In-Memory Processing

Am Beispiel von Apache Spark

Lukas Pietzschmann
lukas.pietzschmann@uni-ulm.de

Institut für Verteilte Systeme
Universität Ulm

12. Januar 2023

Agenda

1. Intro
2. Lernziele
3. In-Memory Processing
4. Apache Spark
   - 4.1 Übersicht
   - 4.2 Datenmodell
   - 4.3 Architektur
   - 4.4 Spark Schritt für Schritt
   - 4.5 Spark im echten Leben
5. Take-Away
6. Referenzen

1. Intro

Motivation

Festplatten-Geschwindigkeit:

https://i.redd.it/1qh4lphop3501.jpg

RAM-Geschwindigkeit:

https://i.kym-cdn.com/entries/icons/original/000/028/987/lightningspeed.jpg

Motivation

Festplatten-Geschwindigkeit:

https://i.redd.it/1qh4lphop3501.jpg

RAM-Geschwindigkeit:

https://i.kym-cdn.com/entries/icons/original/000/028/987/lightningspeed.jpg

$\Rightarrow$ Auswertung in Echtzeit, unabhängig der Datenmenge

Motivation

	Split sizes (MB)	Execution time (s)
MapReduce input splits	128	2376
Spark input splits	256	1392
MapReduce shuffle	100	2371
Spark shuffle	300	1334

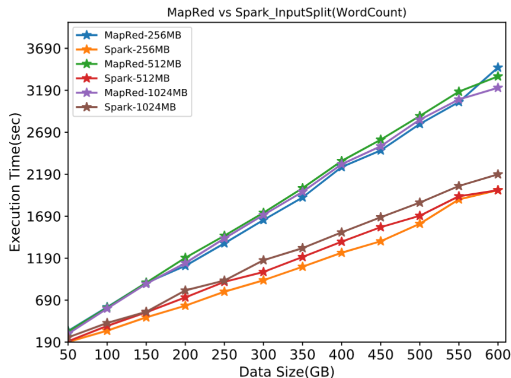
Best execution time of MapReduce and Spark with WordCount workload [Ahm+20, S. 12]

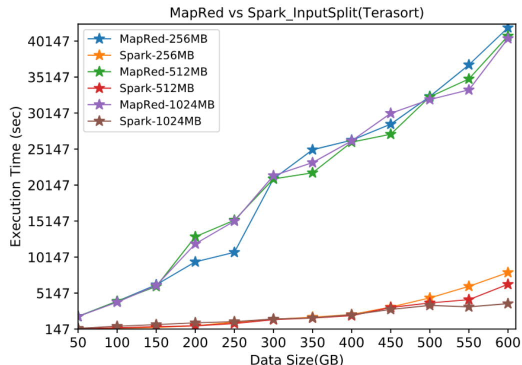
Motivation

	Split sizes (MB)	Execution time (s)
MapReduce input splits	256	21014
Spark input splits	512	3780
MapReduce shuffle	150	24250
Spark shuffle	128	6540

Best execution time of MapReduce and Spark with Terasort workload [Ahm+20, S. 14]

Motivation





Comparison of Hadoop and Spark with WordCount and TeraSort workload with varied input splits and shuffle tasks [Ahm+20, S. 14]

2. Lernziele

Lernziele

Theorie

- Unterschied zwischen reinem Map-Reduce und Spark erkennen
- Sparks Datenmodell und dessen Implikationen verstehen
- Wichtige Bindeglieder und deren Rolle vom Programmieren bis zur Ausführung kennen
- Zumindest von der Existenz weiterer Spark-Bibliotheken wissen

Lernziele

Praxis

- Verschiedene Aktionen und Transformationen
  - kennen,
  - anwenden und
  - kombinieren

  können (und das natürlich sinnvoll)

Lernziele
Praxis

- Verschiedene Aktionen und Transformationen
    - kennen,
    - anwenden und
    - kombinieren

  können (und das natürlich sinnvoll)
- Wörter-zählen in Spark implementieren können 😉

3. In-Memory Processing

Übersicht

Anwendungsfälle

- Echtzeit-Systeme
  - Zahlungsverarbeitung
  - Börsenhandel
  - Simulationen
  - BI
  - ...

Implementierungen

- SAP Hana
- Apache Flink

Übersicht

Anwendungsfälle

- Echtzeit-Systeme
  - Zahlungsverarbeitung
  - Börsenhandel
  - Simulationen
  - BI
  - ...

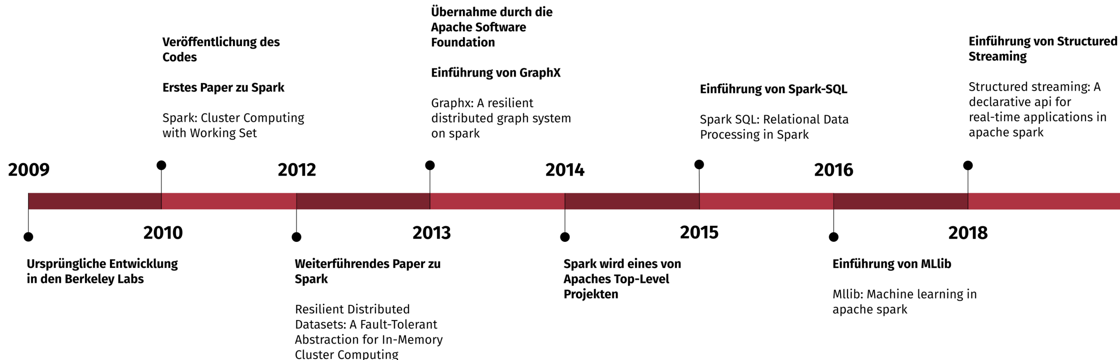
Implementierungen

- SAP Hana
- Apache Flink
- ✨Apache Spark✨

4. Apache Spark

4.1 Übersicht

Historie



Ziele

„[...] reuse a working set of data across multiple parallel operations. This includes many iterative machine learning algorithms, as well as interactive data analysis tools. We propose a new framework called Spark that supports these applications while retaining the scalability and fault tolerance of MapReduce. “ [Zah+10, S. 1]

Ziele

„[...] reuse a working set of data across multiple parallel operations. This includes many iterative machine learning algorithms, as well as interactive data analysis tools. We propose a new framework called Spark that supports these applications while retaining the scalability and fault tolerance of MapReduce. “ [Zah+10, S. 1]

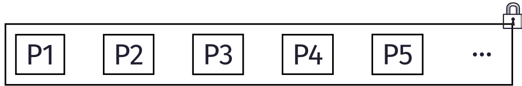
+ mehr Möglichkeiten als reines Map-Reduce

Übersicht

- Generalisiertes Map-Reduce
  $\Rightarrow$ Größere Bandbreite an Anwendungsfällen
- Interaktive Shell + Java-API
  - Python (`pyspark`)
  - Scala (`spark-shell`)
  - R (`sparkR`)
- Ausführung auf
  - Einzelnen Maschinen
  - Clustern aus mehreren Maschinen

4.2 Datenmodell

Arbeits-Daten



RDD (**R**esilient **D**istributed **D**ataset)

Datensatz $\rightarrow$ Partitionen

Arbeits-Daten

P1 P2 P3 P4 P5 ...

RDD (**R**esilient **D**istributed **Dataset**)

Datensatz → Partitionen

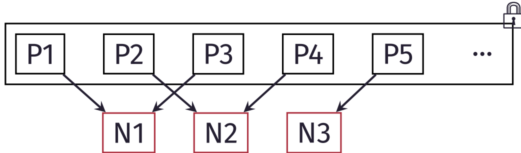
Arbeits-Daten

P1 P2 P3 P4 P5 ...

RDD (**R**esilient **Distributed** **D**ataset)

Datensatz → Partitionen

Arbeits-Daten

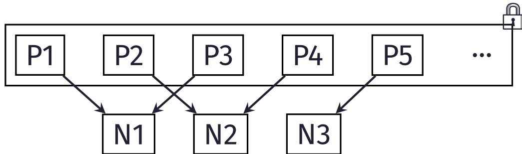


RDD (**R**esilient **D**istributed **D**ataset)

Datensatz → Partitionen

→ parallele Ausführung

Arbeits-Daten



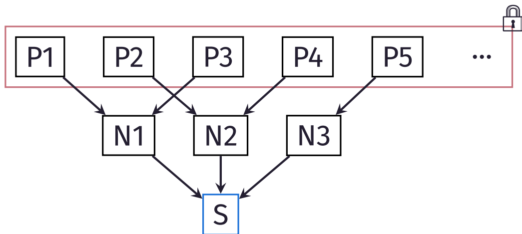
RDD (**R**esilient **D**istributed **D**ataset)

Datensatz → Partitionen

→ parallele Ausführung

Operationen

Arbeits-Daten



RDD (**R**esilient **D**istributed **D**ataset)

Datensatz → Partitionen

→ parallele Ausführung

Operationen

Aktion RDD → Wert

Arbeits-Daten

P1 P2 P3 P4 P5 ...

N1 N2 N3

P1' P2' P3' P4' P5' ...

RDD (**R**esilient **D**istributed **D**ataset)

Datensatz → Partitionen

→ parallele Ausführung

Operationen

Aktion RDD → Wert

Transformation RDD → RDD^n

ARBEITS-DATEN

P1 P2 P3 P4 P5 ...

P1' P2' P3' P4' P5' ...

RDD (**R**esilient **D**istributed **D**ataset)

Datensatz → Partitionen

→ parallele Ausführung

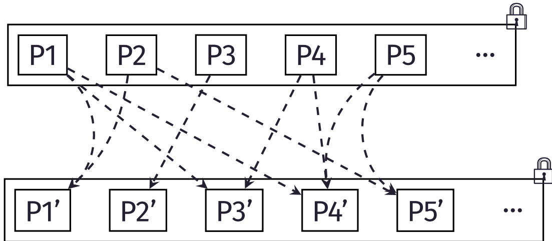
Operationen

Aktion RDD → Wert

Transformation RDD → RDD^n

- Enge Transformation

Arbeits-Daten



RDD (**R**esilient **D**istributed **D**ataset)

Datensatz → Partitionen

→ parallele Ausführung

Operationen

Aktion RDD → Wert

Transformation RDD → RDD^n

- Enge Transformation
- Weite Transformation

Arbeits-Daten

Aktionen und Transformationen

Transformationen:

- map, flatMap, filter
- union, intersection, distinct
- groupBy, join, fullOuterJoin
- keys, values
- sortBy
- cartesian
- zip
- ...

Aktionen:

- reduce, fold, aggregate
- first, take, collect
- foreach
- count, countByKey, countByValue
- mean, max, min
- saveAsTextFile
- ...

Arbeits-Daten

Beispiele

```python
data = [1, 1, 2, 3, 5, 8, 13, 21, 34]
rdd = sc.parallelize(data, 16)

s = rdd.reduce(lambda acc, x: acc + x)
print(s)                                    # 88

rdd_2 = rdd.map(lambda x: x ** 2)
rdd_2.foreach(print)                        # ?
```

Arbeits-Daten

Beispiele

```
data = [1, 1, 2, 3, 5, 8, 13, 21, 34]
rdd = sc.parallelize(data, 16)

s = rdd.reduce(lambda acc, x: acc + x)
print(s)                                # 88

rdd_2 = rdd.map(lambda x: x ** 2)
rdd_2.foreach(print)                    # ?
```

Arbeits-Daten

Beispiele

```
data = [1, 1, 2, 3, 5, 8, 13, 21, 34]
rdd = sc.parallelize(data, 16)

s = rdd.reduce(lambda acc, x: acc + x)
print(s)                                # 88

rdd_2 = rdd.map(lambda x: x ** 2)
rdd_2.foreach(print)                    # ?
```

Arbeits-Daten

Beispiele

```
>>> rdd_2.foreach(print)
9
1
4
441
1156
1
169
64
25
```

```
>>> rdd_2.foreach(print)
441
11

1156
25
9
64
169
4
```

Arbeits-Daten

Beispiele

```
data = range(1, 1000000)
rdd = sc.parallelize(data)

rdd2 = rdd.map(lambda x: x ** x)
rdd2.count()
```

Arbeits-Daten

Beispiele

```
data = range(1, 1000000)
rdd = sc.parallelize(data)

rdd2 = rdd.map(lambda x: x ** x)
rdd2.count()
```

Arbeits-Daten

Lazy-Evaluation

```
data = range(1, 1000000)
r = sc.parallelize(data)

r2 = r.map(lambda x: x**x)🐆
r2.count()🦥

// Ausgabe des Abstammungs-
// Graphen
r2.toDebugString()
```

Lazy-Evaluation

- Transformationen werden mit der ersten Aktion ausgeführt
- Transformationen
  → Abstammungs-Graph
  → Ausführungsplan
- Abstammungs-Graph = DAG (**D**irected **A**cyclic **G**raph)

Arbeits-Daten

Lazy-Evaluation

```
data = range(1, 1000000)
r = sc.parallelize(data)

r2 = r.map(lambda x: x**x)🐆
r2.count()🦥

// Ausgabe des Abstammungs-
// Graphen
r2.toDebugString()
```

Abstammungs-Graph

```
i = sc.textFile("some_file")
o = i.flatMap(lambda l: l.split(" "))
     .map(lambda w: (w, 1))
     .reduceByKey(add)
o.toDebugString()
```

```
(2) ShuffledRDD[6] at reduceByKey
 +-(2) MapPartitionsRDD[5] at map
 |  MapPartitionsRDD[4] at map
 |  log.txt MapPartitionsRDD[1] at textFile
 |  log.txt HadoopRDD[0] at textFile
```

Arbeits-Daten

Persistenz

- Persistenz (`persist()`) $\approx$ Caching (`cache()`)
- Knoten speichern alle von ihnen berechneten Partitionen
- Das Wiederverwenden von Zwischenergebnissen kann zukünftige Aktionen deutlich beschleunigen
- Verschiedene Level an Persistent:
    - MEMORY_ONLY
    - MEMORY_AND_DISK
    - DISK_ONLY

Arbeits-Daten
Persistenz

- Persistenz (`persist()`) ≈ Caching (`cache()`)
- Knoten speichern alle von ihnen berechneten Partitionen
- Das Wiederverwenden von Zwischenergebnissen kann zukünftige Aktionen deutlich beschleunigen
- Verschiedene Level an Persistent:
  - MEMORY_ONLY
  - MEMORY_AND_DISK
  - DISK_ONLY

Arbeits-Daten
Persistenz

- Persistenz (`persist()`) $\approx$ Caching (`cache()`)
- Knoten speichern alle von ihnen berechneten Partitionen
- Das Wiederverwenden von Zwischenergebnissen kann zukünftige Aktionen deutlich beschleunigen
- Verschiedene Level an Persistent:
  - MEMORY_ONLY
  - MEMORY_AND_DISK
  - DISK_ONLY

Arbeits-Daten
Persistenz

- Persistenz (`persist()`) $\approx$ Caching (`cache()`)
- Knoten speichern alle von ihnen berechneten Partitionen
- Das Wiederverwenden von Zwischenergebnissen kann zukünftige Aktionen deutlich beschleunigen
- Verschiedene Level an Persistent:
  - MEMORY_ONLY_2
  - MEMORY_AND_DISK_2
  - DISK_ONLY_2

Arbeits-Daten

Fehlertoleranz

Zur Erinnerung:

- Ein RDD ist ein schreibgeschützter, deterministischer Datensatz
- Ein RDD kennt stets seine Abstammung

Arbeits-Daten

Fehlertoleranz

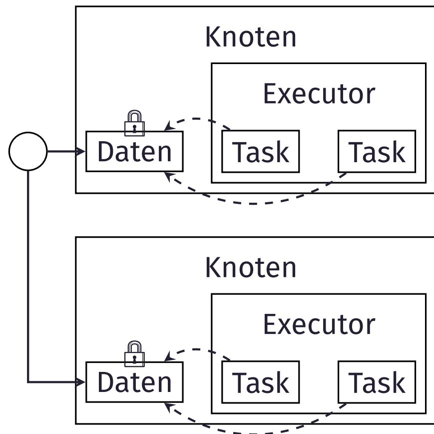
Zur Erinnerung:

- Ein RDD ist ein schreibgeschützter, deterministischer Datensatz
- Ein RDD kennt stets seine Abstammung

⇒ Jedes RDD kann „*einfach*“ neu berechnet werden

Geteilte Daten

Broadcast Variablen



Funktionsweise

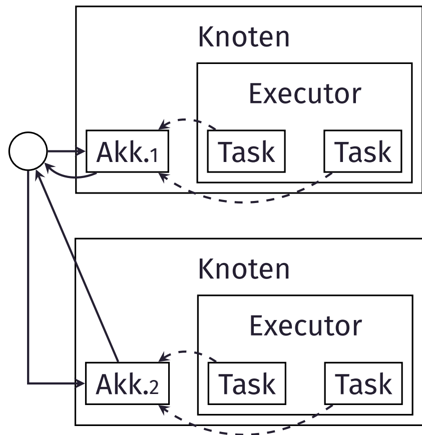
- Daten werden nicht für jeden Task kopiert
- Eine schreibgeschützte Instanz pro Knoten

Anwendungsfälle

- Große „*Nachschlage-Daten*“

Geteilte Daten

Akkumulatoren



Funktionsweise

- Knoten kann Akkumulator verändern ohne für die spätere Zusammenführung sorgen zu müssen
- exactly-once Semantik (zumindest teilweise)

Anwendungsfälle

- Summen oder Zähler
- Operation muss assoziativ und kommutativ sein

Geteilte Daten
Beispiele - Broadcast-Variablen

```
data = [1, 1, 2, 3, 5, 8, 13, 21, 34]
broadcast_data = sc.broadcast(broadcast_data)

print(broadcast_data.value) # [1, 1, 2, 3, 5, ...]
broadcast_data.destroy()
```

Geteilte Daten

Beispiele - Broadcast-Variablen

```python
data = [1, 1, 2, 3, 5, 8, 13, 21, 34]
broadcast_data = sc.broadcast(broadcast_data)

print(broadcast_data.value) # [1, 1, 2, 3, 5, ...]
broadcast_data.destroy()
```

Geteilte Daten

Beispiele - Broadcast-Variablen

```
data = [1, 1, 2, 3, 5, 8, 13, 21, 34]
broadcast_data = sc.broadcast(broadcast_data)

print(broadcast_data.value) # [1, 1, 2, 3, 5, ...]
broadcast_data.destroy()
```

Geteilte Daten

Beispiele - Akkumulatoren

```
acc = sc.accumulator(0)

sc.parallelize(data).foreach(lambda x: acc.add(x))
print(acc.value)             # 88
```

Geteilte Daten

Beispiele - Akkumulatoren

```
acc = sc.accumulator(0)

sc.parallelize(data).foreach(lambda x: acc.add(x))
print(acc.value)           # 88
```

Geteilte Daten

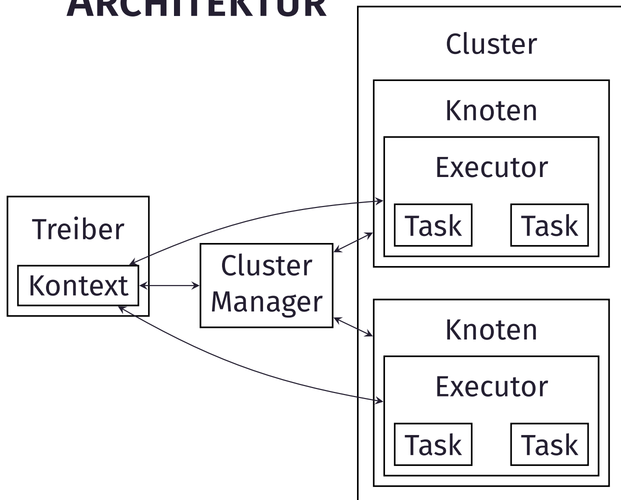
Beispiele - Akkumulatoren

```
acc = sc.accumulator(0)

sc.parallelize(data).foreach(lambda x: acc.add(x))
print(acc.value)          # 88
```

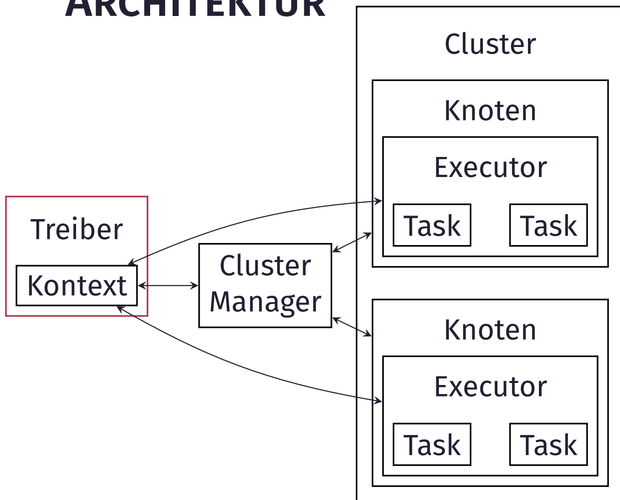
4.3 ARCHITEKTUR

Architektur



https://spark.apache.org/docs/latest/cluster-overview.html#components

Architektur

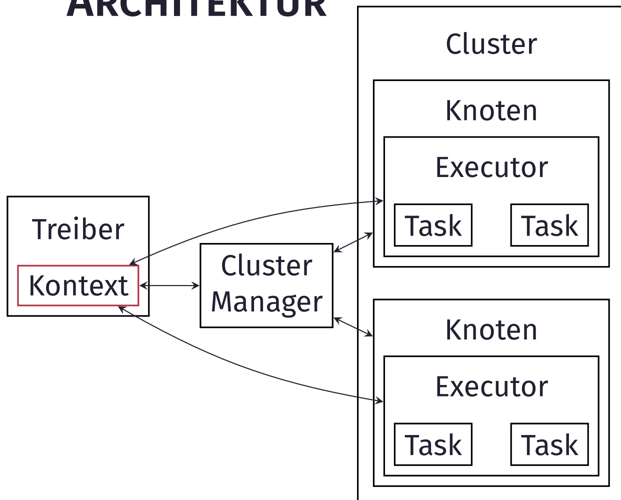


Treiber

- Instruiert Kontext
- Führt keine Berechnungen aus

https://spark.apache.org/docs/latest/cluster-overview.html#components

Architektur



Treiber

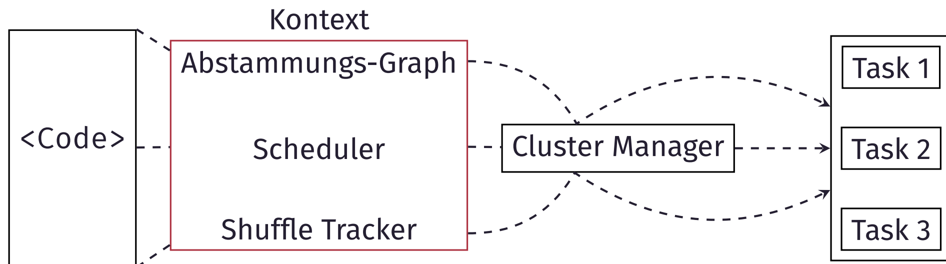
- Instruiert Kontext
- Führt keine Berechnungen aus

Kontext

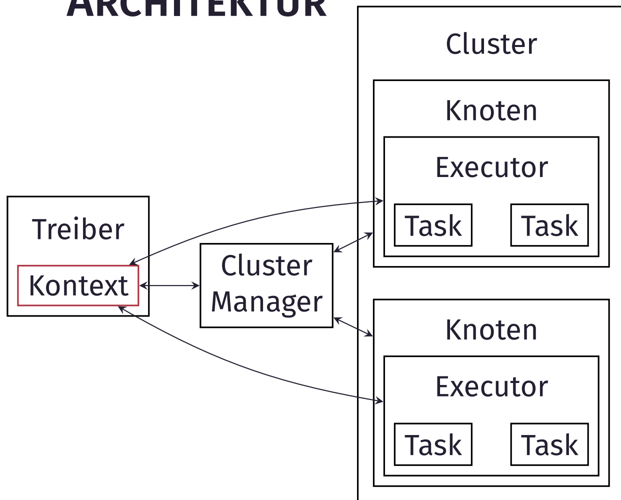
- Weist jedem Executor Tasks zu
- Schnittstelle zu Sparks API

https://spark.apache.org/docs/latest/cluster-overview.html#components

Architektur
Einschub: Kontext



Architektur



Treiber

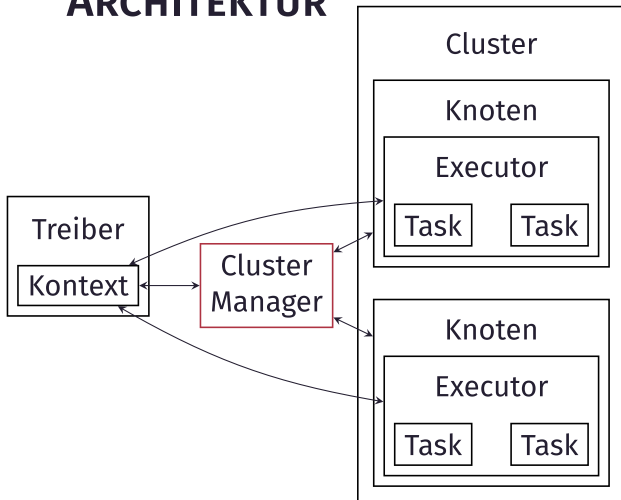
- Instruiert Kontext
- Führt keine Berechnungen aus

Kontext

- Weist jedem Executor Tasks zu
- Schnittstelle zu Sparks API

https://spark.apache.org/docs/latest/cluster-overview.html#components

Architektur



Treiber

- Instruiert Kontext
- Führt keine Berechnungen aus

Kontext

- Weist jedem Executor Tasks zu
- Schnittstelle zu Sparks API

Cluster Manager

- (Externer) Service zur Allokation von Ressourcen auf einem Cluster
- Unterstützte Typen: Standalone, Mesos, YARN, Kubernetes

https://spark.apache.org/docs/latest/cluster-overview.html#components

4.4 Spark Schritt für Schritt

Spark Schritt für Schritt

Jobs, Tasks, Stages, Partitionen, Shuffle, ... 🤯

Cluster Viele Knoten die Spark ausführen.

Treiber Ein Knoten im Cluster der alle anderen Knoten steuert.

Executor Alle Knoten im Cluster außer der Treiber.

Partition Eine Einheit aus einem RDD.

Task Eine Transformation, die auf eine einzelne Partition angewendet wird.

Shuffle Ein RDD wird neu partitioniert und über alle Konten verteilt.

Stage Eine Folge an Tasks die parallel ohne Shuffle ausgeführt werden können.

Job Eine Folge an Stages die durch eine Aktion angestoßen wird.

Logischer Ausführungsplan $\approx$ Abstammungs-Graph.

Physischer Ausführungsplan Tatsächliche Abfolge an Tasks die einem Executor zugewiesen sind.

https://knowyourmeme.com/memes/ight-imma-head-out

4.5 Spark im echten Leben

Weitere Abstraktionen
SQL

- Mehr Informationen → Bessere Optimierungsmöglichkeiten
- Zwei neue „Datentypen“:

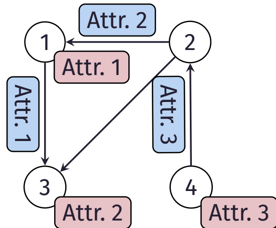
  Dataframe Konzeptuell equivalent zu einer Tabelle

  Dataset Erweitert die Dataframe API um ihre Vorteile mit den Vorteilen der RDDs zu verknüpfen. Starke Typisierung + starke Optimierungen = Dataset

Weitere Abstraktionen

GraphX

- Erweiterung der RDD API um eine Graph-Abstraktion
  - Gerichteter Graph mit Attributen auf Knoten und Kanten
- Zusätzliche Operationen wie
  - `subgraph`, `joinVertices`, `collectNeighbors`, `groupEdges`, ...



V	Attribut
1	Attr. 1
2	$\emptyset$
3	Attr. 2
4	Attr. 3

Knoten-Tabelle

E	Attribut
(1, 3)	Attr. 1
(2, 1)	Attr. 2
(2, 3)	$\emptyset$
(4, 2)	Attr. 3

Kanten-Tabelle

Weitere Abstraktionen

MLlib

ML Applikationen mit Spark auch ohne diese Bibliothek möglich!

Weitere Abstraktionen

MLlib

ML Applikationen mit Spark auch ohne diese Bibliothek möglich!

Ziel: Speziell auf ML abgestimmte Tools zur Verfügung stellen:

Algorithmen Classification, Regression, Clustering, ...

Tools Linear Algebra, Statistik, ...

Pipelines Tools zum Erstellen, Verwalten und Auswerten von Pipelines an Algorithmen

Weitere Abstraktionen

Streaming

- Durchsatzstarke und fehlertolerante Verarbeitung von Echtzeit-Datensätzen
- Verwendung derselben grundlegenden Ideen und Konzepte
- Operationen können auf ein gleitendes Datenfenster angewandt werden

https://spark.apache.org/docs/3.3.1/streaming-programming-guide.html#overview

Weitere Abstraktionen

Streaming

- Durchsatzstarke und fehlertolerante Verarbeitung von Echtzeit-Datensätzen
- Verwendung derselben grundlegenden Ideen und Konzepte
- Operationen können auf ein gleitendes Datenfenster angewandt werden

https://spark.apache.org/docs/3.3.1/streaming-programming-guide.html#overview

5. Take-Away

Take-Away

- Spark basiert auf der Idee von schreibgeschützten, verteilten Datensätzen
- RDDs können mit diversen Transformationen und Aktionen verarbeitet werden
- Aufwändige Transformationen können durch Lazy-Evaluation optimiert werden
- Fehlertoleranz wird durch simples Neuberechnen erzeugt
- Zusätzliche APIs für weitere Anwendungsfälle

6. Referenzen

Literatur

Moodle

[Zah+10] Matei Zaharia u. a. „Spark: Cluster computing with working sets“. In: *2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 10)*. 2010.

[Zah+12] Matei Zaharia u. a. „Resilient distributed datasets: A Fault-Tolerant abstraction for In-Memory cluster computing“. In: *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. 2012.

[CZ18] Bill Chambers und Matei Zaharia. *Spark: The definitive guide: Big data processing made simple*. O’Reilly Media, 2018.

[Dam+20] Jules S Damji u. a. *Learning Spark*. O’Reilly Media, 2020.

Literatur

Weiterführend

[Awa+16] Ahsan Javed Awan u. a. „Architectural impact on performance of in-memory data analytics: Apache spark case study“. In: *arXiv preprint arXiv:1604.08484* (Apr. 2016). DOI: https://doi.org/10.48550/arXiv.1604.08484.

[Ahm+20] Nasim Ahmed u. a. „A comprehensive performance analysis of Apache Hadoop and Apache Spark for large scale data sets using HiBench“. In: *Journal of Big Data* 7.1 (2020), S. 1–18. DOI: https://doi.org/10.1186/s40537-020-00388-5.

[Fou] The Apache Foundation. *Spark Documentation*. URL: https://spark.apache.org/docs/latest/index.html (besucht am 19. 11. 2022).

Lukas Pietzschmann

Ulm, 12. Januar 2023

lukas.pietzschmann@uni-ulm.de