SS
2019

KAFKA

VERWENDUNG DES MESSAGING SYSTEMS IN JAVA

MICHAEL SIEBER

HOCHSCHULE MÜNCHEN IF6

Inhalt

1 Grundprinzip 3
1.1 Record 3
1.2 Topic 3
1.3 Producer 3
1.4 Consumer 4
1.5 Partition 4
1.5.1 Offset 4
1.5.2 Replikation 5
1.6 Consumer Group 6
1.7 Kafka-Server (Broker) 7
2 Vorteile 7
3 Nachteile 8
4 Installation 8
5 Konfigurationsmöglichkeiten 8
5.1 Broker Konfiguration 9
5.2 Topic Konfiguration 9
6 Kafka in Java 9
6.1 Dependencies 9
6.2 Beispiel eines Kafka Consumers in Java 10
6.2.1 Konfiguration 10
6.2.2 Lesen von Daten 11
6.3 Beispiel eines Kafka Producers in Java 12
6.3.1 Konfiguration 12
6.3.2 Schreiben von Daten 12
6.4 Beispiel einer Kafka Streaming Anwendung in Java 13
6.4.1 Konfiguration 13
6.4.2 Verarbeiten von Daten 14
7 Testen von Kafka Anwendungen in Java 14
7.1 Dependencies 14
7.2 Testen von Consumern 15
7.3 Testen von Producern 15

7.4 Testen von Streams 16
8 Beispielanwendung 17
9 Schluss/Fazit 18
10 Abbildungsverzeichnis 19
11 Literatur- und Quellenverzeichnis 20

Erklärung zur selbstständigen Verfassung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig verfasst, noch nicht anderweitig für Prüfungszwecke vorgelegt, keine anderen als die angegebenen Quellen oder Hilfsmittel benützt, sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

________________________ ______________________

Ort, Datum Unterschrift

1 Grundprinzip

Abbildung 1 Kafka (Quelle: https://kafka.apache.org/images/logo.png)

Kafka ist eine Open Source Streamingplattform, welche ursprünglich von LinkedIn entwickelt und später von der Apache Software Foundation weiterentwickelt wurde. Sie dient als Messaging-System zwischen Sendern und Empfängern und nutzt das sogenannte „Publish and Subscribe" Prinzip. Dabei kennen sich Sender und Empfänger nicht. Lesen und schreiben erfolgen lediglich von/zum Messaging-System, somit entsteht keine direkte Kommunikation zwischen Sender und Empfänger, sondern die Kommunikation läuft indirekt über Kafka ab. Genutzt wird dabei das TCP Protokoll, um mit den Sendern und Empfängern zu kommunizieren. Um Kafka nutzen zu können bietet es mehrere API´s an, unter anderem die Consumer Api. Diese erlaubt es Daten (sogenannte Records) aus Kafka zu lesen. Die Producer API dient dazu Records in Kafka zu speichern. Außerdem bietet die Streaming API die Möglichkeit Records als Stream aus Kafka zu lesen, zu verarbeiten und dann wieder als Stream zu Kafka zurück zu schreiben. Kafka kann zudem verteilt, in einem sogenannten Cluster, laufen und somit über mehrere Server verteilt werden.

1.1 Record

Wie bereits beschrieben werden Daten in Kafka in sogenannten Records gespeichert. Diese haben immer einen Schlüssel („Key") und einen Wert („Value"). Beide werden intern binär abgespeichert und über Serializer und Deserializer zu den gewünschten Datentypen umgeformt. Damit können also beliebige Datentypen in Kafka abgespeichert und unter den Anwendungen ausgetauscht werden. Außerdem haben Records immer einen Timestamp, welcher automatisch beim Speichern in Kafka generiert wird.

1.2 Topic

Alle Records werden immer in sogenannten Topics gespeichert. Ein Record wird dabei immer genau einem Topic zugeordnet. Topics sind somit eine Zusammenfassung an Records. In diese Topics können Producer Records hineinschreiben. Consumer können wiederum Records von einem oder mehreren Topics lesen. Von einem Topic können außerdem kein, ein oder mehrere Consumer lesen.

1.3 Producer

Producer schreiben die Records in die Topics. Dabei ist der Producer dafür zuständig, welcher Record in welches Topic geschrieben werden soll.

1.4 Consumer

Consumer lesen die Daten (Records) aus den Topics von Kafka. Consumer können dabei selbst bestimmen, welche Topics gelesen werden sollen.

1.5 Partition

Innerhalb eines Topics werden die Records abhängig von ihrem Key bestimmten Partitionen zugeordnet. Beim Anhängen eines Records bleibt die Reihenfolge innerhalb einer Partition bestehen, aber über Partitionen hinweg wird dies nicht gewährleistet. Die Partitionen dienen dabei der Parallelisierung, sodass mehrere Producer gleichzeitig schreiben und mehrere Consumer gleichzeitig lesen können. Außerdem sorgen sie dafür, dass Topics über mehrere Server verteilt werden können und somit auch größer als die Kapazität eines Servers sein können. Nur eine einzelne Partition darf nicht größer werden als der Server, auf dem die Partition gespeichert ist. Die Partitionen werden somit im Cluster über mehrere Server verteilt, während ein Server meist die Daten und Anfragen mehrerer Partitionen verwaltet. Dies bewirkt außerdem eine Lastverteilung auf die verschiedenen Server im Cluster.

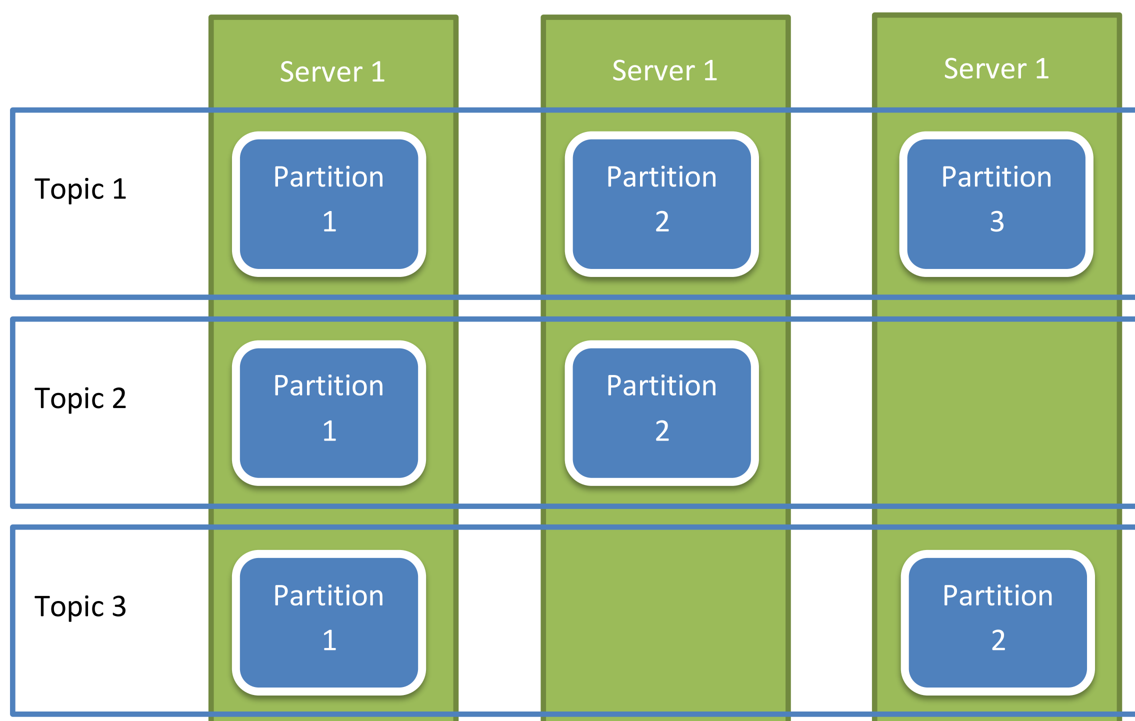

Abbildung 2 Partitions

1.5.1 Offset

Nachrichten bekommen einen Offset innerhalb einer Partition zugeordnet. Nach dem Lesen einer Nachricht wird diese nicht gelöscht, sondern der Consumer selbst schiebt seinen Offset weiter, so bleiben die Records (für eine bestimmte Zeit abhängig von der Konfiguration)

erhalten. Beispielsweise kann die Aufbewahrungszeit auf 2 Tage gesetzt werden, so bleiben die Records immer 2 Tage gespeichert, egal ob sie bereits gelesen wurden oder nicht. Für jeden Consumer werden beim Lesen die Offsets der jeweiligen Partitionen zwischengespeichert, um an der letzten Stelle weiter lesen zu können. Dadurch, dass Kafka auch persistent ist und die Consumer selbst ihren Offset verwalten, können Consumer auch von Beginn an (offset null) lesen und somit alle bisher von Producern erzeugten Nachrichten dieses Topics konsumieren oder der Offset kann aber auch manuell beim Lesen gesetzt werden, um Nachrichten mit einem bestimmten Offset zu lesen. So kann der Consumer zum Beispiel seinen Offset auf einen älteren Zeitpunkt zurücksetzen, um Records erneut zu verarbeiten. Standartmäßig werden die Nachrichten aber der Reihenfolge nach gelesen und der Offset automatisch weiter gesetzt.

1.5.2 Replikation

Um eine bessere Ausfallsicherheit zu erreichen, können die Partitionen zusätzlich repliziert werden. Damit werden die Daten gleichzeitig auf mehreren Servern gespeichert. Dabei ist immer ein Replikat der Leader, in den die Producer schreiben. Die anderen Replikas folgen dann dem Leader. Fällt ein Leader Replika aus übernimmt eines des anderen Replikas das Lead. Die Leader werden dabei auf die verschiedenen Server aufgeteilt, um die Last bestmöglich zu verteilen.

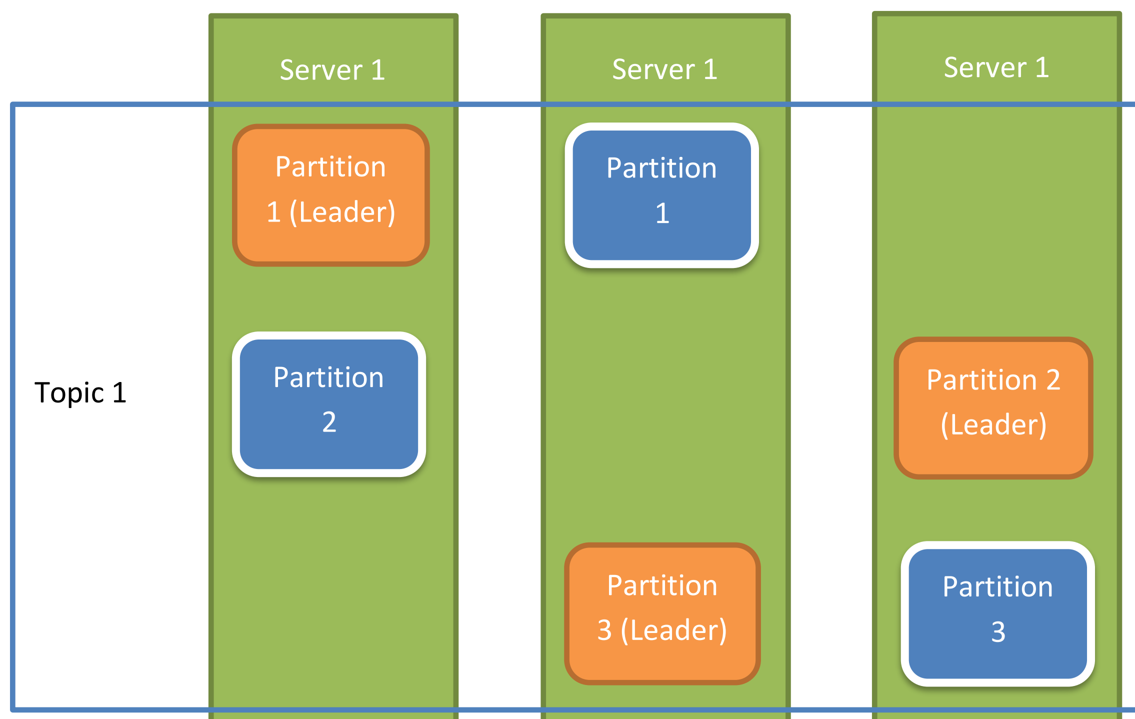

Abbildung 3 Veranschaulichung Replikas für ein Topic

1.6 Consumer Group

Consumer können sich zu Gruppen zusammenschließen. Dadurch kann in einer Gruppe ein Record nur einmal gelesen werden. Um eine genaue Aufteilung zu haben, welcher Consumer von welcher Partition liest, werden die Partitionen jeweils einem Consumer der Gruppe zugeordnet. Diese Zuordnung wird von Kafka dynamisch gehandhabt. So werden Partitionen neu zugeteilt, falls zum Beispiel neue Consumer zur Consumergruppe hinzukommen oder Consumer sich aufgrund eines Fehlers beenden. Gibt es mehr Partitionen als Consumer, so kann auch ein Consumer von mehreren Partitionen lesen. Allerdings sollte es immer mindestens gleich viel Partitionen wie Consumer geben. Consumer können auch allein eine Consumergruppe haben, damit sie von allen Partitionen lesen können. Außerdem können für ein Topic auch mehrere Consumergruppen nebeneinander bestehen. Jede Consumergruppe kann dann unabhängig von den anderen Gruppen die Records verarbeiten. Haben z.B. alle Consumer die gleiche Consumergruppe, werden die Records auf alle Consumer verteilt. Haben alle Consumer unterschiedliche Consumergruppen, werden alle Records an alle Consumer verteilt. Normalerweise werden Kombinationen aus mehreren Consumergruppen mit jeweils einem oder mehreren Consumern benötigt. Wie in folgendem Beispiel (Abbildung 4). Dabei lesen zwei Consumergruppen unabhängig voneinander die Records aus einem Topic mit drei Partitionen. Consumergruppe 1 hat drei Consumer und Consumergruppe 2 hat zwei Consumer. Die Pfeile stellen dabei die Zuordnung der Partitionen zu den Consumern der einzelnen Gruppen dar.

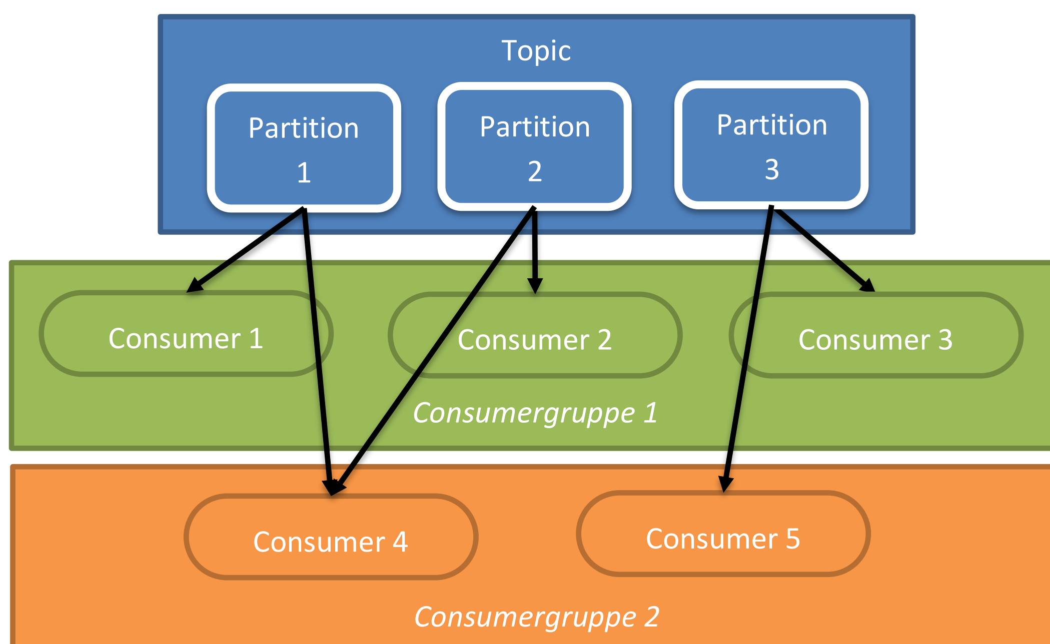

Abbildung 4 Consumergruppen

1.7 Kafka-Server (Broker)

Ein Kafka Cluster besteht meist aus mehreren Servern. Ein Server in einem Kafka Cluster wird dabei auch als Broker bezeichnet. Damit können dann die Partitionen, wie oben beschrieben, auf mehrere Server/Broker verteilt werden. Die Broker sind dann jeweils für die Kommunikation zu den Producern und Consumern zuständig. Folgende Abbildung zeigt vereinfacht einen Kafka Cluster mit drei Brokern.

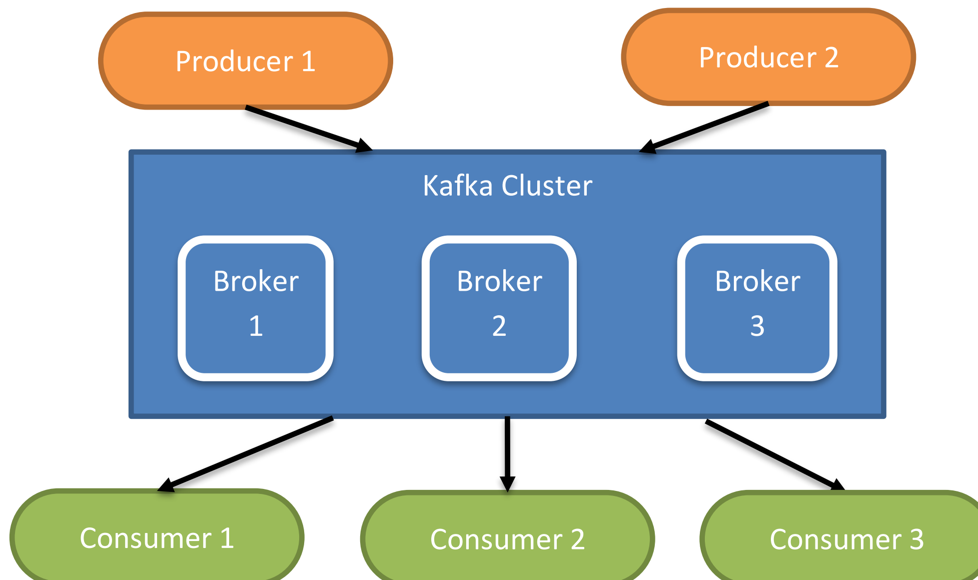

Abbildung 5 Kafka Broker

2 Vorteile

Kafka bietet einige Vorteile gegenüber anderen Messaging Systemen. Zum einen ist Kafka extrem ausfallsicher und bietet eine hohe Verfügbarkeit, da es verteilt auf mehreren Servern laufen kann. Zum anderen löst es die Probleme der Direktverbindung von Sender und Empfänger. Durch das Zwischenspeichern der Records wird der Empfänger nicht mehr so leicht überlastet, falls der Sender zu schnell Daten sendet und es gibt auch keinen Datenverlust, falls der Empfänger mal nicht erreichbar ist. Außerdem ist Kafka sehr gut skalierbar und große Datenmengen können in nahezu Echtzeit verarbeitet werden. Deshalb wird Kafka auch sehr häufig in Echtzeitverarbeitungspipelines eingesetzt. Des Weiteren können die Daten durch Consumergruppen auf mehrere Consumer verteilt werden, wodurch parallele Verarbeitung möglich wird. Ein Vorteil, was Kafka gegenüber anderen Messaging-Systemen bietet, ist die Möglichkeit, Daten über Streams zu verarbeiten, wodurch eine atomare End-zu-End Verarbeitung möglich wird. Dennoch bietet Kafka nicht nur Vorteile:

3 Nachteile

Hauptproblem bei Kafka ist, dass durch das Nutzen von Kafka eine zusätzliche Technologie eingeführt, konfiguriert und gewartet werden muss. Dies erfordert Zeit- und Kostenaufwand, sowie die nötigen Kompetenzen. In den folgenden Kapiteln (4. Installation und 5. Konfiguration) wird auf die wichtigsten Aspekte zur Einführung und Konfiguration von Kafka eingegangen.

4 Installation

Um Kafka lokal auf dem PC zu nutzen, muss zunächst der Code in Form einer Tar-Datei heruntergeladen und entpackt werden. Darin ist auch ein Zookeeper enthalten, welcher zum Verwalten des Clusters zuständig ist und als erstes über folgende Kommandozeile gestartet werden muss:

```
bin/zookeeper-server-start.sh config/zookeeper.properties
```

Danach kann über den Befehl

```
bin/kafka-server-start.sh config/server.properties
```

ein Server gestartet werden.

Über die Kommandozeile können außerdem auch gleich Topics erstellt werden. Diese werden mit der Default-Konfiguration aber auch beim ersten Schreiben automatisch erstellt. Des Weiteren können über Consolen-Producer und -Consumer auch Daten in Topics geschrieben und aus Topics gelesen werden. Dazu dienen folgende Kommandos (hier wird das Topic „test“ verwendet):

```
bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic test --from-beginning
```

```
bin/kafka-console-producer.sh --broker-list localhost:9092 --topic test
```

Für die Verwendung in Java sind zunächst aber nur die Kommandos zum Starten von Zookeeper und Server wichtig. Zunächst sollte dieser aber richtig konfiguriert werden:

5 Konfigurationsmöglichkeiten

Kafka bietet eine Vielzahl an Konfigurationsmöglichkeiten. Auf die Konfiguration von Consumer und Producer wird jeweils in den entsprechenden Kapiteln (5.2.1 und 5.3.1) eingegangen. Aufgrund der großen Menge der Konfigurationsmöglichkeiten wird außerdem im Folgenden nur auf die wichtigsten Parameter von Broker und Topic eingegangen.

5.1 Broker Konfiguration

Die Konfiguration der Broker erfolgt über die server.properties Datei oder über die Kommandozeile. Die wichtigsten Parameter hierbei sind:

BROKER.ID: Eindeutige ID für Broker. Falls nicht gesetzt wird eine ID vom Zookeeper für den Broker generiert.

LOGS.DIR: Verzeichnis an dem die Log-Daten abgespeichert werden.

ZOOKEEPER.CONNECT: Liste von Host/Port Paaren zur Verbindung mit dem Zookeeper.

5.2 Topic Konfiguration

Topic-Parameter haben immer auch eine serverseitige Konfiguration, welche als Default verwendet wird. Allerdings kann diese für jedes Topic einzeln überschrieben werden. Hierbei gibt es zum Beispiel Konfigurationsmöglichkeiten für:

MAX.MESSAGE.BYTES: Überschreibt MESSAGE.MAX.BYTES der Broker Konfiguration und definiert die maximale Record-Batch-Size, welche durch Kafka erlaubt ist.

RETENTION.BYTES: Maximaler Größe einer Topic-Partition, bis angefangen wird alte Segmente zu löschen.

RETENTION.MS: Maximale Zeit, bis ein Log gelöscht wird, um Speicher wieder frei zu geben. Entspricht demnach der Zeit, wie lange ein Consumer Zeit hat Records zu lesen. Maximaler Wert ist 604800000, was einer Woche entspricht.

Diese Parameter werden beim Erstellen der Topics mitgegeben oder können nachträglich angepasst werden (z.B. in der Kommandozeile).

6 Kafka in Java

6.1 Dependencies

Um aus dem eben installierten und konfigurierten Kafka-Cluster Daten mittels Java lesen und schreiben zu können, werden zunächst Dependencies benötigt. In diesem Bespiel wurde Maven genutzt und folgende Dependency hinzugefügt, welche für die Kafka Consumer und Producer benötigt wird.

```xml
<!-- https://mvnrepository.com/artifact/org.apache.kafka/kafka-clients -->
<dependency>
    <groupId>org.apache.kafka</groupId>
    <artifactId>kafka-clients</artifactId>
    <version>2.2.0</version>
</dependency>
```

Abbildung 6 Kafka Maven Dependency

Für die später erwähnte Kafka Streaming Anwendung wird außerdem diese Dependency benötigt:

```xml
<!-- https://mvnrepository.com/artifact/org.apache.kafka/kafka-streams -->
<dependency>
    <groupId>org.apache.kafka</groupId>
    <artifactId>kafka-streams</artifactId>
    <version>2.2.0</version>
</dependency>
```

Abbildung 7 Kafka Streaming Maven Dependency

6.2 Beispiel eines Kafka Consumers in Java

6.2.1 Konfiguration

Sind die Dependencies eingefügt, kann mit der Konfiguration eines Consumers fortgefahren werden. Hierfür muss zunächst eine Property erstellt werden. Darüber können dann die verschiedenen Konfigurationen gesetzt werden.

Die wichtigsten Konfigurationen sind dabei:

BOOTSTRAP_SERVERS_CONFIG:
Liste von Host/Port Paaren zur Verbindung mit dem Kafka Cluster.

GROUP_ID_CONFIG:
Wie oben bereits beschrieben können sich Consumer zu sogenannten Consumergruppen zusammenschließen. Mit dieser Config wird die Gruppe angegeben, zu welcher dieser Consumer gehört.

KEY_DESERIALIZER_CLASS_CONFIG und VALUE_DESERIALIZER_CLASS_CONFIG:
Da die Daten intern in Kafka binär abgespeichert werden, benötigt der Consumer zum Lesen der Daten für Key und Value jeweils einen Deserializer. In diesem Fall werden dazu String Deserializer verwendet.

```
private final static String TOPIC = "testtopicnew";
private final static String BOOTSTRAP_SERVERS = "localhost:9092";

final Properties props = new Properties();
props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, BOOTSTRAP_SERVERS);
props.put(ConsumerConfig.GROUP_ID_CONFIG, "TestConsumer");
props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class.getName());
props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class.getName());
props.put(ConsumerConfig.MAX_POLL_RECORDS_CONFIG, 1);
```

Abbildung 8 Beispiel einer Kafka-Consumer Konfiguration in Java

In diesem Beispiel wurde dazu noch die Config MAX_POLL_RECORDS_CONFIG gesetzt, um sicher zu stellen, dass immer nur ein einzelner Record gelesen wird.

Es gibt noch viele weitere Konfigurationsmöglichkeiten. Diese können in der Kafka Dokumentation nachgelesen werden.

6.2.2 Lesen von Daten

Nun, da die Konfiguration abgeschlossen ist, wird zum Lesen der Daten ein Kafka Consumer benötigt. Diesem werden beim Erstellen die oben beschriebenen Properties übergeben. Um Daten lesen zu können muss dann der Consumer noch einem oder mehreren Topics zugeordnet werden. Dies geschieht mit der subscribe Methode, welche eine Singleton-Liste an Strings erwartet. Nun kann über die poll-Methode aus den Topics gelesen werden.

```
KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);
consumer.subscribe(Collections.singletonList(TOPIC));

System.out.println("consumer opened");
while (true) {
    consumer.poll( timeoutMs: 5000).records(TOPIC).forEach(
            x -> System.out.println("message consumed with value: " + x.value())
    );
    Thread.sleep( millis: 10000);
}
```

Abbildung 9 Beispiel eines Java Codes zum Lesen von Daten aus Kafka

6.3 Beispiel eines Kafka Producers in Java

6.3.1 Konfiguration

Ähnlich wie bei der Konfiguration des Consumers muss auch bei dem Producer zunächst eine Property erstellt werden, um dann die benötigten Konfigurationen setzen zu können.

Für den Producer sind dabei die wichtigsten Konfigurationen:

BOOTSTRAP_SERVERS_CONFIG: Liste von Host/Port Paaren zur Verbindung mit dem Kafka Cluster.

CLIENT_ID_CONFIG: ID des Producers, welche dann bei jedem Request Kafka übergeben wird. Diese ID dient dazu, die Quelle der Requests nachzuvollziehen und loggen zu können.

KEY_SERIALIZER_CLASS_CONFIG und VALUE_SERIALIZER_CLASS_CONFIG: Wie oben beschrieben werden die Daten in Kafka intern binär abgespeichert. Deshalb werden zum Schreiben Serializer für Key und Value benötigt. In diesem Fall werden dazu String Serializer verwendet.

```
private final static String TOPIC = "testtopicnew";
private final static String BOOTSTRAP_SERVERS = "localhost:9092";
final Properties props = new Properties();
props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, BOOTSTRAP_SERVERS);
props.put(ProducerConfig.CLIENT_ID_CONFIG, "TestProducer");
props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, StringSerializer.class.getName());
props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, StringSerializer.class.getName());
```

Abbildung 10 Beispiel einer Kafka-Producer Konfiguration in Java

Wie auch beim Kafka-Consumer gibt es auch für den Producer noch viele weitere Konfigurationsmöglichkeiten, welche ebenfalls in der Kafka Dokumentation nachzulesen sind.

6.3.2 Schreiben von Daten

Um nun Daten nach Kafka schreiben zu können wird zunächst ein KafkaProducer benötigt. Diesem werden beim Erstellen die gerade gesetzten Properties übergeben. In der folgenden while-Schleife werden dann zu Demonstrationszwecken Eingaben von der Konsole erwartet, um diese dann in Kafka zu speichern. Der eingegebene String muss dann in einen ProducerRecord verwandelt werden, welcher zusätzlich das Topic enthält, in welches der Record gespeichert werden soll. Dann kann dieser Record über die send-Methode zu Kafka geschrieben werden. Um sicher zu gehen, dass die Nachricht direkt verarbeitet wird, kann dann noch die flush-Methode aufgerufen werden. Folgender Code zeigt die eben beschriebene Beispielanwendung:

```
KafkaProducer<String, String> producer = new KafkaProducer<String, String>(props);
System.out.println("producer opened");
Scanner userinput = new Scanner(System.in);

while (true) {
    String input = userinput.nextLine();
    producer.send(new ProducerRecord<String, String>(TOPIC, input, input));
    producer.flush();
    System.out.println("messsage produced");
}
```

Abbildung 11 Beispiel eines Java Codes zum Schreiben von Daten zu Kafka

6.4 Beispiel einer Kafka Streaming Anwendung in Java

6.4.1 Konfiguration

Häufig benötigen Anwendungen aber nicht nur entweder einen Consumer oder einen Producer, sondern beides, um in einer Applikation Daten aus Kafka zu lesen, zu verarbeiten und dann wieder zu Kafka zu schreiben. Dafür kann ein Kafka Consumer und ein Kafka Producer erstellt werden und dann dazwischen die Verarbeitungslogik implementiert werden. Dieser Anwendungsfall kann aber mit der von Kafka angebotenen Streaming API deutlich vereinfacht werden. Um Daten mittels Streams verarbeiten zu können, muss wie immer erst eine Konfiguration erstellt werden. Folgende Properties sind für die Streamverarbeitung erforderlich:

BOOTSTRAP_SERVERS_CONFIG: Liste von Host/Port Paaren zur Verbindung mit dem Kafka Cluster.

APPLICATION_ID_CONFIG: ID für die Streamapplikation. Muss eindeutig im KafkaCluster sein.

DEFAULT_KEY_SERDE_CLASS_CONFIG: Angabe einer Klasse zum Serializieren und Deserializieren des Keys.

DEFAULT_VALUE_SERDE_CLASS_CONFIG: Angabe einer Klasse zum Serializieren und Deserializieren der Values.

```
private final static String BOOTSTRAP_SERVERS = "localhost:9092";

final Properties props = new Properties();
props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, BOOTSTRAP_SERVERS);
props.put(StreamsConfig.APPLICATION_ID_CONFIG, "testStreamingApplication");
props.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG, Serdes.String().getClass());
props.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG, Serdes.String().getClass());
```

Abbildung 12 Beispiel einer Kafka-Streams Konfiguration in Java

6.4.2 Verarbeiten von Daten

Als nächstes wird ein Kafka Stream (KStream) benötigt. Dieser wird mithilfe eines StreamBuilders erstellt. Bei der Erstellung wird gleich das Topic übergeben, aus welchem der Stream Daten lesen soll. Danach können beliebige Stream Operationen, wie Filtern, Mapping, etc. ausgeführt werden. Anschließend kann über „to“ ein Topic angegeben werden, zu dem der Stream die Daten wieder schreiben soll. Mit selbem Prinzip können auch noch weitere Streams erstellt werden. Sind die Streams fertig definiert, müssen sie noch mithilfe des Streambuilders gebaut werden. Zusammen mit den Properties werden dann die fertigen KafkaStreams erstellt und über die start-Methode gestartet. Das Programm läuft nun solange bis es explizit beendet wird. Dabei wird von dem/n Kafka Stream/s dauerhaft versucht, Daten zu lesen und sobald Daten vorhanden sind, diese verarbeitet. Folgende Anwendung liest aus dem „testtopic1“, filtert alle Records, welche im Value „Test“ oder „test“ enthalten, und schreibt alle nicht aussortierten Records in das „testtopic2“.

```
private final static String SOURCETOPIC = "testtopic1";
private final static String DESTIANTIONTOPIC = "testtopic2";
private final static String BOOTSTRAP_SERVERS = "localhost:9092";

StreamsBuilder builder = new StreamsBuilder();
KStream<String, String> kafkastream = builder.stream(SOURCETOPIC);

kafkastream.filter((k,v) -> !(v.contains("Test") || v.contains( "test"))).to(DESTIANTIONTOPIC);
KafkaStreams streams = new KafkaStreams(builder.build(), props);
System.out.println("starting stream");
streams.start();
```

Abbildung 13 Beispiel eines Java Codes zum Verarbeiten von Daten aus Kafka mit Kafka Streams

7 Testen von Kafka Anwendungen in Java

7.1 Dependencies

Ist die Anwendung nun soweit fertig, sollte die korrekte Funktionalität bestmöglich über automatisierte Tests sichergestellt werden. Hierfür liefert Kafka bereits Mock-Klassen für Consumer und Producer in der Dependency Kafka-Clients mit. Einzig für Streaming Anwendungen werden die kafka-streams-test-utils benötigt. Außerdem ist ein Testframework (in folgenden Beispielen JUnit) zum Schreiben der Testklassen nötigt.

```
<!-- https://mvnrepository.com/artifact/org.apache.kafka/kafka-streams-test-utils -->
<dependency>
    <groupId>org.apache.kafka</groupId>
    <artifactId>kafka-streams-test-utils</artifactId>
    <version>2.2.0</version>
    <scope>test</scope>
</dependency>
```

Abbildung 14 Dependency zum Testen von Java Streaming Anwendungen

7.2 Testen von Consumern

Wie gerade beschrieben wird zum Testen die Mock-Klasse des Consumers verwendet. Der MockConsumer erwartet beim Erstellen als Übergabewert eine OffsetResetStrategy um zu wissen, ob er von Beginn an lesen soll oder nur neue Records. Dann wird der MockConsumer einem oder mehreren Topics zugewiesen und die jeweiligen Offsets gesetzt.

```
@Before
public void setUp() {
    mockconsumer = new MockConsumer<String, String>(OffsetResetStrategy.EARLIEST);
    mockconsumer.assign(Arrays.asList(new TopicPartition( topic: "testtopic2",  partition: 0)));
    HashMap<TopicPartition, Long> beginningOffsets = new HashMap<>();
    beginningOffsets.put(new TopicPartition( topic: "testtopic2",  partition: 0),  0L);
    mockconsumer.updateBeginningOffsets(beginningOffsets);
}
```

Abbildung 15 Erstellen eines MockConsumer in Java

Ist der Consumer fertig erstellt, können ihm über die addRecord Methode Testrecords übergeben werden, welche später mit dem Consumer gelesen werden sollen. Diese Records benötigen ein Topic, dem sie zugeordnet sind, eine Partition, einen Offset, sowie Key und Value.

```
mockconsumer.addRecord(new ConsumerRecord<String, String>( topic: "testtopic2",  partition: 0,  offset: 0L,  key: "Test1",  value: "Test1"));
mockconsumer.addRecord(new ConsumerRecord<String, String>( topic: "testtopic2",  partition: 0,  offset: 1L,  key: "Test2",  value: "Test2"));
```

Abbildung 16 TestRecords zu Mockconsumer hinzufügen

Nun kann die normale Verarbeitung ausgeführt werden. Der MockConsumer verhält sich dabei wie ein normaler KafkaConsumer, mit dem Vorteil, dass dabei kein laufender Kafka Cluster benötigt wird und die Verarbeitung genau für die vorher eingefügten TestRecords geprüft werden kann.

7.3 Testen von Producern

Auch für den Producer gibt es eine entsprechende Mock-Klasse. Dieser erwartet einen Boolean-Wert, um die Autocompletion (Commit des Records wird abgeschlossen, ohne explizit completeNext aufrufen zu müssen) einzuschalten, sowie Key- und Value-Serializer.

```
mockproducer = new MockProducer<String, String>(
         autoComplete: true, new StringSerializer(), new StringSerializer());
```

Abbildung 17 Erstellen eines MockProducers in Java

Nun kann wie auch beim MockConsumer die eigentliche Verarbeitung stattfinden und beliebig viele Records über den Producer committet werden. Diese werden im MockProducer gespeichert und können später über die Methode history abgefragt und mit Erwartungswerten verglichen werden.

```
List<ProducerRecord<String, String>> history = mockproducer.history();

List<ProducerRecord<String, String>> expected = Arrays.asList(
        new ProducerRecord<String, String>( topic: "testtopic1",  key: "Test1",  value: "Test1"),
        new ProducerRecord<String, String>( topic: "testtopic1",  key: "Test2",  value: "Test2"));

Assert.assertEquals( message: "Sent didn't match expected", expected, history);
```

Abbildung 18 Auslesen der Records aus MockProducer und Vergleich mit Erwartungswerten

7.4 Testen von Streams

Für das Testen von Streams muss zunächst ein TopologyTestDriver aus den kafka-streams-test-utils (siehe 6.1 Dependencies) erstellt werden. Diesem werden eine Topology und Porperties übergeben. Im folgenden Beispiel werden diese mit Gettern aus der eigenen Streaming-Klasse geholt. Anschließend können über die pipeInput-Methode des Drivers der zu testenden Streaminganwendung Testrecords übergeben werden. Der Einfachheit halber wurden diese TestRecords über eine ConsumerRecordFactory erstellt. Um nun die richtige Funktionalität abzuprüfen, können mit der readOutput-Methode die verarbeiteten Records wieder ausgelesen werden. Da die Streaminganwendung in diesem Beispiel alle Records filtert, welche im Value die Wörter „Test" und „test" enthalten, wird nur der zweite Testrecord ausgelesen. Ein weiterer Aufruf der readOutput-Methode gibt folglich null zurück, da keine weiteren Records vorhanden sind.

```
@Test
public void testStream() {
    SplittedStreaming stream = new SplittedStreaming();
    TopologyTestDriver driver = new TopologyTestDriver(stream.getTopology(),stream.getProps());

    ConsumerRecordFactory<String, String> factory = new ConsumerRecordFactory<>(
            new StringSerializer(), new StringSerializer());

    driver.pipeInput(factory.create( topicName: "testtopic1", key: "Test1",  value: "Should not pass because value contains test"));
    driver.pipeInput(factory.create( topicName: "testtopic1", key: "Test2",  value: "Should pass"));
    driver.pipeInput(factory.create( topicName: "testtopic1", key: "Test3",  value: "Should not pass because value contains Test"));

    ProducerRecord<String, String> record1 = driver.readOutput(
            topic: "testtopic2", new StringDeserializer(), new StringDeserializer());

    ProducerRecord<String, String> record2 = driver.readOutput(
            topic: "testtopic2", new StringDeserializer(), new StringDeserializer());

    Assert.assertEquals( expected: "Should pass", record1.value());
    Assert.assertNull(record2);
}
```

Abbildung 19 Test eines KafkaStreams

8 Beispielanwendung

Nun, da die Verwendung von Kafka ausführlich erläutert wurde, wird im Weiteren eine kurze Bespielanwendung vorgestellt. Als Beispiel wird hier eine vereinfachte ETL-Strecke verwendet. Dies ist eine Anwendung, die Daten aus Quellen extrahieren (**E**xtract), diese Daten verarbeiten (**T**ransform) und dann läd/speichert (**L**oad). Ziel dabei ist meist eine möglichst schnelle Verarbeitung der Daten, weshalb Parallelität und Skalierbarkeit eine große Rolle spielen. Dafür ist Kafka, wie oben beschrieben, bestens geeignet. Deshalb können anhand dieses Beispiels die Stärken von Kafka gut demonstriert werden.
Unter der Annahme, dass es drei Applikationen, welche Daten aus unterschiedlichen Datenquellen extrahieren, (Scraper) und zwei Verarbeitungsschritte gibt, könnte eine vereinfachte Architektur zum Beispiel wie folgt aussehen:

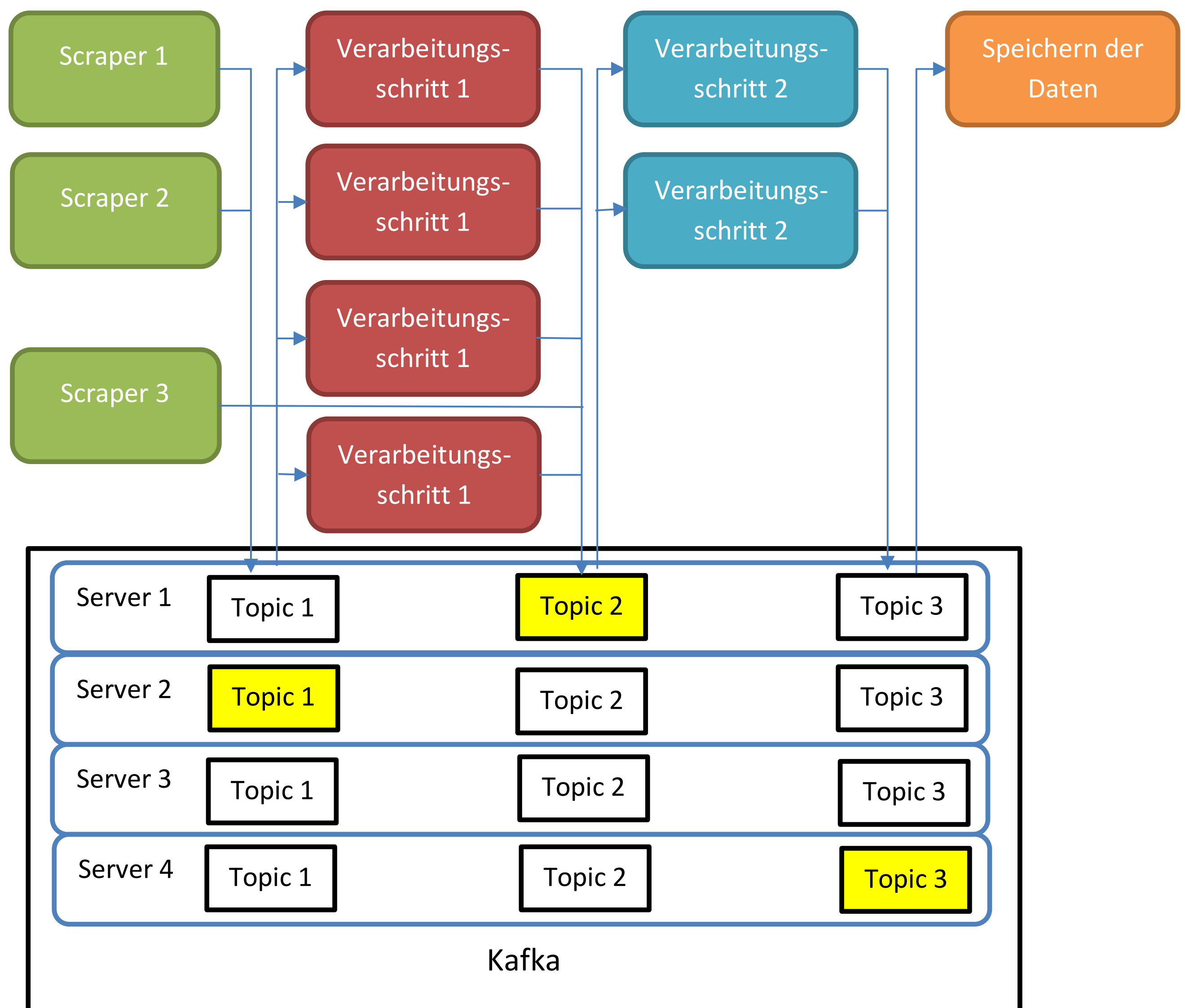

Abbildung 20 Aufbau einer ETL-Strecke mit Kafka (Beispielanwendung)

Die Scraper, welche meist Daten extrahieren, nutzen jeweils Producer, um die gescrapten Daten in einem Topic zu sammeln. Für die Verarbeitungsschritte würden sich Streams gut eignen, da dort die Daten sowohl gelesen als auch geschrieben werden müssen. Sind für unterschiedliche Quellen unterschiedliche Verarbeitungsschritte nötig, können zum Beispiel auch einzelne Schritte übersprungen werden (siehe Scraper 3). Um konkrete Auswertungen auf den verarbeiteten Daten zu machen, können diese zum Beispiel dann von dem letzten Topic auf eine Datenbank abgebildet werden, hierfür würden dann mittels eines Consumers die Daten gelesen werden und dann abgespeichert werden. Alle Anwendungen sollten dann natürlich auch über automatisierte Tests eigenständig getestet werden.

Für komplexere ETL-Strecken wären logischerweise noch mehr Topics nötig, womit der gesamte Aufbau komplexer wird. Dieses vereinfachte Beispiel soll hier nur eine Einsatzmöglichkeit von Kafka veranschaulichen.

Kafka bietet nun den Vorteil die Anwendung leicht skalieren zu können. Dauert in diesem Beispiel Verarbeitungsschritt 1 zum Beispiel doppelt so lange wie der Verarbeitungsschritt 2, kann der Verarbeitungsschritt 1 auf die doppelte Menge laufender Applikationen hochskaliert werden, um dies wieder auszugleichen. So kann die Performance der ETL-Stecke erhöht werden. Um die Parallelität, die Möglichkeit der Skalierung, und eine Ausfallsicherheit erreichen zu können, müssen die Topics über Partitionen und Replikas auf mehrere Server verteilt werden.

Alles zusammen ergibt dies nun ein Konzept, wie eine Anwendung aussehen kann, welche Kafka als Messaging-System nutzt. Diese könnte dann zum Beispiel in einem Kubernetes Cluster einer Cloud deployed und somit verteilt ausgeführt werden.

9 Schluss/Fazit

Kafka bietet eine sehr gute Möglichkeit, Daten zwischen Anwendungen auszutauschen. Wie bereits in Kapitel 2 (Vorteile) beschrieben, bringt es einige Stärken mit sich. Natürlich muss dadurch auch eine weitere Technologie erlernt und gewartet werden, dies ist aus meiner Sicht nicht in jedem Anwendungsfall sinnvoll, aber kann in den richtigen Projekten von großem Vorteil sein, wie das Beispiel in Kapitel 8 zeigt. Die Seminararbeit zeigt außerdem, dass die Nutzung von Kafka in Java nicht übermäßig komplex ist. Zumindest mir fiel der Einstieg nicht besonders schwer. Einzig die richtige Konfiguration bietet einige Schwierigkeiten. Auch das Testen der Verbindungen zu Kafka ist durch die MockKlassen vergleichsweise einfach. Alles in Allem ist Kafka eine sehr spannende Technologie, mit der ich mich in Zukunft ausführlicher befassen werde.

10 Abbildungsverzeichnis

Abbildung 1 Kafka (Quelle: https://kafka.apache.org/images/logo.png) 3
Abbildung 2 Partitions 4
Abbildung 3 Veranschaulichung Replikas für ein Topic...5
Abbildung 4 Consumergruppen 6
Abbildung 5 Kafka Broker 7
Abbildung 6 Kafka Maven Dependency...10
Abbildung 7 Kafka Streaming Maven Dependency 10
Abbildung 8 Beispiel einer Kafka-Consumer Konfiguration in Java 11
Abbildung 9 Beispiel eines Java Codes zum Lesen von Daten aus Kafka 11
Abbildung 10 Beispiel einer Kafka-Producer Konfiguration in Java 12
Abbildung 11 Beispiel eines Java Codes zum Schreiben von Daten zu Kafka.........................13
Abbildung 12 Beispiel einer Kafka-Streams Konfiguration in Java 13
Abbildung 13 Beispiel eines Java Codes zum Verarbeiten von Daten aus Kafka mit Kafka Streams 14
Abbildung 14 Dependency zum Testen von Java Streaming Anwendungen 14
Abbildung 15 Erstellen eines MockConsumer in Java...15
Abbildung 16 TestRecords zu Mockconsumer hinzufügen 15
Abbildung 17 Erstellen eines MockProducers in Java 15
Abbildung 18 Auslesen der Records aus MockProducer und Vergleich mit Erwartungswerten 16
Abbildung 19 Test eines KafkaStreams 16
Abbildung 20 Aufbau einer ETL-Strecke mit Kafka (Beispielanwendung)...............................17

11 Literatur- und Quellenverzeichnis

Anderson, J. (10. November 2016). *Unit Testing Kafka*. Abgerufen am 09. Mai 2019 von http://www.jesse-anderson.com/2016/11/unit-testing-kafka/

Anderson, J. (16. November 2016). *Unit Testing Kafka Consumers*. Abgerufen am 09. Mai 2019 von http://www.jesse-anderson.com/2016/11/unit-testing-kafka-consumers/

Apache. (kein Datum). *Documentation*. Abgerufen am 10. April 2019 von https://kafka.apache.org/documentation/

Apache. (kein Datum). *Testing a Streams Application*. Abgerufen am 13. Mai 2019 von https://kafka.apache.org/11/documentation/streams/developer-guide/testing.html

bealdung. (6. November 2018). *Exactly Once Processing in Kafka*. Abgerufen am 29. April 2019 von https://www.baeldung.com/kafka-exactly-once

Johansson, L. (30. November 2016). *Part 1: Apache Kafka for beginners - What is Apache Kafka?* Abgerufen am 23. April 2019 von https://www.cloudkarafka.com/blog/2016-11-30-part1-kafka-for-beginners-what-is-apache-kafka.html

Luber, S. (05. Oktober 2018). *Was ist Apache Kafka?* Abgerufen am 16. Mai 2019 von https://www.bigdata-insider.de/was-ist-apache-kafka-a-763300/

Wolff, E. (2018). Das Microservices-Praxisbuch. Broschur dpunkt Verlag.