

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 ]+', ''))
# 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-zäüö&A-ZÄÜÖ ]+', '')
)

# split lines into words
words = lines\
    .select(
        explode(
            split(lines.value, " ")
        ).alias("word")
    )\
    .withColumn(
        "value",
        regexp_replace("word", '[^a-zäüö&A-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
```



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()
```

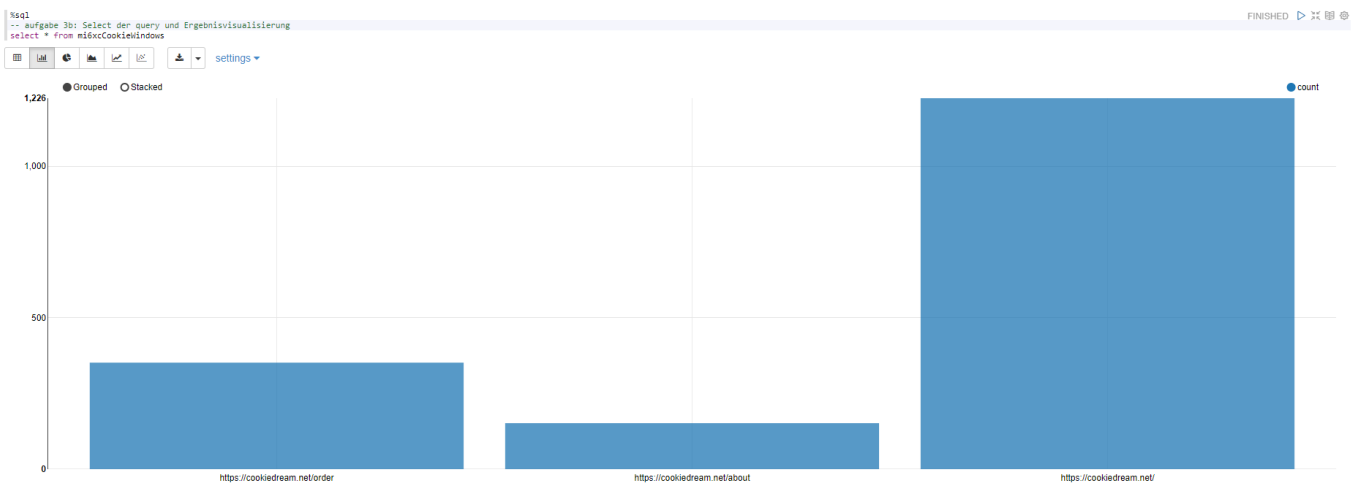
b)

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

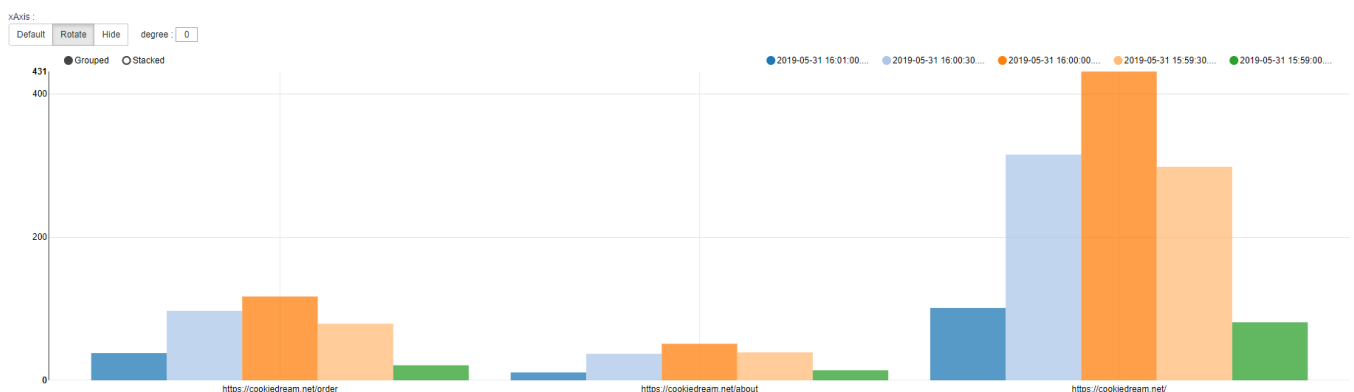
```
%sql
-- aufgabe 3b: Select der query und Ergebnisvisualisierung
select * from mi6xcCookieWindows
```

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

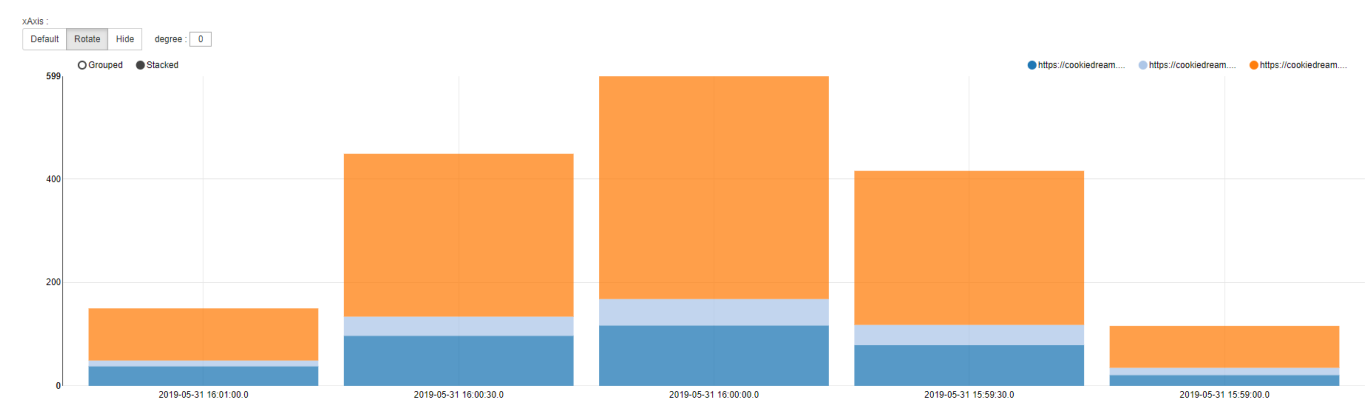
Visualisiert man zum Beispiel mit einer Barchart ohne irgendwelche Einstellungen, so erhält man die Counts abhängig von der URL.



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.



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)

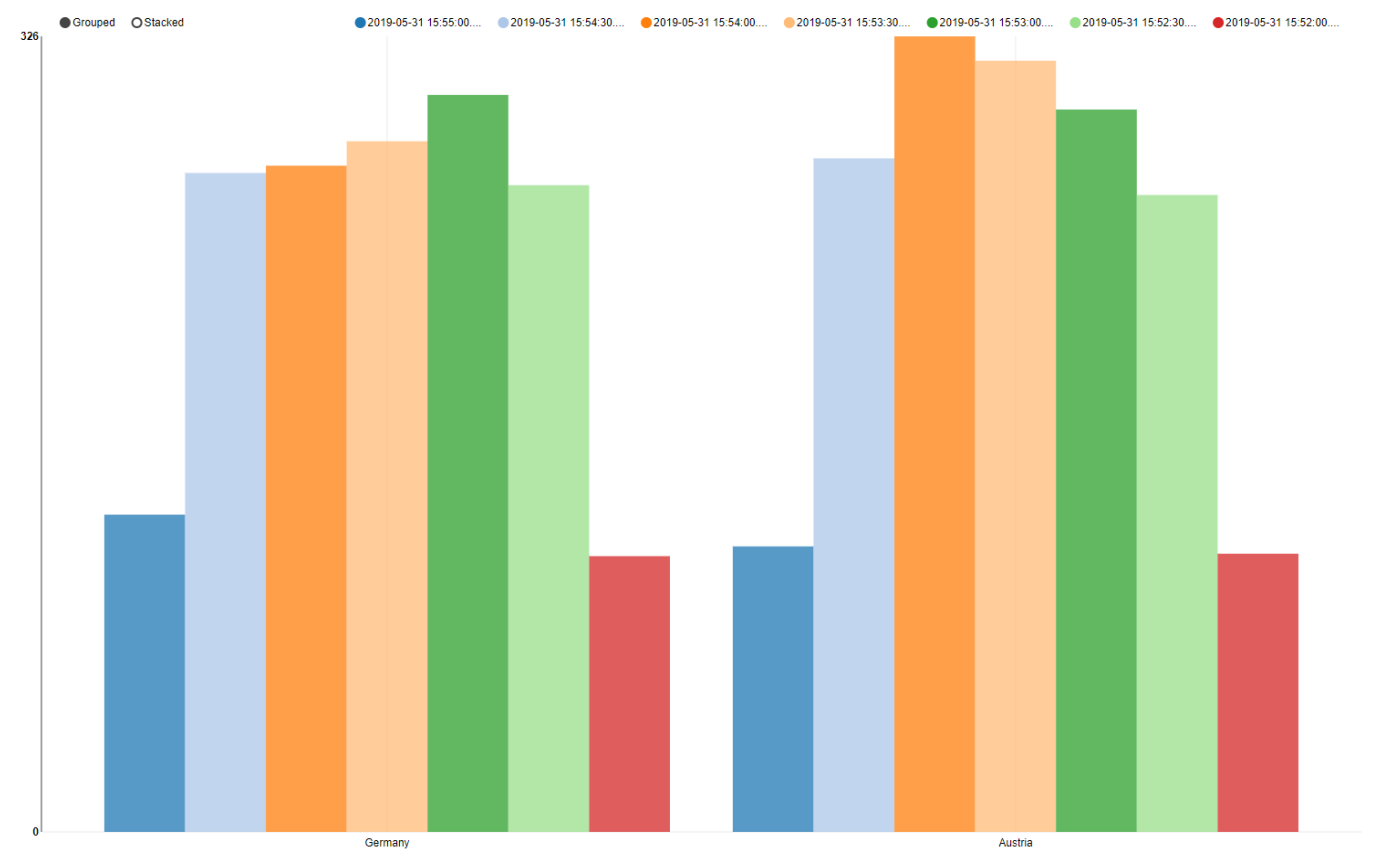
# Start running the query that prints the running counts to memory sink
writer = url_counts\
    .writeStream\
    .queryName("mi6xcCookieWindows")\
    .outputMode("complete")\
    .format("memory")

query = writer.start()
```

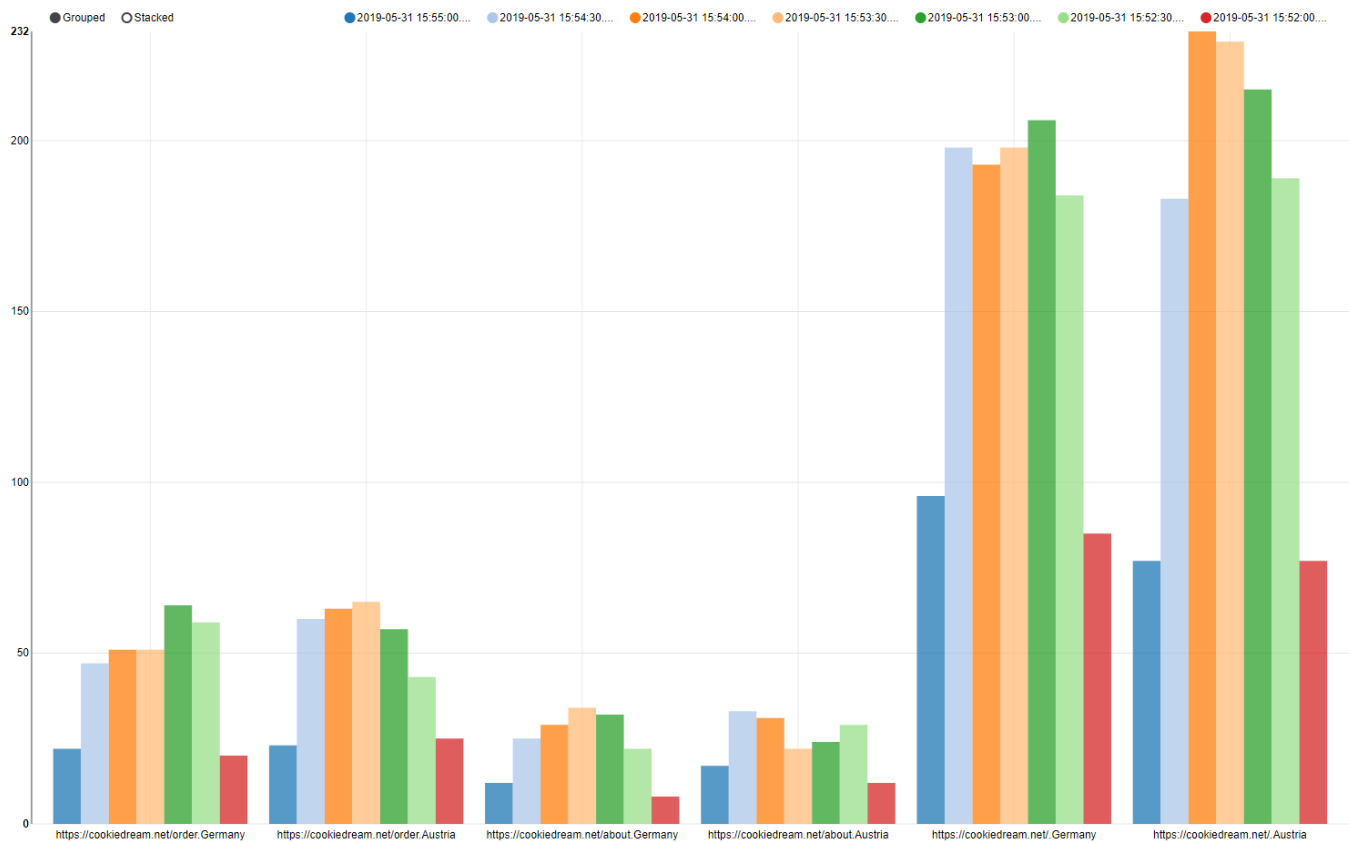
Zuerst einmal aus Interesse geschichtet, wie die Verteilung der Länder ist.



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')

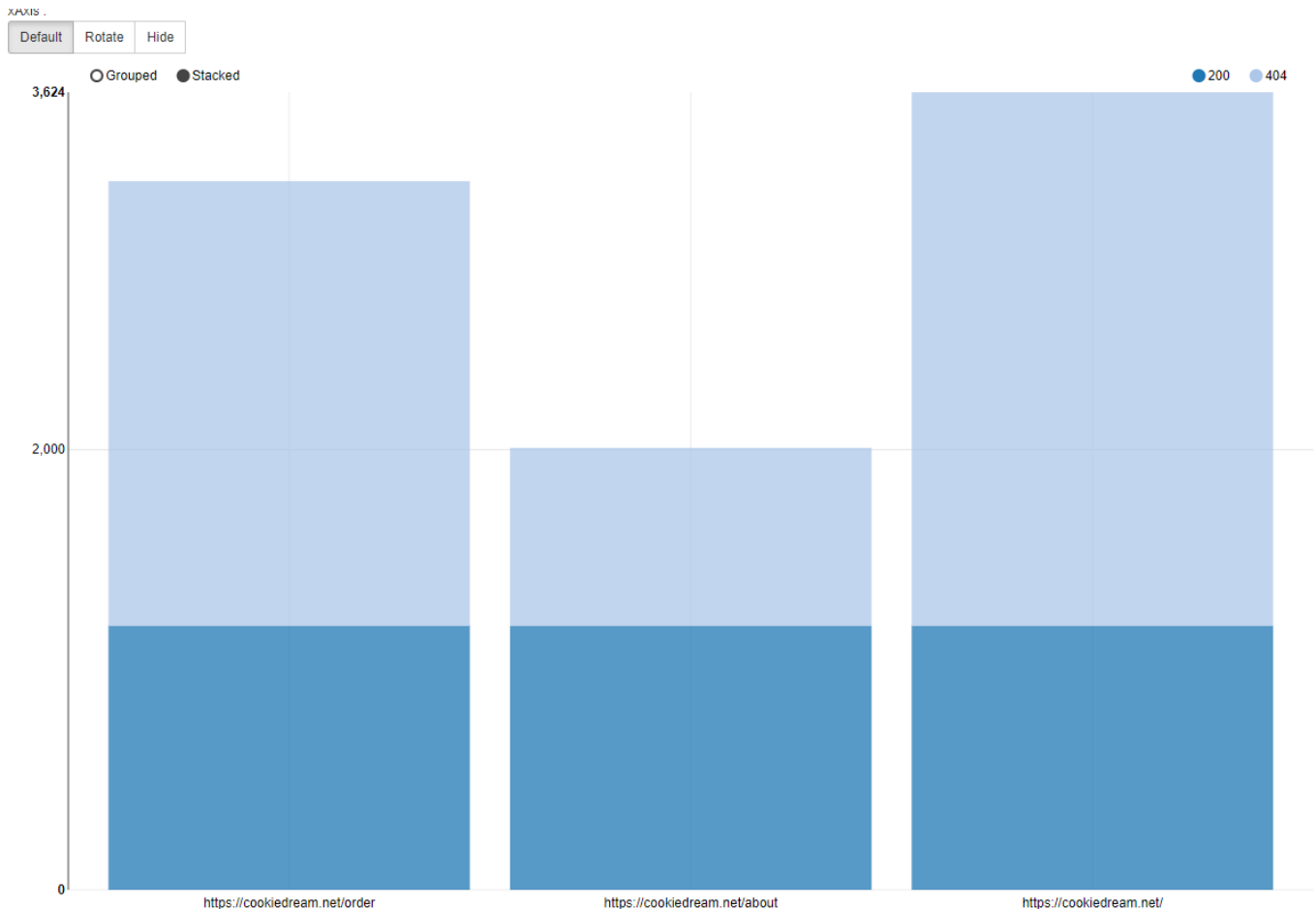
url_counts = lines\
    .select('url', 'country', 'status', 'ts')\
    .filter(lines.country == 'Germany')\
    .groupBy(
        'url',
        'status',
        window('ts', '60 seconds', '30 seconds')
    ).count()\
    .select('url', 'status', 'count', 'window.start', 'window.end')\
    .orderBy(['url', '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** generell nicht auf aggregierten Daten funktioniert und **update** nicht mit sortierung. Dies macht auch Sinn, weshalb wir die ersten beiden Aufgaben mit **complete** bearbeitet haben.

Im Falle von Late Data kann es nun passieren, dass alte Fenster von neuen Daten modifiziert werden. Dazu gibt es in Sparks **watermarks**, welche mit `.withWatermark("timestamp", "xx minutes")` definiert werden können und dann Datenpakete, die mit einer Verzögerung größer als die angegebene Zeit, direkt gedroppt werden.

Lässt man die Sortierung der Daten weg (ist für die Visualisierung eh nicht wirklich nötig), so können wir **update** als Alternative zu **complete** verwenden. Im Gegensatz zu **complete** aktualisiert **update** nur die neusten Zeilen. Dies ist mit Sliding Windows vollkommen in Ordnung, da wir keine Berechnungen auf alten Daten benötigen und wir somit die Performance verbessern können. Eine nennenswerte Veränderung ist nicht zu erwarten, außer es gäbe viele Late Data, dann würde man besonders in **a)** unterschiede sehen, da **complete** die späten Daten mit in die Berechnung einbeziehen würde, während **update** eben dies nicht tut.