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 Datei die lokal auf einem Rechner abgelegt wird wie z. B. eine Comma-separated values (CSV) oder JavaScript Object Notation (JSON) Datei.

Unter einem Data Stream versteht man einen kontinuierlichen Datenstrom wie z. B. die Erstellung von immer wieder neuen Twitter Nachrichten.
Ein wichtiges Merkmal eines Data Streams ist, dass nicht vorherzusehen ist wann der Datenstom zu Ende ist - er könnte theoretisch unendlich sein.
Im Falle des Twitter Datenstroms ist es nicht abzusehen wann die letzte Twitter Nachricht geschrieben wird.

Für unsere Aufgabe ist darauf zu achten, dass der Datenstrom über ein Application Programming Interface (API) öffentlich zugänglich.

Data Storage ist die Speicherung der Daten. Hierbei wurde uns die Anforderung gestellt, das für die Speicherung der Daten die Streaming Plattform Apache Kafka verwende wird. Des Weiteren war es gestattet die Daten zusätzlich in einer relationen Datenbank, NoSQL Datenbank oder mit Spark Streaming zu speichern.

Data Retrieval ist die Beschaffung der Daten aus einer Datenbank mit Structured Query Language (SQL) Abfragen und die abschließende Ausgabe der Ergebnisse in Form von Tabellen oder einfachen Visualisierungen.
Es sollen mindestens drei verschieden SQL Abfragen abgesetzt werden mit unterschiedlichen Filter- und Agreggatsfunktionen sowie einer Teilaggregation wie z. B. GROUP BY.

Die Visualisierung der Daten soll in einem virtuellen Notebook erfolgen.
Als virtuelles Notebook durften wir zwischen Apache Zeppelin oder Jupyter entscheiden.

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 gibt allen „New Yorkern“ und somit auch der ganzen Welt, die Chance, Open Data, also frei zugängliche Daten, einfach zu konsumieren.[1]

NYC Open Data ermöglicht es 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 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 und Programmiersprachen.

Kapitel 3 - *Lösungsansatz* erläutert das gewählte Szenario und die verwendeten Architektur sowie die Lösung zu den Themen Data Ingestion, Data Storage und Data Retrieval.

[1] NYC Open Data. *Our mission: open data for all*. 2017. URL: https://opendata.cityofnewyork.us/overview/ (besucht am 03. 02. 2018).

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.

[1] z.B. TensorFlow von Google

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 lesen. 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.

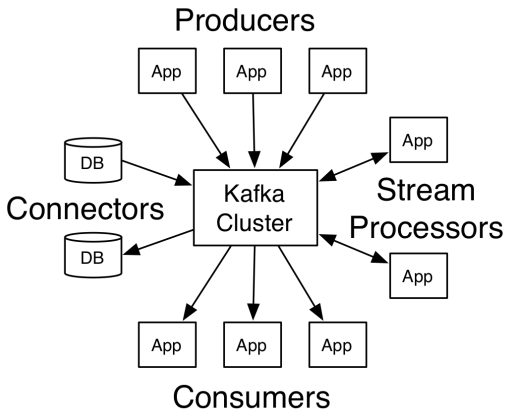

Abbildung 2.1: Apache Kafka Architektur[3]

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

[2] wie z.B. Twitter oder JDBC

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 polyglotte Abfrageskripte ad-hoc gegen die diversen Datenquellen ausgeführt und deren Ergebnisse in einem AngularJS, konfigurierbaren Web-Dashboard visuell und interaktiv dargestellt. Dadurch 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.

Socrata Open Data (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 API im JSON Dateiformat abrufen.

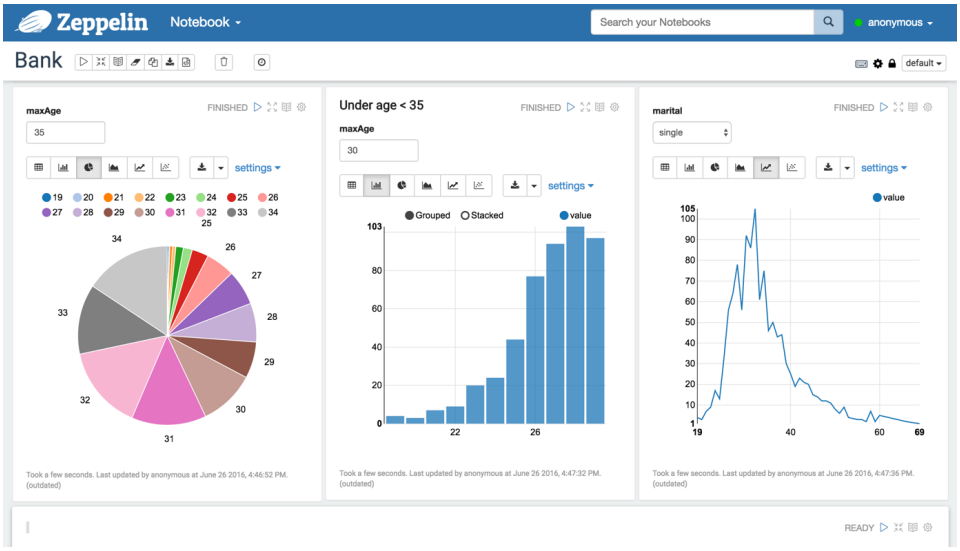

Abbildung 2.2: Beispielhaftes Notebook mit Apache Zeppelin[4]

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: Dieser von Confluent Mitgründer und CEO Jay Kreps entworfene Ansatz verzichtet auf eine Batch-Verarbeitung und kommt somit mit lediglich mit *Ingestion-*, *Speed-* und *Serving-Layer* aus. Damit spart man sich Entwicklung und Betrieb von zwei separaten Schichten und das aufwändige Abmischen von Batch-Daten mit Live-Daten im *Serving-Layer*. Voraussetzung ist allerdings, dass der *Ingestion-Layer* nicht nur Daten volatil durchreicht sondern vielmehr als **Puffer** die Rohdaten persistent im *Master-Dataset* vorhält, um im Falle einer neuen, noch nicht vorberech-

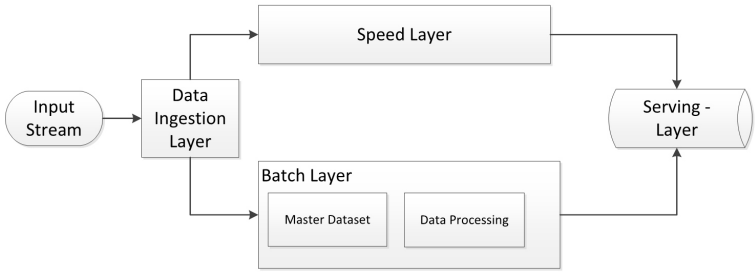

Abbildung 3.1: Schema der λ Architektur[1]

neten Anfrage oder Änderung im *Speed-Layer* die Rohdaten erneut bereit zu stellen. Um ein korrektes Replay der Nachrichten sicherzustellen beruht der Puffer auf einem kanonisches Log, in dem lediglich Nachrichten unverändert hinzugefügt, aber bereits gespeicherte Nachrichten nicht mehr verändert oder in ihrer Reihenfolge verschoben werden können. Um gleichzeitig Nähe als auch Abgrenzung zur λ Architektur zu veranschaulichen wurde dieser Ansatz nach dem griechischen κ benannt.

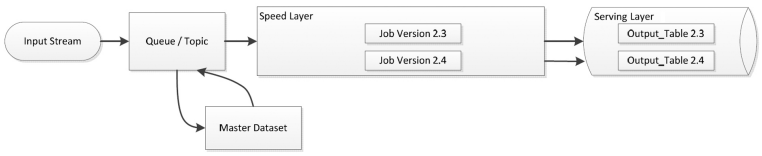

Abbildung 3.2: Schema der κ Architektur[2]

Der erste Lösungsentwurf für unser Problem orientiert sich an der Lambda Architektur. Weil die von uns gewählte Datenquelle kein Stream sondern ein statisches, online abrufbares Datenset ist, sieht unser Lösungsansatz vor, mittels eines Kafka Producers **(1)** die Online-Daten zu laden und sukzessive in einen Kafka Topic zu schreiben, um somit eine Art Pseudo-Stream zu imitieren. Da das Web-API ohnehin *Paging* über das Datenset erlaubt sollte die Implementierung nicht sonderlich kompliziert sein.

In Schritt **(2)** sammelt ein Consumer zeitgesteuert die Nachrichten des Kafka Topics ein und schreibt die Daten per SQL `INSERT` [3] in eine relationale Datenbanktabelle. Auf dieser Tabelle horcht ein `AFTER INSERT TRIGGER` **(3)**, der, sobald Daten in die Tabelle geschrieben wird, eine `FUNCTION` bzw. `STORED PROCEDURE` aufruft um Aggregate über die neuen Datensätze zu berechnen und in einer separaten Tabelle abzuspeichern bzw. mit bereits bestehenden Aggregaten zu verrechnen. Danach kann die Tabelle mit den Rohdaten theoretisch geleert werden, spätestens jedoch sobald der verfügbare Speicherplatz zu neige geht[4].

[3] Um die IO-Last der Datenbankverbindung gering zu halten sollte die Daten per BULK-INSERT erfolgen

[4] Alternativ könnte man einen `INSTEAD OF TRIGGER` benutzen und ausschließlich Aggregate permanent zu speichern

Parallel dazu konsumiert ein in Zeppelin- oder Jupyter-Notebook eingebetteter Consumer[5] die Rohdaten aus dem gleichen Topic **(4)**.

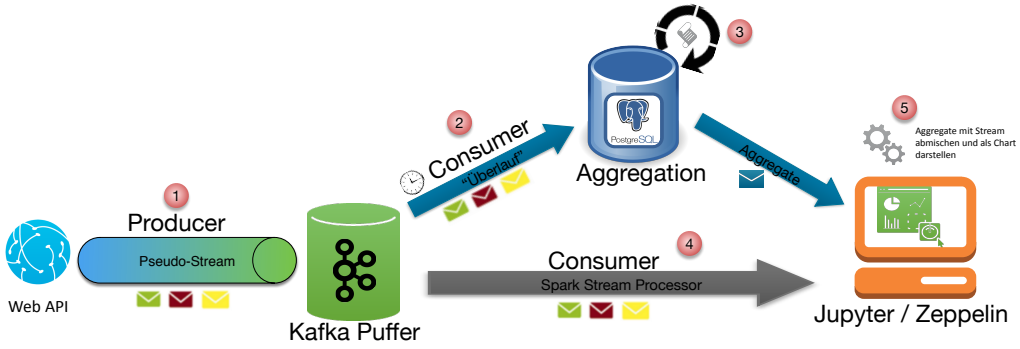

Abbildung 3.3: Lösungsentwurf nach λ

Somit wären im Auswertungs-Dashboard **(5)** sowohl Charting auf Live-Daten als auch über aggregierte Bestandsdaten aus PostgreSQL, die ggf. aus Speicherplatzgründen gar nicht mehr in Kafka vorgehalten werden können, möglich. Der Serving Layer würde in diesem Fall in die Auswertungskomponente fallen.

Da allerdings mit diesem Ansatz die bereits erörterten Nachteile der Lambda Architektur einhergehen und wir in unserem Beispiel keinen unendlichen Stream sondern eine endliche Datenmenge haben, die ganzheitlich in den Kafka-Puffer passt, haben wir den Lösungsentwurf überarbeitet und an die einfachere Kappa-Architektur angeglichen.

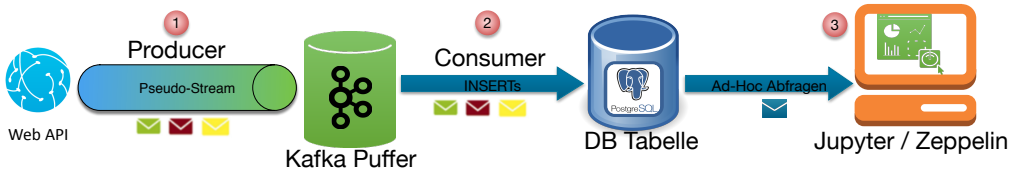

Abbildung 3.4: Finaler Lösungsentwurf nach κ

Schritt **(1)** bleibt unverändert, wohingegen in Schritt **(2)** nur noch ein einziger Consumer die Nachrichten des Kafka Topics subskribiert und direkt in eine Datenbanktabelle weiter leitet. Der zweite Consumer sowie Voraggregationen in PostgreSQL entfallen. Somit muss in den Zeppelin- bzw. Jupyter-Notebooks **(3)** auf nur eine Datenquelle zugegriffen werden, um die Daten auszuwerten und zu visualisieren.

[5] Apache Spark liefert bereits eine Kafka-Consumer Bibliothek

3.2 Data Ingestion

Wie in Kapitel 1 - *Einleitung* erläutert wird für die Erstellung des Prototyps NYC Open Data als Datenquelle genutzt.

NYC Open Data bietet verschiedene Datenätze an um an die unterschiedlichsten Informationen aus New York zu gelangen wie z. B. der Standort von öffentlichen Wi-Fi Hotspopts oder offene Stellenauschreibungen.[6]

Innerhalb dieses Projektes wird der Datensatz mit dem Kürzel *fhrw-4uyv* verwendet. Er beinhält alle Service Request die seit 2011 von den Einwohner von New York City abgesetzt worden sind.

Beispielsweise kann mit diesem Datensatz herausgefunden werden wo welche Straßenlaternen in New York City ausgefallen sind oder in welchem Haus zu welcher Uhrzeit gefeiert wurde da sich jemand über den Lärm beschwert hat.

In unserem Projekt beschaffen wir die Daten über eine CSV Datei die man sich bei NYC Open Data herunterladen kann und direkt über die SODA API die einen eigenen Endpunkt anbietet um die Service Requests abzufragen.

Die Data Ingestion per CSV Datei setzte Julian Ruppel mit der Programmiersprache Java um und den kontinuierlichen Datenstrom über die SODA API wurde von Johannes Weber mit der Programmiersprache Python ausgelesen.

Beide Ansätze werden in diesem Kapitel beschrieben und anschließend miteinander verglichen.

3.2.1 Data Ingestion via SODA Schnittstelle

Folgende Python Bibliotheken wurden genutzt um die Implementierung des Producers zu unterstützen:

- `sodapy`
- `kafka-python`

Im Kern basiert `sodapy` auf dem `Request` Paket von Python und vereinfacht das Absenden von Anfragen an die SODA API.[7]

`kafka-python` ist ein offiziell unterstützter Klient für Apache Kafka in der Programmiersprache Python und basiert lose auf der offiziellen Java Implementierung des Kafka Kli-

[6] NYC Open Data. *List of most popular datasets*. 2017. URL: https://data.cityofnewyork.us/browse?provenance=official&sortBy=most_accessed&utf8=%E2%9C%93 (besucht am 06. 02. 2018).

[7] Cristina. *sodapy*. 2017. URL: https://github.com/xmunoz/sodapy (besucht am 05. 02. 2018).

ents.[8] Gegenüber dem offiziellen Paket `confluent-kafka-python` ist `kafka-python` komplett in Python geschrieben und somit entfällt die Installation des Pakets `librdkafka` , das bei dem offiziellen Klient von Confluent noch installiert werden muss.[9]

Python Quellcode Programmablauf

Der Producer besteht insgesamt aus zwei Python Skripten:

- `SodaHelper.py`
- `producer.py`

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

Da es sich hierbei nicht um ein Live Stream handelt wie z. B. bei der Twitter API, haben wir uns dazu entschieden einen „Fake Stream“ zu erstellen.
Dies wurde erreicht indem nicht alle Daten sofort an Apache Kafka weitergeleitet werden sondern immer ein Abstand zwischen 0 und 1 Sekunde zwischen dem Senden der einzelnen Datensätze erzwungen wird.

Der 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 geladen werden.

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 ein 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 der Algorithmus führt so lange Requests an die API aus bis der gewünschte Zeitraum des Users komplett empfangen und die Request an Apache Kafka gesendet wurden.

Vorgegeben durch das Paket `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 und dann an Apache Kafka gesendet.

Folgendes Code Snippet zeigt den Algorithmus um das Paging zu realisieren.

[8] Dana Powers. *kafka-python*. 2017. URL: http://kafka-python.readthedocs.io/en/master/ (besucht am 05. 02. 2018).
[9] Confluent. *Python*. 2017. URL: https://cwiki.apache.org/confluence/display/KAFKA/Clients#Clients-Python (besucht am 05. 02. 2018).

Mit Hilfe des SodaHelpers werden zunächst die Datensätze von der API - unter Berücksichtigung des Limits, des Anfangs- und Enddatums geholt.
Die Variablen `from_date` und `to_date` werden beim Starten des Programms 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.
Dies geschieht solange die Anzahl der empfanenen Datensätze nicht dem Limit entsprechen oder die verwendeten Anfangs- und Enddatumswerte identisch sind.

```
1     def fetch_data(self, from_date, to_date, limit):
2         requests = self.soda.get_data(dataset_identifier=config.
              SOCRATA_DATASET,
3                                       from_date=from_date,
4                                       to_date=to_date,
5                                       limit=limit)
6         length = len(requests)
7         if length > 0:
8             date = requests[-1]["created_date"]
9         else:
10            date = self.to_date
11        for request in requests:
12            json_string = json.dumps(request)
13            json_byte = b"" + json_string
14            print("Sending to Kafka...")
15            self.producer.send(self.topic, json_byte)
16            # time.sleep(random.randint(0, 50) * 0.1)
17
18        return len(requests), date
19
20    def run(self):
21        number_of_entries, to_date = self.fetch_data(from_date=self.
              from_date,
22                                                     to_date=self.to_date,
23                                                     limit=self.limit)
24        while number_of_entries == self.limit and to_date != self.
              from_date:
25            print("Next Request...")
26            number_of_entries, to_date = self.fetch_data(from_date=
                  self.from_date,
27                                                         to_date=to_date,
```

```
28                                                    limit=self.limit)
29         print("End.")
```

Listing 3.1: Code Snippet aus Producer.py

3.2.2 Data Ingestion via CSV Datei

Als offline-fähige Alternative zur SODA API gibt es einen weiteren Kafka Producer in Java. Dieser liest die Nachrichten zeilenweise aus einer lokalen CSV Datei ein und publiziert die Datensätze als JSON einzeln an einen Kafka Topic. Die CSV Datei wurde vorher aus dem NYC OpenData Portal runtergeladen.

Der Programmablauf lässt sich in wenigen Stichpunkten beschreiben:

1. Verbindung des `KafkaProducer<Long, String>` zum Kafka Cluster konfigurieren.Dazu zählen hauptsächlich Host und Port sowie die Datentypen, um Schlüssel und Wert der Nachrichten zu serialisieren.
2. Zeilenweises einlesen der CSV Datei und dabei jeweils den Datensatz in einen JSON-String transformieren, diesen `String` als Wert in einen `ProducerRecord<Long, String>` setzen und an den bestimmten Kafka Topic senden. Um einen realen Stream zu simulieren wartet der Producer-Thread pro verarbeiteten Datensatz eine zufällige Wartezeit zwischen 0 bis 2 Sekunden.

Listing 3.2 zeigt einen Überblick über die relevanten Methoden.

```
1
2   public static void main(String[] args) throws Exception {
3
4     Reader in = new FileReader("/data/311
          _Service_Requests_from_2010_to_Present.csv");
5     Iterable<CSVRecord> records = CSVFormat.RFC4180.
          withFirstRecordAsHeader().parse(in);
6
7     new MyProducer("").runProducer(records);
8
9   }
10
11  public void runProducer(final Iterable<CSVRecord> records) throws
        Exception {
12
13    try (final Producer<Long, String> producer = createProducer()) {
14
15      for (final CSVRecord csvRecord : records) {
```

```
16          String json = objectMapper.writeValueAsString(csvRecord.toMap
                ());
17          final ProducerRecord<Long, String> record = new ProducerRecord
                <Long, String>(KafkaCommons.loadProperties().getProperty("
                TOPIC", KafkaCommons.TOPIC), Long.parseLong(csvRecord.get(
                "Unique Key")), json);
18          Thread.sleep(new Random().nextInt(2000));
19          RecordMetadata metadata = producer.send(record).get();
20          System.out.printf("sent record(key=%s value=%s) " + "meta(
                partition=%d, offset=%d)\n", record.key(), record.value(),
                 metadata.partition(), metadata.offset());
21
22        }
23
24      }
25    }
26
27    private Producer<Long, String> createProducer() throws IOException{
28      Properties props = new Properties();
29      props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, KafkaCommons.
            loadProperties().getProperty("BOOTSTRAP_SERVERS",
            KafkaCommons.BOOTSTRAP_SERVERS));
30      props.put(ProducerConfig.CLIENT_ID_CONFIG, "KafkaCSVProducer");
31      props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
            LongSerializer.class.getName());
32      props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
            StringSerializer.class.getName());
33      return new KafkaProducer<Long, String>(props);
34    }
```

Listing 3.2: Auszug aus `com.srh.bdba.dataengineering.MyProducer`

Zur Implementierung wurden folgende Bibliotheken über Maven eingebunden:

- `org.apache.kafka:kafka-clients`
- `org.apache.commons:commons-csv`
- `com.fasterxml.jackson.core:jackson-core`
- `com.fasterxml.jackson.core:jackson-databind`

3.3 Data Storage

Apache Kafka wird verwendet um den kompletten Datensatz eines Service Requests abzuspeichern.

Ein Consumer liest die Daten von Apache Kafka aus und speichert die relevanten Attribute eines Datensatzes in der PostgreSQL Datenbank.

Zunächst erstellten wir eine Tabelle mit dem Namen `service_request` um die Service Requests von New York zu speichern.

Eine stichprobenartige Analyse des Datensatzes hat ergeben, dass einzelne Felder des original Datensatzes sehr häufig mit `NULL` Werten belegt sind.

In der Datenbanktabelle `service_request` wurden diese Felder außer Acht gelassen.

Listing 3.3 - *SQL Skript für Datenbanktabelle* zeigt das SQL Statement um die Tabelle `service_request` in der Datenbank anzulegen.

```
1  CREATE TABLE public.service_request
2  (
3      created_date timestamp with time zone,
4      agency_name character(255) COLLATE pg_catalog."default",
5      complaint_type character(255) COLLATE pg_catalog."default",
6      descriptor character(255) COLLATE pg_catalog."default",
7      longitude double precision,
8      latitude double precision,
9      agency character(255) COLLATE pg_catalog."default",
10     location_type character(255) COLLATE pg_catalog."default",
11     incident_zip character(10) COLLATE pg_catalog."default",
12     incident_address character(255) COLLATE pg_catalog."default",
13     street_name character(255) COLLATE pg_catalog."default",
14     cross_street_1 character(255) COLLATE pg_catalog."default",
15     cross_street_2 character(255) COLLATE pg_catalog."default",
16     address_type character(255) COLLATE pg_catalog."default",
17     city character(255) COLLATE pg_catalog."default",
18     status character(50) COLLATE pg_catalog."default",
19     due_date timestamp with time zone,
20     borough character(100) COLLATE pg_catalog."default",
21     resolution_description text COLLATE pg_catalog."default",
22     unique_key character(255) COLLATE pg_catalog."default" NOT NULL,
23     closed_date timestamp with time zone,
```

Listing 3.3: SQL Skript für Datenbanktabelle

In der offiziellen Dokumentation des Datensatzes werden die einzelnen Attribute eines Service Requests genau beschrieben und deren technische Bezeichner aufgelistet. [10]

Die Attribute der Tabelle `service_request` sind identisch zu den Bezeichnern des original SODA Datensatzes.

Um das Handling mit Apache Kafka zu vereinfachen wurden zwei Python Bibliotheken installiert. Diese sind:

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

Genau wie bei dem Producer abstrahiert das Paket `kafka-python` die Verbindung zu unserem Apache Kafka Server und erleichtert den Zugriff auf das Topic.

Durch den Einsatz von `sqlalchemy` ist es möglich auf die Datenbank und deren Tabelle(n) in einer objektorientierten Weise zuzugreifen und die Ausführung von SQL Statements wird vereinfacht bzw. durch Klassenmethoden abstrahiert.

Python Quellcode Programmablauf

Neben den verwendeten Bibliotheken besteht derConsumer aus zwei Python Skripten.

- `DBHelper.py`
- `consumer.py`

Während sich das `consumer` Skript um das Auslesen eines Kafka Topics kümmert ist die `DBHelper` Klasse für die Verbindung zu der Datenbank zuständig, liest Tabellenspalten aus und speichert einen Datensatz in der Tabelle.

Nachfolgend die Code Snippet der Consumer Klasse.

```
1         self.db_helper = DBHelper(config.DATABASE_NAME)
2         self.db_columns = self.db_helper.get_table_column_names(self.
              db_table)
3
4     def run(self):
5         requests = KafkaConsumer(config.KAFKA_TOPIC,
6                                  bootstrap_servers=config.KAFKA_SERVER)
                                      # auto_offset_reset='earliest'
7
8         for message in requests:
9             db_entry = {}
```

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

```
10              message_json = json.loads(message.value)
11              for column in self.db_columns:
12                  if column in message_json: # check if key is available
                        in request object
13                      db_entry[column] = message_json[column]
14              print("Saving in DB...")
15              self.db_helper.insert(db_entry, self.db_table)
```

Listing 3.4: Code Snippet aus Consumer.py

Sobald das Consumer Skript aufgerufen wird, wird eine Verbindung mit der Datenbank aufgebaut und der `KafkaConsumer` stellt eine Verbindung mit dem Apache Kafka Server bzw. dem Topic `ServiceRequests` her. (Zeile 1 - 7)

Sobald von seitens des Producers ein neuer Datensatz in das Topic `ServiceRequests` 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.

Wie eingangs erwähnt bezeichneten wir die Attribute der Datenbanktabelle und des SODA Datensatzes identisch.

Mit diesem kleinen „Kniff“ kann mit Hilfe der Namen der Tabellenspalten auf die Keys des JSON zugegriffen, den zugehörigen Wert ausgelesen und als neuen Wert für das temporäre Dictionary genutzt werden. (Zeile 8 - 15)

Java Quellcode Programmablauf

Auch für den Consumer gibt es eine alternative Implementierung in Java. Diese findet sich in `com.srh.bdba.dataengineering.MyConsumer`. Der Programmablauf ist ähnlich zu der Python Implementierung, weshalb an dieser Stelle auf Codelistings im Anhang TODO verwiesen wird. Kurz zusammengefasst wird ein `KafkaConsumer<Long, String>` konfiguriert und pro 100 Millisekunden am Kafka-Cluster nachgefragt, ob es neue `ConsumerRecord <Long, String>` für das entsprechende Topic gibt. Falls dem so ist wird die JSON Nachricht aus dem `ConsumerRecord<Long, String>` ausgepackt und per `PreparedStatement` über `JDBC` in die PostgreSQL Tabelle eingefügt.

3.4 Data Retrieval

Die Aufgabe im Bereich Data Retrieval ist es die Daten aus der Datenbank auszulesen und diese mit einem virtuellen Notebook zu visualisieren.

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

3.4.1 Data Retrieval mit Apache Zeppelin

Apache Zeppelin bringt standardmäßig einige nützliche Funktionen mit, die es für die Datenauswertung interessant machen. Dazu zählen u.A.:

- SQL via JDBC inkl. PostgreSQL Support
- Interaktive Diagramme über Formulargeneriung und Pivot-Charts
- Frei konfigurierbare Anordnung der Diagramme im Notebook
- Versionsverwaltung für Notebooks um verschiedene Stände abzuspeichern und schnell zwischen Versionen zu wechseln

Über den Befehl `$/bin/zeppelin.sh` startet Zeppelin und ist standardmäßig im Browser über den Port 8080 erreichbar. Die gesamte Konfiguration, Entwicklung und Ausführung des Notebooks erfolgt im Web. Letztlich ist ein Notebook ein Kanvas in den man Kacheln erstellen und verwalten kann. Jede Kachel besteht aus 3 integralen Bestandteilen:

- **Code** der mittels der vielen Interpreter die Daten läd, die angezeigt werden sollen, z.B. SQL.
- **Konfiguration** der Visualisierung. Dazu zählen verschiedene Diagrammtypen und -konfigurationen sowie generierte Formularfelder.
- **Diagramm** welches die Daten responsiv visualisiert oder eine schlichte Tabelle um die Daten darzustellen.

Sobald der Code ausgeführt wird, egal ob manuell oder per automatischen Zyklus, wird das Diagramm in der Kachel aktualisiert.

Leider ist die Auswahl an Diagrammtypen und deren Flexibilität und Erfassbarkeit abhängig von den Daten sehr begrenzt. Standardmäßig gibt es nur eine schlichte Tabelle, Balken-, Flächen-, Linien- und Punktdiagramm die relativ starr sind. Da erweiterte Visualisierungen wie Heatmaps und Karten noch in der Entwicklung sind ist die Auswertung auf die oben genannten Typen begrenzt. Zudem ist die maximale Anzahl an Datensätzen die dargestellt bzw. visualisiert werden könne auf 102400 begrenzt. Nichts desto trotz wurde ein beispielhaftes Notebook erstellt:

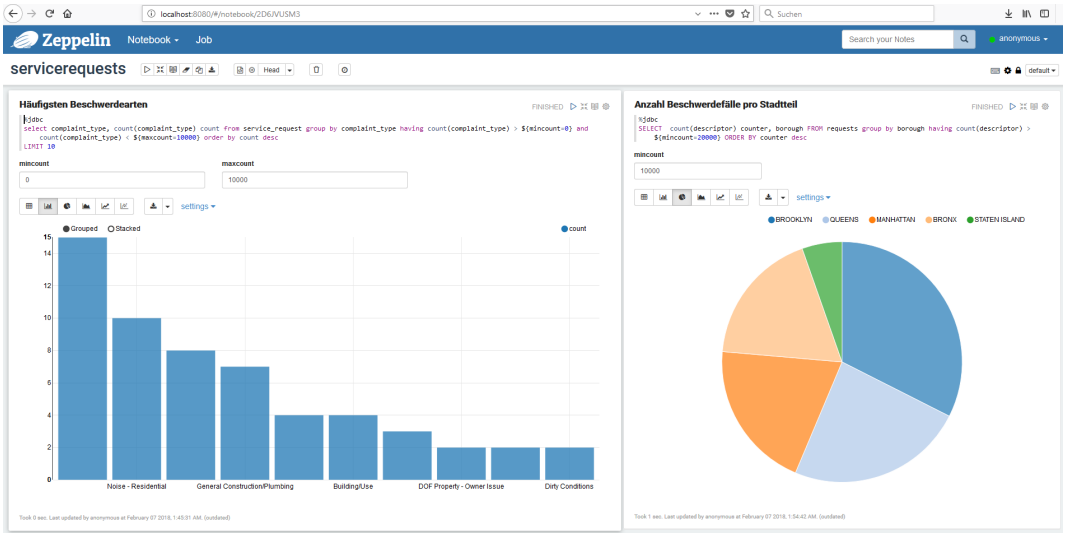

Abbildung 3.5: Ausschnitt des Zeppelin Notebooks

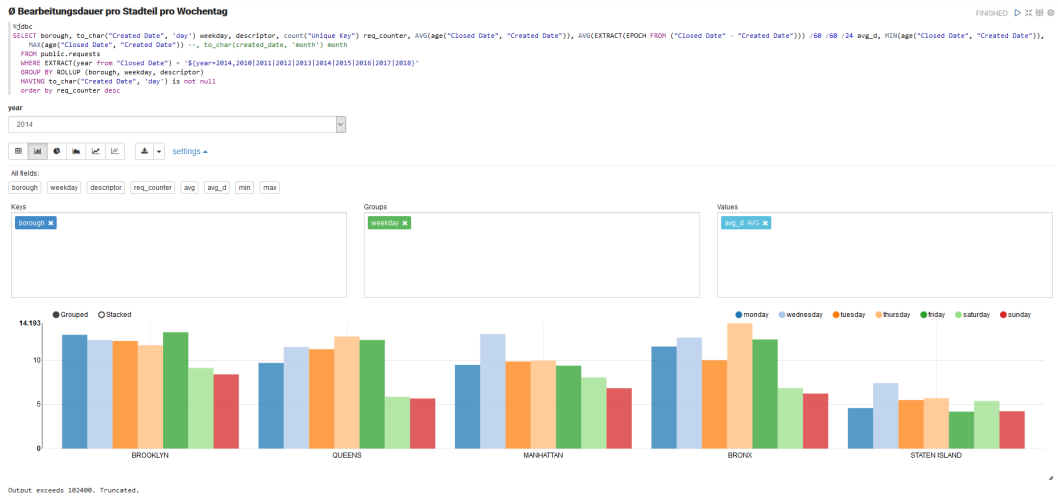

Abbildung 3.6: Pivot Chart im Zeppelin Notebook

3.4.2 Data Retrieval mit Jupyter

Das Jupyter Notebook bietet die Möglichkeit mit *Magic-Commands* direkt aus dem Notebook heraus eine Verbindung mit der Datenbank aufzubauen und SQL Statements abzusetzen.

Einfache Tabellen werden direkt in Jupyter visualisiert wohingegen komplexere Visualisierungen wie z. B. Balkendiagramme oder Geo Plots mit Python Bibliotheken dargestellt werden müssen.
Für die Darstellung in Jupyter werden dann sog. *Widgets* installiert und aktiviert.

Folgende Python Bibliotheken wurden installiert um SQL Statements in Jupyter ausführen und visualisieren zu können.

- `ipython-sql`
- `sqlalchemy` (wird von ipython-sql benötigt)
- `bokeh`
- `gmaps`

Mit `ipython-sql` werden SQL *Magic-Command* in Jupyter aktiviert.

`ipython-sql` nutzt `sqlalchemy` um sich mit der Datenbank zu verbinden. Ein abgesetztes SQL Statement lässt sich entweder direkt in Jupyter ausgeben oder einer beliebigen Variable zuordnen die dann weiterverarbeitet werden kann.

Abbildung 3.7 - *direkte SQL Ausgabe in Jupyter* zeigt ein SQL Statemement mit *Magic-Command* Das Resultat wird direkt in einem Jupyter Notebook als Tabelle gerendert.

Abbildung 3.8 - *Zuweisung der SQL Ausgabe einer Variable* zeigt die Zuweisung eines Resultates zur Variable `works`.

```
In [1]: %load_ext sql

In [2]: %%sql postgresql://will:longliveliz@localhost/shakes
   ...: select * from character
   ...: where abbrev = 'ALICE'
   ...:
Out[2]: [(u'Alice', u'Alice', u'ALICE', u'a lady attending on Princess Katherine', 22)]
```

Abbildung 3.7: direkte SQL Ausgabe in Jupyter[11]

```
In [16]: works = %sql SELECT title, year FROM work
43 rows affected.
```

Abbildung 3.8: Zuweisung der SQL Ausgabe einer Variable[12]

`bokeh` ist eine Python Bibliothek um viele Arten der Visualisierung umzusetzen wie z. B. Balkendiagramme, Scatter Plots, Geo Maps oder Zeitreihen.

Die Visualisierung von Geo Daten erfolgt mit der Bibliothek `gmaps`.
Diese greift auf die Karten von Google Maps zu erlaubt es die Geo Daten in einem Layer über einen beliebigen Kartenausschnitt zu legen.
Der Vorteil von `gmaps` gegenüber `bokeh` ist zum einen die Nutzung des Kartenmaterials von Google aber auch die interaktive Nutzung des Kartenauschnitts mit z. B. Google StreetView.

Folgende Fragestellungen wurden im Rahmen von Data Retrieval beantwortet.

1. Zeige alle Beschwerdetypen die häufiger als 400 aber seltener als 8000 Mal gemeldet wurden?

[11] (Catherine Devlin. *ipython-sql*. 2017. URL: https://github.com/catherinedevlin/ipython-sql [besucht am 06. 02. 2018])
[12] (ebd.)

2. Wie lautet die Beschreibung der häufig vorkommenden Service Requests?
3. An welchen Orten von New York City wurden Service Request vom Typ 'Noise - Residential' abgesetzt?
4. Wieviele Service Requests sind im Jahr 2017 eingegangen? Gruppiert nach Tag und Sortiert nach dem Erstellungsdatum.
5. Wie lange wurde eine Art von Service Request im Durschnitt bearbeitet? Wie lange war die minimale und maximale Bearbeitungszeit?

Nachfolgender Abschnitt listet die SQL Statements zu jeder Fragestellung sowie ein dazugehöriges Beispiel.

zu 1.

```
SELECT complaint_type, COUNT(complaint_type)FROM service_request GROUP
BY complaint_type HAVING COUNT(complaint_type)> 400 AND COUNT(complaint_type
)< 8000
```

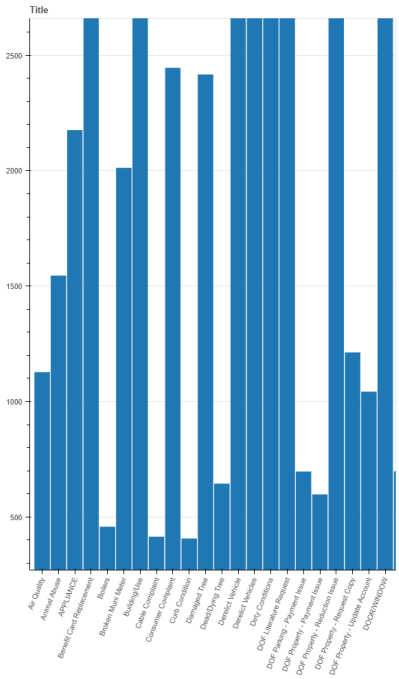

Abbildung 3.9: Tabelle mit allen Beschwerdetypen[13]

zu 2.

```
SELECT descriptor, COUNT(descriptor)FROM service_request WHERE descriptor
 IS NOT NULL GROUP BY descriptor ORDER BY count DESC
```

[13] eigene Darstellung
[14] eigene Darstellung

descriptor	count
HEAT	33615
Loud Music/Party	7771
Street Light Out	5788
Pothole	4403
Banging/Pounding	3512
CEILING	3481
No Access	3361
WALLS	3326
VERMIN	3144
WATER-LEAKS	3018

Abbildung 3.10: Auszug aus den meisten Service Requests[14]

zu 3.

```
SELECT longitude, latitude FROM service_request WHERE complaint_type =
'Noise - Residential'and latitude IS NOT NULL and longitude IS NOT NULL
```

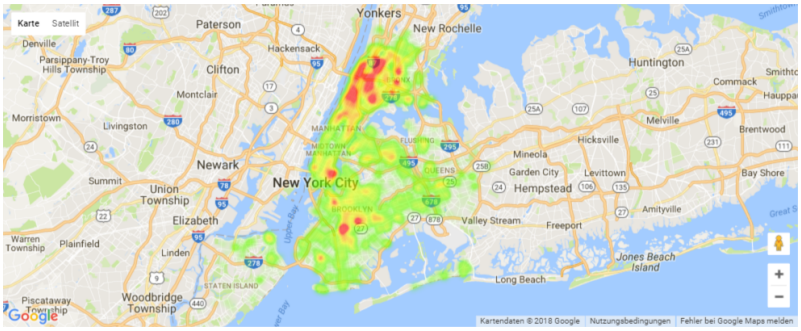

Abbildung 3.11: Heatmap der Lärmquellen in New York[15]

zu 4.

```
SELECT date_trunc('day', created_date)AS dd, COUNT(created_date)as daily_sum
 FROM service_request where EXTRACT(year from created_date)= '2017'GROUP
 BY dd ORDER BY date_trunc('day', created_date)
```

zu 5.

```
SELECT AVG(closed_date - created_date)AS avg_duration, MIN(closed_date
- created_date)AS min_duration, MAX(closed_date - created_date)AS max_duration
, complaint_type FROM service_request where created_date IS NOT NULL and
 closed_date IS NOT NULL GROUP BY complaint_type HAVING MAX(closed_date
```

[15] eigene Darstellung
[16] eigene Darstellung

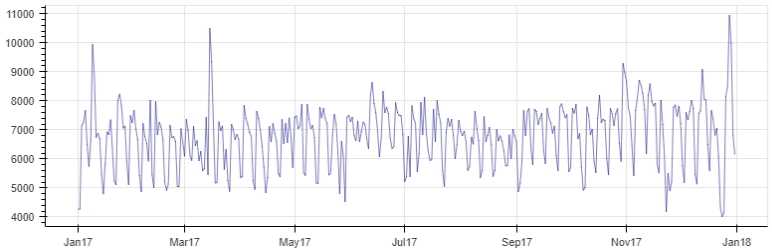

Abbildung 3.12: Timeline aller Service Requests in 2017[16]

```
 - created_date)< INTERVAL '365 days'and MIN(closed_date - created_date
)> '00:00:00'ORDER BY avg_duration asc
```

avg_duration	min_duration	max_duration	complaint_type
0:00:45.055789	0:00:02	14:12:08	Noise Survey
0:01:02.153064	0:00:02	1 day, 5:28:04	Benefit Card Replacement
0:05:19.312102	0:00:38	0:16:46	City Vehicle Placard Complaint
2:43:55.812392	0:04:06	2 days, 9:45:52	Noise - Park
2:47:57.770701	0:07:46	12:40:45	Panhandling
2:49:54.433156	0:06:37	2 days, 9:39:35	Urinating in Public
3:02:27.078261	0:07:17	1 day, 0:05:02	Illegal Fireworks
3:09:43.596154	0:04:06	1 day, 11:16:52	Disorderly Youth
3:21:06.112903	0:05:58	1 day, 7:00:36	Drinking
3:32:48.511976	0:05:03	17 days, 7:52:09	Noise - House of Worship
3:34:28.800000	0:04:37	7:27:16	Squeegee

Abbildung 3.13: Lageparameter zur Bearbeitungsdauer von Service Requests[17]

[17] eigene Darstellung

Kapitel 4

Inbetriebnahme

In diesem Kapitel werden die einzelnen Schritte beschrieben, welche durchgeführt werden müssen um den Prototyp ausführen zu können.

Hinweis:
Hierbei handelt es sich lediglich um eine grob-granulare Beschreibung der Inbetriebnahme des Prototypen. Weitere Details zur Installation und Nutzung können den Quellen entnommen werden.

4.1 Voraussetzung & Infrastruktur

4.1.1 Installation

Folgende Komponente müssen installiert und lauffähig sein:

- Apache Kafka und Apache Zookeeper
- PostgreSQL

Je nachdem welche Technologie man für die Consumer, Producer und Auswertung verwenden möchte ergeben sich folgende zwei alternative Technologiestacks:

- Python 2
- pip
- Jupyter

- JDK 1.8
- Maven
- Apache Zeppelin

4.2 Infrastruktur Setup & Konfiguration

1. Projekt aus Git Repository[1] auschecken
2. Apache Zookeeper starten und initialisieren
3. Apache Kafka starten und initialisieren
4. Kafka Topic anlegen
5. PostgreSQL Datenbankschema initialisieren[2]

4.3 Setup und Starten des Python Prototypen

1. Python Bibliotheken installieren
   - `$pip install sodapy`
   - `$pip install kafka-python`
   - `$pip install psycopg2`
   - `$pip install sqlalchemy`
   - `$pip install jupyter`
   - `$pip install bokeh`
   - `$pip install ipython-sql`
   - `$pip install gmaps`
2. `gmaps` Widget für Jupyter aktivieren
   - `$jupyter nbextension enable --py --sys-prefix widgetsnbextension`
   - `$jupyter nbextension enable --py --sys-prefix gmaps`
3. Google Maps API Key erstellen um die Google Maps Visualisierungen nutzen zu können[3]
4. Applikation Token für die SODA API beziehen[4]
5. Python Konfigurationsdatei `/python/config.py` pflegen

[1] https://github.com/johannesweber/BDBA_DataEngineering.git
[2] DDL Skripte um das Datenbankschema in PostgreSQL aufzubauen befinden sich im Ordner /db
[3] Eine Anleitung zum Erzeugen eines API Keys gibt es http://jupyter-gmaps.readthedocs.io/en/latest/authentication.html
[4] Socrata. *Obtaining an Application Token*. 2017. URL: https://dev.socrata.com/docs/app-tokens.html (besucht am 05. 02. 2018).

6. Python Consumer und Producer starten via `/python/main.py`
7. Jupyter starten und vorkonfiguriertes Notebook BDBA/DataEngineering.ipynb laden

4.4 Setup und Starten des Java Prototypen

1. CSV Datei von https://data.cityofnewyork.us/Social-Services/311-Service-Requests-from-2010-to-Present/erm2-nwe9 runterladen und lokal speichern (ca. 11GB).
2. Java Projekt mit Maven `$mvn clean install` bauen.
3. Konfiguration `/java/src/main/resources/kafka_config.properties` pflegen.
4. Python Consumer und Producer starten via `com.srh.bdba.dataengineering.Main`. Beim Aufruf der `Main()` muss man den Pfad zur CSV Datei angeben.
5. Apache Zeppelin starten und Datenbankverbindung sowie JDBC Treiber im PSQL Interpreter pflegen[5].
6. Vorkonfiguriertes Notebook BDBA/zeppelin_servicerequests_notebook.json laden und Abfragen ausführen.

[5] TODO sceenshot im Anhang

Kapitel 5

Zusammenfassung & Fazit

Gesamtarchitektur

Apache Kafka wird dazu verwendet um alle empfangenen Datensätze ungeachtet ihres Inhaltes in einem Topic zu speichern. Für dieses Projekt relevante Informationen wurden dann in die PostgreSQL Datenbank übertragen und konnten dann von dort mittels Apache Zeppelin und Jupyter ausgewertet werden.

Aufgrund der limitierten Größe unseres Datensatzes und der Rechenleistung des verfügbaren Equipments in unserem Projekt hätte auf Apache Kafka verzichtet werden können, sodass die Daten direkt aus der API in die Datenbank geschrieben werden ohne den „Umweg“ über Apache Kafka.
Jedoch ist in einem realen Szenario dieser Weg über Apache Kafka keineswegs falsch.
Dieser Ansatz ermöglicht es in der Datenbank nur diejenigen Daten vorzuhalten, die für die Beantwortung einer konfirmatorischen Frage notwendig sind und trägt damit zu einer besseren Performance bei. Zudem ist es leichter die Ausfallsicherheit und damit Verfügbarkeit der Daten in einem Kafka Cluster zu erreichen als in einem relationalen DB Cluster. Darüber hinaus kann mit der Verwendung von Kafka Streams das original Topic gezielter angepasst und erneut wiedergeben werden.

Kafka Streams ist eine Klient Bibliothek zur Verfügung gestellt von Confluent. Mit dieser Bibliothek können Anwendungen und Microservices erstellt werden, die eingehende Daten eines Topics auslesen, weiterverarbeiten und in zweites, vor allem separates, Topic schreiben. Kafka Streams bieten, wie alle Komponenten des Kafka Ökosystems, den Vorteil, dass sie untereinander hoch integriert, hoch skalierbar und fehlertolerant sind.
Die dafür empfohlenen Programmiersprache von Konfluent ist Java, jedoch gibt es auch Implementierungen für andere Sprachen wie z. B. Python.[1]

[1] Confluent. *Kafka Streams*. 2017. URL: https://kafka.apache.org/10/documentation/streams/ (besucht am 07. 02. 2018).

Consumer & Producer Implementierungen

Beim Vergleich der beiden Implementierung für jeweils Kafka Producer und Kafka Consumer fällt auf, dass die Java Implementierung eher mit low-level APIs operiert und damit mehr Aufwand für Konfiguration und Konvertierung von Daten anfällt, wohingegen Python durch Verwendung von Frameworks, die Komplexität stärker abstrahieren, und schlankerer Syntax mit deutlich weniger Codezeilen auskommt. Somit stellt Python für einen PoC bzw. Prototypen die schnellere Alternative da, wobei Java und sein feinerer Detailgrad eher in komplexeren Umgebungen seine Stärken ausspielen kann.
Aufgrund dessen, dass die Komplexität unserer Consumer und Producer insgesamt gering ist, da keine inhaltliche Transformation der Daten stattfindet, empfiehlt es sich für den produktiven Betrieb eher Kafka Connector zu evaluieren.
Kafka Connector sind ein Teil der Confluent-Plattform und bilden Connectoren zu gängigen Datenquellen und -senken. Dabei setzen sie auf deklarative Konfiguration anstatt individuelle und imperative Programmskripte. Zudem bieten sie Fehlertoleranz und bessere Skalierbarkeit als einzelne Consumer bzw. Producer Skripte.

Analysewerkzeuge

Obwohl Apache Zeppelin im Mai 2016 den Incubator Status verlies macht es im allgemeinen einen noch sehr unreifen und unfertigen Eindruck, was bei Versionsnummer 0.7 nichts überraschendes ist. Bemerkbar macht sich dies vor allem bei der Installation und Setup insbesondere unter Windows. So glückte der Start von Zeppelin auf nur 1 von 3 Windows PCs. Des weiteren stürze der JDBC Interpreter während des Betriebs mehrfach aus unerfindlichen Gründen ab, was nur durch einen Neustart behoben werden konnte. Zudem bietet Zeppelin 0.7 leider nur sehr rudimentäre Charting-Funktionalitäten. Zwar gibt es eine Vielzahl an externen Plug-Ins die via Zeppelins Erweiterungsframework *Helium* eingebunden werden können und erweiterte Diagrammtypen[2] bereitstellen, doch setzen diese Plug-Ins oft Zeppelin Version 0.8-*SNAPSHOT* voraus. Diese Entwicklungsversion kann man nur per Sourcecode beziehen und selber kompilieren bzw. bauen. Das klappte sogar nach mehreren Versuchen, jedoch ließ sich das fertige *binary* nicht starten.
Letztlich lässt sich zusammenfassen, dass die Konzepte hinter Apache Zeppelin gut sind, die konkrete Implementierung jedoch noch nicht ausgereift ist.

Die Installation und Erstkonfiguration mit Jupyter unter Windows ist in wenigen Minuten erledigt. Sobald Jupyter und die Magic-Commands mit dem Paketmanager *pip* installiert wurden können schon erste SQL Statements abgesetzt werden.
Resultate eines SQL Statements werden grafisch als Tabelle dargestellt. Für jegliche Art von Visualisierung die darüber hinaus geht müssen separate Bibliotheken installiert werden über die dann die Visualisierung erfolgt. Zusätzlich müssen *Widgets* installiert und aktiviert

[2] z.B. Kartendienste und Heatmaps, siehe https://zeppelin.apache.org/helium_packages.html

werden wenn die Grafik integriert in Jupyter angezeigt werden soll.
Auch hier gibt es Einschränkungen: Falls kein Widget für die aktuell genutzte Bibliothek zur Verfügung steht kann nur die Anzeigemöglichkeiten der Bibliothek genutzt werden.
Das dynamische Ändern eines SQL Statements mit einem Textfeld - so wie es in Apache Zeppelin erfolgen kann - benötigt ihn Jupyter zwar zusätzlichen Implmentierungsaufwand kann aber letztlich genauso wie in Apache Zeppelin erfolgen.
Die *Magic-Commands* sind sowohl in Apache Zeppelin als auch Jupyter eine sehr gute Möglichkeit um kleine Code Snippets mit `SQL` oder `R` zu Prototypen ohne tief in die Programmierung einzusteigen. Alles darüber hinaus wie komplexe Visualiserungen oder Manipulation der Ergebnisse erfordert die Nutzung von weiteren Bibliotheken und Visualisierung können nur bedingt in Apache Zeppelin und Jupyter angezeigt werden.

In unserem Projekt haben wir die Erkenntnis gewonnen, dass die virtuellen Notebooks für schnelles Prototyping sehr gut geeignet sind. Will man aber das Notebook so anpassen, dass z. B. dynamisch Daten von einer externen Datenquelle abgerufen werden und sich die genutztes Charts kontinuierlich anpassen - Stichwort Real-Time Monitoring - dann ist wesentlich mehr Aufwand nötig und man stellt sich die Frage ob die virtuellen Notebooks dafür noch geeignet sind und nicht für dieses Szenario ein anderes oder weitere Tools benötigt werden. In diesem Umfeld sind vor allem die Open Source Werkzeuge Kibana von Elastic und Grafana zu nennen.
Die Erstellung eines Monitoring-Dashboards mit Echtzeit Metriken in Kombination mit einem Notebook für explorative Analytik birgt weitere interessante Ansätze und Überlegungen die jedoch, mit Hinblick auf das Projektziel, nicht weiter verfolgt wurden.

Abkürzungsverzeichnis

JSON JavaScript Object Notation

CSV Comma-separated values

API Application Programming Interface

SODA Socrata Open Data

SQL Structured Query Language

Abbildungsverzeichnis

Literatur

Confluent. *Kafka Streams*. 2017. URL: https://kafka.apache.org/10/documentation/streams/ (besucht am 07. 02. 2018).

– *Python*. 2017. URL: https://cwiki.apache.org/confluence/display/KAFKA/Clients#Clients-Python (besucht am 05. 02. 2018).

Cristina. *sodapy*. 2017. URL: https://github.com/xmunoz/sodapy (besucht am 05. 02. 2018).

Data, NYC Open. *List of most popular datasets*. 2017. URL: https://data.cityofnewyork.us/browse?provenance=official&sortBy=most_accessed&utf8=%E2%9C%93 (besucht am 06. 02. 2018).

– *Our mission: open data for all*. 2017. URL: https://opendata.cityofnewyork.us/overview/ (besucht am 03. 02. 2018).

Devlin, Catherine. *ipython-sql*. 2017. URL: https://github.com/catherinedevlin/ipython-sql (besucht am 06. 02. 2018).

Powers, Dana. *kafka-python*. 2017. URL: http://kafka-python.readthedocs.io/en/master/ (besucht am 05. 02. 2018).

Socrata. *Obtaining an Application Token*. 2017. URL: https://dev.socrata.com/docs/app-tokens.html (besucht am 05. 02. 2018).