

BDA, Praktikumsbericht 4

Gruppe mi6xc: Alexander Kniesz, Maximilian Neudert, Oskar Rudolf

Quellen

Das PySpark Notebook findet man [hier](#).

Aufgabe 1

a)

Zuerst verbinden wir uns auf eine Node, die als producer dienen soll, zum Beispiel **saltshore**:

```
mosh istuser@saltshore.fbi.h-da.de
```

anschließend wechseln wir in das Verzeichnis mit den vorbereiteten Scripts und starten den producer:

```
cd /opt/kafka/bin
./kafka-console-producer.sh --broker-list saltshore.fbi.h-da.de:9092 --topic bda-
gruppe3-topic
```

Analog gehen wir vor, verbinden uns auf eine andere Node, wechseln in den Ordner und starten den consumer, der sich zum Producer zunächst ohne **--form-beginning** verbindet:

```
mosh istuser@sunspear.fbi.h-da.de
cd /opt/kafka/bin
./kafka-console-consumer.sh --bootstrap-server saltshore.fbi.h-da.de:9092 --topic
bda-gruppe3-topic
```

```
[/opt/kafka/bin]
istmnneud-> ./kafka-console-producer.sh --broker-list saltshore.fbi.h-da.de:9092 --top
ic bda-gruppe3-topic
>hallo
>dies ist ein Test
>

[/opt/kafka/bin]
istmnneud-> ./kafka-console-consumer.sh --bootstrap-server saltshore.fbi.h-da.de:9092
--topic bda-gruppe3-topic
hallo
dies ist ein Test
```

Wenn wir beim Producer nun Nachrichten schreiben, dann werden diese mit kurzer Verzögerung vom Consumer empfangen und dort auf der Console ausgegeben. Fügen wir nun zusätzlich den Parameter `--form-beginning` hinzu, so erhalten wir erwartungsgemäß alle Nachrichten, die bisher auf dem angegebenen Topic gesendet wurden. Da wohl ein paar Spaßvögel mit Scripts das Topic geflutet haben dauert das sogar eine Weile auszugeben.

```
[/opt/kafka/bin]
istmnneud-> ./kafka-console-producer.sh --broker-list saltshore.fbi.h-da.de:9092 --topic bda-gruppe3-topic
>hallo
>dies ist ein Test
>neue Nachricht
>

Turns reality (turn reality)
They already know they can't fuck with Iggy
Switchin' up the game (switch it up, switch it up now)
It's Iggy Iggs!
Is Iggy the ziggy-iggy the baddest of 'em all?
Turns reality (turn reality)
They already know they can't fuck with Iggy
Switchin' up the game (switch it up, switch it up now)
It's Iggy Iggs!
Is Iggy the ziggy-iggy the baddest of 'em all?
Turns reality (turn reality)
They already know they can't fuck with Iggy
Switchin' up the game (switch it up, switch it up now)
It's Iggy Iggs!
Is Iggy the ziggy-iggy the baddest of 'em all?
Turns reality (turn reality)
They already know they can't fuck with Iggy
Switchin' up the game (switch it up, switch it up now)
dies ist ein Test
neue Nachricht

```

b)

Schauen wir uns die Spalten der Kafka Ausgabe per SQL an



so sehen wir, dass wir neben `key` und `value` auch eine Reihe an Metadaten wie `timestamp`, `topic` und `partition` erhalten.

Starten wir die WordCount Query

```
%pyspark
#aufgabe 1b): Deklaration des Kafka-Consumer-Streams und Start der query

from pyspark.sql.functions import explode
from pyspark.sql.functions import split

# read text file
lines = spark \
    .readStream \
    .format("kafka") \
    .option("kafka.bootstrap.servers", "141.100.62.88:9092") \
    .option("subscribe", "bda-gruppe3-topic") \
    .option("startingOffsets", "earliest") \
    .option("kafkaConsumer.pollTimeoutMs", "8192") \
    .load()

# cast value object to string
lines = lines\
    .withColumn("value", lines.value.cast('string'))\
    .select('value')

# split lines into words
words = lines\
    .select(explode(split(lines.value, " ")))\
    .alias("word")

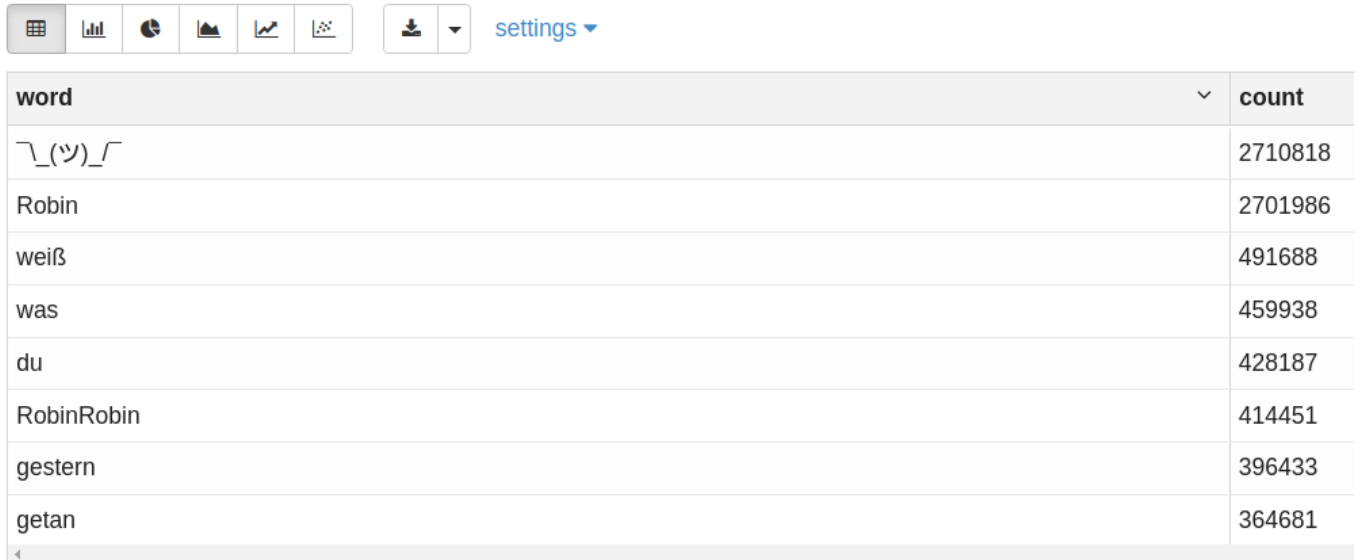
# count words
count_words = words\
    .groupBy("word").count()\
    .orderBy("count", ascending=False)

# Start running the query that prints the running counts to memory sink
writer = count_words \
    .writeStream \
    .queryName("mi6xc_kafkawords") \
    .outputMode("complete") \
    .format("memory")

query = writer.start()
```

so erhalten wir folgendes Ergebnis:

```
%sql
select * from mi6xc_kafkawords
```



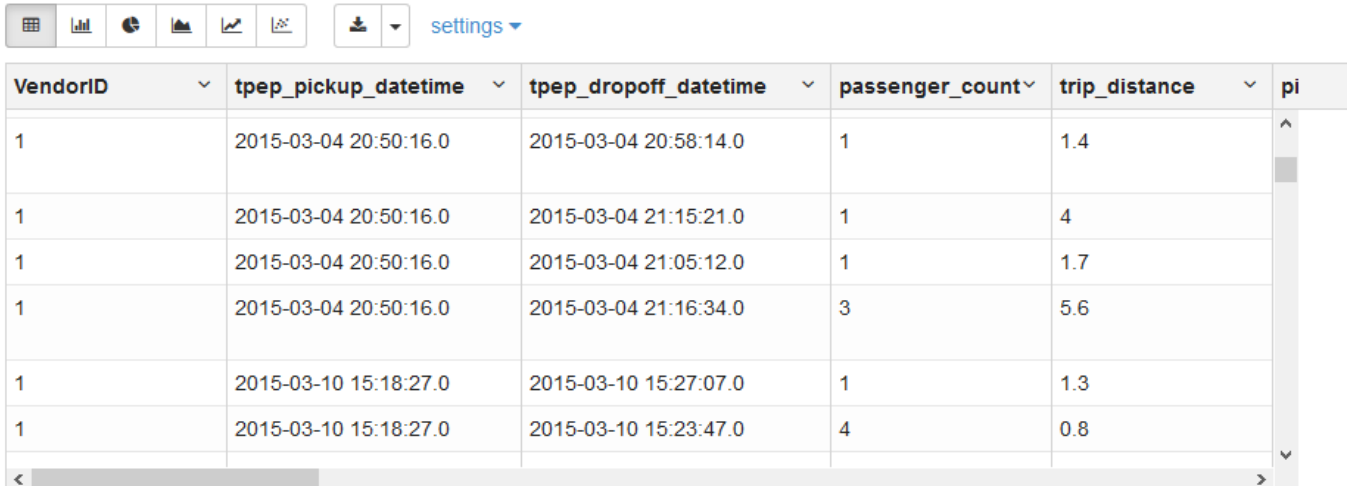
word	count
(ツ)/	2710818
Robin	2701986
weiß	491688
was	459938
du	428187
RobinRobin	414451
gestern	396433
getan	364681

Aufgabe 2

a)

Zunächst haben wir die Daten gesichtet:

```
%sql
/* Betrachten der Daten */
select * from yellow_tripdata limit 100
```



VendorID	tpep_pickup_datetime	tpep_dropoff_datetime	passenger_count	trip_distance	pi
1	2015-03-04 20:50:16.0	2015-03-04 20:58:14.0	1	1.4	
1	2015-03-04 20:50:16.0	2015-03-04 21:15:21.0	1	4	
1	2015-03-04 20:50:16.0	2015-03-04 21:05:12.0	1	1.7	
1	2015-03-04 20:50:16.0	2015-03-04 21:16:34.0	3	5.6	
1	2015-03-10 15:18:27.0	2015-03-10 15:27:07.0	1	1.3	
1	2015-03-10 15:18:27.0	2015-03-10 15:23:47.0	4	0.8	

Anschließend haben wir die Daten als SparkDataFrame importiert und mit Hilfe der `month` und `dayofmonth`-Funktionen den jeweiligen Monat und Tag als separate Spalten hinzugefügt. Danach konnten wir nach dem 7. Monat filtern und die Anzahl der Zeilen gruppiert nach Monatstagen zählen:

```
%pyspark
from pyspark.sql.functions import month,dayofmonth, col

# Einlesen der benötigten Daten
df = spark.sql("SELECT tpep_pickup_datetime FROM yellow_tripdata")

df.withColumn("Month",month("tpep_pickup_datetime")) \ # Monat als neue Spalte
.withColumn("DayOfMonth",dayofmonth("tpep_pickup_datetime")) \ # Tag des Monats als neue Spalte
.filter(col("Month")==7) \ # Alle Julitage herausfiltern
.groupBy("DayOfMonth") \ # Daten nach Tag gruppieren
.count() \ # Gruppierte Daten zählen
.write.mode("overwrite").saveAsTable("mi6xc_julidata") # Neue Tabelle persistieren
```

b)

Nach Filtern & Zählen der Daten überprüfen wir, ob soweit alles geklappt hat:

```
%pyspark
# Erfolgskontrolle
spark.sql("SELECT * FROM mi6xc_julidata").show()
```

DayOfMonth	count
26	349537
27	348629
6	315526
16	400140
9	405656
17	365045
31	397815
28	384234
12	364211
22	392078
1	390084
13	350162
3	304176
20	374211

c)

Zunächst prüfen wir wieder über ssh, ob wir als **Subscriber** die Taxidaten einsehen können. Dazu verbinden wir uns auf eine der Nodes führen

```
cd /opt/kafka/bin
./kafka-console-consumer.sh \
--bootstrap-server 141.100.62.85:9092 \
--topic yellow_tripdata_2015_12 \
--from-beginning
```

aus und erhalten:

```

1,0.5,0,0.3,7.8
2,2015-12-01 18:47:25,2015-12-01 19:09:00,1,2.87,-73.979072570800781,40.76025390625,1,N,-73.949539184570312,40.784782409667969,1,15,1,
0.5,2.5,0,0.3,19.3
2,2015-12-01 18:47:28,2015-12-01 19:04:59,2,1.93,-74.010223388671875,40.705558776855469,1,N,-74.000076293945312,40.728679656982422,1,1
2,1,0.5,2.76,0,0.3,16.56
2,2015-12-01 18:47:29,2015-12-01 19:35:46,2,10.72,-73.982902526855469,40.761100769042969,1,N,-73.86456298828125,40.77020263671875,1,38
.5,1,0.5,8.06,0,0.3,48.36
2,2015-12-01 18:47:30,2015-12-01 19:01:58,1,.75,-73.983161926269531,40.767959594726562,1,N,-73.984466552734375,40.760280609130859,1,10
,1,0.5,2,0,0.3,13.8
1,2015-12-01 18:47:32,2015-12-01 18:57:57,1,1.00,-74.004074096679688,40.747951507568359,1,N,-74.008018493652344,40.739784240722656,1,8
,1,0.5,1.96,0,0.3,11.76
2,2015-12-01 18:47:33,2015-12-01 18:59:04,5,1.32,-73.952033996582031,40.769363403320313,1,N,-73.968307495117188,40.762577056884766,2,9
,1,0.5,0,0,0.3,10.8
2,2015-12-01 18:47:34,2015-12-01 19:06:19,1,1.42,-73.9913330078125,40.7322998046875,1,N,-73.980171203613281,40.748813629150391,2,12,1,
0.5,0,0,0.3,13.8
2,2015-12-01 18:47:35,2015-12-01 18:57:55,1,1.03,-73.9635009765625,40.777400970458984,1,N,-73.952232360839844,40.782299041748047,1,8,1
,0.5,1.96,0,0.3,11.76
Processed a total of 179000 messages

```

Das hat soweit geklappt. Jetzt abonnieren wir diesen Stream mit Spark und folgenden Optionen:

```

# read text file
df = spark \
    .readStream \
    .format("kafka") \
    .option("kafka.bootstrap.servers", "141.100.62.88:9092") \
    .option("subscribe", "yellow_tripdata_2015_12") \
    .option("startingOffsets", "earliest") \
    .option("kafkaConsumer.pollTimeoutMs", "8192") \
    .option("maxOffsetsPerTrigger", 1000000) \
    .load()

```

Da die Datenpunkte hier zeilenweise als Strings in den Stream fließen, müssen wir die Zeilen, mit einem `split`-Befehl aufteilen. Da uns zunächst "nur" die Timestamps interessieren, wählen wir hier nur das zweite Element aus dem gesplitteden Datenpunkt. Anschließend nutzen wir aus, dass die Funktion `dayofmonth` auch String-Datentypen als input akzeptiert und wir mittels dieser Funktion wieder den Monatstag extrahieren können. Damit die Wochentage übereinstimmen, wollen wir im Folgenden den Mittwoch, 1. Juli (**Data at Rest**) mit Mittwoch, dem 2. Dezember (**Data in Motion**) vergleichen. Dafür generieren wir eine zusätzliche Spalte mit versetztem "Day of Month".

```

# extract datetimes
datetime_df = df \
    .withColumn('datetimes', split(df.value, ",").getItem(1)) # 1. Item =
tpep_pickup_datetime

datetime_df = datetime_df \
    .withColumn('DayOfMonth', dayofmonth(datetime_df.datetimes))

datetime_df = datetime_df \
    .withColumn("DayToJoin", datetime_df.DayOfMonth + 1)

```

Jetzt können wir die Daten einfach wieder nach Monatstag (analog zu 2b)) und mit einem Window über die timestamps (mit size = 24 Stunden) aggregieren und zählen:

```

# count
aggregated = datetime_df \

```

```

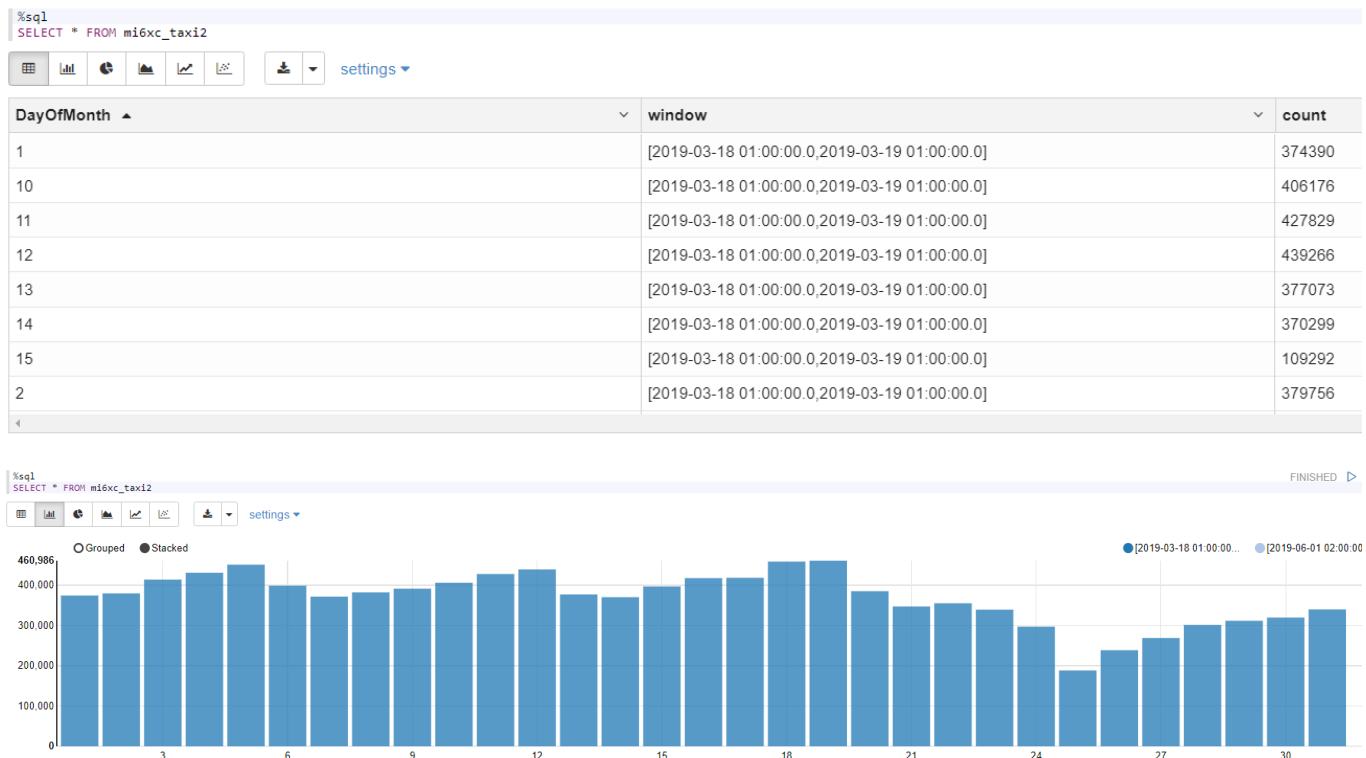
.groupBy("DayOfMonth", window(datetime_df.timestamp, "24 hours"))\
.count() \
.orderBy("count", ascending=False)

writer2 = aggregated \
.writeStream \
.queryName("mi6xc_taxi2") \
.outputMode("complete") \
.format("memory")

query2 = writer2.start()

```

Damit erhalten wir zum als Output:



d)

Wir lesen für die Join-Operation zunächst die Juli-Daten wieder ein und führen dann mithilfe der entsprechenden SpakDataFrame-Methode einen InnerJoin durch:

```

# Sommerdaten
juli_df = spark.sql("select * from mi6xc_julidata")

# Join Operation
all_data = datetime_df.join(juli_df, datetime_df.DayToJoin == juli_df.DayOfMonth)

```

Die Aggregation funktioniert analog zu 2c). Hier müssen wir lediglich noch beachten, dass aufgrund des joins zwei **DayOfMonth** Spalten existieren und wir deshalb die eindeutige Spalte **DayToJoin** als Gruppierungsvariable verwenden. Zusätzlich müssen wir die **count** Spalten entsprechend umbenennen.

```
%pyspark
all_data.show
```

FINISHED ▶ 🔍 ⚙️

```
<bound method DataFrame.show of DataFrame[key: binary, value: binary, topic: string, partition: int, offset: bigint, timestamp: timestamp, timestampType: int, datetimes: string, DayOfMonth: int, DayToJoin: int, DayOfMonth: int, count: bigint]>
```

Took 0 sec. Last updated by istalknie at June 13 2019, 9:00:41 PM.

```
# Winterdaten
datetime_df = df\
    .withColumn('datetimes', split(df.value, ",").getItem(1))

datetime_df = datetime_df\
    .withColumn('DayOfMonth', dayofmonth(datetime_df.datetimes))

datetime_df = datetime_df \
    .withColumn("DayToJoin", datetime_df.DayOfMonth + 1)

df_count_dez = datetime_df \
    .groupBy("DayToJoin", window(datetime_df.timestamp, "24 hours"))\
    .count() \
    .orderBy("count", ascending=False) \
    .selectExpr("count as count_dez", "DayToJoin as day")

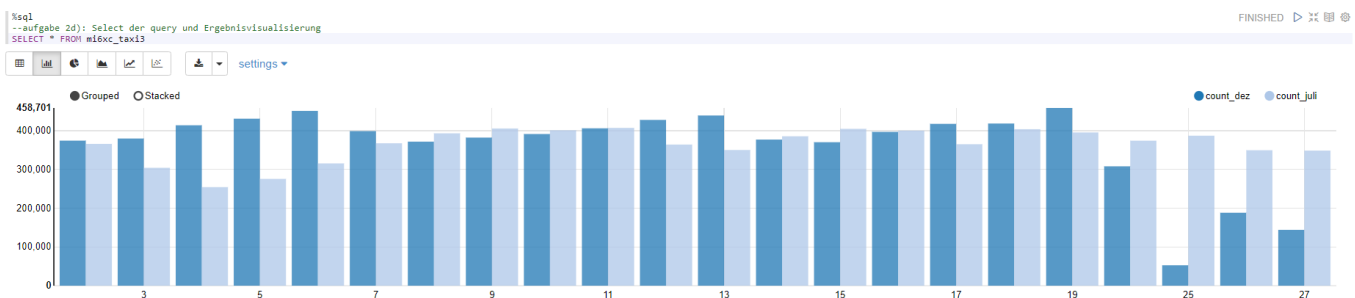
# Sommerdaten
df_count_juli = spark.sql("select * from mi6xc_julidata") \
    .selectExpr("count as count_juli", "DayOfMonth as day")

# Join Operation
df_count_all = df_count_dez.join(df_count_juli, df_count_dez.day ==
df_count_juli.day)

writer3 = df_count_all \
    .writeStream \
    .queryName("mi6xc_taxi3") \
    .outputMode("complete") \
    .format("memory")

query3 = writer3.start()
```

und visualisieren das Ergebnis:



Spontan liegt die Vermutung nahe, dass an wichtigen Feiertagen weniger Taxi gefahren wird. Denn sowohl der 4. Juli als auch Weihnachten sind wichtige Feiertage der USA und dort ist jeweils ein starker Einbruch

verglichen zu beobachten.

e)

Das Vorgehen ist äquivalent zu d), allerdings müssen wir die entsprechenden Variablen von Interesse aus dem Input-Stream extrahieren und als Spalte in unseren SparkDataFrame hinzufügen:

```
datetime_df = df\
    .withColumn('trip_distance', split(df.value, ",").getItem(4)) # 4. Item =
    Trip_Distance
```

Anschließend können wir nach dieser Variablen gruppieren und alle möglichen gewünschten Aggregationsfunktionen darauf anwenden (wir haben `mean` angewendet)

- `avg`
- `count`
- `max`
- `mean`
- `min`
- `sum`

```
%pyspark
# Aufgabe 2e)

from pyspark.sql.functions import split
from pyspark.sql.functions import window
from pyspark.sql.functions import month, dayofmonth, col

df = spark \
    .readStream \
    .format("kafka") \
    .option("kafka.bootstrap.servers", "141.100.62.88:9092") \
    .option("subscribe", "yellow_tripdata_2015_12") \
    .option("startingOffsets", "earliest") \
    .option("kafkaConsumer.pollTimeoutMs", "8192") \
    .option("maxOffsetsPerTrigger", 1000000) \
    .load()

# Winderdaten
df_dez = df\
    .withColumn('datetimes', split(df.value, ",").getItem(1)) \
    .withColumn('trip_distance', split(df.value, ",").getItem(4))
df_dez = df_dez\
    .withColumn('DayOfMonth', dayofmonth(df_dez.datetimes))
df_dez = df_dez\
    .withColumn("DayToJoin", df_dez.DayOfMonth + 1)

df_dist_dez = df_dez \
    .groupBy("DayToJoin", window(df_dez.timestamp, "24 hours")) \
```

```

    .agg({'trip_distance': 'mean'}) \
    .withColumnRenamed("avg(trip_distance)", "mean") \
    .selectExpr("mean as avg_dist_dez", "DayToJoin as day")

# Sommerdaten
df_juli = spark.sql("SELECT tpep_pickup_datetime as datetimes, trip_distance FROM yellow_tripdata")
df_juli = df_juli \
    .withColumn("Month", month("datetimes")) \
    .filter(col("Month")==7)
df_juli = df_juli \
    .withColumn('DayOfMonth', dayofmonth(df_juli.datetimes))

df_dist_juli = df_juli \
    .groupBy("DayOfMonth") \
    .agg({'trip_distance': 'mean'}) \
    .withColumnRenamed("avg(trip_distance)", "mean") \
    .selectExpr("mean as avg_dist_juli", "DayOfMonth as day")

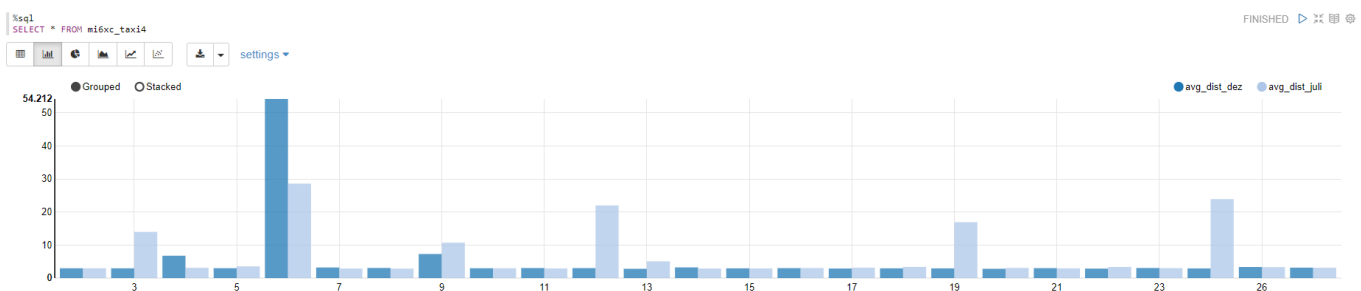
# Join Operation
df_dist_all = df_dist_dez.join(df_dist_juli, df_dist_dez.day == df_dist_juli.day)

writer4 = df_dist_all \
    .writeStream \
    .queryName("mi6xc_taxi4") \
    .outputMode("complete") \
    .format("memory")

query4 = writer4.start()

```

Als Output bekommen wir dann:



Neben Tagen, die generell etwas höheres Aufkommen haben, fallen Ausreißer auf. Da die Daten nicht bereinigt sind, könnte dies auf Rauschen zurückführbar sein. Schauen wir in die Daten

```
SELECT * FROM yellow_tripdata where trip_distance > 200
```

So finden wir einige Ausreißer und ein paar absurd hohe Werte, was so viel bedeutet wie, dass in der Distanz einiges an Rauschen enthalten ist. Dies könnten defekte Taxiuhren sein, die zu spät oder gar nicht gestoppt wurden oder einfach einen Fehler bei der Datenübertragen hatten. Es ist durchaus anzunehmen, dass es

Kunden gibt, die auch weite Strecken mit dem Taxi nehmen, aber schaut man sich die Daten an, so kann man sagen, dass eine Fahrtstrecke von über 100km äußerst unwahrscheinlich ist. Also filtern wir diese heraus.

```
df_dez = df_dez \
    .withColumn("DayToJoin", df_dez.DayOfMonth + 1) \
    .filter(col("trip_distance") <= 100)

df_juli = df_juli \
    .withColumn('DayOfMonth', dayofmonth(df_juli.datetimes)) \
    .filter(col("trip_distance") <= 100)
```

Damit erhalten wir als neuen Output:

