Vergleich von Kafka Streams und Spark Streaming

Studienarbeit

des Studienganges Angewandte Informatik

an der Dualen Hochschule Baden-Württemberg Mannheim

von

Dominik Heßmer

28.05.2018

Bearbeitungszeitraum	12 Wochen
Matrikelnummer, Kurs	1690631, TINF15 AI-BI
Ausbildungsfirma	Atos Information Technology GmbH
Betreuer der Dualen Hochschule	Prof. Dr. Rainer Colgen

Erklärung zur Eigenleistung

Ich versichere hiermit, dass ich meine Studienarbeit mit dem Thema: *Vergleich von Kafka Streams und Spark Streaming* selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Ich versichere zudem, dass die eingereichte elektronische Fassung mit der gedruckten Fassung übereinstimmt.

25.5.18 Paderborn

Ort, Datum

D. Hebmer

Unterschrift

Abstract

Viele Applikationen und Anwendungsfälle bauen auf Streaming Applikationen auf. Dabei werden Daten in einem ununterbrochenen Strom von einer Datenquelle, in der die Daten entstehen, zu einer Datensenke, in der die Daten zur Lagerung oder Auswertung bereit stehen. Dabei werden die Rohdaten aus der Quelle oft nicht im Originalzustand in die Datensenke geschrieben. Meistens werden die Daten noch Verarbeitet. Für diese Anwendungsfälle ist die sequenzielle Verarbeitung der Daten von Vorteil. Mit Kafka Streams ist eine solche sequenzielle Verarbeitung gegeben. In dieser Arbeit wurde die Funktionsweise von Kafka Streams anhand eines beispielhaften Use Case erläutert. Dafür wurden Daten aus dem Social Media Netzwerk Twitter gezogen. Diese Daten wurden in eine Kafka Topic überführt, auf welchem mit der Kafka Streams API eine einfache Verarbeitungslogik definiert wurde.

Inhaltsverzeichnis

Tabellen- und Abbildungsverzeichnis

Kapitelüberblick

Im Folgenden wird ein kurzer Überblick über die Kapitel gegeben.

Das Komplette erste Kapitel beschäftigt sich mit den Theoretischen Grundlagen zum Thema Streaming.

Kapitel 1.1 Definiert den Data Stream zeigt sowohl Probleme bei der Stream Verarbeitung auf und erläutert eine generische Architektur für Streaming.

Das Kapitel 1.2 beschäftigt sich mit der Stream Processing Engine. Dabei werden unterschiedliche Modelle vorgestellt und erläutern die einzelnen Komponenten der Engine.

Das Kapitel 1.3 erläutert Stream Verarbeitungsmodell und stellt dabei zum einen das Cash Register Modell vor, zum anderen das Turnstile Modell.

In dem Kapitel 1.4 wird Kafka diskutiert. Zunächst wird ein allgemeiner Überblick über die Kafka Architektur gegeben. Anschließend wird Kafka Streams in den Konzepten und der Architektur vorgestellt.

Das Zweite Kapitel beschäftigt sich mit der Umsetzung der Aufgabenstellung.

Zunächst wird in 2.1 die Datenverbindung nach Twitter beschrieben.

Kapitel 2.2 beschäftigt sich mit der Datenübertragung nach Kafka.

Im Kapitle 2.3 wird die Konfiguration der Applikation diskutiert.

Das letzte Unterkapitel zeigt die Kafka Streams Applikation, welche auf dem Topic definiert ist.

Aufgabenstellung

Die Aufgabenstellung bestand darin, einen Anwendungsfall für eine Streaming Architektur auf einer OpenStack Umgebung zu implementieren. Die Virtuellen Maschinen sind Linux Betriebssysteme, mit der Ubuntu Distribution. Daten sollten von einer beliebigen Quelle über die beiden Streaming Plattformen Kafka und Scala geleitet werden. Dabei war der Hauptaugenmerk auf die Streaming-Verarbeitung von Kafka und Scala angesetzt. Durch eine mangelhafte Zeitplanung ist es zeitlich nicht mehr möglich gewesen, den Streaming Anwendungsfall mit Scala umzusetzen.

1 Theoretische Grundlagen

Im Folgenden werden die theoretischen Grundlagen für die Arbeit geschaffen. Zunächst werden die Data Streams erläutert. Dabei wird auf die Struktur und Definition von Data Streams eingegangen. Des Weiteren wird die TCS-Architektur vorgestellt. Anschließend wird die Stream Processing Engine, mit 3 unterschiedlichen Modellen vorgestellt.

1.1 Data Streams

Ein Data Stream S ist eine geordnete Sammlung von Data Items s_1, s_2,… mit den folgenden zwei Eigenschaften.

1. Data Items werden kontinuierlich von einer oder mehreren Quellen generiert und an eine oder mehrere Verarbeitungsobjekte.
2. Die Anordnung, in welcher die Data Items gesendet wurden, kann nicht durch die Verarbeitungsobjekte bestimmt werden.

Data Streams können in zwei Kategorien eingeteilt werden, Basis Streams und abgeleitete Streams. Als Basis Stream werden Streams bezeichnet, welche ihren „Ursprung“ direkt aus der Datenquelle haben. Abgeleitete Streams sind der Wortbedeutung nach Basis Streams, welche schon zuvor verarbeitet wurden (1 Liu und Özsu 2009, S. 638).

Die Data Items können je nach Anwendungsgebiet aus Datenpaketen, Rohtext, Events, Tupel oder komplexere Datenstrukturen, wie XML oder JSON bestehen. Des Weiteren besitzen die Data Items in der Regel zwei Timestamps. Ein Timestamp der Erzeugung des Objektes und ein Timestamp bei der Ankunft am Verarbeitungsobjekt.

Die Schwierigkeiten beim Verarbeiten von Data Streams liegen in weiteren Eigenschaften. Data Streams können eine sehr hohe Frequenz haben und ihre Frequenz kann mit der Zeit variieren. Zudem kann die Verarbeitungslogik nicht vorhersehen, wann der Stream aufhört, wobei dieser theoretisch unendlich bestehen kann (1 Liu und Özsu 2009, S. 638).

Die hohe Frequenz der Data Streams wirkt sich vor allem auf die Datenübertragung, die Verarbeitung und die Speicherung der Daten aus. Auf English *transmit* (T), *compute* (C) und *store* (S), kurz TCS. Bei der Datenübertragung könnte die Verbindung langsam oder instabil werden. Dabei kann es zwar zu Verzögerungen der Daten kommen, jedoch werden die Daten früher oder später an die Verarbeitungslogik übertragen. Bei der Verarbeitung der Daten kann es je nach Komplexität und Rechenleistung zu langen Abfragen kommen, jedoch sollte die Verarbeitung im Prinzip funktionieren (2 Muthukrishnan 2005, S. 6).

Die Datenspeicherung macht bei der Verarbeitung von Streams die wenigsten Probleme, da die Skalierung von Speicherplatz über dafür spezielle Datenstrukturen schon erforscht sind (3 Vitter 2001, S. 209–271).

Die Herausforderungen an die TCS Architektur besteht durch automatisch generierten und hochdetaillierten Datenströme. Im Verbund mit der Analyse dieser Datenströme entstehen besonders Anforderungen an die Datenübertragung (T) und Verarbeitung (C). Um diese Herausforderungen zu bewältigen gibt es einige Ansätze.

Ein weit verbreiteter und bekannter Ansatz ist die Parallelisierung. Die Verarbeitung und die Speicherung kann sehr gut parallelisiert werden. Beispielsweise kann durch das Clustern der Verarbeitungslogik und der Festplatten die Kapazität erhöht werden. Die Datenübertragung zur Verarbeitungslogik kann nicht so gut über Parallelisierung verbessert werden. Da eine aktive Verbesserung der Datenübertragungsrate schwer zu erreichen ist, muss anders vorgegangen werden. Die Datenrate kann erhöht werden, indem der Datenstrom entweder durch kontrolliertes Sampling oder unkontrolliertes Shredding verkleinert wird. Ein weiterer Ansatz ist die hierarchisch gegliederte Analyse. Dabei werden zunächst schnelle, dafür aber simple Analysen wie Filterung oder Aggregation durchgeführt. Auf den höheren Leveln werden anschließend komplexere Analysen auf kleineren Datensätzen gefahren (2 Muthukrishnan 2005, S. 7).

1.2 Stream Processing Engine

Die erste generelle, relationale Stream Processing Architektur wurde 2001-2002 entwickelt (1 Liu und Özsu 2009, S. 639). Eine Stream Processing Engine (SPE) unterscheidet sich von einer traditionellen Datenbank Engine in 3 Charakteristika.

1. Continuous Query Modell
2. Inbound Processing Modell
3. Single Process Modell

Beim continuous Query Modell, werden die Abfragen ausgeführt sobald neue Daten zur Verfügung stehen. Dies steht im Gegensatz zum one-time Query Modell, bei dem die Abfragen von den Clients abgesetzt werden und nur ein Ergebnis produzieren.

Beim continuous Query Modell sind die Abfragen persistent und die Daten sind vergänglich. Beim traditionellen Modell sind die Daten persistent und die Abfrage sind vergänglich, (1 Liu und Özsu 2009, S. 639). Ein Vergleich zwischen traditionellem und Stream Processing ist in Abbildung 1 gegeben.

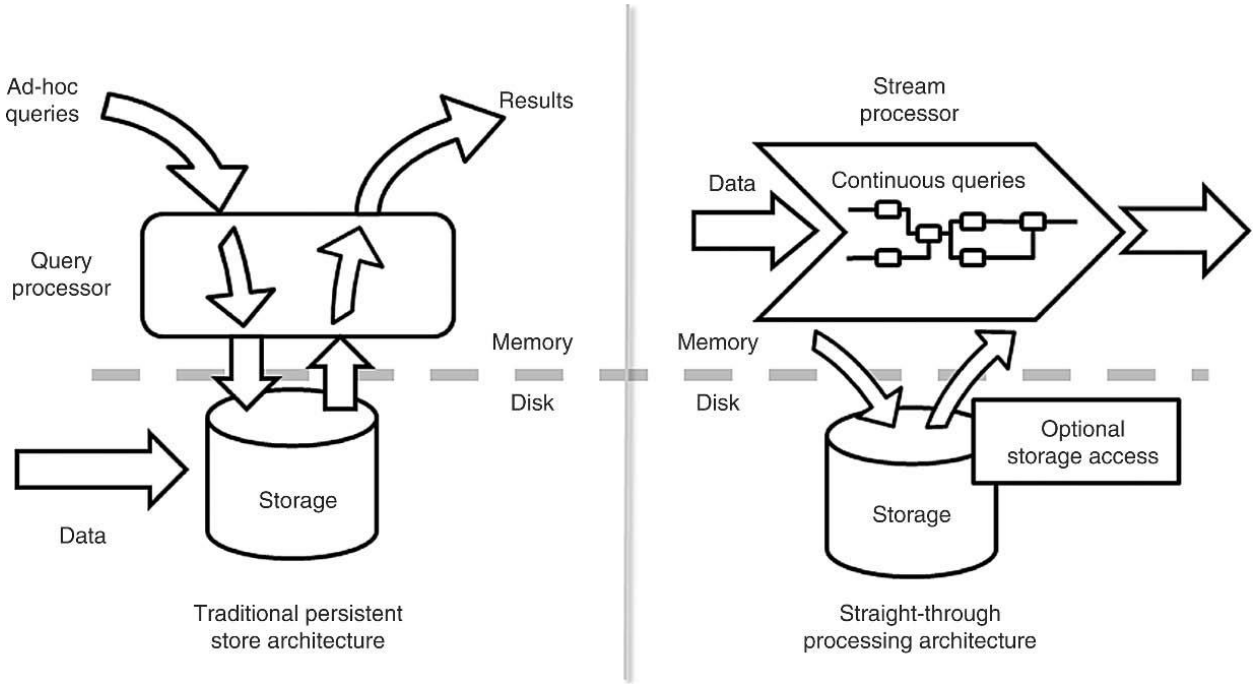

Abb.: 1 Vergleich traditionellem und Stream Processing Modell

Inbound Processing bedeutet, dass eingehende Streams sofort von der Verarbeiteten Logik bearbeitet werden, wenn sie im System ankommen. Dabei werden die Nachrichten ständig verarbeitet und produzieren ständig Ergebnisse. Dies geschieht soweit es geht im Hauptspeicher, wobei Schreiben und Lesen in den Speicher optional und meistens asynchron geschehen. Dadurch das für die Verarbeitung die Daten nicht in den Storage geladen werden müssen, sondern im Hauptspeicher verarbeitet werden,

kann ein erheblicher Performancevorteil gegenüber konventionellen Datenbanken erzielt werden (1 Liu und Özsu 2009, S. 639).

Beim Single Process Modell hingegen wird ein spezieller Verarbeitungsbereich reserviert. In diesem Bereich werden alle zeitkritischen Operationen durchgeführt. Dadurch werden Kontextwechsel, welche mit viel Overhead verbunden werden vermieden (1 Liu und Özsu 2009, S. 639).

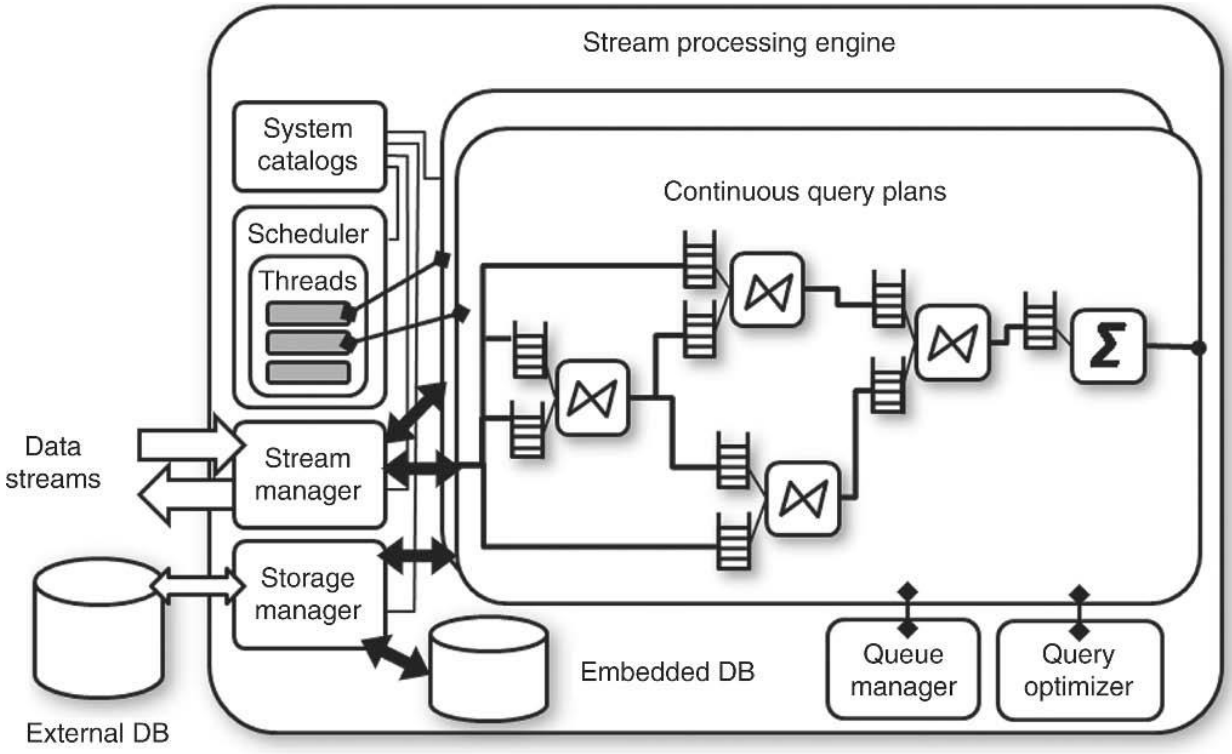

Abb.: 2 Stream Processing Engine

Abbildung 2 stellt die Kernkomponenten eines SPE dar. Der Queue Manager stellt die Speicherressourcen für Input und Output Buffering zu Verfügung. Des Weiteren kümmert er sich um alternative Verarbeitungstechniken, falls benötigte Speicherressourcen nicht garantiert werden können.

Der Storage Manager pflegt Zeiger auf externen Tabellen und führt asynchrone Lese- und Schreibezugriffe aus. Erweiterte Funktionen sind das approximieren der Streams und das Verlagern von Streams auf die Festplatte bei einer schwindenden Speicherverfügbarkeit. Ein Stream Manager interagiert mit der Netzwerkschicht, da die Daten oft über TCP/UDP Sockets gesendet werden. Er ist auch dafür Zuständig, neue

Inputs an den Scheduler zu melden und die Umwandlung von Datenformaten durch integrierte Adapter. Schließlich beinhalten SPE noch Query Optimierer, wie zum Beispiel adaptive Plan-Optimierer (4 Valduriez 2004, S. 407–418). Er überwacht den Status der ausführenden Abfragen in Abhängigkeit von Statistiken um dynamisch vorteilhafte Modifikationen an Ausführungsplänen zu machen.(1 Liu und Özsu 2009, S. 641).

1.3 Data Stream Modells

Die Data Items, aus denen der Data Stream zusammengesetzt wird, beschreiben meist ein grundlegendes Signal. Beispielsweise beschreibt ein Netzwerk Stream die Art und die Menge an Daten, welche zwischen Netzwerkknoten transportiert werden. Ein mögliches Signal dieses Streams wäre das Mapping zwischen den Quell- und Ziel-IP-Adressen zu der Anzahl an übertragenen Bytes. Das Stream Model beschreibt, wie dieses Signal aus den Daten rekonstruiert werden kann. Meistens ist die Komplexität eines Data Stream abhängig von der Komplexität des zugrundeliegenden Stream Modells (1 Liu und Özsu 2009, S. 2834).

Für die Folgenden Modelle sei der Input Stream definiert durch Data Items a_1, a_2, die sequentiell bei der SPE ankommen. Der Stream beschreibt ein Signal A. A ist eine Funktion:

$$A: [1..N] \rightarrow \mathbb{R}^2$$

Die unterschiedlichen Modelle unterscheiden sich darin, wie die Data Items a_i das Signal A beschreiben. In dieser Arbeit sollen das Cash Register Model und das Turnstile Model beschrieben werden.

1.3.1 Cash Register Model

Im Cash Register Model sind die Data Items a_i inkrementieren das Signals $A[j]$. In diesem Modell ist a_i ein Tupel der Form

$$a_i = (j, I_i), \qquad mit\ I_i \geq 0$$

Die Aktualisierung des Signals wird durch

$$A_i[j] = A_{i-1} + I_i$$

Beschrieben wird.

A_i beschreibt den Signalzustand nachdem das i-te Element des Stromes gesehen wurde (2 Muthukrishnan 2005, S. 9). Zur Erläuterung des Modells dient das folgende Beispielproblem. Die Fragestellung, wie viele verschiedene IP Adressen haben in bestimmten Anschluss verwendet, kann mit dem Cash-Register Modell folgendermaßen gelöst werden.

$a_1, a_2 \ldots$: eine Folge ankommender IP-Pakete a_i.

S_i: Source-Adresse, welche jedes Data Item im Payload mit überträgt.

$A[0 \ldots N-1]$: Anzahl der von S_i aus gesendeten Pakete.

Mit dieser Definition ist die Anzahl der verschiedenen IP Adressen gegeben durch:

$$\sum_{i=1}^{N-1} A[i] > 0 \,.$$

1.3.2 Turnstile Model

Das Turnstile Model ist eine Generalisierung des Cash-Register Model. Im Turnstile Model sind die Data Items a_i Updates des Signals $A[j]$. Die Items a_i sind Tupel der Form:

$$a_i = (j, U_i)$$

Hierbei können die Updates auch negative Werte annehmen, daher der Unterschied zum Cash-Register Model. Das Signal $A[j]$ wird, genau wie beim Cash-Register Model durch die Formel:

$$A_i[j] = A_{i-1} + U_i$$

beschrieben.

1.4 Kafka

Kafka ist eine von der Apache Software Foundation entwickelte Streaming Plattform. Es wurde in Java und Scala geschrieben und im Januar 2011 veröffentlicht.

Kafka speichert Daten in einer skalierenden und verteilten Message Queue. Eine Queue ist ein First-In-First-Out Datenkonstrukt. Kafkas Architektur orientiert sich an einem Transaktionslog, wie er in Datenbanken auftritt (5 Apache Kafka Kap. 4.1).

1.4.1 Kafka Architektur

Im Folgenden wird die Architektur von Kafka erläutert. Dabei wird sich auf die, für das Projekt, wichtigsten Funktionen konzentriert.

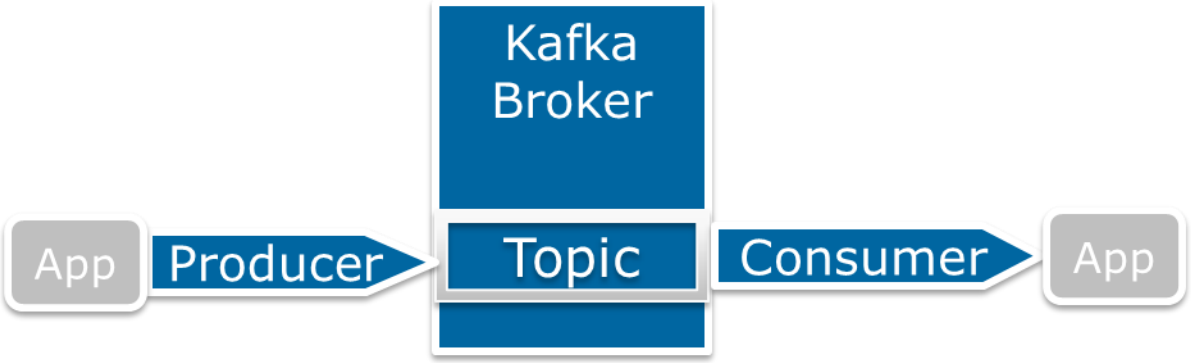

Abb.: 3 Kafka Architektur Überblick

Kafka besteht aus einem Kafka Broker, dies ist der Server auf dem der Service implementiert ist. Auf diesem können Topics definiert werden. Ein Topic ist ein Container in den Messages reingeschrieben und wieder rausgelesen werden können. Durch das Topic können Message Queues thematisiert werden. Das Topic kann mit mehreren Partitions kreiert werden. Jede Partition des Topics wird auf einem anderen Kafka Broker abgespeichert. Hält das Cluster mehrere Kafka Server, so kann dadurch die Last auf das Cluster verteilt werden. Weiterhin besitzt das Topic noch Eigenschaften wie die Retention Time und die Retention Byte. Mit diesen kann eine Obergrenze für Messages festgesetzt werden, je nachdem ob es zu viele werden oder sie sich zu lange in dem Topic aufhalten. Ist einer dieser Schwellwerte erreicht, werden die ältesten Messages in der Queue gelöscht.

Der Producer schreibt Messages in das Topic hinein. Er verbindet sich über den Broker, an den richtigen Port an das dementsprechende Topic und hinterlegt dort eine Message, mit einer eindeutigen ID.

Der Consumer verbindet sich ebenfalls über den Broker an das Topic. Anschließend fragt er zyklisch das Topic nach neuen Nachrichten (Polling) und liest diese aus. Der Consumer speichert seinen Offset, ein Pointer auf den Index der letzten gelesenen Message, im Zookeeper. Damit kann der Consumer, zum Beispiel nach einem Absturz, wieder dort anfangen zu lesen, wo dieser unterbrochen wurde. Werden Consumer zu einer sogenannten Consumer Group zusammengefasst, so werden die Messages eines Topics in dieser Consumer Group aufgeteilt.

An einem Topic können sich mehrere Producer und Consumer verbinden. So können Daten aus unterschiedlichen Quellen an einem zentralen Service für unterschiedliche Anwendungen bereitgestellt werden (6 Kafka Architecture 2018).

1.4.2 Kafka Streams

Kafka Streams ist eine Client Bibliothek, für das Verarbeiten von Daten aus Kafka Topics. Nach Verarbeitung der Daten werden diese wieder in das Topic reingeladen.

Ein Stream repräsentiert eine unbegrenzte, ständig aktualisierende Datenmenge. Der Stream ist geordnet und zurückspielbare Sequenz von Data Items. Ein Data Item ist definiert als ein Key-Value Paar.

Als eine Stream Processing Applikation wird jedes Programm bezeichnet, welches die Kafka Streams Bibliothek verwendet. Dabei werden Processor Topologien verwendet. Eine Processor Topologie kann als ein Graph aus Edges und Nodes dargestellt werden. Dabei sind die Nodes die Stream Processors und die Edges die Streams.

Der Stream Processor wiederum ist der eigentliche Schritt in der Verarbeitungslogik in der die Daten bearbeitet werden. Der Stream Processor verarbeitet ein Data Item nach dem anderen aus seinen Upstream Processors und produziert dabei ein oder mehrere Data Items für seine Downstream Processors.

In der Topologie einer Stream Applikation gibt es zwei Arten von Spezial Processorens.

Der Source Processor ist ein Stream Processor, der keinen Upstream Proceesor vorgeschaltet hat. Er produziert selbst einen Input Stream aus einem Kafka Topic und leitet diesen an seine Downstream Processors weiter.

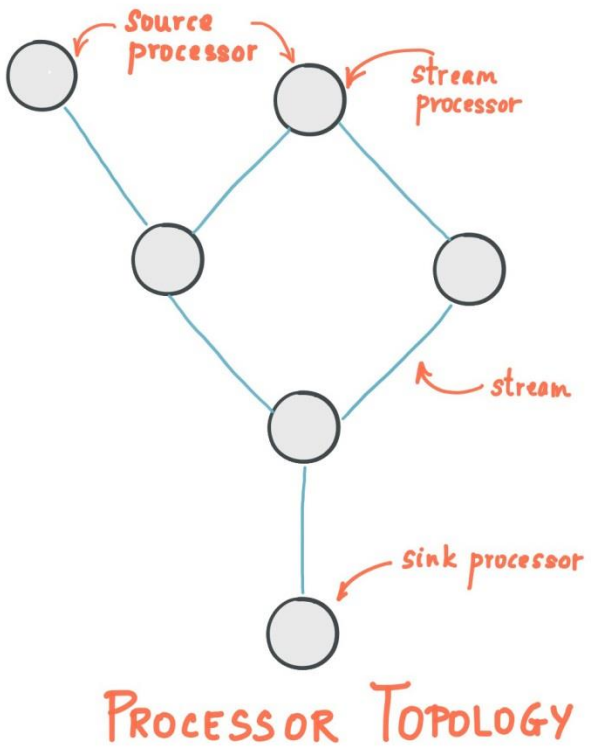

Abb.: 4 Kafka Streams Topologie

Der Sink Processor wiederum besitzt keine Downstream Processors. Er sendet alle Data Items direkt an ein Kafka Topic.

Ein weiterer Aspekt der Stream Verarbeitung und Architektur ist die Implementation der Zeitdimension. Kafka Streams definiert drei Unterschiedliche Zeitpunkte.

Event Time – Ist der Zeitpunkt an dem Event oder ein Data Item an der Quelle erstellt wurde.

Processing Time – Der Zeitpunkt an dem das Event oder der Data Item von der Stream Processing Applikation verarbeitet, bzw. konsumiert wird. Dieser Zeitpunkt kann Sekunden, Stunden oder Tage nach der Event Time sein.

Ingestion Time – Ist der Zeitpunkt an dem ein Event oder Data Item von dem Kafka Broker in einem Topic gespeichert wurde.

Für Anwendungsfälle wie die Aggregation oder das Gruppieren von Data Items bietet Kafka Streams sogenannte State Stores. In diesen können Daten gespeichert und abgefragt werden. Implementiert werden können die State Stores durch ein Key-Value Store oder, HashMap oder eine andere geeignete Datenstruktur.

1.4.3 Kafka Streams Architektur

Im Folgenden soll die Architektur eines Kafka Streams erläutert werden. Dabei wird ein genauerer Blick auf die Partitionierung und das Thread Model geworfen. Kafka Streams partitioniert Daten zur Verarbeitung. Kafka Streams nutzt für die Parallelisierung die Konzepte Partitionierung und Tasks. Zwischen dem Parallelen Modell von Kafka und Kafka Streams existieren einige Gemeinsamkeiten.

Jede Stream Partition ist eine geordnete Sequenz von Data Items und gleicht einet Topic Partition.

Ein Data Item in einem Stream entspricht einer Message in einem Topic.

Die Schlüssel der Data Items bestimmen die Partitionierung sowohl in Kafka, als auch in Kafka Streams.

Die Processor Topologie skaliert dadurch, dass sich die Architektur in mehrere Tasks aufteilt. Kafka Streams erstellt eine statische Anzahl an Tasks, basierend auf der Anzahl der Partitionen des Input Streams. Jeder Task bekommt eine Liste von Partitionen der Input Streams zugeteilt. Da sich die Zuteilung der Partitionen zu den Tasks nicht ändert, kann der Task eine Einheit für Parallelisierung der Applikation gesehen werden.

Der Task erstellt anschließend seine Partition Topologie und hält für jede Partition einen Input Buffer bereit. Die Data Items werden eins nach dem anderen abgearbeitet. Die Tasks können unabhängig voneinander und ohne manuelle Orchestrierung arbeiten.

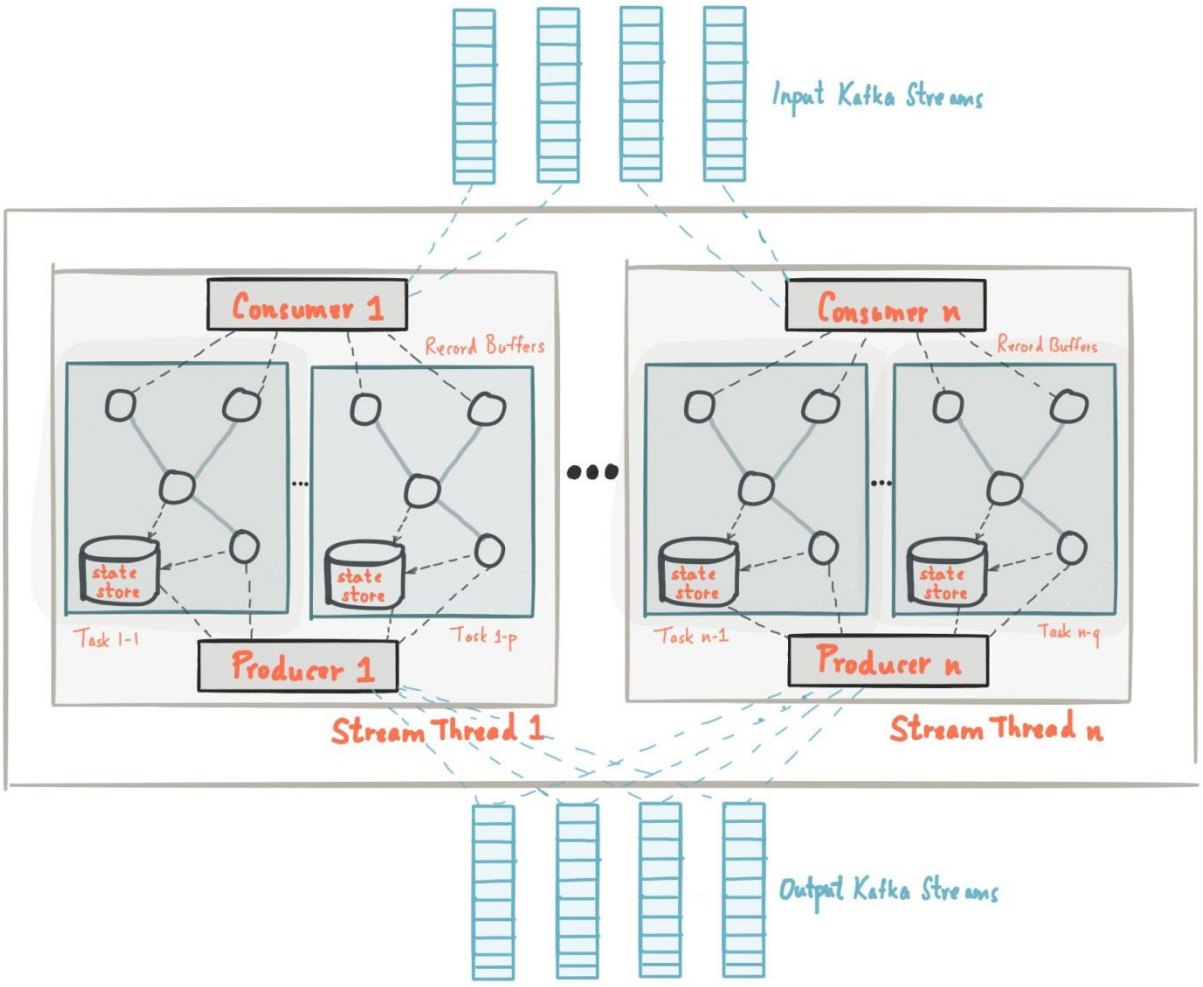

Abb.: 5 Kafka Stream Architektur

Ein weiterer Teil der Parallelisierung einer Kafka Stream Applikation sind die verschiedenen Threads. Die Anzahl der Threads, welche die Applikation benutzt kann konfiguriert werden. In einem Thread können dabei mehrere Tasks ausgeführt werden, wie in der Folgenden Grafik dargestellt ist.

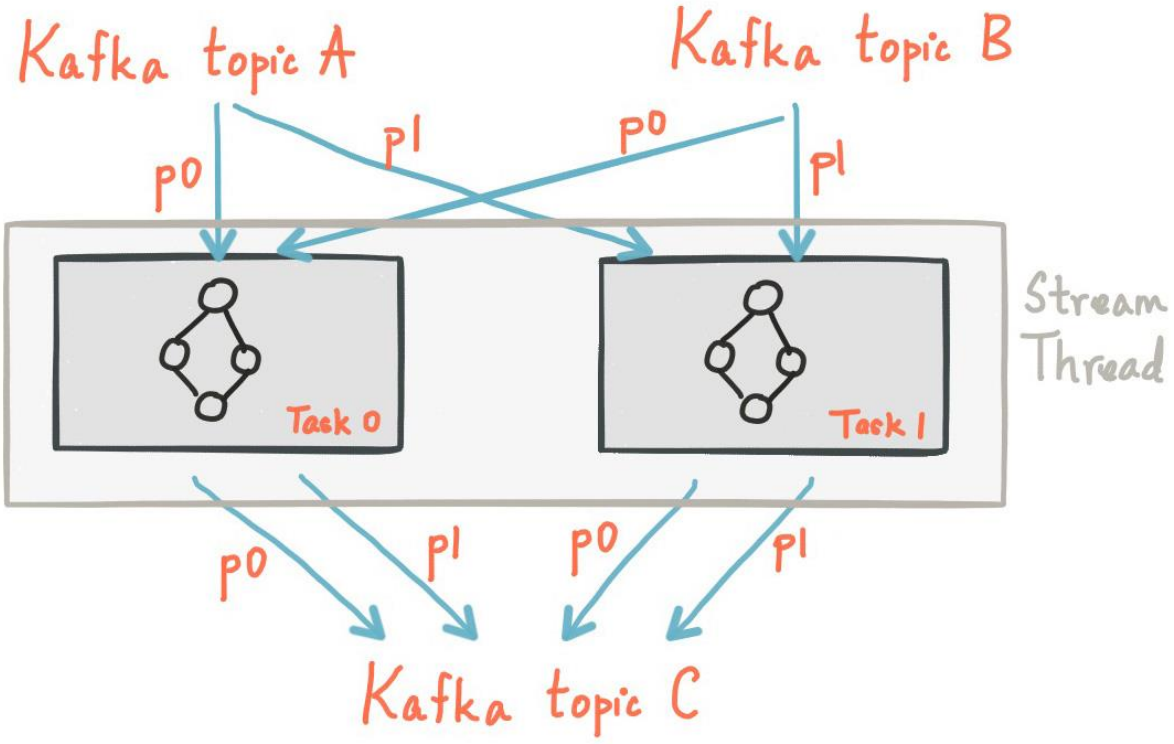

Abb.: 6 Kafka Streams Thread Modell

Das Starten eines weiteren Threads kopiert die Topologie, welche eine Teilmenge des Input Streams verarbeitet. Die Zuteilung von Kafka Topics auf die verschiedenen Threads wird von Kafka intern gemanagt.

Zum Skalieren einer Applikation kann eine weitere Instanz der Applikation gestartet werden. Die Verteilung der Partitionen auf die unterschiedlichen Instanzen wird von Kafka Streams übernommen. Die Anzahl der Threads pro Applikation ist begrenzt durch die Anzahl der Partitionen des Input Streams, sodass jeder Thread mindestens eine Partition verarbeiten kann.

Die zuvor genannten State Stores sind in den Tasks eingebettet. Ein Task kann ein oder mehrere Stores unterhalten. Diese können über eine API angesprochen werden. Die Stores sind Fehlertolerant und haben automatische Recovery (7 Apache Kafka).

2 Praktische Umsetzung

In diesem Kapitel soll die praktische Umsetzung besprochen werden. Dabei werden die Programme und Architekturen besprochen, welche in der Arbeit entwickelt wurden.

Für die Umsetzung der Arbeit wurde im Standalone durchgeführt. Dies bedeutet, dass Kafka nicht als Cluster, sondern nur auf einem Server benutzt wurde.

2.1 Datenbeschaffung

Um die Daten für den Kafka Stream zu beschaffen musste eine kontinuierliche Datenkette zur Verfügung stehen. Als Lösung wurde hier Twitter gewählt. Über die Twitter API wurden Twitter Nachrichten (Tweets) gelesen. Um die Twitter API lesen zu können wurde eine Twitter Applikation erstellt. Die Twitter-Applikation ist mit einem Twitter Account verknüpft. Die Applikation enthält einen ein Consumer Key und ein Consumer Secret. Diese sind notwendig, damit sich die Applikation an der Twitter API registrieren kann. Ohne den Consumer Key ist eine Anwendung nicht in der Lage Twitter Nachrichten zu lesen. Des Weiteren enthält die Anwendung ein Access Token und ein Access Token Secret. Diese beiden Anmeldeinformationen sind auf Userebene definiert. Ein Benutzer der Applikation besitzt ein Access Token und kann so auf die Funktionalitäten eben dieser zugreifen.

Um mit der Twitter API zu kommunizieren und die Twitter Nachrichten zu lesen wurde ein Java HTTP Client genommen. Der Hosebird Client ist ein von Twitter entwickelter Client.

```
36    public BasicClient build(BlockingQueue<String> msgQueue) {
37        StatusesFilterEndpoint hosebirdEndpoint = new StatusesFilterEndpoint();
38        if (followings.isEmpty()) {
39            hosebirdEndpoint.trackTerms(terms);
40            log.warn("followings empty");
41        } else if (terms.isEmpty()) {
42            hosebirdEndpoint.followings(followings);
43            log.warn("terms empty");
44        } else {
45            hosebirdEndpoint.followings(followings);
46            hosebirdEndpoint.trackTerms(terms);
47        }
48        ClientBuilder builder = new ClientBuilder().name(this.getClientName())
49                .hosts(new HttpHosts(this.getStreamHost()))
50                .authentication(new OAuth1(this.getConsumerKey(), this.getConsumerSecret(), this.getToken(),
51                        this.getTokenSecret()))
52                .endpoint(hosebirdEndpoint).processor(new StringDelimitedProcessor(msgQueue));
53        return builder.build();
54    }
```

Abb.: 7 Verbindung zum Twitter Endpoint

Die Klasse TwitterClient.java stellt den Client bereit, über den die Twitter API angesprochen wird. Dazu ist in der obigen Abbildung die Methode abgebildet, über welche der Client gebaut wird. Dieser bekommt als Parameter in Zeile 50 die Authentifikationen der Applikation mit. Ferner bekommt der Client den Host, über welchen er die Tweets abgreifen kann (Zeile 49) und den hoseBirdEndpoint in Zeile 52. Über den Endpoint können die Tweets, welche der Client herunterladen soll, gefiltert werden. Die Tweets werden anschließend in einer Message-Queue gespeichert.

```
35        while (!hosebirdClient.isDone()) {
36            String msg = null;
37            try {
38                msg = msgQueue.take();
39            } catch (InterruptedException e) {
40                log.error("Error when taking a Message out of the Queue", e);
41            }
```

Abb.: 8 Abgriff der Nachrichten aus einer Message Queue

In der Klasse TwitterStreamConn.java wird der Client als hosebirdClient referenziert. Die While-Schleife in Zeile 35 läuft solange, wie der Client arbeitet. Anschließend wird in Zeile 38 eine Nachricht aus der Queue herausgenommen. Die Nachricht befindet sich im JSON Format.

```
42              ObjectMapper mapper = new ObjectMapper();
43              ObjectNode object = new ObjectNode(null);
44              try {
45
46                  object = (ObjectNode) mapper.readTree(msg);
```

Abb.: 9 Überführen der Nachrichten in ein Objekt

Die Nachricht aus der Queue wird in ein Objekt hineingelesen. Dies geschieht in Zeile 46. Durch das Deserialisieren der Nachricht, kann diese besser Verarbeitet werden. Denn die Nachrichten werden im Anschluss gekürzt, um den Datendurchsatz klein zu halten.

```
56 if(object != null) {
57     if (object.get("created_at") == null || object.get("id") == null) {
58             continue;
59         }
60         log.debug("Original Tweet " + object.toString());
61         object.remove(bigPayloadFields);
62         log.debug("Cutted Tweet: " + object.toString());
63
```

Abb.: 10 Abtrennen der überflüssigen Tweet-Objekte

Anschließend wird geprüft, ob der Tweet mindestens die Felder „id“ und „created_at“ besitzt, da diese Felder dazu benutzt werden, den Tweet in das Kafka Topic hineinzuladen. Besitzt der Tweet diese Felder nicht, so wird er verworfen und der nächste Tweet wird aus der Queue gepullt (Zeile 58). In Zeile 61 werden Felder des Tweets, welche wenig Interessante Informationen bieten, durch Redundanz viel Bandbreite verbrauchen würden, abgeschnitten.

```
69 if (kafkaProducer != null) {
70     kafkaProducer.putMessage(object.get("id").toString(), object.toString());
71     totalMsgCount++;
72 }
```

Abb.: 11 Senden der Nachrichten an den KafkaProducer

In Zeile 70 werden der Tweet und seine ID der „putMessage“ Methode des kafkaProducers als Parameter mitgegeben. Der kafkaProducer soll als nächstes Besprochen werden.

2.2 Datenübertragung nach Kafka

Der Producer, welcher die Tweets als Messages in ein Kafka Topic schreibt wird über eine Methode in der TwitterStreamConn-Klasse angesprochen.

```
27  public void putMessage(String key, String message) {
28      producer.send(new ProducerRecord<String,String>(this.getTopic(),key,message));
29      log.debug("Twitter Message Object "+ key + "send to Kafka");
30  }
```

Abb.: 12 Senden der Nachrichten an das Kafka Topic

Die Methode putMessage bekommt zwei Strings, der eine String ist der Key der Message und der andere String ist die Message selbst. Der Key ist die ID des Tweets, diese ID wird von Twitter für jeden Tweet erstellt. Damit ist die eindeutige Zuordnung von Tweet und Kafka Message gewährleistet. In der Zeile 28 wird die Nachricht dann an das entsprechende Topic gesendet. Der Producer wird mit einem Properties Objekt initialisiert. In dem Properties Objekt steht, unter anderem, das Topic, an welches die Nachrichten gesendet werden. In diesem Fall heißt das Topic twitter_dh. Die Parameter hierfür werden durch eine Konfigurationsdatei mitgegeben. Wie das Konfigurationsmanagement aussieht, soll im Folgenden besprochen werden.

2.3 Konfiguration

Die Konfigurationsdatei wird im JSON Format erstellt und enthält alle Variablen Konfigurationen der Applikation. Ein Beispiel für eine Konfigurationsdatei ist in der folgenden Abbildung gegeben.

```
{
    "name": "ProjectConfigurations",
    "twitterClient": {
        "name": "dhbw.twitterConn.TwitterClient",
        "parameters": [
            {"key": "client.name", "value": "TwitterClient1"},
            {"key": "client.consumer.key", "value": "G6gJYLHP3qxvT6lQT1UtzfAsm"},
            {"key": "client.consumer.secret", "value": "vr5FSQUdXOJHb2npjJE9xyxN4m0qMkIbmhiZGVdZK5wIeM8Obp"},
            {"key": "client.token", "value": "3834719313-W1e7tddRJ35TgFaLlD8BueaczKNprK7U0QRCTna"},
            {"key": "client.token.secret", "value": "dbbUi2bFdXsniK9nQkEjciqU4cC4FH6ewZnyLEZaWRCvR"},
            {"key": "client.http.hosts", "value": "https://stream.twitter.com"}
        ],
        "followings": [

        ],
        "terms": [
            "api",
            "twitter"
        ]
    },

    "producer": {
            "name" : "dhbw.kafkaConn.Producer",
        "parameters" : [
            { "key" : "bootstrap.servers", "value" : "141.72.191.168:9092" },
            { "key" : "key.serializer", "value" : "org.apache.kafka.common.serialization.StringSerializer" },
            { "key" : "value.serializer", "value" : "org.apache.kafka.common.serialization.StringSerializer" },
            { "key" : "topic", "value" : "twitter_dh" }
        ]
    }
```


Abb.: 13 Auszug aus Konfigurations-Datei

Die Konfigurationsdatei bekommt einen Namen, in diesem Fall „ProjectConfigurations“. Weiterhin enthält die Konfigurationsdatei Objekte, diese Objekte bilden die konfigurierbaren Klassen in der Applikation wieder. In dieser Konfigurationsdatei sind die Objekte TwitterClient und Producer vertreten. Ein Objekt bekommt ebenfalls einen Namen. Anders wie bei dem Namen der kompletten Konfiguration sind die Namen der einzelnen Objekte nicht frei wählbar. Der Name eines Objektes ist der Klassenpfad der Klasse, für welche das Objekt Konfigurationselemente bereithält. In der klassischen Datenbanktheorie würde der Name eines Objektes als Fremdschlüssel bezeichnet werden. Die Objekte in einer Konfigurationsdatei bekommen, neben dem Namen, ein Array von Parametern. Die Parameter sind Key-Value Paare und bilden die eigentliche Konfiguration. Beispielsweise hat das Objekt producer die Parameter bootstrap.servers, key.serializer, value.serializer und topic. Diese vier Parameter sind notwendig, um den KafkaProducer erstellen zu können.

```java
public static Config importGenConsConfig(String fileName) {
    Config config;
    try {
        log.info("Read config from file: " + fileName);
        ObjectMapper mapperIn = new ObjectMapper();
        mapperIn.configure(DeserializationFeature.UNWRAP_ROOT_VALUE, false);
        Reader in = new BufferedReader(new InputStreamReader(new FileInputStream(fileName), "UTF-8"));
        config = mapperIn.readValue(in, Config.class);
        log.info("done importing config");
        return config;
```

Abb.: 14 Parsen der Konfigurationsdatei in ein Objekt

Die Klasse ConfigLoader lädt die Konfigurationsdatei und parsed diese in ein Config Objekt. In dem Config Objekt werden die Objekte für die Applikation erstellt. Die Referenz auf diese Objekte wird über entsprechende Getter und Setter vom Config Objekt geliefert.

```java
public void setProducer (ProducerWrapper producerWrapper) throws Exception{
    try {
        @SuppressWarnings("unchecked")
        Class<Producer> producerClass = (Class<Producer>) Class.forName(producerWrapper.getName());
        this.kafkaProducer = (Producer) producerClass.newInstance();
        this.kafkaProducer.setParameters(producerWrapper.getParameters());
        this.kafkaProducer.init();
    }catch(Exception e) {
        throw e;
    }
}
```

Abb.: 15 Beispielmethode zur Initialisierung eines Konfigurationsobjektes

Die Methode setProducer erstellt das Producer Objekt. Dabei werden die Parameter aus einem producerWrapper gelesen. Beim Lesen und überführen der Config JSON Datei in das Config Objekt werden die Parameter nicht direkt für das Erstellen des Producers verwendet. Die Parameter werden vorerst in einen Wrapper geladen. Dadurch kann sichergestellt werden, dass wenn in einer Konfiguration ein bestimmtes Objekt nicht konfiguriert wird, es beim Lesen des Config Objektes keine Exception Außerdem kann es sein, dass beim Erstellen der Konfigurationsdatei die Setter des Objektes schon ausgeführt werden, während die erforderlichen Parameter noch nicht vollständig geladen sind. Dies würde Ebenfalls zu einem Fehler führen.

```java
hosebirdClient = config.getTwitterClient().build(msgQueue);
hosebirdClient.connect();
kafkaProducer = config.getProducer();
```

Abb.: 16 Abruf eines Objektes aus der Konfiguration

Diese Abbildung zeigt, wie die einzelnen Objekte aus der Config benutzt werden. Die Codezeilen sind aus der TwitterStreamConn Methode, hier wird ein Hosebird Client und ein Kafka Producer aus der Config rausgelesen.

Das Programm wird über eine Manager Klasse gestartet. Der Manager wird aus der Main Methode definiert und bekommt die Applikations-Argumente mit. Anschließend startet er den ConfigLoader, um die Konfigurationsdate einzulesen und zu parsen. Danach startet er den Twitterstream.

```
14 public Manager(String[] args){
15     Config config = ConfigLoader.importGenConsConfig((args[0]));
16     if(config == null) {
17         log.error("config could not be loaded");
18     }
19     twitterStream = new TwitterStreamConn();
20     twitterStream.start(config);
21 }
22
23 public static void main(String[] args) {
24     new Manager(args);
25 }
```

Abb.: 17 Starten des Programms

2.4 Kafka Streams

Beim Starten des Programms wird zunächst, eine Verbindung mit dem TwitterClient aufgebaut. Wenn die Verbindung steht, werden die Nachrichten gepulled und über den KafkaProducer in das Topic geschrieben.

Das Programm verfügt über zwei Main Klassen. Die erste Main Klasse ist dafür zuständig, die oben genannte Funktionalität zu starten. Über die zweite Main Klasse kann der Teil des Programms gestartet werden, in dem die Kafka Streams API benutzt wird.

```
22 public static void main(String[] args) {
23     Properties props = new Properties();
24     props.put(StreamsConfig.APPLICATION_ID_CONFIG, "tweet-ObjectCount");
25     props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "141.72.191.168:9092");
26     props.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG, Serdes.String().getClass());
27     props.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG, Serdes.String().getClass());
```

Abb.: 18 Konfiguration der Kafka Streams

Der KafkaStream benötigt, wie auch der Producer, einige Basiskonfigurationen. In der Abbildung sind die vier Einstellungen die ID des Programmes, der Bootstrap Server und jeweils ein String Deserialisierungs-Config.

```
29          final StreamsBuilder builder = new StreamsBuilder();
30
31          KStream<String, String> source = builder.stream("twitter_dh");
32              source.flatMapValues(value -> Arrays.asList(value.toLowerCase(Locale.getDefault()).split("\"\\W*\":")))
33                .groupBy((key, value) -> value)
34                .count(Materialized.<String, Long, KeyValueStore<Bytes, byte[]>>as("counts-store"))
35                .toStream()
36                .to("kafka_streams_output_dh", Produced.with(Serdes.String(), Serdes.Long()));
```

Abb.: 19 Definieren der Verarbeitungslogik der KafkaStreams

Anschließend wird ein StreamBuilder definiert. Mit diesem StreamBuilder werden die Topologien des KafkaStreams gebildet. In Zeile 31 wird definiert, auf welches Topic der Stream definiert ist. In den nächsten Zeilen wird die Verarbeitungslogik definiert. Die Values der Records aus dem Topic werden nach einem Regulären Ausdruck aufgeteilt, nach der Value gruppiert und anschließend gezählt. Die Gezählten Werte werden wieder in ein Kafka Topic herausgeschrieben.

Sind die Verarbeitungsschritt des Streams definiert, wird die Topologie gebildet und mit der Topologie wird der KafkaStream erstellt (Zeile 39). Über den Aufruf stream.start() kann der Kafka Stream gestartet werde und mit der Verarbeitung beginnen.

3 Fazit

Kafka Stream ist ein Client Bibliothek für das Verarbeiten und Analysieren von Daten, welche schon in Kafka hineingeladen wurden. Über den Anwendungsfall wurde die Kafka Streams API verwendet. Die Kafka Streams API hat eine niedrige Eintrittshürde. Der Grund dafür liegt in der High Level, Domain Specific Language (DSL), welche für die meisten Anwendungsfälle zur Definition der Verarbeitungslogik ausreicht. Jedoch kann über die Processor API eine feinere Definition vorgenommen werden. Darüber hinaus hat Kafka Streams keine anderen Abhängigkeiten außer zu sich selbst. Kafka Streams unterstützt die exactly-once Processing Semantik welche garantiert, dass ein Record genau einmal verarbeitet wird, selbst im Falle eines Absturzes. Kafka Streams bietet zudem einen fehlertoleranten Lokalen Speicher, welcher Operationen wie Aggregation oder Windowed Joins effizient umsetzen kann. Insgesamt eignet sich Kafka Streams API gut für Anwendungsfälle, in denen ein konstanter Datenstrom sequenziell verarbeitet werden soll.

4 Literaturverzeichnis

1 Liu, Ling; Özsu, M. Tamer (Hg.) (2009): Encyclopedia of database systems. New York, NY: Springer (Springer reference). Online verfügbar unter http://dx.doi.org/10.1007/978-0-387-39940-9.

2 Muthukrishnan, S. (2005): Data streams: Now Publ (Foundations and trends in theoretical computer science, 1,2).

3 Vitter, Jeffrey Scott (2001): External memory algorithms and data structures. Dealing with massive data. In: *ACM Comput. Surv.* 33 (2), S. 209–271. DOI: 10.1145/384192.384193.

4 Valduriez, Patrick (2004): Proceedings of the 2004 ACM SIGMOD international conference on Management of data: ACM. Online verfügbar unter http://dl.acm.org/citation.cfm?id=1007568.

5 5: Apache Kafka. Online verfügbar unter https://kafka.apache.org/documentation/#majordesignelements, zuletzt geprüft am 27.05.2018.

6 6: Kafka Architecture (2018). Online verfügbar unter http://cloudurable.com/blog/kafka-architecture/index.html, zuletzt aktualisiert am 25.04.2018, zuletzt geprüft am 27.05.2018.

7 7: Apache Kafka. Online verfügbar unter https://kafka.apache.org/11/documentation/streams/architecture, zuletzt geprüft am 28.05.2018.