme vertreten, die per SQL-Schnittstelle genutzt werden. Alle drei Systeme unterscheiden sich in der Verwaltung des Speichers und in der Verteilung. MySQL ist ein klassisches zeilenorientiertes und nicht verteiltes Datenbanksystem, Postgres-XL ist zeilenorientiert und verteilt, VectorH ist verteilt und spaltenorientiert. Neben den SQL-Systemen ist mit Apache HBase ein NoSQL-System, das horizontal skaliert, sowie mit Apache Parquet ein reines Dateiformat, das von Spark gelesen werden kann, mit im Vergleich enthalten.

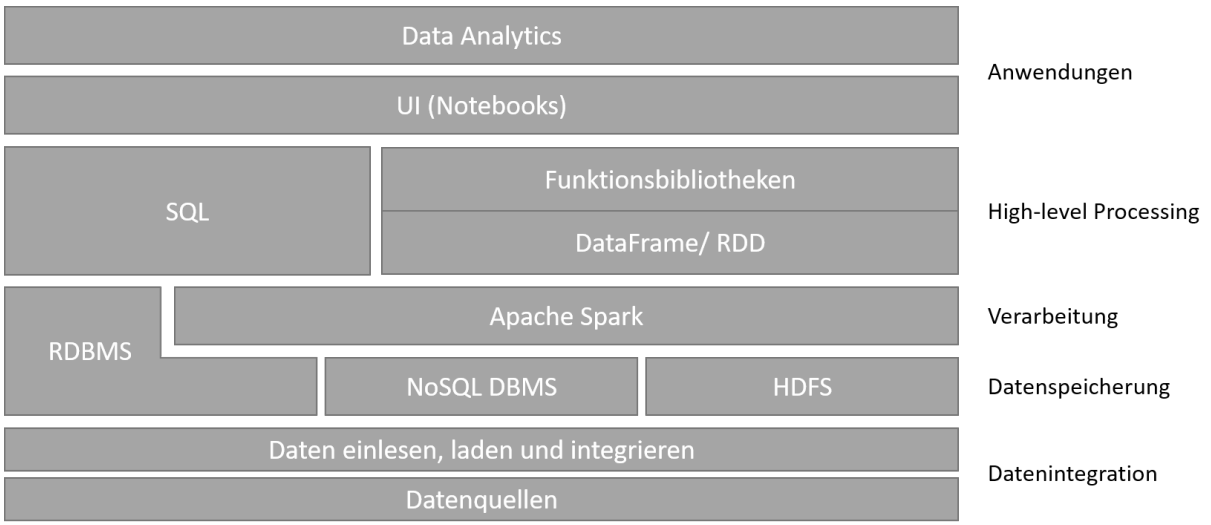


Abbildung 4.1. Softwarestack

4.2. Analyse von Messdaten in Apache Spark

In diesem Abschnitt werden Funktionsbibliotheken, die die Analyse von Fahrzeugmessdaten in Apache Spark ermöglichen sollen, konzeptionell entwickelt. Dafür werden in Unterabschnitt 4.2.1 Anforderungen gesammelt und anschließend in Unterabschnitt 4.2.2 Nutzung von RDDs und DataFrames evaluiert und unterschiedliche Schemata und Transformationen zwischen diesen erläutert. Des Weiteren werden Anforderungen für Ladefunktionen (4.2.3) und Basisanalyseoperationen (4.2.4) vorgestellt. In Kapitel 5 wird gezeigt, wie die hier entworfenen Funktionen implementiert sind.

4.2.1. Anforderungen

Für die Analyse von Messdaten mit Apache Spark gibt es eine Reihe von Anforderungen, die in drei Kategorien aufgeteilt werden: Anforderungen der Entwickler, bzw. Analysten, Anforderungen durch die Struktur der Daten und Anforderungen der Algorithmen, die regelmäßig ausgeführt werden sollen.

Anforderungen der Entwickler Eine der zentralen Anforderungen des Projekts ist der Wunsch der Analysten, möglichst wenig neue Sprachen und Konzepte lernen zu müssen und trotzdem von Parallelisierung zu profitieren. Deshalb konzentriert sich diese Arbeit mit Python auf eine der verbreitetsten Programmiersprachen für Data Analytics. Außerdem sollen die Algorithmen, welche auf Zeitreihen angewendet werden, unabhängig davon, ob sie auf kompletten oder parallelisierten Zeitreihen ausgeführt werden, einheitlich entwickelt werden können. Die Parallelisierung soll möglichst transparent sein und das Ausführen der gleichen Analyse auf vielen Zeitreihen soll wenige Änderungen am Algorithmus erfordern.

Neben der Parallelisierung von Analysen ist der effiziente Zugriff auf die gespeicherten Messdaten eine wichtige Anforderung der Entwickler. Dieser wird durch Funktionen zum Laden ermöglicht. Konzepte zu solchen Funktionen behandelt Unterabschnitt 4.2.3.

Anforderungen durch die Datenstruktur Neben den Bedürfnissen der Entwickler gibt es Anforderungen, die aus den zu analysierenden Daten abgeleitet werden. Da in dieser Arbeit mit Messdaten von Fahrzeugen gearbeitet wird, muss das System die gleiche Analyse parallel auf unterschiedliche lange und unterschiedlich aufgelöste Zeitreihen, wie sie aus MDF-Dateien gelesen werden, anwenden können. Zusätzlich ist zu beachten, dass es zu ermöglichen ist, sehr viele und sehr lange Zeitreihen (mehrere Milliarden Werte, mehrere Terabyte) zu verarbeiten. Es wird in den meisten Fällen mit Experimentalzeit gearbeitet, die als Fließkommazahl in Sekunden vorliegt.

Anforderungen an Algorithmen Viele Algorithmen arbeiten mit mehreren Zeitreihen auf gleichem Zeitraster. Weil in den Messdaten Zeitreihen mit unterschiedlichsten Zeitrastern vorhanden sind, ist das Resampling von mehreren Zeitreihen auf eine einheitliche Zeitachse ein erforderlicher Standardalgorithmus. Dabei ist es möglich, eine neue Zeitachse vorzugeben, oder die Zeitachse einer beteiligten Messreihe als Basis zu nutzen.

4.2.2. Datenstrukturen

Apache Spark bietet zwei grundsätzliche Datenstrukturen: das RDD und die darauf aufbauenden und optimierten DataFrames. Im Folgenden werden verschiedene Schemata zur Datenspeicherung und -verarbeitung vorgestellt und bewertet. Zunächst werden die in Unterabschnitt 3.3.3 vorgestellten Spark-Erweiterungen evaluiert, anschließend werden im Rahmen dieser Arbeit entwickelte Schemata betrachtet.

Bibliotheken Die Erweiterungsbibliotheken für Zeitreihenanalyse eignen sich aufgrund ihres Datenschemas nicht für die Analyse der Fahrzeugmessdaten, wie sie in Abschnitt 3.2 vorgestellt wurden.

Spark-Timeseries eignet sich aus folgenden Gründen nicht zur Analyse von Fahrzeugdaten:

- Das Schema der TimeSeriesRDDs von Spark-Timeseries setzt voraus, dass alle verarbeiteten Zeitreihen die gleichen Zeitstempel nutzen und sich als multivariate Zeitreihe mit sehr vielen Messwerten pro Messzeitpunkt interpretieren lassen. Zu einer Messung gehören jedoch viele Messreihen mit unterschiedlichen Zeitstempeln und eine Vereinheitlichung dieser ist nicht bei jeder Analyse gewünscht und soll nicht zwingend über alle Messreihen vorgenommen werden.
- Die Datenstruktur von Spark-Timeseries erfordert, dass alle Zeitreihen die gleiche Länge haben. Das ist nicht gegeben.
- Einzelne Messungen in den Fahrzeugdaten können sehr lang sein. Dafür ist das TimeSeriesRDD nicht optimiert.
- Zusammenhänge zwischen bestimmten Zeitreihen wie z.B. „von Fahrzeug X“, können mit Spark-Timeseries nicht dargestellt werden.
- Spark-Timeseries unterstützt keine Experimentalzeit in Sekunden, wie sie bei Messreihen üblich ist.

Die Bibliothek flint ist wie spark-ts für den Anwendungsfall nicht gut geeignet:

- In dieser Arbeit sollen sehr viele kleine und große Zeitreihen parallel verarbeitet werden. Die Bibliothek flint ist für die Verarbeitung weniger, sehr großer Zeitreihen gedacht.
- Die Programmierung von Analysen für das Datenmodell von flint ist kompliziert, weil kein Index der Zeitreihe zur Verfügung steht.
- flint unterstützt keine Experimentalzeit in Sekunden, wie sie bei Messreihen üblich ist.

Weil sich die verfügbaren Bibliotheken nicht für die Verarbeitung der vorliegenden Daten eignen, wurden unterschiedliche Schemata für die Verarbeitung der Messreihen mit den Standard-Spark-APIs entworfen, die im Folgenden vorgestellt und evaluiert werden.

Arbeit mit RDDs Wenn RDDs für die Analyse der Zeitreihen verwendet werden sollen, muss definiert werden, welche Datenobjekte im RDD verarbeitet werden. Zur parallelisierten Verarbeitung von Zeitreihen in PySpark besteht das RDD aus Zeitreihen-Objekten, die Zeitstempel und Wertereihen als NumPy-Arrays oder Pandas-Series enthalten. Dazu kommen gegebenenfalls Metadaten, die im Zeitreihen-Objekt oder mit dem Zeitreihen-Objekt in einem Tupel stehen können. Funktionen arbeiten bei Verwendung solcher Schemata mit den Zeitreihen-Objekten oder Tupeln als Eingabe. Es wird die funktionale RDD-API verwendet. Standardalgorithmen können als Methoden der Klasse Zeitreihe implementiert werden. Die Optimierungen und SQL-ähnlichen Transformationen von Spark SQL (siehe Abschnitt 3.3.2) stehen bei diesem Ansatz nicht zur Verfügung.

Die Anforderungen werden durch dieses Schema weitgehend erfüllt. Die RDD-API steht in PySpark zur Verfügung, transparente Parallelisierung kann über Higher-Order-Functions umgesetzt werden, die unterschiedlichen Zeitreihen sind in den Zeitreihen-Objekten gekapselt und deshalb von Spark nicht optimiert und beeinflusst. Das Resampling und andere Standardalgorithmen können über Aggregationen und einmalig implementierte Funktionen umgesetzt werden.

PySpark hat eine spezielle Anforderung an auszuführenden Code. Die Klassen, die in RDDs verwendet werden, müssen bei allen Workern bekannt sein und können nicht in einem Notebook definiert werden. Dafür muss der Entwickler sorgen, indem er den Code paketiert und auf allen Workern installiert, was zusätzlichen Arbeitsaufwand erzeugt.

Arbeit mit DataFrames Im Folgenden werden verschiedene Ansätze für DataFrame-Schemata verglichen. Der Ansatz aus Abbildung 4.2 ist angelehnt an den RDD-Ansatz. Es wird eine Spalte mit dem User Defined Type (UDT) Zeitreihe genutzt. Weitere Spalten enthalten Metadaten wie Kanalname und Fahrzeug-ID.

id	Metadaten...	TS-Objekt
0	...	Array[Zeitstempel], Array[Datenwert]
1	...	Array[Zeitstempel], Array[Datenwert]
2	...	Array[Zeitstempel], Array[Datenwert]
3	...	Array[Zeitstempel], Array[Datenwert]
...	...	...

Abbildung 4.2. DataFrame Schema mit UDT

Vorteil dieses Ansatzes ist, dass UDFs speziell für den Typ Zeitreihe geschrieben werden und wichtige Funktionen als Methoden der Klasse Zeitreihe implementiert werden. Das sorgt für Kapselung und übersichtlichen Code. Dieser Ansatz ist nicht umsetzbar, weil die Implementierung von UDTs in Apache Spark zurzeit nicht offiziell unterstützt wird.

Ein einfacheres Schema enthält keinen UDT, sondern stattdessen die Zeitstempel und Wertereihen in Form von Arrays in einzelnen Spalten. Wie in Abbildung 4.3 dargestellt, enthält dieses Schema die Spalten `time` und `value`, sowie eine beliebige Anzahl von Spalten für Metadaten. Vorteil dieses Schemas ist, dass es in Apache Spark umsetzbar ist. Es ermöglicht die parallele Analyse der Messreihen mit zeilenweisen UDFs. Nachteil ist die fehlende Typisierung. Der Analyst muss beachten, dass die Spalten `time` und `value` bei Transformationen erhalten bleiben und nicht durch Projektion entfernt werden. Apache Spark unterstützt diese Überwachung des Schemas nicht.

Dieses Schema wird im Folgenden als Standardschema bezeichnet.

id	Metadaten...	time	value
0	...	Array[Zeitstempel]	Array[Datenwert]
1	...	Array[Zeitstempel]	Array[Datenwert]
...	...	...	...

Abbildung 4.3. Standardschema: eine Reihe von Zeitwerten pro Reihe von Datenwerten

Um die Analyse von multivariaten Zeitreihen zu unterstützen, wird ein zweites Schema verwendet, dessen `value`-Spalte den Typ `Array[Array[Datenwert]]` hat. So gilt für mehrere Wertereihen die gleiche Zeitachse und UDFs können wie beim vorherigen Schema zeilenweise verwendet werden. Eine nützliche Transformation in diesem Kontext ist das Resampling, beschrieben in Abschnitt 4.2.2. Bei diesem Schema ist es nötig, dem DataFrame ein Attribut `name` hinzuzufügen, das die Elemente des Arrays in `value` identifiziert. Es wird im Folgenden als Array-Schema bezeichnet.

id	Metadaten...	name	time	value
0	...	Array[String]	Array[Zeitstempel]	Array[Array[Datenwert]]
1	...	Array[String]	Array[Zeitstempel]	Array[Array[Datenwert]]
...	...	...	...	...

Abbildung 4.4. Array-Schema: Beliebig viele Reihen von Datenwerten, für die die gleiche Reihe von Zeiten gilt.

Wird eine Analyse mit einer festen Anzahl von Wertereihen pro Zeitreihe durchgeführt, kann das in Abbildung 4.5 dargestellte Schema mit mehreren Spalten für Werte verwendet werden. Vorteil dieses Schemas ist, dass die Werte durch ihren Spaltennamen identifiziert sind und kein zusätzliches Attribut zur Identifizierung nötig ist. Gleichzeitig ist es weniger flexibel, weil die Anzahl der Wertearrays für jede Zeile genau vorgegeben ist. Im Folgenden wird es als Wertespalten-Schema bezeichnet.

id	Metadaten...	time	value1	value2	...
0	...	Array[Zeitstempel]	Array[Datenwert]	Array[Datenwert]	...
1	...	Array[Zeitstempel]	Array[Datenwert]	Array[Datenwert]	...
...	...	...	...	...	...

Abbildung 4.5. Feste Anzahl von Wertereihen, für die die Zeiten gelten. Im Beispiel 2.

Um die Auswertung von sehr langen Zeitreihen zu parallelisieren, können diese wie weiter unten in Abschnitt 4.2.2 beschrieben zerlegt und auf mehrere Tupel aufgeteilt werden. Nach der Anwendung der Analysealgorithmen werden die Ergebnisse durch Gruppierung und Duplikateliminierung zum Endergebnis zusammengefasst. Ein DataFrame, das zerlegte Zeitreihen enthält, wird in Abbildung 4.6 dargestellt. Die `id` speichert die Zugehörigkeit zur Zeitreihe und der `index` die Position innerhalb der Zeitreihe. Arrays sind mit Überlappung auf die Tupel aufgeteilt. Das Schema wird als Standardschema bezeichnet, weil die Typen der Spalten `time` und `value` mit den Typen im Standardschema übereinstimmen.

Mit diesen Schemata und zusätzlich implementierten Transformationen zwischen ihnen werden ebenfalls alle oben genannten Anforderungen erfüllt. Zusätzlich stehen bei dieser Variante die Funktionen, Optimierungen und Datenquellen aus Spark SQL zur Verfügung und die einheitliche Struktur der Metadaten wird ausgenutzt. Deshalb ist die Nutzung von DataFrames der Nutzung von RDDs vorzuziehen [Dam16]. Im Folgenden werden die DataFrame-Schemata verwendet, die keinen UDT enthalten.

id	index	Metadaten...	time	value
0	0	...	[0.0, 0.1, 0.2, 0.3, 0.4]	[0.3, 0.5, 0.4, 0.7, 0.1]
0	1	...	[0.3, 0.4, 0.5, 0.6, 0.7]	[0.7, 0.1, 1.2, 1.0, 0.8]
0	2	...	[0.6, 0.7, 0.8, 0.9 ...]	[1.0, 0.8, 0.1, 0.3, ...]
...	...	...	...	...

Abbildung 4.6. DataFrame-Schema für aufgeteilte Zeitreihen

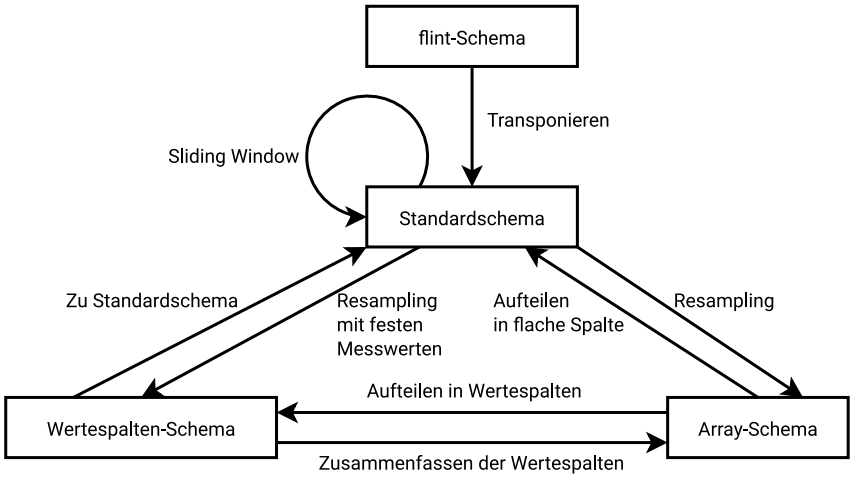


Abbildung 4.7. Verwendete Schemata und Transformationen zwischen diesen

Transformationen

Weil die Zeitreihen nicht aus jeder Datenquelle direkt im Standardschema geladen werden können und für unterschiedliche Analysen verschiedene Schemata benötigt werden, sind einige Standard-Transformationen zu implementieren. Einen Überblick über die Schemata und Transformationen gibt Abbildung 4.7. Die unterstützenden tabellarischen Grafiken sind vereinfacht. Zusätzliche Spalten sollen unterstützt werden.

Transponieren Das Transponieren eines DataFrames ist nötig, wenn eine Zeitreihen in einem DataFrame mit den Spalten `(Zeit, Wert, Zeitreihen-ID)` oder `(Zeit, Wert1, Wert2 ...)` vorliegen (flint-Schema). Durch das Transponieren wird das Standardschema aus 4.2.2 erzeugt. Dazu werden die Zeit-Wert-Paare nach Zeitreihen-ID gruppiert und anschließend sortiert. Weil dies eine teure Operation ist, sollte sie vermieden werden. Stattdessen sollte das Laden der Zeitreihen im Standardschema bevorzugt werden.

time	value	Key
0.0	0.3	0
0.1	0.5	0
0.2	0.4	0
0.0	0.7	1
0.3	0.1	0
0.15	0.1	1
...	...	...

$\rightarrow$

Key	time	value
0	[0.0, 0.1, 0.2, 0.3, ...]	[0.3, 0.5, 0.7, 0.1, ...]
1	[0.0, 0.15, ...]	[0.7, 0.1, ...]
...	...	...

Abbildung 4.8. Veranschaulichung Transponierung

Sliding Window Zur parallelisierten Anwendung von Analysen auf sehr lange Zeitreihen können diese mit einem Sliding Window in mehrere kleinere Zeitreihen in mehreren Zeilen zerlegt werden. Diese Ope-

ration hat zwei Parameter: die Fenstergröße und die Schrittweite. Die Fenstergröße bestimmt die Anzahl der Werte in einem Fenster. Die Schrittweite gibt an, um wie viele Indizes das Fenster verschoben wird. In Abbildung 4.9 gilt `Fenstergröße = 5, Schrittweite = 3`.

id	time	value
0	[0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9 ...]	[0.3, 0.5, 0.4, 0.7, 0.1, 1.2, 1.0, 0.8, 0.1, 0.3, ...]
...	...	...

↓

id	index	time	value
0	0	[0.0, 0.1, 0.2, 0.3, 0.4]	[0.3, 0.5, 0.4, 0.7, 0.1]
0	1	[0.3, 0.4, 0.5, 0.6, 0.7]	[0.7, 0.1, 1.2, 1.0, 0.8]
0	2	[0.6, 0.7, 0.8, 0.9 ...]	[1.0, 0.8, 0.1, 0.3, ...]
...	...	...	...

Abbildung 4.9. Veranschaulichung Sliding Window

Resampling Ziel des Resamplings ist, die Zeitstempel einer Menge von univariaten Zeitreihen zu vereinheitlichen und eine multivariate Zeitreihe zu berechnen. Dazu werden mehrere Zeitreihen mittels `groupBy` zusammengefasst und anschließend auf die Zeitstempel einer beteiligten Zeitreihe, oder neue, generierte Zeitstempel interpoliert. Ergebnis der Transformation ist eine Zeile mit einem Array von Zeitstempeln und eine Zeile mit mehreren Arrays von zugehörigen Messwerten (Array-Schema). Wenn in jeder Gruppe ein Resampling der gleichen Anzahl von Zeitreihen mit gleichen Bezeichnern durchgeführt wird, kann das Wertespalten-Schema als Ergebnis erzeugt werden.

id	Aggregations-Key	name	time	value
0	0	A	Array[Zeitstempel]	Array[Datenwert]
1	0	B	Array[Zeitstempel]	Array[Datenwert]
2	1	C	Array[Zeitstempel]	Array[Datenwert]
3	1	D	Array[Zeitstempel]	Array[Datenwert]
...	...	...	...	...

↓

Aggregations-Key	name	time	value
0	[A, B, ...]	Array[Zeitstempel]	Array[Array[Datenwert]]
1	[D, C, ...]	Array[Zeitstempel]	Array[Array[Datenwert]]
...	...	...	...

Abbildung 4.10. Veranschaulichung Resampling mit Erzeugung des Array-Schemas

Aufteilen der Array-Spalte in Wertespalten Um ein DataFrame mit Array-Schema in ein DataFrame mit mehreren Wertespalten (Abbildung 4.5) umzuwandeln, wird eine Operation implementiert, die Anhand der Bezeichner aus der `name`-Spalte die Werte des `values`-Array auf mehrere Spalten aufteilt.

Aufteilen der Array-Spalte in eine flache Spalte Um DataFrames mit multivariaten Zeitreihen in Array-Schema in DataFrames mit Standardschema bestehend aus univariaten Zeitreihen umzuwandeln, wird eine Operation implementiert, die für jedes Name-Wertereihe-Tupel eine Zeile erzeugt und alle anderen Werte der Zeile kopiert.

Zusammenfassen von Wertespalten zu einer Array-Spalte Diese Transformation ist die Umkehroperation des Aufteilens der Array-Spalte in Wertespalten.

Von Wertespalten zum Standardschema Diese Transformation wandelt ein DataFrame mit fester Anzahl von Wertereihen pro Zeitreihe in ein DataFrame aus univariaten Zeitreihen nach Standardschema um.

4.2.3. Funktionen zum Laden von Daten

Eine Analyseumgebung muss einfachen Zugriff auf die eigentlichen Messdaten geben. Ausgangsdaten aus der Datenbank in die Analyseumgebung zu laden ist eine Routineaufgabe, die für den Analysten in wenigen Zeilen Code erledigt sein sollte. Deshalb sind einfache, aussagekräftige und im Hintergrund optimierte Aufrufe bereitzustellen, die möglichst jede gewünschte Granularität von Daten aus dem Speichersystem filtern können. Die Aufrufe sollten zudem im Rahmen der Arbeit für alle Systeme gleichermaßen funktionieren, sodass die Vergleichbarkeit möglichst groß wird.

Folgende Auswahlkriterien für Daten sollten die Funktionen zum Laden unterstützen: Auswahl eines Messkanals oder einer Kanalgruppe, Auswahl des Fahrzeugs, Auswahl der Messung, Auswahl der Zeit in der Messung und alle Kombinationen der Genannten.

4.2.4. Basisoperationen für die Analyse mit Zeitreihen

Für die Analyse von Zeitreihen in DataFrames sollen eine Reihe von Basisoperationen performant bereitgestellt werden. Als solche Basisoperationen wurden das Ausschneiden von Bereichen aus Zeitreihen, das Berechnen von einfachen Statistiken, sowie das Finden und Entfernen von Ausreißer- und Fehlerwerten identifiziert. Diese Kategorien werden im Folgenden kurz erläutert und begründet.

Ausschneiden von Zeitbereichen Oft sollen nur Teilbereiche von Zeitreihen analysiert werden. Deshalb sind Funktionen zum Ausschneiden von Bereichen aus der Zeitreihe essenziell. Das Ausschneiden soll sowohl anhand von Zeitwerten (Anfangs- und Endzeitpunkt), als auch anhand von Indizes (Anfangs- und Endindex) ermöglicht werden.

Einfache Statistiken Zur Datenexploration und zum Überblick über vorhandene Messreihen sollen die Statistiken Minimum, Maximum, arithmetischer Mittelwert, Anzahl der Werte, Median, Varianz und Standardabweichung für Zeitreihen implementiert werden.

Finden und Entfernen von Ausreißer- und Fehlerwerten Bei Messungen entstehen häufig Ausreißer, die bei der Analyse nicht betrachtet werden sollen. Deshalb sind Funktionen bereitzustellen, die Ausreißer erkennen und entfernen oder markieren. Ähnlich können im Wertebereich codierte Fehler- und Startwerte behandelt werden, die oft fest als höchste oder niedrigste Zahl des Wertebereichs umgesetzt sind.

4.3. Auswahl des Speichersystems

Um unter den Speichertechnologien eine Reihung vorzunehmen, werden nun verschiedene relevante Faktoren vorgestellt, die für jedes Datenbanksystem und jedes Dateiformat evaluiert werden. Zunächst wird untersucht, welche Bibliotheken zur Anbindung der Datenquelle an Apache Spark existieren und wie diese zu nutzen sind. Wichtige Fragestellungen hierbei sind: „Ist die Bibliothek zur Anbindung

offiziell von Spark oder vom Hersteller des Speichersystems unterstützt?“ „Ist die Bibliothek für die neuste Version von Apache Spark verfügbar?“ „Wann wurde die Bibliothek zuletzt aktualisiert, wird sie aktiv weiterentwickelt?“ „Ist die Schnittstelle stabil oder sind umfangreiche Änderungen zu erwarten?“

Die Frage nach der Benutzbarkeit für den Anwendungsfall ist wichtiger als die oben genannten. Es ist zu überprüfen, ob die Bibliothek zur Anbindung der Datenquelle an Spark die genutzten Schemata unterstützt und einfach zu bedienen ist. Zur Einschätzung der Komplexität der Benutzung werden die Systeme untereinander verglichen.

Abschließend ist zu überprüfen, wie schnell der Datenzugriff auf das jeweilige Speichersystem möglich ist. Dazu soll insbesondere die theoretisch erreichbare Geschwindigkeit durch Speichersystem, Netzwerk und Speicherformat mit der experimentell festgestellten Geschwindigkeit beim Laden von Daten verglichen werden.

Das optimale, gesuchte Speichersystem hat eine einfach benutzbare, für alle aktuellen Versionen von Apache Spark verfügbare Anbindungsbibliothek, die aktiv weiterentwickelt wird und die verwendeten Schemata direkt unterstützt. Beim Laden von Daten soll die Geschwindigkeit hauptsächlich durch die Hardware begrenzt sein.

4.4. Benchmarks

Zur Evaluierung des Gesamtsystems werden zwei zentrale Aspekte betrachtet: die Geschwindigkeit des Speichersystems und die Benutzbarkeit. Weil die Analysegeschwindigkeit von den verwendeten Algorithmen abhängt, kann sie hier nicht betrachtet werden. Folgende Fälle werden genauer ausgewertet:

Laden von Daten Zum Vergleich der Geschwindigkeit der verwendeten Speichersysteme werden unterschiedliche Mengen von Daten in Apache Spark geladen und die benötigte Zeit gemessen. Werden Transformationen benötigt, um das Standardschema zu erhalten, wird die Ausführungszeit der Transformationen mit gemessen. Zu testende Granularitäten ergeben sich aus den Möglichkeiten der Ladefunktion. Wichtige Größen sind: Eine vollständige Messdatei, der gleiche Kanal aus vielen Messungen, mehrere Kanäle aus vielen Messungen und viele vollständige Messungen mit allen Kanälen.

Beispielanalysen Aus der Anwendung sind verschiedene Beispielanalysen bekannt, die zum Test der Nutzbarkeit implementiert wurden und in Abschnitt 6.4 vorgestellt und ausgewertet werden werden.

Im folgenden Kapitel wird die Implementierung der bis hier konzipierten Transformationen und Funktionen zum Abruf der Daten vorgestellt. Die entwickelten Schemata sind Ein- und Ausgabe der implementierten Transformationen. Darauf aufbauend werden in Kapitel 6 die Benchmarks vorgestellt und die Speichersysteme verglichen.

5. Implementierung

Die im letzten Kapitel entwickelten Konzepte für einen Softwarestack und die Analyse von Messdaten in Apache Spark werden im Folgenden in ihrer Implementierung vorgestellt. Dabei wird in Abschnitt 5.1 auf die Installation und Inbetriebnahme der verwendeten Software eingegangen und in Abschnitt 5.2 die Implementierung der konzeptionierten Funktionsbibliotheken vorgestellt. In Kapitel 6 wird die vorgestellte Implementation evaluiert.

5.1. Installation und Inbetriebnahme

Zur Implementierung und zum Test wurden die verwendeten Softwaresysteme auf einem virtuellen Cluster installiert. Im Folgenden werden wichtige Hard- und Softwareparameter vorgestellt.

5.1.1. Systemüberblick

Das verwendete Cluster besteht aus zwölf virtuellen Maschinen (VM) die wie folgt eingerichtet sind: sechs der Maschinen führen Apache Spark und alle Hadoop-basierten Systeme aus. Dabei dient eine der Maschinen als Gateway und die anderen als Worker. Alle VMs sind mit zwei virtuellen CPU-Kernen, 16 GB RAM und 1 TB Festplattenspeicher (nur Worker) konfiguriert und virtuell mit zwei Mal 10-Gbit/s-Ethernet verbunden. Der Speicher ist über iSCSI via Ethernet angebunden. Die anderen sechs virtuellen Maschinen sind identisch konfiguriert und werden zum Betrieb der verbleibenden Datenbanksysteme genutzt. Alle zwölf VMs befinden sich im gleichen Subnetz.

5.1.2. Installation und Konfiguration

Apache Spark ist im Standalone-Modus auf dem Hadoop-Cluster installiert, wobei die Gateway-VM sowohl Master als auch Driver ausführt. Auf den fünf Worker-Maschinen wird jeweils ein Executor mit zwei CPU-Kernen und maximal 8 GB RAM bereitgestellt. Somit stehen dem Cluster insgesamt zehn virtuelle CPUs und 40 GB RAM zur Verfügung. Zum Speichern von temporären Dateien sind bei jeder Worker-VM 20 GB SSD-Speicher konfiguriert. Der Spark-Driver wird über das zusätzlich auf dem Gateway installierte Jupyter-Notebook in Form einer PySpark-Shell bereitgestellt. Zum Betrieb von PySpark ist auf allen sechs Maschinen im Hadoop-Cluster die Anaconda-Python-Distribution mit den gleichen Paketen installiert.

Neben Apache Spark ist auf dem Gateway ein HDFS-Namenode und auf den Workern jeweils ein Datanode eingerichtet. So kann die Datenlokalität im Zusammenspiel mit HDFS optimal ausgenutzt werden.

5.1.3. Ausführung

Zum Start der Notebook-Umgebung mit PySpark-Shell wird ein Shellscript auf der Gateway-VM ausgeführt. Anschließend ist die Notebook-Oberfläche per HTTP unter Port 9999 erreichbar. Geöffnete Notebooks starten automatisch eine PySpark-Shell als Kernel. Das Nutzen von mehreren Notebooks gleichzeitig ist aktuell nicht unterstützt. Um die parallele Nutzung des Clusters zu ermöglichen, müssen zusätzliche Einstellungen vorgenommen werden und das Cluster sollte vergrößert werden.

5.2. Funktionsbibliotheken

Nachdem der Aufbau und die Konfiguration des Clusters bekannt ist, wird in diesem Abschnitt die Implementierung der Funktionsbibliotheken beschrieben. Zunächst werden allgemeine Erwägungen und Vorgehensweisen bei der Implementierung von Funktionen in Apache Spark vorgestellt. Anschließend wird die Implementierung der DataFrame-Transformationen (5.2.2), der Funktionen zum Laden von Daten (5.2.3) und der Basisoperationen (5.2.4) erläutert.

5.2.1. Allgemeines

Die meisten im Rahmen dieser Arbeit implementierten Funktionen sind Spark-UDFs. Apache Spark unterscheidet User Defined Function (UDF) und User Defined Aggregate Function (UDAF). UDFs in Spark sind mit Zeilenfunktionen (siehe Abschnitt 2.5) vergleichbar, weil sie mehrere Parameter und mehrere Ausgaben ermöglichen. UDAFs sind Aggregatfunktionen, nehmen Spalten als Eingabe entgegen und berechnen ein oder mehrere Aggregate in Form einer (strukturierten) Spalte. Aufgrund der Architektur von PySpark (siehe Abschnitt 3.3) ist es performant, häufig verwendete Funktionen in Scala oder Java zu implementieren, sodass sie in der Spark-JVM ausgeführt werden. Das verhindert das Serialisieren von DataFrames zur Übertragung an Python-Interpreter. Der Performancevorteil dieser Herangehensweise liegt laut verschiedenen Quellen zwischen Faktor 2 und Faktor 1000 [WB 18; Mej18]. Ein weiteres Argument für die Nutzung von Scala ist, dass aktuell nicht alle UDAFs in Python implementiert werden können, insbesondere nicht solche, die mit Vektoren arbeiten, weil Spark das Serialisieren von diesen für Aggregatfunktionen nicht unterstützt.

Aufgrund dieser Erwägungen werden die Basisfunktionen in Scala implementiert und mithilfe eines Wrappers in Python verfügbar gemacht. Hier vorgestellter Python-Code besteht deshalb aus Aufrufen von Spark-Funktionen, die in Scala implementiert sind. Die Analysen der Zeitreihen in den Zeilen der DataFrames können und sollen in Python implementiert werden. Beispiele folgen in Abschnitt 6.4.

Apache Spark enthält aktuell zwei Array-Typen für die Speicherung von Double-Werten: den `ArrayType` aus Spark SQL und den `VectorType` aus Spark MLlib. Die Verwendung unterscheidet sich in PySpark. Die Arrays aus Spark SQL werden in Python auf Listen abgebildet, während die Vektoren aus MLlib intern NumPy-Arrays verwenden. Weil hier davon ausgegangen wird, dass die meisten Analysen auf Basis von NumPy implementiert werden, wird zur Verringerung der Anzahl von Konvertierungen der `VectorType` verwendet, um Zeitstempel und Wertereihen in DataFrames zu speichern und zu verarbeiten.

Implementierung von UDFs in Spark User Defined Functions (UDFs) für Spark können in Python und Scala implementiert werden. Dazu werden in beiden Sprachen reguläre oder anonyme Funktionen mithilfe der Funktion `udf()` in einem `UserDefinedFunction`-Objekt gekapselt. Dieses Vorgehen ist in Abbildung 5.1 beispielhaft dargestellt. Das Funktionsobjekt kann nach der Deklaration in der DataFrame-DSL verwendet werden und nimmt Spalten eines DataFrames als Positionsparameter entgegen. Apache Spark wendet die Funktion zeilenweise an und bildet eine neue Spalte aus den Ergebnissen.

```scala
import org.apache.spark.sql.expressions.
↪ UserDefinedFunction
import org.apache.spark.sql.functions.udf

object Functions {
  val square_udf: UserDefinedFunction =
    udf((x: Double) => x * x)

  def increment(n: Int): Int = {
    n * 1
  }

  val increment_udf = udf(increment)
}
```

```python
from pyspark.sql.functions import udf
from pyspark.sql.types import LongType
from pyspark.sql.types import IntegerType

squared_udf = udf(lambda x: x * x,
↪ LongType())

def increment(n):
  return n + 1

increment_udf = udf(increment,
↪ IntegerType())
```

Programmcode 5.1 Definition von zwei Spark-UDFs in Scala (links) und in Python (rechts). In Python ist die Angabe des Ergebnistyps erforderlich. In Scala kann der Ergebnistyp automatisch durch den Compiler abgeleitet werden. UDFs können sowohl aus anonymen Funktionen (`square_udf`) als auch aus benannten Funktionen (`increment_udf`) erzeugt werden.

Implementierung von Scala-UDAFs in Spark Die Schnittstelle für das Implementieren von UDAFs in Apache Spark ist für assoziative und kommutative Aggregatfunktionen konzipiert, da für diese einfacher Zwischenergebnisse erstellt und parallel verarbeitet werden können. Aggregatfunktionen, die nicht kommutativ oder nicht assoziativ sind, können über UDFs implementiert werden. Dazu werden die zu aggregierenden Werte mit der Funktion `collect_list` zusammengefasst und mit der UDF zeilenweise verarbeitet. Für die Implementierung von UDAFs in Spark sind die in Tabelle 5.1 gelisteten Parameter zu spezifizieren.

Name	Beschreibung
inputSchema	Schema mit Typen der Eingabe
bufferSchema	Schema mit Typen der Zwischenwerte (Buffer) bei der Aggregation
dataType	Schema der Ausgabe
deterministic	Ist die Funktion deterministisch?
initialize	Eine Funktion, die einen Zwischenwert erzeugt, der als Null-Element fungiert
update	Diese Funktion „addiert“ eine Zeile des zu aggregierenden DataFrames zu einem Buffer
merge	Diese Funktion führt zwei Buffer zusammen
evaluate	Diese Funktion berechnet aus dem Buffer, der aus allen Buffern gemerged wurde, das Endergebnis der Aggregation.

Tabelle 5.1. Parameter einer UDAF in Apache Spark

5.2.2. Transformationen

Im Folgenden werden die Implementierungen der in Abschnitt 4.2.2 konzipierten Transformationsfunktionen für DataFrames vorgestellt. Dabei wird auf das Transponieren von DataFrames, das Zerlegen von

Zeitreihen mittels Sliding-Window-Techniken, das Resampling und die Transformationen zwischen den drei Hauptschemata eingegangen.

Transponieren Für das Transponieren von Zeitreihen, wie in Abschnitt 4.2.2 beschrieben, wird eine neue Teilfunktion benötigt. Der gesamte Algorithmus wurde in drei Teile aufgeteilt, die in Python aufgerufen werden (siehe Programmcode 5.2).

```
1  def transpose_by_key(df: DataFrame, time: str, value: str, key: str) -> DataFrame:
2    agged = df.groupBy(key).agg(functions.collect_list(time).alias("time"),
     ↪  functions.collect_list(value).alias("value"))
3    sorted_df = agged.withColumn("sorted", sort_time_value_udf(functions.col("time"),
     ↪  functions.col("value")))
4    result = flatten_column(sorted_df, "sorted", [time, value])
5    return result
```

Programmcode 5.2 Transponieren: Tupel im DataFrame werden nach dem Attribut `key` gruppiert. Dabei werden Listen von Zeit und Wert aggregiert, die anschließend nach der Zeit sortiert werden. Es entsteht das Schema `(key, time, value)`.

Die Funktion aggregiert die Zeitwerte und Messwerte jeder Zeitreihe zu zwei Listen. Anschließend wird die Scala-Funktion `sort_time_value` aufgerufen, die die Zeit-Wert-Paare nach der Zeit sortiert. Die Aggregation und das Sortieren der Werte sind sehr aufwändig. Deshalb sollte diese Transformation vermieden werden. Die Funktion `sort_time_value_udf` ist ein Beispiel für einen Python-Wrapper um eine Scala-Funktion (siehe Programmcode 5.3). Über den genutzten `SparkContext` wird eine Referenz auf das Funktionsobjekt in der JVM erzeugt (`_sort_time_value`). Anschließend wird die `apply`-Methode des Objekts aufgerufen.

```
1  def sort_time_value_udf(time: Union[str, Column], value: Union[str, Column]) -> Column:
2    sc = SparkContext._active_spark_context
3    _sort_time_value = sc._jvm.adrianlu.BA.UDF.SortTimeValue.getUDF()
4    return Column(_sort_time_value.apply(_to_seq(sc, [time, value], _to_java_column)))
```

Programmcode 5.3 Beispiel für einen Python-Wrapper um eine Scala-Funktion. Über den genutzten `SparkContext` wird eine Referenz auf das Funktionsobjekt in der JVM erzeugt (`_sort_time_value`). Anschließend wird die `apply`-Methode des Objekts mit den Spaltennamen als Scala-Sequenz aufgerufen.

Sliding Window Die Sliding-Window-Funktion ist eine Scala-UDF, abgebildet in Programmcode 5.4, die aus einer Python-Funktion aufgerufen wird. Besonderheit ist hier, dass die Funktion neben den Vektoren aus dem DataFrame die Parameter `window_size` und `step_size` besitzt, die einmalig pro Anwendung der UDF auf ein DataFrame festgelegt werden. Die Funktion erzeugt als Ergebnis eine Spalte vom Typ `Map[Int, (Vector, Vector)]`. Die Verschachtelung wird über die Spark-SQL-Funktion `explode` im aufrufenden Python-Code aufgelöst. Die `explode`-Funktion erzeugt eine neue Zeile für jedes Element in einem übergebenen Mapping.

Resampling Das Resampling von vielen Zeitreihen auf eine gemeinsame Zeitachse (siehe Abschnitt 4.2.2) hat als Argument Zeit- und Wertespalte vom Typ Vektor. Ausgabe ist ein gemeinsamer Zeitvektor und die Wertvektoren. Die Aggregatfunktion ist in drei Varianten mit jeweils zwei Ergebnisschemata implementiert, die im Folgenden vorgestellt werden:

- mit neuer Zeitachse und Array-Schema als Ergebnis
- mit neuer Zeitachse und Wertespalten-Schema als Ergebnis

```scala
def call(window_size: Int, step_size: Int)(time: Vector, value: Vector): Map[Int, (Vector,
↪  Vector)] = {

  val vector_size = time.size
  var nr_of_windows = (vector_size - window_size) / step_size

  (0 to nr_of_windows).map { i =>
    val start = i * step_size
    val end = i * step_size + window_size
    i -> (Vectors.dense(time.toArray.slice(start, end)),
    ↪  Vectors.dense(value.toArray.slice(start, end)))
  }.toMap[Int, (Vector, Vector)]
}
```

Programmcode 5.4 Sliding-Window-UDF: Es werden Fenster über die Anzahl der enthaltenen Werte und die Schrittgröße gebildet. Dabei sind nur komplette Fenster enthalten. Werte am Ende gehen verloren. Ergebnis der Anwendung ist eine Spalte vom Typ `Map[Int, (Vector, Vector)]`, die durch die Spark-SQL-Funktion `explode` aufgelöst wird.

- mit vorgegebener Zeitachse von einer Zeitreihe und Array-Schema als Ergebnis
- mit vorgegebener Zeitachse von einer Zeitreihe und Wertespalten-Schema als Ergebnis
- unter der Voraussetzung, dass alle Zeitachsen gleich sind und mit Array-Schema als Ergebnis
- unter der Voraussetzung, dass alle Zeitachsen gleich sind und mit Wertespalten-Schema als Ergebnis

Die Interpolation wird über das Zero-Order-Hold-Verfahren umgesetzt. Zusätzlich wurde die Interpolation mit neuer Zeitachse für einzelne univariate Zeitreihen implementiert.

Wenn eine neue Zeitachse erzeugt wird, kann das Resampling als UDAF implementiert werden, weil die Zeitachse iterativ bei den `update`- oder `merge`-Schritten erzeugt werden kann. Die Zeitachse des Zwischenergebnisses wird im `update`-Schritt verlängert, wenn die hinzuzufügende Zeitreihe länger ist. Anschließend wird die Zeitreihe in der aktuellen Zeile auf die Zeitachse des bestehenden Zwischenergebnisses interpoliert und die Werte werden an das Array angehängt (Array-Schema). Im Merge-Schritt (Programmcode 5.5) wird überprüft, welche der Zeitreihen länger ist und die längere Zeitachse übernommen. Die Wertearrays werden verkettet[1]. Wenn die am Resampling beteiligten Messkanäle mit ihrem Namen bekannt sind, kann das Wertespalten-Schema erzeugt werden. Dazu wird eine veränderte Form der UDAF verwendet, die die Spaltennamen als zusätzliches Argument nutzt und interpolierte Wertevektoren nach Name in die zugehörige Spalte einordnet, statt sie an ein Array anzuhängen.

Das Resampling mit vorgegebener Zeitachse ist als UDF implementiert und hat als zusätzliche Argumente eine Spalte mit Identifikatoren für die Tupel aus Zeiten und Werten, sowie den Identifikator für das Tupel, dessen Zeitachse für die Ergebniszeitreihe verwendet werden soll. Die in Programmcode 5.6 dargestellte Funktion setzt den Algorithmus um. Die UDF (Programmcode 5.7) sucht dabei über den Wert der Identifikationsspalte die zu nutzende Zeitachse und interpoliert alle Zeitreihen auf diese.

Sollen mehrere Zeitreihen, von denen bekannt ist, dass sie sich eine Zeitachse teilen, zu einer multivariaten Zeitreihe zusammengefasst werden, handelt es sich nicht um ein Resampling im eigentlichen Sinne. Die Transformation wird an dieser Stelle vorgestellt, weil die Schemaveränderung der des Resamplings gleicht. Die Funktion ist als UDAF implementiert, die die Wertevektoren sammelt und die Zeitachsen aller Zeitreihen bis auf eine verwirft. Dieses Vorgehen hat mutmaßlich Performancevorteile gegenüber den anderen Implementierungen des Resamplings, da keine Berechnungen durchgeführt und weniger Daten übertragen werden.

[1] Soll das Wertespalten-Schema erzeugt werden, werden die Wertevektoren aus beiden Buffern in einen übernommen, anstatt zwei Arrays zu verketten.

```scala
override def merge(buffer1: MutableAggregationBuffer, buffer2: Row): Unit = {
  val buffer1_time = buffer1.getAs[Seq[Double]](0)
  val buffer1_values = buffer1.getAs[Seq[DenseVector]](1)
  val buffer2_time = buffer2.getAs[Seq[Double]](0)
  val buffer2_values = buffer2.getAs[Seq[DenseVector]](1)

  if (buffer2_time.isEmpty) return
  if (buffer1_time.isEmpty && buffer2_time.isEmpty) return
  if (buffer1_time.isEmpty) {
    buffer1(0) = buffer2.get(0)
    buffer1(1) = buffer2.get(1)
    return
  }

  val buffer1_time_bigger = buffer1_time.last >= buffer2_time.last

  if (!buffer1_time_bigger) {
    buffer1(0) = buffer2.get(0)
  }

  val new_values = buffer1_values ++ buffer2_values

  buffer1(1) = new_values
}
```

Programmcode 5.5 Merge beim Resampling mit neuer Zeitachse und Erzeugung des Array-Schemas.

```python
def resample_given_array(df: DataFrame, time: str, value: str, idents: str, aggregation_key: Union[str, List[str], Column], base: str, tolerance: float) -> DataFrame:
  grouped = df.groupBy(aggregation_key).agg(
    functions.collect_list(time).alias(time),
    functions.collect_list(value).alias(value),
    functions.collect_list(idents).alias(idents)
  )
  resampled = grouped.withColumn("resampled", resample_given_array_udf(time, value, idents, base, tolerance))
  result = flatten_column(resampled, "resampled", [time, value])
  return result
```

Programmcode 5.6 Resampling mit gegebener Zeitachse: Zunächst die Spalten `time` und `value`, sowie eine Identifikationsspalte mit `collect_list` aggregieren, dann die UDF anwenden und zuletzt Umbenennungen vornehmen.

```scala
def call(name: Any, tolerance: Double)(times: Seq[Vector], values: Seq[Vector], identifiers: Seq[Any]): TimeValuesTuple = {
  val name_index = identifiers.indexOf(name)
  assert(name_index >= 0)

  val base_times = times(name_index).toArray

  val interpol_values: Seq[Vector] = times.zip(values).map({
    case (time_vector, value_vector) =>
      Vectors.dense(Interpolation.leftNearest(time_vector.toArray, value_vector.toArray, base_times, tolerance))
  })

  TimeValuesTuple(Vectors.dense(base_times), interpol_values)
}
```

Programmcode 5.7 Resampling mit gegebener Zeitachse: Zunächst wird die gültige Zeitachse gesucht, anschließend werden alle Zeitreihen interpoliert.

```
def explode_together(df: DataFrame, *cols_to_explode: str) -> DataFrame:
  zipped = df.withColumn("zip", functions.arrays_zip(*cols_to_explode))
  cols_to_keep = zipped.columns
  cols_to_keep.remove("zip")
  for col in cols_to_explode:
    cols_to_keep.remove(col)

  exploded = zipped.select(*cols_to_keep, functions.explode("zip").alias("exploded"))
  return flatten_column(exploded, "exploded", cols_to_explode)
```

Programmcode 5.8 Funktion zur Erweiterung der Funktionalität von `explode()`

Weitere Transformationen zwischen Schemata Die im vorherigen Kapitel vorgestellten Transformationen zwischen Array- und Wertespaltenschma und die Transformationen zum Standardschema wurden ebenfalls implementiert. Besonderheit der UDF zum bilden des Wertespaltenschemas ist der von den Argumenten abhängige Funktionstyp. Die Anzahl der erzeugten Spalten ist variabel.

Zum Aufteilen der multivariaten in univariate Zeitreihen wurde die in Programmcode 5.8 dargestellte Funktion `explode_together(df, *cols_to_explode)` entwickelt. Sie nutzt die Skalarfunktion `arrays_zip(*columns)`, die eine neue Spalte vom Typ `Array[Tuple]` erzeugt. Die entstandenen Arrays werden anschließend mit `explode()` aufgeteilt.

5.2.3. Laden und Speichern

Für das Laden von Daten aus Postgres-XL und MySQL wurde die JDBC-Unterstützung von Apache Spark genutzt. Mit dieser können Tabellen, Sichten oder Ergebnisse von Anfragen aus SQL-Datenbanksystemen als Quellen für DataFrames verwendet werden. Das hierdurch erzeugte DataFrame wird nicht automatisch in Partitionen unterteilt. Bei großen Tabellen kommt es deshalb zu Überlauf der Partition und die Parallelisierung von Spark wird nicht ausgenutzt. Deshalb sollte eine der zwei Möglichkeiten zur manuellen Partitionierung genutzt werden:

Angabe von Prädikaten Spark bietet die Möglichkeit zur Erzeugung eines DataFrames aus einer JDBC-Datenbank eine Liste von String-Prädikaten zu übergeben. Jedes dieser Prädikate wird als where-Klausel in einer JDBC-Anfrage zur Erstellung einer Partition verwendet.

Angabe von `partitionColumn`, `lowerBound`, `upperBound` und `numPartitions` Durch die Angabe der genannten Werte bildet Spark Prädikate in der Anzahl der zu erstellenden Partitionen über die angegebene Spalte. Dazu werden zwischen `lowerBound` und `upperBound` Ranges gebildet, die als where-Klausel in den Anfragen verwendet werden.

Postgres-XL Für die Arbeit mit den in Postgres-XL (Postgres) gespeicherten Daten, müssen diese aus dem normalisierten Datenbankschema abgerufen und in das Standardschema überführt werden. Dazu ist es nötig, Joins über das gesamte Schema, wie in Programmcode 5.9 und unter diesem Absatz dargestellt, durchzuführen. Diese Verbundoperationen können sowohl in Apache Spark, als auch im Datenbanksystem durchgeführt werden.

$$timeseries \bowtie channel\_dict \bowtie \beta_{filename \leftarrow name}(file) \bowtie \beta_{dname \leftarrow name}(device) \bowtie channelgroup$$

Einen Verbund in Postgres durchzuführen ermöglicht die Nutzung der Optimierungen im Datenbanksystem. Zudem ist es beim verwendeten Schema erforderlich, den Join im Datenbanksystem durchzuführen, um die Zeitreihen nach Kanalname, Dateiname oder Start der Messung zu filtern. Wenn der Join nicht

```
file(fid integer, name varchar, comment varchar, starttime timestamp)
channelgroup(fid integer, gid smallint, raster varchar, t double[])
timeseries(fid integer, gid smallint, cid integer, v double[])
channel_dict(cid integer, did smallint, name varchar)
device(did smallint, name varchar)
```

Programmcode 5.9 Datenbankschema Postgres-XL

im Datenbanksystem durchgeführt wird, muss Apache Spark die gesamte `timeseries`-Relation laden, bevor diese durch einen Verbund mit der `channel_dict`-Relation über das `name`-Attribut gefiltert werden kann. Das sorgt für die Übertragung von vielen nicht benötigten Tupeln und verlängert die Abrufzeit deutlich.

Der Join im Datenbanksystem erzeugt großen Übertragungsaufwand vom Datenbanksystem zu Apache Spark, wenn er nicht zum Filtern genutzt wird. Beim Join von `timeseries` und `channelgroup` werden alle Folgen von Zeitstempeln in der Spalte `t` für jede zugehörige Wertereihe in `v` kopiert und müssen vielfach vom Datenbanksystem zu Apache Spark übertragen werden. Wenn nicht ein Großteil der durch den Join entstehenden Tupel durch Selektion entfernt wird, sollte dies vermieden werden.

Weil Apache Spark den Verbund nicht wie Projektion und Selektion im Rahmen der Optimierung an das Datenbanksystem auslagert, muss manuell entschieden werden, ob es performanter ist, einen Join von Spark durchführen zu lassen, oder ihn in der SQL Query an Postgres zu formulieren. Wie oben erläutert, verlangsamt das Kopieren der Zeitstempel durch einen Join das Laden aus Postgres. Deshalb wurden zwei Varianten für den Verbund aller Relationen implementiert. Wenn hauptsächlich Zeitreihen aus unterschiedlichen Kanalgruppen geladen werden, wird der gesamte Join in Postgres durchgeführt. Wenn viele Zeitreihen aus der gleichen Kanalgruppe geladen werden, wird der Join von `file`, `timeseries`, `device` und `channel_dict` in Postgres durchgeführt und das erzeugte DataFrame in Spark mit `channelgroup` verbunden. Dieses Vorgehen ist in Abbildung 5.1 dargestellt.

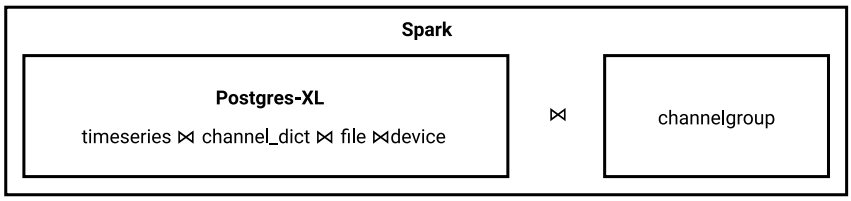


Abbildung 5.1. Verteilung der Joins auf die Systeme, um die vielfache Übertragung der Zeitstempel `channelgroup` zu vermeiden.

Die Lösung ist unflexibel, weil der Entwickler entscheiden muss, ob ein Join vom Datenbanksystem oder von Spark durchgeführt werden soll. In Zukunft wird der Spark-Optimierer das Weitergeben von Verbundoperationen an Datenquellen ermöglichen und logische und kostenbasierte Optimierung nutzen, um zu entscheiden, welches System welchen Join ausführt [Li19; Fan18].

Um Tupel aus Postgres-XL gleichmäßig auf Partitionen aufzuteilen, wird ein Prädikat verwendet. Die Tupel werden nach der Summe von `fid` und `cid` modulo 100 aufgeteilt. Dadurch sind die Partitionen sowohl beim Abruf eines Kanals über viele Messungen, als auch beim Abruf von vielen Kanälen aus einer Messung über die Partitionen verteilt, um Parallelisierung zu ermöglichen und Überlauf von Partitionen zu verhindern.

Neben der Verteilung der Joins auf Spark und Postgres ist das Verwenden von Filtern für die Daten wichtig für die Performance und Nutzbarkeit des Systems. Weil Spark SQL Selektionen und Projektionen automatisch an JDBC-Datenquellen abgibt, wird diese Funktionalität mit der DataFrame-API umgesetzt. Die

```
file(fid integer, name varchar, comment varchar, starttime timestamp)
channelgroup(fid integer, gid smallint, raster varchar)
timestamps(fid integer, gid integer, i integer, t integer)
timeseries(fid integer, cid smallint, i integer, v double)
channel_dict(cid integer, did smallint, name varchar)
device(did smallint, name varchar)
channel(fid integer, gid integer, cid integer)
```

Programmcode 5.10 Datenbankschema MySQL

Funktion `filter_dataframe(df, fid, cid, gid, did, name, filename, start_time, time)` wendet nacheinander Selektionen auf das übergebene DataFrame `df` an, die vom Spark-Optimierer an Postgres delegiert werden. Die Funktion wird aus den `load`-Funktionen aufgerufen und bekommt die gleichnamigen Parameter von diesen übergeben. Die Typen der Parameter sind identisch.

Bei den Parametern `cid`, `did`, `fid` und `gid` wird unterschieden, ob diese vom Typ `int`, `Tuple[int, int]`, oder `List[int]` sind. Wird ein `int` übergeben, wird eine Exact-Match-Klausel erzeugt. Bei Übergabe eines Tupels werden alle Werte im geschlossenen Intervall $[tupel[0], tupel[1]]$ selektiert. Hat der Parameter den Typ `List[int]`, wird eine `IN`-Klausel gebildet. Für den Parameter `name` wird unterschieden, ob es sich um eine Variable vom Typ `str` oder `List[str]` handelt. Bei Ersterem wird eine Exact-Match-, bei Zweiterem eine `IN`-Klausel erzeugt. `start_time` kann als `datetime.datetime` oder als Tupel von zwei dieser Werte angegeben werden. Der Parameter `time` ist ein `Tuple[float, float]` und wird genutzt, um alle Zeitreihen auf den angegeben Zeitbereich zuzuschneiden. Alle Parameter akzeptieren `None` als Wert. Dann wird kein Filter für das Attribut erzeugt.

MySQL MySQL hat keine Unterstützung für Array-Typen. Deshalb enthält das MySQL-Schema zusätzlich Index-Spalten, um die Ordnung der Werte einer Zeitreihe festzulegen. Das Schema ist in Programmcode 5.10 dargestellt. Zusätzlich zum Verbund der in MySQL genutzten Relationen muss die Transponierung genutzt werden, um das Standardschema zu erhalten. Das Abrufen der Daten wird aufwändiger, weil die Relationen `timestamps` und `timeseries` eine viel größere Kardinalität haben, als alle Relationen in Posgres-XL.

Aufgrund erster Tests mit vergleichsweise langen Ladezeiten wird von einer weitergehenden Evaluation von MySQL in dieser Arbeit abgesehen.

Parquet Zur Speicherung der Daten in Parquet-Dateien wurden diese aus Postgres-XL im Standardschema geladen und mit der Spark-SQL-Funktion `spark.write.parquet()` in HDFS abgespeichert. Die Daten sind nicht normalisiert. Es werden zwei unterschiedliche Ordnerstruktur-Partitionierungen genutzt: nach `fid` und nach `(channel_dict.name, fid)`. Dadurch stehen Indexe für Filterung nach den Attributen (Kanal-)Name und File-ID zur Verfügung. Das Einlesen der Daten erfolgt über die Funktion `spark.read.parquet()`. Die Parquet-Datenquelle ermöglicht wie die JDBC-Datenquelle Projektion und Selektion. Die Funktion `filter_dataframe` kann ohne Änderungen auf die Ausgabe des Lesevorgangs angewendet werden, weil dasselbe Schema verwendet wird.

HBase und Vector in Hadoop Im Gegensatz zu Datenbanken mit JDBC-Treiber und Parquet-Files gibt es für Apache HBase keine standardmäßige Unterstützung in Apache Spark. Stattdessen muss auf eine Drittbibliothek zurückgegriffen werden und Anfragen müssen mit den bereitgestellten Primitiven implementiert werden. Der Arbeitsaufwand für diese Implementierung hat sich als außerhalb der Reichweite dieser Arbeit herausgestellt. Deshalb wird auf die Implementierung und einen Vergleich mit den anderen beteiligten Systemen verzichtet.

Zum Zeitpunkt der Arbeit lag kein Schema für und keine Instanz von Vector in Hadoop vor. Deshalb wird die Implementierung der `load`-Funktion für VectorH vorgenommen, sobald die Bedingungen erfüllt sind. Aufgrund der Nichtverfügbarkeit von zwei der fünf geplanten Systemen und dem Entfallen von MySQL wird im folgenden Kapitel Postgres-XL mit Apache Parquet verglichen.

5.2.4. Basisoperationen

Die in Unterabschnitt 4.2.4 vorgestellten Basisoperationen wurden als Scala-UDFs mit Python-Wrapper implementiert. Dazu wurde die Bibliothek `Breeze` [Hal19] für schnelle mathematische Standardoperationen genutzt. Zum Ausschneiden von Bereichen aus Zeitreihen werden Funktionen bereitgestellt, die entweder mit Indizes, oder mit Zeitbereichen der Form $[Startzeit, Endzeit)$ arbeiten.

Die Signaturen der Python-Wrapper und Beschreibungen der Funktionsweise sind in Tabelle 5.2 aufgelistet.

Die implementierte Funktionsbibliothek besteht insgesamt aus Funktionen zum Laden der Daten aus Postgres-XL und Apache-Parquet-Dateien gespeichert in HDFS, Transformationen zwischen den drei in Kapitel 4 vorgestellten Schemata und Basisoperationen für die Analyse von Zeitreihen. Alle implementierten Funktionen nutzen soweit wie möglich Basisfunktionalität von Apache Spark und erweitern diese wenn nötig mit in Scala implementierten UDFs und UDAFs. Die Funktionen werden über Wrapper für die Programmierung in Python bereitgestellt. Im Folgenden werden die Funktionsbibliothek und die Speichersysteme für den Anwendungsfall evaluiert.

Signatur	Beschreibung
`find_slice_udf(vector, start_value, end_value)`	Sucht in den Vektoren in der Spalte `vector` die Indizes zum Bereich $[start\_value, end\_value)$ und gibt diese als Spalte mit dem Typ `(Int, Int)` zurück.
`slice_index_udf(vector, start_end=None, start=None, end=None)`	Schneidet den Bereich gegeben durch die Index-Spalten aus `vector`. Kann entweder mit einer Tupel-Spalte `start_end` oder mit den Integer-Spalten `start` und `end` genutzt werden.
`slice_value_udf(vector, start_value, end_value)`	Sucht in der Spalte `vector` den Bereich $[start\_value, end\_value)$ und gibt diesen zurück.
`slice_time_value_udf(time, values, start_time, end_time)`	Sucht in der Spalte `time` den Bereich $[start\_value, end\_value)$ gibt `time` und `value` zugeschnitten auf diesen Bereich zurück.
`slice_time_values_udf(time, values, start_time, end_time)`	Sucht in der Spalte `time` den Bereich $[start\_value, end\_value)$ gibt `time` und alle Vektoren in `values` zugeschnitten auf diesen Bereich zurück.
`vector_min_udf(vector)`	Findet den minimalen Wert jedes Vektors in der Spalte `vector`
`vector_max_udf(vector)`	Findet den maximalen Wert jedes Vektors in der Spalte `vector`
`vector_mean_udf(vector)`	Berechnet den Mittelwert der Werte jedes Vektors in der Spalte `vector`
`vector_std_udf(vector)`	Berechnet die Standardabweichung der Werte jedes Vektors in der Spalte `vector`
`vector_var_udf(vector)`	Berechnet die Varianz der Werte jedes Vektors in der Spalte `vector`
`vector_median_udf(vector)`	Berechnet den Median der Werte jedes Vektors in der Spalte `vector`
`vector_count_udf(vector)`	Gibt die Länge jedes Vektors in der Spalte `vector` zurück
`std_removal(vector)`	Ersetzt in allen Vektoren der Spalte `vector` Werte außerhalb von $[mean - 3 * std, mean + 3 * std]$ durch `Double.NaN`
`value_removal(vector, value: Union[str, float])`	Wenn `value` ein float: Ersetzt alle Vorkommen von `value` in den Vektoren in der Spalte `vector` durch `Double.NaN`, sonst: Ersetzt alle vorkommen des Attributwerts `value` im Vektor des gleichen Tupels durch `Double.NaN`

Tabelle 5.2. Basisoperation-UDFs mit Funktionsbeschreibung

6. Beispielanalysen und Evaluierung

In diesem Kapitel werden die konzipierten und implementierten Funktionen und die Anbindung von Postgres-XL und Parquet-Dateien in HDFS evaluiert. Dazu werden in Abschnitt 6.1 die Funktionen zum Laden der Daten und die erreichte Geschwindigkeit untersucht, in Abschnitt 6.2 die Datenquellen abschließend verglichen und in Abschnitt 6.4 Beispielanalysen vorgestellt. Abschnitt 6.5 diskutiert Verbesserungen und genauere Untersuchungen der vorgestellten Implementierungen.

6.1. Benchmarks

Im Folgenden werden die Ausführungszeiten für das Laden von Daten und das Resampling getestet, um zu bestimmen, welches Speichersystem sich am Besten für die Nutzung mit Spark eignet, und um die Geschwindigkeit einzelner Transformationen zu vergleichen.

Benchmarking einzelner Funktionen in Spark ist durch die Lazy Evaluation und die Optimierung schwierig. Die Lazy Evaluation sorgt für die gemeinsame Ausführung aller angewendeter Transformationen mit der Aktion, die zur Auswertung eines DataFrames führt. Deshalb ist im Driver Program nur die Ausführungszeit der gesamten Auswertung, nicht aber die benötigte Zeit einzelner Transformationen messbar. Die Optimierung sorgt zusätzlich für das Zusammenfassen von mehreren Transformationen zu Stages, sodass die Zeitmessung über die Ausführungszeit von Stages unmöglich wird. Das Messen eines gesamten Algorithmus vom Selektieren der Daten bis zum Speichern der Ergebnisse ist von diesen Einschränkungen nicht betroffen.

6.1.1. Laden von Zeitreihen

Um die genutzten Speichersysteme für Messdaten und ihre Nutzbarkeit im Zusammenspiel mit Apache Spark zu bestimmen, wird mit unterschiedlichen Anfragen die Geschwindigkeit der Datenbereitstellung getestet.

Die Benchmarks werden automatisiert durchgeführt. Für jedes Szenario wird mit der `load`-Funktion ein DataFrame erzeugt, dessen Auswertung anschließend zehn Mal (bei zeitaufwendigen Messungen drei Mal) gestartet wird. Dabei wird für jeden Durchlauf die Zeit gemessen und im Anschluss werden alle Zeiten ausgegeben. Problem bei der Messung der Ladezeiten ist die Optimierung von Spark, die versucht, das Laden von nicht benötigten Daten zu verhindern. Weil Aktionen wie `show` und `count` im Allgemeinen nur auf einer Teilmenge der Daten arbeiten[1], sind sie für die Tests nicht geeignet. Um das Laden der gesamten Daten eines DataFrames `df` zu erzwingen, wird `df.repartition(spalte).write.parquet` verwendet. Die Vergleichbarkeit der Messungen wird durch den Speichervorgang nicht beeinflusst, weil die benötigte Zeit nur von der Datenmenge abhängig ist, die über die einzelnen Messungen eines Szenarios konstant ist. Das Repartitionieren verhindert optimiertes Speichern, bei dem Spark Partitionen parallel zum Einlesen abspeichert.

[1] Für `show` sind nur die ersten n Zeilen erforderlich, für `count` nur eine Spalte

Die zum Testen genutzte Datenbasis umfasst 43 Messungen mit jeweils circa 3600 Kanälen. Von diesen gehören über 99 % zur längsten Kanalgruppe der Datei. Die Kanäle einer Datei haben eine Länge von mindestens 313, maximal 748870 und im Durchschnitt 273466 Werten. Die Datenbasis enthält insgesamt circa 42 Milliarden Messwerte und ist unkomprimiert ungefähr 330 GB groß.

Ein Kanal über alle Messungen Hier werden die Kanäle `nmot_w` und `pvd_w` aus allen im System gespeicherten Messungen geladen. So ergeben sich Zeitreihen mit einer Gesamtlänge von 11,76 Millionen Werten und Zeiten. Die Parquet-Dateien sind mit Index über das `name`-Attribut gespeichert, das zum Filtern der Daten verwendet wird. Parquet ist beim bereitstellen der Daten in diesem Anwendungsfall 10 bis 12 Mal schneller als Postgres-XL. Die unterschiedlichen Ladezeiten zwischen den Kanälen bei gleichem Speichersystem kann der Autor nicht erklären. Es handelt sich bei diesem Test um eine kleine Datenmenge für das System, größere Datenmengen haben sich als konsistenter erwiesen.

	Postgres		Parquet	
Messung Nr.	nmot_w	pvd_w	nmot_w	pvd_w
1	25,671	28,841	4,399	4,969
2	21,812	26,938	3,469	3,546
3	20,310	27,382	2,704	2,593
4	21,132	27,280	2,783	3,549
5	21,270	27,412	2,920	2,993
6	21,016	27,677	2,860	3,074
7	21,765	28,051	2,865	3,110
8	21,338	26,293	2,969	2,621
9	20,859	27,360	2,813	2,596
10	21,503	27,352	2,818	4,463
Durchschnitt	21,668	27,458	3,060	3,351

Tabelle 6.1. Zeitmessungen für das Laden einzelner Kanäle aus allen verfügbaren Messdateien nach System in Sekunden. Parquet mit Index nach Kanalname gespeichert.

Wenn für die Parquet-Daten kein Index über `name` zur Verfügung steht, wird die gesamte gespeicherte Tabelle durchsucht. Dadurch steigt die Laufzeit des Lesens auf ungefähr 20 Minuten. Daraus kann abgeleitet werden, dass ein Lesen von kleinen Untermengen der Daten ohne Index dringend vermieden werden sollte.

	Postgres		Parquet	
Messung Nr.	kurz	lang	kurz	lang
1	24,617	2500,902	8,037	429,986
2	24,607	2445,367	6,061	431,903
3	24,145	2487,382	7,402	426,510
Durchschnitt	24,456	2477,884	7,167	429,466

Tabelle 6.2. Zeitmessungen für das Laden von Messdateien aus Postgres und Parquet in Sekunden. Parquet mit Index nach `fid` gespeichert.

Kurze und lange Messdatei Bei diesem Test werden alle Kanäle einer kurzen Messung mit ungefähr 36 Millionen Messwerten und 10.000 Zeitstempeln und die längste verfügbare Messung mit circa 2,7 Milliarden Messwerten und 750.000 Zeitstempeln geladen. Auffällig ist, das Postgres-XL für das Laden der kurzen Messung nur wenig mehr Zeit benötigt als für die Zeitreihen aus dem ersten Anwendungsfall, obwohl es sich um die 1,5-fache Datenmenge handelt. Parquet zeigt bei dieser Messung, für die ein Index über `fid` genutzt wurde, gutes Skalierungsverhalten gegenüber dem Abruf der Daten über den `name`-

Index in den vorherigen Messungen. Postgres-XL ist beim Bereitstellen der großen Datenmenge 5,5 Mal langsamer als Parquet.

Zehn Kanäle aus allen Messungen Hier werden 10 Kanäle (`nmot_w`, `rl_w`, `wnwe_w`, `wnwa_w`, `wdkba_w`, `PCR_rGov_VW`, `pvd_w`, `ps_w`, `prist_w` und `tmot`) aus allen Messungen der Datenbasis geladen. Alle Kanäle zusammen umfassen ca. 117,6 Millionen Werte und 11,76 Millionen zugehörige Zeiten. Die Ergebnisse der Zeitmessung sind in Tabelle 6.3 dargestellt. Postgres-XL benötigt für die 5,5-fache Datenmenge im Vergleich zu einem einzelnen Messkanal über alle Messungen 2,6 Mal mehr Zeit, Parquet benötigt das 6,25-fache der Zeit.

Messung Nr.	Postgres	Parquet
1	89,016	33,655
2	88,031	21,665
3	88,738	19,702
4	86,849	18,902
5	87,571	17,953
6	88,432	17,431
7	88,546	17,332
8	89,369	16,630
9	89,225	16,817
10	88,767	17,637
Durchschnitt	88,454	19,772

Tabelle 6.3. Zeitmessungen für das Laden von 10 Kanälen in Sekunden. Parquet mit Index nach Kanalname gespeichert.

Mehrere Messungen Beim letzten Test soll überprüft werden, wie die Speichersysteme mit beim Abruf von großen Datenmengen skalieren. Dazu wurden Zeitreihen mit insgesamt ca. 8,62 Milliarden Werten und 2,4 Millionen Zeitstempeln ausgewählt. Das Laden dieser Datenmenge hat mit Postgres-XL viele Stunden benötigt, oder ist abgestürzt. Parquet als Datenquelle skaliert linear mit der Datenmenge. Abbildung 6.1 stellt das Skalierungsverhalten der beiden Datenquellen dar.

Messung Nr.	Parquet	Postgres
1	1534,829	∞
2	1528,124	∞
3	1519,341	∞
Durchschnitt	1527,431	∞

Tabelle 6.4. Zeitmessungen vieler Dateien in Sekunden. Die Datenmenge hat bei Postgres-XL zu Abstürzen geführt. Die Messungen konnten deshalb nicht abgeschlossen werden. Parquet mit Index nach `fid` gespeichert.

6.1.2. Resampling von Zeitreihen

Mit dem Resampling wird die wichtigste Transformation zwischen Schemata getestet, die in vielen Varianten implementiert ist. Andere in dieser Arbeit implementierte Transformationen werden nicht evaluiert, weil sie bei den aktuellen Daten nicht angewendet werden (Transponieren, Sliding Window) oder rein aus Spark-Aufrufen bestehen (z.B. Aufteilen des Array-Schemas auf mehrere Zeilen, Zusammenfassen des Wertespalten-Schemas zum Array-Schema).

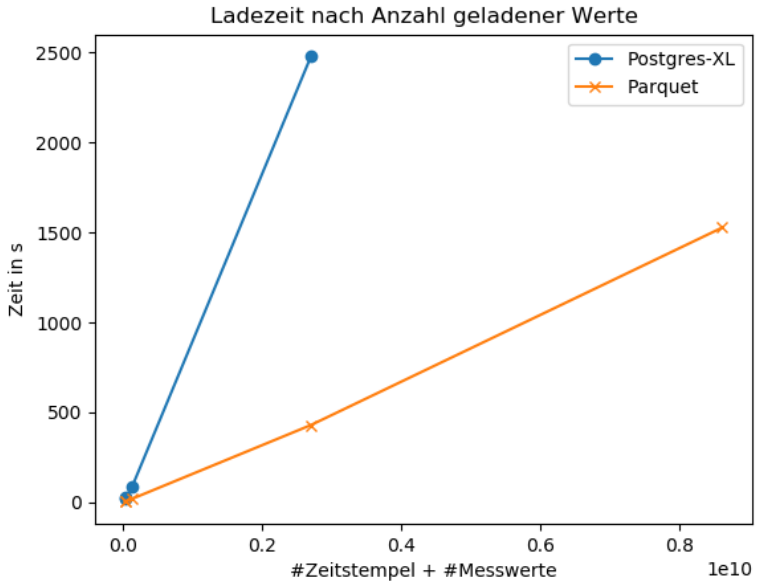


Abbildung 6.1. Skalierung der Abrufzeit nach Datenmenge.

Um Unterschiede in der Ausführungszeit der verschiedenen Implementierungen des Resamplings zu überprüfen, wurde folgende Methode getestet:

1. Lade die zehn Kanäle `nmot_w`, `rl_w`, `wnwe_w`, `wnwa_w`, `wdkba_w`, `PCR_rGov_VW`, `pvd_w`, `ps_w`, `prist_w` und `tmot` aus allen 43 Messungen.
2. Weise Spark an, das DataFrame im RAM zu cachen, um die Geschwindigkeit des Resamplings unabhängig vom Laden zu bestimmen.
3. Erstelle ein DataFrame durch Gruppieren nach Messung und Resampling mit
   a) neuer Zeitachse mit 0.01 s Abstand zwischen Messpunkten und Nutzung des Array-Schemas,
   b) neuer Zeitachse mit 0.01 s Abstand zwischen Messpunkten und Nutzung des Wertespalten-Schemas,
   c) neuer Zeitachse mit 0.1 s Abstand zwischen Messpunkten und Nutzung des Array-Schemas,
   d) mit der Zeitachse von `nmot_w` als Referenz,
   e) mit der Zeitachse von `pvd_w` als Referenz,
   f) einer beliebigen Zeitachse, weil alle Zeitachsen gleich sind.
4. Miss die Zeit zum Auswerten des DataFrames mit `show()`

Gemessene Zeiten für die Ausführung sind in Tabelle 6.5 festgehalten. Aus den Messergebnissen können folgende Erkenntnisse abgeleitet werden:

- Das gewählte Ausgabeschema erzeugt einen Laufzeitunterschied, der wahrscheinlich auf das Array-Schema zurückzuführen ist. Das Serialisieren von Arrays ist möglicherweise weniger optimiert als die Serialisierung eines festen Schemas.
- Der implementierte Algorithmus für das Resampling nach gegebener Zeitachse ist bei gleicher Länge der beteiligten Zeitreihen langsamer als der Algorithmus für das Resampling mit neuer Zeitachse. Das ist durch die Implementierung der Aggregationen zu erklären (siehe Abschnitt 5.2.2). Beim Resampling mit gegebener Zeitachse aggregiert der Algorithmus alle Zeiten und alle Werte der beteiligten univariaten Zeitreihen, bevor die Interpolation durchgeführt wird und alle nicht benötigten

Messung Nr.	a	b	c	d	e	f
1	28,147	24,443	4,199	34,899	32,893	19,906
2	24,635	22,905	3,790	30,011	31,427	20,656
3	25,980	23,578	3,863	30,079	30,915	19,928
4	25,160	23,522	3,967	32,836	31,732	21,106
5	26,632	22,801	3,999	30,857	31,514	20,801
6	24,990	22,236	4,155	32,374	29,827	21,061
7	26,560	24,987	3,797	30,316	28,981	19,316
8	26,582	21,487	3,672	32,439	32,317	23,964
9	27,355	23,571	3,978	30,860	29,366	22,008
10	25,612	22,593	3,735	32,422	29,644	19,922
Durchschnitt	26,165	23,212	3,915	31,709	30,862	20,867

Tabelle 6.5. Zeitmessungen für das Resampling in Sekunden. Spaltennamen nach der Erläuterung in Unterabschnitt 6.1.2. Ausgangsdaten im RAM vorhanden.

Zeitachsen verworfen werden. Beim Resampling mit neuer Zeitachse werden multivariate Zeitreihen als Zwischenergebnisse gebildet und die Übertragung von mehreren Zeitachsen wird eingespart.

- Wenn nur die Werte der Zeitreihen aggregiert werden [f], ist die nötige Ausführungszeit geringer. Die zusätzlich für diesen Anwendungsfall implementierten Funktionen sind zweckmäßig.
- Wenn die Zeitachsen durch das Resampling mit neuer Zeitachse deutlich kürzer werden (in [c] Verkürzung auf $\frac{1}{10}$), sinkt die Laufzeit im Vergleich zu allen anderen Tests deutlich. Um zu überprüfen, wie die Funktion mit der Länge der Zeitreihen skaliert, wurden Messungen mit unterschiedlichen Längenfaktoren durchgeführt. Die Ergebnisse werden in Abbildung 6.2 dargestellt. Die Funktion skaliert Linear mit der Anzahl der Werte in den Zeitreihen.

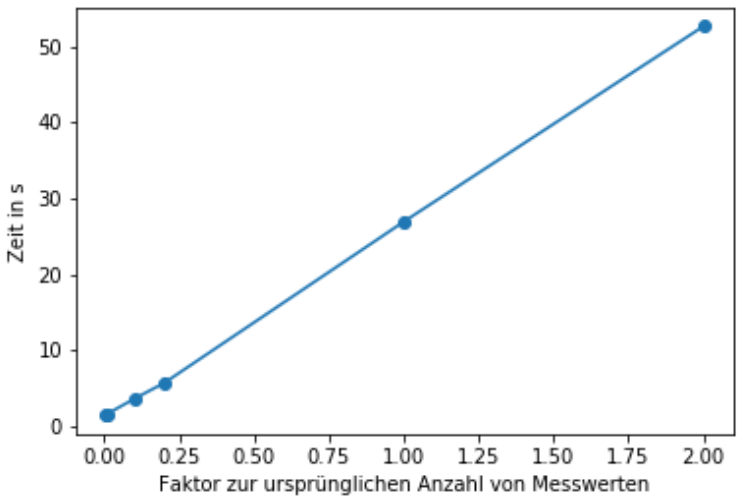


Abbildung 6.2. Benötigte Zeit für das Resampling nach Faktor für die Anzahl der Messwerte.

6.2. Bewertung der Datenquellen

Im Folgenden werden die beiden Systeme Postgres-XL und Apache Parquet in HDFS verglichen.

Die Anbindung beider Systeme ist über die DataSource-API von Spark SQL umgesetzt. Dadurch können Daten in Form von DataFrames geladen werden. Weil es sich um eine offizielle Schnittstelle handelt, ist die-

se in allen aktuellen Versionen vorhanden, stabil und wird aktiv weiterentwickelt. Die DataSource-API unterstützt die Verteilung von Postgres-XL nicht. Verbindungen werden zu einem festgelegten Koordinator aufgebaut. Parallele Nutzung mehrerer Koordinatoren findet nicht statt.

Beide Datenquellen unterstützen das in Spark verwendete Schema. So können die Zeitreihen ohne vorheriges Transponieren verwendet werden. Postgres-XL speichert die Daten normalisiert und komprimiert, was für geringen Speicherbedarf sorgt. Durch das denormalisierte Speichern in Parquet-Dateien ist der Speicherbedarf viel größer, im Gegenzug sind keine Verbundoperationen nötig.

Die Nutzung der Parquet-Files hat sich als weniger kompliziert als Postgres-XL herausgestellt. Der Entwickler muss Partitionen nicht durch die Angabe von Prädikaten erzeugen, Verbunde werden generell von Spark durchgeführt. Die Anwendung von Selektionen durch das Speichersystem hat bei beiden Datenquellen effektiv die übertragene Datenmenge begrenzt. Bei Parquet-Dateien hängt die Geschwindigkeit der Selektionen stark von der Partitionierung im Dateisystem ab. Ist kein Index für das genutzte Attribut verfügbar, müssen die Daten geladen und im Arbeitsspeicher durchsucht werden. Die Erstellung von Indexen erfordert eine redundanten Speicherung der Daten und sollte nur auf schwach selektiven Attributen genutzt werden, um die Blockgröße des HDFS mit großen Dateien auszunutzen.

Die Lesegeschwindigkeit der Parquet-Datenquelle war der von Postgres-XL in allen Tests um mindestens Faktor 3 überlegen. Trotzdem ist sie deutlich niedriger als die theoretische Geschwindigkeit der genutzten Speichermedien. Das Lesen von Datenmengen im Gigabyte-Bereich aus Postgres-XL ist im Laufe der Tests häufig gescheitert. Die Nutzung von Apache Parquet war unabhängig von der Menge der Daten problemlos.

Insgesamt wird Apache Parquet als Speicherformat für die Messdaten zur Analyse in Apache Spark bevorzugt. Mit höherer Lesegeschwindigkeit und Stabilität ist das Dateiformat Postgres-XL in der aktuellen Konfiguration in vielen Kriterien überlegen. Die Kompression der Daten und die Selektion nach beliebigen Attributen, die bei Parquet-Dateien mit hohem Zeitaufwand verbunden ist, sind Vorteile von Postgres-XL.

6.3. Verbesserte Konfiguration von Postgres-XL

Im Laufe der Erarbeitung der Messwerte wurde die Performance von Postgres-XL mehrmals signifikant verbessert. Nachdem die oben vorgestellten Messungen abgeschlossen wurden, konnten die Antwortzeit um bis zu 60 % verkürzt werden. In Tabelle 6.6 sind als Beispiel die Messergebnisse für das Abrufen von 10 Kanälen aus allen verfügbaren Messungen vor und nach den Optimierungen abgebildet. Durch diese Verbesserung allein ändert sich das Fazit aus Abschnitt 6.2 nicht. Parquet ist weiterhin die schnellere Datenquelle, stabiler und einfacher mit Spark benutzbar. Es erscheint jedoch sinnvoll, Postgres-XL weiter zu untersuchen, um die Performance besser auszureizen.

6.4. Beispielanalysen

Um die Nutzbarkeit des Systems zu bewerten, wurden verschiedene Beispielanalysen implementiert. Das Plotten von Grafiken (Unterabschnitt 6.4.1), das Finden von Ausschnitten in multivariaten Zeitreihen (Unterabschnitt 6.4.2), die Standardberichte (Unterabschnitt 6.4.3) und das Durchführen von arithmetischen Operationen auf Zeitreihen (Unterabschnitt 6.4.4) sind vereinfachte, reale Anwendungsfälle, die im Folgenden vorgestellt werden. Der Code für die Implementierung ist im Anhang A zu finden.

Messung Nr.	Postgres alt	Postgres neu
1	89,016	38,302
2	88,031	37,348
3	88,738	37,457
4	86,849	38,736
5	87,571	39,353
6	88,432	37,983
7	88,546	36,560
8	89,369	36,597
9	89,225	37,186
10	88,767	38,112
Durchschnitt	88,454	37,763

Tabelle 6.6. Zeitmessungen für das Laden von 10 Kanälen in Sekunden. Vergleich von zwei Konfigurationen von Postgres-XL.

6.4.1. Erzeugen von Grafiken

Grafiken und Plots dienen zur verständlichen Darstellung von Analyseergebnissen. Deshalb sollte ein Analysesystem in der Lage sein, diese zu generieren. Es wurde eine UDF implementiert, die die Zeitreihen in einem DataFrame in Grafiken plottet und diese als `bytearray` in einer neuen Spalte im DataFrame abspeichert. Weil Spark das Speichern von Binärdaten aus Feldern eines DataFrames nicht unterstützt, muss die Spalte zum Speichern aus Spark extrahiert werden und durch das Driver Program gespeichert werden. So werden die unterschiedlichen Zeitreihen parallel in Plots umgewandelt, die sequentiell abgespeichert werden. Das Erstellen einzelner Grafiken wird nicht parallelisiert. Für die Ausgabe im produktiven Einsatz sollten die Plots um wichtige Informationen wie Einheiten, Titel und Legenden ergänzt werden. An dieser Stelle wird darauf zur Verringerung der Komplexität verzichtet.

6.4.2. Finden von Ausschnitten in mulivariaten Zeitreihen

Als Beispiel für die Durchführung von Analysen auf Zeitreihen mit Python ist in Programmcode 6.1 die Erstellung einer Python-UDF dargestellt. Die gegebene UDF sucht in der Zeitreihe `nmot_w` nach allen Werten größer als Null und schneidet die gesamte multivariate Zeitreihe auf diesen Bereich zu.

An dieser Stelle soll auf zwei Besonderheiten eingegangen werden: Die Eingabe- und Ausgabetypen der Funktion `finder` und die Angabe des Typs im Aufruf von `udf`. Bei den Eingabe- und Ausgabetypen von Funktionen werden die Vektoren aus Spark MLlib verwendet. Innerhalb der Funktion können mit `Vector.toArray()` NumPy-Arrays aus den Vektoren entnommen werden und mit `Vectors.dense()` die Arrays in Vektoren verpackt werden. Wichtig bei der Erstellung der UDF ist die Angabe des Typs als zweiter Parameter der Funktion `udf`. Weil die Funktion eine Zeitreihe erzeugt, handelt es sich hier um einen strukturierten Typ. Anhand der Angabe des Typs verfolgt Spark die Änderung des Schemas eines DataFrames.

6.4.3. Standardberichte

Beim Standardbericht (siehe Algorithmus 1) werden nacheinander aus mehreren Messungen zufällig fünf Kanäle ausgelesen, ein Resampling auf eine Zeitachse durchgeführt und anschließend eine Funktion mit der erzeugten multivariaten Zeitreihe aufgerufen. Hierbei werden das Abrufen der Daten und das Resampling getestet.

```
def find_gt_zero(time, values, names, base):
  base_vector = values[names.index(base)]
  slicer = base_vector.toArray() > 0

  sliced_vectors = list()
  for v in values:
    arr = v.toArray()
    sliced_vectors.append(Vectors.dense(slice_array(arr, slicer)))

  sliced_time = Vectors.dense(slice_array(time, slicer))
  return sliced_time, sliced_vectors

def slice_array(array, slicer):
  if len(slicer) > len(array):
    return array[slicer[:len(array)]]
  else:
    part = array[:len(slicer)]
    return part[slicer]

def find_nmot_gt_zero(time, values, names):
  find_gt_zero(time, values, names, "nmot_w")

find_udf = udf(
  find_nmot_gt_zero,
  StructType(
    [
      StructField("time", VectorUDT()),
      StructField("values", ArrayType(VectorUDT()))
    ]
  )
)
```

Programmcode 6.1 Erstellen einer UDF zum Finden von Ausschnitten in einer Zeitreihe.

Data : files with channels
for $file \in files$ **do**
  for $sample \leftarrow 0$ **to** 100 **do**
    channels ← load_sample_channels(file, 5);
    ts ← resample(channels);
    call_subfunction(ts)
  end
end

Algorithmus 1 : Standardbericht.

Für die Implementierung der Standardberichte in Spark wird die Ausführung der inneren Schleife parallelisiert. Es werden 500 Messgrößen gleichzeitig aus der Datenbasis geladen und in Gruppen von fünf Signalen aufgeteilt. Diese Signale werden auf ein 100 ms-Zeitraster interpoliert. Um die Messgrößen und die Gruppen zufällig zu bestimmen, wird die Liste aller Dateien und enthaltener Kanäle aus der Datenbank abgerufen und ein Sample der Länge 500 pro Datei genommen. Der Liste von Kanal-IDs werden Gruppen-IDs zugeordnet und die entstandene Tabelle per Join in Spark den Daten zugeordnet, um anhand der Gruppen-ID das Resampling durchzuführen. Spezielle zusätzliche UDFs sind nicht nötig. Weitere Funktionen können über die neue Spalte auf das Ergebnis zugreifen. Der genutzte Programmcode ist wie der Code der anderen Analysen im Anhang zu finden.

6.4.4. Intervalle finden und arithmetische Operationen

Data : 10 files with channels
for $file \in files$ **do**
    channels ← load("nmot_w", "vfzg_w", "gangi", "rl_w", "lamsoni_w");
    ranges ← time ranges longer than 5 s with $nmot\_w > 0$ and $gangi > 0$;
    for $range \in ranges$ **do**
        values ← ∅;
        for $channel \in channels$ **do**
            values ← values ∪ { channel[range] };
        end
        resampled = resample(channels[range]);
        values ← values ∪ vector_div(resampled["vfzg_w"], resampled["nmot_w"]);
        values ← values ∪ vector_div(resampled["rl_w"], resampled["lamsoni_w"]);
        call_subfunction(values);
    end
end

Algorithmus 2 : Intervalle finden und arithmetische Operationen durchführen.

Mit diesem Anwendungsfall soll die Implementierung von komplexeren Analysen und Transformationen von Zeitreihen evaluiert werden. Es werden Zeitabschnitte gesucht, in denen eine Bedingung erfüllt ist. Anschließend werden diese Abschnitte aus allen fünf Zeitreihen ausgeschnitten und zusätzlich die Verhältnisse zwischen den Werten aus zweimal zwei Zeitreihen berechnet. Alle relevanten Werte werden abschließend an eine Funktion zur Weiterverarbeitung übergeben.

Der Algorithmus wurde wie bei der vorherigen Analyse angepasst, um die Parallelisierung von Apache Spark auszunutzen. Statt den Vorgang für zehn Messungen zu wiederholen, wird er für 43 Messdateien parallel ausgeführt. Das Resampling wird für alle Messgrößen auf die Zeitachse von `nmot_w` vorgenommen. Dies wird vor dem Suchen der Zeitbereiche durchgeführt. Statt der Übergabe der Ergebniswerte an eine nicht näher definierte Unterfunktion, wird eine neue Spalte mit den Ergebnissen gebildet. Die UDF arbeitet mit dem Array-Schema und erzeugt als Ergebnis ein Mapping von Kanalnamen auf geordnete Listen von Vektoren, die die gefundenen Abschnitte enthalten. Die berechneten Werte sind mit den Schlüsseln `div1` und `div2` im Mapping enthalten. Einer nächste Funktion kann die erzeugte Spalte als Eingabe verwenden.

Die Beispiele zeigen, dass die Analyse von Zeitreihen in Apache Spark mit einfachen Funktionen umgesetzt werden kann. Die Bereitstellung der Daten erfolgt über die parametrisierbare `load`-Funktion. Selektion, Projektion und Aggregation werden mit den Methoden der DataFrames oder mit im Rahmen dieser Arbeit implementierten Transformationen durchgeführt. Analyseschritte in den Zeitreihen werden als Python-Funktionen implementiert, die als Spark-UDFs parallel auf allen Zeitreihen ausgeführt werden. Verschiedene Analysen werden verkettet, indem die Ergebnisspalten der vorherigen als Eingabe für die nächsten UDFs verwendet werden.

6.5. Zukünftige Betrachtungen

In Zukunft sollten weitere zusätzliche Evaluationen des Systems vorgenommen werden, um das Konzept weiter zu untersuchen und die in dieser Arbeit beobachteten Effekte zu erklären.

Besonders interessant ist das Verhalten von Postgres-XL, das für diese Arbeit nicht stabil betrieben werden konnte und durch Konfigurationsanpassungen deutlich schneller im Bereitstellen der Daten geworden ist. Es stellt sich die Frage, warum die Performance trotz schneller Hardware niedrig ausfällt. Tests im Laufe der Arbeit haben erahnen lassen, dass das Problem mit der Nutzung von sehr großen Arrays zusammenhängt. Möglicherweise ist die Dekompression der Arrays aufwendig, oder sie werden wegen fehlender Optimierung häufiger als nötig zwischen den Datanodes von Postgres-XL versendet. Ein Vergleich mit einem nicht verteilten PostgreSQL könnte bei diesen Fragen aufschlussreich sein. Es stellt sich zusätzlich die Frage, wann die Dekompression durchgeführt wird. Am günstigsten wäre eine Dekompression durch den JDBC-Client des Spark-Executors, wahrscheinlicher ist jedoch, dass die Dekompression beim Lesen der Daten vom Sekundärspeicher stattfindet. Ein Vergleich mit einem Schema ohne Arrays könnte zeigen, ob der Arraydatentyp vorteilhaft ist.

Zusätzlich stellt sich die Frage, wie viel Overhead die JDBC-Verbindungen zwischen Postgres und den Spark-Executors erzeugt und wie der partitionierte Abruf der Daten verbessert werden kann. Die Notwendigkeit, Joins manuell auf die Systeme zu verteilen und Prädikate für die Partitionierung in Spark anzugeben, erweckt den Eindruck, dass die JDBC-Anbindung von Spark nicht für das Laden von Daten im Gigabyte-Bereich konzipiert ist. Nichtsdestotrotz sind Optimierungen der Partitionierung ein Ansatz zur weiteren Beschleunigung des Datenabrufs aus Postgres-XL, der evaluiert werden sollte.

In einer weiterführenden Arbeit könnte die Nutzung eines (teilweise) normalisierten Schemas in Parquet evaluiert werden. Zusätzlich können verschiedene Arten der Partitionierung verglichen werden. Dieser Ansatz ermöglicht den Vergleich der Join-Geschwindigkeit von Spark und Postgres-XL oder anderen relationalen Datenbanksystemen. So kann überprüft werden, wann das Ausnutzen der Joins von RDBMS mit anschließender Übertragung zu Spark schneller ist als Daten mit Spark zu laden und den Join intern durchzuführen.

Zudem sollte in weiteren Evaluationen die Ausführungszeit für ganze Analysen vom Laden der Daten bis zur Speicherung der Ergebnisse vorgenommen werden. Als erster Schritt könnte die Ausführungszeit der Beispielanalysen getestet werden. Dadurch wird die Geschwindigkeit der Speichersysteme nicht einzeln gemessen, sondern der Effekt der Optimierung von Spark mit in die Messung einbezogen. Durch das Anwenden von Projektionen in den Datenquellen und das Ausführen von Operationen beim Laden wird Spark in vielen Fällen schneller sein, als die oben vorgestellten Messungen suggerieren. Die Evaluation ganzer Analysen würde zusätzlich die Optimierung der verwendeten Algorithmen innerhalb von Spark mehr in den Fokus bringen. Es wäre interessant, das Skalierungsverhalten der implementierten Algorithmen zu untersuchen und es wenn möglich zu verbessern. Wichtigster Ansatz dafür ist die Partitionierung der Daten in Spark zur Minimierung von Kommunikation zwischen den Executors.

Ein Vergleich zwischen herkömmlichen Analysesystemen und dem Vorgeschlagenen Apache-Spark-Stack wurde in der Arbeit nicht vorgenommen, ist aber notwendig, um eine Entscheidung für oder gegen die Nutzung zu treffen. Ein Test mit mehr Speichersystemen, Daten und nach einschlägigen Anleitungen aufgebautem Cluster (zum Beispiel [The18c]) sollte vorgenommen werden.

7. Zusammenfassung und Ausblick

Diese Arbeit stellt ein Konzepte für die Analyse von Fahrzeugmessdaten mit Apache Spark vor und evaluiert dieses.

Dazu wurden Techniken zur Parallelisierung von Datenauswertungen mittels Datenparallelität zusammengefasst und Systeme und Formate untersucht, die zur Speicherung und Bereitstellung der Messdaten für Apache Spark geeignet sein könnten. Die Funktionsweise von Apache Spark und Erweiterungen für die Zeitreihenanalyse wurden genauer vorgestellt, um ihre Anforderungen und Fähigkeiten mit den Anforderungen und Zielen aus dem Anwendungsfall zu vergleichen.

Basierend auf der Analyse bestehender Erweiterungen wurden drei Schemata für die Verarbeitung von uni- und multivariaten Zeitreihen in DataFrames konzipiert und Transformationen zwischen diesen implementiert. Basisfunktionen für die Analyse der Zeitreihen und Änderung der Schemata wurden in Scala implementiert, um größtmögliche Performance zu erreichen. So ist eine Funktionsbibliothek für die Analyse von Fahrzeugmessdaten in Apache Spark entstanden. Die Anwender des Systems können Analysen in der weit verbreiteten Sprache Python entwickeln und als User Defined Functions ausführen. Dabei stehen durch die gewählten Schemata beim Funktionsaufruf vollständige uni- oder multivariate Zeitreihen zur Verfügung, was einen Übergang von hauptspeicherbasierter und nicht parallelisierter Analyse vereinfacht. Die Parallelisierung der Ausführung wird von Apache Spark transparent durchgeführt. Die Nutzerschnittstelle wird über Jupyter-Notebooks bereitgestellt und ermöglicht die browserbasierte Programmierung des Systems. Die Benutzung des Systems wurde in verschiedenen Beispielanalysen demonstriert.

Für den Einsatz von Apache Spark wurde ein Softwarestack entworfen, der unter anderem verschiedene Speichersysteme vorsieht. Im Rahmen der Arbeit wurden Postgres-XL und in HDFS gespeicherte Apache-Parquet-Dateien als Datenquellen evaluiert. Zum Abruf von Daten aus den Systemen wurden Funktionen entwickelt, die das einfache Anwenden von Selektionen zur effektiven Verkleinerung der geladenen Datenmenge ermöglichen und nicht mit SQL parametrisiert werden.

Bei der Evaluation der Speichersysteme hat sich Apache Parquet mit Stabilität und Geschwindigkeit als am besten geeignete Datenquelle im Vergleich erwiesen. Die Evaluation von MySQL, das zunächst ebenfalls angedacht war, wurde nach vergleichsweise langsamen ersten Laufzeittests abgebrochen. Postgres-XL konnte nicht stabil betrieben werden und erreichte deutlich geringere Lesegeschwindigkeiten, die sich durch Optimierung der Konfiguration verbesserten.

Das Ziel der Arbeit hat sich im Laufe der Bearbeitung von der reinen Datenvorbereitung und Anbindung unterschiedlicher Speichersysteme hin zur Betrachtung der Analyse der Daten in Apache Spark bewegt. Dies ist durch Schwierigkeiten beim Betrieb der Speichersysteme begründet. Weil die Anbindung mangels benutzbarer Systeme nicht implementiert und getestet werden konnte, wurden die zwei vorhandenen Systeme genutzt und zusätzlich Konzepte und eine Funktionsbibliothek für die Analyse der geladenen Daten in Apache Spark entwickelt und implementiert.

In Zukunft sollten weitere Speichersysteme für effizienteren Zugriff auf die Daten evaluiert werden. Dazu gibt es vier vielversprechende Ansätze:

- die Verwendung von tabellarischen Dateiformaten mit Index wie Apache Parquet unter Einsatz von Normalisierung
- die Verwendung von weiteren (No)-SQL-Datenbanksystemen wie Apache HBase und Vector in Hadoop
- die Verwendung spezieller Datenbanken für die Speicherung und Analyse von Zeitreihen
- die Entwicklung einer Spark-SQL-Datasource mit Indexen für MDF-Dateien oder ein auf die Datenstruktur angepasstes, anderes Format

Bei der Nutzung von Datenquellen mit eigenen Analysefähigkeiten wie z.B. relationalen Datenbanken ist die Verwendung weiterer Operatoren wie Join und Aggregationen aus dem Speichersystem zu evaluieren. Auch die Ausgabemöglichkeiten für nicht-tabellarische Ergebnisse in Spark sind erweiterungswürdig. Zudem könnte die Nutzung von User Defined Types, Datasets und tiefergreifenderen Erweiterungen in Spark SQL untersucht werden. Dies erfordert einen Wechsel auf Scala oder Java als Programmiersprache und könnte höhere Performance durch Optimierung, Kompression und Einsparung von Kommunikation mit Python-Interpretern ermöglichen. Des Weiteren muss die Benutzbarkeit und der Gewinn im produktiven Einsatz genauer untersucht werden. Eine Implementierung und Evaluation von realen Anwendungsfällen mit Vergleich zum herkömmlichen Vorgehen und Befragung der Analysten sind hierfür sinnvolle Ansätze. Dabei könnte die in dieser Arbeit begonnene Funktionsbibliothek anwendungsgetrieben weiterentwickelt werden.

Grundsätzlich lohnt sich der Einsatz eines parallelen, verteilten Systems wie Apache Spark nur, wenn die vorhandene Aufgabe nicht in angemessener Zeit auf einzelnen Maschinen gelöst werden kann. Im Kontext der Fahrzeugmessdaten bedeutet das: solange die Daten in den Hauptspeicher passen und die Zugriffslücke nicht die Ausführungszeit von Algorithmen dominiert, ist ein klassisches, nicht verteiltes System zu empfehlen.

Um einen weiteren Überblick über Big-Data-Analyseplattformen zu erhalten müssten weitere Systeme wie Apache Flink oder Hadoop MapReduce untersucht werden. Systeme, die parallel zum Abruf der Daten aus dem Speichersystem mit der Analyse beginnen, haben hohes Potential.

Literaturverzeichnis

[Act19a] Actian Corporation. *In-Memory Columnar Database for Big Data | Vector Analytic Database*. Actian. 2019. URL: https://www.actian.com/analytic-database/vector-analytic-database/ (besucht am 18. 11. 2018).

[Act19b] Actian Corporation. *Repository for the Spark-Vector connector. Contribute to ActianCorp/spark-vector development by creating an account on GitHub*. original-date: 2016-02-10T15:30:30Z. 28. Jan. 2019. URL: https://github.com/ActianCorp/spark-vector (besucht am 08. 02. 2019).

[Afr11] Afrank99. *Datei:Java-Logo.svg*. In: *Wikipedia*. Page Version ID: 95163279. 24. Okt. 2011. URL: https://de.wikipedia.org/w/index.php?title=Datei:Java-Logo.svg&oldid=95163279 (besucht am 09. 02. 2019).

[Apa19] Apache HBase Team. *Apache HBase ™ Reference Guide*. 2. März 2019. URL: http://hbase.apache.org/book.html#datamodel (besucht am 02. 03. 2019).

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

[ASA19] ASAM e.V. *ASAM MDF - Wiki*. 2019. URL: https://www.asam.net/standards/detail/mdf/wiki/ (besucht am 07. 02. 2019).

[Bar16] John Bard. *Accelerating Spark with Actian Vector in Hadoop*. Actian. 9. Mai 2016. URL: https://www.actian.com/company/blog/accelerating-spark-with-actian-vector-in-hadoop/ (besucht am 08. 02. 2019).

[Dag18] Barthélémy Dagenais. *Welcome to Py4J — Py4J*. 2018. URL: https://www.py4j.org/ (besucht am 28. 02. 2019).

[Dam16] Jules Damji. *A Tale of Three Apache Spark APIs: RDDs vs DataFrames and Datasets*. Databricks. 14. Juli 2016. URL: https://databricks.com/blog/2016/07/14/a-tale-of-three-apache-spark-apis-rdds-dataframes-and-datasets.html (besucht am 07. 02. 2019).

[Dat19a] Databricks. *Databricks Unified Analytics Platform*. Databricks. 2019. URL: https://databricks.com/product/unified-analytics-platform (besucht am 02. 03. 2019).

[Dat19b] DataStax. *DataStax Enterprise | Enterprise Data Management*. DataStax: Active Everywhere, Every Cloud | Hybrid Cloud | Apache Cassandra | NoSQL. 2019. URL: https://www.datastax.com/products/datastax-enterprise (besucht am 02. 03. 2019).

[Fan18] Wenchen Fan. *[SPARK-22386] Data Source V2 improvements - ASF JIRA*. 16. Nov. 2018. URL: https://issues.apache.org/jira/browse/SPARK-22386 (besucht am 18. 02. 2019).

[Fly66] Michael J. Flynn. „Very high-speed computing systems“. In: *Proceedings of the IEEE* 54.12 (1966), S. 1901–1909.

[Gru16] Joel Grus. *Einführung in Data Science*. Übers. von Kristian Rother. 1. Auflage. OCLC: 936175093. Heidelberg: O'Reilly, 2016. 329 S. ISBN: 978-3-96009-021-2.

[Hal19] David Hall. *Breeze is a numerical processing library for Scala.: scalanlp/breeze*. original-date: 2009-07-08T23:22:52Z. 22. Feb. 2019. URL: https://github.com/scalanlp/breeze (besucht am 24. 02. 2019).

[HMD17] Amir Haroun, Ahmed Mostefaoui und François Dessables. „A Big Data Architecture for Automotive Applications: PSA Group Deployment Experience“. In: *Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGRID 2017, Madrid, Spain, May 14-17, 2017*. 2017, S. 921–928. DOI: 10.1109/CCGRID.2017.107. URL: https://doi.org/10.1109/CCGRID.2017.107.

[Hor18] Hortonworks Inc. *Manage Data-at-Rest and Deliver Big Data Analytics with Hortonworks Data Platform (HDP)*. Hortonworks. 2018. URL: https://de.hortonworks.com/products/data-platforms/hdp/ (besucht am 06. 11. 2018).

[IBM] IBM. *SQL Query - SQL Query | IBM*. IBM.com. URL: https://www.ibm.com/cloud/sql-query (besucht am 06. 11. 2018).

[IBM14a] IBM. *CREATE FUNCTION (SQL scalar, table, or row) statement*. IBM Knowledge Center. 24. Okt. 2014. URL: https://www.ibm.com/support/knowledgecenter/en/SSEPGG_10.5.0/com.ibm.db2.luw.sql.ref.doc/doc/r0003493.html (besucht am 08. 02. 2019).

[IBM14b] IBM. *Specifying a partitioning method*. 24. Okt. 2014. URL: https://www.ibm.com/support/knowledgecenter/en/SSZJPZ_11.5.0/com.ibm.swg.im.iis.ds.pxop.dev.doc/topics/specpartmeth.html (besucht am 07. 02. 2019).

[IBM19] IBM. *SPSS Statistics - Überblick*. 25. Jan. 2019. URL: https://www.ibm.com/de-de/products/spss-statistics (besucht am 11. 02. 2019).

[Li19] Jia Li. *[SPARK-24130] Data Source V2: Join Push Down - ASF JIRA*. 11. Jan. 2019. URL: https://issues.apache.org/jira/browse/SPARK-24130 (besucht am 18. 02. 2019).

[Luc+15] André Luckow u. a. „Automotive big data: Applications, workloads and infrastructures“. In: *2015 IEEE International Conference on Big Data, Big Data 2015, Santa Clara, CA, USA, October 29 - November 1, 2015*. 2015, S. 1201–1210. DOI: 10.1109/BigData.2015.7363874. URL: https://doi.org/10.1109/BigData.2015.7363874.

[Mar17] Marchete. *What is SSE and AVX? - SSE & AVX Vectorization*. CodinGame. 2017. URL: https://www.codingame.com/playgrounds/283/sse-avx-vectorization/what-is-sse-and-avx (besucht am 08. 02. 2019).

[Mat19] MathWorks. *MATLAB - MathWorks*. 2019. URL: https://de.mathworks.com/products/matlab.html (besucht am 11. 02. 2019).

[McK15] Wes McKinney. *Datenanalyse mit Python =: Python for data analysis*. Übers. von Christian Tismer und Kristian Rother. 1. Auflage. OCLC: 930534372. Heidelberg: O'Reilly, 2015. 467 S. ISBN: 978-3-96009-000-7.

[Mej18] Marcin Mejran. *Apache Spark: Scala vs. Java v. Python vs. R vs. SQL*. Mindful Machines. 26. Juni 2018. URL: https://mindfulmachines.io/blog/2018/6/apache-spark-scala-vs-java-v-python-vs-r-vs-sql26 (besucht am 15. 02. 2019).

[Mel+10] Sergey Melnik u. a. „Dremel: Interactive Analysis of Web-Scale Datasets“. In: *Proc. of the 36th Int'l Conf on Very Large Data Bases*. 2010, S. 330–339. URL: http://www.vldb2010.org/accept.htm (besucht am 18. 11. 2018).

[Mey18] Holger Meyer. „Datenbank-Anwendungsprogrammierung“. Vorlesung. Rostock, 4. Mai 2018.

[Ora19a] Oracle Corporation. *Database VLDB and Partitioning Guide*. 7. Feb. 2019. URL: https://docs.oracle.com/database/121/VLDBG/GUID-BE424ACC-F746-4CA8-973C-F578CF98FF10.htm#VLDBG00225 (besucht am 07. 02. 2019).

[Ora19b] Oracle Corporation. *MySQL :: MySQL Partitioning :: 3.2 LIST Partitioning*. 6. Feb. 2019. URL: https://dev.mysql.com/doc/mysql-partitioning-excerpt/5.7/en/partitioning-list.html (besucht am 07. 02. 2019).

[Ora19c] Oracle Corporation. *Using User-Defined Aggregate Functions*. Database Data Cartridge Developer's Guide. 2019. URL: https://docs.oracle.com/cd/B28359_01/appdev.111/b28425/aggr_functions.htm (besucht am 08. 02. 2019).

[ORe19a] O'Reilly Media, Inc. *O'Reilly Search: Python*. 2019. URL: https://ssearch.oreilly.com/?i=1;q=Python;q1=Books;x1=t1&act=pg_1 (besucht am 11. 02. 2019).

[ORe19b] O'Reilly Media, Inc. *O'Reilly Search: R*. 2019. URL: https://ssearch.oreilly.com/?q=R (besucht am 11. 02. 2019).

[ÖV11] M. Tamer Özsu und Patrick Valduriez. *Principles of Distributed Database Systems*. 3. Aufl. New York: Springer-Verlag, 2011. ISBN: 978-1-4419-8833-1. URL: https://www.springer.com/de/book/9781441988331 (besucht am 12. 01. 2019).

[Pyt] Python Software Foundation. *Welcome to Python.org*. Python.org. URL: https://www.python.org/ (besucht am 11. 02. 2019).

[Rei07] James Reinders. *Understanding task and data parallelism*. ZDNet. 10. Sep. 2007. URL: https://www.zdnet.com/article/understanding-task-and-data-parallelism-3039289129/ (besucht am 22. 02. 2019).

[RXO16] Joshua Rosen, Reynold Xin und Sean Owen. *PySpark Internals - Spark - Apache Software Foundation*. 22. Nov. 2016. URL: https://cwiki.apache.org/confluence/display/SPARK/PySpark+Internals (besucht am 09. 02. 2019).

[Ryz19] Sandy Ryza. *A library for time series analysis on Apache Spark: sryza/spark-timeseries*. original-date: 2015-03-11T08:14:49Z. 29. Jan. 2019. URL: https://github.com/sryza/spark-timeseries (besucht am 07. 02. 2019).

[Sci19] SciPy developers. *SciPy.org — SciPy.org*. 2019. URL: https://www.scipy.org/ (besucht am 11. 02. 2019).

[Sha05] Yakov Shafranovich. *Common Format and MIME Type for Comma-Separated Values (CSV) Files*. Okt. 2005. URL: https://tools.ietf.org/html/rfc4180 (besucht am 07. 02. 2019).

[SSH11] Gunter Saake, Kai-Uwe Sattler und Andreas Heuer. *Datenbanken: Implementierungstechniken*. 3. Aufl. OCLC: 844941544. Heidelberg: mitp, 2011. 630 S. ISBN: 978-3-8266-9156-0.

[SSH18] Gunter Saake, Kai-Uwe Sattler und Andreas Heuer. *Datenbanken: Konzepte und Sprachen*. 6. Auflage. OCLC: 1022119123. Frechen: mitp, 2018. 777 S. ISBN: 978-3-95845-777-5.

[The18a] The Apache Software Foundation. *Apache Parquet*. 2018. URL: http://parquet.apache.org/ (besucht am 18. 11. 2018).

[The18b] The Apache Software Foundation. *Cluster Mode Overview - Spark 2.4.0 Documentation*. 2018. URL: http://spark.apache.org/docs/latest/cluster-overview.html (besucht am 09. 02. 2019).

[The18c] The Apache Software Foundation. *Hardware Provisioning - Spark 2.4.0 Documentation*. 2018. URL: http://spark.apache.org/docs/latest/hardware-provisioning.html (besucht am 08. 03. 2019).

[The18d] The Apache Software Foundation. *Parquet Files - Spark 2.4.0 Documentation*. 2018. URL: http://spark.apache.org/docs/latest/sql-data-sources-parquet.html (besucht am 07. 02. 2019).

[The18e] The Apache Software Foundation. *RDD Programming Guide - Spark 2.4.0 Documentation*. 2018. URL: http://spark.apache.org/docs/latest/rdd-programming-guide.html (besucht am 11. 11. 2018).

[The18f] The Postgres-XL Global Development Group. *51.2. GTM and Global Transaction Management*. 25. Okt. 2018. URL: https://www.postgres-xl.org/documentation/xc-overview-gtm.html#XC-OVERVIEW-COORDINATOR (besucht am 02. 03. 2019).

[The18g] The Postgres-XL Global Development Group. *CREATE TABLE*. Postgres-XL Documentation. 25. Okt. 2018. URL: https://www.postgres-xl.org/documentation/sql-createtable.html (besucht am 07. 02. 2019).

[The18h] The Postgres-XL Global Development Group. *Overview | Postgres-XL*. 2018. URL: https://www.postgres-xl.org/overview/ (besucht am 07. 02. 2019).

[The19a] The Apache Software Foundation. *Apache Spark™ - Unified Analytics Engine for Big Data*. 2019. URL: http://spark.apache.org/ (besucht am 07. 02. 2019).

[The19b] The R Foundation. *R: The R Project for Statistical Computing*. 2019. URL: https://www.r-project.org/ (besucht am 11. 02. 2019).

[Two19] Two Sigma Open Source, LLC. *A Time Series Library for Apache Spark. Contribute to twosigma/flint development by creating an account on GitHub*. original-date: 2016-10-19T17:44:15Z. 6. Feb. 2019. URL: https://github.com/twosigma/flint (besucht am 07. 02. 2019).

[Unb08] Unbekannt. *Internet Archive Wayback Machine*. MIMD | Intel® Developer Zone. 19. Mai 2008. URL: https://web.archive.org/web/20131016215430/https://software.intel.com/en-us/articles/mimd (besucht am 08. 02. 2019).

[Unb19] Unbekannt. *Python Data Analysis Library — pandas: Python Data Analysis Library*. 2019. URL: https://pandas.pydata.org/ (besucht am 02. 03. 2019).

[Vec19] Vector Informatik GmbH. *Measurement Data Format MDF | Vector*. 2019. URL: https://www.vector.com/de/de/produkte/anwendungsgebiete/steuergeraete-kalibrierung/messen/mdf/ (besucht am 07. 02. 2019).

[WB 18] WB Advanced Analytics. *Using Scala UDFs in PySpark*. wbaa. 1. Feb. 2018. URL: https://medium.com/wbaa/using-scala-udfs-in-pyspark-b70033dd69b9 (besucht am 15. 02. 2019).

[Wik18] Wikipedia. *CSV (Dateiformat)*. In: *Wikipedia*. Page Version ID: 175517005. 29. März 2018. URL: https://de.wikipedia.org/w/index.php?title=CSV_(Dateiformat)&oldid=175517005 (besucht am 07. 02. 2019).

[Wik19a] Wikipedia. *Data parallelism*. In: *Wikipedia*. Page Version ID: 883775644. 17. Feb. 2019. URL: https://en.wikipedia.org/w/index.php?title=Data_parallelism&oldid=883775644 (besucht am 22. 02. 2019).

[Wik19b] Wikipedia. *MySQL*. In: *Wikipedia*. Page Version ID: 184522317. 8. Jan. 2019. URL: https://de.wikipedia.org/w/index.php?title=MySQL&oldid=184522317 (besucht am 07. 02. 2019).

[www08] www.python.org. *English: Python logo*. 6. Aug. 2008. URL: https://commons.wikimedia.org/wiki/File:Python_logo_and_wordmark.svg (besucht am 09. 02. 2019).

[XR15] Reynold Xin und Josh Rosen. *Project Tungsten: Bringing Apache Spark Closer to Bare Metal*. Databricks. 28. Apr. 2015. URL: https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html (besucht am 07. 02. 2019).

[Zah+10] Matei Zaharia u. a. „Spark: Cluster computing with working sets.“ In: *HotCloud* 10.10 (2010), S. 95.

[Zah+12] Matei Zaharia u. a. „Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing“. In: *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012*. 2012, S. 15–28. URL: https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia.

[ZB12] Marcin Zukowski und Peter A. Boncz. „Vectorwise: Beyond Column Stores“. In: *IEEE Data Eng. Bull.* 35.1 (2012), S. 21–27. URL: http://sites.computer.org/debull/A12mar/vectorwise.pdf.

Anhang A. Programmcode der Beispielanalysen

```
import matplotlib.pyplot as plt
import io
def plotAll(time, vectors):
  for v in vectors:
    plt.plot(time, v)
  buffer = io.BytesIO()
  plt.savefig(buffer, format="png")
  buffer.seek(0)
  bytes = buffer.read(-1)
  buffer.close()
  return bytes

plot_udf = udf(plotAll, BinaryType())
```

Programmcode A.1 Vereinfachter Beispielcode für das Plotten von multivariaten Zeitreihen.

```
plot_list = df.select('plot').collect()
plot_list = [row[0] for row in plot_list]
for i, plot in enumerate(plot_list):
  with open("/tmp/" + str(i) + ".png", "wb") as file:
    file.write(plot)
```

Programmcode A.2 Speichern der Plots.

```
data_df = load_postgres_joined(spark, cid=cids, fid=fid)
group_df = spark.createDataFrame(groups)
joined = data_df.join(broadcast(group_df), ["cid"])
resampled = resample_new_array(joined, "t", "v", "name", "group", 0.1, 1.0)
resampled.show()
```

Programmcode A.3 Standardbericht: Gegeben ein Sample von 500 Kanal-IDs (`cids`) und eine Zuordnung in Gruppen (`groups`), sorgt der gegebene Code für ein Resampling der Gruppen.

```
df = load_postgres_joined(spark, name=["nmot_w", "vfzg_w", "gangi", "rl_w", "lamsoni_w"])
df.persist()
dfg = resample_given_array(df, "t", "v", "name", "fid", "nmot_w", 1.0)

def calc(time, vectors, names):
  nmot = vectors[names.index("nmot_w")].toArray()
  gangi = vectors[names.index("gangi")].toArray()
  vfzg = vectors[names.index("vfzg_w")].toArray()
  rl_w = vectors[names.index("rl_w")].toArray()
  lams = vectors[names.index("lamsoni_w")].toArray()

  intervals = find_intervals(time.toArray(), nmot, gangi)

  sliced_vectors = dict()
  for i, v in enumerate(vectors):
    arr = v.toArray()
    sliced_vectors[names[i]] = [Vectors.dense(arr[start:end]) for start, end in intervals]

  div1 = [Vectors.dense(np.divide(vfzg.toArray(), nmot.toArray())) for vfzg, nmot in
  ↪  zip(sliced_vectors["vfzg_w"], sliced_vectors["nmot_w"])]
  div2 = [Vectors.dense(np.divide(rl_w.toArray(), lams.toArray())) for rl_w, lams in
  ↪  zip(sliced_vectors["rl_w"], sliced_vectors["lamsoni_w"])]

  sliced_vectors["div1"] = div1
  sliced_vectors["div2"] = div2

  return sliced_vectors

my_udf = udf(calc, MapType(StringType(), ArrayType(VectorUDT())))
calculated = dfg.withColumn("calc", my_udf("t", "v", "name"))
calculated.show()
```

Programmcode A.4 Finden von Abschnitten und arithmetische Operationen: Das Finden der Abschnitte wird durch die Funktion `find_intervals` erledigt. Die UDF enthält gewöhnlichen Python-Code.

Abkürzungsverzeichnis

CAN	Controller Area Network
CSV	Comma Seperated Values
DSL	Domain-specific Language
GPL	GNU General Public License
MDF	Measurement Data Format
MIMD	Multiple Instruction Multiple Data
MISD	Multiple Instruction Single Data
RAM	Random Access Memory
RDBMS	Relational Database Management System
RDD	Resilient Distributed Dataset
SIMD	Single Instruction Multiple Data
SISD	Single Instruction Single Data
UDAF	User Defined Aggregate Function
UDF	User Defined Function
UDT	User Defined Type
VM	Virtuelle Maschine

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Bachelorarbeit selbstständig und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe. Die aus fremden Quellen direkt oder indirekt übernommen Stellen sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Rostock, den 25.03.2019 _____