BDA, Praktikumsbericht 3

Gruppe mi6xc: Alexander Kniesz, Maximilian Neudert, Oskar Rudolf

Quellen

Das PySpark Notebook findet man hier.

Aufgabe 1

Wir lesen das `txt` file als DataFrame ein. Anschließend kann man mit `withColumn` diverse Operationen auf den Spalten ausführen. Das DataFrame enthält alle Zeilen des `txt` Files als Zeilen. In unserem Fall haben wir die Daten zuerst bereinigt, sprich Sonderzeichen und leere Zeilen entfernt. Anschließend haben wir mittels `split` die Strings in den Zeilen in Wörter Arrays umgewandelt, danach die Arrays mit `explode` in weitere Zeilen erweitern und abschließend MapReduce mit `lit` und `groupBy` gemacht.

```
%pyspark
from pyspark.sql.functions import explode, split, lit, trim, regexp_replace

# read text file
df = spark.read.text('hdfs://141.100.62.85:9000/data/lorem/lorem.txt')

# remove end of line whitespace
df = df.withColumn("value",trim(df.value))
# filter rows with empty strings
df = df.filter("value != ''")
# remove special characters
df = df.withColumn("value", regexp_replace("value", '[^a-zäüößA-ZÄÜÖ ]+', ''))
#df = df.withColumn("value",split(df.value,'\s+|,\s+|\.\s+|\!\s')) # split by "," and "whitespace"
# split by whitespace
df = df.withColumn("value",split(df.value,' '))
# expand list into rows
df = df.select(explode("value").alias("key"))
# add new column with 1's
df = df.withColumn("value",lit(1))
# sum counts
#df = df.groupBy("key").sum("value").select(col("key"),col("sum(value)").alias("value"))
df = df.groupBy("key").sum("value")
# rename
df = df.withColumnRenamed('sum(value)', 'value')
# order by value
df = df.orderBy("value", ascending=False)

# show result
df.show()
```

```
+--------+-----+
|     key|value|
+--------+-----+
|      et|   57|
|   dolor|   32|
|     sed|   27|
|    diam|   27|
|     sit|   26|
|   ipsum|   26|
```

Aufgabe 2

Wir verbinden uns mittels ssh auf:

```
141.100.62.87
```

dort haben wir eine `tmux` session mittels

```
tmux -S /tmp/smux new -s amo
```

erstellt, auf die wir uns dann mittels

```
tmux -S /tmp/smux attach -t amo
```

gemeinsam verbinden und mit netcat arbeiten können.

Code

```
%pyspark
#aufgabe 2: Deklaration und Start der query
from pyspark.sql.functions import explode, split, regexp_replace

sparkmi6xc = spark\
    .builder\
    .appName("mi6xcWordCounter")\
    .getOrCreate()

# read lines
lines = sparkmi6xc\
    .readStream\
    .format("socket")\
    .option("host", "141.100.62.87")\
    .option("port", 2242)\
    .load()

# remove special characters like '.' and ','
lines = lines.withColumn(
    "value",
    regexp_replace("value", '[^a-zäüößA-ZÄÜÖ ]+', '')
)

# split lines into words
words = lines\
    .select(
        explode(
            split(lines.value, " ")
        ).alias("word")
    )\
    .withColumn(
        "value",
        regexp_replace("word", '[^a-zäüößA-ZÄÜÖ ]+', '')
    )

# count words
wordCounts  = words\
    .groupBy("word").count()\
    .orderBy("count", ascending=False)

# Start running the query that prints the running counts to the console
writer = count_words \
    .writeStream \
    .queryName("mi6xcWordCount") \
    .outputMode("complete") \
    .format("memory")

query = writer.start()
```

unterschiede in den Parameters

`outputMode` besitzt folgende Paramter:

- **append**: Nur neue Zeilen werden in den Output geschrieben. Exklusiv für de Verwendung ohne Aggregationen.
- **complete**: Alle Zeilen werden jedes mal in den Output geschrieben. Exklusiv für die Verwendung mit Aggregationen.
- **update**: Nur veränderte Zeilen werden in den Output geschrieben. Ohne Aggregation wie **append**.

und `format` besitzt unter anderem folgende Parameter:

- **console**: Schreibt den verarbeiteten Stream in die Console sprich in den Standard Output.

- **memory**: Schreibt den verarbeiteten Stream in eine in Memory Datenbank. Folglich für große Datenmengen eher ungeeignet.

Testergebnisse

Wir haben die ersten 100 Wörter von Lorem Ipsum ein paar mal über `netcat` abgesendet und erhalten folgendes Ergebnis:

```
%sql
--aufgabe 2: Select der query und und Ergebnisvisualisierung
select * from mi6xcWordCount
```

settings

word	count
faucibus	9
ipsum	9
et	9
ut	9
elit	9
placerat	6
in	6
tincidunt	6

Wir modifizieren die query, indem wir die Splits anpassen, neue Spalten generieren und anschließend `url` auswählen und mit dieser gruppieren und zählen.

```
%pyspark
#aufgabe 3a: Deklaration und Start der query
from pyspark.sql.functions import explode, split, regexp_replace

sparkmi6xc = spark\
    .builder\
    .appName("mi6xcCookies")\
    .getOrCreate()

# read lines
stream = sparkmi6xc\
    .readStream\
    .format("socket")\
    .option("host", "starfall.fbi.h-da.de")\
    .option("port", 3333)\
    .load()

# split stream into lines
lines = stream\
    .select(
        explode(
            split(stream.value, '\n')
        ).alias("line")
    )

# split lines into columns
cols = split(lines['line'], '\t')
lines = lines\
    .withColumn('ts', cols.getItem(0))\
    .withColumn('url', cols.getItem(1))\
    .withColumn('status', cols.getItem(2))\
    .withColumn('country', cols.getItem(3))\
    .withColumn('userID', cols.getItem(4))\
    .drop('line')

url_counts  = lines\
    .select('url')\
    .groupBy('url').count()\
    .orderBy('count', ascending=False)

# Start running the query that prints the running counts to memory sink
writer = url_counts\
    .writeStream\
    .queryName("mi6xcCookieLines")\
    .outputMode("complete")\
    .format("memory")

query = writer.start()
```

Wir erhalten damit folgendes exemplarisches Ergebnis:

```
%sql
select * from mi6xcCookieLines
-- aufgabe 3a: Select der query
```

settings

url	count
https://cookiedream.net/about	99
https://cookiedream.net/	745
https://cookiedream.net/order	185

Aufgabe 3

a)

Sliding windows lassen sich durch die `sql` Funktion `window` im `groupBy` Befehl erstellen.

```
%pyspark
#aufgabe 3b: Deklaration und Start der query
from pyspark.sql.functions import explode, split, window

sparkmi6xc = spark\
    .builder\
    .appName("mi6xcCookieWindows")\
    .getOrCreate()

# read lines
stream = sparkmi6xc\
    .readStream\
    .format("socket")\
    .option("host", "starfall.fbi.h-da.de")\
    .option("port", 3333)\
    .load()

# split stream into lines
lines = stream\
    .select(
        explode(
            split(stream.value, '\n')
        ).alias("line")
    )

# split lines into columns
cols = split(lines['line'], '\t')
lines = lines\
    .withColumn('ts', cols.getItem(0))\
    .withColumn('url', cols.getItem(1))\
    .withColumn('status', cols.getItem(2))\
    .withColumn('country', cols.getItem(3))\
    .withColumn('userID', cols.getItem(4))\
    .drop('line')

url_counts  = lines\
    .select('url', 'ts')\
    .groupBy(
        'url',
        window('ts', '60 seconds', '30 seconds')
    ).count()\
    .select('url', 'count', 'window.start', 'window.end')\
    .orderBy(['url', 'start'], ascending=False)

# Start running the query that prints the running counts to memory sink
writer = url_counts\
    .writeStream\
    .queryName("mi6xcCookieWindows")\
    .outputMode("complete")\
    .format("memory")

query = writer.start()
```

b)

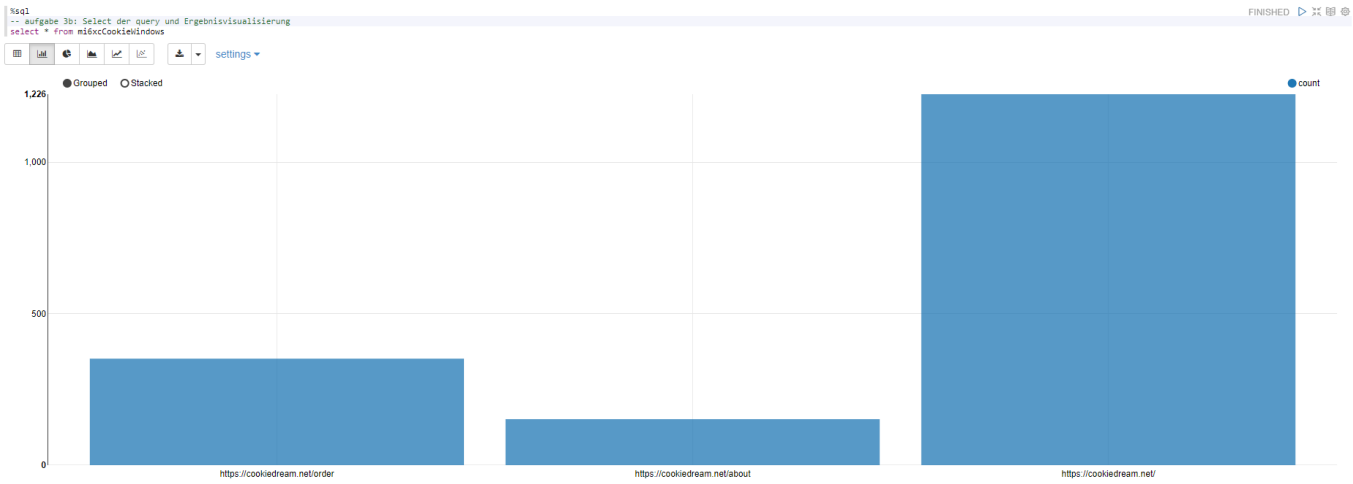
Tabellarisch erhalten wir folgenden Output sortiert nach url und Startzeit des Windows:

```
%sql
-- aufgabe 3b: Select der query und Ergebnisvisualisierung
select * from mi6xcCookieWindows
```

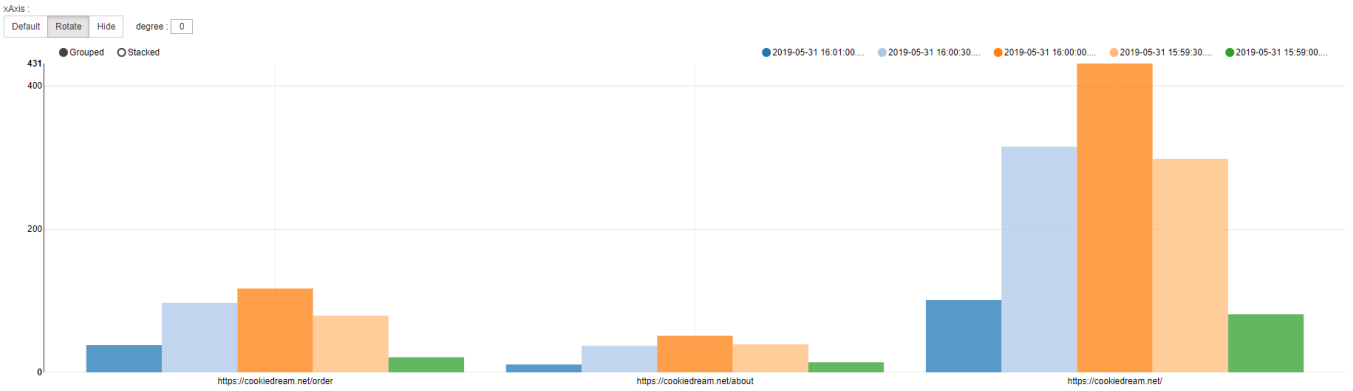
settings

url	count	start	end
https://cookiedream.net/order	38	2019-05-31 16:01:00.0	2019-05-31 16:02:00.0
https://cookiedream.net/order	97	2019-05-31 16:00:30.0	2019-05-31 16:01:30.0
https://cookiedream.net/order	117	2019-05-31 16:00:00.0	2019-05-31 16:01:00.0
https://cookiedream.net/order	79	2019-05-31 15:59:30.0	2019-05-31 16:00:30.0
https://cookiedream.net/order	21	2019-05-31 15:59:00.0	2019-05-31 16:00:00.0
https://cookiedream.net/about	11	2019-05-31 16:01:00.0	2019-05-31 16:02:00.0
https://cookiedream.net/about	37	2019-05-31 16:00:30.0	2019-05-31 16:01:30.0
https://cookiedream.net/about	51	2019-05-31 16:00:00.0	2019-05-31 16:01:00.0
https://cookiedream.net/about	39	2019-05-31 15:59:30.0	2019-05-31 16:00:30.0
https://cookiedream.net/about	14	2019-05-31 15:59:00.0	2019-05-31 16:00:00.0
https://cookiedream.net/	101	2019-05-31 16:01:00.0	2019-05-31 16:02:00.0
https://cookiedream.net/	315	2019-05-31 16:00:30.0	2019-05-31 16:01:30.0
https://cookiedream.net/	431	2019-05-31 16:00:00.0	2019-05-31 16:01:00.0
https://cookiedream.net/	298	2019-05-31 15:59:30.0	2019-05-31 16:00:30.0
https://cookiedream.net/	81	2019-05-31 15:59:00.0	2019-05-31 16:00:00.0

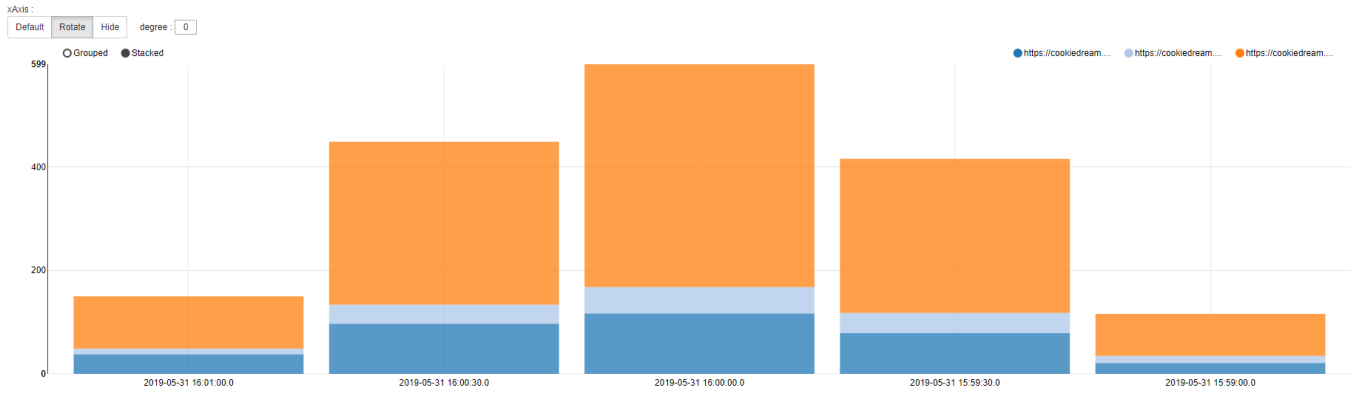
Visualisiert man zum Beispiel mit einer Barchart ohne irgendwelche Einstellungen, so erhält man die Counts abhängig von der URL.



Unter settings kann man dann ähnlich wie mit Tableau Daten gruppieren und aggregieren. Ein schöner Plot ergibt sich zum Beispiel, indem man mit der Startzeit des Windows zusätzlich gruppiert:



Im Prinzip macht es nur wirklich Sinn in diesem Fall `start` und `url` zwischen `keys` und `groups` zu tauschen. Ein weiterer schöner Graph ist `start` als Key und `url` als Group stacked.


xAxis :
Default
Rotate
Hide
degree : 0
Grouped
Stacked
https://cookiedream....
https://cookiedream....
https://cookiedream....
599
400
200
0
2019-05-31 16:01:00.0
2019-05-31 16:00:30.0
2019-05-31 16:00:00.0
2019-05-31 15:59:30.0
2019-05-31 15:59:00.0

Aufgabe 4

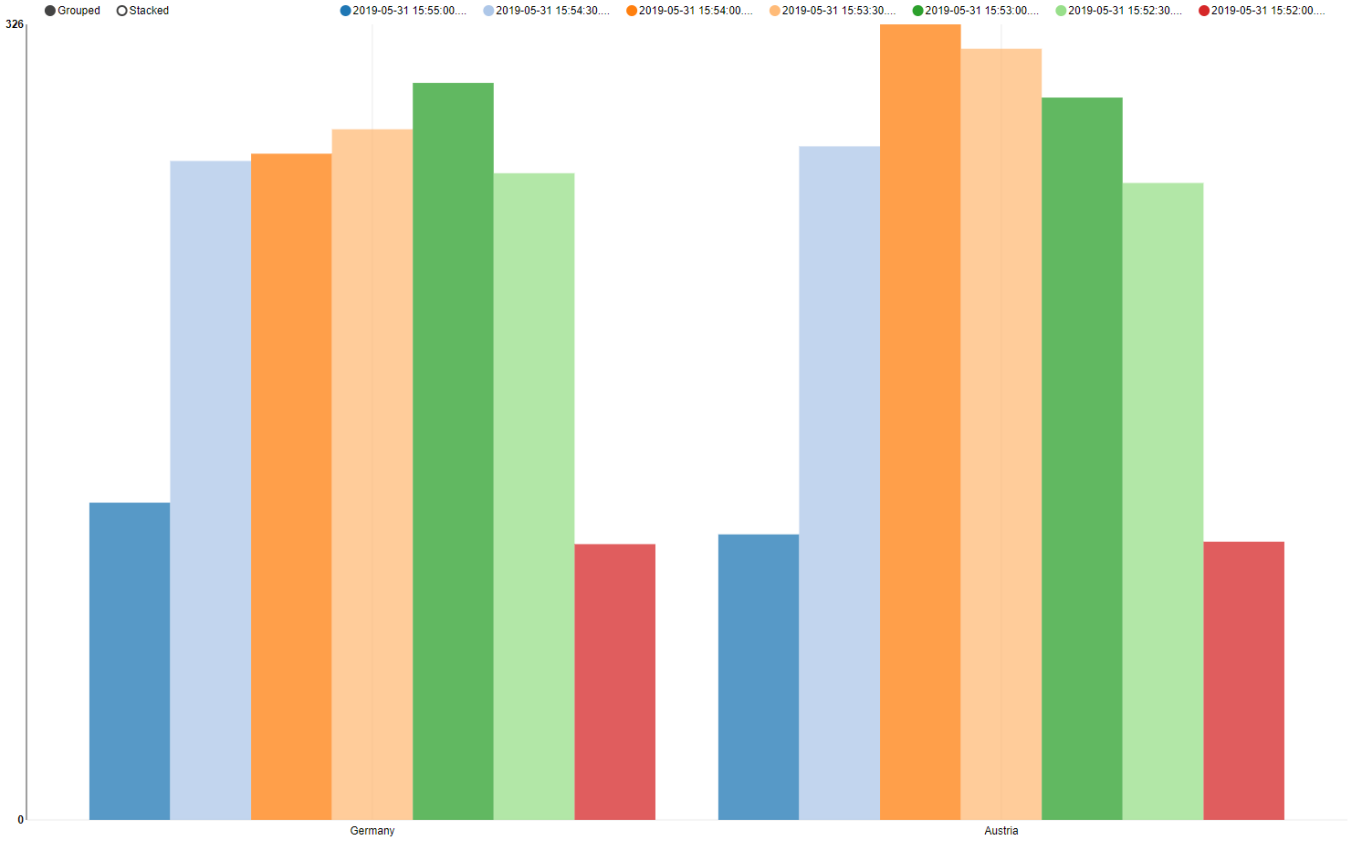
a)

Als Modifikation fügen wir eine weitere Spalte zur Gruppierung hinzu.

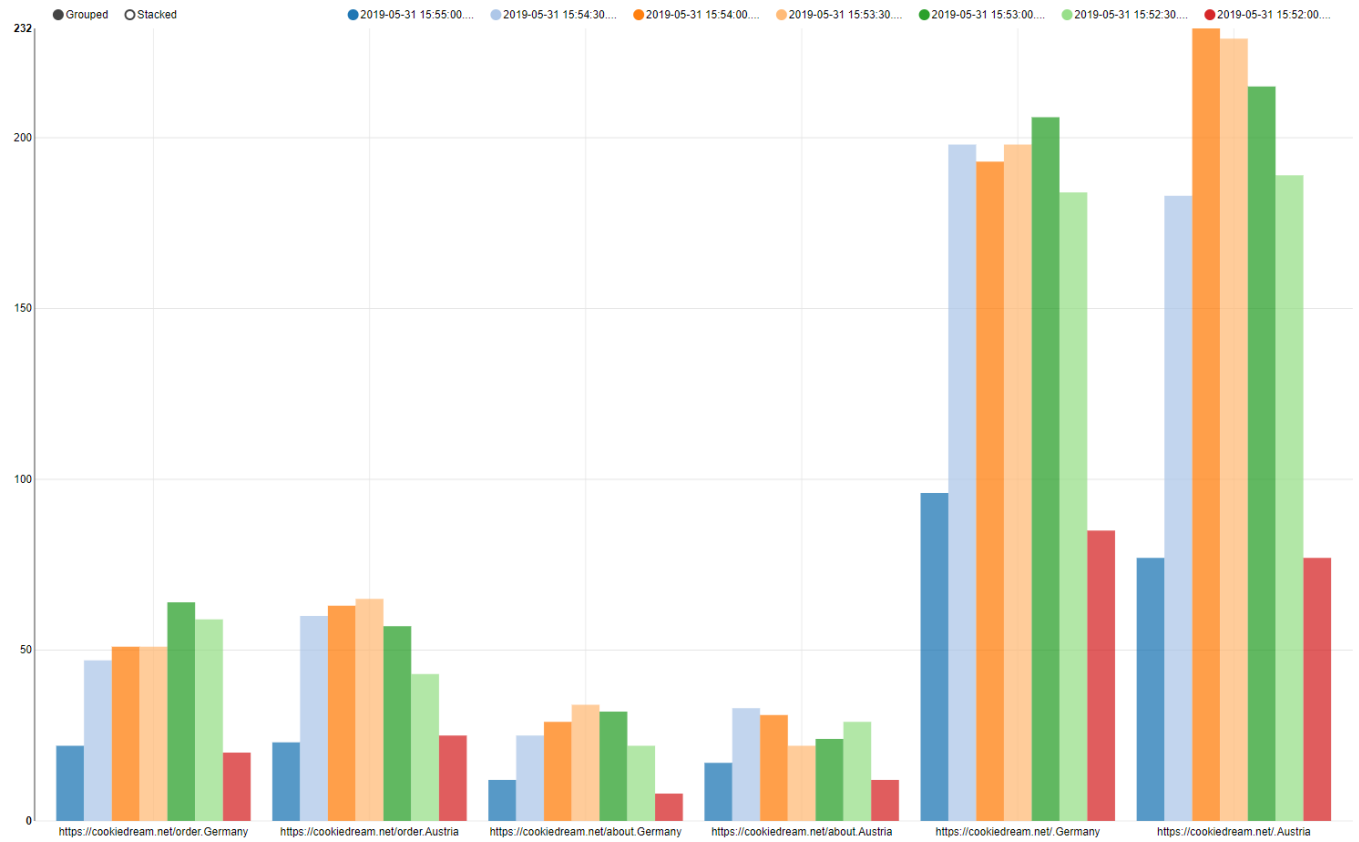
```
%pyspark
#aufgabe 4a: Deklaration und Start der query
from pyspark.sql.functions import explode, split, window

sparkmi6xc = spark\
    .builder\
    .appName("mi6xcCookieWindows")\
    .getOrCreate()

# read lines
stream = sparkmi6xc\
    .readStream\
    .format("socket")\
    .option("host", "starfall.fbi.h-da.de")\
    .option("port", 3333)\
    .load()

# split stream into lines
lines = stream\
    .select(
        explode(
            split(stream.value, '\n')
        ).alias("line")
    )

# split lines into columns
cols = split(lines['line'], '\t')
lines = lines\
    .withColumn('ts', cols.getItem(0))\
    .withColumn('url', cols.getItem(1))\
    .withColumn('status', cols.getItem(2))\
    .withColumn('country', cols.getItem(3))\
    .withColumn('userID', cols.getItem(4))\
    .drop('line')

url_counts  = lines\
    .select('url', 'country', 'ts')\
    .groupBy(
        'url',
        'country',
        window('ts', '60 seconds', '30 seconds')
    ).count()\
    .select('url', 'country', 'count', 'window.start', 'window.end')\
    .orderBy(['url', 'country', 'start'], ascending=False)

# Start running the query that prints the running counts to memory sink
writer = url_counts\
    .writeStream\
    .queryName("mi6xcCookieWindows")\
    .outputMode("complete")\
    .format("memory")

query = writer.start()
```

Zuerst einmal aus Interesse gesichtet, wie die Verteilung der Länder ist.

Dies kann man dann um die Länder erweitern.



Und schlussendlich um die URLs mit den Ländern als Gruppierung.



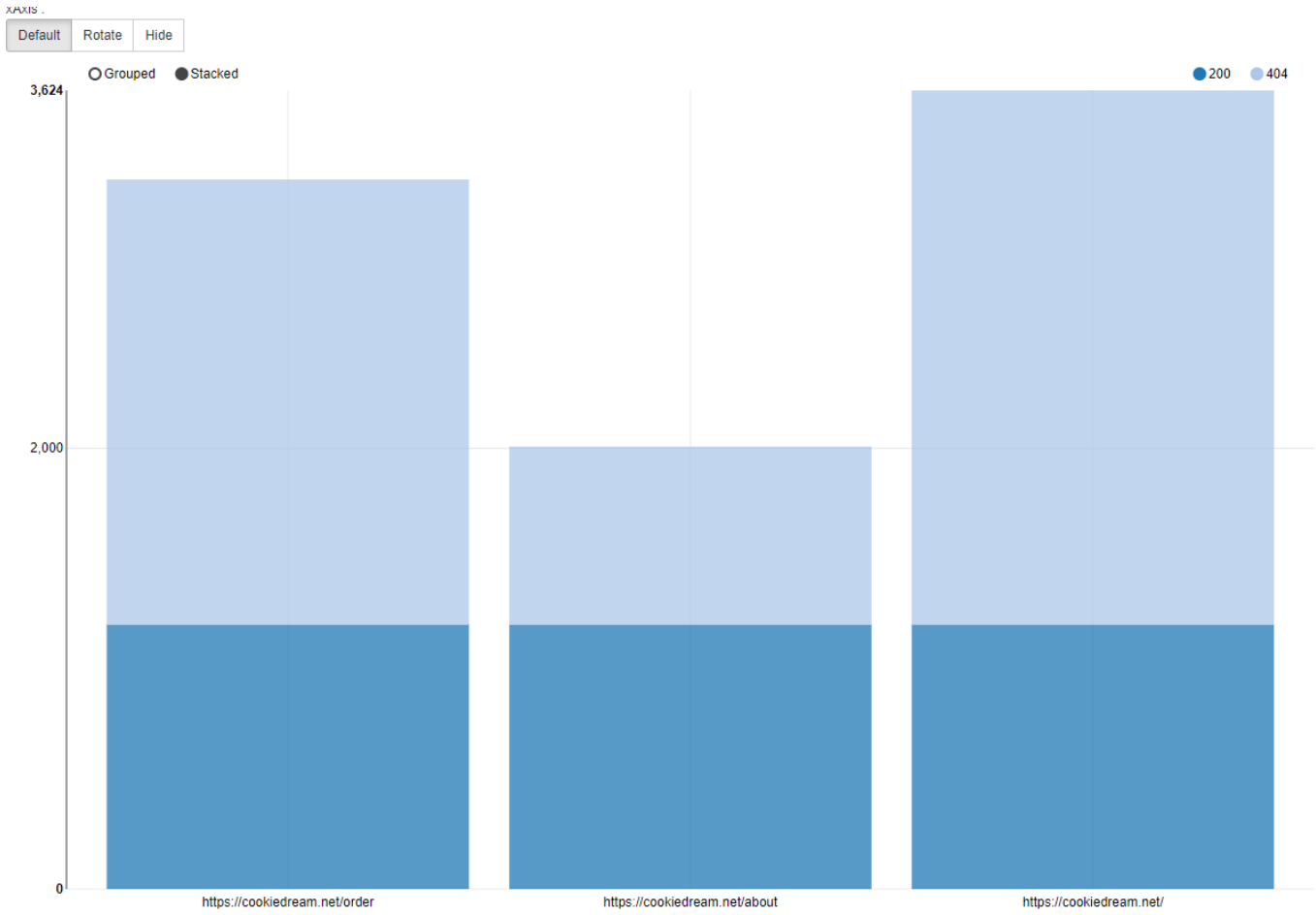
Dann erkennt man zum Beispiel, dass die meisten Aufrufe aus Österreich auf die Basisseite um 15:54 bis 15:55 waren.

b)

Wir bauen einen Filter mit `lines.country == 'Germany'` und ersetzen `country` mit `status`.

```
%pyspark
#aufgabe 4b: Deklaration und Start der query
from pyspark.sql.functions import explode, split, window

sparkmi6xc = spark\
    .builder\
    .appName("mi6xcCookieWindows")\
    .getOrCreate()

# read lines
stream = sparkmi6xc\
    .readStream\
    .format("socket")\
    .option("host", "starfall.fbi.h-da.de")\
    .option("port", 3333)\
    .load()

# split stream into lines
lines = stream\
    .select(
        explode(
            split(stream.value, '\n')
        ).alias("line")
    )

# split lines into columns
cols = split(lines['line'], '\t')
lines = lines\
    .withColumn('ts', cols.getItem(0))\
    .withColumn('url', cols.getItem(1))\
    .withColumn('status', cols.getItem(2))\
    .withColumn('country', cols.getItem(3))\
    .withColumn('userID', cols.getItem(4))\
    .drop('line')

url_counts  = lines\
    .select('url', 'country', 'status', 'ts')\
    .filter(lines.country == 'Germany')\
    .groupBy(
        'url',
        'status',
        window('ts', '60 seconds', '30 seconds')
    ).count()\
    .select('url', 'status', 'count', 'window.start', 'window.end')\
    .orderBy(['url', 'status', 'start'], ascending=False)

# Start running the query that prints the running counts to memory sink
writer = url_counts\
    .writeStream\
    .queryName("mi6xcCookieWindows")\
    .outputMode("complete")\
    .format("memory")

query = writer.start()
```

Anschließend erhalten wir folgendes Ergebnis:



Man sieht, dass `order` und `root` in `Germany` deutlich öfter `404` sprich nicht gefunden werfen als `about`.

c)

Zu den Unterschiedenlichen Ausgabemodi muss man zuerst daran denken, dass `append` (ohne Watermarks) nicht auf aggregierten Daten funktioniert, `complete` nur auf aggregierten Daten funktioniert und `update` nicht mit Sortierung funktionier. Deshalb haben wir die ersten beiden Aufgaben mit `complete` bearbeitet.

Late Data bedeutet, dass Daten mit einem späteren Eventzeitpunkt ankommen als deren Timestamp. Zur Bearbeitung von Late Data gibt es in Spark `watermarks`, welche mit `.withWatermark("timestamp", "threshold")` definiert werden können. Dabei ist `timestamp` die Eventzeit und `threshold` die maximale Zeit, die Daten zu spät sein dürfen (zum Beispiel `60 seconds`). Kommen nun Daten später an als der angegebene Threshold an, so werden diese nicht mehr für das Fenster aggregiert.

Arbeitet man mit Aggregation ohne Watermarks, so gibt es folgende Möglichkeiten für Late Data:

- Complete: Es werden grundsätzlich alle Daten für die Output Sink mit einbezogen und entsprechend aktualisiert. Späte Daten werden damit auch verarbeitet, da alle Daten inklusive der späten Daten zum Triggerzeitpunkt aggregiert werden.
- Update: Im Gegensatz zu Complete werden zum Triggerzeitpunkt alle neuen Daten zu den Alten aggregiert, somit auch späte Daten.

Arbeitet man mit Aggregation und mit Watermarks, so gibt es folgende Möglichkeiten für Late Data (Complete ist nicht mehr sinnvoll, da für Complete alle Daten für die Output Sink zu erhalten sind, aber man mit Watermarks gerade zu späte Daten ignorieren möchte.):

- Append: Zum Triggerzeitpunkt werden Daten noch bis zum Ende des angegebenen Delay Threshold gesammelt und dann gemeinsam zur Output Sink hinzugefügt. Alle Daten, die danach ankommen, werden ignoriert.
- Update: Zu jedem Triggerzeitpunkt werden die Daten in der Output Sink aktualisiert. Daten, deren Timestamp noch innerhalb des angegebenen Thresholds liegen werden dazu mit einbezogen, während Daten, die außerhalb liegen ignoriert werden.

Das heißt, dass die Output Sink mit Append später aktualisiert wird, dafür weniger Zeilenoperationen nötig sind und somit der Aufwand sinkt, während Update die Output Sink zu jedem Triggerzeitpunkt aktualisiert aber höheren Aufwand dafür besitzt.