Data Engineering mit Apache Kafka

Projektbericht

von

Johannes Weber

Matrikelnummer: 11010021

und

Julian Ruppel

Matrikelnummer: 11010020

09.02.2018

SRH Heidelberg

Fakultät für Information, Medien und Design

Big Data und Business Analytics

Dozent

Frank Schulz

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Aufgabe und Ziel

Im Rahmen dieses Projektes war es die Zielstellung sich mit Data Ingestion, Data Storage sowie Data Retrieval vertraut zu machen.

Data Ingestion ist die Beschaffung der Daten. Dies kann entweder mit Hilfe eines Data Streams erfolgen oder einer statischen Datenquelle - also eine einfache Datei die lokal auf einem Rechner angelegt wird und Daten beinhät wie z. B. eine Comma-separated values (CSV) oderJavaScript Object Notation (JSON) Datei. Unter einem Data Stream versteht mann einen kontinuierlichen Datenstrom wie z. B. die Erstellung von immer wieder neuen Twitter Nachrichten. Ein wichtiges Merkmal eines Data Streams ist, dass man nicht vorhersehen kann wann der Datenstom zu Ende ist - er könnte theoretisch unendlich sein. Im Falle von eines Datenstroms von Twitter Nachrichten ist es nicht abzusehen wann jemals die letzte Twitter Nachricht geschriebenb wird. Für unsere Aufgabe ist darauf zu achten, dass der Datenstrom über eine API öffentlich zugänglich ist und immer auf dem aktuellsten Stand gehalten wird.

Data Storage ist die Speicherung der Daten. Hierbei wurde uns lediglich die Anforderung gestellt, das wir für die Speicherung die Streaming Plattform Apache Kafka verwenden. Des Weiteren war es uns gestattet die Daten in einer relationen Datenbank, NoSQL Datenbank oder mit Spark Streaming speichern, falls nur die Nutzung von Apache Kafka unsere Anforderungen nicht genügt.

Data Retrieval ist die Beschaffung der Daten aus einer Datenbank mit SQL Abfragen und die abschließende Ausgabe der Ergebnisse in Form von Tabellen oder einfachen Visualisierungen. Es sollten mindestens drei verschieden SQL Abfragen abgesetzt werden mit unterschiedlichen Filter- und Agreggatsfunktionen sowie einer Teilaggregation wie z. B. GROUP BY. Die Visualisierung der Daten sollte in einem virtuellen Notebook erfolgen.

Als virtuelles Notebook durften wir uns entscheiden zwischen Apache Zeppelin oder Jupyter.

Unsere Aufgabe ist es eine geeignete Vorgehensweise für die Bewältigung dieser Aufgabe zu finden und umzusetzen.

Wir entschieden uns für unser Szenario Daten von NYC Open Data zu nutzen. NYC Open Data ermöglicht es allen "New Yorkernünd somit auch der ganzen Welt sog. Open Data also frei zugängliche Daten einfach zu konsumieren.[1].

NYC Open Data ermöglicht es uns sowohl einen kontinuierlichen Data Stream als auch eine statische CSV Datei zu konsumieren. Dank diesem Umstand entschieden wir uns im Rahmen dieses Projektes beide Möglichkeiten umzusetzen und zu vergleichen. Auch bei der Data Retrieval entschieden wir uns dafür sowohl Apache Zeppelin als auch Jupyter zu nutzen und zu vergleichen.

Kapitel 2 - *Werkzeuge und technische Rahmenbedingungen* beschäftigt sich detailierter mit den verwendeten Tools, Frameworks und Programmiersprachen und Kapitel 3 - *Lösungsansatz* erläutert das gewählte Szenario mit den Unterkapitel Data Ingestion, Data Storage und Data Retrieval sowie der verwendeten Architektur. [2]

[1] **?**

[2] Einleitung von Johannes Weber

Kapitel 2

Werkzeuge und technische Rahmenbedingungen

Im folgenden Abschnitt werden einige wichtige technische Werkzeuge, die im späteren Verlauf eingesetzt werden, kurz und prägnant erläutert.

Java ist eine objektorientierte Open-Source Programmiersprache die ursprünglich von Sun Microsystems entwickelt wurde und heute zum Oracle Konzern gehört. In den letzten beiden Hauptversionen wurde der Sprachumfang um funktionale und reaktive Aspekte erweitert. Java ist dank der *Java Virtual Maschine* (JVM) als Laufzeitumgebung plattformunabhängig und hat sich vorwiegend in Enterprise Systemen und Web-Backends etabliert. Im Zuge der Verbreitung von BigData Projekten unter dem Dach der Apache Software Foundation, allem voran Hadoop und Spark, werden JVM sprachen wie Java und Scala nun auch im Bereich BigData eingesetzt.

Python ist eine Open-Source Skripsprache, die sich hauptsächlich durch eine gut lesbare und knappe Syntax auszeichnet und unter anderem das objektorientierte und funktionale Programmierparadigma unterstützt. Im Gegensatz zu Java ist Python dynamisch typisiert und wird interpretiert anstatt kompiliert. Dank eines sehr umfangreichen und ausgereiften Ökosystems aus Frameworks und Bibliotheken zur Datenanalyse und maschinelles Lernen[1] ist Python im Bereich BigData und dank der minimalinvasiven Eigenschaften zum Rapid Prototyping beliebt.

Apache Kafka ist eine verteilte Data-Streaming Plattform der Apache Software Foundation, die ursprünglich von LinkedIn entworfen wurde. Beliebt ist Kafka im BigData Umfeld wegen seiner Skalierbarkeit und Fehlertoleranz. Zu den Einsatzszenarien zählen vor allem Stream Processing, es kann aber auch als reiner Message Broker oder Speichersystem für Streaming Data verwendet werden. Die wesentlichen Komponenten von Kafka sind ***Producer*** um einen Stream für einen ***Topic*** zu veröffentlichen, ***Kafka Cluster*** um die die Streaming-Daten verteilt pro *Topic* im Dateisystem zu speichern und ***Consumer*** um einen *Topic* zu abonnieren und dessen Nachrichten zu le-

[1] z.B. TensorFlow von Google

sen. Zudem können mit ***Kafka Streams*** Nachrichten im Cluster transformiert werden. Mittels ***Kafka Connectors*** kann man per Konfiguration gängige Datenquellen und -senken[2] anschließen und stellen somit eine deklarative alternative zu den imperativen *Producer API* und *Consumer API* dar. 2014 haben sich die verantwortlichen LinkedIn Mitarbeiter vom Mutterkonzern getrennt um sich mit der neu gegründeten Firma Confluent dediziert dem Apache Kafka Ökosystem zu widmen. Entwickelt wurde die quelloffene Software in der JVM-basierten Programmiersprache Scala, welche objektorientierte und funktionale Aspekte vereint.

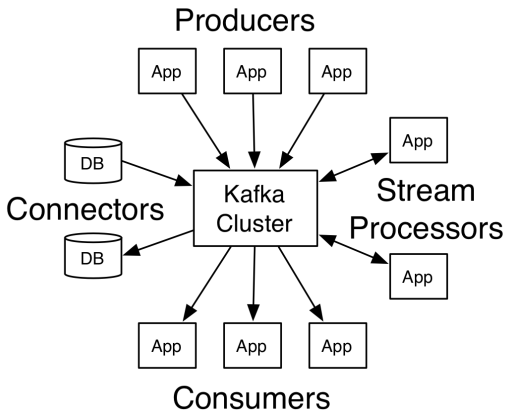

Abbildung 2.1: Apache Kafka Architektur[**?**]

PostgreSQL ist ein objektrelationales Datenbankmanagementsystem. Das vollständig ACID-konforme und in C geschriebene quelloffene System zeichnet sich durch einen breiten Funktionsumfang, Stabilität, Standardkonformität, hohe Erweiterbarkeit, und als Resultat dessen, eine weite Verbreitung aus. Neben dem traditionellen zeilenorientierten Eigenschaften bietet PostgreSQL zudem Erweiterungen hinsichtlich verteilter, hoch-parallelisierter und spaltenorientierter Datenverarbeitung, ein Geoinformationssystem sowie Volltextsuche. Auch im Bereich NoSQL bietet PostgreSQL eine dokumentenorientierter Speicherung und durch Erweiterungen sogar Graphen und Schlüssel-Werte-Datenstrukturen. Diese Flexibilität eröffnet PostgreSQL vielseitige Einsatzszenarien, darunter sowohl OLTP als auch OLAP.

Apache Zeppelin ist eine Web-basierte Open-Source Software mit der man sog. Notebooks zur datengetriebenen, interaktiven und kollaborativen Analyse erstellen kann. Es werden eine Vielzahl an Speicher- und Analysetechnologien unterstützt, darunter SQL, Scala, Spark, Python und R. Im wesentlichen werden in einem Notebook Abfrageskripte ad-hoc gegen die diversen Datenquellen ausgeführt und deren Ergebnisse in einem konfigurierbaren Web-Dashboard visuell und interaktiv dargestellt. Dadurch

[2] wie z.B. Twitter oder JDBC

eignet es sich sowohl zur explorativen Datenanalyse als auch zum veröffentlichen und teilen von Analyseergebnissen.

Jupyter ähnelt in den meisten Aspekten Apache Zeppelin, sodass sich alle oben zu Zeppelin genannten Punkte auch zu Jupyter nennen lassen. Die Unterschiede liegen eher im Detail der einzelnen Funktionen sowie der historisch und organisatorisch bedingten Nähe zu bestimmten Schlüsseltechnologien. Für das hier behandelte Forschungsprojekt spielen die individuellen Stärken und Schwächen der beiden Werkzeuge jedoch keine Rolle, weshalb beide Werkzeuge ebenbürtig eingesetzt werden.

SODA ist eine quelloffene Open Data Web-Programmierschnittstelle des U.S. Amerikanischen Dienstleisters Socrata. Anhand URLs werden Datasets adressiert und mittels der an SQL angelehnten *Socrata Query Language* (SoQL) per HTTP GET in unterschiedlichen Datenformaten abgefragt. Zudem stehen SDKs für diverse Programmiersprachen zur Verfügung. Im Rahmen des Projektes werden wir die Datensätze der Application Programming Interface (API) im JSON Dateiformat abrufen.

Kapitel 3

Lösungsansatz

In diesem Kapitel wird ein Konzept und die Herangehensweise zur prototypischen Lösung der in Kapitel 1 - *Einleitung* genannten Problemstellung erörtert.

3.1 Architektur

Grundsätzlich lassen sich zwei unterschiedliche Architekturansätze im Bereich Data Streaming unterscheiden:

Lambda Architektur: Benannt nach dem griechischen (λ) zeichnet sich dieser Ansatz dadurch aus, dass die Streaming Daten zweigleisig verarbeitet werden. Zum einen wird der Datenstrom direkt in einen häufig auf In-Memory Technologie basierenden *Speed Layer* geleitet, der diese in Echtzeit verarbeitet und dem *Serving Layer* zur Verfügung stellt. Da Streams per Definition unendlich und der Speed Layer teuer und physikalisch endlich ist, wird der Stream von *Ingestion Layer* parallel in den *Batch Layer* geleitet. Dieser speichert zunächst die Daten persistent und startet nach einem fest definierten Intervall einen Batch-Job um die bis dahin angelaufenen Daten zu verarbeiten und zum Serving Layer zu übertragen. Es ist also die nicht zu unterschätzende Verantwortung des Serving Layers, die aggregierten Bestandsdaten aus dem Batch Layer mit den Echtzeitdaten des Speed Layers, die noch nicht vom Batch Layer verarbeitet worden sind, abzumischen.

Kappa Architektur:

3.2 Data Ingestion

Wie schon in Kapitel 1 - *Einleitung* erläutert haben wir uns dafür entschieden die NYC Open Data als Datenquelle zu nutzen. Genauer gesagt entschieden wir uns für den Datensatz mit

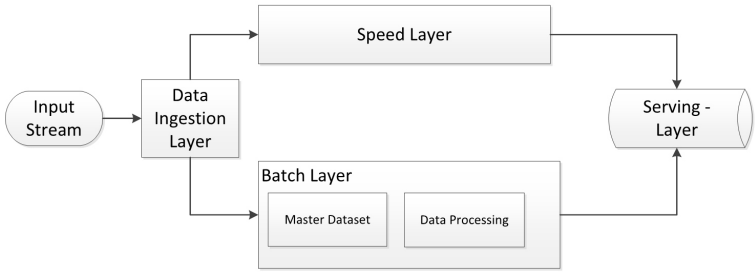

Abbildung 3.1: Schema der λ Architektur[**?**]

dem Kürzel *fhrw-4uyv*. Dieser Datensatz beinhält alle Service Request die seit 2011 von den Einwohner von New York City abgesetzt wurden sind. Beispielsweise kann mit diesem Datensatz herausgefunden werden wo welche Straßenlaternen in New Yor City ausgefallen sind oder in welchem Haus zu welcher Uhrzeit viel Party gemacht wurde da sich jemand über den Lärm beschwert hat.

In unserem Projekt beschaffen wir die Daten einmal über eine CSV Datei die man sich bei NYC Open Data runter laden kann und direkt über die SODA API die einen eigenen Endpunkt anbietet um Request für die verschiedenen Datensätze abzusetzen. Die Data Ingestion mit der CSV Datei setzte Julian Ruppel mit der Programmiersprache Java um und den kontunierlichen Datenstrom über die Socrata API wurde von Johannes Weber mit der Programmiersprache Python ausgelesen.

Beide Ansätze werden in diesem Kapitel beschrieben und miteinander verglichen.

3.2.1 Data Ingestion via CSV Datei

Da muss was stehen

3.2.2 Data Ingestion via SODA Schnittstelle

Wie in Abschnitt 3.1 - *Architektur* erläutert ist es die Aufgabe des Data Ingestion Prototyps zum einen Daten aus der externen Quelle auszulesen aber auch die Daten direkt an die Apache Kafka Plattform weiterzuleiten und in ein Topic zu speichern.

Die Umsetzung des "Producersërfolgte mit Python. Zusätzlich wurden folgende Frameworks benutzt um die Implementierung des *producers* zu unterstützen:

- sodapy
- kafka-python

Sobald der *producer* ausgeführt wird, wird der Nutzer gebeten ein Anfangs- und Enddatum anzugeben. In diesem Zeitfenster werden dann alle Datensätze abgefragt und in das Topic SServiceRequests"von Apache Kafka geschrieben.

Quellcode

Der *producer* besteht insgesamt aus zwei Python Skripten:

- SodaHelper.py
- producer.py

Das SodaHelper Skript ist ein separatzer Wrapper um die sodapy Bibliothek um die Verbindung zu der API herzustellen und die Daten zu holen. Somit wird eine klare Aufgabentrennung erreicht. Das Skript SodaHelper ist für die Verbindung zu der API zuständig und das producer Skript nur für die Weiterleitung der empfangenen Daten an Apache Kafka.

Da es sich hierbei nicht um ein Live Stream handelt wie z. B. bei Twitter API, haben wir uns dazu entschieden einen "Fake Streamßu erstellen, indem nicht alle Daten sofort an Apache Kafka weitergeleitet werden sondern immer ein gewisser Abstand zwischen dem Senden der einzelnen Datensätze erzwungen wird.

Unser ausgewählter Datensatz ist sehr groß. Wenn z. B. der Nutzer Daten von einem Monat abrufen will kann es vorkommen, dass in diesem Zeitraum mehr als 100.000 Datensätze bereit zum Abruf stehen. Da der Abruf von einer solch großen Menge an Daten über die API eine sehr hohe Rechenleistung erfordert und diese im Rahmen des Projektes nicht zur Verfügung steht wurde eine Art *Paging* entwickelt. Durch die feste Angabe eines Limits von 10.000 wird sichergestellt, dass ein Request an die SODA API nur maximal 10.000 Datensätze liefern kann und durch einen entwickelten Algorithmus werden so lange Requests ausgeführt bis der gewünschte Zeitraum des Users komplett empfangen und die Request an Apache Kafka gesendet wurden.

Vorgegeben durch das Framework *kafka-python* können die Einträge eines Topics nur als byte String abgelegt werden. Aus diesem Grund wird der empfangende Datensatz zuerst in ein byte string umgewandelt bevor er an Apache Kafka gesendet wird.

Folgendes Code Snippet zeigt den entwickelten Algorithmus um das Paging zu realisieren. Mit Hilfe des SodaHelpers werden zunächst die Datensätze von der API - unter Berücksichtigung des Limits, des Anfangs- und Enddatums geholt. Wie in dem Snippet zu erkennen wird die Variable *limit* in dem Skript gesetzt. die Variablen *from_date* und *to_date* werden beim Starten des Skripts von dem User gesetzt.

Der Algorithmus beruht auf der Annahme, dass wenn der empfangene Datensatz genau die Länge des Limits hat es immer noch weitere Datensätze gibt die von der API abgerufen werden müssen. Wenn also das Limit erreicht wurde wird das Datum des letzten Datensatzes

als neues Enddatum festgelegt und der Prozess beginnt von vorne, solange die Anzahl der empfanenen Datensätze nicht mehr dem Limit entsprechen oder die verwendeten Anfangs- und Enddatumswerte identisch sind.

```
limit = 10000


def process():
    requests = soda.get_data(dataset_identifier="fhrw-4uyv", from_date=from_date,
        to_date=to_date, limit=limit)
    length = len(requests)
    if length > 0:
        date = requests[-1]["created_date"]
    else:
        date = to_date
    for request in requests:
        print("Starting...")
        json_string = json.dumps(request)
        json_byte = b"" + json_string
        producer.send(topic, json_byte)
        time.sleep(random.randint(0, 50) * 0.1)
        print("Done...")

    return len(requests), date


number_of_entries, to_date = process()

while number_of_entries == limit and to_date != from_date:
    print("Next Request...")
    number_of_entries, to_date = process()
```

3.3 Data Storage

Wie in Abschnitt 3.1 - *Architektur* schon erläutert benutzen wir Apache Kafka um die kompletten Datensätze zwischenzuspeichern. Ein sog. "Consumer"liest die Daten von Apache Kafka aus und speichert die relevanten Teile der Datensätze in der Datenbank. Als Datenksystem haben wir in unserem Projekt für PostgreSQL entschieden.

In unserer Datenbank erstellten wir eine Tabelle mit dem Namen ***service_request*** um die Service Requests von New York zu speichern.

In der offiziellen Dokumentation des Datensatzes werden die einzelnen Attribute eines Service Requests genau beschrieben und deren technische Bezeichner aufgelistet. [1]

Wir erstellten unsere Datenbanktabelle bzw. die Attribute der Service Requests Tabelle mit genau den gleichen Bezeichnern wie die des SODA Datensatzes.

[1] https://dev.socrata.com/foundry/data.cityofnewyork.us/fhrw-4uyv

Um den Consumer technisch umzusetzen benutzen wir die Programmiersprache Python wie auch weitere Bibliotheken um das Handling mit Apache Kafka und der Datenbank zu vereinfachen. Diese sind:

- kafka-python
- sqlalchemy
- psycopg2 (wird von sqlalchemy benötigt)

Genau wie bei dem "Produceräbstrahiert das Framework ***kafka-python*** die Verbindung zu unserem Apache Kafka Server und erleichtert den Zugriff auf das Topic. Durch den Einsatz von ***sqlalchemy*** können wir auf die Datenbank und auf deren Tabelle in einer objektorientierten Weise zugreifen und müssen uns keine Gedanken über SQL Statements machen.

Quellcode

Neben den verwendeten Bibliotheken besteht der ***Consumer*** aus zwei Python Skripten.

- DBHelper.py
- consumer.py

Während sich das ***consumer*** Skript um das Auslesen eines Kafka Topics kümmert ist der ***DBHelper*** dafür zuständig die Verbindung zu der Datenbank aufzubauen, Tabellenspalten auszulesen und einen Datensatz in der Tabelle zu speichern.

Nachfolgend das Consumer Skript.

```
1  from kafka import KafkaConsumer
2  from DBHelper import DBHelper
3  import json
4
5  db_helper = DBHelper('bdba')
6  db_table = 'service_request'
7  db_columns = db_helper.get_table_column_names(db_table)
8
9  requests = KafkaConsumer("ServiceRequests", bootstrap_servers='localhost:9092') #
       auto_offset_reset='earliest'
10
11 for message in requests:
12     db_entry = {}
13     message_json = json.loads(message.value)
14     for column in db_columns:
15         if column in message_json: # check if key is available in request object
16             db_entry[column] = message_json[column]
17
18     db_helper.insert(db_entry, db_table)
```

Sobald das Consumer Skript gestartet wird, wird eine Verbindung mit der Datenbank aufgebaut und der sog. KafkaConsumer stellt eine Verbindung mit dem Apache Kafka Server bzw. dem Topic SServiceRequest"her. (Zeile 1 - 9)

Sobald von seitens des Producers ein neuer Datensatz in das Topic SServiceRequests"geschrieben wird, wird der Datensatz ausgelesen in ein JSON umgewandelt, die benötigten Attribute aus dem JSON gelesen und in ein temporäres Dictionary geschrieben. Dieses Dictionary wird abschließend mit Hilfe des DBHelper Skripts in die Datenbanktabelle geladen. (Zeile 11 - 18)

Wie eingangs erwähnt bezeichneten wir die Attribute der Datenbanktabelle und des SODA Datensatzes gleich und konnten sie somit als Schlüsselattribute für den jeweils anderen Datensatz benutzen.

Mit diesem kleinen "Kniff"können wir jetzt mit Hilfe der Namen der Tabellenspalten auf die Keys des JSON zugreifen den zugehörigen Wert auslesen und als neuen Wert für das temporäre Dictionary nutzen. (Zeile 14 - 16)

3.4 Data Retrieval

Die Beschaffung und Auswertung der Daten mit Apache Zeppelin übernahm Julian Ruppel. Johannes Weber bereitete die Daten mit der Programmiersparche Python auf und visualisierte sie in einem Jupyter Notebook.

3.4.1 Data Retrieval mit Apache Zeppelin

Da muss was stehen

3.4.2 Data Retrieval mit Jupyter

Da muss was stehen

Kapitel 4

Fazit

fazit in die loesung packen? Fazit = Fazit zum Vergleich der Tools und der eingesetzetn Architektur

Kapitel 5

Ablauf

Inhalt: Was muss man machen damit unsere Programme zum Laufen gebracht werden können? Eigentlich der Inhalt der README.md

Abbildungsverzeichnis

Abkürzungsverzeichnis

JSON JavaScript Object Notation

CSV Comma-separated values

API Application Programming Interface

SODA Socrata Open Data