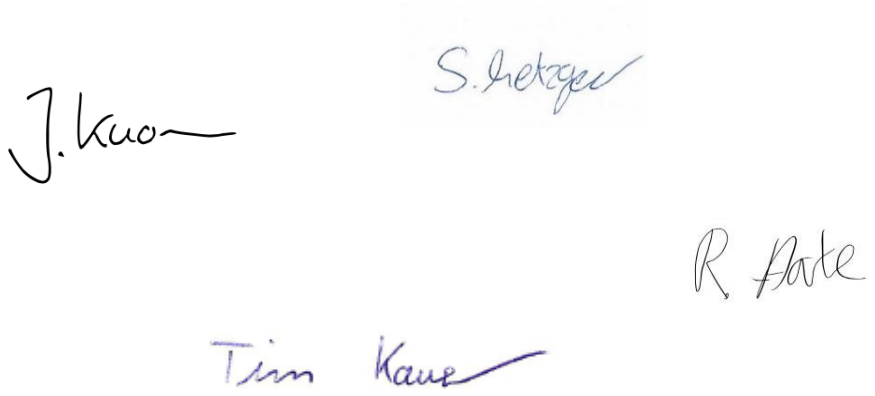
LAMBDA STREAMING ARCHITEKTUR PROJEKTDOKU

Analyse der geographischen Herkunft von Corona Tweets

Eigenständigkeitserklärung

Hiermit versichern wir, dass wir die vorliegende Arbeit selbständig angefertigt und keine anderen als die im Text genannten Hilfsmittel verwendet haben.



(Unterschriften)	**(Datum)**
Jannik Kuom, Tim Kauer, Sven Metzger, Ruben Harle	*13.09.2020*

Inhaltsverzeichnis

1. Motivation

Streaming Daten finden immer häufiger Anwendung im alltäglichen Leben, besonders bei Plattformen wie beispielsweise *Netflix* und *YouTube*. Bei steigender Frequentierung und somit steigender Datenbelastung müssen hierfür stabile und möglichst effiziente Architekturen geschaffen werden. In Fachkreisen wird bei solch großen Datenmengen auch von Big Data gesprochen. An dieser Thematik setzt diese Arbeit an und beschäftigt sich umfassend mit dem Aufbau einer funktionsfähigen und leistungsstarken Lambda Architektur, welche im Anwendungsfall Corona Tweets analysieren kann. Hierbei können beispielsweise Orte bzw. Länder nach Anzahl abgesendeter Tweets sortiert und aufgearbeitet werden. Ziel dieser Arbeit ist das Sammeln von tiefgehenden Erfahrungen im Zusammenhang mit Streaming-Architekturen und zugleich das sinnvolle Verwenden der gesammelten Daten. Die Konferenz [3] diente als wissenswerter Informationsleitfaden im Bereich der Effizienz.

2. Lambda Architektur

Die Lambda Architektur wurde entwickelt, um eine erhöhte Skalierbarkeit bei der Datenverarbeitung im Big Data Umfeld zu gewährleisten. Dabei spielt auch Fehlertoleranz bei Hardware- und menschlichen Fehlern eine essenzielle Rolle. Das Gesamtsystem sollte linear skalierbar sein und Lesevorgänge mit geringer Latenz behandeln [1]. Insgesamt ermöglicht die Architektur einen hybriden Ansatz mit Stapel- und Stream-Verarbeitungsmethoden [2]. Die Lambda Architektur besteht grundsätzlich aus den folgenden drei Layern: Batch Layer, Speed Layer und Serving Layer.

2.1. Batch Layer

Der Batch Layer ist eine routinierte Komponente der Architektur, d.h. er arbeitet nach vordefinierten Zeitscheiben und verarbeitet die Daten als Gesamtpaket. Damit ermöglicht er eine sorgfältige Verwaltung des Stammdatensatzes. Es gilt jedoch allgemein zu berücksichtigen, dass die Verarbeitungs- und Berechnungszeit mit zunehmender Datenmenge steigt.

2.2. Speed Layer

Der Speed Layer wird auch als Streaming Layer bezeichnet, da er Echtzeit Ansichten der aktuellsten Daten berechnet, um diese wiederum externen Systemen zur Verfügung zu stellen. Im Gegensatz zum Batch Layer ist hier die Genauigkeit und Vollständigkeit eher zweitrangig. Ziel ist es, die langwierige Berechnungszeit des Batch Layers mit einer vorläufigen Datensicht auszugleichen, welche im Nachgang mit den abgeschlossenen Berechnungen des Batch Layer ergänzt wird.

2.3. Serving Layer

Der Serving Layer wird zur Datenspeicherung und Datenbereitstellung verwendet. Ergebnisse des Speed- und Batch Layers werden hier abgelegt, wodurch Ad-hoc Anfragen auf vorberechnete Datensichten ermöglicht werden können.

2.4. Praxisbezogene Systemarchitektur

Systemarchitektur: BigDataArchitecture.pdf

Der Data Ingestion Layer ist allgemein zum einen für die Ablage der Daten zuständig, zum anderen wird dieser aber auch als Wiederherstellungskomponente verwendet. In diesem Projekt wird *Apache Kafka* verwendet, um die Kommunikation zwischen dem Ingestion Layer und dem Batch Layer, aber auch zwischen dem Ingestion Layer und Speed Layer zu garantieren. Die Wahl fiel auf Kafka, da Kafka aufgrund seiner hohen Nachrichten- Durchsatzrate bestens für das streamen von Tweets geeignet ist. Somit beinhaltet der Ingestion Layer einen Kafka Producer. Wenn somit Daten durch Twitter den Ingestion Layer erreichen und dort verarbeitet werden, werden die Daten an den Speed Layer als auch den Batch Layer weitergegeben. Der Speed Layer verarbeitet die Daten sofort unmittelbar nach der Weitergabe vom Ingestion Layer. Zudem enthält er einen Kafka Consumer, um Daten zu empfangen und eine *Apache Spark* Komponente, welche die Daten nachfolgend verarbeitet. Durch Apache Spark können große Datenmengen performant und parallel verarbeitet werden, weswegen hierbei Apache Spark zum Einsatz kommt. Der Batch Layer nimmt die Daten zeitversetzt in

bestimmten Intervallen ab. Dieser enthält ebenfalls einen Kafka Consumer. Im Batch Layer werden die Daten durch SQL Batch Processing an eine SQL Datenbank übermittelt. Als SQL Datenbank fällt die Wahl auf *Postgresql,* da dieses bereits in einem anderen Projekt zum Einsatz kam und somit Vorerfahrungen die Implementierung erleichtern. Der Serving Layer übernimmt die Rolle des Datenlieferanten. Somit ist er wie der Batch Layer mit der postgresql Datebank verbunden. Nachfolgend ist der Serving Layer mit einem *Flask* Webserver verbunden. Hier fällt die Wahl auf Flask, da man durch Flask schnell Webanwendungen unter Verwendung einer einzigen Python Datei erstellen kann. Außerdem ist Flask erweiterbar und setzt keine bestimmte Verzeichnisstruktur voraus. Desweiteren wird ein Cache Server eingesetzt, welcher durch die Erweiterung von Flask (Flask-Caching) eingebunden wird. Der Cache Server ist wichtig in der Architektur zu berücksichtigen bzw. zu implementieren, da dadurch die Daten im Speicher gehalten werden, um ein Abrufen zu ermöglichen, ohne das die Daten von der ursprünglichen Quelle angefordert werden müssen, wenn sich diese Daten nicht häufig ändern. Zum Schluss wird zusätzlich ein Load Balancer, hier *NGINX*, verwendet, um die Ressourcennutzung zu optimieren, sowie den Durchsatz zu maximieren. Mit NGINX wird der Datenverkehr auf mehrere Anwendungsserver verteilt und die Leistung bzw. die Skalierbarkeit von Webanwendungen, wie unser Flask Webserver verbessert.

3. Fazit

Das Projekt hat uns in vieler Hinsicht vor zeitintensive Herausforderungen gestellt, welche wir aber im Gesamtheitlichen sehr gut meistern konnten. Die Integration von Technologien wie Spark im Zusammenspiel mit Kafka, Docker und Postgresql waren zum Teil neu für uns, weswegen wir einiges an wertvollen Erfahrungen mitnehmen konnten. Besonders herausfordernd war die Implementierung des Speed Layers, bei dem wir uns von Fehlermeldung zu Fehlermeldung kämpften. Abschließend sollte aber im Bezug darauf betont werden, dass gerade diese Fehlermeldungen die Lernkurve im Positiven beeinflusst haben. Wir stufen unser Projektziel somit als erfolgreich abgeschlossen ein.

Literaturverzeichnis

[1] Lambda Architecture, Michael Hausenblas & Nathan Bijnens, http://lambda-architecture.net/ abegrufen am 05.09.2020

[2] Lambda Architecture, Databricks, https://databricks.com/glossary/lambda-architecture abgerufen am 07.09.2020

[3] M. Kiran, P. Murphy, I. Monga, J. Dugan and S. S. Baveja, "Lambda architecture for cost-effective batch and speed big data processing," 2015 IEEE International Conference on Big Data (Big Data), Santa Clara, CA, 2015, pp. 2785-2792, doi: 10.1109/BigData.2015.7364082.