Spezielle Kapitel Sozialer Web – **Technologien** – Spark

Seminaristischer Unterricht

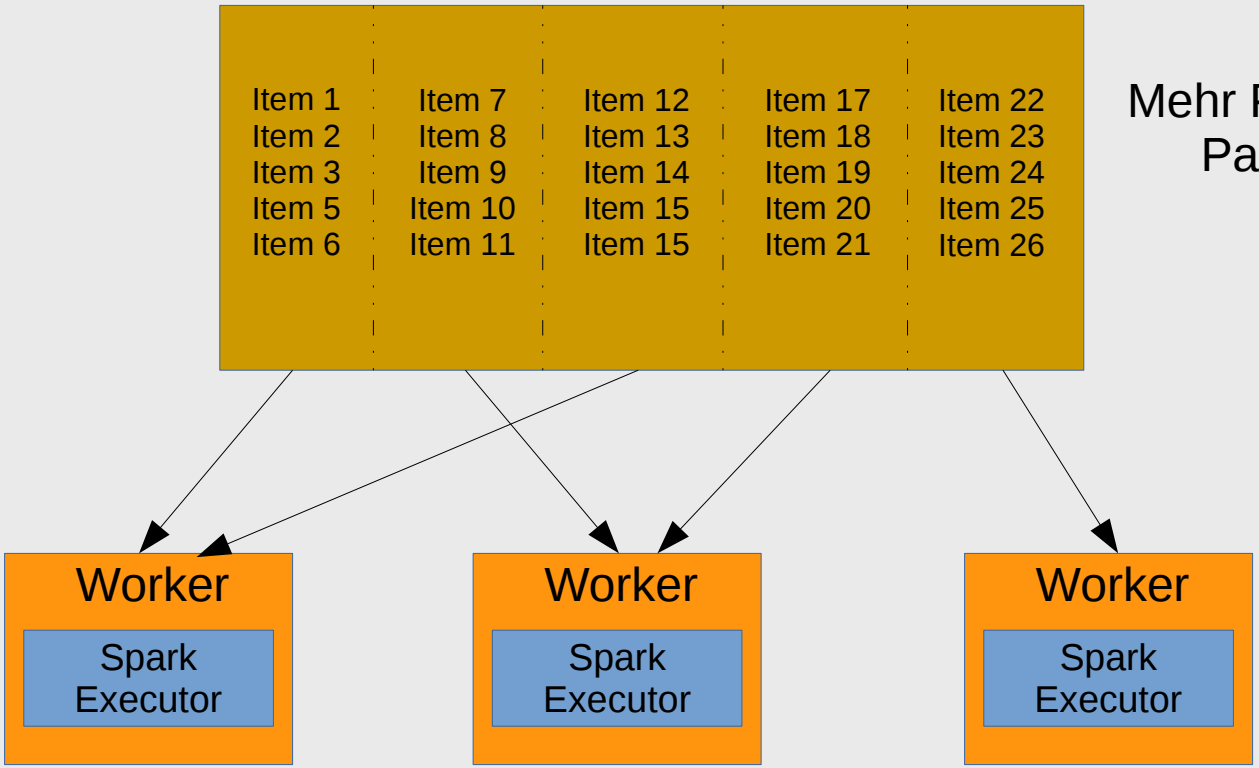
Prof. Dr.-Ing. Hendrik Gärtner

Gliederung

- Wiederholung Spark
  - Ansatz Spark/Abstraktionsschicht RDDs
  - Spark Transformations und Actions
  - Spark Architektur
- Fortgeschrittene Spark-Techniken
  - Closures und Serialisierung
  - Broadcast-Variablen
  - Akkumulatoren
  - Caching
  - Funktionen für Key/Value-Paar

Resilient Distributed Dataset

Programmierer spezifiziert die Nummer der Partitionen des RDDs



Mehr Partitionen mehr Parallelisierung

Operationen auf RDDs

- Spark bietet zwei Typen von Operationen: Transformationen und Actions
- Transformation sind „lazy“ (werden erst berechnet, wenn erforderlich)
- Transformationen auf dem RDD werden ausgeführt, wenn eine Action aufgerufen wird
- Persist (Cache) RDDs werden im Memory oder auf der Disk gecacht

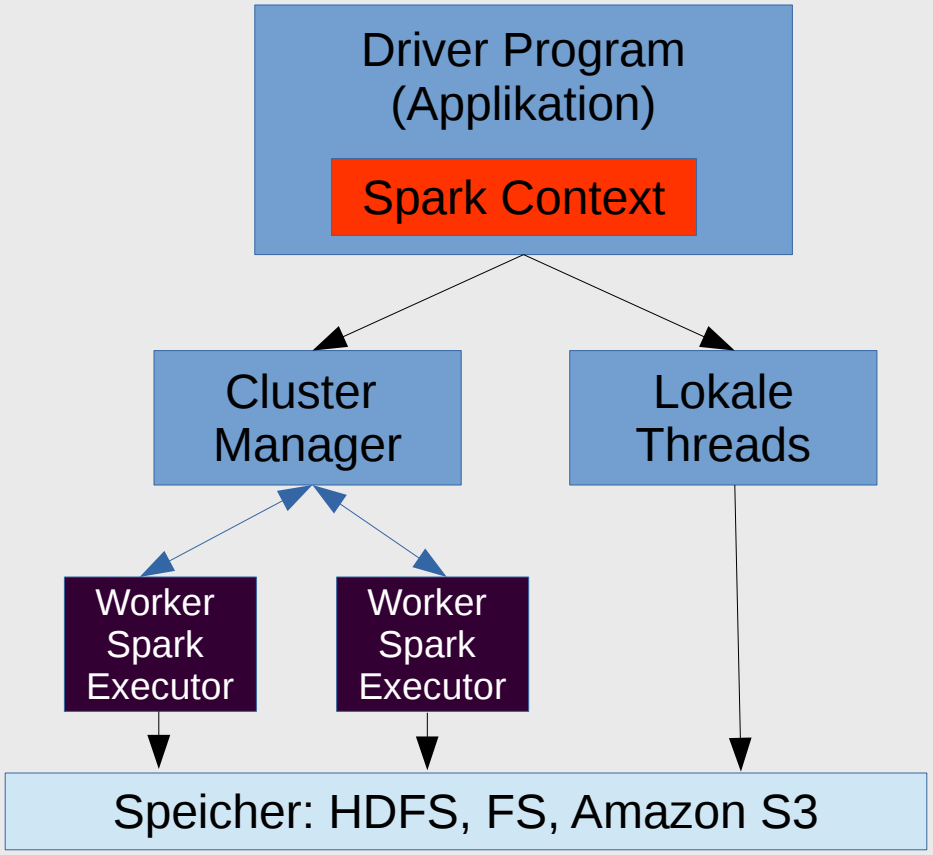
Arbeiten mit RDDs

1) Erzeugen eines RDDs über eine Datenquelle
2) Verändern des RDDs mittels der Transformationen
3) Sammeln des Ergebnisses durch das Durchführen einer Action

Spark Architektur

Eine Spark-Applikation besteht aus zwei Programmen:

- Ein sogenannter Treiber (Driver Program) sowie die
- Worker, die die eigentliche Arbeit verrichten



Ein Worker läuft entweder:

- innerhalb eines Clusters oder
- in Localen Threads

Die RDDs sind verteilt

Wie kommen Code und Daten vom Treiber zu den Workern?

Beispiel Skalierung der Sentimentanalyse

```
// Definition eines Dictionaries mit den Sentimentwerten der Wörter
val sentimentVals: Map[String, Double]= …
// Definition einer Funktion zur Berechnung der Sentiment-Werte
def calculateSentimentValues
        (line:String, sentimentDict:Map[String, Double]):Double

// Anwendung auf das Distributed Resillient Dataset
dataRDD.map(x=> calculateSentimentValues(x,sentimentVals))
```

Spark muss sowohl den Funktionscode von calculateSentimentValues als auch die Variable sentimentVals zu den Workern transferieren

Vorgehensweise von Spark

Wenn Spark eine Transformation ausführt wird automatisch ein sogenanntes **Closure** erzeugt und dieses

- auf dem Treiber Knoten **serialisiert**,
- zum entsprechden Knoten im Cluster transferiert,
- **deserialisiert** und
- und am Ende auf dem Knoten ausgeführt.

Serialisierung

Die Serialisierung ist eine Abbildung von strukturierten Daten auf eine sequenzielle Darstellungsform. Serialisierung wird hauptsächlich für die Persistierung von Objekten in Dateien und für die Übertragung von Objekten über das Netzwerk bei verteilten Softwaresystemen verwendet.

- Klassen in Java/Scala, die Serialisierbar sind, implementieren das Interface Serializable
- Serializable ist ein sogenanntes Marker Interface, das keine Methoden enthält und nur anzeigt, dass die Klasse serialisiert werden kann
- Serialisiert wird über sogenannte OutputStreams, die die Methode writeObjekt implementieren
- Es gibt Klassen, die nicht serialisiert werden können z.B. Thread, Socket und auch **SparkContext**
- Elemente können mit der Annotation @transient ausgenommen werden

Closures

Closures sind Funktionen, deren Ergebnis von einem oder mehr Variablen abhängen, die außerhalb der Funktion deklariert werden. Beispiel:

```
def filterBelowFirst(xs:List[Int]):List[Int]  = {
    val firstEl = xs.head
    val isBelow =  (y:Int)=>(y < firstEl)
    xs filter isBelow
}
```

- Das Ergebnis der Funktion isBelow hängt von der Variablen firstEl ab
- Die Variable firstEl wird im Kontext der Funktion filterBelowFirst definiert
- Scala definiert automatisch einen Closure um die Funktion isBelow, die den erforderlichen Kontext (die Funktion filterBelowFist) enthält

Erweiterung des Kontextes

```
class xyz{...
   var offset=3
   def filterBelowFirst(xs:List[Int]):List[Int]  = {
      val firstEl = xs.head
      val isBelow =  (y:Int)=>(y < firstEl)
      xs filter isBelow
   } ...
}
```

- Das Ergebnis der Funktion isBelow hängt von der Variablen firstEl und offset ab
- Die Variable offset wird im Kontext der Klasse xyz definiert
- Scala definiert automatisch einen Closure um die Funktion isBelow, die den erforderlichen Kontext (die gesamte Klasse xyz) enthält

Serialisierbarkeit von Funktionen

```
class SearchFunctions(val query:String){
    def isMatch(s:String):Boolean ={s.contains(query)}

     def getMatchesFunctionReference(rdd:RDD[String]):RDD[Boolean]={
        // problem: isMatch means this.isMatch), so we pass all of this
        rdd.map(isMatch)}

     def getMatchesFieldReference(rdd:RDD[String]):RDD[Array[String]]={
        // problem: isMatch means this.isMatch), so we pass all of this
        rdd.map(x=>x.split(query))}

   def getMatchesNoReference(rdd:RDD[String]):RDD[Array[String]]={
       // Safe: Extracts just the field we need into a local variable
       val query_ = this.query
       rdd.map(_.split(query_))}
}
```

Beispiel aus: Learning Spark – Lightning Fast Data Analysis, O'Reilley, 2015, page 32

Problemstellungen der Spark-Closures

Beispiel Sentimentanalyse:

dataRDD.map(x=> calculateSentimentValues(x,sentimentVals))

- Wie wird damit umgegangen, wenn große, statische Datenmengen (z.B. das sentimentDict) an die Worker gesendet werden müssen? Was ist, wenn diese immer wieder benötigt werden?

- Was ist, wenn bestimmte Ereignisse an den Driver zurückgemeldet werden sollen ? (z.B. wenn eine Textpassage unter einen bestimmten Sentiment-Wert rutscht)

- **Broadcast-Variablen**
- **Akkumulatoren**

Broadcast-Variablen

- Senden sehr effizient „Read-Only“-Daten zu allen Workern
- Werden auf allen Workern gespeichert, um sie für mehrere Operationen verwenden zu können
- Sind z.B. für das Versenden großer „Lookup“-Tabellen geeignet

Anwendung von Broadcast-Variablen

Beispiel Sentimentanalyse:

```
val sentimentVals:Map[String,Double]= loadDictionary(...)

// Definition einer Broadcast-Variablen:

val sentimentValsBroadcast= sc.broadcast(sentimentVals)

// Übergabe der Broadcast-Variable

dataRDD.map(x=> calculateSentimentValues(x,sentimentValsBroadcast))


def calculateSentimentValues(line:String,
                             sentimentDict:Broadcast[Map[String,Double]]):Double={

...

val dictionary= sentimentDict.value

}
```

Richtung von Closures

Beispiel in Scala:

```
val l=List(1,2,3,4,5,6,7,8,9,10)

var counter=0

l.foreach(x=>counter=counter+x)
```

→ Counter ist 55

Beispiel mit Spark

```
val rdd= sc.parallelize(l,8)

var counter=0

rdd.foreach(x=>counter=counter+x)
```

→ Counter ist 0

Mit dem Closure geht die Verbindung zum Driver verloren

Anwendung von Akkumulatoren

```
val l=List(1,2,3,4,5,6,7,8,9,10)
val rdd= sc.parallelize(l,8)
val counteraccu= sc.accumulator(0)
def count(element:Int, counter:Accumulator[Int]):Unit={
   counter += element
}
rdd.foreach(count(_,counteraccu))
```

→ counteraccu ist 55!

- += ist ein spezieller Operator, der definiert sein muss
- Definition eigener Akkumulatorentypen möglich

Eigenschaften Akkumulatoren

- Variablen s that can only be “added” to by associative op
- Used to efficiently implement parallel counters and sum
- Only driver can read an accumulator’s value, not tasks
  -Tasks see accumulators as write-only variables
- Accumulators can be used in actions or transformations:
  - Actions: each task’s update to accumulator is applied only once
  - Transformations: no guarantees (use only for debugging)
- Types: integers, double, long, float
- See lab for example of custom type

Definition eigener Akkumulatorentypen

```
object ListAccumulatorParam extends AccumulatorParam[List[String]]{
    def zero(initialValue: List[String]): List[String] = { List():List[String] }
    def addInPlace(v1: List[String], v2: List[String]): List[String] = {v1 ++ v2}
 }

val l= List("hallo","die","ist","ein","Test")
val resultList = sc.accumulator(l)(ListAccumulatorParam) oder
val resultList= sc.accumulator(ListAccumulatorParam.zero(l))(AccumulatorParam)
val rdd= sc.parallelize(l)
rdd.foreach(x=>resultList+=List(x))
```

Performance-Optimierung

```
val set=Range(1,1000000)

val rdd= sc.parallelize(set,8)

val res= rdd.map(x=>Math.sqrt(x)).filter(x=>(x %2)==0).filter(x=>(x%3)==0)

res.count

res.collect
```

- Code-Fragment enthält zwei Actions: count und collect
- Für jede Action wird res einmal berechnet
- In diesem Fall wäre eine zweite Berechnung (sofern genügend Speicherplatz vorhanden) nicht erforderlich

Caching

```
val set=Range(1,1000000)

val rdd= sc.parallelize(set,8)

val res= rdd.map(x=>Math.sqrt(x)).filter(x=>(x %2)==0).filter(x=>(x%3)==0).cache

res.count

res.collect
```

- Cache speichert das berechnete RDD auf dem JVM Heap
- Egal wie viel Actions auf dem RDD ausgeführt werden, es wird nur einmal berechnet
- Ist der Speicher voll, so wird nach dem Prinzip LRU (Least Recently Used) der Speicher freigegeben
- Fällt ein Knoten aus, so werden die Daten für den Knoten neu berechnet

Operationen: cache und persist

- Die Operation cache entspricht der Operation persist(StorageLevel.MEMORY_ONLY)
- Unpersist löscht das RDD wieder aus dem Speicher

Level	Space Used	CPU time	In Memory	On Disk
MEMORY_ONLY	High	Low	Yes	No
MEMORY_ONLY_SER	Low	High	Yes	No
MEMORY_AND_DISK	High	Medium	Some	Some
MEMORY_AND_DISK_SER	Low	High	Some	Some
DISK_ONLY	Low	High	No	Yes

Operationen der Relationalen Algebra

Arbeiten mit Key/Value-Paaren

Kartesisches Produkt

Verknüfung von zwei Mengen, in dem jedes Element der einen Menge mit jedem Element der anderen Menge verknüpft wird.

Umsetzung in Spark:

```
val l=List(1,2,3,4,5)

val rdd=sc.parallelize(l)

val res=rdd.cartesian(rdd)

res.collect
```

Array[(Int, Int)] = Array((1,1), (1,2), (1,3), (1,4), (1,5), (2,1), (2,2), (2,3), (2,4), (2,5), (3,1), (3,2), (3,3), (3,4), (3,5), (4,1), (4,2), (4,3), (4,4), (4,5), (5,1), (5,2), (5,3), (5,4), (5,5))

Key/Value-Paare

- Key/Value-Paare in Scala sind 2er-Tupel, bei denen der erste Wert als Schlüssel fungiert und der zweite als Value
- Klasse PairRDDFunctions erlauben viele hilfreiche Operationen auf Key/Value-Paaren (reduceByKey, aggregateByKey,…)
- Jedes RDD, das aus 2er-Tupeln besteht, kann diese Funktionen nutzen – Konvertierung erfolgt implizit
- Über Key/Value-Paare können Operationen der Relationalen Algebra abgebildet werden

Arbeiten mit Key/Value-Paaren: Joins

```
val m1=List((1,"a"),(2,"b"),(3,"c"))
val m2=List((1,"1"),(2,"2"),(4,"3"))
val rdd1= sc.parallelize(m1)
val rdd2= sc.parallelize(m2)
val res= rdd1 join rdd2
res.collect
 Array[(Int, (String, String))] = Array((1,(a,1)), (2,(b,2)))
```

Arbeiten mit Key/Value-Paaren: LeftOuterJoins

```
val m1=List((1,"a"),(2,"b"),(3,"c"))

val m2=List((1,"1"),(2,"2"),(4,"3"))

val rdd1= sc.parallelize(m1)

val rdd2= sc.parallelize(m2)

val res= rdd1 leftOuterJoin rdd2

res.collect

 Array[(Int, (String, Option[String]))] = Array((1,(a,Some(1))),
(2,(b,Some(2))), (3,(c,None)))
```

Arbeiten mit Key/Value-Paaren: RightOuterJoins

```
val m1=List((1,"a"),(2,"b"),(3,"c"))

val m2=List((1,"1"),(2,"2"),(4,"3"))

val rdd1= sc.parallelize(m1)

val rdd2= sc.parallelize(m2)

val res= rdd1 RightOuterJoin rdd2

res.collect

 Array[(Int, (Option[String], String))] = Array((1,(Some(a),1)),
(2,(Some(b),2)), (4,(None,3)))
```

Arbeiten mit Key/Value-Paaren: FullOuterJoins

```
val m1=List((1,"a"),(2,"b"),(3,"c"))

val m2=List((1,"1"),(2,"2"),(4,"3"))

val rdd1= sc.parallelize(m1)

val rdd2= sc.parallelize(m2)

val res= rdd1 fullOuterJoin rdd2

res.collect

Array[(Int, (Option[String], Option[String]))] = Array((1,
(Some(a),Some(1))), (2,(Some(b),Some(2))), (3,
(Some(c),None)), (4,(None,Some(3))))
```

Arbeiten mit Key/Value-Paaren: subtractByKey

```
val m1=List((1,"a"),(2,"b"),(3,"c"))
val m2=List((1,"1"),(2,"2"),(4,"3"))
val rdd1= sc.parallelize(m1)
val rdd2= sc.parallelize(m2)
val res= rdd1 subtractByKey rdd2
res.collect
Array[(Int, String)] = Array((3,c))
```

Vielen Dank für

Ihre Aufmerksamkeit

Prof. Dr.-Ing. Hendrik Gärtner