Hadoop - MapReduce

Team

- Marc Sieber
- Steve Vogel
- Samuel Keusch
- Kevin Buman

Aufgabe 1: Verkaufsanalyse (Verkaufsanalyse.java)

Verarbeitung Inputdatei

Die Map Funktion wird für jede Zeile aufgerufen. Die Zeile wird dann anhand von Tabulator-Zeichen aufgetrennt. Die Zeit ist dabei der 2. Wert (im Array der getrennten Werte). Die Zeit wird anhand des `:` (Doppelpunkts) aufgetrennt und anschliessend wird die Stunde in ein `Long` umgewandelt.
Auch der Betrag wird nach selbem Vorgehen extrahiert und in ein `Double` umgewandelt.

Output Map Funktion

Die Ausgabe der Map-Funktion ist ein Key-Value-Pair für jede Textzeile:

- **Key**<`LongWritable`>: Stunde extrahiert aus der Zeit
- **Value**<`DoubleWritable`>: Betrag des Einkaufs

Reduce Funktion

Die Reduce Funktion summiert alle Beträge derselben Stunde (also derselben Keys) auf und berechnet damit den durchschnittlichen Einkaufsbetrag.
Die Ausgabe der Reduce Funktion ist folgendes Key-Value-Pair:

- **Key**<`CustomLongWritable`>: Stunde
- **Value**<`Double`>: Durchschnittlicher Einkaufsbetrag

`CustomLongWritable` ist eine Subklasse von `LongWritable` und überschreibt die `toString` Methode. Damit können wir die Form des Outputs beeinflussen.

CustomPartitioner

Bei der Verwendung von mehreren Reducern ist uns aufgefallen, dass die Ausgabe dann nicht mehr sortiert ist. Wir haben herausgefunden, dass pro Reducer am Schluss eine Datei erstellt wird. Die Werte sind zwar innerhalb der Datei sortiert, aber nicht über mehrere Dateien hinweg. Der verwendete `HashPartitioner` stellt sich als Problem heraus. Deshalb haben wir einen eigenen Partitioner `CustomPartitioner` erstellt. Dieser weist der ersten Partition die tiefsten Keys (Stunden) zu und dem letzten Partitioner die grössten. So ist am Ende die Ausgabe wieder aufsteigend nach Stunden sortiert.
Die Funktion für das Berechnen, an welche Partition ein Key hinzugewiesen wird, hat aber einige kleine Nachteile:

- Die Werte werden nicht gleichmässig wie beim HashPartition auf die Reducer verteilt. Wir haben dass bereits ein wenig behoben, indem wir davon ausgehen, dass zwischen 00:00 und 06:00 sowie 20:00 und 24:00 keine Verkäufe durchgeführt werden. Auch wenn unsere gewählte Funktion nicht optimal ist, kann so die Leistung durch mehrere Reducer erhöht werden und gleichzeitig die Ausgabereihenfolge beibehalten werden.

Die könnte man natürlich auch noch nachträglich beheben in dem man die Textdateien z.B. mit Commandlinetools sortiert.

Auszug aus dem Output der Aufgabe 1 (alle Dateien konkateniert)

```
Stunde: 9       249.67
Stunde: 10      250.06
Stunde: 11      249.93
Stunde: 12      249.86
Stunde: 13      250.26
Stunde: 14      249.82
Stunde: 15      250.07
Stunde: 16      250.24
Stunde: 17      249.74
```

Aus der obigen Ausgabe kann man keinen nennenswerten Zusammenhang zwischen Stunde und Betrag ermitteln, da sich minimaler und maximaler Wert um weniger als 60 Rappen unterscheiden.

Source Code

Verkaufsanalyse.java

Quellen

https://intellipaat.com/community/43196/how-do-i-implement-a-custom-partitioner-for-a-hadoop-job

Aufgabe 2: Die 10 umsatzstärksten Verkaufsläden

Programm I (GroupByUmsatz_1.java)

Verarbeitung Inputdatei

Ähnlich wie in der vorherigen Aufgabe werden auch hier die Zeilen zuerst einzeln der `map` Funktion von `UmsatzMapper` übergeben. Dort wird wieder auf das Tabulator Zeichen `\t` gesplittet. Es werden Store (an dritter Stelle stehend) und der Preis (an fünfter Stellle stehend) extrahiert.

Output Map Funktion

Als Output der `map` Funktion bilden die Preise pro Filiale ein Key-Value-Pair. Hierbei handelt es sich immer noch um einen einzelnen Einkauf (Zeile in der Inputdatei)

- **Key**<`Text`>: Name der Filiale
- **Value**<`DoubleWritable`>: Einkaufspreis

Reduce Funktion

Die `reduce` Funktion summiert pro Key die Beträge auf. Der Output hat folgende Form:

- **Key**<`Text`>: Name der Filiale
- **Value**<`DoubleWritable`>: Summierte Preise über alle Einkäufe hinweg.

Auszug Output (Formatiert)

Ausgabe aller Umsätze pro Filiale.

```
Anaheim             1.007641635999996E7
Buffalo             1.0001941190000031E7
Chandler            9919559.860000001
Colorado Springs    1.0061105870000025E7
Dallas              1.0066548450000016E7
Durham              1.0153890209999988E7
Fremont             1.0053242359999955E7
Fresno              9976260.260000044
Irvine              1.0084867449999917E7
Jacksonville        1.0072003330000045E7
Laredo              1.014460497999991E7
Long Beach          1.0006380250000054E7
Madison             1.0032035539999941E7
Miami               9947316.070000034
New York            1.0085293549999993E7
Norfolk             1.0088563169999903E7
North Las Vegas     1.002965250999993E7
Orlando             1.0074922520000027E7
....
```

Program II (UmsatzRanking_2.java)

Verarbeitung Inputdatei

Als Input wird nun der Output des vorhergehenden Programms verwendet. In der Klasse `RankingMapper` wird eine `PriorityQueue<StoreRevenuePair>` verwaltet. Auch hier werden die Zeilen wieder einzeln eingelesen und der `map` Funktion von `RankingMapper` übergeben. Jede Zeile wird dann wieder auf `\t` gesplittet und die Filiale und Umsatz pro Filiale wird in einer `PriorityQueue` gespeichert. Die Intention ist, dass in der `PriorityQueue` für diesen Mapper die 10 umsatzstärken Filialen gespeichert werden. Sobald die `PriorityQueue` eine Grösse von 10 überschreitet, wird jeweils der letzte Key gelöscht. In der `cleanup` Funktion des Reducers werden dann die 10 umsatzstärksten Filialen mit `Context.write` geschrieben.

Output Map Funktion

Gibt in der `cleanUp` Funktion die 10 umsatzstärksten Filialen zurück.

- **Key**<`Text`>: Name der Filiale
- **Value**<`DoubleWritable`>: Summierte Preise über alle Einkäufe hinweg (für diese Filiale).

Reduce Funktion

Da es bei grossen Eingabedateien mehrere Mappers geben kann, reicht es nicht nur im Mapper zu sortieren und die 10 besten weiterzugeben. Gibt es mehrere Mappers könnte es passieren, dass wir am Schluss mehr als 10 Filialen haben. Der `RankingReducer` übernimmt also die Aufgabe, aus allen Mappers die 10 besten Filiale auszuwählen. Hierbei wird wieder auf das gleiche Konzept mit der `PriorityQueue` gesetzt wie beim Mapper.

Im `cleanup` des Reducers werden dann die 10 Einträge aus dem `PriorityQueue` extrahiert und sortiert ausgegeben. Dabei wird der Betrag auch noch entsprechend formatiert.

Auszug Output (Formatiert)

Ausgabe der Top 10 Umsatzstärksten Filialen.

```
Philadelphia    10'190'080.26
Durham          10'153'890.21
Laredo          10'144'604.98
Newark          10'144'052.80
Cincinnati      10'139'505.74
Washington      10'139'363.39
Irving          10'133'944.08
Fort Wayne      10'132'594.02
Baton Rouge     10'131'273.23
Sacramento      10'123'468.18
```

Source Code

GroupByUmsatz_1.java
UmsatzRanking_2.java

Quellen

https://www.geeksforgeeks.org/how-to-find-top-n-records-using-mapreduce/
https://data-flair.training/forums/topic/how-to-calculate-number-of-mappers-in-hadoop/

Validierung der Resultate

Um frühzeitig eventuelle Fehler entdecken zu können, haben wir vorgängig die erwarteten Werte innerhalb eines Jupyter Notebooks mittels `Pandas` ermittelt. Die erhaltenen Resultate stimmen mit dem obigen überein.

Resultate der Aufgabe 1

Berechnen diverser statistischen Werten

```
1 df.groupby('time_split').mean()
```

```
                price  time_decimal
time_split
9           249.672185      9.491523
10          250.056104     10.491549
11          249.928080     11.491673
12          249.855358     12.491879
13          250.258298     13.491775
14          249.822076     14.491908
15          250.073113     15.491356
16          250.243684     16.491623
17          249.740575     17.491734
```

Resultate der Aufgabe 2

Top 10 der Umsatzstärksten Filialen

```
1 df.groupby('place')[['place', 'price']].sum().sort_values(by='price')[-10:][::-1]
```

	price
place	
Philadelphia	10190080.26
Durham	10153890.21
Laredo	10144604.98
Newark	10144052.80
Cincinnati	10139505.74
Washington	10139363.39
Irving	10133944.08
Fort Wayne	10132594.02
Baton Rouge	10131273.23
Sacramento	10123468.18

Source Code

purchases analysis.ipynb