Institut für Parallele und Verteilte Systeme (IPVS)
Universität Stuttgart
Universitätsstraße 38
70569 Stuttgart
Germany

Bachelorarbeit

Realisierung des Zonenreferenzmodells auf Datenströmen

Fabian Geiger

Studiengang:	Softwaretechnik
Prüfer/in:	Prof. Dr.-Ing. Bernhard Mitschang
Betreuer/in:	Corinna Giebler
Beginnt am:	23.03.2020
Endet am:	18.11.2020

Kurzfassung

Durch die zunehmende Digitalisierung erheben und verarbeiten Unternehmen und Organisationen eine wachsende Menge verschiedenster Daten. Ein Konzept für eine Software-Plattform zur Verwaltung, Verarbeitung und Analyse solch großer Datenmengen ist ein *Data Lake*. Die Kernidee eines Data Lake besteht darin, sämtliche Rohdaten zu erfassen, zu speichern und erst dann zu verarbeiten, wenn diese verwendet werden müssen.

In einem *zonenbasierten Data Lake* werden Daten und Verarbeitungslogik abhängig vom Verarbeitungsgrad der Daten in verschiedene Zonen eingeteilt. Generell mangelt es in der wissenschaftlichen Literatur allerdings an Vorgehensweisen, Architekturbeschreibungen und Implementierungen von Data-Lake-Plattformen. Um eine Referenzarchitektur für zonenbasierte Data Lakes zu schaffen, entwickelten Giebler, Gröger et al. das *Zonenreferenzmodell*.

Diese Arbeit umfasst die Konzeption, den Entwurf, die prototypische Implementierung sowie die Evaluation einer zonenbasierten Data-Lake-Architektur unter Verwendung des Zonenreferenzmodells. Dabei soll die Datenübertragung und -verarbeitung mithilfe von Datenströmen geschehen. Das Zonenreferenzmodell kann grundsätzlich auf eine Datenstromverarbeitung angewandt werden, allerdings lässt sich in der wissenschaftlichen Literatur noch keine Beschreibung des Modells im Streaming-Kontext auffinden. Diese Arbeit liefert ein entsprechendes Konzept nach.

Anhand eines fiktiven Anwendungsszenarios, in welchem Daten zur Ausbreitung der Coronavirus-Pandemie gesammelt, verarbeitet und explorativ ausgewertet werden sollen, entstand eine zonenbasierte Data-Lake-Architektur, die Datensätze aus mehreren Datenstromquellen erhält und diese zur weiteren Nutzung transformiert, aggregiert und kombiniert. Die Implementierung erfolgte unter Einsatz der Technologien *Apache Spark*, *Apache Kafka* sowie *Apache Cassandra*.

Bei der Evaluation der entstandenen Architektur und Implementierung zeigte sich eine hohe Wiederverwendbarkeit der rohen und vorverarbeiteten Daten sowie eine gute Skalierbarkeit der Komponenten. Zudem wurde deutlich, dass die Datenverarbeitung über mehrere Zonen hinweg zwar erheblich langsamer geschieht als mithilfe einer einzelnen Applikation, sich aber dennoch im Echtzeitbereich bewegt. Für äußerst zeitkritische Anwendungsfälle ist die entstandene Implementierung allerdings ungeeignet.

Inhaltsverzeichnis

1 Einleitung 5

2 Grundlagen 8

2.1 Data Lakes 8

2.2 Zonenbasierte Data-Lake-Architektur 8

2.2.1 Grundkonzept zonenbasierter Data Lakes 9

2.2.2 Zonenreferenzmodell 9

2.3 Konzepte für die Datenstromverarbeitung von Big Data 11

2.3.1 Lambda-Architektur 11

2.3.2 Kappa-Architektur 12

3 Verwandte Arbeiten 14

3.1 Zonenbasierte Datenarchitekturen und Data Lakes 14

3.2 Datenstromverarbeitung innerhalb von Big-Data-Plattformen 16

4 Beschreibung des Anwendungsfalls 18

5 Spezifikation des Zonenreferenzmodells 21

5.1 Allgemeine Spezifikation des Zonenreferenzmodells für Datenströme 21

5.2 Spezifikation der Datenverarbeitung für den Anwendungsfall 24

6 Beschreibung der prototypischen Implementierung 29

6.1 Beschreibung der Architekturpatterns 29

6.2 Beschreibung der Komponenten 30

6.2.1 Data Streams Generator 32

6.2.2 Zonenapplikationen 33

6.3 Beschreibung der eingesetzten Technologien 35

6.3.1 *Spring Boot* Java-Applikation 35

6.3.2 Apache Kafka 35

6.3.3 *Apache Spark* und *Spark Structured Streaming* 36

6.3.4 Scala 38

6.3.5 Apache Cassandra 38

6.4 Implementierungsbeispiel 39

7 Evaluation 45

7.1 Bewertung der entstandenen Architektur und Implementierung 45

7.2 Evaluation der Echtzeitfähigkeit 48

8 Zusammenfassung und Ausblick 54

Abbildungsverzeichnis

Abbildung 1: Zonenreferenzmodell 9
Abbildung 2: Lambda-Architektur 11
Abbildung 3: Kappa-Architektur 12
Abbildung 4: Datenarchitektur nach Madsen 15
Abbildung 5: Pi-Architektur 17
Abbildung 6: Datenflüsse in der Zonenreferenzmodell-basierten Streaming-Architektur 21
Abbildung 7: Lambda- und Kappa-Architektur mit Datenströmen 23
Abbildung 8: Architektur- und Datenflussschaubild der entwickelten Architektur 31
Abbildung 9: Datenflüsse im *DataStreamsGenerator* 32
Abbildung 10: Datenfluss und Abhängigkeiten einer Zonen-Applikation 33
Abbildung 11: Schematische Darstellung einer Architektur um ein Kafka-Cluster 35
Abbildung 12: Aufbau einer Kafka-Topic mit mehreren Partitionen 36
Abbildung 13: Kafka Consumer Groups 36
Abbildung 14: Architekturschaubild von *Apache Spark* 37
Abbildung 15: Verortung der Messpunkte innerhalb des Datenflusses 49
Abbildung 16: Mittelwerte und Mediane der gemessenen Verarbeitungszeiten 50
Abbildung 17: Mittelwerte und Mediane der Verarbeitungszeiten in Variante 1 aufgeschlüsselt nach einzelnen Messpunkten 51

Tabellenverzeichnis

Tabelle 1: Überblick über die Anforderungen an die Data-Lake-Plattform 19
Tabelle 2: Datenverarbeitung in der Landing Zone 24
Tabelle 3: Datenverarbeitungen in Raw und Harmonized Zone 26
Tabelle 4: Datenverarbeitungen in Distilled und Delivery Zone 27

Verzeichnis der Listings

Listing 1: Hauptklasse der Harmonized Zone (gekürzt) 40
Listing 2: Datenprozessor der Länderdaten in der Harmonized Zone (gekürzt) 41
Listing 3: Konvertierung der JSON-strukturierten Nachricht in ein *DataFrame*-Objekt 42
Listing 4: Beispiel einer simplen *UserDefinedFunction* in der Harmonized Zone 42
Listing 5: *KafkaDataStreamWriter*-Implementierung 43
Listing 6: Implementierung des *KafkaForeachSink* 44
Listing 7: Mögliche Signaturen generischer Datentransformationsmethoden 48

Abkürzungsverzeichnis

API *„Application Programming Interface“*; Programmierschnittstelle

BI *„Business Intelligence“*; Geschäftsanalytik; systematische Analyse von Daten innerhalb eines Unternehmens. Bei einer **SSBI** (*„Self-Service Business Intelligence“*) greifen Nutzer aus den Fachabteilungen einer Organisation eigenständig auf die Datenmanagement-Plattform der Organisation zu, um die enthaltenen Daten auszuwerten und zu analysieren.

COVID-19 *„Coronavirus Disease 2019“*; durch das Coronavirus SARS-CoV-2 ausgelöste Krankheit, die im Dezember 2019 ausbrach und sich 2020 zu einer globalen Pandemie entwickelte

CQL *„Cassandra Query Language“*; eine an SQL (*Structured Query Language*) angelehnte Abfragesprache für Datenabfragen und Mutationen an eine *Apache-Cassandra*-Datenbank

CSV *„Comma-separated values“*; spalten- und zeilenbasiertes Datenformat in einer Textdatei

ERP *„Enterprise-Resource-Planning”*; bezeichnet die Steuerung und Planung von Ressourcen in einem Unternehmen. Ein **ERP-System** ist ein IT-System, das diese Aufgaben übernimmt

HTTP *„Hypertext Transfer Protocol“*; zustandsloses Datenübertragungsprotokoll in einem Rechnernetz – hauptsächlich dazu genutzt, Hypertext-Dokumente im Internet zu übertragen. **HTTP-POST-Requests** sind Anfragen an einen Server, welche in ihrem *Request Body* Daten enthalten, die vom Server weiterverarbeitet werden sollen.

JSON *„JavaScript Object Notation“*; kompaktes Datenformat

MPP *„Massively Parallel Processing“*; massiv-parallele Verarbeitung (von Datenströmen)

No-SQL-Datenbank *„not only SQL”*; nicht-relationale Datenbanken, die für die Speicherung und Verwaltung großer Datenmengen mit häufigen Lese- und Schreibzugriffen optimiert sind

RDBMS *„Relational Database Management System”*; das zu einer relationalen Datenbank gehörende Datenbank-Management-System

REST *„Representational State Transfer“;* Einfacher, zustandsloser Architekturstil, der die Kommunikation zwischen Komponenten von verteilten Systemen beschreibt. Eine **REST-Schnittstelle** oder **REST-API** ist eine einheitliche Schnittstelle für diese Kommunikation.

1 Einleitung

Die zunehmende Digitalisierung in zahlreichen Gesellschaftsbereichen führt dazu, dass Unternehmen und Organisationen eine wachsende Menge verschiedenster Daten erheben und verarbeiten. Allen voran in der industriellen Produktion werden große Datenmengen verwendet, um komplexe Produktionsprozesse automatisieren, analysieren und stetig optimieren zu können (*Industrie 4.0*). Aber auch das Gesundheitswesen ist ein bedeutendes Anwendungsfeld von Big Data [Fis2019]. In diesem Bereich werden enorme Datenquantitäten zur medizinischen Diagnostik [Man2018], zur individuellen Gesundheitskontrolle [Dim2016] oder auch zur Vorhersage von Epidemien und Pandemien [Pyn2015] genutzt.

In der Vergangenheit wurden Datenanalysen in Organisationen meist auf relativ statische Daten aus einer beschränkten Anzahl an Datenquellen durchgeführt [Yua2015]. In der Big-Data-Ära machen statische Daten oft nur einen kleinen Teil der Datengesamtheit aus, die von einer Organisation verarbeitet wird. Als Datenquellen für Big-Data-Analysen werden hingegen oft kontinuierliche Datenströme verwendet, die von Geräten und Sensoren generiert werden, die an das *Internet of Things* angeschlossen sind. Wenn ein Temperatursensor in einer Fertigungslinie mehrmals minütlich einen Temperaturwert übermittelt oder ein Gerät sämtliche Nutzerinteraktionen aufzeichnet, entstehen schnell große Datenmengen.

Datenströme als Übertragungsform eignen sich aber auch dann, wenn dynamische Daten aus verschiedenen Quellen an einer zentralen Stelle gesammelt werden. Ein Beispiel für einen solchen Anwendungsfall ist die Bekämpfung einer Pandemie. Dabei werden sich ständig aktualisierende Daten von Gesundheitsbehörden aus allen Teilen der Welt gesammelt, um so Prognosen über den weiteren Verlauf der Ausbreitung erstellen und strategische Entscheidungen treffen zu können. Während der COVID-19-Pandemie ab Anfang 2020 nutzte die John-Hopkins-Universität in Baltimore eine „halbautomatisierte lebende Datenstrom-Strategie“, um kumulierte Fallzahlen in nahezu Echtzeit sammeln, aggregieren und darstellen zu können [Gar2020]. Die Daten der Universität wurden Anfang 2020 weltweit als zentrale Datenquelle für Kennzahlen zur Ausbreitung der Krankheit genutzt [vgl. Ble2020].

Bei der Bekämpfung einer Pandemie muss eine große Menge an Daten ständig aktuell gehalten werden. Verschiedenste Daten können für die Analysen eine Rolle spielen. So wurde im Zuge der Corona-Krise beispielsweise diskutiert, Bewegungsdaten von Telekommunikationsprovidern auszuwerten, oder auch eine App entwickelt, mithilfe derer Nutzer von Fitness-Armbändern aufgezeichnete medizinische Daten wie Puls und Schlafrhythmus zur Auswertung durch ein Gesundheitsinstitut freigeben können. Die beschriebene Situation erfordert also ganzheitliche Datenanalysen auf Basis von äußerst umfangreichen, heterogenen und komplexen Datenmengen. Um den Wert der heterogenen Daten optimal ausschöpfen zu können, rückt bei der Datenspeicherung und -verarbeitung das Konzept des *Data Lake* in den Fokus [vgl. Gie2020a].

Ein Data Lake erfüllt drei grundsätzliche Aufgaben: Er muss „Daten erfassen, Daten verarbeiten und Daten zur weiteren Verwendung bereitstellen“ [Mad2015]. Zunächst werden in einem Data Lake sämtliche potenziell wertvolle Daten in ihrer Rohform erfasst. Auf Grundlage dieser Daten sollen später explorative und flexible Analysen ausgeführt werden können.

Speichert ein Data Lake aber ausschließlich Rohdaten, müssen diese für jede Analyse neu aufbereitet werden und gegebenenfalls unterschiedliche Datenquellen mehrfach integriert werden. Eine Kernidee des Data-Lake-Konzepts ist deshalb, häufig verwendete Daten in vorverarbeiteten Formaten verfügbar zu machen, ohne dabei Anwendungsfälle vorzudefinieren [Gie2020b].

Der populärste Ansatz der konzeptionellen Data-Lake-Architektur ist das Zonenmodell. In einem zonenbasierten Data Lake werden Daten je nach Verarbeitungsgrad in verschiedene Zonen eingeteilt. In den „äußeren" Zonen werden kurzlebige und rohe Daten abgelegt (z. B. *Landing Zone, Transient Zone*, *Raw Zone*), während vorverarbeitete, qualitätsgesicherte oder transformierte Daten in den „inneren" Zonen des Data Lake gespeichert werden (z. B. *Harmonized Zone, Trusted Zone*, *Refined Zone*) [vgl. Gie2020a; Gie2020b; Mad2015; Zik2015].

Da es in der Literatur an Referenzarchitekturen für zonenbasierte Data Lakes mangelt, entwickelten Giebler, Gröger et al. das *Zonenreferenzmodell* [Gie2020b]. In diesem Referenzmodell werden die verschiedenen Zonen detailliert und standardisiert beschrieben. Das Modell kann unabhängig davon angewendet werden, ob Batch-Daten, Datenströme oder beides als Datenquellen dienen. Allerdings existiert bisher noch keine Beschreibung des Zonenreferenzmodells für Datenströme.

Im Rahmen dieser Arbeit wird das Zonenreferenzmodell auf Datenströme angewandt. Anhand eines beispielhaften Anwendungsfalles, in welchem Daten zur Ausbreitung des Coronavirus gesammelt und ausgewertet werden sollen, wird ein Data Lake konzipiert und spezifiziert, der Datenströme erfasst, transformiert und bereitstellt. Die entstehende zonenbasierte Data-Lake-Architektur wird anschließend prototypisch implementiert. Schließlich wird – unter anderem durch Messungen der Latenz – die Echtzeitfähigkeit der Architektur evaluiert.

Ziel der Arbeit ist es, Erkenntnisse darüber zu gewinnen, wie sich das Zonenreferenzmodell auf Datenströme im Data Lake anwenden lässt. Neben der Architektur selbst sollen Erfahrungen bei der Konzeption, Spezifikation und Implementierung des Data Lake gewonnen werden. Die Arbeit soll dazu beitragen, eine der in [Gie2020a] identifizierten fünf großen Forschungslücken („Fehlende Data-Lake-Referenzarchitekturen") zu schließen, indem eine solche Referenzarchitektur für einen zonenbasierten datenstromverarbeitenden Data Lake entwickelt wird.

Die Arbeit wird wie folgt gegliedert:

In **Kapitel 2** werden grundlegende Data-Lake-Konzepte und -Architekturen – mit besonderem Fokus auf zonenbasierten Data Lakes – vorgestellt. Zudem erfolgt eine Beschreibung des Zonenreferenzmodells.

Kapitel 3 beinhaltet einen Überblick über verwandte Arbeiten, die sich mit der Architektur von zonenbasierten Data Lakes sowie Konzepten zur Datenstromverarbeitung im Big-Data-Kontext befassen.

In **Kapitel 4** folgt eine Beschreibung des beispielhaften Anwendungsfalles, anhand dessen im Folgenden eine Data-Lake-Architektur konzipiert und implementiert wird.

Die Spezifikation des zonenbasierten Data Lakes für Datenströme erfolgt in **Kapitel 5**. Dafür wird zunächst eine generische Architektur des zonenbasierten, datenstrom-verarbeitenden Data Lakes spezifiziert. Im Anschluss werden die einzelnen Datenverarbeitungsschritte, die sich aus den in Kapitel 4 definierten Anforderungen ergeben, detailliert beschrieben. Die Beschaffenheit und Struktur der Daten nach der abgeschlossenen Verarbeitung in jeder Zone wird anhand von Beispieldatensätzen verdeutlicht.

Kapitel 6 beinhaltet die Beschreibung der prototypischen Implementierung der Architektur. Dabei werden die eingesetzten Software-Architekturpatterns, die Plattform-Komponenten sowie die eingesetzten Schlüsseltechnologien beschreiben. Für die Kernkomponenten der Implementierung werden einige Codebeispiele aufgezeigt.

Die Evaluation in **Kapitel 7** enthält eine Bewertung der entstandenen Architektur und Implementierung. Des Weiteren wird die Echtzeitfähigkeit des Konzepts anhand von Latenzmessungen untersucht.

Kapitel 8 enthält eine Zusammenfassung der wichtigsten Ergebnisse und Erkenntnisse, benennt Forschungslücken und gibt einen Ausblick auf die zukünftigen Entwicklungen im Forschungsgebiet Data Lakes.

2 Grundlagen

Dieses Kapitel enthält zunächst eine Definition des Begriffs „Data Lake“ sowie eine Beschreibung des Konzeptes – auch in Abgrenzung zum klassischen Datenmanagement mit einem *Data Warehouse* (Kapitel 2.1). Anschließend wird das allgemeine Konzept der zonenbasierten Data-Lake-Architektur vorgestellt und im Speziellen das Zonenreferenzmodell beschrieben (Kapitel 2.2). Es folgt die Vorstellung zweier Architekturpatterns zur Datenstromverarbeitung innerhalb eines Data Lake (Kapitel 2.3).

2.1 Data Lakes

Ein Data Lake ist „eine skalierbare Datenmanagementplattform für analytische Zwecke“ [Gie2020a]. Die Kernidee einer solchen Plattform besteht darin, sämtliche Rohdaten zu erfassen und zu speichern und erst dann zu verarbeiten, wenn diese verwendet werden müssen [Dat2019]. Daten im Data Lake sind „polystrukturiert“, das heißt, es existieren darin sowohl strukturierte und semi-strukturierte als auch nicht-strukturierte Daten [Mar2015; Dul2015]. In einem Data Lake können große Volumina heterogener und komplexer Daten verwaltet und für explorative Analysen genutzt werden, ohne dass bestimmte Anwendungsfälle vordefiniert werden müssen [Gie2020a]. Nach Madsen sind die drei grundsätzlichen Aufgabenbereiche einer Data-Lake-Plattform die Datenerfassung, die Datenverarbeitung und die Datenbereitstellung [vgl. Mad2015].

Klassischerweise wurden in Organisationen oft Data Warehouses zum Datenmanagement verwendet. Ein Data Warehouse ist eine zentral organisiertes Datenmanagementsystem, in der ausschließlich strukturierte Daten abgelegt werden. Eingehende Rohdaten werden bei ihrer Speicherung nach einem bestimmten Schema strukturiert (*Schema on Write*) [Bal2017]. Die Rohdaten in ihrem Quellformat gehen dabei verloren.

Data Lakes sind im Gegensatz dazu große Rohdatenspeichersysteme (*Big Data Repositories)* [Hai2016]. Sämtliche Rohdaten werden abgespeichert – auch unstrukturierte Daten wie Webserver-Protokolle, Sensordatenströme, Bilder oder Videos, die sich aufgrund fehlender Schemata nicht für die Speicherung in einem Data Warehouse eignen würden. Im Data Lake erhalten die Rohdaten erst dann eine Struktur, wenn sie verwendet werden (*Schema on Read*) [Bal2017]. Die Rohdaten bleiben im Data Lake immer erhalten, auch dann, wenn sie bereits verarbeitet wurden. Dadurch können die Daten zu einem späteren Zeitpunkt für weitere Anwendungsfälle in anderer Form aufbereitet werden. Dies hat den Vorteil, dass zum Zeitpunkt der Datenerfassung noch nicht sämtliche Anwendungsfälle der Daten bekannt sein müssen.

2.2 Zonenbasierte Data-Lake-Architektur

In der Literatur werden verschiedene Konzepte für einzelne Teilaspekte eines Data Lake beschrieben. Es mangelt dort allerdings an durchgängigen Beschreibungen konkreter Data-Lake-Architekturen [Gie2020a].

Eine Data-Lake-Architektur umfasst sowohl konzeptionelle als auch technische Aspekte. Während die konzeptionelle Data-Lake-Architektur die Datenflüsse innerhalb des Data Lake beschreibt und die Grundlage für die Datenmodellierung darstellt, beschreibt die technische Architektur das Zusammenspiel der unterschiedlichen technischen Komponenten [Gie2020a].

In der Literatur werden zwei verschiedene Ansätze zur konzeptionellen Data-Lake-Architektur beschrieben – die *Data-Pond-Architektur* [Inm2016] und die *zonenbasierte Data-Lake-Architektur* [Mad2015]. Für die Anwendung des Zonenreferenzmodells auf Datenströme spielt die Data-Pond-Architektur keine Rolle und wird deshalb an dieser Stelle nicht genauer behandelt.

In Kapitel 2.2.1 wird die Kernidee zonenbasierter Data Lakes erläutert. Kapitel 2.2.2 widmet sich dem Zonenreferenzmodell, einer Referenzarchitektur für zonenbasierte Data Lakes.

2.2.1 Grundkonzept zonenbasierter Data Lakes

Ein zonenbasierter Data Lake basiert auf einer Datenarchitektur, in welcher die Daten je nach ihrem Verarbeitungsgrad in mehrere Zonen eingeteilt werden [vgl. Mad2015]. Die Zone bestimmt beispielsweise, ob in ihr rohe, bereinigte oder stark transformierte Daten vorliegen.

In der Literatur werden verschiedene Zonenarchitekturen beschrieben, die sich in Anzahl und Semantiken der Zonen unterscheiden [Gie2020a]. Diese Zonenarchitekturen vereint, dass sie alle über eine Rohdatenzone (*Raw Zone*) verfügen. Die Raw Zone entspricht oft dem „naiven" Verständnis eines Data Lake. In dieser landen die eingehenden Daten in ihrer rohen, unverarbeiteten Form. Innerhalb eines zonenbasierten Data Lake können dieselben Daten mehrmals in unterschiedlichen Verarbeitungsgraden vorliegen. Die Daten in der Raw Zone bleiben aber auch nach ihrer Verarbeitung immer erhalten [Gie2020a].

Aufgrund der unterschiedlichen Zonenbeschreibungen in der Literatur entwickelten Giebler, Gröger et al. 2020 das *Zonenreferenzmodell*, welches eine standardisierte Beschreibung einer zonenbasierten Data-Lake-Architektur darstellt [Gie2020b].

2.2.2 Zonenreferenzmodell

Das Zonenreferenzmodell umfasst sechs Zonen, die von den eingehenden Daten nacheinander durchlaufen werden können – aber nicht müssen: Je nach Art und Verwendung der jeweiligen Daten werden nicht sämtliche Datensätze zwangsläufig in allen Zonen verarbeitet. Die folgende Beschreibung des Referenzmodells basiert auf dem genannten Paper von Giebler, Gröger et al. [Gie2020b]. Abbildung 1 zeigt die Zonen des Zonenreferenzmodells sowie die Datenflüsse zwischen den Zonen.

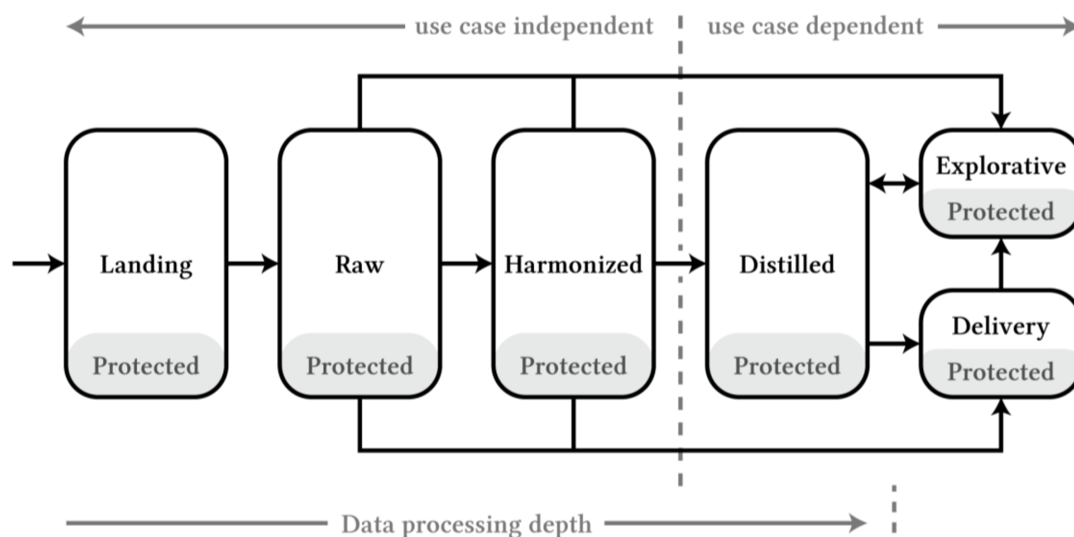


Abbildung 1: Zonenreferenzmodell
Eigene Darstellung, basierend auf [Gie2020b]

Im Zonenreferenzmodell gelangen eingehende Daten zunächst in die *Landing Zone*. Dort werden sie entweder unverändert weitergegeben oder sie durchlaufen geringe syntaktische Anpassungen. So könnten bereits in der Landing Zone Telefonnummern in ein einheitliches Format überführt [Mad2015] oder sensible Daten anonymisiert [Gie2020b] werden. Nutzer gelangen nicht an die Daten in dieser Zone und es erfolgt noch keine Persistierung.

Die Persistierung und Historisierung ist Aufgabe der *Raw Zone*. Sie ist der Kern eines Data Lake und muss deshalb zwangsläufig vorhanden sein. Um dem Konzept der Erhaltung der Rohdaten zu entsprechen (siehe Kapitel 2.1), dürfen die Rohdaten bei der Weitergabe an eine folgende Zone nicht aus der Raw Zone gelöscht werden. Sobald Daten die Raw Zone durchschritten haben, können Data Scientist auf sie zugreifen und mit ihnen arbeiten.

Eine Weitergabe der Rohdaten von der Raw Zone an die *Harmonized Zone* findet nur dann statt, wenn es eine Verwendung für die Daten gibt. Eine solcher Verwendungszweck kann auch erst mit der Zeit entstehen – die Überführung der Daten in die Harmonized Zone erfolgt immer bedarfsgesteuert. Daten in der Harmonized Zone sind nach einer standardisierten Modellierungstechnik strukturiert, gehören aber nicht alle zum selben Schema. Die harmonisierten Daten sind anwendungsfallunabhängig aufbereitet, es werden keine semantischen Änderungen an den Daten durchgeführt. Außerdem sind sie qualitätsgesichert und lassen sich bei Bedarf einfacher konsolidieren.

Eine solche Datenkonsolidierung kann in der *Distilled Zone* erfolgen. Die dort befindlichen Daten sind „destilliert“, das heißt, für bestimmte Verwendungszwecke aufbereitet. Hierzu werden Daten aggregiert, semantische Fehler korrigiert oder irrelevante Bestandteile der Daten entfernt. Wenn Datenanalysten später in einem neuen Anwendungsfall auf Daten zugreifen möchten, die in der Distilled Zone aus den Datensätzen entfernt wurden, müssen diese erneut aus der Raw oder Harmonized Zone gewonnen werden. In der Distilled Zone werden sie anschließend speziell für eine bestimmte Verwendung aufbereitet. Zusätzlich zu Data Scientists dürfen auch Domänenexperten auf die Daten in der Distilled Zone zugreifen.

Die *Delivery Zone* bietet nicht nur Datenanalysten, sondern einem breiten Spektrum an Nutzern Zugriff auf die aufbereiteten Daten. Die Daten in dieser Zone sind *„ready to use“*, also für bestimmte Anwendungsfälle präpariert oder für die Weiterverarbeitung mit bestimmten Tools vorbereitet. Die Menge an Schemata, in denen die aufbereiteten Daten vorliegen können, ist bestimmt durch die vielfältigen Möglichkeiten der Weiterverwendung. Es lässt sich also keine Aussage zur Struktur und Strukturiertheit der Daten treffen, jedoch ist die Datenqualität und -sicherheit (*Data Governance*) in dieser Zone am höchsten.

Die *Explorative Zone* dient Datenanalysten als Spielwiese, in die sie Daten aus sämtlichen anderen Zonen laden können. Auf Grundlage dieser Daten können explorative Analysen durchgeführt werden. Die hierbei gewonnenen Daten werden typischerweise nicht dauerhaft gespeichert, sondern nach einer gewissen Zeit gelöscht. Werden bei den Analysen Daten gewonnen, die sich für bestimmte Anwendungsfälle verwenden lassen, werden diese in die Harmonized oder Distilled Zone des Data Lake überführt. Da in der Explorative Zone Daten mit unterschiedlichen Datengranularitäten aus sämtlichen anderen Zonen beliebig verarbeitet werden können, ist die Data Governance in der Explorative Zone äußerst gering.

Alle beschriebenen Zonen verfügen über einen geschützten Bereich (*Protected Part*), in dem sensible, schützenswerte oder hinsichtlich Datenschutz besonders relevante Daten verarbeitet werden. Beim Zugriff und bei der Nutzung der Daten aus dem Protected Part gelten strengere Regeln, deshalb müssen meist verschiedenste Sicherheitsvorkehrungen in der Implementierung getroffen werden. Hinsichtlich Granularität, Strukturiertheit und Qualität obliegen die Daten im Protected Part aber den Eigenschaften ihrer jeweiligen Zone.

2.3 Konzepte für die Datenstromverarbeitung von Big Data

Da in dieser Arbeit das Zonenreferenzmodell auf Datenströme im Data Lake angewandt wird, sind Architekturkonzepte vonnöten, welche eine kontinuierliche Verarbeitung von Datenströmen im Data Lake ermöglichen. In der Literatur lassen sich zwei gängige Patterns für Streaming-Architekturen im Big-Data-Umfeld identifizieren: die *Lambda*- und die *Kappa-Architektur.*

2.3.1 Lambda-Architektur

Die Lambda-Architektur wurde 2011 von Nathan Marz vorgestellt [Mar2015] und bildet den Ausgangspunkt für die später entwickelte Kappa-Architektur.

Bei einer Lambda-Architektur (siehe Abbildung 2) kommen sämtliche Daten als Datenstrom in einer Datenaufnahme-Ebene (*Data Ingestion Layer*) im System an. Dort werden die Daten simultan auf zwei verschiedene Arten weiterverarbeitet. Alle ankommenden Datenströme werden durch den *Batch Layer* in einem *Master Dataset* gespeichert. In fest definierten Zeitintervallen werden sorgenannte *Batch Jobs* gestartet, die sämtliche zum Startzeitpunkt im Master Dataset befindliche Daten verarbeiten. Die Ergebnisse der Batch Jobs werden im *Serving Layer* bereitgestellt.

Zeitgleich werden eingehende Datenströme mithilfe des *Speed Layer* verarbeitet. Diese Verarbeitung geschieht sofort und meistens *in-memory* [Ber2017]. Die Daten werden sofort als Datenströme weiterverarbeitet, ohne sie in einem Master Dataset zwischenzuspeichern. Die aus dieser Echtzeit-Verarbeitung resultierenden Daten werden genutzt, um die Daten im *Serving Layer* zu aktualisieren bzw. zu inkrementieren [vgl. Mun2018].

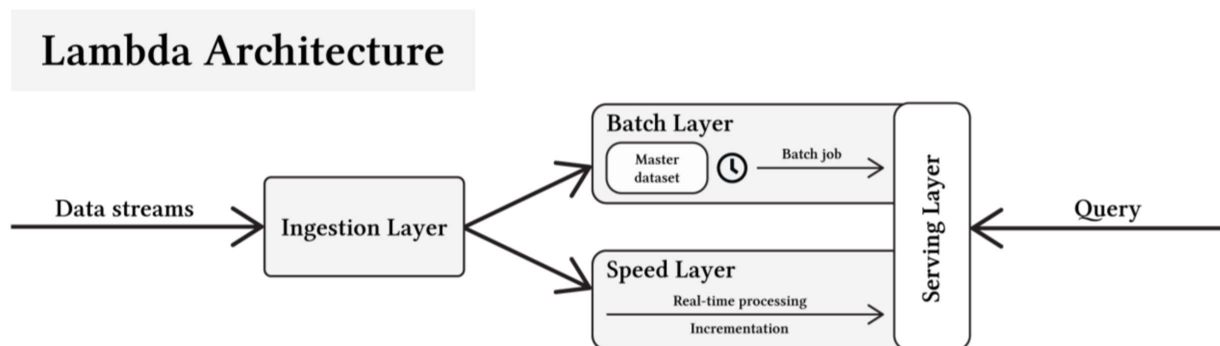


Abbildung 2: Lambda-Architektur

Eigene Darstellung auf Basis mehrerer Quellen [Mun2018; Ber2017; Gie2020a]

Die Funktionsweise einer Lambda-Architektur soll anhand eines Beispiels verdeutlicht werden, welches sich an einem von Lukas Berle in einem Vortrag beschriebenen Szenario orientiert [vgl. Ber2018].

Mehrere Sensoren eines IoT-Geräts senden im Millisekunden-Takt Messwerte an das System. Aus den Messdaten werden im Speed Layer Durchschnittswerte für jede Minute und jede Stunde des Tages berechnet und im Serving Layer bereitgestellt. Allerdings fällt im Beobachtungszeitraum für 30 Minuten die Internetverbindung eines Sensors aus. Einige Messdaten werden erst übermittelt, nachdem die Verbindung wiederhergestellt wurde. Da der Serving Layer nur die Messdaten aus den vergangenen fünf Minuten im Arbeitsspeicher hält, kann er die meisten der zu spät eintreffenden Daten nicht nachträglich bei der Ermittlung der Durchschnittswerte berücksichtigen.

Die berechneten Ergebnisse, die nun vom Serving Layer bereitgestellt werden, sind also auf Basis unvollständiger Messdaten entstanden und daher nicht korrekt. Da aber sämtliche Datenströme – auch die, die aufgrund des Ausfalls erst zu spät eingetroffen sind – im Master Dataset abgelegt wurden, kann ein Batch Job diese Fehler korrigieren: Zu einem fest definierten Zeitpunkt, zum Beispiel am Ende des Tages, werden die Berechnungen im Batch Layer auf Grundlage der vollständigen Daten im Master Dataset wiederholt. Die fehlerhaften Durchschnittswerte im Serving Layer werden durch den Batch Job mit den korrekten Berechnungen ersetzt.

Während der Speed Layer also eine Echtzeit-Verarbeitung von Datenströmen ermöglicht und so eine hohe Datenaktualität gewährleistet, stellt der Batch Layer langfristig die Datenqualität sicher.

2.3.2 Kappa-Architektur

Da der Speed Layer in einer Lambda-Architektur für die Datenaktualität sorgt, stellte sich Jay Kreps, ein Initiator bekannter Big-Data-Technologien wie *Apache Kafka*, die Frage, ob nicht auf eine Batch-Ebene verzichtet werden könne. Als Resultat seiner Überlegungen entstand die Kappa-Architektur (siehe Abbildung 3) analog zur Lambda-Architektur [Kre2014].

Auch bei der Kappa-Architektur erreichen alle Daten das System durch einen Data Ingestion Layer. Dort werden sie in sogenannten *Topics* organisiert. Außerdem existiert wie auch bei der Lambda-Architektur ein Master Dataset. An diesem Punkt verzichtet die Kappa-Architektur allerdings auf eine Batch-Ebene. Sämtliche Daten durchlaufen die Speed-Ebene und werden in verschiedenen *Jobs* kontinuierlich als Datenströme weiterverarbeitet. Die Verarbeitung im Speed Layer resultiert in einer Inkrementierung der Daten, die im Serving Layer bereitgestellt werden.

Die Daten werden zu einem späteren Zeitpunkt erneut verarbeitet, wenn sich die Datenverarbeitungslogik im Speed Layer verändert – beispielsweise durch ein Update des Datenprozessor-Programms. Die Daten im Master Dataset durchlaufen dann eine neue Version desselben Jobs wie die Datenströme, die bereits bei Ankunft im System kontinuierlich verarbeitet wurden. Aus der erneuten Verarbeitung resultieren aktualisierte Daten, die durch Abfragen an den Serving Layer bezogen werden können.

Abbildung 3: Kappa-Architektur
Eigene Darstellung, basierend auf [Ber2017]

Ein zentraler Vorteil der Kappa-Architektur ist der geringe Wartungs- und Pflegeaufwand. In einer Lambda-Architektur werden Batch und Speed Layer mithilfe unterschiedlicher Programme realisiert. Dies hat zur Folge, dass zwei Programme durch die Entwickler instandgehalten werden müssen. Wenn die Datenverarbeitung aktualisiert wird, müssen in der Regel beide Programme angepasst werden. Durch den Wegfall der Batch-Ebene wird bei einer Kappa-Architektur dagegen immer nur ein Programm gepflegt.

Auf der anderen Seite ist das Mantra der Kappa-Architektur – „Alles ist ein Datenstrom" – aber ein bedeutender Rechenkapazitäts- und Kostenfaktor. Dies hat zwei Gründe: Zum einen wird in einer Kappa-Architektur bei jeder Änderung der Verarbeitungslogik ein Job gestartet, der sämtliche Daten im Master Dataset erneut verarbeitet. In einer Lambda-Architektur würde nur ein Batch Job am Ende des definierten Zeitintervalls gestartet. Bei einer sich häufig ändernden Programmlogik kann die Job-Frequenz in einer Kappa-Architektur deshalb um ein Vielfaches höher liegen. Zum anderen ist auch die Datenmenge, die im Hauptspeicher gehalten werden muss, in einer Kappa-Architektur oft deutlich größer – beispielsweise, wenn zu einem späteren Zeitpunkt auf weit zurückliegende Daten zurückgegriffen werden muss [Ber2017]. Mario Meir-Huber kommt in einer Beispielrechnung anhand eines Anwendungsfalls auf dreizehnfach höhere Kosten einer Kappa-Architektur im Vergleich zu einer Lambda-Architektur [Mei2019].

Grundsätzlich kommen beide Datenstromverarbeitungskonzepte für eine Verwendung innerhalb einer Data-Lake-Architektur in Frage. Bei der Entscheidung für eines der Konzepte muss abgewogen werden, ob eine ressourcen- und kostensparende oder eine entwicklungs- und wartungsarme Lösung bevorzugt wird.

3 Verwandte Arbeiten

Im Rahmen dieser Arbeit wird eine kontinuierliche Datenstromverarbeitung innerhalb einer zonenbasierten Data-Lake-Architektur auf Basis des Zonenreferenzmodells realisiert. Daher baut sie vorwiegend auf wissenschaftlichen Arbeiten auf, die entweder zonenbasierte Data-Lake-Architekturen beschreiben (Kapitel 3.1) oder Konzepte für die Verarbeitung von Datenströmen im Big-Data-Umfeld erörtern (Kapitel 3.2).

Die Konzepte aus diesen verwandten Arbeiten werden im Folgenden zusammengefasst und diskutiert. Zudem werden die Forschungslücken identifiziert, welche durch diese Arbeiten noch nicht geschlossen werden konnten.

3.1 Zonenbasierte Datenarchitekturen und Data Lakes

Als Grundlage für die Entwicklung einer eigenen zonenbasierten Daten- und Data-Lake-Architektur wird auf Erkenntnisse bei der Konzeption ähnlicher Architekturen zurückgegriffen, die bereits in wissenschaftlicher Literatur dokumentiert wurden. Die Ergebnisse dieser Arbeiten werden nachfolgend diskutiert.

Eine Referenzarchitektur für zonenbasierte Data Lakes wurde im Paper **„The Data Lake Reference Architecture“ von Zaloni, Inc.** entwickelt [Zal2018]. Das Unternehmen, das sich auf die Architektur von Data Lakes spezialisiert hat, beschreibt darin einen Data Lake mit fünf Zonen.

Eingehende Daten landen darin zunächst in der *Transient Landing Zone*, in der sicherheits- und datenschutzrelevante Daten vor der Persistierung einige Sicherheitsvorkehrungen und Qualitätsprüfungen durchlaufen. Werden nur unkritische Daten verarbeitet, kann diese Zone entfallen. Von der Transient Landing Zone werden die Daten an die *Raw Zone* übermittelt. Dort werden sie in unveränderter, roher Form persistiert und historisiert, sodass sie später für verschiedenste Anwendungsfälle genutzt werden können.

Die *Trusted Zone* enthält standardisierte, qualitätsgesicherte Daten. In der *Refined Zone* werden die Daten in ein einheitliches Format überführt und durchlaufen weitere Mechanismen zur Qualitätssicherung. Die Daten sind nun dazu geeignet, Schemata aus ihnen abzuleiten oder sie für Datenanalysen und -visualisierungen zu verwenden.

Die letzte Zone wird in Zalonis Referenzarchitektur als *Sandbox* bezeichnet. Diese erlaubt es Datenanalysten und Managern, innerhalb neuer Anwendungsfälle und explorativer Analysen temporäre Daten zu generieren, die – abhängig von der Brauchbarkeit des Ergebnisses – entweder weiterverwendet oder verworfen werden. Zalonis Referenzarchitektur beinhaltet keine Konzepte zur Datenstromverarbeitung in einem zonenbasierten Data Lake.

Das von IBM veröffentlichte Buch **„Big Data Beyond the Hype“ von Zikopoulos** [Zik2015] enthält eine umfangreiche Sammlung von Konzepten und Architekturen für Software im Big-Data-Umfeld. Dabei ist für diese Arbeit insbesondere das Kapitel 4 relevant (*„The Data Zones Model: A New Approach to Managing Data“*), in dem das *Data Zones Model*, eine zonenbasierte Datenarchitektur, beschrieben wird.

Das Modell beinhaltet – wie auch das Zonenreferenzmodell – eine *Landing Zone* für Rohdaten. In der *Staging Zone* befinden sich die inhaltlich unveränderten Rohdaten, deren Struktur aber bereits vereinheitlicht bzw. an das Schema des Zielsystems angepasst wurde.

Die Daten aus der Staging Zone können dann in eine *Queryable Archive Zone* überführt werden, wo sie in einem verteilten Dateisystem abgelegt und mithilfe einer Abfragesprache durchsucht werden können.

Des Weiteren existiert eine *Explorative Zone* für explorative Analysen auf den rohen und verarbeiteten Daten aus Landing und Staging Zone sowie eine *Deep Analytics Zone* für komplexere Analysen unter Einbezug von RDBMS-Technologien. Die *Trusted Data Zone* wird mit einem Enterprise Data Warehouse realisiert und enthält ausschließlich aktuelle, geprüfte und häufig benötigte Daten. Die Funktion der Trusted Data Zone ähnelt dabei der der Delivery Zone im Zonenreferenzmodell von Giebler, Gröger et al. [Gie2020b].

Die für die Datenstromverarbeitung zentrale Zone aus Zikopoulos' Modell ist die *Real-Time Processing and Analytics Zone*. Diese ist für Daten vorgesehen, die in Echtzeit prozessiert und analysiert werden müssen. Streaming-Daten aus unterschiedlichen Quellen werden dort transformiert, aggregiert und ausgewertet. Der Eingangsdatenstrom besteht aus Rohdaten, der Ausgangsdatenstrom enthält gefilterte und/oder transformierte Datensätze. Aktionen können direkt durch die Logik in der Echtzeitverarbeitungs- und Analysezone gestartet oder – bei komplexeren Bedingungen – von einer separaten *Decision-Management-*Anwendung ausgelöst werden [vgl. Zik2015].

Auch **Madsens „How to build an Enterprise Data Lake"** [Mad2015] widmet sich einer Datenarchitektur bestehend aus mehreren Zonen.

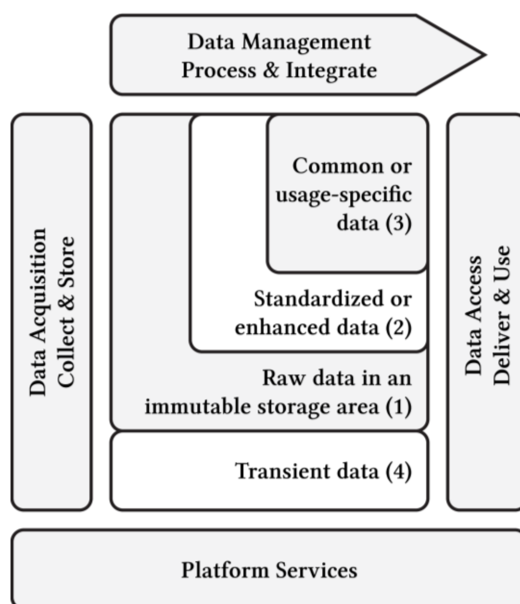


Abbildung 4: Datenarchitektur nach Madsen
Eigene Repräsentation der Grafik aus [Mad2015]

Er gruppiert die Daten je nach Verarbeitungstiefe in vier Bereiche (siehe Abbildung 4):

- unveränderlich persistierte Rohdaten (1)
- standardisierte, aufbereitete oder aus Rohdaten abgeleitete Daten (2)
- anwendungsfallspezifisch aufbereitete oder gemeinsam nutzbare Daten (3)
- transiente Daten, die nur während der Datenverarbeitung verfügbar sind (4)

Aus der Datenarchitektur leitet Madsen eine zonenbasierte Data-Lake-Architektur ab.

Madsens Zonenmodell umfasst eine *Collection Zone* für Rohdaten, eine *Management Zone* für standardisierte und strukturierte Daten, eine *Delivery Zone* für anwendungsspezifische oder generische Daten sowie eine *Transient Zone* für kurzlebige Daten während der Verarbeitung.

In der Transient Zone werden allerdings – anders als der Begriff vermuten lassen könnte – keine Datenströme verarbeitet, sondern Daten abgelegt, die in Zwischenschritten bei der Datenverarbeitung entstehen, zum Beispiel bei der Aufbereitung von Daten für Machine-Learning-Anwendungen. Trotzdem bezieht Madsen in seinen Überlegungen explizit auch Datenströme mit ein. Dies verdeutlicht er anhand der Persistierung in der Collection Zone.

Die Daten werden dort unveränderlich gespeichert – unabhängig davon, ob es sich um Daten aus Batch- oder Streaming-Datenquellen handelt [vgl. Mad2015].

Auch wenn sie sich in Anzahl der Zonen unterscheiden, haben die zonenbasierten Modelle von Zikopoulos [Zik2015], Madsen [Mad2015] und Zaloni, Inc. [Zal2018] eine große Überschneidung hinsichtlich der Gliederung ihrer Datenarchitekturen und der Semantik einiger Zonen. In allen Arbeiten existiert eine Übergangszone für eingehende Daten, eine Rohdatenzone zur Persistierung der Rohdaten, eine Zone zur einheitlichen Strukturierung der Daten, eine Zone für bereinigte und aufbereite Daten sowie einen Bereich, der für explorative Analysen mit transienten Daten bereitsteht. An diesen Erkenntnissen bezüglich der Data-Lake-Strukturierung hält auch das Zonenreferenzmodell von Giebler, Gröger et al. [Gie2020b] fest, welches in Kapitel 2.2.2 vorgestellt wurde.

Während die Referenzarchitektur von Zaloni keine Konzepte zur Datenstromverarbeitung enthält, beinhalten die Data Lakes von Madsen und Zikopoulos eine eigene Zone für Datenströme. Dennoch fällt es schwer, aus den Modellen eine prototypische Implementierung abzuleiten. Beide Beschreibungen beinhalten keine Details zur praktischen Umsetzung der Konzepte.

Zudem machen beide Modelle bei der Datenstromverarbeitung kaum Gebrauch von einem zentralen Vorteil von zonenbasierten Data Lakes. Der Data Lake wird in Zonen eingeteilt, damit die Daten in unterschiedlichen Granularitäten und Verarbeitungsgraden vorliegen und so später wiederverwendet oder weiterverarbeitet werden können. Dadurch dass die Echtzeitverarbeitung von Datenströmen bei Madsen und Zikopoulos ausschließlich in einer Zone stattfindet, kommt dieser Vorteil in beiden Modellen für Datenströme nicht zum Tragen.

Das Zonenreferenzmodell sieht hingegen die Verarbeitung von Datenströmen über alle Zonen im Data Lake hinweg vor. Allerdings fehlt es darin an Beschreibungen der einzelnen Zonen im Datenstrom-Kontext. Diese Arbeit soll dazu beitragen, diese Lücke zu schließen.

3.2 Datenstromverarbeitung innerhalb von Big-Data-Plattformen

Das bereits in Kapitel 3.1 erwähnte Buch von Zikopoulos widmet sich auch der Datenstromverarbeitung im Big-Data-Umfeld. Kapitel 7 („*'In the Moment' Analytics: InfoSphere Streams*") beschreibt ein Streaming-Konzept innerhalb einer Big-Data-Plattform, die Techniken zur massiv-parallelen Verarbeitung (MPP) nutzt. Das Kapitel fokussiert sich dabei auf die IBM-Softwareplattform *InfoSphere Streams*, die auch in der bereits beschriebenen Real-time Processing and Analytics Zone zum Einsatz kommt.

Zunächst beschreibt Zikopoulos den Anwendungsfall eines Krankenhauses, in welchem eine vorausschauende medizinische Versorgung gewährleistet werden soll. Dies geschieht mithilfe einer Streaming-Anwendung, die medizinische Daten von kritischen Patienten sammelt und analysiert. Nach der Beschreibung des Szenarios definiert Zikopoulos Streaming-Applikationen im Allgemeinen als „Graphen mit Knoten und gerichteten Kanten", wobei die Knoten den Datenprozessoren und die Kanten dem Datenfluss zwischen diesen Prozessoren entsprechen [vgl. Zik2015]. Anschließend geht er sehr konkret auf die zugrundeliegende Technologie InfoSphere Streams und darauf ausgerichtete Implementierungskonzepte ein, weshalb die folgenden Inhalte dieses Kapitels an dieser Stelle nicht weiter besprochen werden.

Marschalls und Baars „Pi-Architektur" [Mar2019] enthält ein Konzept zur Verarbeitung von Datenströmen in Data Lakes. Anders als die Lambda- und Kappa-Architektur ist die Pi-Architektur kein Konzept zur Datenstromverarbeitung an sich, sondern stellt eine Möglichkeit der Zusammenwirkung verschiedener Komponenten einer *Self-Service Business Intelligence* (SSBI) dar. Abbildung 5 zeigt die Komponenten einer Pi-Architektur.

Die Pi-Architektur berücksichtigt auch (aber nicht ausschließlich) eine Verarbeitung von Daten in Form von kontinuierlichen Datenströmen. Die Datenstromverarbeitung geschieht dabei parallel zu einem *Enterprise Data Reservoir*, welches sowohl ein Data Warehouse als auch einen Data Lake enthält. Während der Datenverarbeitungsphase werden Datenströme mit dem Enterprise Data Reservoir ausgetauscht.

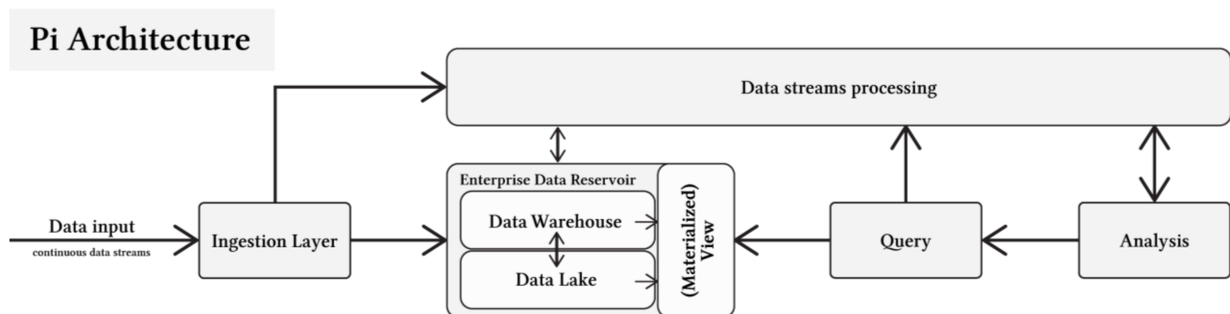


Abbildung 5: Pi-Architektur
Eigene Darstellung, basierend auf [Mar2019]

Marschall und Baars berücksichtigen dabei zwar die Verarbeitung von Datenströmen in ihrem Architekturschaubild, vertiefen aber im zugehörigen Paper nicht, wie die Datenstromverarbeitung mit dem Data Lake integriert wird und wie genau der Datenaustausch zwischen dem Enterprise Data Reservoir und den Datenstrom-Komponenten vonstattengeht.

Munshi und Mohamed [Mun2018] realisieren einen cloudbasierten Big Data Lake für smarte Energiesysteme. Als Datenquellen erhalten sie Datenströme von Energieerzeugern (z. B. Windkraftwerke, Photovoltaikanlagen) und Konsumenten (z. B. Privathaushalte, Industrieanlagen, E-Auto-Ladestationen) sowie Daten von Wetterstationen. Die Daten sind polystrukturiert und beinhalten neben Sensordaten von intelligenten Stromzählern und anderen Messwerten auch unstrukturierte Daten wie Bilder und Kamera-Aufzeichnungen.

Um die massive Menge der heterogenen Daten echtzeitnah verarbeiten zu können, konzipieren Munshi und Mohamed eine Data-Lake-Plattform unter Verwendung einer Lambda-Architektur für Datenströme, die die Vorteile von Batch- und Echtzeit-Verarbeitung kombiniert. Aufgrund ähnlicher Anforderungen der Anwendungsfälle hinsichtlich der Verarbeitung von Datenströmen lassen sich einige von Munshi und Mohameds Konzepten auf die zu entwickelnde Data-Lake-Architektur übertragen. Allerdings widmet sich das Paper ausschließlich dem Datenstrom-Aspekt, es werden keine Aussagen zur Datenarchitektur getroffen und auch keine Data-Lake-Zonen spezifiziert.

Diese Aussage lässt sich auf alle in diesem Unterkapitel genannten Paper übertragen. Sämtliche Arbeiten wenden verschiedene Konzepte zur Datenstromverarbeitung innerhalb einer Data-Lake-Plattform an. Allerdings wird in keinem Paper explizit eine Zonenarchitektur verwendet, sodass es in der Literatur an Beschreibungen von Streaming-Architekturen im Kontext eines zonenbasierten Data Lake fehlt.

4 Beschreibung des Anwendungsfalls

Der Anwendungsfall, auf dessen Basis ein Data Lake konzipiert werden soll, bezieht sich auf die Coronavirus-Pandemie, die im Dezember 2019 in China ihren Ursprung fand.

Die internationale Presse nutzt für die Berichterstattung häufig Datenbestände der John-Hopkins-Universität in Baltimore, USA, als Grundlage. Die Universität hat dazu eine öffentlich einsehbare Karte in Form einer Web-Applikation entwickelt, die die Anzahl der weltweit mit dem Coronavirus Infizierten nach Territorium aufschlüsselt.[1]

Ein Territorium ist dabei entweder ein ganzes Staatsgebiet, eine Region oder eine Gemeinde. Regionen entsprechen den föderalen Verwaltungseinheiten eines Staates, wie zum Beispiel den US-amerikanischen Bundesstaaten oder den chinesischen Provinzen. Die USA ist im November 2020 der einzige Staat, für den die John-Hopkins-Universität Daten zur Coronavirus-Pandemie auf kommunaler Ebene pflegt. Die Infektions- und Todeszahlen liegen hier für jede Gemeinde (*County*) einzeln vor. Für ausgewählte große oder durch das Coronavirus besonders betroffene Staaten sammelt die Universität Fallzahlen auf regionaler Ebene – neben den USA zum Beispiel für China, Russland, Deutschland, Spanien und Italien. Die Fallzahlen vieler Staaten werden aber in einem einzelnen Datensatz gepflegt. Das Territorium entspricht hier dem gesamten Staatsgebiet.

Die Karte der John-Hopkins-Universität dient für den Anwendungsfall einer Data-Lake-Architektur als Inspiration. Die Parameter der Datengewinnung und -verarbeitung werden dabei allerdings stark vereinfacht.

Der Anwender im fiktiven Anwendungsszenario ist eine beliebige Universität, die – wie auch die John-Hopkins-Universität in der Realität – Daten über den Pandemieverlauf von den verschiedenen Gesundheitsämtern auf der Welt zur Verfügung gestellt bekommt. Zusätzlich erhält die Universität Wetterdaten für jedes Territorium, die sie mit den Pandemie-Daten kombinieren und Korrelationen untersuchen soll.

Die Daten, die die Universität erhält und generiert, sollen der Öffentlichkeit, der Presse und weiteren Forschungsinstitutionen zugänglich gemacht werden. Für die breite Öffentlichkeit sollen die Daten aufbereitet und auf dem Dashboard einer neuen Web-Applikation visualisiert werden. Zudem sollen aber auch die Rohdaten verfügbar sein, sodass die Universität und andere Forschungseinrichtungen diese für weitere Anwendungen oder explorative Analysen verwenden können. Aufgrund der sehr schnellen Ausbreitung des Virus und der damit einhergehenden dynamischen Situation sollen die Daten möglichst echtzeitnah aktuell gehalten werden.

Insgesamt verfügt die Universität in dem erdachten Szenario über drei Datenquellen.

Datenquelle 1 ist ein Datensatz, der Einwohnerzahl, Fläche und Bevölkerungsdichte zu jedem Staat und Territorium enthält. Dieser Datensatz ist eher statisch, die Daten werden in der Regel einmal im Monat aktualisiert. Die Universität erhält dann eine Aktualisierung in Form einer CSV-Datei.

Datenquelle 2 ist die zentrale Datenquelle für das Szenario. Über eine von der Universität bereitgestellte Schnittstelle übermitteln die verschiedenen Gesundheitsämter auf der Welt einmal täglich die Anzahl der Infizierten in ihrem Staat oder Territorium. Die Daten werden

[1] https://coronavirus.jhu.edu/map.html

ebenfalls im CSV-Format übermittelt. Ein empfangener Datensatz enthält dabei den Namen des Territoriums, Standortdaten sowie Anzahlen der infizierten, gegenwärtig erkrankten, gesundeten und verstorbenen Personen. Dabei müssen kommunale und regionale Fallzahlen aggregiert werden, um die Gesamtzahl der Infizierten für alle Regionen und Staaten zu erhalten. Bei ihrer Verarbeitung soll eine Plausibilitätsprüfung durchgeführt werden, um mögliche Fehler – wie zum Beispiel negative Infektions- oder Todeszahlen, die zu fehlerhaften Aggregationen führen würden – auszuschließen.

Um die Funktion der Karte zu erweitern, wird **Datenquelle 3** hinzugezogen. Über eine Wetter-API kann die Universität mithilfe der Standortdaten eines Territoriums aktuelle Wetterdaten beziehen. Sofern die API zuverlässige Wetterdaten für den Standort eines Territoriums liefern kann, werden diese einmal täglich abgefragt. Der Datensatz wird im JSON-Format übermittelt und enthält verschiedene Wetterdaten wie unter anderem die aktuelle Temperatur und Tagesdurchschnittstemperatur in Celsius, Luftfeuchtigkeit und Windgeschwindigkeit. Für die weitere Verarbeitung der Daten spielt im ausgedachten Szenario zunächst die durchschnittliche Tagestemperatur eine Rolle.

Auf Grundlage der Daten aus den Datenquellen 2 und 3 sollen Data Scientists verschiedene explorative Analysen durchführen. Eine solche explorative Analyse könnte beispielsweise untersuchen, ob die Entwicklung der Neuinfizierten-Anzahlen mit der Tagesdurchschnittstemperatur am jeweiligen Standort korreliert. Um Diagnose- und Meldeverzögerungen einzubeziehen, wird für jedes Territorium die durchschnittliche Tagestemperatur von vor einer Woche mit der Zu- oder Abnahme der Anzahl der Neuinfizierten pro Tag (zweite Ableitung der Anzahl der Infizierten eines Territoriums) verglichen. Eine Woche entspricht hierbei der mittleren Inkubationszeit des Virus, also der Zeit zwischen Ansteckung und dem Auftreten erster Symptome [Rob2020].

Für die Analysen stellt die Data-Lake-Architektur Schnittstellen bereit, an denen die benötigten Rohdaten bezogen werden können. Auf Grundlage der Rohdaten führen Datenanalysten explorative Analysen durch. Wird dabei ein gehaltvolles Ergebnis erzielt, können die gewonnenen Daten über eine weitere Schnittstelle in die entsprechende Zone des Data Lake zurückgespeist werden. Die Analyse selbst findet aber außerhalb der Data-Lake-Architektur statt.

Die gesammelten rohen und transformierten Daten dienen als Grundlage für eine neue Web-Applikation, die die Universität bereitstellt. Tabelle 1 zeigt die Anforderungen an das Dashboard der Applikation und die Schritte, die notwendig sind, um die jeweiligen Daten zu erhalten.

Tabelle 1: Überblick über die Anforderungen an die Data-Lake-Plattform

#	Anforderung	Aufgaben
1	Die Anzahl der Infizierten für jedes Territorium.	Prüfung der Daten aus Datenquelle 2
2	Die Anzahl der Infizierten für jeden Staat, also beispielsweise auch die Gesamtzahl der Infizierten in den USA oder in China.	Aggregation der geprüften Daten aus Datenquelle 2

<table>
<tr><td>3</td><td>Die Anzahl der Infizierten pro 100.000 Einwohner für jeden Staat.</td><td rowspan="3">Kombination der geprüften, aggregierten Daten aus Datenquelle 2 mit Daten aus Datenquelle 1</td></tr>
<tr><td>4</td><td>Die Anzahl der Infizierten pro km² für jeden Staat.</td></tr>
<tr><td>5</td><td>Ein Diagramm, welches die Bevölkerungsdichte der Staaten mit den Infizierten pro 100.000 Einwohnern in Verhältnis setzt.</td></tr>
<tr><td>6</td><td>Ein Diagramm für jedes Territorium, das den zeitlichen Verlauf der Zu- und Abnahme der Neuinfizierten ins Verhältnis mit der durchschnittlichen Tagestemperatur am Standort vor sieben Tagen setzt.</td><td>Explorative Analyse auf Grundlage der geprüften, aggregierten Daten aus Datenquelle 2 und den aggregierten, transformierten Daten aus Datenquelle 3</td></tr>
</table>

5 Spezifikation des Zonenreferenzmodells

In diesem Kapitel wird das Zonenreferenzmodell für die Anwendung im Datenstromkontext spezifiziert. Dazu werden in Kapitel 5.1 zunächst die Datenflüsse in einer generischen Architektur definiert, die auf dem Zonenreferenzmodell aufbaut und Datenströme als Übertragungsmittel nutzt. In Kapitel 5.2 werden die Datenflüsse und die Datenverarbeitungsschritte im konkreten Anwendungsfall beschrieben.

5.1 Allgemeine Spezifikation des Zonenreferenzmodells für Datenströme

In einer zonenbasierten Data-Lake-Architektur im Datenstromkontext kann eine „Zone“ als ein Verarbeitungsschritt für den jeweiligen Datenstrom verstanden werden. Jeder Eingangsdatenstrom einer Zone wird mithilfe eines Service verarbeitet, der die Transformationen auf den Daten ausführt. Dieser Service wird im Folgenden als *Datenprozessor* bezeichnet. Am Ende der Verarbeitung durch den Datenprozessor wird ein Ausgangsdatenstrom erzeugt, der die transformierten Daten enthält.

In Abbildung 6 werden die Datenströme als schwarze Pfeile mit ausgefüllten Pfeilspitzen dargestellt. Die Anzahl der Eingangs- und Ausgangsdatenströme sowie der Datenprozessoren kann von Zone zu Zone variieren, zur Veranschaulichung werden jeweils drei Prozessoren und drei Datenstrom-Pfeile abgebildet.

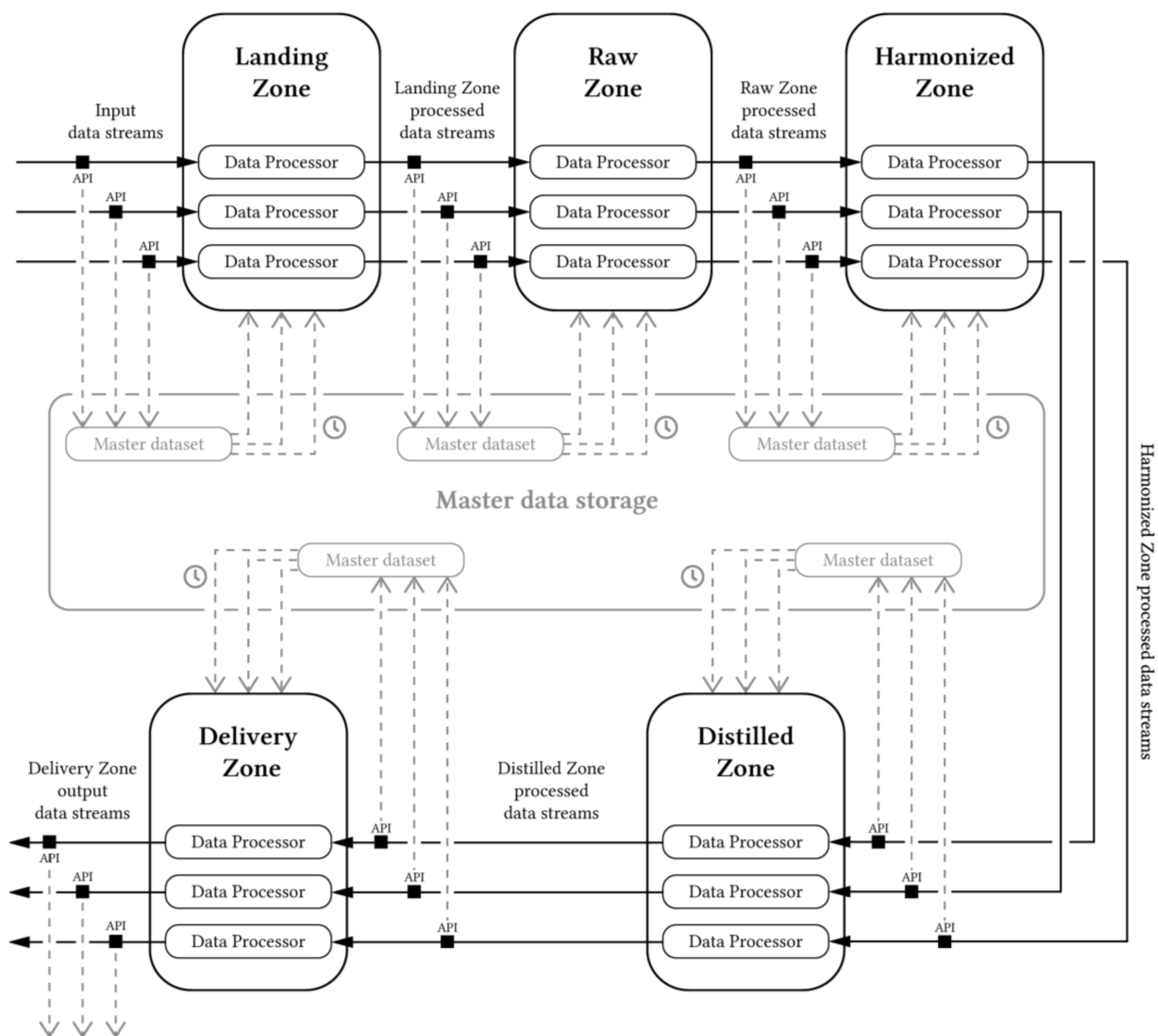


Abbildung 6: Datenflüsse in der Zonenreferenzmodell-basierten Streaming-Architektur

Die Anordnung der Zonen entspricht der des Zonenreferenzmodells, wobei in Abbildung 6 auf die Darstellung der Protected Parts und der Explorative Zone verzichtet wurde. Protected Parts könnten durch zusätzliche Datenprozessoren realisiert werden, die die Daten beispielsweise vor ihrer Verarbeitung anonymisieren oder nach der Verarbeitung wieder verschlüsseln. Die Explorative Zone benötigt keine eigenen Datenprozessoren, sondern nutzt die Abfrage-APIs der anderen Zonen für den Datenimport, um Daten aus mehreren Zonen und Quellen zu kombinieren und explorative Analysen durchführen zu können.

Das Zonenreferenzmodell für Datenströme kann als reine Streaming-Architektur oder mit zusätzlichem persistentem Speicher – dem *Master Dataset* – realisiert werden. Für die Spezifikation des Zonenreferenzmodells für Datenströme wird keine Persistierung benötigt. Aus diesem Grund ist der persistierende Teil der Architektur in Abbildung 6 in einem Grauton gehalten und Datenflüsse in und aus dem Master Dataset mit gestrichelten Pfeilen dargestellt. In realistischen Anwendungsszenarien in der Praxis ist eine Persistierung allerdings fast immer erforderlich.

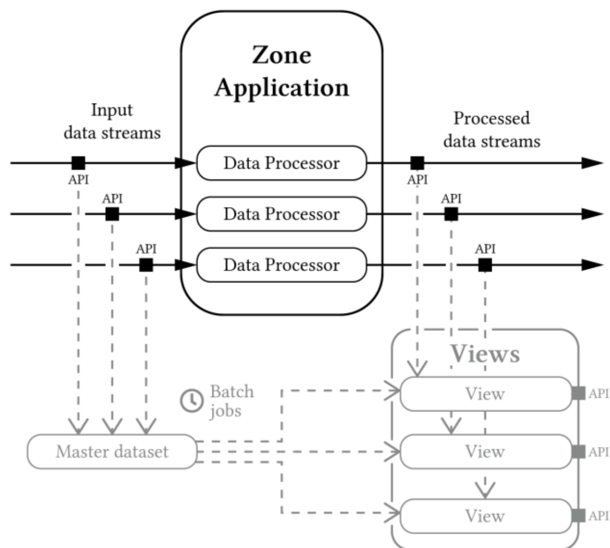
Werden nur die Streaming-Komponenten der Architektur betrachtet (schwarz gezeichnete Komponenten in Abbildung 6), liegen sämtliche Daten während ihrer Verarbeitung zu jedem Zeitpunkt ausschließlich als Datenstrom vor. Dies ist ein zentraler Unterschied gegenüber einer Batch-Architektur eines Data Lake, in der die Daten nach jedem Verarbeitungsschritt in der jeweiligen Zone persistiert werden.

Da die Daten in der reinen Streaming-Architektur nirgendwo persistiert werden, müssen Schnittstellen für alle Datenströme existieren, mithilfe derer die darin enthaltenen Daten abgegriffen werden können. Diese Schnittstellen sind in Abbildung 6 in Form von schwarzen Quadraten dargestellt. In der Abbildung verfügen sie lediglich über Pfeil-Verbindungen zum optionalen Master Dataset. Die zentrale Funktion der Schnittstellen ist aber, die Daten der jeweiligen Zone für Nutzer sowie für externe Systemen oder Services innerhalb der Data-Lake-Plattform verfügbar zu machen, damit die Daten abgegriffen und weiter genutzt werden können. Da die Schnittstellen auf unterschiedlichste Weise von verschiedenen Nutzern und Applikationen verwendet werden, wurde auf die Darstellung weiterer ausgehender Pfeile verzichtet.

Beim Versand und der Abfrage der Daten bietet sich ein einheitliches Übertragungsformat für strukturierte und semi-strukturierte Daten an (z. B. JSON), da auf diese Weise einfacher Kombinationen von Daten aus mehreren Quellen durchgeführt werden können.

Soll eine Lambda- oder Kappa-Architektur umgesetzt werden, sind Master Datasets zur Persistierung zwingend erforderlich. Dies kann verschiedene Arten der Datenverarbeitung in einer Streaming-Architektur vereinfachen: Werden die Daten zusätzlich persistiert, können sie später erneut verarbeitet werden, um Fehler in den Daten zu korrigieren, Datenverluste zu vermeiden, Aggregationen durchzuführen oder Daten aus mehreren Datenstromquellen und Zonen zu kombinieren. Die für die Lambda- und Kappa-Architektur notwendige erneute Verarbeitung der Daten aus dem Master Dataset wurde in Abbildung 6 mit gestrichelten Pfeilen und einem Uhr-Symbol dargestellt. In einer Lambda-Architektur erfolgt die erneute Verarbeitung zeitgesteuert, in einer Kappa-Architektur wird sie durch eine Veränderung der Programmlogik der Datenprozessoren getriggert (siehe Kapitel 2.3).

Abbildung 7 zeigt den Aufbau einer Zone in einer Lambda- (links) und in einer Kappa-Architektur. Anhand der Abbildung werden im Folgenden die notwendigen zusätzlichen Komponenten und Datenflüsse innerhalb einer Zone verglichen.



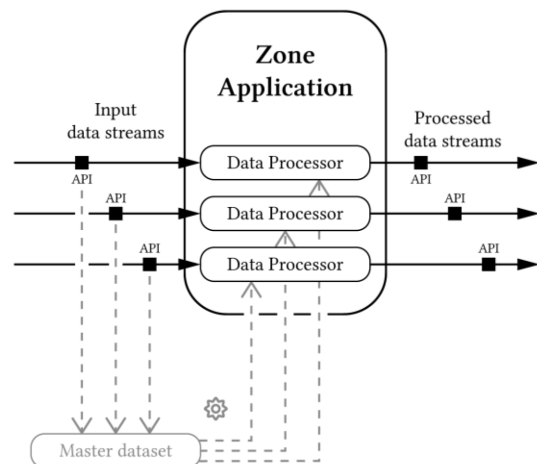


Abbildung 7: Lambda- und Kappa-Architektur mit Datenströmen

In einer **Lambda-Architektur** existieren View-Komponenten, welche Schnittstellen zur Datenabfrage für die Nutzer bereitstellen. Die eingehenden Datenströme einer Zone werden einerseits im Master Dataset persistiert, andererseits in Echtzeit durch die Datenprozessoren verarbeitet. Die View-Komponenten haben die Funktion der in Kapitel 2.3.1 vorgestellten *Serving*-Ebene, das Master Dataset mit der zugehörigen Verarbeitungslogik entspricht der *Batch*-Ebene und die Datenprozessoren stellen die *Speed*-Ebene dar [vgl. Mar2015]. Haben die Daten ihren jeweiligen Datenprozessor durchlaufen, können sie sofort mithilfe der Views durch die Nutzer abgefragt werden.

Die Daten aus dem Master Dataset werden hingegen in einem festgelegten Intervall erneut von *Batch Jobs* verarbeitet. Dabei kann die Verarbeitung in den Batch Jobs sich von der Verarbeitung durch die Datenprozessoren unterscheiden. Die aus den Batch Jobs resultierenden Daten werden nun dazu genutzt, die Daten in den Views zu aktualisieren, Fehler zu beheben oder Datenlücken zu schließen [vgl. Ber2018]. Nachdem die Batch Jobs abgeschlossen sind, gelangen die Nutzer über die View-Komponente an die korrigierten Daten. Mithilfe der API der View-Komponente ließen sich die aktualisierten Daten in Streams umwandeln und könnten so an die folgende Zone gesendet werden.

Auch bei einer **Kappa-Architektur** landen die Daten der Eingangsdatenströme zum einen im Master Dataset, zum anderen werden sie durch den zugewiesenen Datenprozessor verarbeitet. Anders als bei der Lambda-Architektur existieren allerdings weder Batch-Job- noch View-Komponenten. Verändert sich die Verarbeitungslogik in den Datenprozessoren, werden die Daten aus dem Master Dataset nochmals an die veränderten Prozessoren gesendet [vgl. Ber2017]. Sie durchlaufen dann erneut die Stream-Verarbeitung in den Zonenapplikationen. Die Veränderung der Verarbeitungslogik und die daraus resultierende erneute Verarbeitung soll in Abbildung 7 durch das Zahnrad-Symbol dargestellt werden. Die Streaming-Daten können nach ihrer Verarbeitung durch die aktualisierten Prozessoren über die Schnittstellen direkt an den Datenströmen ausgelesen werden.

5.2 Spezifikation der Datenverarbeitung für den Anwendungsfall

In diesem Kapitel werden die Datenverarbeitungen in den Zonen anhand der Anforderungen des konkreten Anwendungsfall (siehe Kapitel 4) spezifiziert. Dabei werden die Datenflüsse sowie die Datenstrukturen und -beschaffenheiten nach jedem Verarbeitungsschritt definiert. Die beigefügten Tabellen zeigen beispielhaft die Verarbeitungsschritte der Pandemiedaten anhand eines konkreten Datensatzes. Es handelt sich hierbei um einen fiktiven Datensatz, der angepasst und gekürzt wurde, um ein repräsentatives und übersichtliches Beispiel darstellen zu können. Die Struktur der Rohdaten entspricht der Datenstruktur der von der John-Hopkins-Universität bereitgestellten Pandemiedaten.

Je nach Datenquelle liegen die Rohdaten entweder im CSV- oder im JSON-Format vor. In der **Landing Zone** werden sämtliche Daten ins JSON-Format konvertiert. JSON wird als Datenübertragungsformat gewählt, da es im Gegensatz zu CSV nicht nur Attributwerte speichert, sondern zu jedem Wert auch den zugehörigen Schlüssel. Dadurch können einzelne Werte eines Datensatzes einfacher extrahiert und direkt ausgewertet werden.

Auch die Reihenfolge der Attribute spielt im JSON-Format keine Rolle, da die Werte als Schlüssel-Wert-Paare vorliegen. Da die Rohdaten aus Datenquelle 1 und 2 im CSV-Format eintreffen, wobei nicht zwischen String- und Zahlentypen unterschieden wird, werden zunächst alle Werte des Objekts zu JSON-Attributen vom Typ String konvertiert, um Datenverluste durch fehlerhafte Datentypkonvertierungen in Landing und Raw Zone zu vermeiden.

Tabelle 2 zeigt die Struktur und Eigenschaften der Rohdaten (linke Spalte) und der Daten, die bereits die vollständige Verarbeitung in der Landing Zone durchlaufen haben (rechte Spalte).

Tabelle 2: Datenverarbeitung in der Landing Zone

Verarbeitung	Roh- / Quelldaten	Landing Zone
Datenformat	CSV oder JSON	JSON
Datentypen	Strings bei CSV, verschiedene Datentypen bei JSON	Datentypen wie aus Quelle (JSON) oder Konvertierung in Strings (CSV)
Schlüsselnamen	durch Quelle definiert	wie aus Quelle
Verschachtelte Strukturen	bei JSON möglich	werden beibehalten
Datumsformat	durch Quelle definiert	wie aus Quelle
Aggregationen/ Kombinationen	Rohdaten	Rohdaten

Beispiel	FIPS,Admin2,Province_State,
	Country_Region,Last_Update,
	Lat,Long_,Confirmed,Deaths,
	Recovered,Active,Combined_Key
	...
	45001,Abbeville,Florida,US,
	2020-05-19 02:32:18,34.22333378,
	-82.4617065,35,0,0,35,"Abbeville,
	Florida, US"
	...
		{
		"FIPS": "45001",
		"Admin2": "Abbeville",
		"Province_State": "Florida",
		"Country_Region": "US",
		"Last_Update": "2020-05-19
		02:32:18",
		"Lat": "34.22333378",
		"Long_": "-82.46170658",
		"Confirmed": "35",
		"Deaths": "0",
		"Recovered": "0",
		"Active": "35",
		"Combined_Key": "Abbeville,
		Florida, US",
		"Arrived_In_Landing_Zone_At":
		"2020-05-20 12:32:18.003",
		"Processed_In_Landing_Zone_At":
		"2020-05-20 12:32:18.133"
		}

In der **Raw Zone** werden alle Attributschlüssel umbenannt, sodass sie einer festgelegten und einheitlichen Konvention zu entsprechen. Es wird eine Formatierung der Attributschlüssel im „Upper Snake Case" festgelegt: Mehrere Wörter im Schlüsselnamen werden durch einen Unterstrich getrennt, wobei neue Wörter mit einem Großbuchstaben beginnen. Abkürzungen werden vermieden, die Wörter in Attributschlüsselnamen grundsätzlich ausgeschrieben.

Aus dem Attributschlüssel `pop2020` aus Datenquelle 1 wurde so beispielsweise der Schlüsselname `Population_2020`. Diese Konvention sorgt für eine bessere Lesbarkeit der JSON-Objekte, auch wenn diese unformatiert vorliegen. Im Kontext von Streaming-Architekturen kann es häufig vorkommen, dass Datenstromobjekte auf der Konsole ausgegeben werden müssen – zum Beispiel zu Debugging-Zwecken oder, wenn ein Tool über keine grafische Oberfläche verfügt. Deshalb sind eine gute Lesbarkeit und eindeutige Zuordenbarkeit der Attribute in einem Datensatz hier besonders wichtig.

Die Wetterdaten aus Datenquelle 3 liegen bereits initial im JSON-Format vor. In der Raw Zone wird – wenn möglich – eine Verflachung verschachtelter JSON-Objekte auf eine Ebene vorgenommen, um eine spätere Speicherung des Datensatzes in einer einzigen flachen Tabelle zu ermöglichen.

In der **Harmonized Zone** werden die Attributwerte selbst in ihrer Struktur vereinheitlicht. So erfolgt zunächst eine Konvertierung der Zahlen-Attributwerte von Strings zum jeweiligen Nummerntyp (*Integer*, *Float*, *Double*, *Long*, ...). Außerdem werden alle Datumsangaben und Zeitstempel ins ISO-8601-Format transformiert.

Die Bevölkerungszahlen in den Länderdaten (`Population_2019`, `Population_2020`) liegen bislang in der Einheit „1.000 Einwohner" vor. Um Missinterpretationen der Daten zu vermeiden, werden diese Zahlen mit 1.000 multipliziert, sodass nach der Verarbeitung die tatsächlichen absoluten Bevölkerungszahlen vorliegen.

Tabelle 3 zeigt die Datenstrukturen und -eigenschaften der Daten nach ihrer Verarbeitung durch die Raw Zone (linke Spalte) sowie durch die Harmonized Zone (rechte Spalte).

Tabelle 3: Datenverarbeitungen in Raw und Harmonized Zone

Verarbeitung	Raw Zone	Harmonized Zone
Datenformat	JSON	JSON
Datentypen	i.d.R. *String*	typisiert (*String*, *Integer*, *Float*, *Double*, *Date*, *Timestamp*, ...)
Schlüsselnamen	einheitliche Benamung nach festgelegter Konvention	einheitliche Benamung
Verschachtelte Strukturen	Verflachung verschachtelter Objekte zur besseren Datenbankspeicherung	flache Struktur
Datumsformat	wie aus Quelle	standardisiert (ISO-8601)
Aggregationen/ Kombinationen	nicht-aggregierte Daten aus einzelner Quelle	nicht-aggregierte Daten aus einzelner Quelle

Beispiel	{
	"FIPS_US_County_Code": "45001",
	"County": "Abbeville",
	"Region": "Florida",
	"Country": "US",
	"Updated_At": "2020-05-19 02:32:18",
	"Latitude": "34.22333378",
	"Longitude": "-82.46170658",
	"Confirmed_Cases": "35",
	"Death_Cases": "0",
	"Recovered_Cases": "0",
	"Active_Cases": "35",
	"Combined_Key": "Abbeville, Florida, US",
	...
	"Arrived_In_Raw_Zone_At": "2020-05-20 12:32:18.201",
	"Processed_In_Landing_Zone_At": "2020-05-20 12:32:18.417"
	}
		{
		"FIPS_US_County_Code": 45001,
		"County": "Abbeville",
		"Region": "Florida",
		"Country": "US",
		"Updated_At": "2020-05-19T02:32:18.000Z",
		"Latitude": 34.22333378,
		"Longitude": -82.46170658,
		"Confirmed_Cases": 35,
		"Death_Cases": 0,
		"Recovered_Cases": 0,
		"Active_Cases": 35,
		"Combined_Key": "Abbeville, Florida, US",
		...
		"Arrived_In_Harmonized_Zone_At": "2020-05-20T12:32:18.499Z",
		"Processed_In_Harmonized_Zone_At": "2020-05-20T12:32:18.709Z"
		}

In der **Distilled Zone** werden die Pandemiedaten aggregiert. Aus den Einträgen aller Gemeinden einer Region sowie aller Regionen eines Staates wird ein aggregierter Eintrag erzeugt. So liegt für alle Staaten und Regionen mit bestätigten Coronavirus-Fällen ein Eintrag mit kumulierten Fall- und Todeszahlen vor.

Jeder (aggregierte und nicht-aggregierte) Pandemie-Datensatz erhält ein neues Attribut `Administrative_Level`, das beschreibt, ob es sich beim jeweiligen Eintrag um Daten einer Gemeinde (`"County"`), Region (`"Region"`) oder eines Staates (`"Country"`) handelt.

In der **Delivery Zone** werden aus den Datensätzen aller Datenquellen kombinierte Datensätze gebildet, sodass zueinander gehörende Daten bequem und mit nur wenigen Abfragen für die Endnutzer ausgegeben werden können – beispielsweise auf dem Dashboard einer Webanwendung.

Ebenfalls in der Delivery Zone werden den Pandemie-Datensätzen mithilfe ihrer Standort-Attribute passende Wetterdaten zugeordnet. Dabei erhalten die Pandemie-Einträge zwei neue Attribute `Temperature_Current` und `Temperature_Week_Ago`.

Einträge mit dem Aggregationslevel `"Country"` werden mit den passenden Länderdaten aus Datenquelle 1 verknüpft. Sie erhalten die zusätzlichen Attribute `Area`, `Population` und `Density`. Mithilfe dieser Attribute werden für die länderübergreifenden Datensätze außerdem die Attributwerte `Cases_Per_100000_Inhabitants` (Fälle pro 100.000 Einwohner) und `Cases_Per_Square_Kilometers` (Fälle pro Quadratkilometer) berechnet. Zuletzt werden sämtliche Pandemieeinträge mit den vorausgegangenen Daten abgeglichen. Hieraus wird der Anstieg der Infektionsfälle im Vergleich zum Vortag (`Confirmed_Cases_Daily_Increase`) sowie der durchschnittliche Anstieg der Fälle pro Tag in der vergangenen Woche (`Confirmed_Cases_Weekly_Average_Increase`) abgeleitet.

Zum Zweck der in Kapitel 7 vorgenommenen Evaluation der Echtzeitfähigkeit der Data-Lake-Architektur erhalten die Datensätze **in jeder Zone zusätzliche Attribute mit Zeitstempeln**. Diese Attribute enthalten den Zeitpunkt des Eintreffens der Daten in der jeweiligen Zone (z. B. `Arrived_In_Landing_Zone_At`) sowie den Zeitpunkt des Abschlusses der Datenverarbeitung (z. B. `Processed_In_Landing_Zone_At`).

Tabelle 4 zeigt die Dateneigenschaften sowie den Beispieldatensatz nach der Verarbeitung in der Distilled Zone (linke Spalte) und nach der abschließenden Verarbeitung in der Delivery Zone (rechte Spalte).

Da im Anwendungsbeispiel keine sicherheits- oder datenschutzrelevanten, sondern ausschließlich öffentlich einsehbare Daten genutzt werden, ist die Implementierung von **Protected Parts** in keiner Zone notwendig.

Tabelle 4: Datenverarbeitungen in Distilled und Delivery Zone

Verarbeitung	Distilled Zone	Delivery Zone
Datenformat	JSON	JSON
Datentypen	typisiert	typisiert
Schlüsselnamen	einheitliche Benamung	einheitliche Benamung
Verschachtelte Strukturen	flache Struktur	flache Struktur
Datumsformat	standardisiert (ISO-8601)	standardisiert (ISO-8601)
Aggregationen/ Kombinationen	aggregierte und nicht-aggregierte Daten	aggregierte und kombinierte Daten aus verschiedenen Quellen

	(aggregierter Datensatz)	(aggregierter, kombinierter Datensatz)

Beispiel	{
	"FIPS_US_County_Code": null,
	"County": "",
	"Region": "",
	"Country": "US",
	"Administrative_Level":
	"Country",
	...
	"Confirmed_Cases": 5961884,
	"Death_Cases": 182798,
	"Recovered_Cases": 3408908,
	"Active_Cases": 2370178,
	...
	"Arrived_In_Distilled_Zone_At":
	"2020-08-30T12:32:18.781Z",
	"Processed_In_Distilled_Zone_At":
	"2020-08-30T12:32:19.149Z"
	}
		{
		"FIPS_US_County_Code": null,
		"County": "",
		"Region": "",
		"Country": "US",
		"Administrative_Level":
		"Country",
		...
		"Confirmed_Cases": 5961884,
		...
		"Population": 328239523,
		"Area": 9833520,
		"Density": 33.6871,
		"Cases_Per_100000_Inhabitants":
		1816.32118689,
		"Cases_Per_Square_Kilometers":
		0.60628177981,
		...
		"Temperature_Current": 17.3,
		"Temperature_7_Days_Ago": 21.8,
		...
		"Arrived_In_Delivery_Zone_At":
		"2020-08-30T12:32:18.781Z",
		"Processed_In_Delivery_Zone_At":
		"2020-08-30T12:32:19.149Z"
		}

6 Beschreibung der prototypischen Implementierung

In diesem Kapitel wird die entwickelte prototypische Data-Lake-Architektur für Datenströme erläutert. Dabei werden die angewandten Architekturpatterns (Kapitel 6.1), die einzelnen Komponenten der Architektur (Kapitel 6.2) sowie eingesetzte Technologien (Kapitel 6.3) beschrieben. In Kapitel 6.4 werden Schlüsselkomponenten und -funktionen anhand von Implementierungsbeispielen verdeutlicht.

6.1 Beschreibung der Architekturpatterns

Bei einer Zonenarchitektur bietet es sich an, die Datenverarbeitungen in den einzelnen Zonen als unabhängige **Microservices** zu entwickeln. Da ein Data Lake üblicherweise nicht auf einem einzigen Rechner, sondern in einem Rechnercluster ausgeführt wird, können die Zonen als unabhängige Applikationen auf verschiedenen Knoten im Rechnernetz operieren. Deshalb wurde die Datenverarbeitung in der entwickelten prototypischen Architektur in mehrere Applikationen mit jeweils mehreren Services aufgeteilt.

Handelt es sich um eine komplexe oder stark dynamisch wachsende Data-Lake-Architektur wäre es auch möglich, jeden einzelnen Datenprozessor in eigene Mikroanwendungen zu verpacken. Auf diese Weise würde für jeden Eingangsdatenstrom in jeder Zone eine eigene Anwendung existieren, die die zum Datenstrom passende Datenverarbeitung enthält. Auf diese Weise könnte jeder Prozessor unabhängig von anderen Anwendungen bestimmten Rechnerknoten zugewiesen werden. Da die Datenverarbeitungsschritte im zugrundeliegenden Anwendungsfall pro Datenquelle und Zone wenig komplex sind, wurde auf diese Aufteilung verzichtet. In der entwickelten Architektur existieren also fünf Applikationen – je eine pro Zone –, die über mehrere Datenprozessoren verfügen.

Auch wenn die prototypische Implementierung auf einem einzigen Rechner betrieben wird, wurde bei der Konzeption und Auswahl der Technologien darauf geachtet, dass alle Komponenten innerhalb eines Rechnernetzes operieren können. Die Zuweisung von neuen Clusterknoten kann bei jeder Anwendung durch eine Änderung an der Konfiguration mit wenig Aufwand realisiert werden. Hieraus resultiert eine verteilte und **stark skalierbare Architektur**.

Die Kommunikation zwischen den Anwendungen funktioniert auf Grundlage einer **unidirektionalen nachrichtenbasierten Datenverarbeitung**. Nach Abschluss der Datentransformation in einer Zone werden die resultierenden Daten als Nachricht an einen Datenstromverteiler (*Apache Kafka*) geschickt. Die darauffolgende Zonenapplikation enthält einen Empfänger für die verarbeitete Nachricht aus der Vorgängerzone (*Kafka Consumer*). Sobald der Empfänger eine neue Nachricht registriert, stößt er die Datenverarbeitung in der eigenen Zone an, welche nach Abschluss der Verarbeitung wiederum eine ausgehende Nachricht erzeugt (*Kafka Producer*).

Die Datenverarbeitungen in den jeweiligen Zonen nutzen das „**Pipes and Filter**"-Architekturmuster, wobei mehrere Datenverarbeitungsschritte aneinandergehängt werden. Eine Beschreibung der zugrundeliegenden Technologie (*Apache Spark Structured Streaming*) und deren Funktionsweise folgt in Kapitel 6.3.3.

Da sich die entwickelte Architektur ausschließlich auf die Verarbeitung von Datenströmen fokussiert, handelt es sich **weder um eine typische Lambda- noch um eine Kappa-Architektur**. Beides ließe sich aber durch Erweiterungen einfach umsetzen. Die in einer Zone ankommenden Datenströme werden bereits in einer Datenbank gespeichert.

In der beispielhaften Implementierung werden die gespeicherten Daten dazu genutzt, Aggregationen durchzuführen und kombinierte Datensätze zu erzeugen. Die Datenbank könnte allerdings darüber hinaus als Master Dataset in einer Lambda- oder Kappa-Architektur dienen.

Um eine Lambda-Architektur zu erhalten, wäre ein regelmäßig ausgeführter Batch Job notwendig, der alle im definierten Zeitraum angefallen Daten im Master Dataset erneut verarbeitet. In einer Kappa-Architektur könnten die gesammelten Daten bei einer Veränderung der Programmlogik erneut als Streams verarbeitet werden.

6.2 Beschreibung der Komponenten

In diesem Kapitel werden die einzelnen Komponenten der Architektur sowie deren Aufgaben und innere Struktur beschrieben. Die Architektur setzt sich im Wesentlichen aus den fünf Zonenapplikationen, einem „Data Streams Generator“, dem Datenstromverteiler und einer Datenbank zusammen.

Abbildung 8 folgt umseitig und zeigt eine Übersicht der gesamten entwickelten Data-Lake-Architektur. Darin sind alle Komponenten und Datenflüsse zwischen den Komponenten abgebildet.

Bei Pfeilen mit dünner ausgefüllter Spitze handelt es sich um Datenströme, die vom Datenstromverteiler Apache Kafka in Topics organisiert werden. Bei den gestrichelten Pfeilen mit nicht-ausgefüllter Spitze handelt es sich jeweils um Datenbankmutationen, die Tabellen in *Apache Cassandra* erstellen oder ändern. Durchgezogene Pfeile mit breiten ausgefüllten Pfeilspitzen stellen die Datenbankabfragen dar. Die anderen Pfeile mit nicht-ausgefüllter Pfeilspitze stehen für sonstige Datenströme, wie etwa HTTP-Requests und -Responses, Git-Pull-Operationen sowie interne Datenflüsse zwischen mehreren Packages und Komponenten.

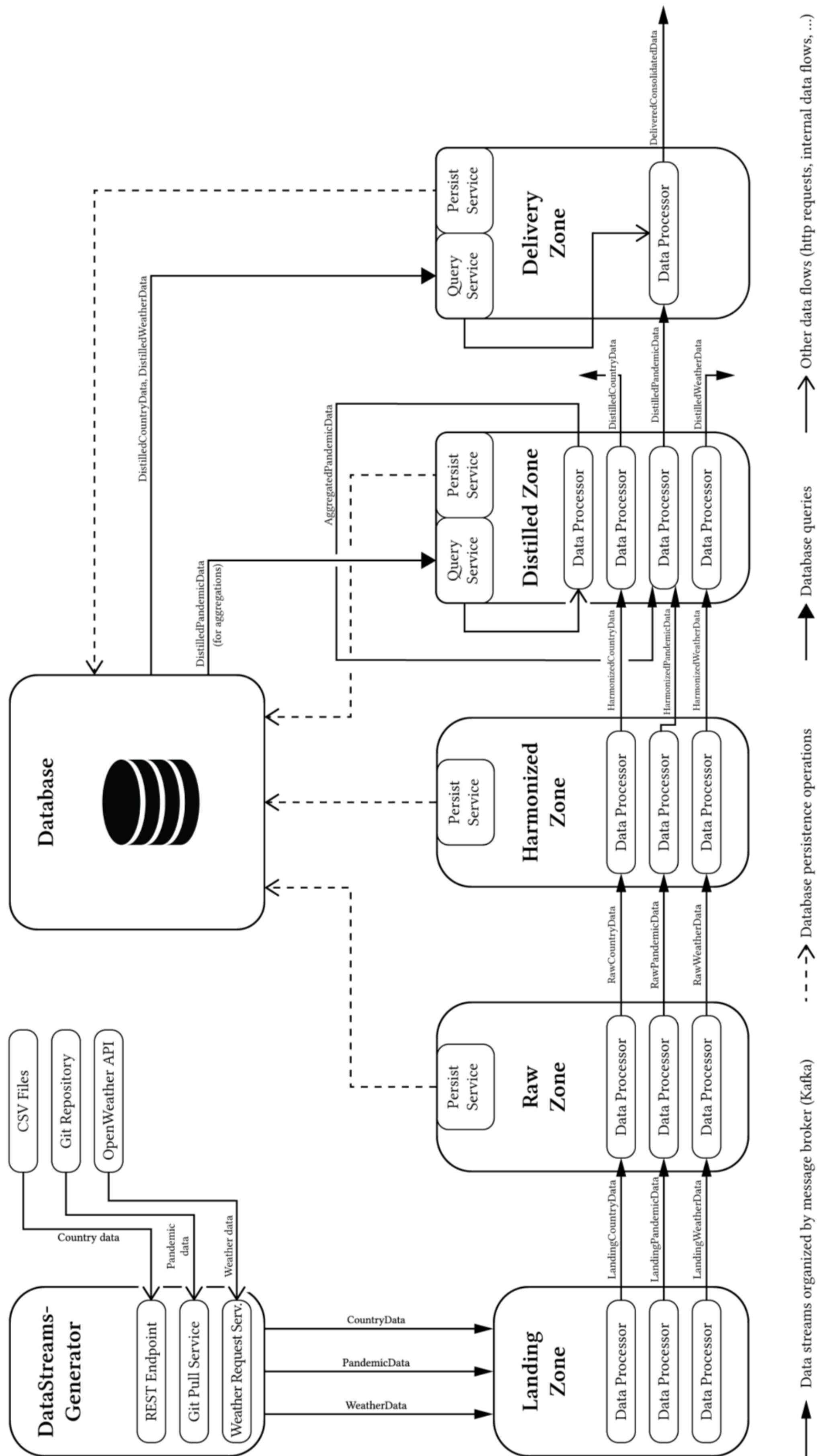


Abbildung 8: Architektur- und Datenflussschaubild der entwickelten Architektur

6.2.1 Data Streams Generator

Der „Data Streams Generator“ wurde entwickelt, um kontinuierliche Datenströme zu erzeugen, welche anschließend vom Data Lake weiterverarbeitet werden können. Dazu ruft der Generator reale Daten aus mehreren Quellen ab und wandelt sie in Datenströme um.

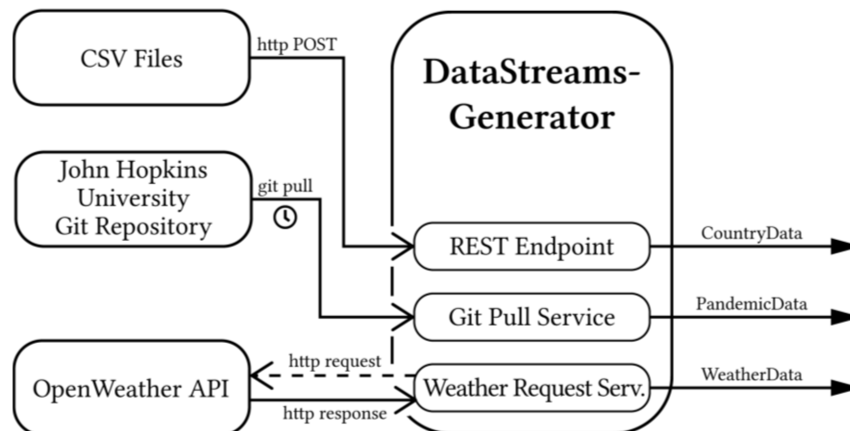


Abbildung 9: Datenflüsse im *DataStreamsGenerator*

Die Applikation enthält zunächst eine REST-Schnittstelle, an die mittels HTTP-POST-Request CSV-Dateien übermittelt werden können. Mithilfe einer solchen Datei kann die Länderdaten-Tabelle aktualisiert werden, die zu jedem Land Fläche, Bevölkerung und Bevölkerungsdichte enthält (Datenquelle 1). Sendet ein Nutzer eine CSV-Datei über die Schnittstelle an die Applikation, teilt diese den Datensatz in einzelne Zeilen auf und sendet die Einträge nacheinander als Datenstrom an Apache Kafka, welches als Verteiler für die Datenströme fungiert.

Des Weiteren fragt die Applikation in regelmäßigen Abständen Daten aus dem Git-Repository der John-Hopkins-Universität[2] ab. Das Repository ist öffentlich zugänglich und enthält die Corona-Infektions- und Todeszahlen sowie weitere Kennzahlen zur Pandemie pro Territorium (Datenquelle 2). Der gesamte Datensatz wird durch den Data Streams Generator in einzelne Datenzeilen zerlegt. Die Daten werden dann Zeile für Zeile in einem konfigurierbaren Intervall – standardmäßig alle drei Sekunden – als kontinuierlicher Datenstrom an den Verteiler gesendet.

Damit die Applikation bei einem Neustart nicht immer mit der ersten Zeile beginnt, wird der Index der zuletzt bearbeiteten Zeile in einer Datei gespeichert. Der Index wird bei einem Neustart ausgelesen und als Startzeile verwendet. Hat die Applikation alle Zeilen als Datenstrom versendet, wird *git pull* ausgeführt und erneut bei Zeile 1 begonnen.

Neben der Umwandlung in einen Datenstrom wird aus dem Datensatz außerdem eine Liste der Geo-Koordinaten (Längen- und Breitengrad) und Ortsbezeichnungen aller Territorien extrahiert. Diese kommt bei der Abfrage der Wetterdaten zum Einsatz (Datenquelle 3). Die Wetterdaten-Schnittstelle *OpenWeatherMap*[3] erlaubt mit einem kostenlosen Basis-Account bis zu 60 Wetterdaten-Abfragen pro Minute. Eine Anfrage an die Schnittstelle enthält die Längen- und Breitengrade eines Standortes. Als Antwort werden diverse Wetterdaten wie Temperatur, Wind, Niederschlag und Luftdruck für den angefragten Standort im JSON-Format zurückgegeben.

[2] https://github.com/CSSEGISandData/COVID-19
[3] https://openweathermap.org/api

Genau wie bei den Pandemiedaten arbeitet der Data Streams Generator die Liste aller Koordinaten ab und speichert dabei den Index der gegenwärtig bearbeiteten Zeile. Für jeden Standort wird in einem festen Intervall eine Anfrage mit den entsprechenden Koordinaten an die Wetterdaten-Schnittstelle gestellt. Enthält die HTTP-Response den Statuscode 200 (Erfolg), wird die Antwort unverändert als JSON-String an den Datenstromverteiler gesendet.

6.2.2 Zonenapplikationen

Die Datenverarbeitung in jeder Zone wurde in einer eigenen Anwendung umgesetzt – es existiert also je eine Applikation für Landing Zone, Raw Zone, Harmonized Zone, Distilled Zone und Delivery Zone. Die Explorative Zone benötigt keine eigene Anwendung, da dort lediglich Daten aus anderen Zonen importiert und in Analysen ausgewertet werden. Diese Analysen finden in separaten Tools und Anwendungen statt und sind explizit kein Teil der entwickelten Data-Lake-Architektur.

Jede Zonenanwendung ist aufgeteilt in mehrere *Packages*. Alle Zonen verfügen über ein *dataProcessors*-Package, ein *streamWriter*-Package sowie ein Package für Hilfsmethoden (*utils*). Abbildung 10 zeigt schematisch den Aufbau und die Aufgaben der Packages im Kontext des Datenflusses. Neben den genannten obligatorischen Packages verfügt die Zone im Schaubild über je ein Package zur Datenbankspeicherung und Datenbankabfrage.

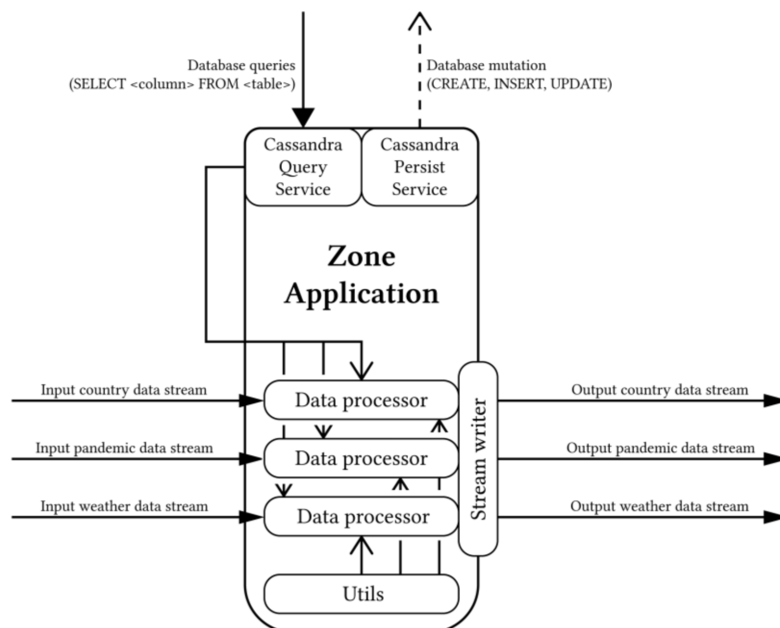


Abbildung 10: Datenfluss und Abhängigkeiten einer Zonen-Applikation

Als Kommunikator zwischen den Zonen sowie als Datenstromempfänger und -sender kommt Apache Kafka zum Einsatz. Die Funktionsweisen und Eigenschaften von Kafka werden in Kapitel 6.3.2 beschrieben.

Die zentrale Datenverarbeitungs-Logik befindet sich im dataProcessors-Package. Apache Kafka organisiert die Datenströme in verschiedenen *Topics*. Jeder Datenprozessor im dataProcessors-Package reagiert auf den Eingangsdatenstrom der ihm zugewiesenen Topic. Registriert der Prozessor einen neuen Datensatz, führt er verschiedene Transformationen auf den Daten aus – beispielsweise Schlüsselumbenennungen, Datentypkonvertierungen, Berechnungen auf mehreren Feldern oder auch Datenbankabfragen für Aggregationen und Kombinationen von Daten aus mehreren Quellen.

In den meisten Zonen existiert je ein Prozessor für die Länderdaten (Datenquelle 1), Pandemiedaten (Datenquelle 2) und Wetterdaten (Datenquelle 3). In der Distilled Zone ist zusätzlich ein Datenprozessor für die aggregierten Pandemiedaten vorhanden. In der Delivery Zone der prototypischen Architektur existiert nur noch ein einziger Datenprozessor, der den Pandemiedatenstrom aus Datenquelle 2 mithilfe von Datenbankabfragen mit passenden Länder- und Wetterdaten (Datenquelle 1 und 3) ergänzt und einen Ausgangsdatenstrom erzeugt. Dieser Datenstrom enthält die kombinierten Daten, die für die im Anwendungsfall geforderte Visualisierung auf dem Web-Dashboard aufbereitet sind.

Aufgrund der gewählten Technologie (Spark Structured Streaming*)* ist der Empfang des jeweiligen Eingangsdatenstroms vom Verteiler (Kafka Consumer) mit im dataProcessors-Package integriert. Ein Implementierungsbeispiel eines solchen Datenprozessors folgt in Kapitel 6.4. In jeder Zonenanwendung existiert außerdem das Package streamWriter, welches dafür zuständig ist, die prozessierten Daten zurück an den Verteiler zu senden (Kafka Producer).

Alle Zonen verfügen über ein utils-Package, welches diverse meist statische Hilfsmethoden zur Dateitypkonvertierung, Datumsformatierung, Kommunikation mit der Datenbank und Umwandlung von JSON-Strings zu strukturierten Objekten enthält.

Für die Persistierung der Daten in der Datenbank kommt das verteilte Datenbankmanagementsystem *Apache Cassandra* zum Einsatz. Die technischen Eigenschaften von Apache Cassandra werden in Kapitel 6.3.5 beschrieben.

In der Raw Zone sowie in allen folgenden Zonen werden die transformierten Daten mithilfe von Services in den Zonenapplikationen persistiert. Für die prototypische Architektur ist nur die Persistierung in Distilled und Delivery Zone relevant, da dort auf die Datenbank zugegriffen werden muss, um Aggregationen und Kombinationen durchzuführen. Dennoch werden die Daten aber in der Raw Zone persistiert, um die spätere Erweiterung der bestehenden Architektur zu einer Lambda- oder Kappa-Architektur zu ermöglichen. Die Raw Zone und alle weiter innen angesiedelten Zonen enthalten deshalb ein Package namens *cassandraPersist*. Je nach Zone werden die Daten pro Quelle mithilfe eines Services in eine einzige Rohdatentabelle oder mehrere tagesbasierte Tabellen geschrieben.

Somit werden die verarbeiteten Daten ab der Raw Zone also auf zwei Arten weiterverwertet: Zum einen werden sie in einer neuen Topic zurück an den Datenstromverteiler gesendet, zum anderen aber auch in einer Datenbank persistiert. Die so persistierten Daten könnten auch als Master Dataset genutzt werden, um den in Kapitel 6.1 erwähnten Batch-Layer umzusetzen und eine Lambda-Architektur zu erhalten.

In der Distilled Zone müssen die Pandemiedaten unterschiedlicher administrativer Ebenen (Land, Region, Gemeinde) aggregiert werden. Diese Aggregationen können nicht zum Zeitpunkt des Eintreffens eines Pandemie-Datensatzes durchgeführt werden, da gleichzeitig auf alle Pandemieeinträge einer Region oder eines Landes zugegriffen werden muss.

Dies geschieht mithilfe der persistierten Daten im *cassandraQuery*-Package. Das Package enthält sowohl die Datenbankabfragen zur Realisierung der Aggregation als auch Methoden zur Umwandlung eines Apache Cassandra „Row“-Objekts zum Spark-internen Typ *DataFrame*. Für die Kombination von Länder-, Pandemie- und Wetterdaten benötigt auch die DeliveryZone das cassandraQuery-Package.

In der Hauptklasse der Zonenanwendungen werden der *Spark Context* und anschließend alle Datenprozessoren initialisiert. In der Distilled Zone wird zudem ein Timer gestartet, der in einem konfigurierbaren Zeitintervall die Aggregationen der Pandemiedaten durchführt.

6.3 Beschreibung der eingesetzten Technologien

In diesem Kapitel werden die Technologien beschrieben, die in den Komponenten zum Einsatz kommen.

6.3.1 *Spring Boot* Java-Applikation

Beim Data Streams Generator handelt es sich um eine Java-Applikation, die das *Spring-Boot-*Framework nutzt. Dieses eignet sich sehr gut zur Bereitstellung schlanker Applikationen innerhalb von Microservice-Architekturen. Zudem beinhaltet das Framework eine einfache Implementierungsmöglichkeit für REST-Schnittstellen, welche im Data Streams Generator für den Empfang der CSV-Länderdaten benötigt wird (siehe Kapitel 6.2.1).

6.3.2 Apache Kafka

Apache Kafka ist der am weitesten verbreitete *Message Broker* im Datenstromkontext und wurde daher in der entwickelten Architektur als Datenstromverteiler genutzt. Die Open-Source-Software kommt in verschiedensten Kontexten beim Versand und Empfang sowie bei der Organisation mehrerer Datenstromquellen zum Einsatz. In der prototypischen Architektur dient Kafka als Kommunikator zwischen dem Data Streams Generator und der Landing Zone sowie zwischen den Zonenapplikationen untereinander. Die folgende Beschreibung basiert auf der Dokumentation der Apache Software Foundation [Apa2020b].

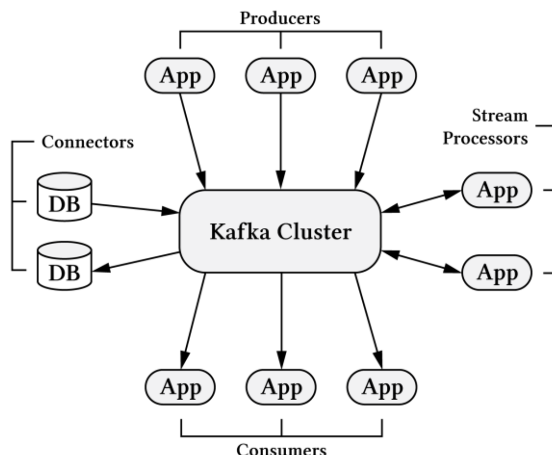


Abbildung 11: Schematische Darstellung einer Architektur um ein Kafka-Cluster
Eigene Darstellung einer Grafik aus [Apa2020b]

Kafka operiert innerhalb eines Rechnernetzes auf mehreren Knoten. Abbildung 11 zeigt eine schematische Darstellung der Komponenten einer Architektur, welche ein Kafka-Cluster als Kommunikator nutzt. Das Cluster wird an mehrere Apps oder Services angebunden, die Datenströme produzieren (*Producers*) oder konsumieren (*Consumers*) oder nacheinander konsumieren, transformieren und wieder produzieren (*Stream Processors*).

Kafka organisiert die Datenströme in sogenannten *Topics*. Diese Topics werden unterteilt in Partitionen (*Partitions*), die von Kafka wiederum auf verschiedene Rechnerknoten verteilt werden. Von einer solchen Partition existieren meist mehrere Repliken im Rechnercluster. Eine Partition ist eine geordnete, unveränderbare Folge von Datensätzen, an die kontinuierlich angehängt wird. Jedem Datensatz in den Partitionen wird eine eindeutige ID-Nummer zugewiesen, die als *Offset* bezeichnet wird (siehe Abbildung 12). Ein neuer Datensatz, der in eine Topic geschrieben wird, wird im Folgenden als Nachricht bezeichnet.

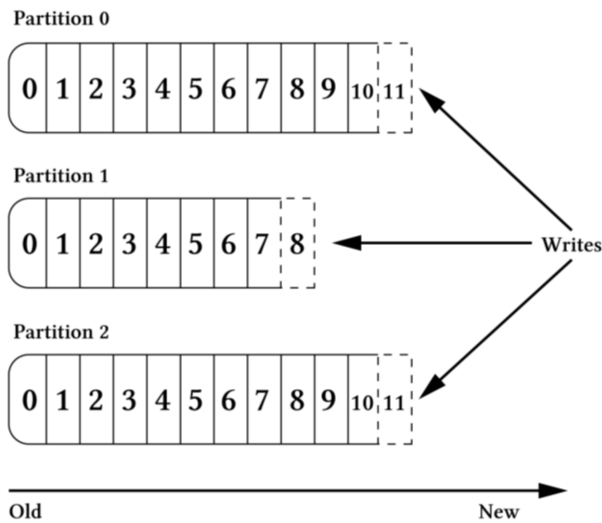


Abbildung 12: Aufbau einer Kafka-Topic mit mehreren Partitionen
Eigene Darstellung einer Grafik aus [Apa2020b]

Als *Kafka Producer* werden Anwendungen oder Services bezeichnet, welche neue Nachrichten in eine oder mehrere Topics schreiben. *Kafka Consumer* lesen hingegen neu ankommende Nachrichten aus einer oder mehreren Topics aus.

Kafka Consumer können in sogenannte *Consumer Groups* unterteilt werden (siehe Abbildung 13). Wenn alle Consumer-Instanzen zur selben Consumer-Gruppe gehören, werden die Datensätze durch das Cluster auf die Instanzen verteilt. Wenn alle Consumer zu unterschiedlichen Gruppen zugeordnet werden, sendet Kafka jeden Datensatz an alle Consumer-Prozesse.

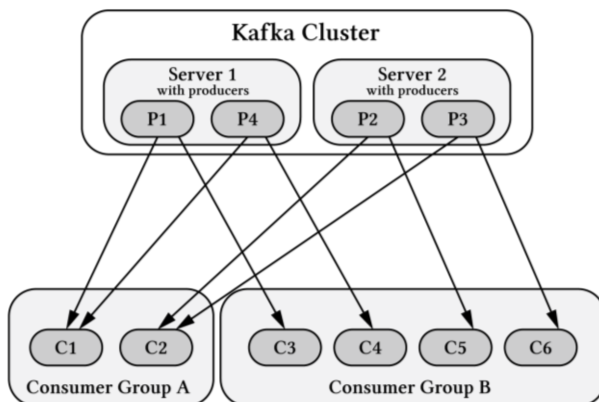


Abbildung 13: Kafka Consumer Groups
Eigene Darstellung einer Grafik aus [Apa2020b]

Apache Spark, welches in den Datenprozessoren zum Einsatz kommt, bietet Schnittstellen zur Integration von Kafka und Spark sowie simple Implementierungskonzepte für den Versand und den Empfang von Kafka-Nachrichten an. Eine besonders einfache Möglichkeit, strukturierte Daten von Kafka zu empfangen und zu verarbeiten, bietet das Sub-Package *Spark Structured Streaming*. Beide Technologien sind in den Zonenapplikationen eng miteinander verknüpft. Ein Codebeispiel ist in Kapitel 6.4 zu finden.

6.3.3 *Apache Spark* und *Spark Structured Streaming*

Apache Spark ist ein Open-Source-Framework zur verteilten Datenverarbeitung in einem Rechnercluster. In der Konzeptionsphase für die entwickelte Architektur fiel die Wahl der Technologie für die Datenstromverarbeitung auf Spark, weil es sich – wie auch Kafka – um das weitest verbreitete Tool handelt und weil es sich sehr einfach gemeinsam mit Kafka verwenden lässt.

Die folgende Beschreibung von Apache Spark beruht auf der ebenfalls frei verfügbaren Apache-Spark-Dokumentation [Apa2020c].

Spark-Applikationen werden in Prozessgruppen organisiert und in einem Cluster ausgeführt. Der *SparkContext* kann als Sparks Haupt- und Treiberprogramm verstanden werden, das eine Verbindung zu einem Cluster-Manager herstellt – Spark liefert einen eigenen Standalone-Cluster-Manager mit – und die Prozessgruppen auf die Knoten im Cluster verteilt (siehe Abbildung 14).

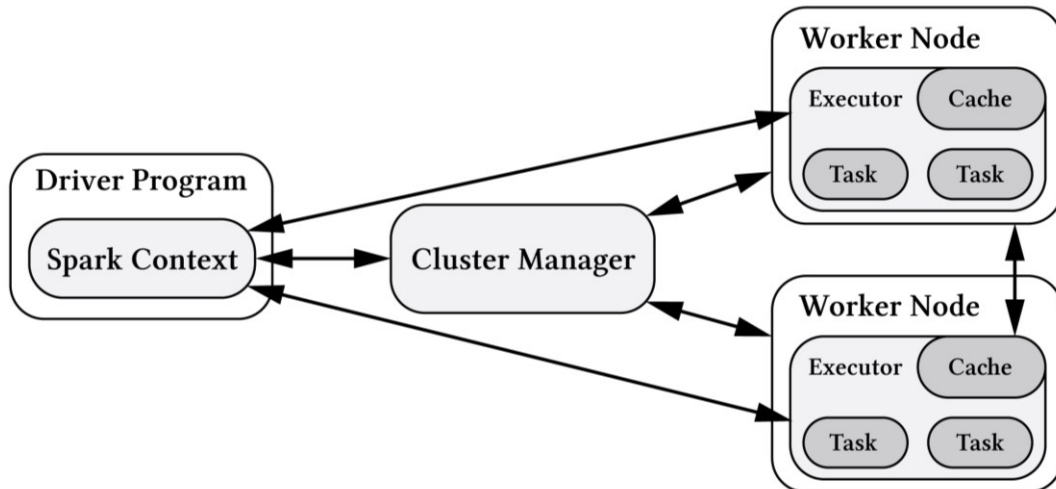


Abbildung 14: Architekturschaubild von *Apache Spark*
Eigene Darstellung der Grafik aus [Apa2020c]

Zunächst werden durch den SparkContext sogenannte Exekutoren auf den Knoten initialisiert. Daraufhin wird der Anwendungscode der Spark-Applikationen, der in JAR- oder Python-Dateien definiert ist, an die Exekutoren gesendet. Nachdem der Code den Exekutoren bekannt ist, kann der SparkContext diesen einzelne Aufgaben bzw. *Tasks* zuweisen [vgl. Apa2020c]. Solche Tasks sind atomare Transformationsoperationen auf einzelnen Datensätzen, zum Beispiel *Map*-, *Reduce*- oder *Filter*-Operationen.

Spark basiert auf dem sogenannten *Micro-Batching*. Dabei werden Datensätze innerhalb eines Datenstroms über einen sehr geringen Zeitraum gesammelt und als kleine Stapel (*Batches*) verarbeitet. Die Größe der Batches kann konfiguriert werden.

Das Spark-Framework ist in mehrere Sub-Packages gegliedert. Die wichtigsten und meist verwendeten sind *Spark Core*, *Spark SQL* und *Spark Streaming*. Spark Core wird für die Nutzung von Spark SQL und Spark Streaming vorausgesetzt.

Operationen in Spark basieren immer auf den sogenannten RDDs („*Resilient Distributed Datasets*“; robuste verteilte Datensätze). Alle Datensätze werden in Form von solchen RDDs repliziert und fehlertolerant im Rechnernetz verwaltet. Auf den RDDs können verschiedene Transformationsfunktionen ausgeführt werden, wie beispielsweise die genannten Map-, Reduce- oder Filter-Operationen.

Seit Veröffentlichung von Spark 2.x wird von den Spark-Entwicklern die Verwendung der *DataFrame*-API empfohlen, welche eine Abstraktionsebene für die Implementierung von RDD-Operationen darstellt. Mithilfe der API können einfach Datentransformationen implementiert werden, ohne RDD-Operationen zu verwenden. Wird die DataFrame-API verwendet, optimiert Spark automatisch die interne Verteilung und Verarbeitung der RDDs.

Für die Implementierung der Datenprozessoren in den Zonenapplikationen kommen grundsätzlich zwei Möglichkeiten infrage. Wird das Spark-Streaming-Package verwendet, basiert die Implementierung der Datenverarbeitung auf den RDD-Operationen. Spark Structured Streaming, welches sich im Spark-SQL-Package befindet, basiert auf der DataFrame-API. Dies vereinfacht insbesondere die Syntax und Codestruktur wenig komplexer Transformationen wie Datentypkonvertierungen oder Spaltenumbenennungen.

Da im Anwendungsfall drei Datenquellen existieren, die allesamt strukturierte und semistrukturierte Daten übermitteln (JSON- und CSV-Format), eignet sich Spark Structured Streaming besonders gut als Technologie für die Datentransformationen. Zudem kann mit nur wenigen Zeilen Code ein Kafka Consumer als Eingangsdatenstrom konfiguriert werden (siehe Kapitel 6.4, Listing 2).

Die Datentransformation selbst funktioniert mit Spark Structured Streaming nach dem „Pipes and Filter"-Architekturmuster. Die Filter stellen dabei verschiedene Spaltenoperationen wie Typkonvertierungen oder sogenannte *User Defined Functions* dar, mithilfe derer Transformationen auf einem oder mehreren Spaltenwerten innerhalb eines DataFrame umgesetzt werden können. Die Verbindung (*Pipes*) mehrerer aufeinanderfolgender Datenverarbeitungsschritte innerhalb der Datentransformation passiert implizit durch das Spark-Framework.

6.3.4 Scala

Das Spark-Framework ist in den Programmiersprachen *Java*, *Scala* und *Python* verfügbar. Als Programmiersprache für die Zonenapplikationen wurde Scala ausgewählt, da die Sprache im Gegensatz zu Python über eine statische Typisierung verfügt und die Spark-Bibliothek für Scala im Vergleich zur Java-Bibliothek erweitert ist. Einige Transformationen können mithilfe von Scala mit weniger Code umgesetzt werden als mit Java.

Scala ist eine rein objektorientierte und funktionale Sprache. Sämtliche Java-Bibliotheken können in einer Scala-Applikation verwendet werden. Aus der fertigen Applikation lässt sich eine Java-Archivdatei (JAR) erzeugen.

Mithilfe des *spark-submit*-Befehls kann diese innerhalb eines Spark-Kontextes ausgeführt werden. Dadurch können mehrere Applikationen im selben Spark-Kontext operieren, ohne dass dieser für jede Anwendung einzeln konfiguriert und gestartet werden muss.

6.3.5 Apache Cassandra

Apache Cassandra ist ein freies Datenbankmanagementsystem, welches für die Verwaltung sehr großer Datenmengen entwickelt wurde. Cassandra ist eine NoSQL-Datenbank, welche – wie auch Kafka – auf einem Rechnercluster basiert, in dem Daten redundant auf mehrere Knoten verteilt werden. Die Performanz von Lese- und Schreiboperationen steigt linear mit der Anzahl der Knoten im Cluster. Aus diesen Gründen wurde Cassandra als Technologie ausgewählt, mit der die Datenströme persistiert werden können. Die folgende Beschreibung der Cassandra-Technologie fußt auf der Dokumentation der Apache Software Foundation [Apa2020a].

Für Cassandra existieren verschiedene Konnektoren, mithilfe derer Schnittstellen zwischen Cassandra und beispielsweise Apache Spark oder Apache Kafka umgesetzt werden können. In der Data-Lake-Architektur ist keine direkte Kommunikation zwischen Kafka und Cassandra erforderlich, daher kommt nur der Spark-Cassandra-Connector[4] zum Einsatz.

[4] https://mvnrepository.com/artifact/com.datastax.spark/spark-cassandra-connector

Die von Cassandra genutzte Datenbankabfragesprache *Cassandra Query Language* (CQL) ist stark ähnlich zu SQL. Eine Cassandra-Tabelle ist eine multidimensionale Map, die mithilfe des Primärschlüssels indexiert wird. Aus diesem Grund können Sortier- und Filteroperationen – im Gegensatz zu SQL – nur auf Spalten angewandt werden, die Teil des Primärschlüssels sind oder die mithilfe des "`CREATE INDEX`"-Befehls indiziert wurden.

Für die Aggregationen auf den Pandemiedaten müssen Aggregate (Durchschnittswerte, Minimum, Maximum) sämtlicher persistierten Pandemiedaten gebildet werden. Die resultierenden aggregierten Datenzeilen werden daraufhin schnell aufeinanderfolgend in die Datenbank geschrieben. Bei ausreichender Rechenkapazität ist Cassandra in der Lage, sowohl die Abfragen und die Berechnungen der Aggregate als auch die Schreibprozesse innerhalb weniger Millisekunden durchzuführen.

6.4 Implementierungsbeispiel

Für die prototypische Implementierung der entwickelten Data-Lake-Architektur ist die Gestaltung der Datenverarbeitung in den einzelnen Zonen besonders relevant. Während bei der Verwendung von Kafka nur wenige Konfigurationen durch den Entwickler vorgenommen werden können, kann die Implementierung der Datentransformationen in den Spark-/Scala-Applikationen auf viele unterschiedliche Weisen umgesetzt werden.

In diesem Kapitel werden deshalb die Schlüsselbausteine einer exemplarisch gewählten Spark-Zonenapplikation dargestellt – von der Initialisierung des Spark-Kontextes und der Datenprozessoren über den Empfang einer Nachricht von Kafka, den Funktionen zur Datentransformation selbst bis hin zur Übermittlung zurück an Kafka.

Die Persistierung und Abfrage von Daten mithilfe von Apache Cassandra wird hierbei vernachlässigt, da diese lediglich für die Realisierung der Aggregationen und Kombinationen verwendet wurde. Für das Ziel der Arbeit, eine zonenbasierte Architektur für die Datenstromverarbeitung zu entwickeln, ist die Batch-Datenverarbeitung mit Cassandra weniger relevant.

Als Beispiel wird im Folgenden der Datenprozessor der Länderdaten in der Harmonized Zone verwendet, da dieser einerseits verschiedene Datentransformationen beinhaltet, andererseits aber auch durch eine geringere Spaltenanzahl des Datensatzes mit weniger Codezeilen auskommt und sich hier übersichtlich darstellen lässt.

```scala
1   object HarmonizedZoneApplication extends App {
2     override def main(args: Array[String]) {
3       val servers = "127.0.0.1:9092"
4       val master = "local[*]"
5
6       val sparkConf = new SparkConf()
7         .setAppName("HarmonizedZoneApplication")
8         .setMaster(master)
9
10      val spark = SparkSession.builder().config(sparkConf).getOrCreate()
11
12      // ...
13      // Länderdaten-Prozessor
14      val countryDataProcessor: CountryDataProcessor =
15        new CountryDataProcessor(spark, servers)
16      val countryDf = countryDataProcessor.processCountryDataStream()
```

```
17      // Transformierte Daten an Kafka Topic senden
18      val dataStreamWriter= new KafkaDataStreamWriter(spark, servers)
19      dataStreamWriter.writeJsonDataStream(countryDf, "HarmonizedCountryData")
20
21      // ...
22      // Spark Session offen halten
23      spark.streams.awaitAnyTermination()
24    }
25  }
```

Listing 1: Hauptklasse der Harmonized Zone (gekürzt)

In der Hauptklasse jeder Zonenapplikation (siehe Listing 1) werden die Datenprozessoren initialisiert. Dazu muss zunächst die Spark-Konfiguration (`SparkConf`, Zeile 6-8) definiert und die Spark Session erstellt werden. Dies geschieht mithilfe eines Builders (Zeile 10). Durch die Methode `awaitAnyTermination()` in Zeile 24 wird die Session so lange offen gehalten, bis der definierte Spark-Kontext terminiert. In diesem Fall ist keine Termination des Spark-Kontextes vorgesehen, das bedeutet, die Session wird offengehalten, solange die Applikation läuft.

In den Zeilen 14 bis 16 wird der Länderdatenprozessor erstellt und die Methode aufgerufen, die die eigentliche Datentransformation ausführt. Die verarbeiteten Daten werden anschließend durch den `KafkaDataStreamWriter` in eine neue Kafka-Topic `"HarmonizedCountryData"` geschrieben (Zeile 17-19). Die Implementierung des Versands der Daten an Kafka wird mithilfe von Listing 5 und Listing 6 erläutert.

```
1   class CountryDataProcessor(sparkSession: SparkSession, servers: String) {
2
3     def processCountryDataStream(): DataFrame = {
4
5       // Kafka Stream Reader
6       val countryDataFromRawZoneDataFrame =
7         sparkSession.readStream.format("kafka")
8                                .option("kafka.bootstrap.servers", servers)
9                                .option("subscribe", "RawCountryData").load()
10
11      val columnsStruct = new StructType(Array(
12        StructField("Rank", StringType),
13        StructField("Name", StringType),
14        StructField("Population_2020", StringType),
15        StructField("Growth_Rate", StringType),
16        StructField("Area", StringType),
17        StructField("Density", StringType),
18        StructField("Arrived_In_Raw_Zone_At", StringType),
19        StructField("Processed_In_Raw_Zone_At", StringType)))
20
21      // Konvertierung JSON-String zu DataFrame
22      val structuredCountryDataFromRawZone = JsonDataStreamUtils
23        .jsonStructuredStringToStructuredDataFrame(
24          countryDataFromRawZoneDataFrame, columnsStruct, sparkSession)
```

```
25      // Datenumformung
26      structuredCountryDataFromRawZone
27        .withColumn("Rank", col("Rank").cast(IntegerType))
28        .withColumn("Population_2020",
29          UserDefinedFunctions.multiplyBy1000Udf.apply(col("Population_2020"))
30                                              .cast(IntegerType))
31        .withColumn("Growth_Rate", col("Growth_Rate").cast(DoubleType))
32        .withColumn("Area", col("Area").cast(IntegerType))
33        .withColumn("Density", col("Density").cast(DoubleType))
34        .withColumn("Arrived_In_Raw_Zone_At",
35          col("Arrived_In_Raw_Zone_At").cast(TimestampType))
36        .withColumn("Arrived_In_Harmonized_Zone_At",
37          col("Arrived_In_Harmonized_Zone_At").cast(TimestampType))
38        .withColumn("Processed_In_Raw_Zone_At",
39          col("Processed_In_Raw_Zone_At").cast(TimestampType)
40        )
41    }
42  }
```

Listing 2: Datenprozessor der Länderdaten in der Harmonized Zone (gekürzt)

In Listing 2 ist die Implementierung des Datenprozessors der Landing Zone abgebildet. Diese ist gekürzt, so wurden beispielsweise die Felder `Population_2019`, `Arrived_In_Landing_Zone_At` und `Processed_In_Landing_Zone_At` zum Zwecke der Übersichtlichkeit weggelassen.

In den Zeilen 5 bis 9 wird mithilfe von `SparkSession.readStream()` eine Subscription auf die Kafka-Topic „RawCountryData“ gestartet. Hierbei können mithilfe von `option()` weitere Optionen zur Konfiguration des Kafka Consumers übermittelt werden. Die entsprechenden Zeilen wurden hier ebenfalls gekürzt, um die Übersichtlichkeit zu wahren.

Die Kafka-Topic „RawCountryData“ liefert – wie alle Topics ab der Landing Zone – JSON-strukturierte Strings zurück. Um diese in ein `DataFrame`-Objekt überführen zu können, müssen zunächst die Objektstruktur und die Attribute spezifiziert werden. Dies geschieht mithilfe des Spark-internen Typs `StructType` in den Zeilen 11 bis 19. Da sämtliche Datenattribute in der Raw Zone als Strings vorliegen, wird bei der Konvertierung von JSON-String zu `DataFrame` zunächst der Typ String angenommen.

Die Konvertierung findet in den Zeilen 21 bis 24 mithilfe der eigens definierten Funktion `jsonStructuredStringToStructuredDataFrame(...)` statt. Diese Funktion erhält den StructType als Parameter und wird nachfolgend anhand von Listing 3 erläutert.

Die Transformationen der Daten werden in den Zeilen 25 bis 40 implementiert. Hier erhalten die String-Daten ihre jeweiligen Datentypen (*Integer, Double, Timestamp*) mithilfe der Spark-SQL-Funktion `cast(<Type>)`. Schlägt diese Konvertierung fehl, enthält der entsprechende Datensatz anschließend in der nicht-konvertierbaren Spalte den Wert `null`.

In den Zeilen 29 bis 31 wird eine `UserDefinedFunction` verwendet, um die Bevölkerungszahl mit 1.000 zu multiplizieren. Der Aufbau dieser Funktion wird anhand von Listing 4 genauer beschrieben. Die Multiplikation stellt eine sehr simple Transformation dar. Die nutzerdefinierten Funktionen werden allerdings in der Distilled und Delivery Zone auch dazu genutzt, Abfragen an die Cassandra-Datenbank zu tätigen und so Werte aus den Datenströmen mit bereits in der Datenbank persistierten Daten zu kombinieren.

Da die Abfragen an die externe Datenbank zeitintensiver sind als die „Spark-internen“ Umformungen, fallen diese Transformationen bei der Betrachtung der Echtzeitfähigkeit der Datenverarbeitung über alle Zonen hinweg besonders stark ins Gewicht (siehe Kapitel 7).

```
1   object JsonDataStreamUtils {
2
3     def jsonStructuredStringToStructuredDataFrame(dataFrame: DataFrame,
4       structType: StructType, spark: SparkSession): DataFrame = {
5
6       import spark.implicits._
7
8       dataFrame
9         .select($"value" cast "string" as "json", $"timestamp")
10        .select(from_json($"json", structType) as "data", $"timestamp")
11        .select("timestamp", "data.*")
12        .withColumnRenamed("timestamp", "Arrived_In_Harmonized_Zone_At")
13    }
14  }
```

Listing 3: Konvertierung der JSON-strukturierten Nachricht in ein *DataFrame*-Objekt

Die `jsonStructuredStringToStructuredDataFrame()`-Funktion wird in Listing 3 gezeigt. Sie erhält als Parameter den DataFrame, der aus dem `readStream()`-Aufruf resultiert (Listing 2, Zeile 7). Im `value`-Attribut des DataFrame ist der komplette Datensatz als JSON-strukturierter String enthalten. Mithilfe des zweiten Parameters – des StructTypes – und der `from_json()`-Funktion wird dieser String automatisch in ein DataFrame-Objekt gemäß der im StructType definierten Struktur überführt. Da es sich bei der `from_json()`-Methode um eine Spark-SQL-interne Funktion handelt, wird hierfür auch die Spark Session und das Paket `SparkSession.implicits` benötigt.

Neben der Umformung in einen DataFrame extrahiert die Funktion außerdem das `timestamp`-Attribut, welches von Kafka an jede Nachricht angehängt wird. Das Attribut wird in eine neue DataFrame-Spalte namens `Arrived_In_Harmonized_Zone_At` geschrieben.

```
1   object UserDefinedFunctions {
2     val multiplyBy1000Function: Double => Double = doubleToMultiply => {
3       doubleToMultiply * 1000
4     }
5
6     val multiplyBy1000Udf: UserDefinedFunction = udf(multiplyBy1000Function)
7
8     // ...
9   }
```

Listing 4: Beispiel einer simplen *UserDefinedFunction* in der Harmonized Zone

Listing 4 zeigt eine *UserDefinedFunction* für die Multiplikation mit 1.000. Die Funktion, die einen Parameter des Typen Double annimmt, gibt ebenfalls wieder einen Wert des Typen Double zurück (Zeile 2-4). In Zeile 6 wird die Funktion mithilfe der `udf()`-Methode in Spark-SQLs UserDefinedFunction-Objekt verpackt. Anschließend kann die Funktion im Datenprozessor auf eine DataFrame-Spalte angewandt werden (Listing 2, Zeile 30).

```scala
1  class KafkaDataStreamWriter(sparkSession: SparkSession,
2                              servers: String) {
3
4    def writeJsonDataStream(dataFrame: DataFrame,
5                            topic: String): StreamingQuery = {
6      import sparkSession.implicits._
7
8      val dfWithTimestamp = dataFrame
9        .withColumn("Processed_In_Harmonized_Zone_At",
10         current_timestamp())
11
12     val outputDf = dfWithTimestamp.selectExpr(
13       "CAST(to_json(struct(*)) AS STRING) AS value"
14     ).as[String]
15
16     val kafkaSink = new KafkaForeachSink(topic, servers)
17
18     outputDf.writeStream.foreach(kafkaSink).outputMode("update").start()
19   }
20 }
```

Listing 5: *KafkaDataStreamWriter*-Implementierung

Der KafkaDataStreamWriter ist in Listing 5 abgebildet und verfügt über eine `writeJsonDataStream()`-Methode. Diese erhält als Parameter den DataFrame sowie den Namen der Topic, an welche der Datensatz gesendet werden soll. In den Zeilen 8 bis 10 wird der Zeitstempel hinzugefügt, an dem die Datenverarbeitung abgeschlossen ist (`Processed_In_Harmonized_Zone_At`).

Daraufhin werden alle Attribute aus dem DataFrame in einen JSON-String konvertiert (Zeile 12-14). Zuletzt wird der JSON-String mithilfe eines *ForeachWriters* (`KafkaForeachSink`) an die Kafka-Topic gesendet (Zeile 16-18). Der ForeachWriter wird benötigt, um die DataFrames kontinuierlich an ein externes System zu senden. Die Implementierung des Writers folgt in Listing 6.

```scala
1 protected class KafkaForeachSink(topic: String, servers: String)
2                   extends ForeachWriter[String] {
3
4   val kafkaProperties = new Properties()
5   kafkaProperties.put("bootstrap.servers", servers)
6   kafkaProperties.put("key.serializer",
7     "org.apache.kafka.common.serialization.StringSerializer")
8   kafkaProperties.put("value.serializer",
9     "org.apache.kafka.common.serialization.StringSerializer")
10
11  var producer: KafkaProducer[String, String] = _
12
13  def open(partitionId: Long, version: Long): Boolean = {
14    producer = new KafkaProducer(kafkaProperties)
15    true
16  }
17
18  def process(value: String): Unit = {
19    producer.send(new ProducerRecord(topic, value))
20  }
```

```
21  def close(errorOrNull: Throwable): Unit = {
22    producer.close()
23  }
24}
```

Listing 6: Implementierung des *KafkaForeachSink*

Der KafkaForeachSink muss drei Methoden der abstrakten Superklasse ForeachWriter implementieren – `open()`, `process()` und `close()`. In der open()-Methode wird der Kafka Producer erstellt (Listing 6, Zeile 13-16). Dazu wird die Kafka-Konfiguration verwendet, die bereits in den Zeilen 4 bis 9 vorgenommen wurde. Die Kafka-Serverkonfiguration sowie der Name der Topic werden dem ForeachSink bereits im Konstruktor übergeben.

Der eigentliche Versand der Nachrichten an Kafka erfolgt in der process()-Methode. Dort erhält der Kafka Producer den `value` – also den JSON-formatierten String – sowie den Namen der Topic, an die der String gesendet werden soll (Zeile 18-20). In der close()-Methode in den Zeilen 21 bis 23 wird der Kafka Producer schließlich beendet.

7 Evaluation

In diesem Kapitel wird die in Kapitel 6 beschriebene Architektur evaluiert. Zunächst wird eine allgemeine Einschätzung der Architektur und ihrer Implementierung vorgenommen. Dabei werden die Vor- und Nachteile der Architektur beschrieben und verbesserungsfähige Komponenten benannt (Kapitel 7.1). In Kapitel 7.2 wird anschließend die Echtzeitfähigkeit der entstandenen Data-Lake-Architektur bewertet. Als Grundlage für diese Bewertung dienen Zeitmessungen der Datenverarbeitung in verschiedenen Konfigurationsvarianten der Architektur.

7.1 Bewertung der entstandenen Architektur und Implementierung

Im Folgenden werden einige zentrale Vorteile der entstandenen Architektur benannt. Daraufhin werden einige Schwachpunkte der Implementierung identifiziert und mögliche Maßnahmen zur Behebung dieser Probleme beschrieben. Abschließend wird eine Möglichkeit zur bequemen Konfiguration der Datenverarbeitung in den Datenprozessoren mithilfe von generischen Methoden diskutiert.

Der vorrangige Grund, einen zonenbasierten Data Lake zu nutzen, ist zugleich der größte Vorteil einer solchen Architektur: die hohe Wiederverwendbarkeit der rohen und vorverarbeiteten Daten. Da die Daten in all ihren Verarbeitungsgraden im Data Lake verfügbar sind, ist eine hohe Flexibilität bei der späteren Wiederverwendung in neuen Anwendungsfällen gewährleistet. Im konkreten Anwendungsszenario könnte es vorkommen, dass die Pandemiedaten später nicht nur mit Wetterdaten verschiedener Standorte kombiniert werden sollen, sondern beispielsweise auch mit Verkehrsdaten, um den Zusammenhang des Infektionsgeschehens mit der Mobilität der Bevölkerung zu untersuchen. Dazu könnten bereits die harmonisierten Pandemiedaten verwendet werden; es wäre nicht notwendig, die Pandemie-Rohdaten erneut aus dem Git-Repository der John-Hopkins-Universität zu beziehen und die Daten müssten für den neuen Anwendungsfall nicht nochmals sämtliche Verarbeitungsschritte durchlaufen.

Ein weiterer wichtiger Vorteil der entstandenen Architektur ist die gute Skalierbarkeit ihrer Komponenten. Alle Bestandteile der Architektur – das Kafka-Cluster, die Spark-Zonenapplikationen und die Cassandra-Datenbank – können auf mehreren Knoten in einem Rechner-Cluster operieren. Werden die Daten in einer hohen Frequenz in den Datenströmen verschickt, könnten die Datensätze durch eine Partionierung der Kafka-Topics auf mehrere identische Spark-Zonenapplikationen auf verschiedenen Rechnerknoten verteilt und so der Durchsatz gesteigert werden. Auch bei der Cassandra-Datenbank-Komponente steigt die Anzahl möglicher gleichzeitiger Lese- und Schreiboperationen linear mit der Anzahl der Datenbank-Knoten im Cluster (siehe Kapitel 6.3.5).

Durch Verwendung der Spark-DataFrame-API in den Zonenapplikationen gestaltet sich außerdem die Veränderung der Logik in den Datenprozessoren sehr einfach. Durch das Hinzufügen einer einzigen Codezeile kann eine neue Spalte in die Datenverarbeitung eines Datenprozessors integriert werden. Durch wenige Anpassungen im Code ist die Veränderung der Verarbeitungslogik für eine Spalte des zu verarbeitenden Datensatzes möglich. Ändert sich zum Beispiel der Datentyp eines Datensatzattributs vom Typen Double zum Typen String, reicht es aus, den Parameter der Cast-Funktion im Datenprozessor zum neuen Datentyp abzuändern (vgl. Listing 2 in Kapitel 6.4).

In diesem Zusammenhang lässt sich allerdings auch ein Problem der Implementierung in der beschriebenen Form identifizieren. In der Landing Zone werden die eingehenden Rohdatensätze erstmalig in ein DataFrame-Objekt umgewandelt. Liegt der Rohdatensatz dabei im CSV-Format vor, werden die Spalten der Quell-CSV nach der Reihenfolge ihres Auftretens den Spalten des DataFrame-Konstruktors (vgl. Listing 2, Zeile 11-19) zugeordnet. Wenn sich in der Quell-CSV also die Reihenfolge der Spalten verändert oder eine Spalte hinzukommt, erfolgt eine falsche Zuordnung der Attribute zu den DataFrame-Spalten.

Falsch zugeordnete Daten können zu Fehlschlägen bei der Datentypkonvertierung oder bei der Verwendung von UserDefinedFunctions führen, die einen bestimmten Datentypen als Input-Parameter erwarten. Dies resultiert im schlimmsten Fall zu Abstürzen der jeweiligen Zonenapplikation. Um dieses Problem zu umgehen, könnten die Header-Zeilen der CSV-Datensätze, die die Spalten mit ihren Bezeichnungen in der richtigen Reihenfolge enthalten, an jede Kafka-Nachricht angehängt werden. Die Zuordnung der Rohdaten zu den DataFrame-Spalten würde dann nicht mithilfe der Reihenfolge geschehen, sondern durch Auslesen der Header-Zeile und eine anschließende dynamische Zuordnung.

Ein weiteres Verbesserungspotential wurde bei der Implementierung der Aggregation in der Distilled Zone identifiziert. Bei der Aggregation werden von den Datensätzen zwei Datenprozessoren hintereinander durchlaufen. Der `AggregatedPandemicDataProcessor` erhält Aggregationen von Datenbank-Zeilen als Eingangsdatenstrom, die mithilfe von CQL-Abfragen aus der Cassandra-Datenbank gewonnen werden. Der Prozessor fügt anschließend unter anderem Geo-Koordinaten an die aggregierten Datensätze an. Der erzeugte Ausgangsdatenstrom mit den aggregierten Daten durchläuft anschließend den `PandemicDataProcessor` der Distilled Zone. Dieser verarbeitet gleichermaßen aggregierte und nicht-aggregierte Pandemiedaten. Dabei wird auch das `Administrative_Level`-Attribut an die Datensätze angefügt, sodass später identifiziert werden kann, ob es sich bei der jeweiligen Datenzeile um einen aggregierten oder nicht-aggregierten Datensatz handelt. Zuletzt werden – entsprechend der Aufgabe der Distilled Zone – unbrauchbare Spalten aus den Datensätzen entfernt.

Da bei der Evaluation der Echtzeitfähigkeit (siehe Kapitel 7.2) festgestellt wurde, dass der Versand zwischen den Datenprozessoren über das Kafka-Cluster deutlich mehr Zeit in Anspruch nimmt als die Datenverarbeitung selbst, könnte die Durchlaufzeit für aggregierte Datensätze verringert werden, indem die Verarbeitung der aggregierten Pandemiedaten in einen einzigen Datenprozessor verlagert wird – auch wenn dies zusätzliche Redundanzen im Programmcode zur Folge hätte.

Betrachtet man den Anwendungscode über alle Zonenapplikationen hinweg, können zahlreiche Redundanzen im Code festgestellt werden. So enthalten beispielsweise alle Zonenapplikationen ein Package für Hilfsmethoden, dessen Inhalt in allen Anwendungen weitgehend identisch ist. Auch der Kafka Producer, der für den Versand der verarbeiteten Daten an das Kafka-Cluster zuständig ist, existiert in jeder Applikation. Im Sinne einer Microservice-Architektur könnten diese Redundanzen vermieden werden, indem die Hilfsfunktionen (*utils*-Package) sowie der Kafka Producer (*streamWriter*-Package) in eigene Applikationen oder Services ausgelagert werden. Dabei wäre allerdings zu prüfen, ob der daraus resultierende zusätzliche Datenversand zwischen den Applikationen oder Services zu einem deutlichen Anstieg der Datengesamtverarbeitungszeit führen würde.

Auch wenn die entstandene Architektur über mehrere Master Datasets in Form von Cassandra-Datenbanktabellen verfügt, handelt es sich um keine Lambda- oder Kappa-Architektur. Anders als in einer Lambda- oder Kappa-Architektur werden die Datenströme erst nach ihrer Verarbeitung mithilfe der Spark-Zonenapplikationen persistiert. Fallen also gleichzeitig alle redundanten Applikationen einer Zone aus, führt dies zu Datenverlusten, da in diesem Fall auch keine Persistierung der Daten vorgenommen werden kann.

Bei einer konsequenten Anwendung des Lambda- oder Kappa-Architekturpatterns würden die Datensätze über Schnittstellen an den Kafka-Topics abgegriffen und persistiert (vgl. Abbildung 7 in Kapitel 5.1). Die Persistierung der Daten würde bereits vor der Verarbeitung durch die jeweilige Zonenapplikation erfolgen. Durch eine zeit- oder manuell gesteuerte erneute Verarbeitung der im Master Dataset gespeicherten Daten wäre dann sichergestellt, dass keine Datenlücken durch Ausfälle von Komponenten entstehen.

Bei der Implementierung der Architektur wurden als Technologien Apache Kafka, Apache Spark sowie Apache Cassandra ausgewählt. Für jede dieser Technologien existieren allerdings mehrere Alternativen. Da in dieser Arbeit keine Technologie-Evaluation durchgeführt wurde, ist im Rahmen weiterführender wissenschaftlicher Arbeiten zu prüfen, ob es Technologien gibt, die sich in einem ähnlichen Anwendungskontext – innerhalb einer Data-Lake-Plattform im Datenstromkontext – besser eignen.

Insbesondere ist hierbei *Apache Flink* zu nennen, welches eine Alternative zu Spark darstellt. Im Gegensatz zu Spark werden Datenströme nicht intern in sogenannte Micro-Batches aggregiert und dann verarbeitet, sondern sie verbleiben in Form eines reinen kontinuierlichen Datenstroms. Dadurch lässt sich unter Verwendung von Flink eine geringere Latenz bei der Datenverarbeitung erzielen als mit Spark [Kar2018]. Aus diesem Grund wäre ein Vergleich zwischen Implementierungen von Datenprozessoren unter Verwendung von Spark und Flink sowie deren jeweilige Performanz interessant.

Gegenüber dem in Kapitel 4 beschriebenen beispielhaften Anwendungsszenario ist die Anzahl der Datenquellen und verschieden strukturierten Eingangsdatenströme in einem realen Anwendungsfall meist deutlich höher. Es wären also deutlich größere Mengen an Datenprozessoren vorhanden, die einerseits bei jeder Veränderung der Struktur der Eingangsdaten angepasst werden müssten, andererseits aber häufig aus denselben oder sehr ähnlichen atomaren Verarbeitungsschritten bestehen würden.

In der bestehenden Implementierung der Architektur würde mit der Erhöhung der Menge der Datenprozessoren auch die Anzahl der Code-Redundanzen proportional steigen. Um die Zahl der Redundanzen zu verringern und um es Data Scientists und Nutzern der Data-Lake-Plattform zu vereinfachen, die Verarbeitungslogiken der Datenprozessoren abzuändern, wäre die Einführung von generischen Methoden sinnvoll, mithilfe derer ein Datenprozessor dynamisch zusammengesetzt werden könnte.

```
1   def castDataTypesInDataFrame(    dataFrame: DataFrame,
2                                    columnNames: Array[String],
3                                    dataTypes: Array[DataType])     : DataFrame
4
5   def renameColumnsInDataFrame(    dataFrame: DataFrame,
6                                    oldColumnNames: Array[String],
7                                    newColumnNames: Array[DataType]): DataFrame
8
9   def trimStringColumnsInDataFrame(dataFrame: DataFrame,
10                                   columnsToTrim: Array[String])   : DataFrame
```

```
11  def formatDateColumnsInDataFrame(dataFrame: DataFrame,
12                                   dateColumnNames: Array[String],
13                                   dateFormatPattern: String)      : DataFrame
```

Listing 7: Mögliche Signaturen generischer Datentransformationsmethoden

Listing 7 zeigt mögliche Methodensignaturen solcher generischer Basis-Funktionen zur Datentransformation. Als Parameter enthält die Methode das DataFrame-Objekt, welches alle Attribute des Datensatzes enthält, sowie eine Liste der DataFrame-Spalten, auf die die jeweilige Funktion angewandt werden soll. Je nach gewünschter Funktionalität müssen noch weitere Parameter übergeben werden. Jede Funktion erzeugt ein verändertes DataFrame-Objekt als Rückgabe.

- Mithilfe einer Liste der Spaltennamen sowie einer weiteren Liste mit Datentypen könnten Datentypkonvertierungen (*Cast*-Funktionen) auf alle Spalten angewandt werden (Zeile 1-3).
- Durch Übergabe zweier String-Arrays (`oldColumnNames`, `newColumnNames`) könnten sämtliche Spaltennamen im DataFrame umbenannt werden (Zeile 5-7).
- Die String-*Trim*-Funktion könnte in einem Schritt auf mehrere DataFrame-Spalten angewandt werden (Zeile 8-9).
- Alle Spalten, die Datums- oder Zeitangaben enthalten, könnten nach einem bestimmten Pattern formatiert werden (Zeile 11-13).

Existiert eine hohe Anzahl solcher generischer Methoden, ließe sich in vielen Anwendungsfällen ein großer Teil der Datentransformationen über diese Methoden abbilden. Auf Grundlage dieses Konzepts könnte eine eigene Komponente außerhalb der Zonenapplikation entstehen, mithilfe derer Data Scientists die Verarbeitung in den Datenprozessoren konfigurieren könnten. Die Parameter der generischen Funktionen – wie beispielsweise die Spalten- und Datentyp-Listen – würden dann über diese externe Komponente in die Zonenapplikationen eingelesen werden. Die Datenverarbeitung eines Datenprozessors könnte so sogar im laufenden Betrieb angepasst werden. Eine Umsetzung dieser Idee würde allerdings eine Prüfung voraussetzen, ob und inwieweit eine weitere Komponente und der zusätzliche Datenversand die Performanz und Echtzeitfähigkeit der Zonenapplikationen beeinträchtigen würde.

7.2 Evaluation der Echtzeitfähigkeit

Im Folgenden soll die Echtzeitfähigkeit der entstandenen Architektur bewertet werden. Hierzu werden die Zeitstempel genutzt, die von den Zonenapplikationen als Attribute an die Datensätze angehängt werden (siehe Kapitel 5.2). Die Attribute werden in jeder Zone am Anfang und am Ende der Transformationen durch Apache Spark gesetzt. Die folgenden Zeitstempel sind vorhanden:

- `Arrived_In_Landing_Zone_At`
- `Processed_In_Landing_Zone_At`
- `Arrived_In_Raw_Zone_At`
- `Processed_In_Raw_Zone_At`
- `Arrived_In_Harmonized_Zone_At`
- `Processed_In_Harmonized_Zone_At`
- `Arrived_In_Distilled_Zone_At`
- `Processed_In_Distilled_Zone_At`
- `Arrived_In_Delivery_Zone_At`
- `Processed_In_Delivery_Zone_At`

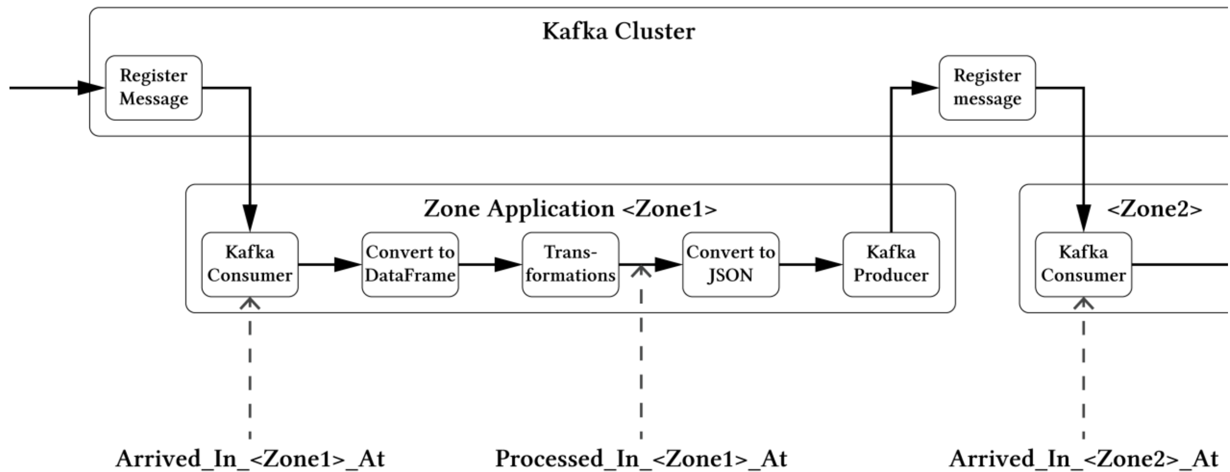


Abbildung 15: Verortung der Messpunkte innerhalb des Datenflusses

Abbildung 15 zeigt, an welchen Stellen innerhalb des Datenflusses die Zeitstempel an den Datensatz angehängt werden. Die Ankunftszeitstempel (`Arrived_In`...) entsprechen den Zeitstempeln, die Kafka automatisiert an jede Nachricht anhängt, wenn ein neuer Datensatz als Nachricht an das Kafka-Cluster gesendet und vom Kafka Consumer in der Zonenanwendung registriert wird. Die Implementierung der Auslese dieses Attributs ist in Listing 3 (Kapitel 6.4) dargestellt. Die Zeitstempel am Ende der Verarbeitung (`Processed_In`...) werden gesetzt, während der jeweilige DataFrame seinen letzten Verarbeitungsschritt in der jeweiligen Zonenapplikation durchläuft (siehe Listing 5 in Kapitel 6.4). Erst danach folgt die Umwandlung des DataFrame in einen JSON-strukturierten String, der dann als Kafka-Nachricht versandt werden kann.

Um die Echtzeitfähigkeit der Zonenarchitektur vergleichen zu können, wurde eine Applikation namens *Single Zone* entwickelt. Diese führt die Transformationen aller Zonen der Mehr-Zonen-Architektur in einer einzigen Applikation durch und verfügt über je einen Datenprozessor pro Datenquelle. Dabei werden keine Zwischenergebnisse an Kafka gesendet – wie beispielsweise die harmonisierten (Harmonized Zone) oder destillierten Daten (Distilled Zone) in der Mehr-Zonen-Architektur –, sondern lediglich die komplett verarbeiteten Daten am Ende. Dadurch kommt es während der Verarbeitung zu weniger Verzögerungen durch die mehrfache Umwandlung von DataFrames in JSON-Strings, den Versand von Nachrichten an das Kafka-Cluster sowie die Auslese von Kafka-Nachrichten durch die Kafka Consumer.

Als Dateneingangsstrom erhält jeder Datenprozessor der Single-Zone-Anwendung die Rohdaten vom DataStreamsGenerator. Als Ergebnis wird ein Datenstrom produziert, der die aggregierten und nicht-aggregierten Pandemiedaten in Kombination mit den zugehörigen Länder- und Wetterdaten enthält. In der Single-Zone-Variante können deshalb nur zwei Zeitstempel an den Datensatz angehängt werden – zu Beginn und am Ende der vollständigen Datentransformation:

- `Arrived_In_Single_Zone_At`
- `Processed_In_Single_Zone_At`

Die Verarbeitungszeiten der Single-Zone und der Mehr-Zonen-Architektur werden in jeweils zwei verschiedenen Konfigurationen gemessen – mit Anbindung an die Datenbank-Komponente Cassandra und ohne Datenbankverbindung. Daraus ergeben sich die folgenden vier Konfigurationsvarianten:

1. **die entstandene Zonenarchitektur in der in Kapitel 6 beschriebenen Form**
2. **die Zonenarchitektur ohne Nutzung von Apache Cassandra**: Ohne die Datenbank-Komponente Apache Cassandra ist es nicht möglich, die Pandemiedaten zu aggregieren oder die Pandemie-Datensätze mit den zugehörigen Länder- und Wetterdaten zu kombinieren. Die Architektur verfügt in dieser Konfiguration damit lediglich über eine eingeschränkte Funktionalität. Allerdings lässt sich mithilfe dieser Variante herausfinden, ob und inwieweit der parallele Betrieb der Cassandra-Datenbank sowie die Persistierung und die Datenabfragen die Performanz der Zonenarchitektur beeinträchtigen.
3. **die oben beschriebene Single-Zone-Anwendung**
4. **die Single-Zone-Anwendung ohne Anbindung an Apache Cassandra** mit der oben beschriebenen eingeschränkten Funktionalität

Bei der Messung der Performanz aller Varianten wurden die Verarbeitungszeiten von etwa 30.000 Datensätzen erhoben, die die vollständige Datenverarbeitung in der jeweiligen Architekturvariante durchlaufen haben. Die Messung wurde einige Minuten begonnen, nachdem alle Komponenten der Architektur erfolgreich hochgefahren wurden. In den ersten Minuten nach dem Starten der Anwendungen benötigen die Datensätze erheblich länger (teilweise über 10 Sekunden), da es durch den parallelen Start aller Applikationen zu einem hohen Ressourcenverbrauch kommt und zudem nicht alle Komponenten zum selben Zeitpunkt bereit für die Datenverarbeitung sind.

Die Messungen wurden auf einem Rechner mit Ubuntu-Betriebssystem (*Ubuntu 16.04.2 LTS*), 32 GB Arbeitsspeicher (RAM), 50 GB Festplattenspeicher sowie sechs virtuellen Prozessoren (6 VCPUs) durchgeführt.

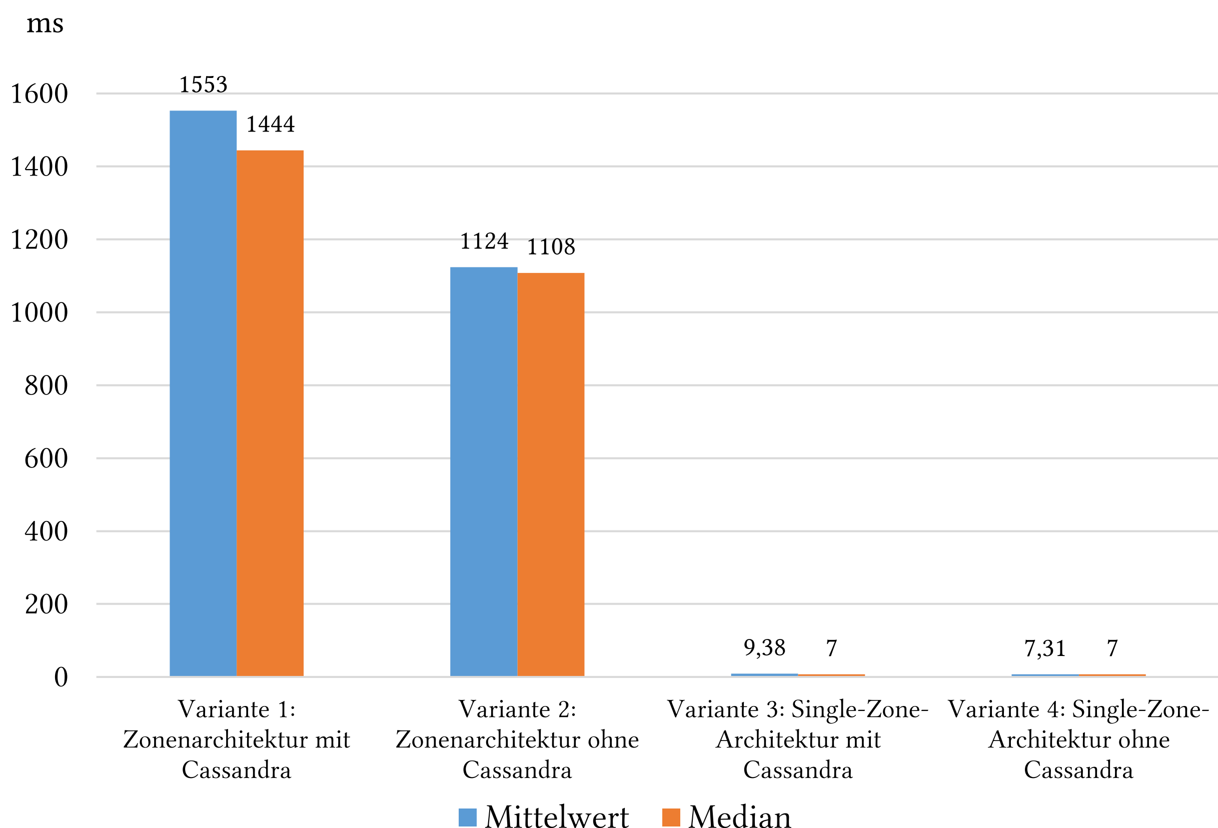


Abbildung 16: Mittelwerte und Mediane der gemessenen Verarbeitungszeiten

Abbildung 16 zeigt Mittelwerte und Mediane der gemessenen Gesamtverarbeitungszeiten in allen vier Varianten. Die Zonenarchitektur mit Cassandra (Variante 1) benötigt im Durchschnitt 1.553 Millisekunden für die vollständige Datenverarbeitung durch alle Zonen. Im Median liegt die Verarbeitungszeit etwas darunter, bei 1.444 Millisekunden. Die Diskrepanz zwischen Mittelwert und Median ist darauf zurückzuführen, dass einzelne Datensätze durch Ausfälle und Verzögerungen der Cassandra-Komponente über 10 Sekunden für die vollständige Verarbeitung benötigen. Solche „Ausreißer" treten bei Variante 2 ohne Cassandra nicht auf.

Beim Vergleich der beiden Mehr-Zonen-Varianten miteinander fällt auf, dass Variante 2 ohne Cassandra die Daten im Median 336 Millisekunden schneller verarbeitet als Variante 1 mit Cassandra, im Mittel sogar 429 Millisekunden schneller. Dies lässt sich zum einen durch den zusätzlichen parallelen Betrieb der Cassandra-Datenbank auf demselben Rechner, zum anderen durch die Datenbankabfragen in den Datenprozessoren erklären, die zusätzliche Zeit in Anspruch nehmen.

Die Verarbeitungszeiten der Single-Zone-Varianten 3 und 4 fallen deutlich geringer aus. So wurde bei Variante 3 mit Cassandra-Anbindung im Median eine Gesamtverarbeitungszeit von 7 Millisekunden gemessen. Im Mittel liegt die Verarbeitungszeit geringfügig höher, bei etwa 9 Millisekunden. Bei Variante 4 ohne Cassandra benötigt die komplette Datentransformation 7 Millisekunden im Mittel und etwa 7,38 Millisekunden Median. Eine Erklärung für den großen Unterschied zwischen Single- und Mehr-Zonen-Varianten folgt anhand von Abbildung 17.

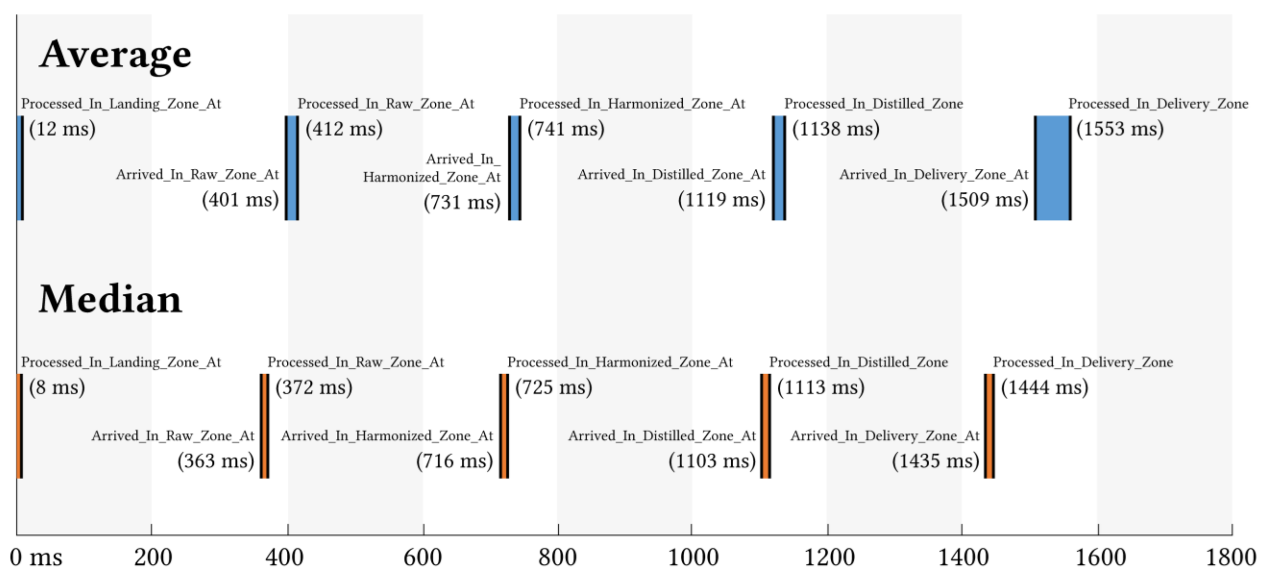


Abbildung 17: Mittelwerte und Mediane der Verarbeitungszeiten in Variante 1 aufgeschlüsselt nach einzelnen Messpunkten

Abbildung 17 zeigt die Mittelwerte und Mediane der Verarbeitungszeiten in der Zonenarchitektur mit Cassandra (Variante 1). Dabei sind die einzelnen Messpunkte als schwarze Linien dargestellt. Die gemessenen Zeiten ergeben sich aus den Medianen und Mittelwerten der Verarbeitungszeiten in und „zwischen" den einzelnen Zonenapplikationen.

Bei Betrachtung der Messwerte fällt auf, dass die eigentlichen Datenumformungen durch Apache Spark in den Datenprozessoren kaum Zeit in Anspruch nehmen (in der Abbildung blau bzw. orange hinterlegt). Je nach Zone liegt der Median der Verarbeitungszeit bei 8 bis 10 Millisekunden und der Mittelwert bei 10 bis 44 Millisekunden.

Die Umwandlung in einen JSON-String, der Versand der Nachricht durch den Kafka-Producer sowie die Registrierung der neuen Nachricht im Kafka-Cluster und durch den Kafka Consumer findet erst statt, nachdem der Datensatz den `Processed_In_<Zone>_At`-Zeitstempel erhalten hat. Diese Operationen nehmen scheinbar deutlich mehr Zeit in Anspruch (zwischen 300 und 400 Millisekunden) als die eigentliche Datentransformation und sind für etwa 90 Prozent der Gesamtverarbeitungszeit verantwortlich.

Durch diesen Umstand lässt sich erklären, warum die Single-Zone-Varianten (Variante 3 und 4) eine deutlich geringere Verarbeitungszeit aufweisen als die Mehr-Zonen-Architektur (Variante 1 und 2; siehe Abbildung 16). In der Single-Zone-Variante werden die Daten während der Verarbeitung nicht mehrmals an das Kafka-Cluster versendet. Die Zeiten, die für den Versand an das Cluster und die Auslese der Kafka-Nachrichten durch den Consumer in Anspruch genommen werden, sind in den Gesamtverarbeitungszeiten der Single-Zone-Anwendung also nicht enthalten, da in der Single Zone lediglich zwei Messpunkte vor der Datentransformation und vor dem Versand der verarbeiteten Daten an das Kafka-Cluster existieren (siehe Abbildung 15). Vernachlässigt man die Zeiten, die durch die Umwandlung in und den Versand von Kafka-Nachrichten zustande kommen, ist die reine Datenverarbeitung in der Single Zone dennoch deutlich performanter.

In den Single-Zone-Varianten wird pro Datenquelle lediglich ein Datenprozessor durchlaufen. Dieser Datenprozessor führt zwar komplexere Datentransformationen durch als die Datenprozessoren in der Mehr-Zonen-Architektur, allerdings verfügt die Single-Zone-Anwendung über mehr Ressourcen, da kein paralleler Betrieb mehrerer Spark-Applikationen auf demselben Rechner stattfindet. Dadurch lässt sich erklären, dass der komplexere Datenprozessor der Single Zone die Datensätze trotzdem schneller verarbeitet als die weniger komplexen Datenprozessoren in Variante 1 und 2.

In Abbildung 17 sind außerdem die großen Unterschiede zwischen Mittelwert (blau) und Median (orange) der Verarbeitungszeiten in Distilled und Delivery Zone auffällig. In der Distilled und der Delivery Zone werden in den Datenprozessoren mehrere Datenabfragen aus Cassandra getätigt. Einzelne Datensätze benötigen in der Distilled und Delivery Zone wesentlich länger für die Verarbeitung. Dies lässt sich darauf zurückführen, dass die Menge der simultanen Datenabfragen und Persistierungsoperationen an Cassandra zu manchen Zeitpunkten besonders hoch ist, zum Beispiel während der zeitgesteuerten Aggregation der Pandemiedaten. Die Cassandra-Komponente ist dann voll ausgelastet und Datenabfragen werden erst mit Verzögerung zurückgeliefert. Die Datenverarbeitung einzelner Datensätze dauert deshalb dementsprechend länger.

In der Single-Zone-Variante mit Cassandra (Variante 3) stehen der Datenbank-Komponente mehr Ressourcen zur Verfügung, weshalb Cassandra in dieser Architektur weniger häufig voll ausgelastet ist. Es sind äußerst wenige Datensätze vorhanden, die eine deutlich höhere Gesamtverarbeitungszeit benötigen: Bei nur 0,5 Prozent der Datensätze lag die Verarbeitungszeit über 50 Millisekunden, lediglich 0,16 Prozent der Datensätze benötigten für die Verarbeitung länger als eine Sekunde. Daher liegt der Mittelwert gegenüber dem Median nur geringfügig höher (Mittelwert 7 Millisekunden; Median 9,38 Millisekunden; siehe Abbildung 16).

Zusammenfassend lässt sich sagen, dass alle Varianten sich bezüglich ihrer Verarbeitungszeit im Echtzeit-Bereich bewegen. Dennoch verringert die Zonenarchitektur in der entwickelten Form die Performanz der Datenverarbeitung gegenüber der Single Zone deutlich. Etwa 1,5 Sekunden in der Zonenarchitektur stehen 0,007 Sekunden in der Single-Zone-Architektur gegenüber. Die Verarbeitungszeit in Variante 1 fällt im Median also um den Faktor 200 höher aus als die Gesamtverarbeitungszeit in Variante 3.

Bei besonders zeitkritischen Anwendungsfällen muss die Performanz einer solchen Zonenarchitektur also deutlich gesteigert werden. Es ist zu überprüfen, ob die Zeit, bis eine neue Nachricht vom Kafka-Cluster registriert und von der folgenden Zone verarbeitet werden kann, durch spezielle Konfigurationen von Apache Kafka und des SparkContext verringert werden kann.

Allerdings ist in den Verarbeitungszeiten aller Varianten die Zeit, die benötigt wird, um die final verarbeiteten Daten aus der Delivery Zone (bzw. Single Zone) an die entsprechende Kafka-Topic zu versenden, nicht einberechnet. Die verarbeiteten Daten können also auch unter Verwendung der Single-Zone-Anwendung nicht bereits nach 7 Millisekunden abgegriffen und durch externe Systeme weiterverwendet werden, sondern erst etwa 300 bis 400 Millisekunden später.

Werden also die Zeiten von der Ankunft der Rohdaten bis zur Verwendungsfähigkeit der vollständig verarbeiteten Daten verglichen, erreicht man mit der Mehr-Zonen-Architektur im Mittel Zeiten von etwa 1,8 bis 2 Sekunden gegenüber mittleren Zeiten von 300 bis 400 Millisekunden mit der Single-Zone-Architektur. Damit ist die Verarbeitungszeit der Mehr-Zonen-Architektur ungefähr um den Faktor 5 höher als die der Single Zone. Dies entspricht auch der Anzahl der Anwendungen, die in den jeweiligen Architekturen in Betrieb sind: Während in der Mehr-Zonen-Architektur fünf Anwendungen parallel ausgeführt und von allen Datensätzen durchlaufen werden müssen, geschieht die Datenverarbeitung in der Single-Zone-Architektur mithilfe einer einzigen Applikation.

Bei der Entscheidung für oder gegen eine Zonenarchitektur im Datenstrom-Kontext muss also abgewogen werden, ob der jeweilige Anwendungsfall eine äußerst schnelle Verarbeitung in wenigen Millisekunden erfordert oder ob die Zwischenergebnisse der Datenverarbeitung für zukünftige Anwendungszwecke wertvoll sind. In vielen zeitunkritischen Anwendungsfällen kann eine Verarbeitungszeit von zwei Sekunden akzeptabel sein, wenn die Daten im Gegenzug in ihren verschiedenen Verarbeitungsstufen wiederverwendet werden können. Werden die Daten zu einem späteren Zeitpunkt für einen neuen Anwendungsfall benötigt, muss keine ressourcenintensive erneute Verarbeitung aller Datensätze durchgeführt werden.

8 Zusammenfassung und Ausblick

Obwohl der Data-Lake-Begriff bereits seit Anfang der 2010er-Jahre bekannt ist und das Konzept heute bereits häufig bei der Erfassung, Speicherung, Verarbeitung und Analyse großer Datenmengen angewandt wird, mangelt es in der wissenschaftlichen Literatur an Vorgehensweisen bei der Erstellung einer solchen Plattform, an Referenzarchitekturen und an Beschreibungen von Implementierungen in realistischen Anwendungsszenarien.

Diese Arbeit leistet einen Beitrag zur Schließung dieser Lücken. Sie umfasst die Konzeption, den Entwurf, die prototypische Implementierung sowie die Evaluation einer zonenbasierten Data-Lake-Architektur. Dabei wird eine Referenzarchitektur – das Zonenreferenzmodell von Giebler, Gröger et al. [Gie2020b] – im Kontext von Datenströmen als Übertragungsmittel angewandt.

Hierfür wurden zunächst die wissenschaftlichen Grundlagen zusammengetragen, welche zur Umsetzung einer solchen Architektur erforderlich sind, und verschiedene Architekturen zonenbasierter Data Lakes sowie Konzepte zur Datenstromverarbeitung aus verwandten wissenschaftlichen Arbeiten verglichen und diskutiert. Dabei wurde deutlich, dass sämtliche betrachtete Arbeiten sich entweder mit zonenbasierten Data-Lake-Architekturen oder aber mit Konzepten zur Datenstromverarbeitung im Data Lake befassen. Allerdings wurden beide Aspekte in keiner dieser Arbeiten vereint. Es mangelt also an der Beschreibung einer zonenbasierten Data-Lake-Architektur, die Datenströme als Übertragungsmittel verwendet.

Aus diesem Grund wurde eine Data-Lake-Architektur auf Grundlage des Zonenreferenzmodells entwickelt, in welcher Daten in Form von Datenströmen verarbeitet werden. Als Ausgangspunkt für die zu entwickelnde Data-Lake-Plattform diente ein fiktiver Anwendungsfall. In diesem Szenario sollten während der Coronavirus-Pandemie Daten zur Virus-Ausbreitung aus verschiedenen Teilen der Welt gesammelt, verarbeitet und – in Kombination mit standortbezogenen Wetterdaten – explorativ ausgewertet werden. Dazu erhielt die Data-Lake-Plattform Datensätze aus drei Datenstromquellen, welche durch die Datenprozessoren transformiert, aggregiert und kombiniert wurden.

Zur Umsetzung der Plattform wurde zunächst eine generische Architektur spezifiziert. Eine solche Datenstrom-Architektur weist einige Charakteristika auf. Da die Daten in einer reinen Streaming-Architektur zunächst nicht persistiert werden, müssen Schnittstellen existieren, an denen die Daten in ihrer Datenstrom-Form bezogen werden können. Aufgrund großer Datenmengen, die von der Plattform verarbeitet werden sollen, und der dadurch notwendigen Skalierbarkeit bietet sich eine Microservice-Architektur an, in welcher einzelne Komponenten – wie beispielsweise die Zonenapplikationen – zur Erhöhung des Durchsatzes dupliziert werden können. Zudem ist ein einheitliches Übertragungsformat für strukturierte und semi-strukturierte Daten bereits ab der initialen Datenaufnahme von Vorteil. Die Plattform kann je nach Anforderungen des Anwendungsszenarios entweder in Form einer reinen Streaming-Architektur oder als Lambda- oder Kappa-Architektur realisiert werden. Da in realen Anwendungsfällen nahezu immer eine Persistierung der Daten erfolgen muss, ist eine Lambda- oder Kappa-Architektur in der Praxis weitaus häufiger.

Aufbauend auf der generischen Architektur und einer Spezifikation sämtlicher Datenverarbeitungsschritte wurde schließlich eine Implementierung der Data-Lake-Plattform umgesetzt. Dabei entstand eine Microservice-Architektur mit je einer „Zonenapplikation“ pro Data-Lake-Zone und unter Verwendung der Technologien *Apache Kafka*, *Apache Spark* und *Apache Cassandra*.

Obwohl zur Realisierung der Aggregationen und Kombinationen Daten in der Cassandra-Datenbank persistiert und abgefragt werden, wurde aufgrund des vereinfachten Anwendungsfalles auf die Entwicklung einer vollständigen Lambda- oder Kappa-Architektur verzichtet. Allerdings wurden Erweiterungsmöglichkeiten diskutiert, mithilfe derer eine solche Architektur umgesetzt werden könnte (siehe Kapitel 5.2 und 6.2.2).

Die entstandene Plattform wurde schließlich hinsichtlich ihrer Architektur und Implementierung bewertet. Als zentrale Vorteile dieser Architekturform stellten sich die hohe Wiederverwendbarkeit der rohen und vorverarbeiteten Daten in späteren Anwendungsfällen sowie die gute Skalierbarkeit ihrer Komponenten heraus. Hinsichtlich der prototypischen Implementierung wurden einige Schwachstellen und Verbesserungspotentiale identifiziert (siehe Kapitel 7.1).

Außerdem erfolgte im Zuge der Bewertung eine Evaluation der Echtzeitfähigkeit der entstandenen Plattform in verschiedenen Konfigurationsvarianten. Dabei wurde ersichtlich, dass sich die Datengesamtverarbeitungszeit mit der vorliegenden Data-Lake-Plattform zwar im Echtzeitbereich bewegt, die Implementierung in der entwickelten Form allerdings mit Gesamtverarbeitungszeiten von durchschnittlich bis zu 2 Sekunden für äußerst zeitkritische Anwendungsfälle ungeeignet ist.

Aus der Evaluation lassen sich unter anderem folgende Forschungsfragen ableiten, die Gegenstand weiterführender Arbeiten sein könnten:

- Welche Technologien eignen sich am besten zur Steigerung der Performanz einer Datenstromverarbeitung im zonenbasierten Data Lake? Lässt sich beispielsweise mit *Apache Flink* eine schnellere Datenverarbeitung erzielen?
- Wie lässt sich die Datenverarbeitung in einer zonenbasierten Data-Lake-Plattform weiter generalisieren, sodass sich zum einen die Konfiguration der Datenprozessoren vereinfacht, zum anderen die Anzahl der Code- und Paketredundanzen reduziert?
- Wie würde die Implementierung einer zonenbasierten Data-Lake-Plattform unter konsequenter Anwendung des Lambda- oder Kappa-Architekturpatterns aussehen?
- Welche Alternativen gibt es zu *Apache Cassandra* als performantes, verteiltes und skalierbares Master Dataset in einer Lambda- oder Kappa-Architektur?

Da Data Lakes gegenwärtig in der Praxis immer häufiger zum Einsatz kommen, ist zu erwarten, dass die Wissenschaft sich dieser Thematik in den kommenden Jahren verstärkt widmen wird. Die im Rahmen dieser Arbeit beschriebene Architektur ist dabei als Machbarkeitsnachweis (*Proof of Concept*) zu verstehen. Auch wenn die Implementierung an mehreren Stellen Verbesserungspotenziale aufweist, so wurde deutlich, dass zonenbasierte Data Lakes grundsätzlich auch bei der Streaming-Datenverarbeitung angewandt werden können.

Die zentralen Gründe für die Einführung einer Data-Lake-Plattformen sind die Wiederverwendbarkeit der Rohdaten und die damit einhergehende Flexibilität in der Datenanalyse. Da von Unternehmen und Organisationen in der Praxis mehr und mehr Daten erhoben und gespeichert werden, ist anzunehmen, dass Data Lakes und insbesondere auch zonenbasierte Data Lakes in Zukunft an Bedeutung gewinnen werden.

9 Literaturverzeichnis

[Apa2020a] APACHE SOFTWARE FOUNDATION (2020), “Apache Cassandra Documentation”, cassandra.apache.org/doc/latest/, zuletzt geprüft am 12.10.2020.

[Apa2020b] APACHE SOFTWARE FOUNDATION (2020), “Kafka 2.0 Documentation”, kafka.apache.org/20/documentation.html, zuletzt geprüft am 19.09.2020.

[Apa2020c] APACHE SOFTWARE FOUNDATION (2020), “Spark 3.0.1 Documentation”, spark.apache.org/docs/latest/, zuletzt geprüft am 20.09.2020.

[Bal2017] BALLHORN C. (2017), “Data Warehouse vs. Data Lake – das sind die Unterschiede”, www.silicon.de/blog/data-warehouse-vs-data-lake-das-sind-die-unterschiede, zuletzt geprüft am 18.04.2020.

[Ber2017] BERLE L. (2017), “Lambda vs. Kappa: Streamingarchitekturen in der Praxis”, *BT Magazin*, 04/2017, www.opitz-consulting.com/fileadmin/user_upload/Collaterals/Artikel/business-technology-2017-04-lambda-vs-kappa-streamingarchitekturen-in-der-praxis_berle_sicher.pdf, zuletzt geprüft am 19.04.2020.

[Ber2018] BERLE L. (06.11.2018), "Big-Data-Architekturen und zehn typische Stolpersteine auf dem Weg dahin", Vortrag auf der W-JAX Conference 2018, www.youtube.com/watch?v=FyCu6tmPQq8, zuletzt geprüft am 04.05.2020.

[Ble2020] BLEIKER C. (08.04.2020), “Johns Hopkins University: Die Corona-Experten”, *Deutsche Welle*, www.dw.com/de/johns-hopkins-university-die-corona-experten/a-53068103, zuletzt geprüft am 05.11.2020.

[Dat2019] DATENBANK LEXIKON (2019), “Data Lake: Definition & Erklärung”, www.datenbanken-verstehen.de/lexikon/data-lake/, zuletzt geprüft am 18.04.2020.

[Dim2016] DIMITROV D. V. (2016), “Medical internet of things and big data in healthcare”, *Healthcare informatics research*, vol. 22 no. 3, p. 156-163.

[Dul2015] DULL T. (2015), “Data Lake vs Data Warehouse: Key Differences”, www.kdnuggets.com/2015/09/data-lake-vs-data-warehouse-key-differences.html, zuletzt geprüft am 21.04.2020.

[Fis2019] FISKE A., BUYX A., PRAINSACK B. (2019), “Health information counselors: a new profession for the age of big data”, *Academic Medicine*, vol. 94 no. 1, p. 37.

[Gar2020] GARDNER L. (2020), “Mapping 2019-nCoV”", Baltimore, USA, systems.jhu.edu/research/public-health/ncov/, zuletzt geprüft am 14.04.2020.

[Gie2020a] GIEBLER C., GRÖGER C., HOOS E., EICHLER R., SCHWARZ H., MITSCHANG B. (2020), “Data Lakes auf den Grund gegangen”, *Datenbank-Spektrum*, p. 1-13.

[Gie2020b] GIEBLER C., GRÖGER C., HOOS E., SCHWARZ H., MITSCHANG B. (2020), “A Zone Reference Model for Enterprise-Grade Data Lake Management”, *Proceedings of the 24th Enterprise Computing Conference (EDOC 2020).*

[Hai2016] HAI R., GEISLER S., QUIX C. (2016), “Constance: An intelligent data lake system”, *Proceedings of the 2016 International Conference on Management of Data*, p. 2097-2100.

[Inm2016] INMON B. (2016), "Data Lake Architecture: Designing the Data Lake and avoiding the garbage dump", Technics publications.

[Kar2018] KARIMOV J., RABL T., KATSIFODIMOS A., SAMAREV R., HEISKANEN H., MARKL V. (2018), “Benchmarking distributed stream data processing systems”, p. 1507-1518.

[Kre2014] KREPS J. (2014), “Questioning the lambda architecture”, www.oreilly.com/radar/questioning-the-lambda-architecture/, zuletzt geprüft am 05.11.2020.

[Mad2015] MADSEN M. (2015), “How to Build an enterprise data lake: important considerations before jumping in”, *Third Nature Inc.*

[Man2018] MANOGARAN G., VIJAYAKUMAR V., VARATHARAJAN R., KUMAR P. M., SUNDARASEKAR R., HSU C.-H. (2018), “Machine learning based big data processing framework for cancer diagnosis using hidden Markov model and GM clustering”, *Wireless personal communications*, vol. 102 no. 3, p. 2099-2116.
[Mar2015] MARZ N., WARREN J. (2015), "Big Data: Principles and best practices of scalable realtime data systems", Manning Publications Co.
[Mar2019] MARSCHALL T., HENNING B. (2019), “Pi-Architektur: Agiles Datenmanagement in Big-Data-Umgebungen”, *BI-SPEKTRUM*, no. 2, p. 36-40.
[Mei2019] MEIR-HUBER M. (2019), “Kappa-Architecture isn't solving all our problems for Data Handling”, cloudvane.net/2019/04/09/kappa-architecture-isnt-solving-all-our-problems/, zuletzt geprüft am 27.04.2020.
[Mun2018] MUNSHI A. A., MOHAMED Y. A.-R. I. (2018), “Data lake lambda architecture for smart grids big data analytics”, *IEEE Access*, vol. 6, p. 40463-40471.
[Pyn2015] PYNE S., VULLIKANTI A. K. S., MARATHE M. V. (2015), 2015, “Big data applications in health sciences and epidemiology” in: *Handbook of statistics*, Elsevier, p. 171-202.
[Rob2020] ROBERT-KOCH-INSTITUT (10.04.2020), “SARS-CoV-2 Steckbrief zur Coronavirus-Krankheit-2019 (COVID-19)”, www.rki.de/DE/Content/InfAZ/N/Neuartiges_Coronavirus/Steckbrief.html, zuletzt geprüft am 14.04.2020.
[Yua2015] YUAN W., DENG P., TALEB T., WAN J., BI C. (2015), “An unlicensed taxi identification model based on big data analysis”, *IEEE Transactions on Intelligent Transportation Systems*, vol. 17 no. 6, p. 1703-1713.
[Zal2018] ZALONI, INC. (2018), “The Data Lake Reference Architecture: Leveraging a Data Reference Architecture to Ensure Data Lake Success”", www.zaloni.com, resources.zaloni.com/i/913381-the-data-lake-reference-architecture/5?, zuletzt geprüft am 17.03.2020.
[Zik2015] ZIKOPOULOS P. (2015), "Big data beyond the hype: A guide to conversations for today’s data center", McGraw-Hill Education.

Erklärung

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

____

Ort, Datum, Unterschrift