Spezielle Kapitel Sozialer Web – Technologien - Schleifen

Seminaristischer Unterricht

Prof. Dr.-Ing. Hendrik Gärtner

Gliederung

- Übungsaufgaben
- Dictionaries Erzeugung - GroupBy-Funktion
- Faltungsfunktionen
- MapReduce-Beispiel WordCount
- MapReduce – Die Google-Variante
  - Aufbau / Beschreibung der einzelnen Stufen
  - Beispiel WordCount

Aufgaben

```
def *(value:Double):Matrix={
        new Matrix(for (row <- data) yield row map (_ * value))
  }


// multiplies each row with the vector
 def *(v:Vector):Vector={
  require(columns==v.size)
  for {col <- data} yield
              col zip v map (X=>X._1*X._2) sum
 }
```

Aufgaben

```
// multiplies two matrices
  def *(matrix: Matrix):Matrix={


    require(this.columns==matrix.rows)
       new Matrix(for (row <- data)
                        yield for(col <- matrix.transpose.toArray)
                              yield row zip col map (x=>x._1 * x._2) reduceLeft (_ + _))
  }
```

Aufgaben

```
def createFromSparse(l:List[((Int,Int),Double)], row:Int, col:Int):Matrix={


  val init:Array[Array[Double]]= Array.tabulate(row,col)((X,Y)=>0)
  l foreach (X=>init(X._1._1)(X._1._2)=X._2)
  new Matrix(init)
}
```

Aufgaben – Sparse Matrix

```
def getRow(row_nr:Int):Vector={
    (for (i <- 0 to columns-1) yield data.getOrElse((row_nr,i), 0.0)).toArray
}


def *(matrix:SparseMatrix):SparseMatrix = {
    val m= new Matrix ((for (row <- 0 to rows-1) yield
            (for (col <- 0 to matrix.columns-1) yield Matrix.vectorMult(getRow(row),
                matrix.getColumn(col))).toArray).toArray)
      m.toSparseMatrix
  }
```

GroupBy-Funktion

Die groupBy-Funktion in Scala gruppiert eine Menge von Elementen nach einem Kriterium, das als Funktion übergeben wird: z.B.

val l= List(1,2,3,4,5,6)

l.groupBy(_ %3) → Map(2 -> List(2, 5),1 -> List(1, 4), 0 -> List(3, 6))

```
def groupBy[T,U](it:Iterable[T],
gbFun:T=>U):Map[U,List[T]]={

  val res:scala.collection.mutable.Map[U,List[T]]=
              scala.collection.mutable.Map()
    for (i <- it){
      res(gbFun(i))= i::res.getOrElse(gbFun(i),List())
    }
    // Mutable to immutable
    res.toMap
  }
```

Anwendung von GroupBy

Entwerfen Sie einen Funktionsaufruf, der eine Líste von Strings nach der Länge gruppiert. val s= List("Hallo", "dies", "ist","ein", “doller“,"Test")

groupBy[String,Int](s,_.length)
→ *Map(6->List(doller),5 -> List(Hallo), 4 -> List(dies, Test), 3 -> List(ist, ein))*

Entwerfen Sie einen Funktionsaufruf, der eine Liste von Wörtern nach ihren Anfangsbuchstaben gruppiert.

groupBy[String,Int](s,_.length)
→ *Map(e -> List(ein), T -> List(Test), i -> List(ist), H -> List(Ha*llo),
d -> List(doller, dies))

Faltungsfunktionen in Scala[1]

[1]Die hier abgebildeten Faltungsfunktionen sind in der Scala-Bibliothek anders implementiert. Sie dienen hier nur der Verdeutlichung der Funktionsweise.

Ansatz

- Parameter 1: Funktion zum Aggregieren des Ergebnisses
- Parameter 2: Basiswert – soll ausgegeben werden, wenn die Menge leer ist (ist in der Regel ein neutrales Element).
- Parameter 3: die Menge von Werten

```
def foldLeft[B,R](foldFun:(R,B)=>R, base:R, l:List[B]):R= l match{

      case Nil => base
      case x::xs => foldFun(foldLeft(foldFun, base, xs),x)
}
```

Zusammenfassen/Reduzieren von Listen: FoldLeft

FoldLeft reduziert von Links nach rechts:

foldLeft: (op, base, List(x_1,..x_n) ->op(...(op(base, x_1), x_2), ..., x_n)

Was ist der Typ: der Funktion?

foldLeft:: (((β, α) → β), β, List[α]) → β

```
def foldL[T,R] (f:(R,T)=>R, acc:R, list:List[T]):R = list match{
    case Nil => acc
    case x::xs => foldL(f,f(acc,x),xs)
}
```

Zusammenfassen/Reduzieren von Listen: FoldRight

FoldRight reduziert von rechts nach links:

foldRight: (op, acc, Liste(x_1,…,x_n)) ->

op(x_1, op(x_2, ... op(x_{n-1},op(x_n, acc))...))

Was ist der Typ der Funktion?

foldRight:: (((α,β) → β), β, List[α]) → β

```
def foldR[T,R](f:(T,R)=>R, base:R, list:List[T]):R = list match{
   case Nil => base
   case x::xs => f(x,foldR(f,base,xs))
}
```

GroupBy-Funktion

Die groupBy-Funktion in Scala gruppiert eine Menge von Elementen nach einem Kriterium, das als Funktion übergeben wird: z.B.

val l= List(1,2,3,4,5,6)

l.groupBy(_ %3) → Map(2 -> List(2, 5),1 -> List(1, 4), 0 -> List(3, 6))

```
def groupBy[T, U](in: Iterable[T], f: T => U) = {
          in.foldLeft(Map.empty[U, List[T]]) {
        (map, t) =>
          val groupByVal = f(t)
          map.updated(groupByVal, t :: map.getOrElse(groupByVal, List.empty))
      }.mapValues(_.reverse)
    }
```

Entwerfen Sie einen Funktionsaufruf, die eine Líste von Strings nach der Länge gruppiert. val s= List("Hallo", "dies", "ist","ein", "Test")

s.groupBy(_.length) → *Map(5 -> List(Hallo), 4 -> List(dies, Test), 3 -> List(ist, ein))*

MapReduce

MapReduce-Algorithmen kombinieren die Anwendung einer Map-Funktion mit der Anwendung einer Reduce-Funktion. Wie könnte solch eine Funktion aussehen?

```
def mapReduce[S,B,R](mapFun:(S=>B),
                     redFun:(R,B)=>R,
                     base:R,
                     l:List[S]):R =

    l.map(mapFun).foldLeft(base)(redFun)
```

Beispiel Wörter zählen

Gegeben sei ein Text, aus dem wir die Wörter extrahieren und zählen.

Wie sieht die Vorgehensweise aus?

- Extrahieren der Wörter
- Zuordnung einer Wertigkeit 1 zu einem Wort
- Wortbasiertes zusammenzählen der Einsen

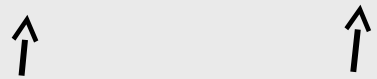
Wie sieht die Map-Funktion aus und wie die Reduce-Funktion? Welche Typen haben sie?

Map-Funktion

Die Map-Funktion bildet einen String auf einen Tupel bestehend aus einem String und einem Integer ab.

Beispiel: “Test“ => (“Test“,1)

String => (String, Int)



Typ S Typ B

Und was soll rauskommen?

Wordcount(„Dies dies ist ein Test“) => List((“dies“,2), (“ist“,1), (“ein“,1), (“test“,1) also **List[(String, Int)]**

Was macht die Reduzierfunktion?

Reduzier-Funktion

Die Reduzierfunktion bekommt ein Tupel, bestehend aus einem String und einem Integer und fügt diesen in eine Liste von Tupeln ein – also

redFun:(String,Int), List[(String,Int))=> List[(String,Int)

```
def insertL(l:List[(String, Int)], el:(String,Int)):List[(String, Int)] = l match {

    case Nil => List(el)
    case x::xs if (el._1.equals(x._1)) => (el._1, el._2 + x._2)::xs
    case x::xs => x::insertL(xs,el)
}
```

Wie sieht jetzt die Gesamtfunktion aus?

WordCount - Gesamtfunktion

```
def countWords(text:String):List[(String,Int)]= {

    val t= text.toLowerCase.replaceAll("[^a-z]", " ")
    val wl= t.split(" ").toList

    mapReduce[String, (String, Int), List[(String, Int)]]
                (X=>(X,1),      <—— MapFun
                insertL,        <—— RedFun
                List(),         <—— Basis
                wl)
    }
```

Drei Arten die Map-Reduce-Definition

```
def mapReduce[S,B,R](mapFun:(S=>B), redFun:(R,B)=>R, base:R, l:List[S]):R =

    foldLeft[B,R](redFun, base, map[S,B](mapFun, l))
```

```
def mapReduce2[S,B,R](mapFun:(S=>B), redFun:(R,B)=>R, base:R, l:List[S]):R
    = l match {

    case Nil => base
    case x::xs => redFun( mapReduce2(mapFun, redFun, base, xs), mapFun(x))
}
```

Drei Arten die Map-Reduce-Definition

```
def mapReduce3[S,B,R](mapFun:(S=>B), redFun:(R,B)=>R,
        akku:R, l:List[S]):R = l match {

    case Nil => akku
    case x::xs => mapReduce3(mapFun, redFun, redFun(akku, mapFun(x)), xs)
}
```

Welche der Definitionen ist die Effizienteste?

- Variante 1: Nutzung der effizienten Implementierungen der Scala-Bibliothek (hier nicht – Funktionen lassen sich austauschen)
- Variante 2 und 3: Das Ergebnis der Map-Funktion muss nicht im Speicher gehalten werden
- Variante 3 ist Tail-rekursiv

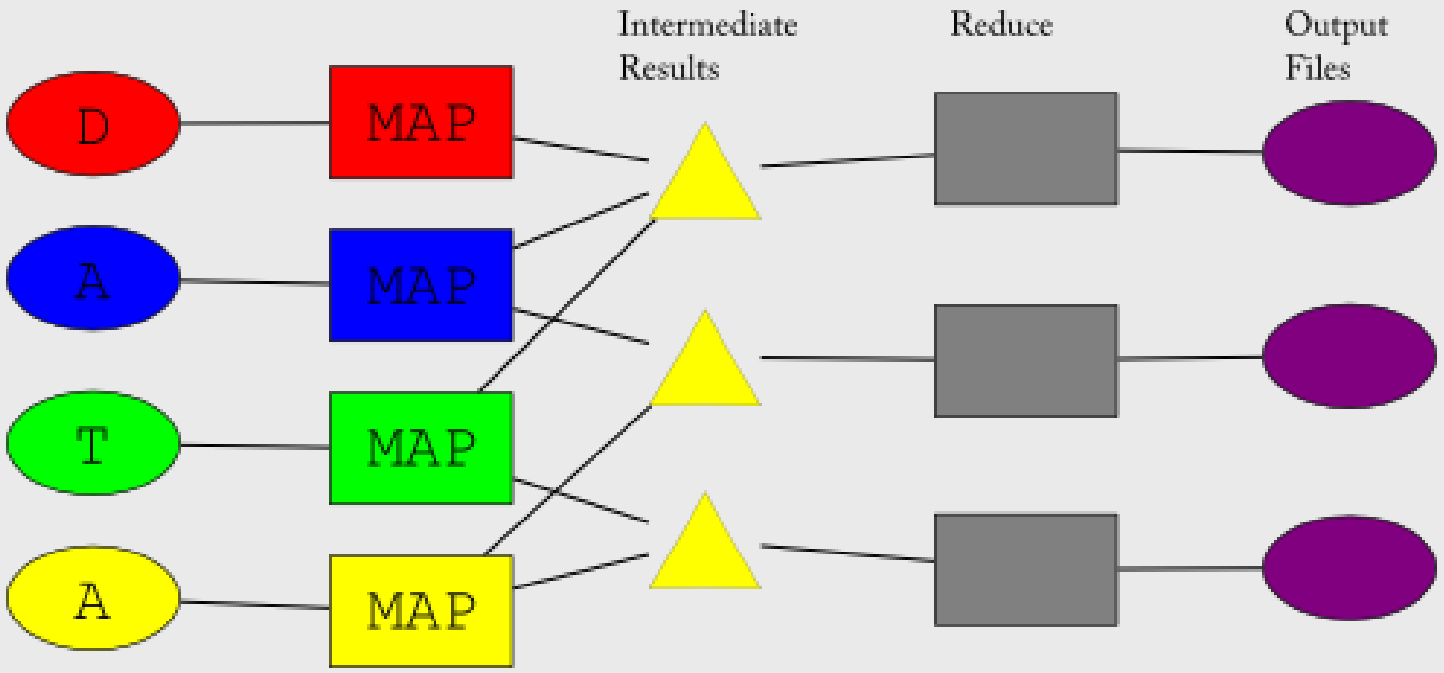
Tail-Rekursion am Beispiel Summierung

```
def sum(i:Int):Int= i match{
    case _ if (i<=0) => 0
    case _=>i+sum(i-1)
}

import scala.annotation.tailrec
def sumTR(i:Int):Int={

    @tailrec def sum(i:Int, acc:Int):Int = i match {
        case _ if (i<=0) => acc
        ase _ => sum(i-1,acc+i)
    }
    sum(i, 0)
}
```

Tailrekursionen

- Tail-Rekursionen sind rekursive Funktionen, deren rekursiver Aufruf als letztes Statement in der Funktion steht
- Tail-Rekursionen haben den Vorteil, dass auf dem Stack der letzte Funktionsaufruf durch den aktuellen überschrieben wird (funktioniert dann wie eine Schleife, d.h. der Stack kann nicht überlaufen)
- Damit die Ersetzung stattfindet, muss die Funktion in Scala mit @tailrec annotiert werden
- @tailrec überprüft außerdem, ob die Rekursion überhaupt das Kriterium erfüllt

Verteiltes MapReduce



- MapReduce kommt aus der Funktionalen Programmierung
- Paradigma für das Behandeln von Massendaten
- Google führte 2004 ein Framework dafür ein
- Freie Nachimplementierung Hadoop
- Viele NoSQL-DB basieren auf MapReduce in Kombination mit B-Bäumen

MapReduce - Mapper

```
def mapper[KeyIn ,ValueIn, KeyMOut, ValueMOut]     (mapFun:
    ((KeyIn,ValueIn))=>List[(KeyMOut,ValueMOut)],
        data:List[(KeyIn, ValueIn)]):List[(KeyMOut,ValueMOut)] = {

        data.flatMap(mapFun(_))
}
```

- Der Mapper bildet jedes Tupel auf eine Liste von Tupeln ab
- Der Typ der Funktion ist somit:
  (mapFun:((KeyIn,ValueIn))=>**List**[(KeyMOut,ValueMOut)]
- Damit alle entstehende Tupellisten auf eine Ebene gezogen werden, erfolgt ein flatMap

MapReduce - Sorter

```
def  sorter[KeyMOut,ValueMOut]
    (data:List[(KeyMOut,ValueMOut)]):List[(KeyMOut,List[ValueMOut])]= {

        data.groupBy(_._1) mapValues(X=> X.map(_._2)) toList
    }
```

- Der Sorter gruppiert alle Elemente nach dem Schlüssel
- mapValues wendet eine Funktion auf alle Werte der Map an
- Über die Funktion mapValues wird der Schlüssel aus den Tupeln entfernt, so dass nur noch eine Liste von Werten übrig bleibt

MapReduce - Reducer

```
def reducer[KeyMOut,ValueMOut,KeyROut, ValueROut]

    (redFun:((KeyMOut,List[ValueMOut]))=>List[(KeyROut,ValueROut)],

    data:List[(KeyMOut,List[ValueMOut])]):List[(KeyROut, ValueROut)]={

        data flatMap (redFun(_))
}
```

- Der Reducer wandelt das (Schlüssel, Liste von Werten)-wieder in eine Liste von Tupeln um
- Die Reduce-Funktion ist also vom Typ: ((KeyMOut,List[ValueMOut]))=>**List**[(KeyROut,ValueROut)]
- Damit alle entstehende Tupellisten auf eine Ebene gezogen werden, erfolgt ein flatMap

Übungsaufgabe

Schreiben Sie eine Funktion findAnagrams(l:List[String]):List[(String, String)], die aus einer Liste von Texten alle Annagramme findet. Verwenden Sie dafür die MapReduce-Funktion. Ergebnis soll eine Liste von Tupeln sein, die alle Anagramme aufdeckt. Beachten Sie dabei, dass Anagramme nicht doppelt vorkommen.

```
def findAnagrams(l:List[String]):List[(String,String)] = {
   val start= l map ((-1,_))
   mapReduce[Int, String, String, String,String, String](
(X)=>{
          val t= X._2.toLowerCase.replaceAll("[^a-z]", " ")
          t.split(" ").toList.filter(Y=>Y!="").map(Z=>(Z.sorted,Z))
      },
       (AL) => (for (x <- AL._2; y <-AL._2 if (x != y)) yield
{if (x<y) (x,y) else (y,x)}).toSet.toList,
start
    )
}
```

Vielen Dank für

Ihre Aufmerksamkeit

Prof. Dr.-Ing. Hendrik Gärtner