DHBW Mannheim

Big Data Analytics

Lecture 6

Frank Schulz

Data Streaming

Data Streaming

Backofen

Backstraße

Batch Processing

Stream Processing

Data Streams

Wo begegnen uns Data Streams?

- Social Media (z.B. Twitter Streams)
- Internet of Things (z.B. Sensor Daten)

Einzelne Datensätze eines Streams

- Meistens mit Zeitstempel
- Oft mit Geo-Tagging

Eigenschaften von Streaming Systemen

- Verarbeiten von jeweils einem Datensatz oder einem kleinen Zeitfenster
- Unabhängig von vorausgehenden Datensätzen
- Oft mehrere Datenquellen (verschiedene Sensoren)

"Data in Motion" vs. **"Data at Rest"**

Data Streams

Streaming = Verarbeitung einer potentiell unendlichen Datenmenge

Gegensatz: Verarbeitung von begrenzten Datenmengen (Batches)

Weitere Anforderungen

- Schnelle Antwortzeit / geringe Latenz
- Konsistente Auswertung / Korrektheit

Beachte:
Die Definition ist unabhängig von konkreten Systemarchitekturen wie Ein-Pass-Verarbeitung (nur einmaliges Lesen der Daten möglich).

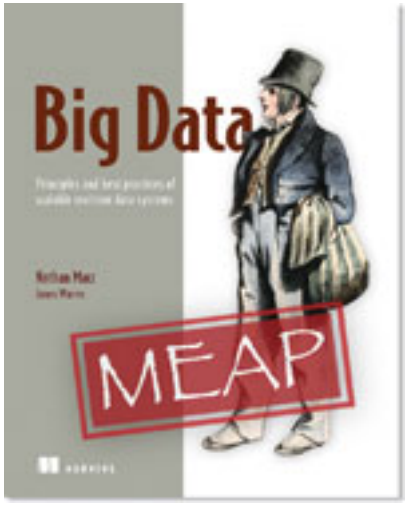
Data Streams

Wie kann man den Spagat lösen zwischen *Real Time* und *Batch Processing*?

- Echtzeit-Anforderung
- Korrekte Datenanalyse unter Berücksichtigung aller historischer Daten?

Lambda-Architektur

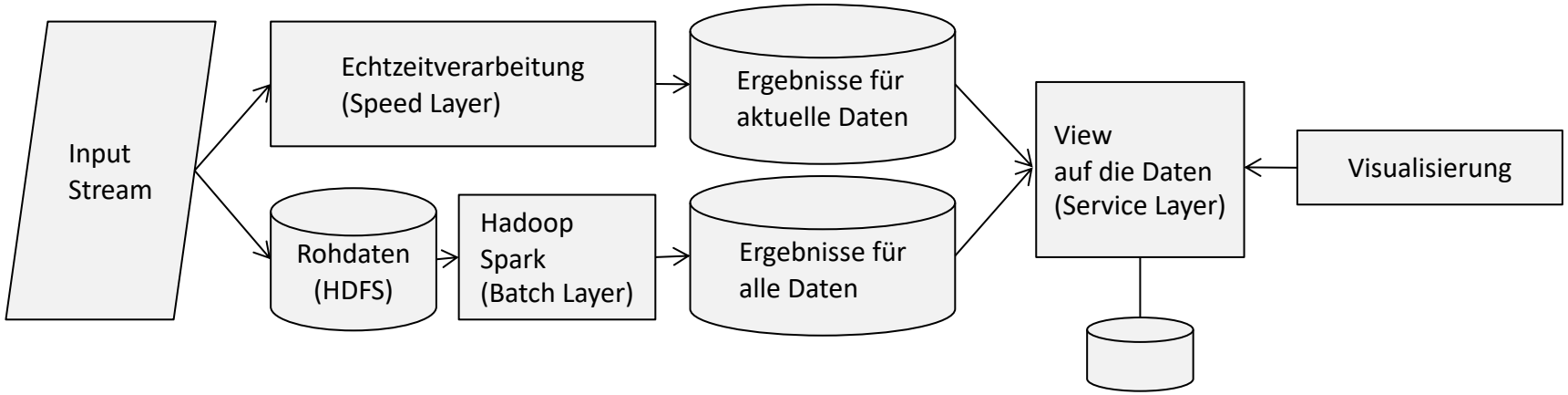
- Nathan Marz: „How to beat the CAP theorem“ (October 2013)
- lambda-architecture.net
- Nathan Marz und James Warren: Big Data (April 2015)



Lambda-Architektur

- Ein Vorschlag zum Aufbau von Big Data Systemen,
  die skalierbar, robust und fehlertolerant sind und
  die schnelle Lese- und Schreibzugriffe (low latency) unterstützen.
- Insbesondere Verarbeitung von Data Streams,
  d.h. Analysen in Echtzeit oder Fast-Echtzeit

- Idee: Kombination von Echtzeit-Auswertung und Batch-Verarbeitung
  für vordefinierte Queries bzw. Auswertungen.



Lambda-Architektur

Idee

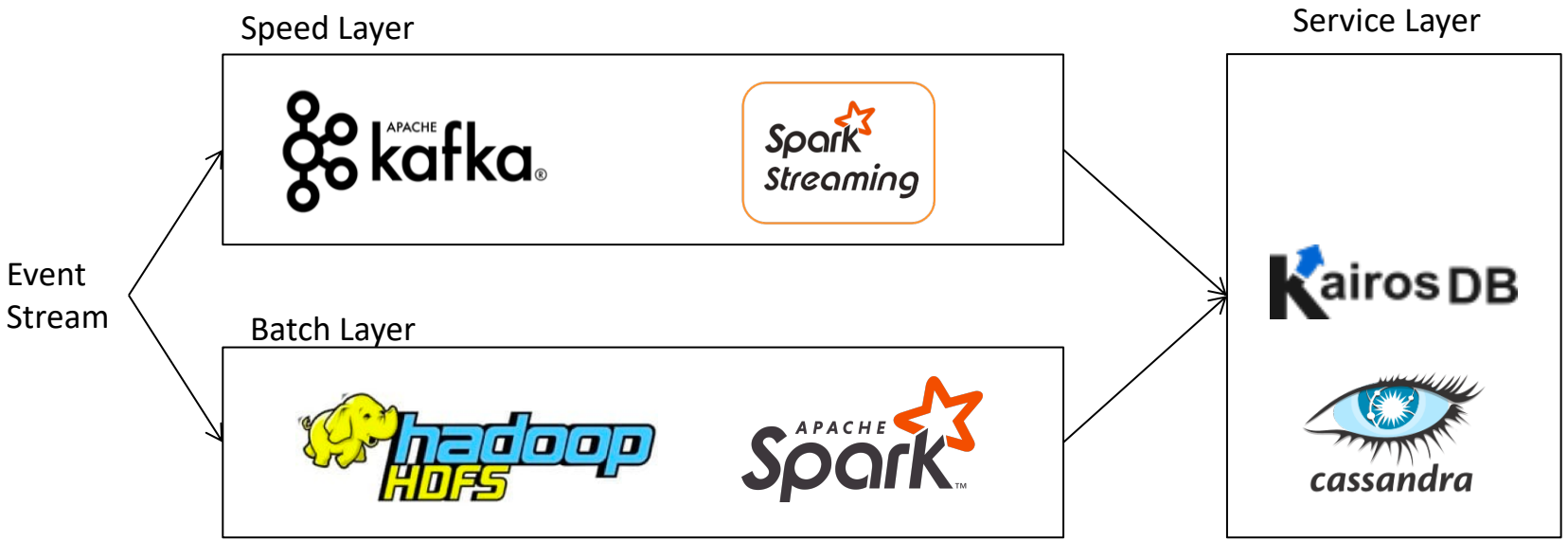
Kombination von Echtzeit-Auswertung und Batch-Verarbeitung für vordefinierte Queries (Views) bzw. Auswertungen.

- **Batch Layer** berechnet periodisch die Auswertung auf allen vorhandenen Daten (Map Reduce / Hadoop)
- **Speed Layer** berechnet inkrementell die Auswertung auf den neuen Daten seit dem letzten MapReduce-Job
  z.B. Apache Storm, Spark Streaming
- **Service Layer** bietet read-only Schnittstelle und gemeinsame Sicht auf Batch Daten und Realtime Daten.

Lambda-Architektur: mögliche Realisierung

Produktive Nutzung bei **Walmart.com**: Analyse von ClickStreams der Webseite/Online Shop

75.000 Clicks/Sekunde im Durchschnitt, 250.000 Clicks/Sekunde im Maximum



KairosDB
Open Source Zeitreihen-Datenbank
mit Persistenz in Cassandra

See: https://medium.com/walmartlabs/how-we-built-a-data-pipeline-with-lambda-architecture-using-spark-spark-streaming-9d3b4b4555d3

Lambda-Architektur

Vorteile gegenüber reinen Stream-Processing-Systemen

- Rohdaten bleiben erhalten: Später hinzugefügte Auswertungen (oder Programmkorrekturen) sind möglich

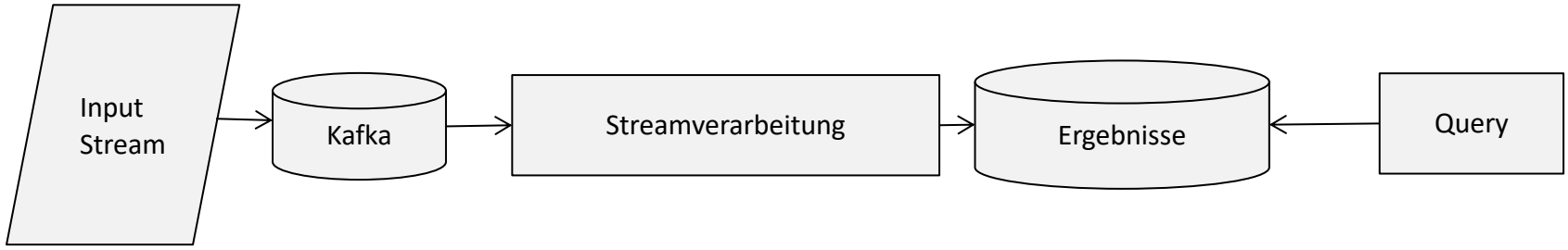
Nachteile

- Jede Auswertung muss zweimal programmiert werden: für Batch Layer und für Speed Layer in verschiedenen Umgebungen / verschiedenen Programmiersprachen
- Komplexes Abmischen im Service Layer notwendig

Kappa-Architektur

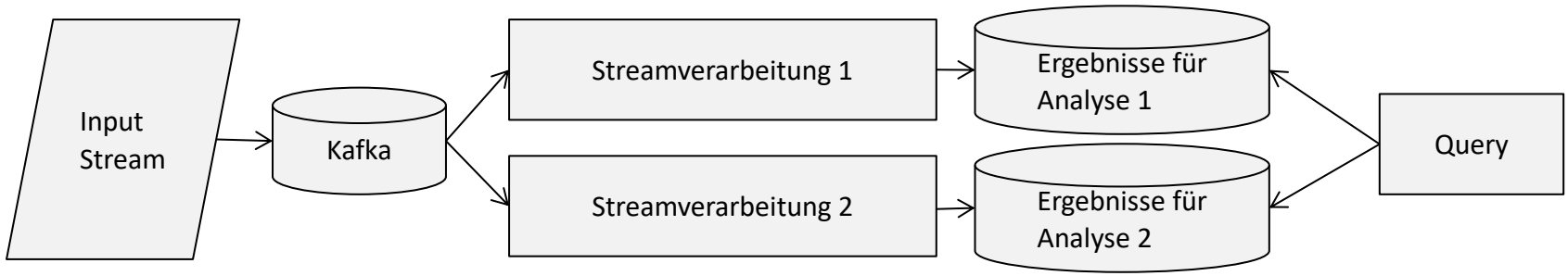
Einfacher Ansatz von Jay Kreps: „Questioning the lambda architecture“ (July 2014)

- Sämtliche Rohdaten bleiben in einem Input-Log erhalten (z.B. Apache Kafka) und können jederzeit von dort wieder abgerufen werden.
- Verarbeitung mit Stream Processing (z.B. Spark Streaming)



Kappa-Architektur

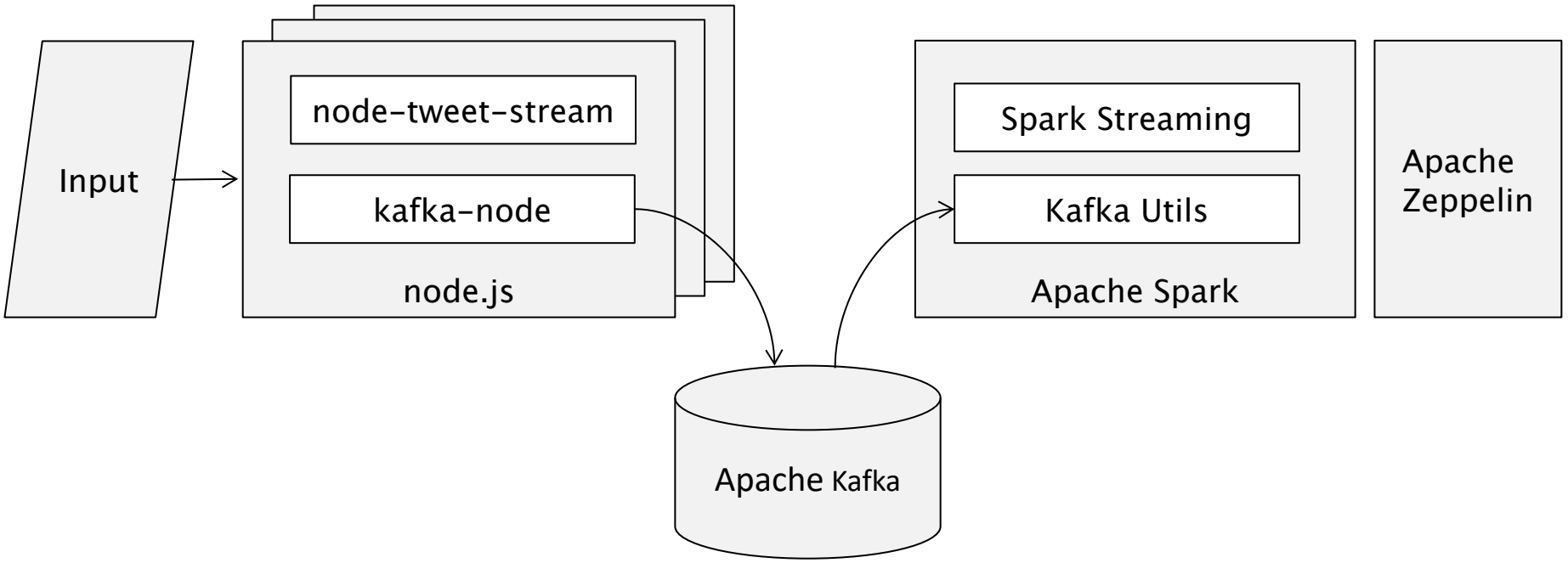
Bei einer zusätzlichen Query oder Änderung der Analyse wird ein zweiter Stream Processing Job gestartet, der die Rohdaten ab einem beliebigem Zeitpunkt in der Vergangenheit neu analysiert.



Praktisches Beispiel

Beispielhafter Aufbau einer Data Pipeline zum Verarbeiten von Realtime Tweets

Kafka als Datenpuffer zur Entkopplung des Producers vom Consumer



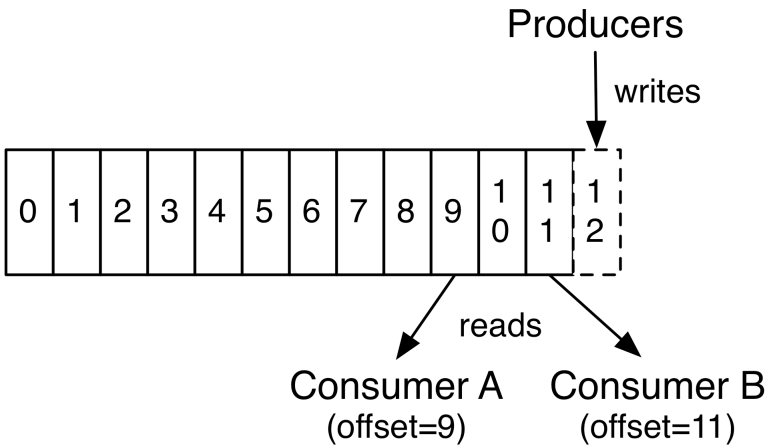
Apache Kafka

Verteiltes Messaging System

- 2011 entwickelt bei LinkedIn, 2012 Open Source

Publish-Subscribe

- **Producer** erzeugen Nachrichten (Input Datensätze)
- Ein **Topic** ist eine logische Queue, in die Nachrichten geschrieben werden
- **Consumer** lesen Nachrichten aus Topics



Apache Kafka

Reihenfolge-Garantie

- Die Nachrichten eines Producers in einer *Topic Partition* werden in genau der Reihenfolge gespeichert, in der sie gesendet worden sind.
- Ein Consumer liest die Nachrichten in genau der Reihenfolge, in der sie in der *Topic Partition* stehen.

Anwendungsszenarien

- Nachrichten-orientierte Middleware mit Publish&Subscribe Kommunikation
- Skalierbarer Datenpuffer – Entkoppelt Producer und Consumer
- Berechnungen und Transformationen von Datenströmen (Stream Processor)

Semantische Garantien / Fehlersemantik

- **At least once delivery** (acknowledged delivery)
  Das System stellt sicher, dass die Nachricht vom Producer mindestens einmal am Broker ankommt. Wenn das Acknowledgement vom Broker ausbleibt, schickt der Producer die Nachricht nochmal. Doppelte Nachrichten sind möglich.

- **At most once delivery** (fire and forget)
  Das System stellt sicher, dass die Nachricht vom Producer höchstens einmal am Broker ankommt. Wenn die Bestätigung vom Broker an den Producer ausbleibt, wird die Nachricht nicht nochmal geschickt. Es können Nachrichten verlorengehen.

- **Exactly once delivery** (assured delivery)
  Das System stellt sicher, dass jede Nachricht vom Producer genau einmal an einem Consumer ankommt. In einem verteilten System ist "exactly once *delivery*" unmöglich. Das kann aus dem "Two generals problem" (https://en.wikipedia.org/wiki/Two_Generals%27_Problem) abgeleitet werden.
  Mit Hilfe von Transaktionen auf Anwendungsebens ist aber "exactly once *processing*" möglich, siehe https://www.confluent.io/blog/exactly-once-semantics-are-possible-heres-how-apache-kafka-does-it/

Apache Kafka

Topic Modeling: How many topics should be used?

No general answer, but some guidelines:

- If the sequence is important, messages must go to the same topic. There is no ordering guarantee between the messages in different topics.
- Different message types in the input should be sent to different topics. Consumers should not have to distinguish message types themselves and apply different processing logic based on the type.

See https://www.confluent.io/blog/put-several-event-types-kafka-topic/

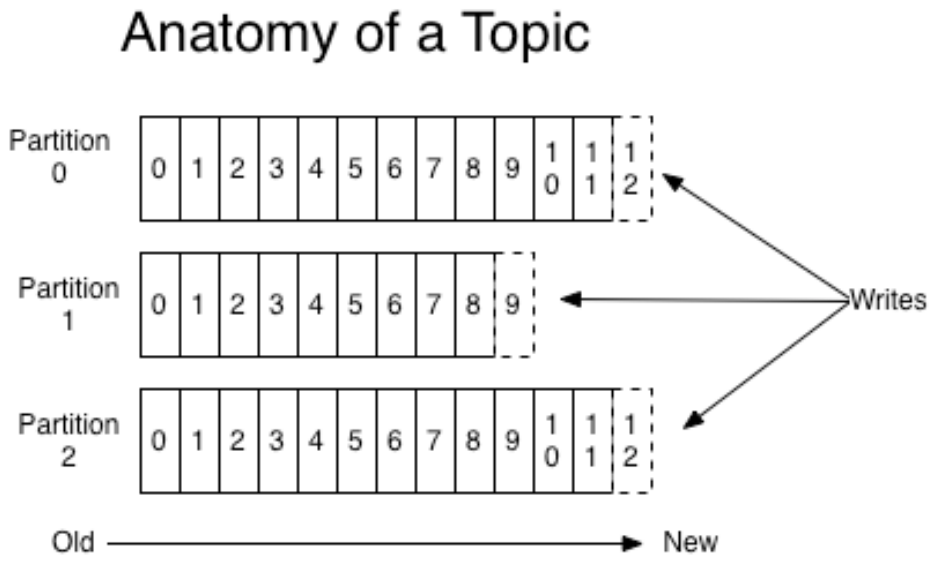
Apache Kafka Partitionen

Topics are divided into **partitions**.

Partitions are the units which can be processed in parallel for increasing the throughput of the system. **Goal: paralellism on producer side**

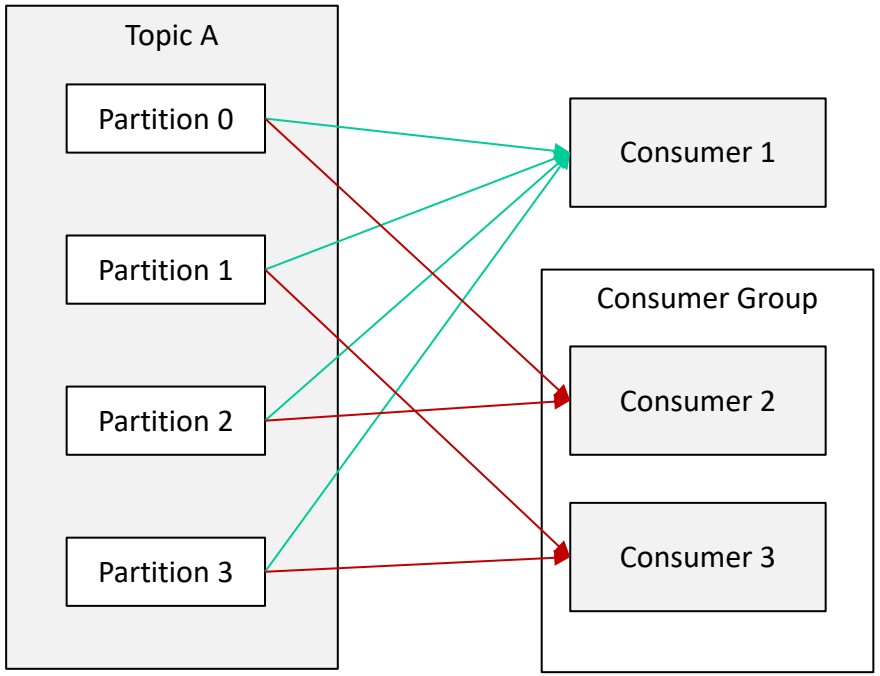
The bottleneck for parallelism is not Kafka, but the underlying Zookeeper key-value storage.

Around 10000 partitions can be supported by Zookeeper. The number of topics is not relevant. Whether one topic is divided into 1000 partitions, or 1000 topics have one partition each, is irrelevant for the achievable parallelization.



See https://kafka.apache.org/documentation/#intro_consumers

Apache Kafka Partitionen



Goal: paralellism on consumer side

Several consumers can be combined into a **consumer group** with help of a GroupID attribute. Messages are delivered to the consumer group, and each consumer within a group is assigned to a subset of the partitions of a topic. Each consumer of a group receives only some of the messages.

For example, given a topic with four partitions (0,1,2,3). Consumer 1 sees all messages. Consumers 2 and 3 are grouped together. Comsumer 2 receives the messages from partitions 0 and 2, and consumer 3 the messages of partitions 1 and 3.

See https://kafka.apache.org/documentation/#intro_consumers

Apache Kafka Internals

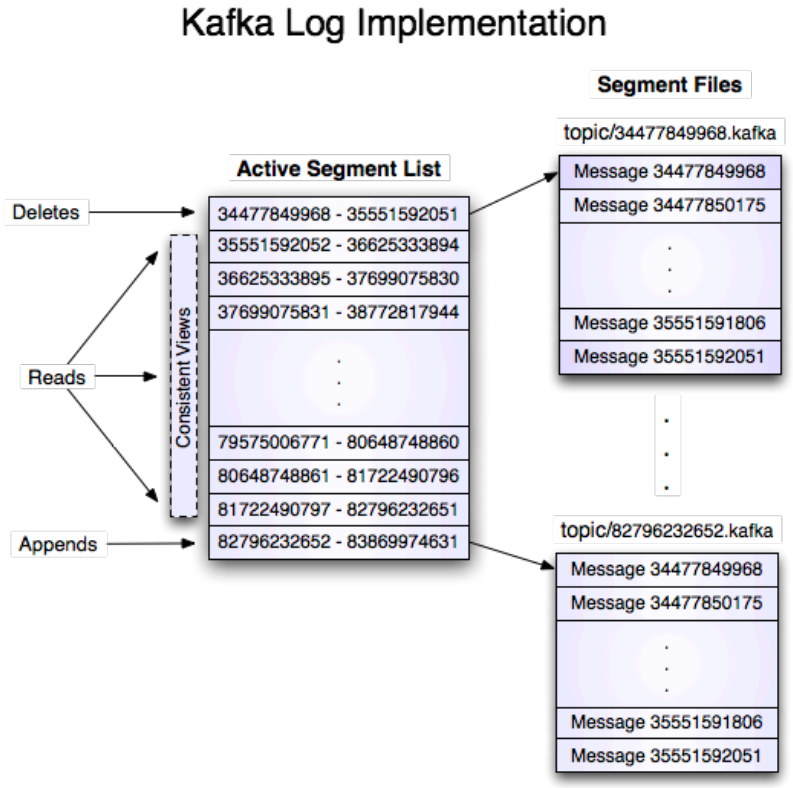
Eine Topic Partition wird als eine Folge von "Segment Files" gespeichert, die über eine Index Datei (active segment list) adressiert werden.

Ein neues Segment File wird begonnen, wenn einer der beiden folgenden Parameter erreicht ist:

- `segment.bytes`: maximale Größe
- `segment.ms`: maximales Alter

Segmente werden nur als Ganzes gelöscht.

Alter eines Segmentes ist das Alter der jüngsten Nachricht.



Source: https://kafka.apache.org/documentation/#log

Apache Kafka Configuration

How long can / should data be stored in Kafka?

By default, old messages are discarded automatically after a configurable period.

It is possible and can be reasonable to keep data in Kafka "forever":
https://www.confluent.io/blog/okay-store-data-apache-kafka/
https://www.confluent.io/blog/publishing-apache-kafka-new-york-times/

Message deletion per segment

Topic setting: The default retention time of a segment before deletion is 7 days:
`retention.ms = 604800000`

Depending on policy: `log.retention.policy = delete | compact | compact, delete`

Consumer group setting: The default time period for a consumer group to keep its reading offset (the pointer to the message last read) is 7 days: `offsets.retention.minutes=10080`

Apache Kafka Replication

Topic partitions can be replicated for fault tolerance and robustness.

The replication factor is set per topic when creating the topic.
Each partition has a single leader, and zero or more followers.

Producers and consumers always communicate with the leader.
If the leader crashes, a follower will be selected as new leader.

- Consumers can be sure that only committed messages are read.
- Producers can configure for how many replica servers to acknowledge a successful write before returning a response (`acks = 0 | 1 | all`), thereby realizing a tradeoff between latency and durability / consistency.

- `acks = 0` : fire and forget (at most once delivery) - typically used for unit tests etc

Apache Kafka

Follow https://kafka.apache.org/quickstart

Windows

- Install Java
- Install Zookeeper
- Install Kafka

MacOSX with Homebrew

```
$ brew cask install homebrew/cask-versions/java8
$ brew install kafka
```

Start Zookeeper and Kafka in the background

```
zookeeper-server-start /usr/local/etc/kafka/zookeeper.properties &
kafka-server-start /usr/local/etc/kafka/server.properties &
```

Check that Zookeeper is running

```
$ telnet localhost 2181
> ruok                    # Are you okay?
imok                      # I am okay.
```

Working with topics

```
$ kafka-topics --create --zookeeper localhost:2181 --replication-factor 1 --partitions 1 --topic pizza
Created topic "pizza".

$ kafka-topics --zookeeper localhost:2181 --list
__consumer_offsets
Pizza

$ kafka-topics --zookeeper localhost:2181 --topic pizza --describe

$ kafka-topics --zookeeper localhost:2181 --topic pizza --delete # mark for deletion
```

Consumer and producer console programs

```
# Start a producer and send some messages
$ kafka-console-producer --broker-list localhost:9092 --topic pizza
>Pizza test
>I am hungry
>Where can I get pizza?


# Start a consumer (in another windows)
$ kafka-console-consumer --bootstrap-server localhost:9092 --topic pizza --from-beginning

# Start consumer with offset (need to specifiy the partition)
$ kafka-console-consumer --bootstrap-server localhost:9092 --topic pizza --partition 0 --offset 2

# Start a consumer in an group
$ kafka-console-consumer --bootstrap-server localhost:9092 --topic pizza –-group food

# List all consumers of a consumer group including their current read offsets
$ kafka-consumer-groups --bootstrap-server localhost:9092 --group food --describe
```

Kafka Connect

- Framework to stream data into and out of Apache Kafka
- Contains several built-in connectors that can be used to stream data to or from commonly used systems or data sources
  - relational databases
  - HDFS
  - blob storages like Amazon S3
  - local file system
  - ...
- Can be extended by custom connectors

- Developed by Confluent

Connectors

- Source Connectors (producer for getting data into Kafka)
  - FileSourceConnector
  - JDBCSourceConnector
  - ...
- Sink Connectors (consumer for getting data out of Kafka)
  - HDFSSinkConnector
- MongoDB Kafka Connector (both source and sink)
  - https://www.mongodb.com/kafka-connector
- See https://www.confluent.io/hub/ for full list

Converter

- Translate between the internal runtime format and external serialization formats. For String, JSON, Avro, ...

Consumer and producer console programs

```
# Start a producer and send some messages
$ kafka-console-producer --broker-list localhost:9092 --topic pizza
>Pizza test
>I am hungry
>Where can I get pizza?


# Start a consumer (in another windows)
$ kafka-console-consumer --bootstrap-server localhost:9092 --topic pizza --from-beginning

# Start consumer with offset (need to specifiy the partition)
$ kafka-console-consumer --bootstrap-server localhost:9092 --topic pizza --partition 0 --offset 2

# Start a consumer in an group
$ kafka-console-consumer --bootstrap-server localhost:9092 --topic pizza –-group food

# List all consumers of a consumer group including their current read offsets
$ kafka-consumer-groups --bootstrap-server localhost:9092 --group food --describe
```

Install kafka-python module

```
$ pip install kafka-python
```

Simple producer sending one message to a topic, and simple consumer

```
from kafka import KafkaProducer

producer = KafkaProducer(bootstrap_servers="localhost:9092")
producer.send("pizza", "My message".encode())
producer.close()
```

```
from kafka import KafkaConsumer

consumer = KafkaConsumer(bootstrap_servers="localhost:9092")
consumer.subscribe("pizza")
for msg in consumer:
    print (msg.value.decode("utf-8"))
```

Creating a topic

```
from kafka import KafkaAdminClient

admin = KafkaAdminClient(bootstrap_servers="localhost:9092")
admin.create_topics(["pizza"])
```

Explicit Acknowledgement of Offset by Consumer

```
from kafka import KafkaConsumer

consumer = KafkaConsumer(bootstrap_servers="localhost:9092", enable_auto_commit=False)
consumer.subscribe("pizza")
for msg in consumer:
    print (msg.value.decode("utf-8"))
    consumer.commit_async()
```

```python
import tweepy
import json

consumer_key = "xxx"
consumer_secret = "yyy"
access_token = "zzz"
access_token_secret = "www"

if __name__ == '__main__':

    auth = tweepy.OAuthHandler(consumer_key, consumer_secret)
    auth.set_access_token(access_token, access_token_secret)
    api = tweepy.API(auth)

    donald = api.get_user("realDonaldTrump")
    print("User '{0}' (@{1}) has {2} followers".format(donald.name,
        donald.screen_name, donald.followers_count))

    # return the latest tweets (at most 10)
    timeline = api.user_timeline(id=donald.id, count=10)

    for status in timeline:
        print("{0}: {1}\n".format(status.created_at, status.text))
```

```python
import tweepy
import json

consumer_key = "xxx"
consumer_secret = "yyy"
access_token = "zzz"
access_token_secret = "www"

class StdOutListener(tweepy.streaming.StreamListener):
    def on_data(self, data):
    tweet = json.loads(data)
    if "text" in tweet:
        text = tweet["text"]
        created_at = tweet["created_at"]
        user_name = tweet["user"]["name"]
        user_sn = tweet["user"]["screen_name"]
        print("Posted by '{0}' (@{1}) at {2}: \n{3}\n".format(user_name, user_sn,
            created_at, text))
        return True

    def on_error(self, status):
        print(status)

if __name__ == '__main__':
    listener = StdOutListener()

    auth = tweepy.OAuthHandler(consumer_key, consumer_secret)
    auth.set_access_token(access_token, access_token_secret)
    api = tweepy.API(auth)

    stream = tweepy.Stream(auth, listener)
    stream.filter(track=["pizza"])
   # filter see https://developer.twitter.com/en/docs/tweets/filter-realtime/guides/basic-stream-parameters
```

Zusammenfassung

- Stream Processing
- Lambda-Architektur
- Kappa-Architektur
- Apache Kafka