Big Data
03
Spark & Kafka

Eric MSP Veith <veith@offis.de>

March 3, 2020

Organisatorisches

- Prüfungsleistung: Seminararbeit
  - 10 Seiten Ausarbeitung zum Thema: Arial o.ä., Schriftgröße 12pt, einfacher Zeilenabstand, 2cm Rand, keine separate Titelseite, Seitenzahl inkl. Literaturverzeichnis
  - 15 Minuten Vortrag zum Thema
- Seminarthemen & Folien online verfügbar:
  `https://vhome.offis.de/~eveith`

Python für Eilige

Python

- Entwickelt 1991 von Guido van Rossum
- Leicht zugänglich;
  "*There is one—and preferable only one—way to do things.*"
- Versatile Programmiersprache, u. a. für UNIX-Systemwerkzeuge, Webanwendungen und wissenschaftliche Analysewerkzeuge
- Dynamische Typsisierung (*Duck Typing*)
- Viele hochoptimierte (C/C++-Kern, Python-API) (z. B. NumPy, SciPy, TensorFlow, . . .)

Python Byte-for-Byte

- Einfache Syntax, basierend auf **Einrückung** und Zeilenumbrüchen
- Skalare (primitive Datentypen)

$$a = 42, 13, 37, \pi, \sqrt{2}, \sqrt{-1}, \ldots$$

```
>>> a = 42.0
42.0
>>> type(a)
<class 'float'>
>>> a += 0.3
>>> a
42.3
```

Python Byte-for-Byte

- Einfache Syntax, basierend auf **Einrückung** und Zeilenumbrüchen
- Skalare (primitive Datentypen)

$$a = 42, 13, 37, \pi, \sqrt{2}, \sqrt{-1}, \ldots$$

```
>>> a = 42.0
42.0
>>> type(a)
<class 'float'>
>>> a += 0.3
>>> a
42.3
```

- Verzweigungen

```
person = "Viking"
if person == "Viking":
    print("Spam, spam, spam, lovely spam!")
```

Listen in Python

- Definition mit freistehenden **eckigen Klammern**

```
a_list = [1, 2, 3]
```

- Eckige Klammern: **Indexoperator** zum Elementzugriff

```
print(a_list[0])     # => 1
print(a_list[1])     # => 2
```

- Index relativ zum Ende der Liste: **negativer Index**

```
print(a_list[-1])     # => 3
print(a_list[-2])     # => 2
```

Listen in Python

- Definition mit freistehenden **eckigen Klammern**

```
a_list = [1, 2, 3]
```

- Eckige Klammern: **Indexoperator** zum Elementzugriff

```
print(a_list[0])     # => 1
print(a_list[1])     # => 2
```

- Index relativ zum Ende der Liste: **negativer Index**

```
print(a_list[-1])     # => 3
print(a_list[-2])     # => 2
```

- **Iterieren** über Listenelemente

```
for i in a_list:
    print("%d" % i, end=', ')        # => 1, 2, 3,
```

Listen in Python

- Definition mit freistehenden **eckigen Klammern**

```
a_list = [1, 2, 3]
```

- Eckige Klammern: **Indexoperator** zum Elementzugriff

```
print(a_list[0])     # => 1
print(a_list[1])     # => 2
```

- Index relativ zum Ende der Liste: **negativer Index**

```
print(a_list[-1])     # => 3
print(a_list[-2])     # => 2
```

- **Iterieren** über Listenelemente

```
for i in a_list:
    print("%d" % i, end=', ')        # => 1, 2, 3,
```

- Typische `for`-Schleife mit **Laufvariable kein ideomatisches Python** – auch bei tatsächlichen „Zählaufgaben“:

```
for i in range(0, 10):
    print("%d" % i)  # => 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
```

Slices

- Notation mit `:` im Index-Operator:
  `[ lower_boundary : upper_boundary ]`
- Intevalldefinition: [*lowerBoundary*; *upperBoundary*)
  – links inklusive, rechts exklusive
- Notation ermöglicht auch Schrittweite:
  `[ lower_boundary : upper_boundary : step ]`

```
another_list = ['a', 'b', 'c', 'd', 'e']
print(another_list[1:3])    # => ['b', 'c' ]
print(another_list[-2:-1])  # => ['d']
print(another_list[0:-1:2]) # => ['a', 'c']
```

List Comprehensions

```
integers      = [i for i in range(0, 5)]
# => [0, 1, 2, 3, 4]
squares       = [i**2 for i in range(0, 5)]
# => [0, 1, 4, 9, 16]
even_numbers = [i for i in integers
                 if i % 2 == 0]
# => [0, 2, 4]
even_squares = [i**2 for i in range(0, 5)
                 if i**2 % 2 == 0]
# => [0, 4, 16]
```

Python-Funktionen

```
def quicksort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr) // 2]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]
    return quicksort(left) + middle + quicksort(right)

print(quicksort([3,6,8,10,1,2,1]))
```

Klassen, Objekte und Vererbung

```
class Answer:
    def __init__(self):
      self._answer = 42

    def compute(self):
        print("The answer to life, the universe "\
              "and everything is %d." % self._answer)

class ConfigurableAnswer(Answer):
    def __init__(self, answer):
        super().__init__()
        self.answer_ = answer


a = Answer(); b = ConfigurableAnswer(42)
a.compute();  b.compute()
```

NumPy

```
import numpy as np
```

- Hochoptimiert, C/C++/Fortran-Kernbibliotheken
- Im Kontext *Deep Learning* vor allem für lineare Algebra

```
a = np.array([ x**2 for x in range(0, 5) ])
m = np.array([ [ 3, 1 ],
               [ 2, 4 ] ],
             float)
m[:, 1]           # => array([ 1., 4.])
m[1, :]           # => array([ 2., 4.])
m.max()           # => 4.0
m.max(axis=0)     # => array([ 3.,  4.])
m.max(axis=1)     # => array([ 3.,  4.])
```

Elemente der linearen Algebra

```
import numpy as np
```

- Liste: `l = np.array([1, 2, 3])`

Elemente der linearen Algebra

```
import numpy as np
```

- Liste: `l = np.array([1, 2, 3])`
- Vektor

$$\boldsymbol{a} = \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix}$$

```
a = np.array([[1, 2, 3]], float).T
# => array([[1.], [2.], [3.]])
```

Elemente der linearen Algebra

```
import numpy as np
```

- Matrix

$$\boldsymbol{A} = \begin{bmatrix} A_{1,1} & \dots & A_{1,j} & \dots & A_{1,n} \\ \vdots & & A_{i,j} & & \vdots \\ A_{m,1} & \dots & A_{m,j} & \dots & A_{m,n} \end{bmatrix}$$

```
np.array([ [1, 2], [3, 4] ])
# => array([[1, 2],
#           [3, 4]])
```

Elemente der linearen Algebra

```
import numpy as np
```

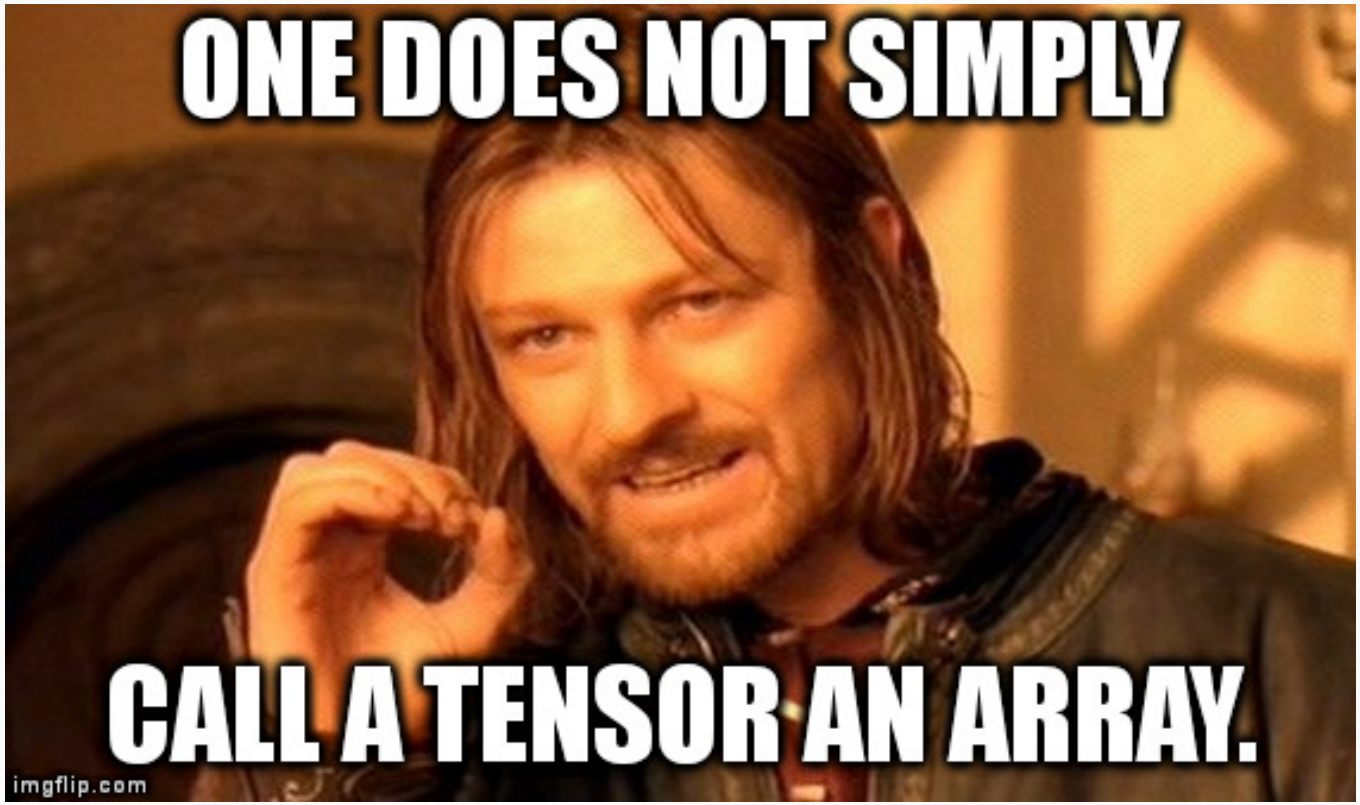
- Tensor. . .„nur“ eine Matrix mit > 2 Achsen

```
t = np.array([[[ 0,  1,  2],
               [ 3,  4,  5],
               [ 6,  7,  8]
              ],
              [[ 9, 10, 11],
               [12, 13, 14],
               [15, 16, 17]
              ],
              [[18, 19, 20],
               [21, 22, 23],
               [24, 25, 26]]])
t.shape     # => (3, 3, 3)
```

Elemente der linearen Algebra

```
import numpy as np
```

- Tensor. . . „nur“ eine Matrix mit > 2 Achsen



. . . nicht ganz wissenschaftlich korrekt. ;-)

Wichtigste Operatoren (1)

```
import numpy as np
```

- **Transposition**

$$\mathbf{x}^\top = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}^\top = \begin{bmatrix} x_1 & x_2 & x_3 \end{bmatrix}$$

```
x = np.array([ 1, 2, 3 ])
x.transpose()
```

$$\left(\mathbf{A}^\top\right)_{i,j} = A_{j,i}$$

```
m = np.matrix([[1.0, 2.0],
               [3.0, 4.0]])
m.transpose()
# => matrix([[1., 3.],
#            [2., 4.]])
```

Wichtigste Operatoren (2)

```
import numpy as np
```

- **Skalarmultiplikation**

$$a \cdot \boldsymbol{x} = \begin{bmatrix} a \cdot x_1 & a \cdot x_2 & \dots & a \cdot x_n \end{bmatrix}^\top$$

```
x = np.array([ 1.0, 2.0, 3.0 ])
x * 5
# => array([ 5.0, 10.0, 15.0 ])
```

Wichtigste Operatoren (3)

```
import numpy as np
```

- **Matrix-Vektor-Produkt**

$$\boldsymbol{Ax} = \begin{bmatrix} A_{1,1} & \dots & A_{1,j} & \dots & A_{1,n} \\ \vdots & & A_{i,j} & & \vdots \\ A_{m,1} & \dots & A_{m,j} & \dots & A_{m,n} \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}$$

$$= \begin{bmatrix} A_{1,1}x_1 & \dots & A_{1,j}x_j & \dots & A_{1,n}x_n \\ \vdots & & A_{i,j}x_j & & \vdots \\ A_{m,1}x_1 & \dots & A_{m,j}x_j & \dots & A_{m,n}x_n \end{bmatrix}$$

```
m = np.array([ [1.0, 1.0], [1.0, 1.0] ])
v = np.array([ 2.0, 2.0 ])
m * v
# => array([ [2.0, 2.0],
#            [2.0, 2.0] ])
```

Wichtigste Operatoren (4)

```
from numpy import array
```

- **Matrixprodukt**

$$\mathbf{C} = \boldsymbol{AB}$$

$$C_{i,j} = \sum_k A_{i,k} B_{k,j}$$

$$\mathbf{A} \in \mathbb{R}^{m \times n}, \boldsymbol{B} \in \mathbb{R}^{n \times p} \quad \rightarrow \quad \boldsymbol{C} \in \mathbb{R}^{m \times p}$$

```
a = array([[1., 2.], [3., 4.]])
b = array([[2., 2.], [2., 2.]])
a @ b
# => array([ [6.,  6.  ],
#            [14., 14. ]])
```

Wichtigste Operatoren (4)

```
from numpy import array
```

- **Hadamard-Produkt**

$$\boldsymbol{C} = \boldsymbol{A} \odot \boldsymbol{B}$$

$$C_{i,j} = A_{i,j} \cdot B_{i,j}$$

```
a = array([[1., 2.], [3., 4.]])
b = array([[2., 2.], [2., 2.]])
a * b
# => array([ [2., 4.],
#            [6., 8.] ])
```

Initialisierungsfunktionen für NumPy-Arrays

- Gefüllt mit **Nullen**

```
np.zeros((2, 2))        # => array([[0., 0.],
                        #           [0., 0.]])
```

- Gefüllt mit **Einsen**

```
np.ones((2, 2))         # => array([[1., 1.],
                        #           [1., 1.]])
```

- Gefüllt mit **konstantem Wert**

```
np.full((2, 2), 42.)    # => array([[42., 42.],
                        #           [42., 42.]])
```

- **Identitätsmatrix**

```
np.eye(2)               # => array([[1., 0.],
                        #           [0., 1.]])
```

- Mit **Zufallszahlen** gefüllt

```
np.random.random(2)
```

Datenexploration

```
import h5py
import pandas as pd
import matplotlib.pyplot as plt

data_file = h5py.File(
    'data/simulation_data.hdf5',
    'r')
p_res = data_file['commercial/P'][()]
data = pd.DataFrame(p_res)
```

- Ca. ⅓ der Daten als Testdaten sofort segmentieren und nicht ansehen (*Data Bias*)

```
In [5]: 1 data.head()
```

Out[5]:

	0
0	609.0
1	603.0
2	599.0
3	614.0
4	606.0

```
In [6]: 1 data.describe()
```

Out[6]:

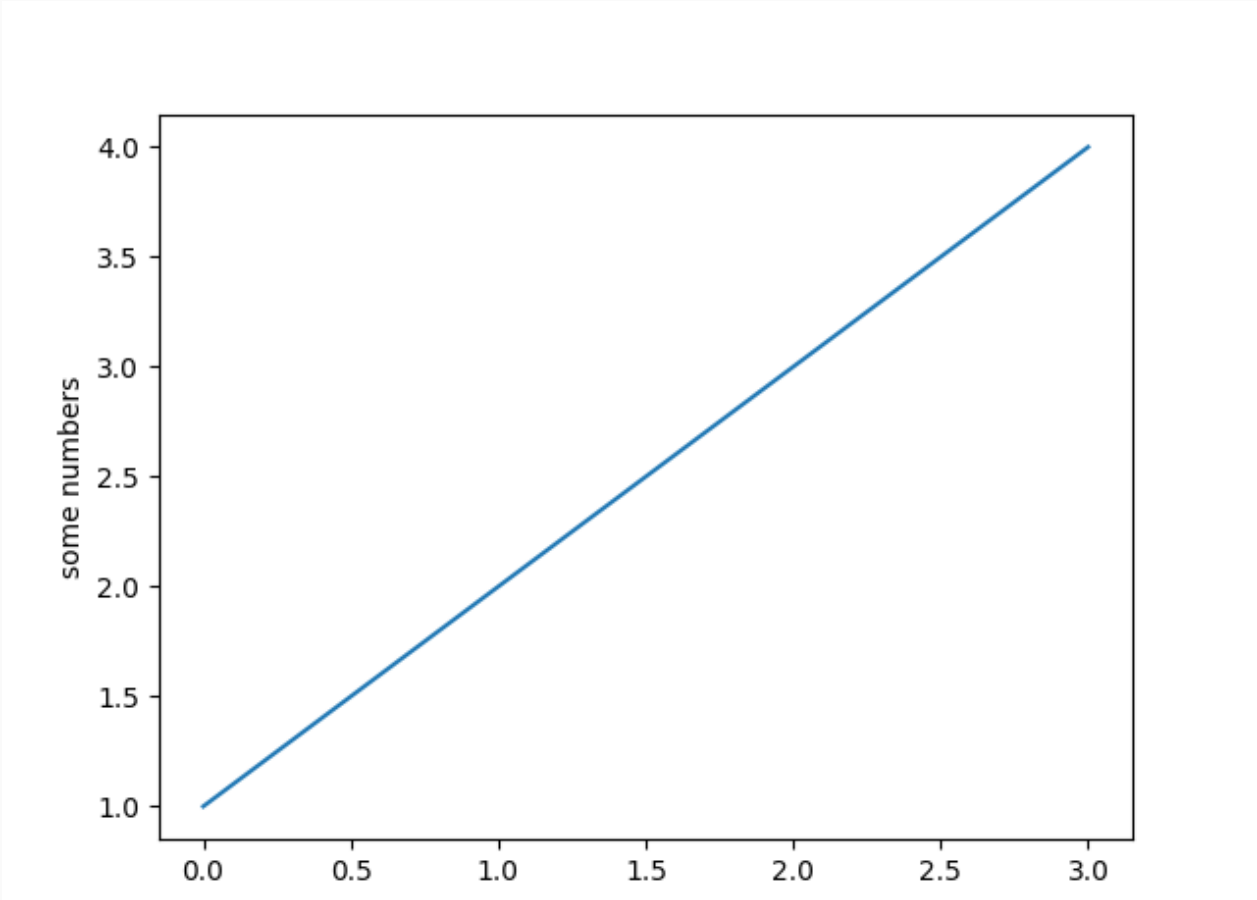
	0
count	35040.000000
mean	615.043522
std	139.995843
min	413.000000
25%	503.000000
50%	569.000000
75%	715.000000
max	1108.000000

Schöne Graphen mit Matplotlib

```
import matplotlib.pyplot as plt
```

- Grundfunktion: `plt.plot(·)`

```
plt.plot([1, 2, 3, 4])
plt.ylabel('some numbers')
```

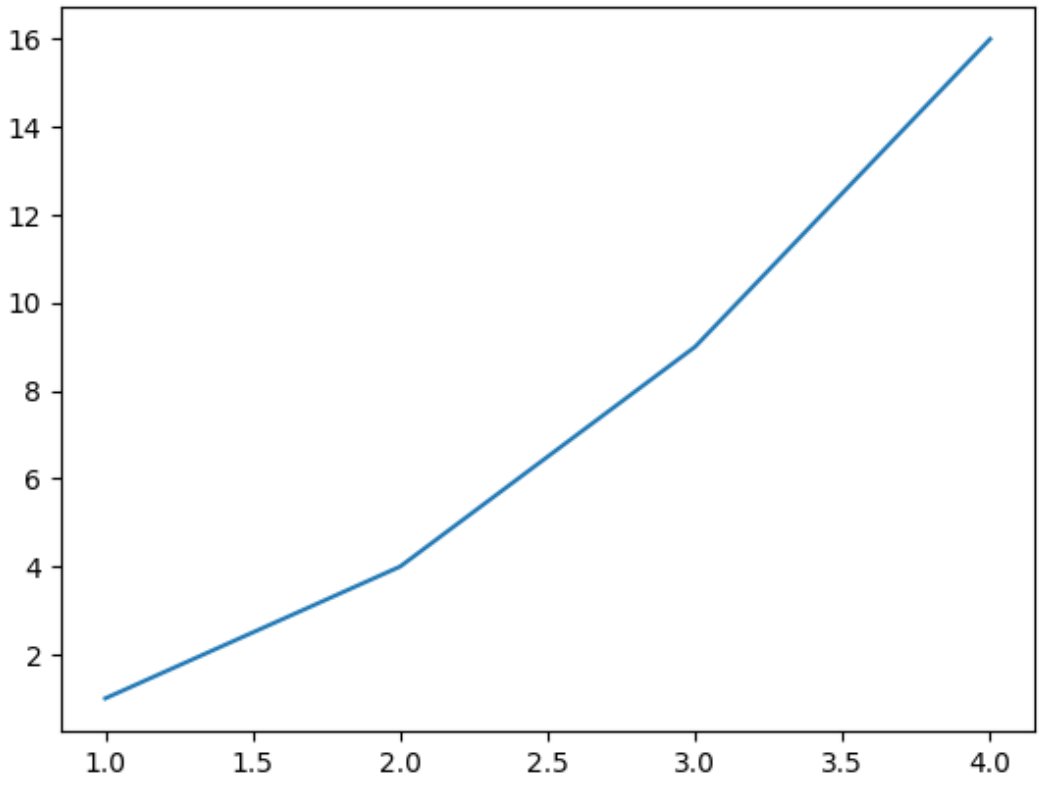


Schöne Graphen mit Matplotlib

```
import matplotlib.pyplot as plt
```

- Grundfunktion: `plt.plot(·)`
- Darstellung von x gegen y

  `plt.plot([1, 2, 3, 4], [1, 4, 9, 16])`



Graphen mit Matplotlib

```
ts = pd.date_range('2008-01-01', periods=len(p_res),
                   freq=pd.DateOffset(minutes=15))
# => DatetimeIndex(['2008-01-01 00:00:00',
                    '2008-01-01 00:15:00', ...])
fig, ax = plt.subplots(2, 1, figsize=(20, 12))
_ = ax[0].plot_date(ts[train_begin * 96:train_end * 96],
                    p_res_train)
_ = ax[1].plot_date(ts[test_begin * 96:test_end * 96],
                    p_res_test)
```

Graphen mit Matplotlib

Apache Spark

MapReduce vs. Spark

- Unterschiedliche Paradigmen zur Datenverarbeitung
- MapReduce:
  - Azyklischer, gerichteter Graph
  - Stapelverarbeitung
  - Einmalige Iteration eines Datensatzes pro Job
- Spark:
  - *Resilient Distributed Datasets & Data Frames*
  - In-Memory-Abfragen
  - Datenstromorientiert
  - Gut für Algorithmen des maschinellen Lernens
- . . . keines der beiden Paradigmen eindeutig besser als das andere

Vom *Resilient Distributed Dataset* (RRD) zum *Data Frame*

Resilient Distributed Dataset

- Ursprüngliche Basis von Spark
- Redundant gespeichert, leicht wiederherstellbar
- Untypisiert

Data Frame

- Typisiert
- Dadurch SQL-artige Abfragen möglich
- Optimierung im Speicher

Ingestion Layer mit Spark

- **Veracity** & **Value**:
  1. Daten vorverarbeiten: Ausreißer säubern, fehlende Werte imputieren, ggfs. normieren/zentrieren, ...
  2. Verwertung zuführen: ML-Algorithmen benötigen *immer* vorverarbeitete Daten (bspw. äquidistante Zeitreihen)
- Spark arbeitet auf **Mini Batches**
- Eigene Variante von SQL, **Spark SQL**
- Klassen für maschinelles Lernen (**Spark ML**), z. B.
  - Scaler
  - Imputer
  - PCA
  - ...

„Hello, World!“ mit Apache Spark

Shakespeares Hamlet:

```
wget -O hamlet.txt
 ↪  http://www.gutenberg.org/cache/epub/1787/pg1787.txt
spark-shell --master local
```

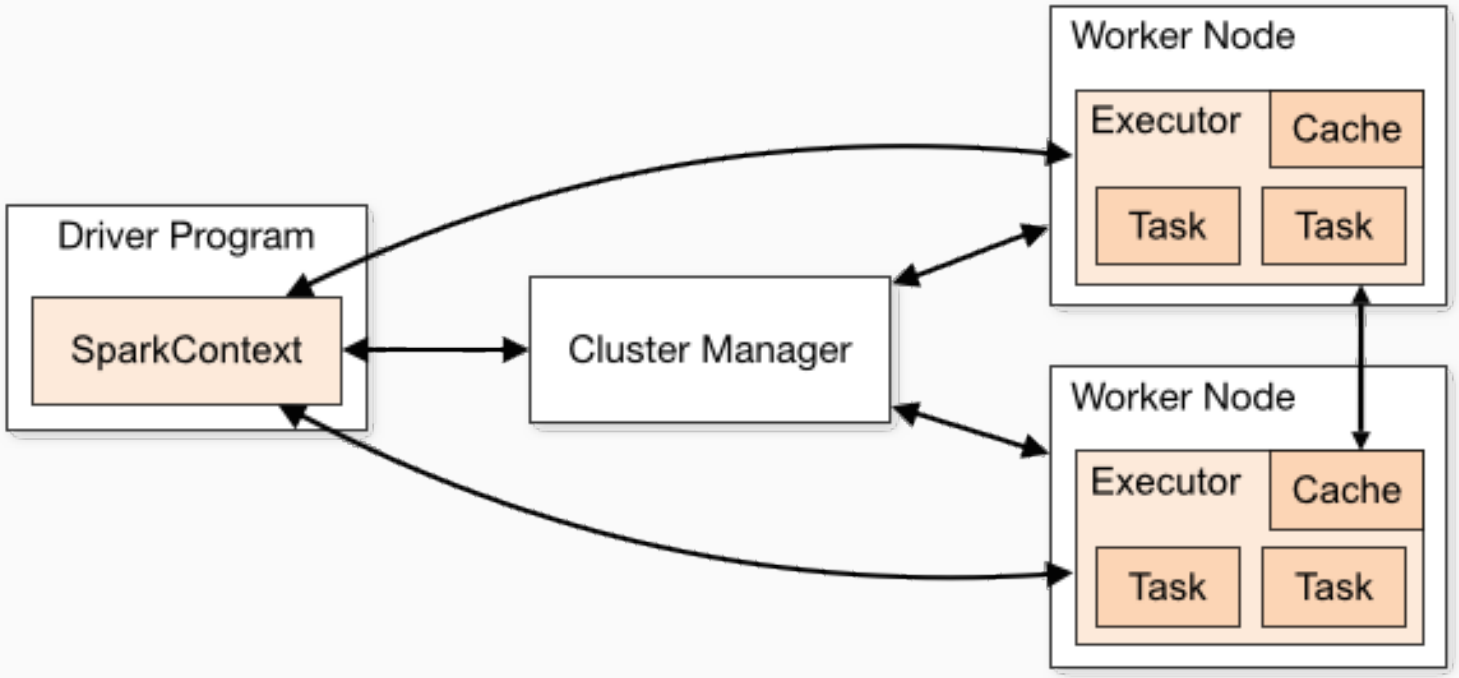
Häufigkeit einzelner Wörter, absteigend sortiert:

```
val tf = sc.textFile("hamlet.txt")
val tokenized = tf.flatMap(line => line.split("\\W+"))
val kvMap = tokenized.map(word => (word.toLowerCase(), 1))
val wCounts = kvMap.reduceByKey((acc, v) => acc + v)
val sortedCounts = wCounts.sortBy(kv => kv._2, false)
sortedCounts.saveAsTextFile("words-in-hamlet.md")
```

„Hello World“ in Apache Spark

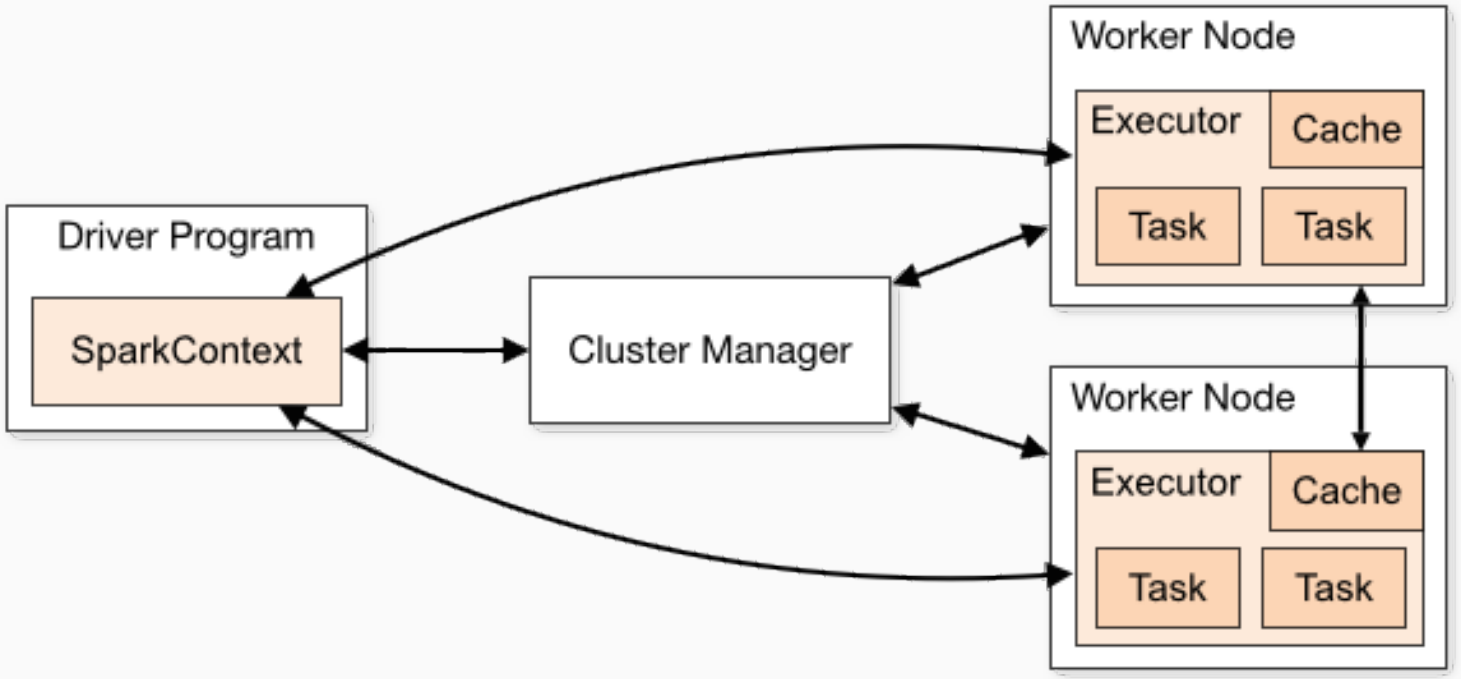
```
from pyspark import SparkContext

sc = SparkContext("local", "hello world app")
tf = sc.textFile("hamlet.txt")
tokenized = tf.flatMap(lambda line: line.split("\\W+")) \
      .map(lambda word: (word.toLowerCase(), 1)) \
      .kvMap.reduceByKey(lambda acc, v: acc + v) \
      .sortBy(lambda kv: kv._2, false) \
      .saveAsTextFile("words-in-hamlet.md")
```

Die Architektur von Apache Spark



Apache Spark Project. “Cluster Mode Overview.”

Die Architektur von Apache Spark

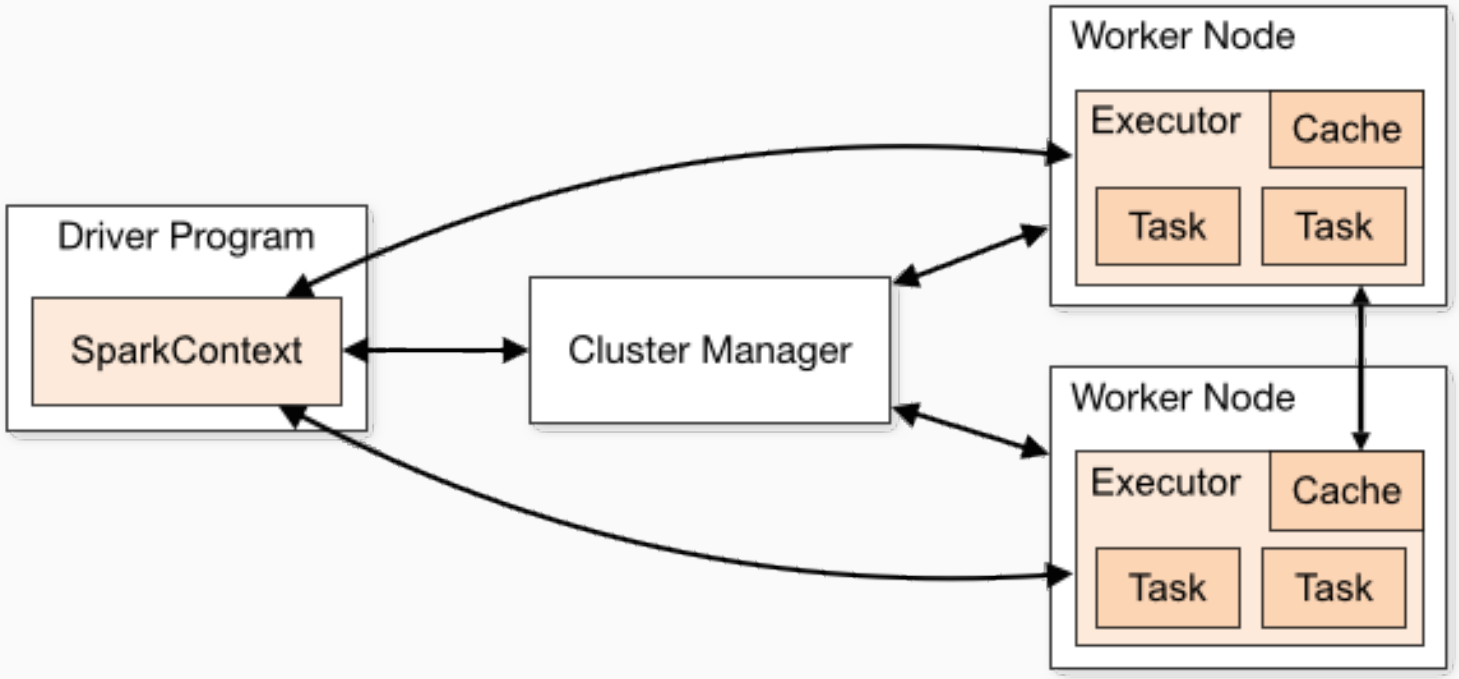


Apache Spark Project. "Cluster Mode Overview."

Executor: 1 pro Anwendungsinstanz (Isolation); mehrere Tasks pro *Executor*

```
sc = SparkContext("local", "app")   # Singleton
```

Die Architektur von Apache Spark

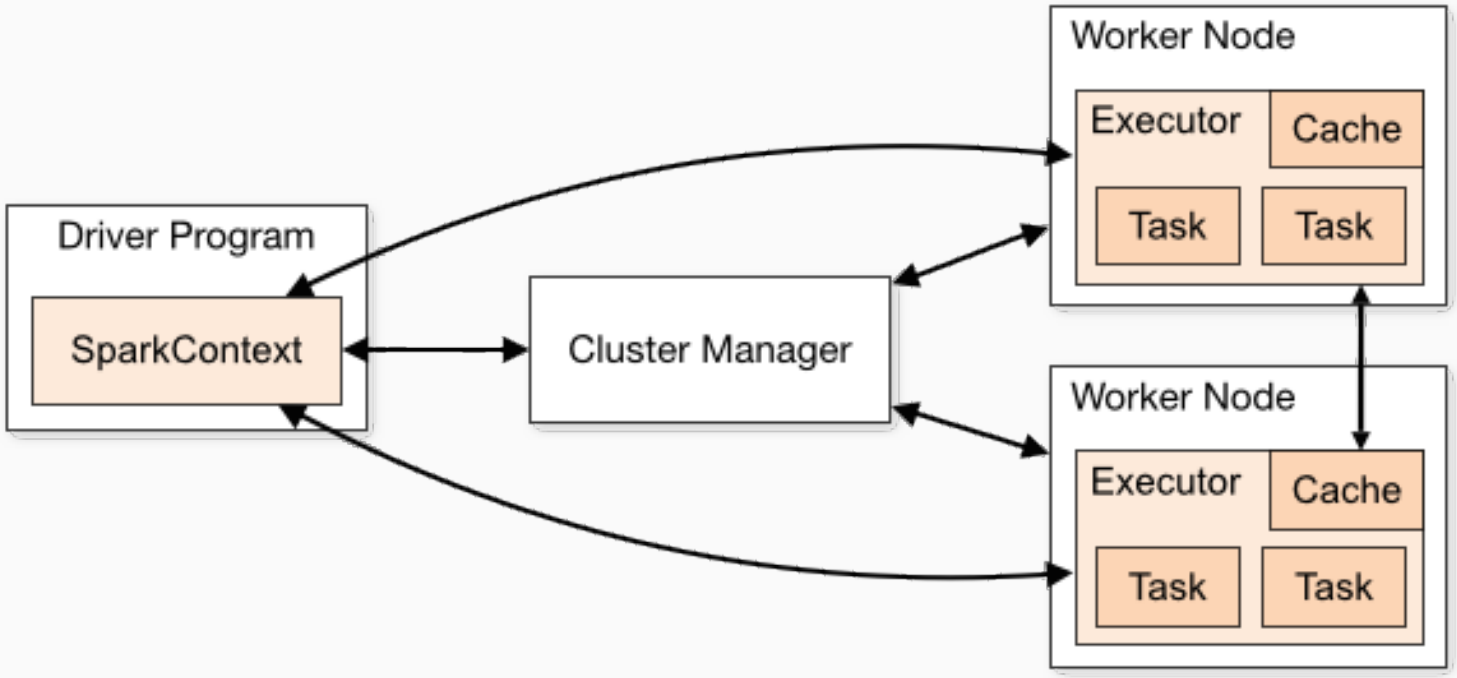


Apache Spark Project. "Cluster Mode Overview."

Driver Program: Koordiniert die *Executor*-Prozesse auf den *Worker Nodes* vom Hauptprogramm aus (*Scheduler*

```
sc = SparkContext("local", "app")   # Singleton
```

Die Architektur von Apache Spark



Apache Spark Project. "Cluster Mode Overview."

Cluster Manager: Startet & beendet die *Executor*-Prozesse
Spark Standalone, Apache Mesos, Hadoop YARN, Kubernetes

Prozesse auf einem Spark-Cluster ausführen i

Lokal ausführen, 8 CPU-Kerne:

```
./bin/spark-submit \
  --class org.apache.spark.examples.SparkPi \
  --master local[8] \
  /path/to/examples.jar \
  100
```

Prozesse auf einem Spark-Cluster ausführen ii

Standalone Cluster:

```
./bin/spark-submit \
  --class org.apache.spark.examples.SparkPi \
  --master spark://207.184.161.138:7077 \
  --executor-memory 20G \
  --total-executor-cores 100 \
  /path/to/examples.jar \
  1000
```

Prozesse auf einem Spark-Cluster ausführen iii

Hadoop YARN:

```
export HADOOP_CONF_DIR=XXX
./bin/spark-submit \
  --class org.apache.spark.examples.SparkPi \
  --master yarn \
  --deploy-mode cluster \  # can be client for client mode
  --executor-memory 20G \
  --num-executors 50 \
  /path/to/examples.jar \
  1000
```

Prozesse auf einem Spark-Cluster ausführen iv

Python-Skript, Standalone Cluster:

```
./bin/spark-submit \
  --master spark://207.184.161.138:7077 \
  examples/src/main/python/pi.py \
  1000
```

Prozesse auf einem Spark-Cluster ausführen v

Kubernetes:

```
./bin/spark-submit \
  --class org.apache.spark.examples.SparkPi \
  --master k8s://xx.yy.zz.ww:443 \
  --deploy-mode cluster \
  --executor-memory 20G \
  --num-executors 50 \
  http://path/to/examples.jar \
  1000
```

Das *Data-Frame*-Konzept

- **Dataset:** Verteilte Ansammlung von Daten
- **Data Frame:** Dataset, geordnet nach benannten Spalten
- Vielfältige **Datenquellen**:
  - **Parquet** (Standard):
    ```
    df = spark.read.load("users.parquet")
    ```
  - **JSON:**
    ```
    spark.read.load("people.json", format="json")
    ```
  - **CSV:** `spark.read.load("people.csv", format="csv", sep=":", inferSchema="true", header="true")`
  - Hive
  - SQL (via JDBC)

Zugriff auf Hive i

```
from pyspark.sql import Row
from pyspark.sql import SparkSession
from os.path import expanduser, join, abspath

# warehouse_location points to the default location for managed
 ↪  databases and tables
warehouse_location = abspath('spark-warehouse')

spark = SparkSession \
    .builder \
    .appName("Python Spark SQL Hive integration example") \
    .config("spark.sql.warehouse.dir", warehouse_location) \
    .enableHiveSupport() \
    .getOrCreate()
```

Zugriff auf Hive ii

```
# spark is an existing SparkSession
spark.sql("CREATE TABLE IF NOT EXISTS src (key INT, value STRING)
 ↪  USING hive")
spark.sql("LOAD DATA LOCAL INPATH
 ↪  'examples/src/main/resources/kv1.txt' INTO TABLE src")

# Queries are expressed in HiveQL
spark.sql("SELECT * FROM src").show()
# +---+-------+
# |key|  value|
# +---+-------+
# |238|val_238|
# | 86| val_86|
# |311|val_311|
# ...
```

Spark und JDBC i

```
jdbcDF = spark.read \
    .format("jdbc") \
    .option("url", "jdbc:postgresql:dbserver") \
    .option("dbtable", "schema.tablename") \
    .option("user", "username") \
    .option("password", "password") \
    .load()

jdbcDF2 = spark.read \
    .jdbc("jdbc:postgresql:dbserver", "schema.tablename",
          properties={"user": "username", "password":
           ↪  "password"})

jdbcDF.write \
    .format("jdbc") \
```

Spark und JDBC ii

```
    .option("url", "jdbc:postgresql:dbserver") \
    .option("dbtable", "schema.tablename") \
    .option("user", "username") \
    .option("password", "password") \
    .save()

jdbcDF2.write \
    .jdbc("jdbc:postgresql:dbserver", "schema.tablename",
          properties={"user": "username", "password":
           ↪  "password"})
```

Data Frames – eine kurze Einführung

- Spaltenzugriff über Index- oder Punkt-Operator:

```
nameCol = df.name
nameCol = df["name"]
```

- SQL-artige Verarbeitungsweise:

```
people.filter(people.age > 30) \
    .join(department, people.deptId == department.id) \
    .groupBy(department.name, "gender") \
    .agg({"salary": "avg", "age": "max"})
```

Structured Streaming

- Arbeitsweise wie bei Stapelverarbeitung, aber für Datenströme
- Lösung: *Mini-Batches*
- Ende-zu-Ende-Latenz 100 ms möglich

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import explode
from pyspark.sql.functions import split

spark = SparkSession.builder \
    .appName("StructuredNetworkWordCount") \
    .getOrCreate()
lines = spark.readStream.format("socket") \
   .option("host", "localhost").option("port", 9999) \
   .load()

words = lines.select(explode(split(lines.value, " ")).alias("word"))
wordCounts = words.groupBy("word").count()
```

Imputer

- In Sensor-Datenströmen können Werte fehlen (Sensorausfall, etc.)
- Existierende Werte als Stützwerte
- Imputationsstrategie:
  - Durchschnitt (Voreinstellung — Achtung, begünstigt Ausreißer!)
  - Median
- Ablauf: `fit(·) -> model`, dann `model.transform(df)`

Vorverarbeitung mit Apache Spark

```
# ...
from pyspark.ml.feature import Imputer

df = spark.readStream.format("kafka") \
          .option("kafka.bootstrap.servers",
                  "host1:port1,host2:port2") \
          .option("subscribe", "raw_sensor_1") \
          .load()
df.selectExpr("CAST(key AS STRING)", "CAST(value AS STRING)")

imputer = Imputer(inputCols=["value"], outputCols=["value_out"])
model = imputer.fit(df)
model.transform(df).show()
```

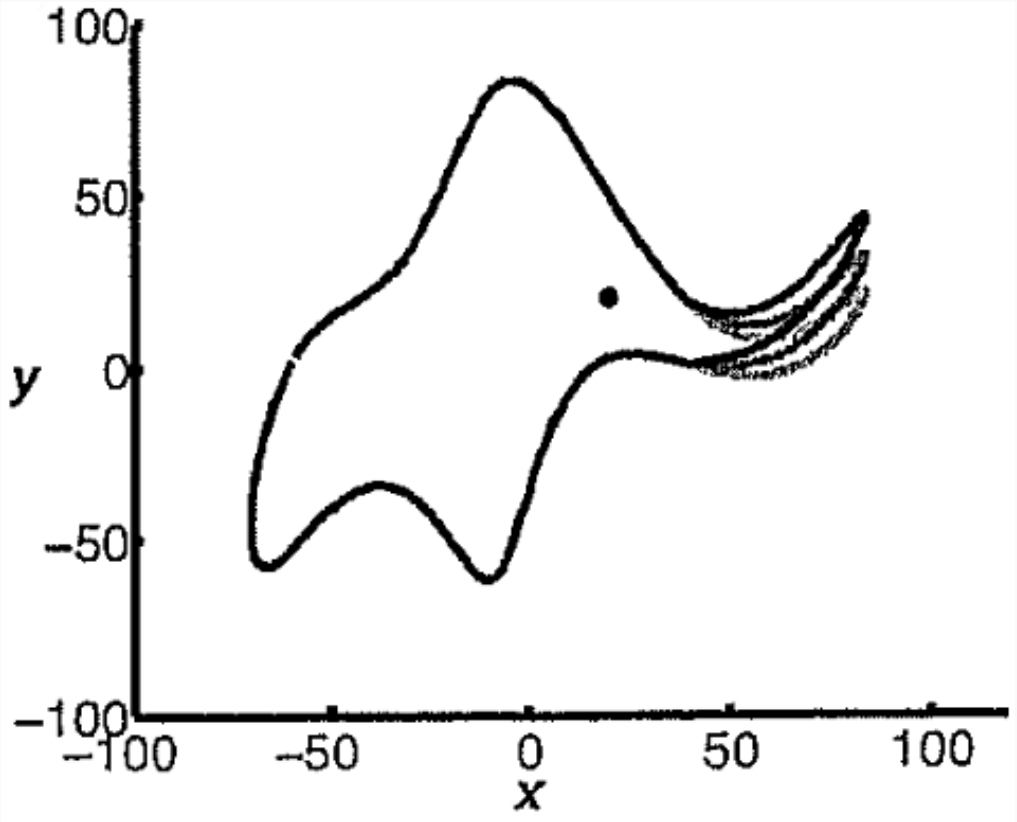
Dimensionalitätsreduktion

- Reduktion der Anzahl der Variablen in einem System
- Ein n Datenpunkte mit p Merkmalen: n Punkte in $\mathbb{R}^p$
- Ziel: Projektion in einen Unterraum $\mathbb{R}^q$ ($q < p$), so das möglichst wenig Informationen verloren gehen
- Hilft ML-Algorithmen
- Zwei Methoden in Spark

  SVD *Singular Value Decomposition* (Singulärwertzerlegung)

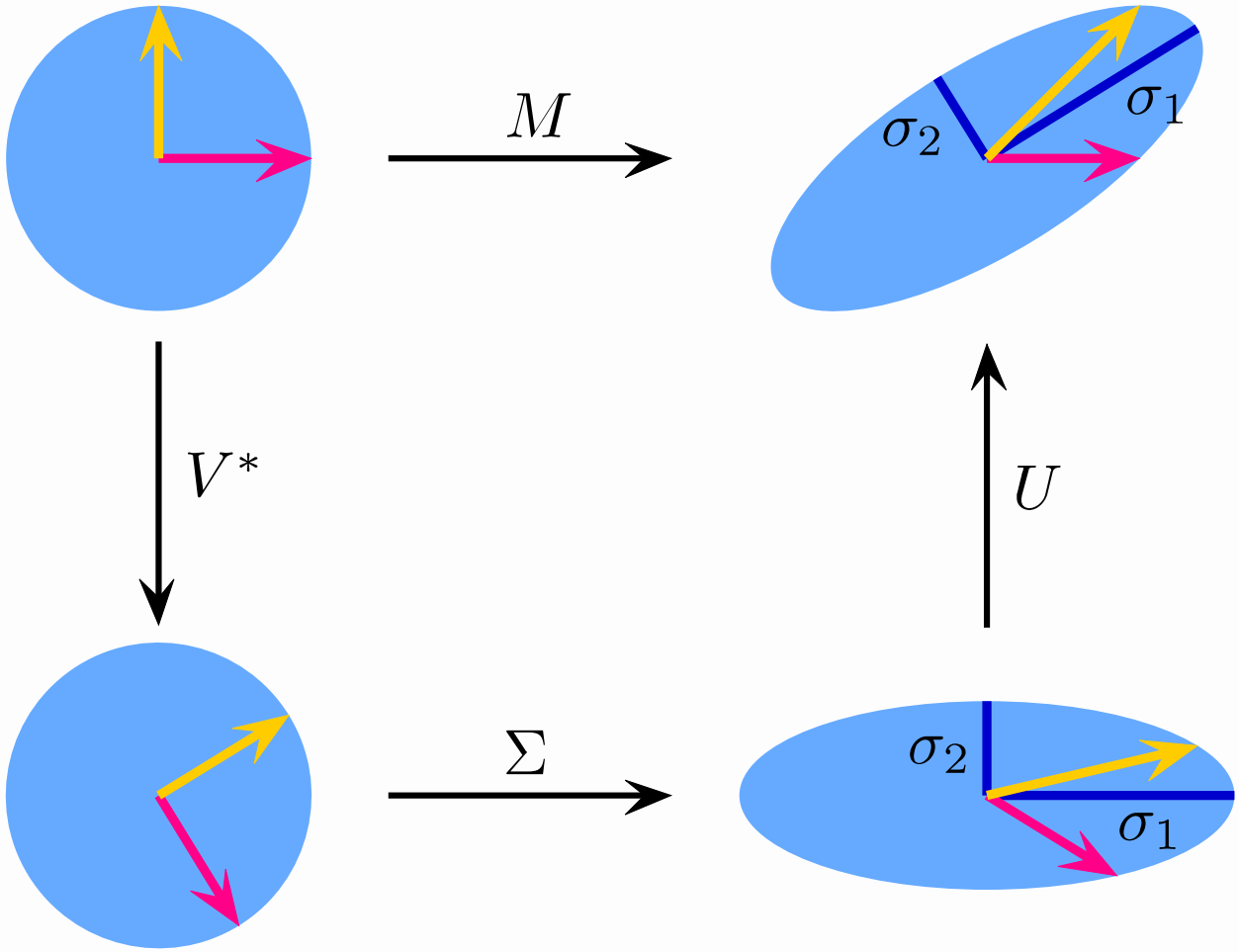
  PCA *Principal Component Analysis* (Hauptkomponentenanalyse, HKA)

Mehr Parameter machen kein besseres Modell



*With **four** parameters, I can **draw an elephant**,*
*and with **five**, I can **make ihm wiggle his trunk**.*
— John von Neumann

Singular Value Decomposition



$$M = U \cdot \Sigma \cdot V^*$$

Singular Value Decomposition

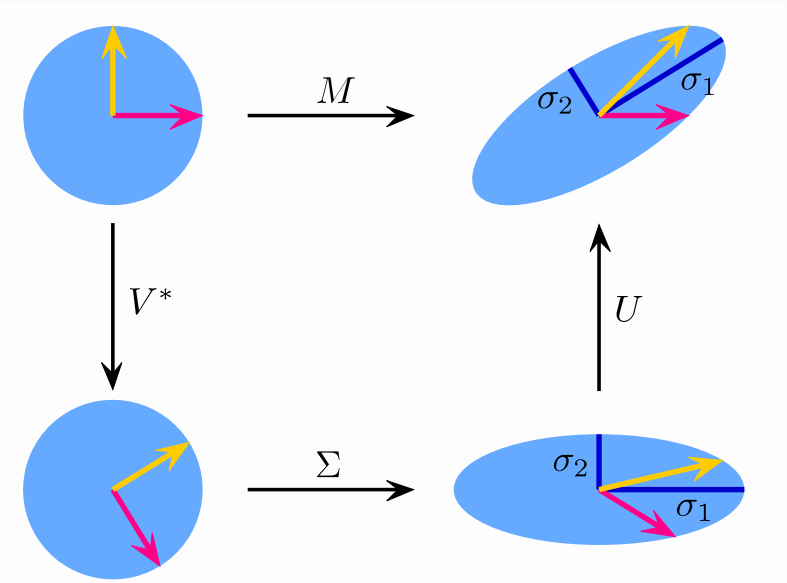
- $\mathbf{M}$: $m \times n$-Matrix
- $\mathbf{U}$: Unitäre $m \times m$-Matrix

  Unitär: Zeilen- & Spaltenvektoren orthogonal bzgl. Standardskalarprodukt; $\mathbf{U}^H\mathbf{U} = \mathbf{I}$
- $\mathbf{V}^*$: Adjungierte einer unitären $n \times n$-Matrix
- $\mathbf{\Sigma}$: Reelle $m \times n$-Matrix, so dass:

$$\Sigma = \left(\begin{array}{ccc|ccc} \sigma_1 & & & & \vdots & \\ & \ddots & & \cdots & 0 & \cdots \\ & & \sigma_r & & \vdots & \\ \hline & \vdots & & & \vdots & \\ \cdots & 0 & \cdots & \cdots & 0 & \cdots \\ & \vdots & & & \vdots & \end{array}\right)$$

Singular Value Decomposition

- $\mathbf{M} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^*$ berechnen
- k oberen Eigenwerte behalten:
  - $\mathbf{U} : m \times k$
  - $\mathbf{\Sigma} : k \times k$
  - $\mathbf{V} : n \times k$
- Ergebnis: $\mathbf{M}$ mit weniger Variablen dargestellt



$$M = U \cdot \Sigma \cdot V^*$$

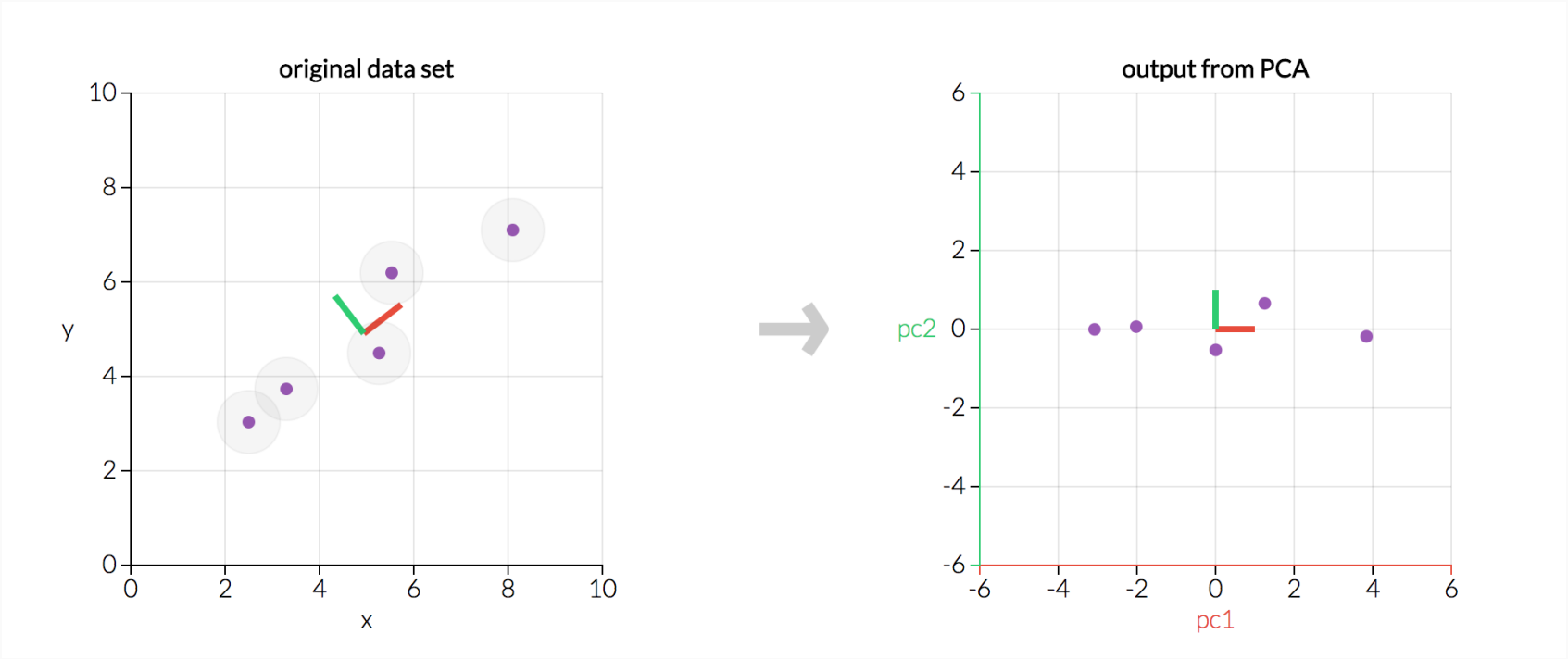
SVD in Spark mit Python

```
from pyspark.mllib.linalg import Vectors
from pyspark.mllib.linalg.distributed import RowMatrix

rows = sc.parallelize([
    Vectors.sparse(5, {1: 1.0, 3: 7.0}),
    Vectors.dense(2.0, 0.0, 3.0, 4.0, 5.0),
    Vectors.dense(4.0, 0.0, 0.0, 6.0, 7.0)
])
mat = RowMatrix(rows)

# SVD mit k = 5:
svd = mat.computeSVD(5, computeU=True)
U = svd.U  # RowMatrix.
s = svd.s  # Vektor der Eigenwerte
V = svd.V
```

Hauptkomponentenanalyse

- Reduktion der Anzahl der Variablen in einem System
- Ein n Datenpunkte mit p Merkmalen: n Punkte in $\mathbb{R}^p$
- Ziel: Projektion in einen Unterraum $\mathbb{R}^q$ ($q < p$), so das möglichst wenig Informationen verloren gehen
- Redundanzen in Form von Korrelation zusammenfassen
- Erste Achse für Daten mit der höchsten Varianz, zweite Achse für zweithöchste, etc.
- Annahme: Hohe Varianz entspricht hohem Informationsgehalt
- Mittel: **Hauptachsentransformation**

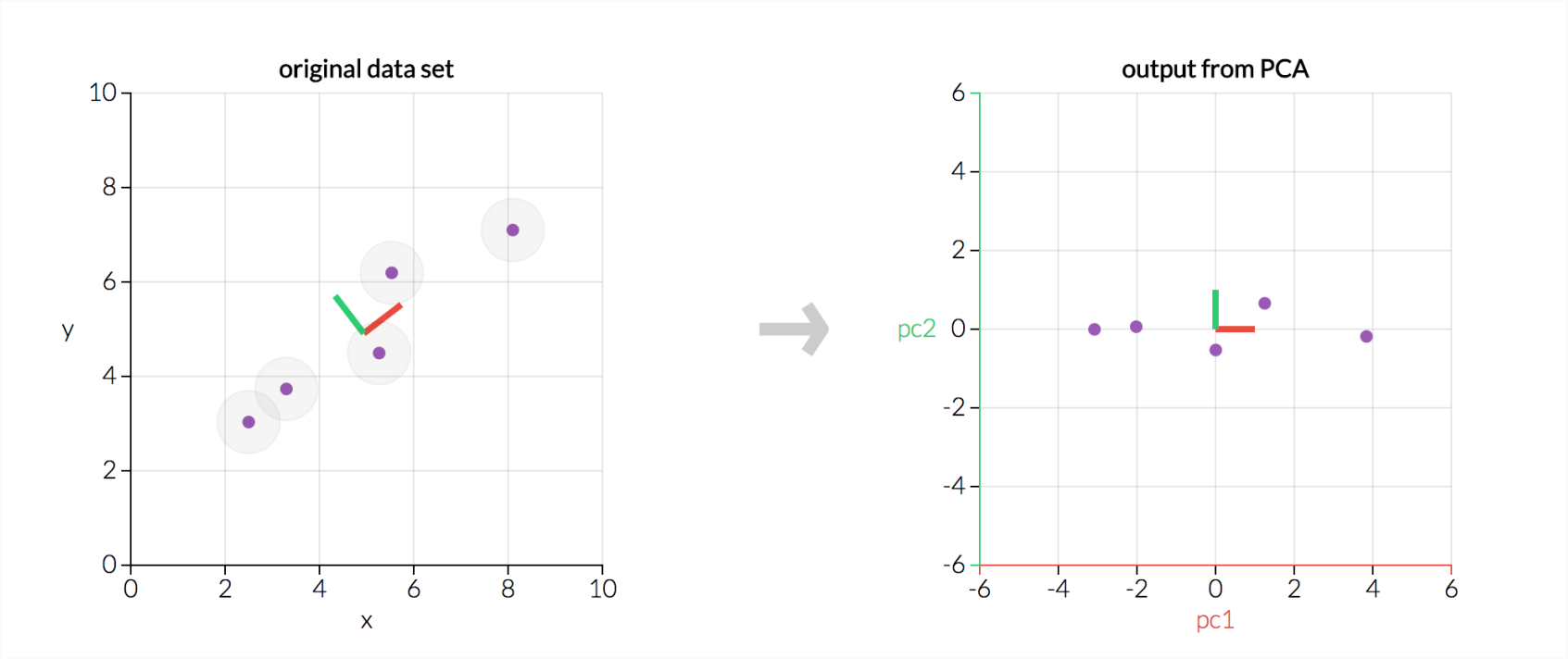
“PCA in a Picture”



Matt Brems. “A One-Stop Shop for Principal Component Analysis.”

- Welche Achse ist wichtiger? (Wie würde eine Linie durch die Punkte liegen?)

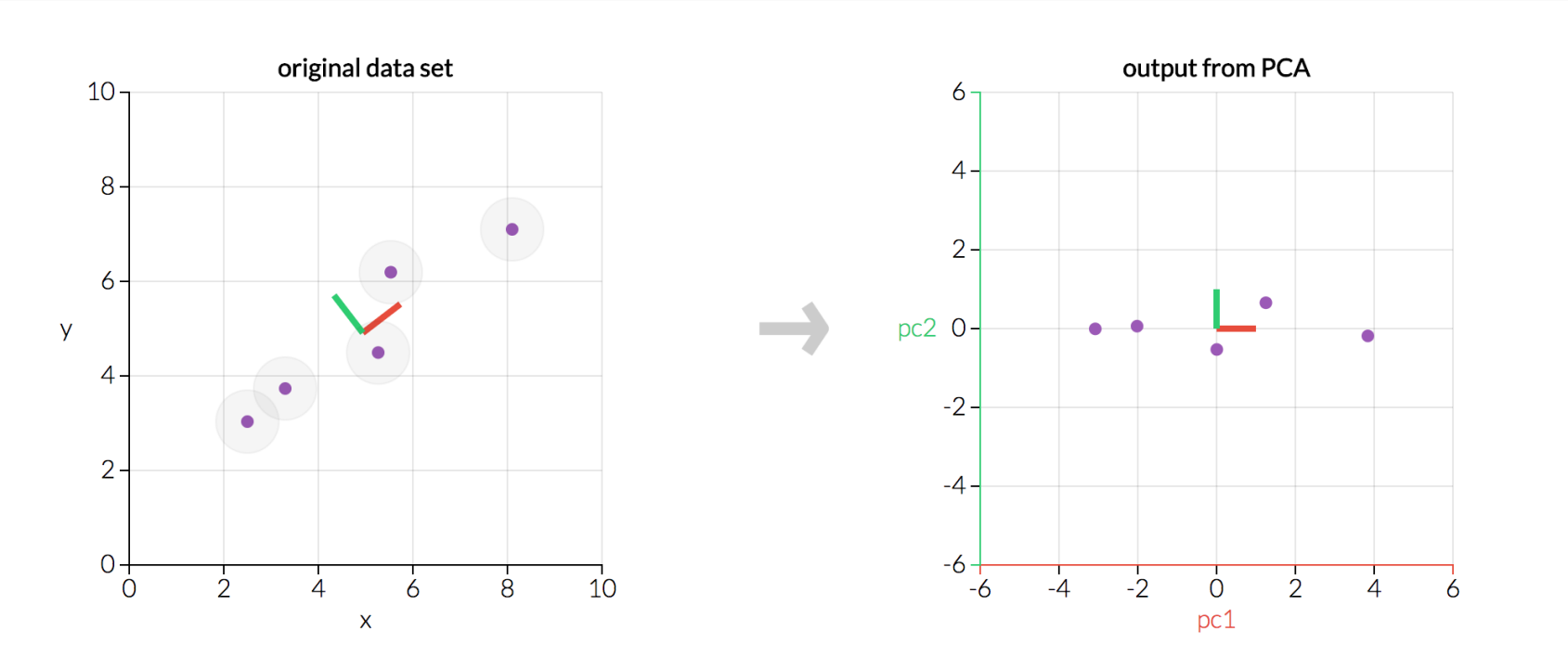
“PCA in a Picture”



Matt Brems. “A One-Stop Shop for Principal Component Analysis.”

- $\mathbf{X}$, spaltenweise zentriert: $\mathbf{Z}$
- $\mathbf{Z}^\top\mathbf{Z}$: Covarianzmatrix – Zusammenhang jeder Variable in $\mathbf{Z}$ mit jeder anderen Variable in $\mathbf{Z}$

“PCA in a Picture”



Matt Brems. “A One-Stop Shop for Principal Component Analysis.”

- Eigenvektoren aus $\mathbf{Z}^\top \mathbf{Z}$: bestimmen Richtung

PCA in Spark mit Python

```python
from pyspark.mllib.linalg import Vectors
from pyspark.mllib.linalg.distributed import RowMatrix

rows = sc.parallelize([
    Vectors.sparse(5, {1: 1.0, 3: 7.0}),
    Vectors.dense(2.0, 0.0, 3.0, 4.0, 5.0),
    Vectors.dense(4.0, 0.0, 0.0, 6.0, 7.0)
])

mat = RowMatrix(rows)
pc = mat.computePrincipalComponents(4)  # 4 Komponenten

# Ergebnis: Die Projektion
projected = mat.multiply(pc)
```

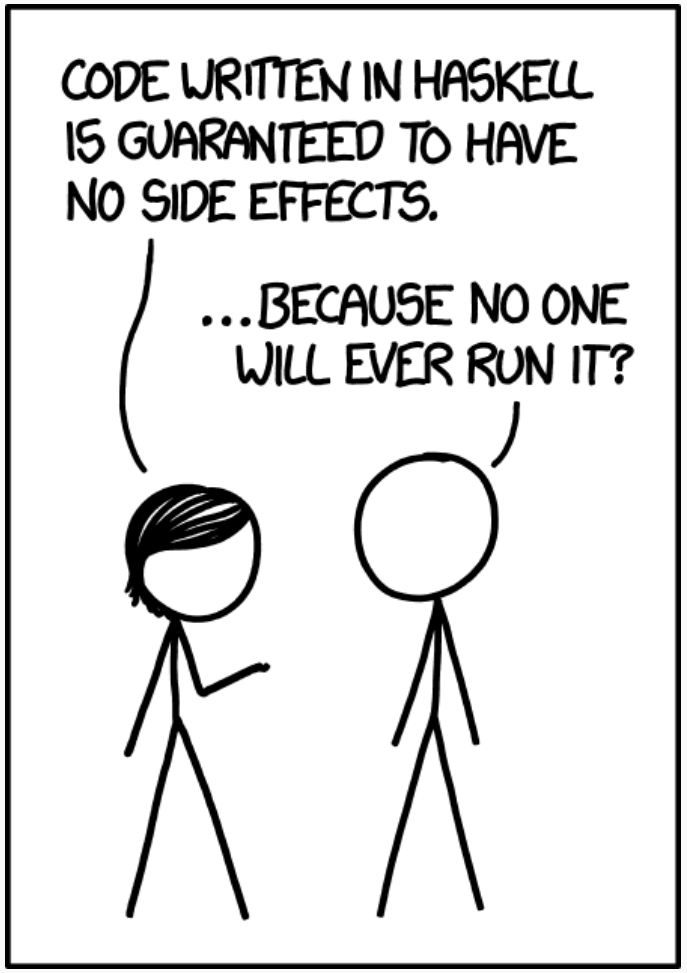
Funktionale Programmierung

Imperativ, objektorientiert – funktional?

Imperative Programmierung Befehl im Mittelpunkt, linearer Programmablauf (mit Verzweigungen)

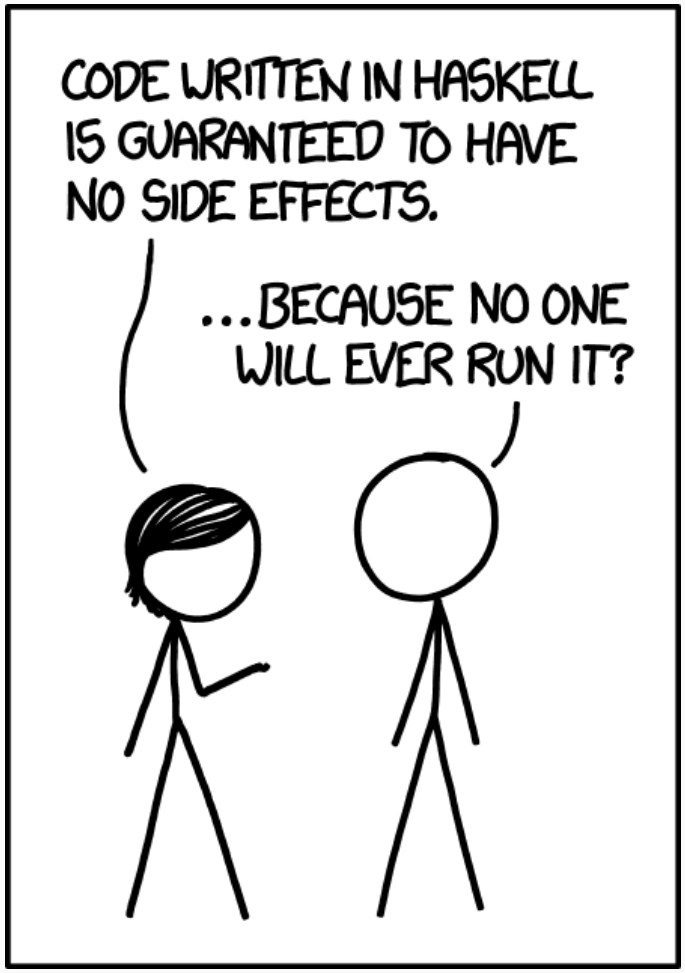
Objektorientierte Programmierung Klassen, Objekte und die Interaktion von Objekten im Vordergrund

Funktionale Programmierung Funktionsbegriff (im mathematischen Sinne!) im Mittelpunkt



Imperativ, objektorientiert – funktional!

- Funktionen sind seiteneffektfrei
- Kein Verwalten eines internen Zustands
- Vorteil: Algorithmen ohne Rücksicht auf die Beschaffenheit von Daten beschreibbar
- Wichtigster praktischer Vertreter: Haskell
- (Fast) Jede Programmiersprache hat funktionale Teile (Multi-Paradigmen-Sprachen)
- Gutes Paradigma für Big Data!



Funktionale Programmierung: Definition

1. Programme sind Funktionen
2. Die Ausgabe von Funktionen ist nur abhängig von ihrer Eingabe
3. Funktionen speichern keinen internen Zustand
4. Alle Funktionen sind idempotent (!)
5. Funktionen sind ineinander verschachtelte Funktionen (keine Befehle!)
6. Funktionen sind gegenüber allen Datenobjekten gleichberechtigt (Ein-/Ausgaben gleich; Funkionen sind auch Datenobjekte)
7. Funktionen benötigen keinen Namen (*Lambdas*)

Datenstrukturen in der funktionalen Programmierung

- Wichtigste Datenstrukturen neben Funktionen:
  - Bäume
  - Listen (die Grunddatenstruktur schlechthin!)
- Sei A ein Datentyp
- Sei A^* eine beliebig lange Liste des Typs A:

$$A^* = Nil | Cons(A, A^*) \tag{1}$$

- *Nil*: Leere Liste
- *Cons*: Operation zum „Anhängen“:

$$Cons : A \times A^* \mapsto A^* \tag{2}$$

Katamorphismus

- Katamorphismus: Kata (gr. entlang, herab); Morphe (gr. Form)
- Katamorphismen zerlegen Listen und ermitteln dabei einen Wert
- B: Datentyp
- $\otimes A \times B \mapsto B$: Abbildung
- Katamorphismus:

$$h : A^* \mapsto B$$
$$Nil \mapsto b$$
$$Cons(a, L) \mapsto a \otimes h(L)$$

- Übliche Notation: $(|b, \otimes|)$
- In funktionalen Programmiersprachen: `reduce(·)` oder `fold(·)`

Katamorphismus

$$h : A^* \mapsto B$$
$$Nil \mapsto b$$
$$Cons(a, L) \mapsto a \otimes h(L)$$

$x := b$
for $i := Length(L)$ downto 1
do
  $x := L_i \otimes x$
end for
return x

Viele grundlegende Funktionen sind Katamorphismen:

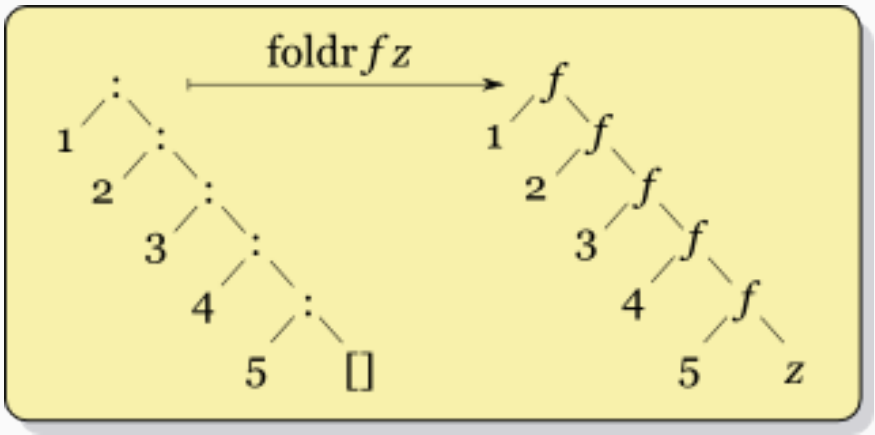
$(|0, +|)$ Summe einer Zahlenliste

$(|\eta, \cdot|)$ Verkettung von Zeichenketten

$(|0, Inc|)$ $Inc : (n, k) \mapsto k + 1$ – Länge einer Liste

Fold/Reduce

- Konsitentes Ersetzen der strukturellen Komponenten einer Datenstruktur
- Ersetzen mit Funktionen und Werten (Funktionen sind auch nur Daten!)
- Beispiel Liste:
  - Entweder leere Liste (`[]` oder *Nil*); *oder*
  - Element an eine Liste vorne anstellen
    Cons Node: $Cons(x_1 1, Cons(x_2, Cons(\ldots(Cons(x_n, Nil)))))$



Fold/Reduce – Beispiele

- $\mathit{fold}(+)\,[1,2,3,4,5] = 15$
  - Erste Näherung: „,“ durch „+“ ersetzen?
  - Tatsächlich: $1 + (2 + (3 + (4 + 5)))$
- $\mathit{fold}(!)\,[1,2,3,4,5] = 120$
  - Analog zu oben? $1 + (2 + (3 + (4 + 5)))$
  - Fakultät zeigt Wichtigkeit eines Identitätselements: $0 \cdot (1 \cdot (2 \cdot (3 \cdot (4 \cdot 5))))$
  - Grund für $\mathit{Nil} \mapsto b$

Anamorphismus

- Katamorphismus: Liste „eindampfen“
- Anamorphismus: Liste aus Einzelteilen aufbauen
- $p\colon B \to \{w, f\}$: Ein Prädikat
- $g\colon B \to A \times B$: Eine Abbildung
- Anamorphismus:

$$h : b \mapsto \mathit{Nil} \text{ falls } p(b) = w \tag{3}$$

$$b \mapsto \mathit{Cons}(a, h(b')) \text{ mit } [a, b'] = g(b) \text{ falls } p(b) = f \tag{4}$$

- Notation mit Generator g und Prädikat p: $h = [(p, g)]$

Funktionale Programmierung in Python

- Funktionen als Daten:

```
def add(a, b):
    return a + b

plus = add
plus(3, 7)  # => 10
```

- Funktionen geben Funktionen zurück:

```
def make_adder():
   return add

plus = make_adder()
plus(3, 7)  # => 10
```

Anonyme Funktionen (Lambdas)

- Lambdas: Funktionsdefinition ohne Deklaration

```
plus = lambda a, b: a + b
plus(3, 3)    # => 6

(lambda x, y: x-y)(10, 6)  # => 4
```

- Funktionen = Parameter = Daten, z.B.

```
authors = ['Octavia Butler', 'Isaac Asimov', 'Neal
 ↪  Stephenson', 'Margaret Atwood', 'Usula K Le Guin',
 ↪  'Ray Bradbury']
sorted(authors, key=len)  # Sortiert nach Länge des
 ↪  Namens
sorted(authors, key=lambda name: name.split()[-1])
```

Nicht-funktional...

```
d = ['fox', 'boss', 'orange', 'toes', 'fairy', 'cup']
def puralize(words):
    for i in range(len(words)):
        word = words[i]
        if word.endswith('s') or word.endswith('x'):
            word += 'es'
        if word.endswith('y'):
            word = word[:-1] + 'ies'
        else:
            word += 's'
        words[i] = word

def test_pluralize():
    pluralize(d)
    assert dictionary == ['foxes', 'bosses', 'oranges',
    ↪  'toeses', 'fairies', 'cups']
```

. . . und funktional

```
def puralize(words):
    result = []
    for word in words:
        word = words[i]
        if word.endswith('s') or word.endswith('x'):
            plural = word + 'es')
        if word.endswith('y'):
            plural = word[:-1] + 'ies'
        else:
            plural = +  's'
        result.append(plural)
     return result
```

Höherwertige Funktionen

- Partiell definierte Funktionen:

```
def power(base, exp):
    return base ** exp
cube = partial(power, exp=3)
cube(5)
```

- Funktionen geben Funktionen zurück:

```
from functools import map, reduce

val = [1, 2, 3, 4, 5, 6]

list(map(lambda x: x * 2, val)) # [2, 4, 6, 8, 10, 12]
reduce(lambda: x, y: x * y, val, 1) # 1 * 1 * 2 * 3 *
↪  4 * 5 * 6
```

Decorator

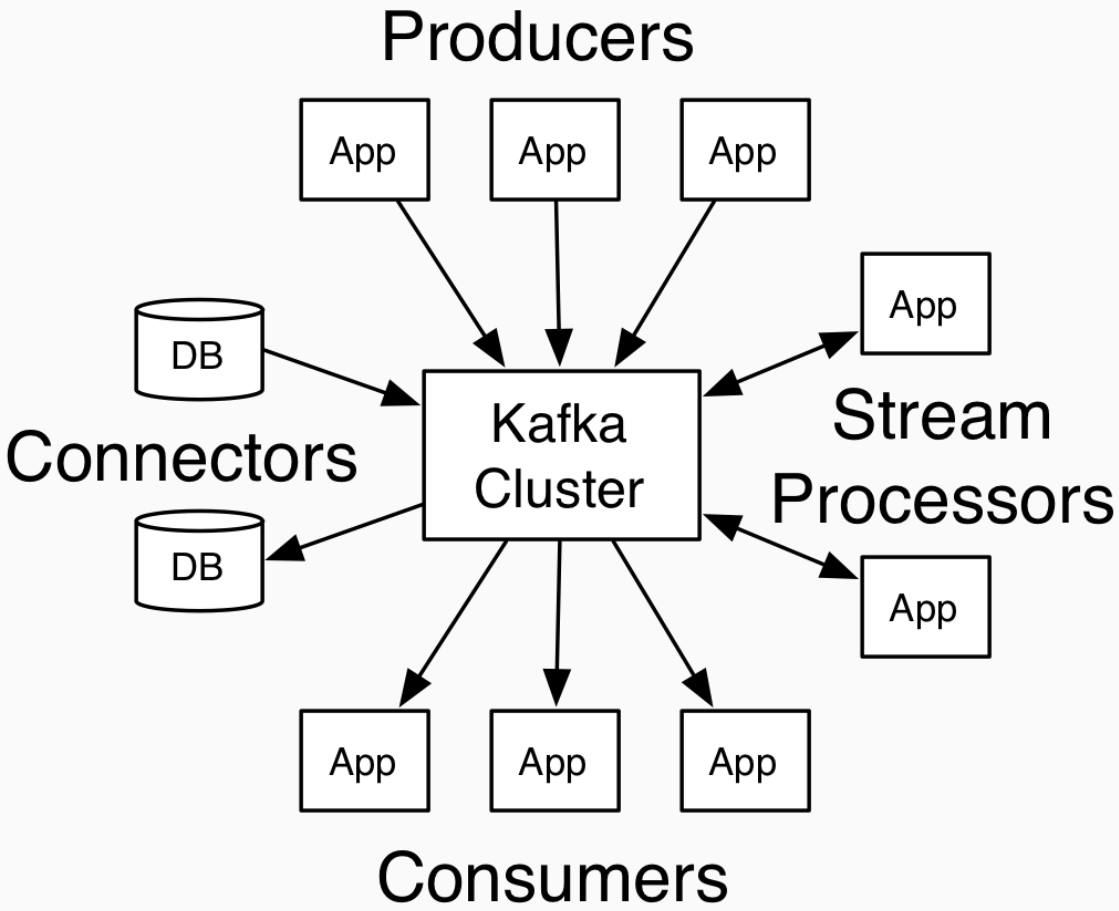
```
def retry(func):
    def retried_function(*args, **kwargs):
        exc = None
        for _ in range(3):
            try:
                return func(*args, **kwargs)
            except Exception as exc:
                print("Exception raised while calling %s
                ↪  with args:%s, kwargs: %s. Retrying" %
                ↪  (func, args, kwargs).
        raise exc
     return retried_function

@retry
def do_something_risky():
    # ...
```

Apache Kafka

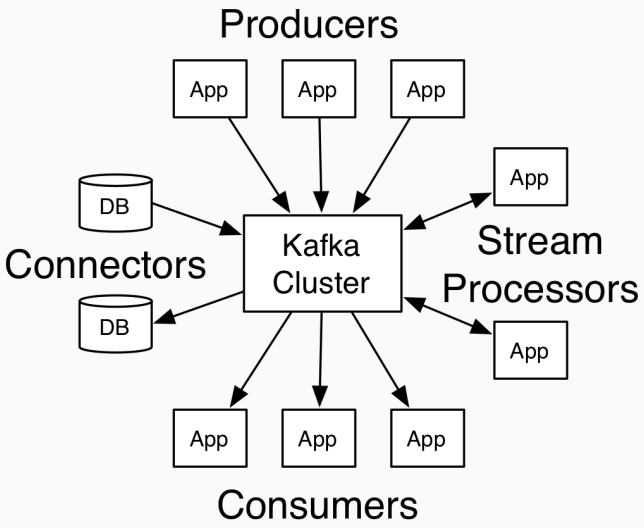
Motivation

1. Empfänger offline (z.B. Wartung); Sender hat keine Kapazität zum Puffern
2. Sender erzeugt Nachrichten schneller, als der Empfänger sie verarbeiten kann (DoS!)
3. Empfänger stürzt während der Verarbeitung einer Nachricht ab – was weiß der Sender über die Nachricht?

Apache Kafka



Apache Kafka

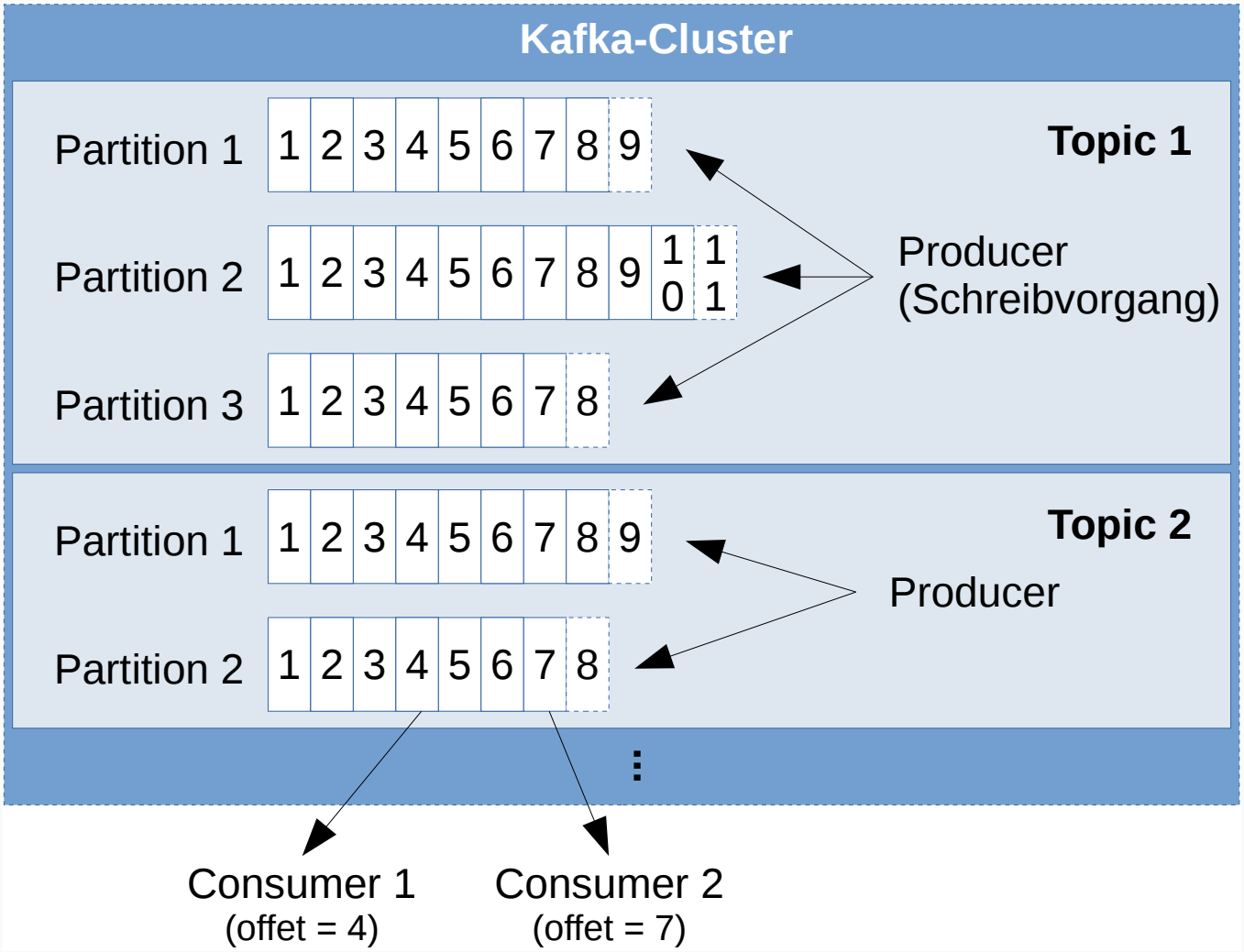


- **Distributed Streaming Platform**: Effizienter Nachrichtenverteiler
- Nachrichtenverteilung über **Producer-Consumer**-Muster
- Einfaches Nachrichtenformat: $m = (key, value)$
- *Topics* zur Organisation: $\{m_1, m_2, \ldots, m_n\}$
- Elegante Enkopplung verschiedener Komponenten im *Big Data Stack*
- Unterstützt auch Microservices-Architekturen

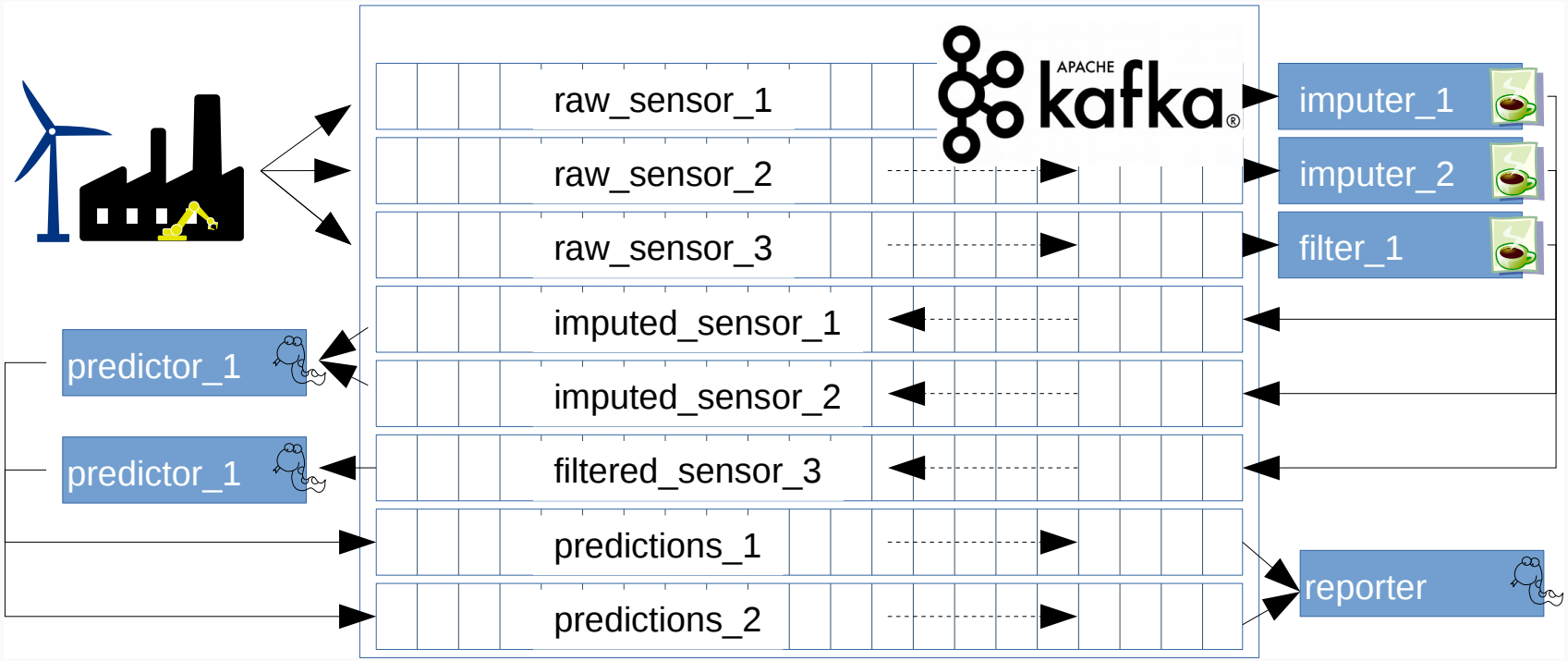
Apache Kafka: Architektur

- **Topics**: geordnet Strom von **Records**
- Records werden von einem **Producer** geliefert
- Ein Topic hat beliebig viele **Consumer**
- Ein Topic besteht aus beliebig vielen **Partitionen**
- **Partition**: Geordnete, unveränderliche Sequenz von Records (**append only**)
- Eine Partition muss auf einem Server Platz finden; ansonsten für Clustering genutzt
- Records verbleiben bis zum Ablauf der **Retention Period** in einer Partition
- Einzelne Consumer haben ein individuelles **Offset**
- Quintessenz: Kafka ist eine Kombination aus **Nachrichtendienst**, **Nachrichtenspeicher** und **Datenstrom**

Apache Kafka: Architektur



Einsatz von Kafka



- Kafka erlaubt Segmentierung, damit Aufteilung auf verschiedene Geräte und ggfs. verschiedene Programmiersprachen
- Begünstigt **Microservices-Architekturen**

Producer für Apache Kafka

```
import asyncio; from aiokafka import AIOKafkaProducer
event_loop = asyncio.get_event_loop()

async def say_hello():
    producer = AIOKafkaProducer(loop=event_loop,
                                bootstrap_servers='br1,br2',
                                acks='all')   # 0, 1, all
    await producer.start()  # Wait for Future
    try:
        await producer.send_and_wait("my_topic",
                                     b"Hello, World!")
    finally:
        await producer.stop()

event_loop.run_until_complete(say_hello())
```

Consumer für Kafka

```
consumer = aiokafka.AIOKafkaConsumer("my_topic",
    loop=event_loop,
    bootstrap_servers='localhost:9092',
    group_id="my_group",
    enable_auto_commit=True,
    auto_commit_interval_ms=1000)
await consumer.start()
try:
    async for msg in consumer:
        print("{}:{:d}:{:d}: key={} value={} timestamp_ms={}" \
              .format(msg.topic, msg.partition, msg.offset,
                      msg.key, msg.value, msg.timestamp))
finally:
    await consumer.stop()
```