MapReduce mit Hadoop

Lernziele / Inhalt

- Wiederholung MapReduce
- Map in Hadoop
- Reduce in Hadoop
- Datenfluss
- Erste Schritte
- Alte vs. neue API
- Combiner Functions
- mehr als Java

Wiederholung MapReduce

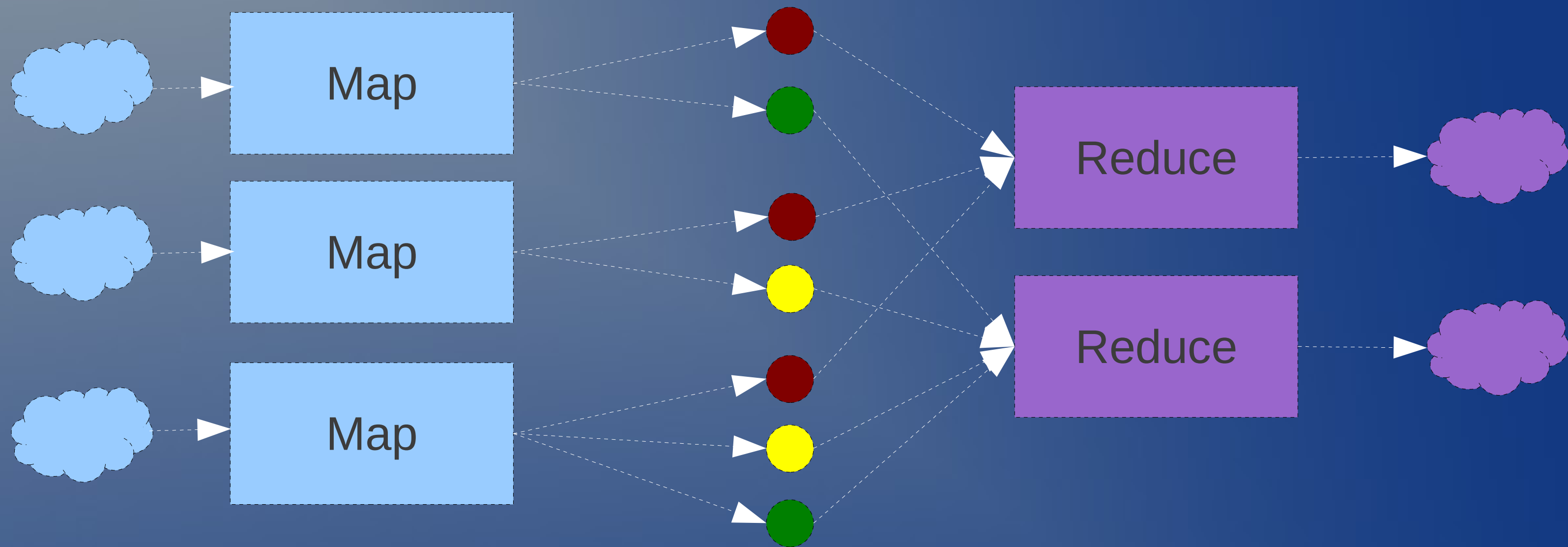
Eigenschaft von MapReduce

- Programmiermodell für Datenverarbeitung
- Inhärent parallel
- shared nothing
- Zwei Phasen
  - Map-Phase
  - Reduce-Phase
- Algorithmen typischerweise umgesetzt als Sequenz von MapReduce Operationen

Key-Value

	Input	Output
map	<k1, v1>	list(<k2,v2>)
reduce	<k2,list(v2)>	list(<k3, v3>)

Wordcount mit MapReduce



Shuffle + Sort

<1, rot grün gelb>
<2, gelb rot rot >….

<rot, 1>
<grün,1>
<gelb,1>

<gelb,1>
..

<rot, (1,1,..)>
<grün,(1,1,..)>
<gelb,(1,1,...>
..

<rot, 101>
<grün,77>
<gelb,98>
..

Map

Mapper

- `public static class Map extends Mapper<LongWritable, Text, Text, IntWritable>`
- Abstrakte generische Klasse `Mapper`
  - Package: `org.apache.hadoop.mapreduce`
  - Vier *type parameter:*
    - *Input Key*
    - *Input Value*
    - *Output Key*
    - *Output Value*

Types

- Hadoop-eigene Basis-Typen
  - optimiert für Netzwerk Serialisierung
  - in package `org.apache.hadoop.io`
    - `LongWritable` statt `long`
    - `IntWritable` statt `int`
    - `Text` statt `String`
    - `...`

map - context

- `map` Methode

```
public void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {.....}
```

- `org.apache.hadoop.mapreduce`

  `Class Mapper.Context`

  - auch für für Ergebnis

Reduce

Reducer

- `public static class Reduce extends Reducer<Text, IntWritable, Text, IntWritable>`
- Abstrakte generische Klasse `Reducer`
  - Package: `org.apache.hadoop.mapreduce`
  - Vier *type parameter:*
    - *Input Key* (selber Typ, wie map output key!)
    - *Input Value* (selber Typ, wie map output value!)
    - *Output Key*
    - *Output Value*

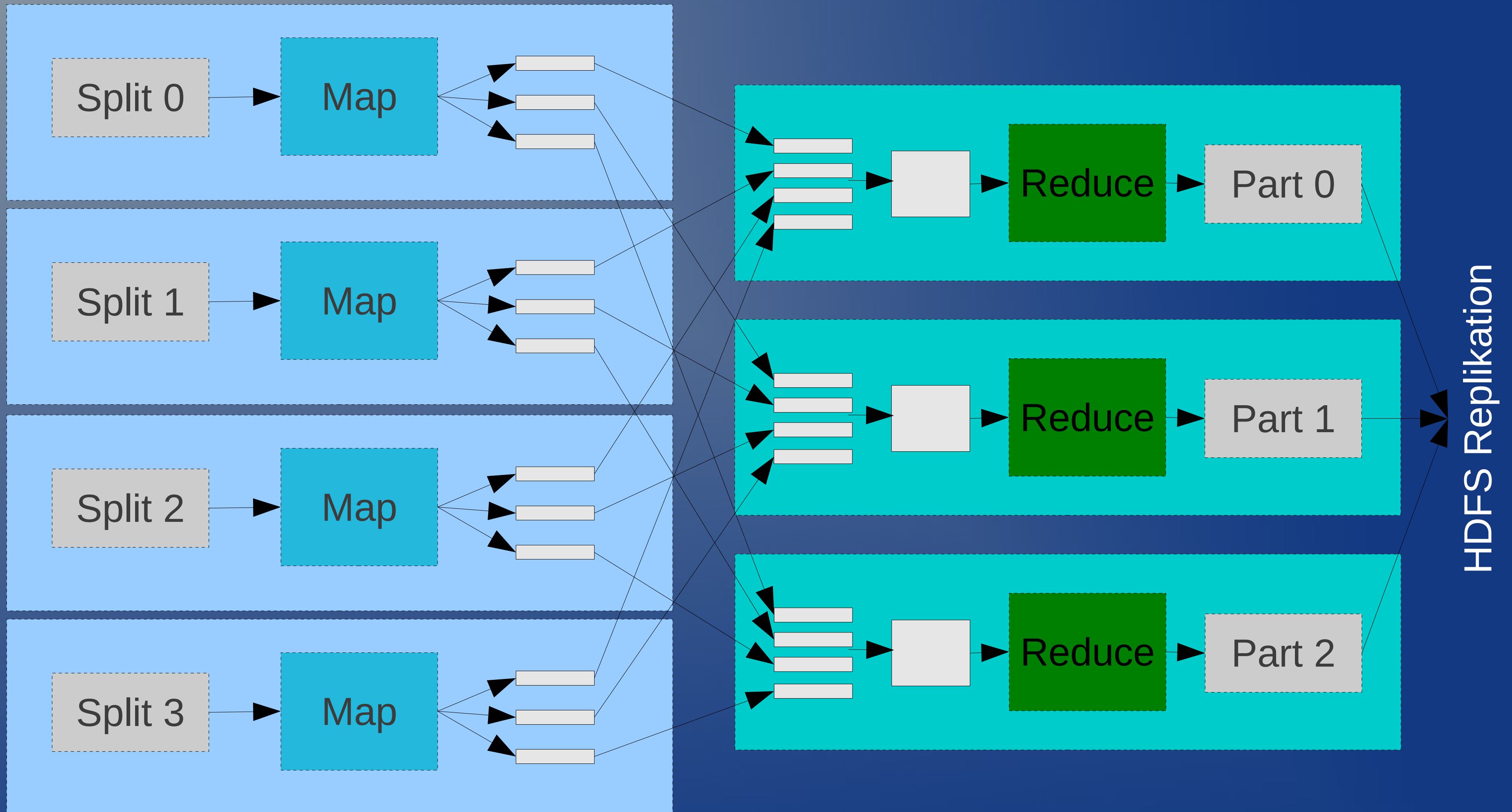
reduce method

- `public void reduce (Text key, Iterable<IntWritable> values, Context context) throws IOException, InterruptedException`

	Input	Output
reduce	<k2,list(v2)>	list(<k3, v3>)

Datenfluss

Datenfluss



- Map Output wird auf die lokale Platte gespeichert
- Reduce Tasks – keine *data locality*
- Reduce Output nach HDFS
  - Netzwerk Bandbreite wird verbraucht wegen Replikation
- Mehrere Reducer => map task teilen ihren output auf (eine Partition pro reducer)
  - kann durch user defined partitioning gesteuert werden
- Ohne Reducer: Shuffling nicht nötig (Optimierung!)
  - volle Parallelität bis auf HDFS Replikation

Output

- Output in HDFS
- Eine Datei pro Reducer

MapReduce Job ausführen

- mit Hilfe des `Job` Objekts
  - Spezifiziert den Job
  - Kontrolliert, wie Job ausgeführt wird
  - siehe Beispielcode

Erste Schritte

- Hadoop (pseudo distributed) installieren, siehe http://hadoop.apache.org/common/docs/r1.0.1/single_node_setup.html
- Wordcount bauen (mit Maven)
  - mvn clean package
- Hadoop starten
  - ${HADOOP_HOME}/bin/start-all.sh
  - eventuell log-Files kontrollieren (${HADOOP_HOME}/logs
- Text-Input Dateien ins HDFS kopieren
  - ./bin/hadoop dfs -copyFromLocal /home/chris/tmp/files /inputText

Erste Schritte

- Kontrollieren: bin/hadoop dfs -ls /
- ausführen: ./bin/hadoop jar ~/workspace/wordcount/target/wordcount-0.0.1-SNAPSHOT-job.jar /inputText /outputCounts/
- output gibt nützliche Infos, wie
  - ID des Jobs: job_201203291058_0002
  - Zahl der Map und Reduce Tasks
- ./bin/hadoop dfs -ls /outputCounts
- ins lokale FS kopieren und kontrollieren
  - ./bin/hadoop dfs -getmerge /outputCounts ~/tmp/outputcounts
  - head ~/tmp/outputcounts
  - sort -rnk 2 ~/tmp/outputcounts > ~/tmp/outputcountssorted
  - head ~/tmp/outputcountssorted

alte API

alte vs neue API

- alte API vor Hadoop 0.20.0
  - in org.apache.hadoop.mapred
  - type-incompatible zur neuen 1.x (0.20)
- neue API nicht vollständig in 1.x
  - => alte wird empfohlen
- Unterschiede in der Neuen
  - abstrakte Klassen statt Interfaces
  - Context Objects für Kommunikation mit MapReduce System
    - `Context` statt `JobConf, OutputCollector, Reporter`
  - mapper und reducer können `run()` überschreiben
  - `InteruptedException` Handling möglich
  - ...

Optimierung mit *Combiner Functions*

Aufgabe von Combiner Functions

- Viele MapReduce Jobs sind durch die Bandbreite des Datentransfers „begrenzt“
- Ziel von *Combiner Functions*: Datentransfer zwischen Map und Reduce Phasen minimieren
- Combiner Functions arbeiten auf dem Map-Output und reduzieren die zu transferierende Datenmenge
- Wirken wie „mini-reducer“
- Hadoop-Framework garantiert nicht wie oft Combiner angewendet wird (auch überhaupt)

Beispiel Combiner

- Min-Value für jeden Map-Output Key finden
- Map liefert z.B. outputs:
  - node0: <“DB1“, 1.3>, <“DB1“, 3.0>, <“DB1“,2.7>
  - node1: <“DB1“, 1.7>, <“DB1“, 3.3>, <“DB1“,2.3>
- Combiner kann auf einem Node schon Min-Funtion aufführen
  - combine(<“DB1“, 1.3>, <“DB1“, 3.0>, <“DB1“,2.7>) = <“DB1“, 1.3>
  - combine(<“DB1“, 1.7>, <“DB1“, 3.3>, <“DB1“,2.3>) = <“DB1“, 1.7>

Reducer als Combiner

- Reducer als Combiner möglich, falls die Reducer Operation:
  - kommutativ: a * b = b * a
  - assoziativ: a * (b * c) = (a * b) * c
- Im Allgemeinen sind Combiner und Reducer nicht austauschbar

Beispiel: Reducer nicht als Combiner einsetzbar

- Durchschnitt (Mean)
  - Mean(7, 6, 3, 4, 5) = Mean(Mean(7, 6), Mean(3, 4, 5))
- Ansatz
  - Statt Mean berechnet Combiner zwei Values: `sum` und `count`
  - Reducer berechnet aus sum und count den Durchschnitt

Prinzipieller Ansatz (nicht lauffähig) aus [LD]

- class Mapper
  - method Map(string t, integer r)
    - Emit(string t, integer r)
-
- class Combiner
  - method Combine(string t, integers [r1 , r2 , . . .])
    - sum ← 0
    - cnt ← 0
    - for all integer r ∈ integers [r1 , r2 , . . .] do
      - sum ← sum + r
      - cnt ← cnt + 1
    - Emit(string t, pair (sum, cnt))
    -
- class Reducer
  - method Reduce(string t, pairs [(s1 , c1), (s2 , c2) . . .])
    - sum ← 0
    - cnt ← 0
    - for all pair (s, c) ∈ pairs [(s1 , c1), (s2 , c2) . . .] do
      - sum ← sum + s
      - cnt ← cnt + c
    - ravg ← sum/cnt
    - Emit(string t, integer ravg)
-

Input/Output für Combiner

- Input für Combiner muss gleich Output sein
- Input/Output von Combiner muss dem Map-Output bzw. dem reduce Input entsprechen
- Combiner stellen lediglich eine Optimierung dar!
  - Was würde passieren wenn Combiner nicht aufgerufen werden, bzw. mehrfach aufgerufen werden?

Korrekte Lösung (lauffähig) aus [LD]

```
class Mapper
  method Map(string t, integer r)
   Emit(string t, pair (r, 1))

class Combiner
   method Combine(string t, pairs [(s1, c1 ), (s2, c2 ) ...])
    sum ← 0
    cnt ← 0
    for all pair (s, c) ∈ pairs [(s1, c1 ), (s2, c2 ) ...] do
      sum ← sum + s
      cnt ← cnt + c
    Emit(string t, pair (sum, cnt))

class Reducer
  method Reduce(string t, pairs [(s1 ,c1 ), (s2 ,c2 ) ...])
    sum ← 0
    cnt ← 0
    for all pair (s, c) ∈ pairs [(s1 , c1 ), (s2 , c2 ) . . .] do
       sum ← sum + s
       cnt ← cnt + c
    ravg ← sum/cnt
    Emit(string t, integer ravg )
```

Combiner Klassen anwenden

- ```
  Job job = new Job();
  ...
  job.setCombiner(myRed.class)
  ```
  ```

Zustand und Seiteneffekte

Mapper Object

- Hook für Initialisierungs-Code für jeden Map Task für Mapper Objekt:
  - `protected void setup(Mapper.Context context) throws IOException, InterruptedException`
- Zustand kann zwischen map-Aufrufen gehalten werden
- Hook für Abschluss-Code bei Beendigung des Map Task möglich
  - `protected void cleanup(Mapper.Context context) throws IOException, InterruptedException`

Reducer Object

- Hook für Initialisierungs-Code für jeden Map Task für Reducer Objekt:
  - `protected void setup(Reducer.Context context) throws IOException, InterruptedException`
- Zustand kann zwischen Reduce-Aufrufen gehalten werden
- Hook für Abschluss-Code bei Beendigung des Map Task möglich
  - `protected void cleanup(Reducer.Context context) throws IOException, InterruptedException`

Seiteneffekte

- Mapper und Reduce können „Side Effects“ haben
- Internal Side Effect
  - Lediglich interner Zustand wird geändert

    => kein Synchronisatonsproblem
- External Side Effect
  - z.B. Schreiben in HDFS
  - Vorsicht: Syncronisationsprobleme möglich

Ausblick auf weitere Themen

Patitioner

- Teilen den *intermediate key space* auf und weisen die keys den Reducern zu
- Einfachste Implementierung
  1) Hash Value auf Key
  2) HashValue modulo nbReducer
  - Ungeeignet für *skewed distributions*

Scheduling Problem

- Task unterschiedlicher Jobs werden parallel ausgeführt
- Bei großen Jobs mehr Tasks als Clients => task queue (Priorität)
- Langsame Tasks bestimmen Zeitdauer der Map-Phase (*stragglers* = Nachzügler)
  - Speculative Execution hilft bei Hardware Problemen
  - Bei Skew in Daten: Techniken wie *local aggregation*

Anzahl Map-Tasks

- Anzahl der Map Tasks: Entwickler kann Hinweise geben, aber bestimmt wird es vom Framework
  - Anzahl der Files und Blocks
- Anzahl der Reduce Task kann vom Entwickler bestimmt werden

Hadoop vs. Google MR

- Intermediate Values in Googles MR sortierbar über secondary sort
- In Googles MR:
  - Reducer input key = output key

Abwägung der Split Größe

- kleinere *splits*
  - besseres *load balancing*
  - *load balacing* für unterschiedliche Rechenpower der Nodes, failed processes, andere gleichzeitiglaufende *jobs*
- *größere splits*
  - management overhead (splits)
  - map task creation
- split Größe entspricht HDFS Block Größe (default 64MB)
  - max. mögliche Größe, die sicher auf einem Node gespeichert ist – data locality

Komplexere Keys oder Values

- Um komplexe Objekte als Key oder Value zu verwenden müssen diese das Interface `Writable` implementieren
- Keys müssen zusätzlich `WritableComparable` implementieren

Hadoop und andere Programmiersprachen als Java

nicht nur Java

- Hadoop Streaming
  - für andere Programmiersprachen
  - unix standard streams
- Hadoop Pipes: C++ Interface zu Hadoop RapReduce
  - über sockets – nicht JNI

Versionen

Versionen

- bis 0.20: alte API
- 1.x (formerly 0.20) release series
- newer 0.22 and 0.23 series
  - Hadoop 0.23: new MapReduce runtime, MapReduce 2 basiert auf YARN

Apache Hadoop 0.23

http://hadoop.apache.org/common/docs/r0.23.0/

- HDFS Federation
  - Skalieren des NameServers horizontal über verschiedene unabhängige NameNodes
- YARN (MapReduce Version2)
  - Aufspalten des JobTracker in 2 Dämonen-Prozesse
    - resource management
    - job scheduling/monitoring

Literatur

- Hadoop MapReduce Tutorial
  - http://hadoop.apache.org/common/docs/current/mapred_tutorial.html
- Tom White, „Hadoop The Definite Guide“, third edition, 2012, O'Reilly
  - Code: https://github.com/tomwhite/hadoop-book
- [LD] Jimmy Lin, Chris Dyer: "Data-Intensive Text Processing With MapReduce", Chapter 2-3
- JavaDoc
  http://hadoop.apache.org/common/docs/current/api/
- Hadoop Dokumentation
  http://hadoop.apache.org/common/docs/current/index.html