# **ЗАДАНИЕ**

Осмыслить происходящее на практике. Чтобы понять, советую смотреть и читать документацию, там более подробно написано про каждую функцию, про каждый режим.  
Запустить foreachBatch.  
Запустить агрегацию по временному окну в разных режимах.

Сджойнить стрим со статикой.  
Сджойнить стрим со стримом по желанию.

# **foreachBatch**

Запускаем mobaXterm

Создаем сессию с подключеним к worker-0

Выполняем команды:

*export SPARK\_KAFKA\_VERSION=0.10*

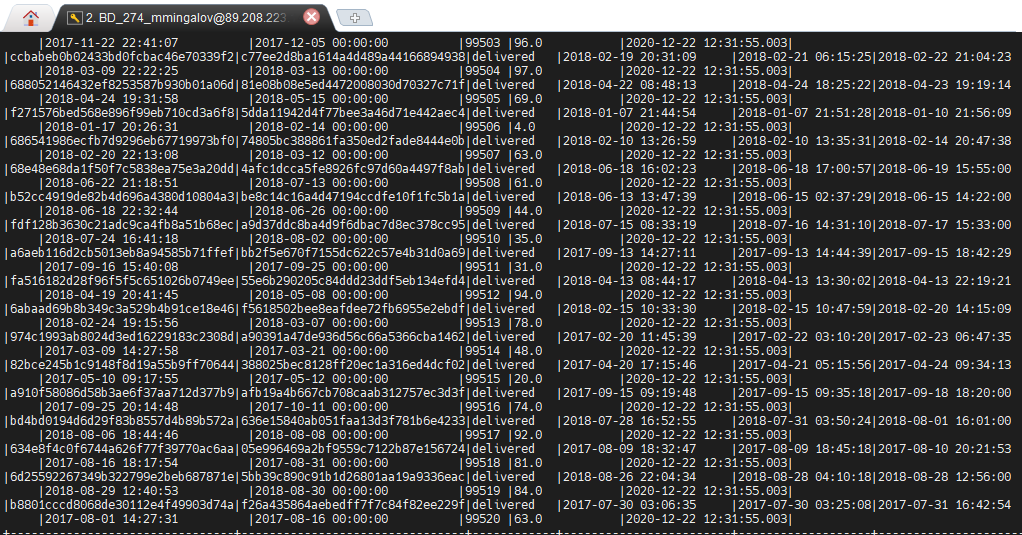
*/spark2.4/bin/pyspark --packages org.apache.spark:spark-sql-kafka-0-10\_2.11:2.4.5 --driver-memory 512m --num-executors 1 --executor-memory 512m --master local[1]*

Здесь делаем ограничение драйвера по памяти в 512Мб

**from** pyspark.sql **import** SparkSession, DataFrame  
**from** pyspark.sql **import** functions **as** F  
**from** pyspark.sql.types **import** StructType, StringType, IntegerType, TimestampType  
  
kafka\_brokers = **"bigdataanalytics-worker-0.novalocal:6667"**raw\_orders = spark.readStream. \  
 format(**"kafka"**). \  
 option(**"kafka.bootstrap.servers"**, kafka\_brokers). \  
 option(**"subscribe"**, **"orders\_json"**). \  
 option(**"maxOffsetsPerTrigger"**, **"20"**). \  
 option(**"startingOffsets"**, **"earliest"**). \  
 load()  
  
*##разбираем value*schema = StructType() \  
 .add(**"order\_id"**, StringType()) \  
 .add(**"customer\_id"**, StringType()) \  
 .add(**"order\_status"**, StringType()) \  
 .add(**"order\_purchase\_timestamp"**, StringType()) \  
 .add(**"order\_approved\_at"**, StringType()) \  
 .add(**"order\_delivered\_carrier\_date"**, StringType()) \  
 .add(**"order\_delivered\_customer\_date"**, StringType()) \  
 .add(**"order\_estimated\_delivery\_date"**, StringType())  
  
parsed\_orders = raw\_orders \  
 .select(F.from\_json(F.col(**"value"**).cast(**"String"**), schema).alias(**"value"**), **"offset"**) \  
 .select(**"value.\*"**, **"offset"**)  
  
extended\_orders = parsed\_orders \  
 .withColumn(**"my\_extra\_column"**, F.round( F.rand() \* 100 ) ) \  
 .withColumn(**"my\_current\_time"**, F.current\_timestamp())

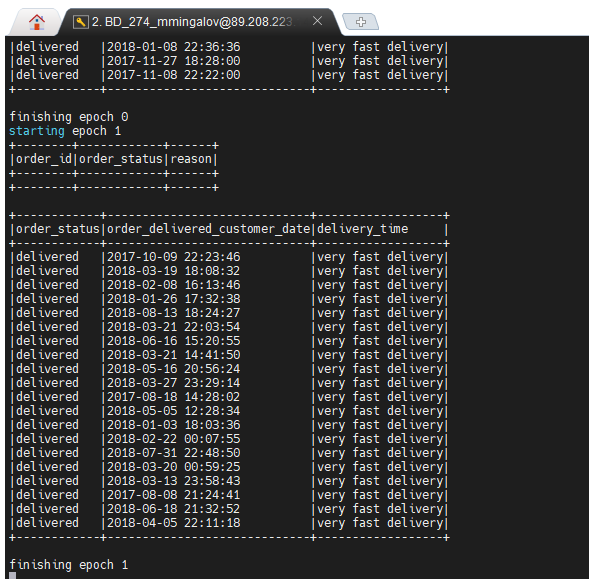
выводим на консоль посмотреть, что будет происходить:

**def** console\_output(df, freq):  
 **return** df.writeStream \  
 .format(**"console"**) \  
 .trigger(processingTime=**'%s seconds'** % freq ) \  
 .options(truncate=**False**) \  
 .start()  
  
out = console\_output(extended\_orders, 5)  
out.stop()



*#FOREACH BATCH SINK***def** foreach\_batch\_sink(df, freq):  
 **return** df \  
 .writeStream \  
 .foreachBatch(foreach\_batch\_function) \  
 .trigger(processingTime=**'%s seconds'** % freq ) \  
 .start()  
  
**def** foreach\_batch\_function(df, epoch\_id):  
 print(**"starting epoch "** + str(epoch\_id) )  
 df.persist() *#сохранили в оперативку* df.filter(F.col(**"order\_status"**)!=**"delivered"**). \  
 select(**"order\_id"**, **"order\_status"**). \  
 withColumn(**"reason"**, F.lit(**"too slow"**)). \  
 show(truncate=**False**)  
 df.filter(F.col(**"order\_status"**)==**"delivered"**). \  
 select(**"order\_status"**, **"order\_delivered\_customer\_date"**). \  
 withColumn(**"delivery\_time"**, F.lit(**"very fast delivery"**)). \  
 show(truncate=**False**)  
 df.unpersist()  
 print(**"finishing epoch "** + str(epoch\_id))  
*#важно, что здесь появляется статический df по размеру равный микробатчу  
#это позволяет нам выводить его на консоль через show например и вообще использовать весь функционал и логику статического Spark внутри стриминга*

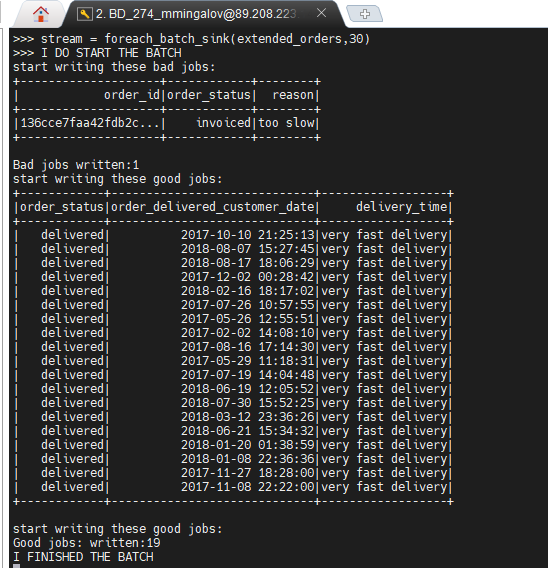
stream = foreach\_batch\_sink(extended\_orders,20)  
stream.stop()



Каждые 20 сек происходит срабатывание функции foreachBatch

**ВИДОИЗМЕНИМ ЛОГИКУ ФУНКИИ**

*#with persist***def** foreach\_batch\_function(df, epoch\_id):  
 print(**"I DO START THE BATCH"**)  
 df.persist() *#сохрянем в память для многократного фильтра  
 #получаем DF хороших доставок* good\_job = df.filter(F.col(**"order\_status"**)==**"delivered"**). \  
 select(**"order\_status"**, **"order\_delivered\_customer\_date"**). \  
 withColumn(**"delivery\_time"**, F.lit(**"very fast delivery"**))  
 *#получаем DF плохих доставок* bad\_job = df.filter(F.col(**"order\_status"**)!=**"delivered"**). \  
 select(**"order\_id"**, **"order\_status"**). \  
 withColumn(**"reason"**, F.lit(**"too slow"**))  
 *#сохраняем DF плохих доставок для многократных action* bad\_job.persist()  
 print(**"start writing these bad jobs:"**)  
 bad\_job.show()  
 bad\_job.write.mode(**"append"**).parquet(**"my\_bad\_job"**) *#записываем плохие доставки в папку на hdfs* print(**"Bad jobs written:"** + str(bad\_job.count()))  
 bad\_job.unpersist() *#удаляем из памяти плохие доставки* good\_job.persist() *#сохраняем DF хороших доставок для многократных action* print(**"start writing these good jobs:"**)  
 good\_job.show()  
 df.unpersist() *#удаляем из памяти полный набор* print(**"start writing these good jobs:"**)  
 good\_job.write.mode(**"append"**).parquet(**"my\_good\_job"**) *#записываем хорошие доставки в папку на hdfs* print(**"Good jobs: written:"** + str(good\_job.count()))  
 good\_job.unpersist() *#удаляем из памяти хорошие доставки* print(**"I FINISHED THE BATCH"**)

ЗАПУСКАЕМ:  
  
stream = foreach\_batch\_sink(extended\_orders,30)  


stream.stop()

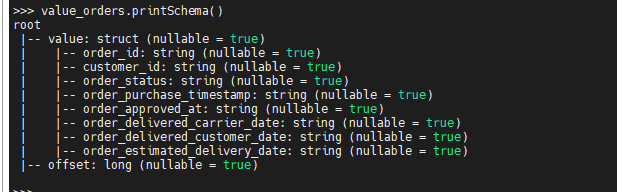
# **Watermark**

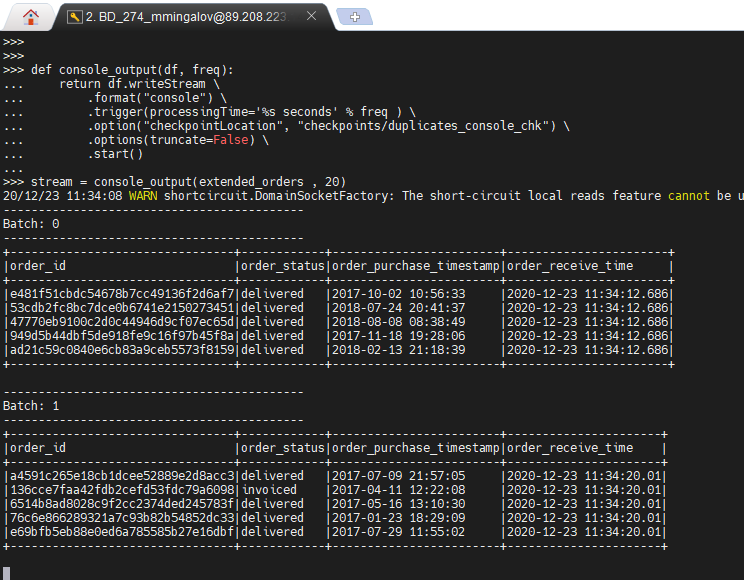
Запускаем mobaXterm

Создаем сессию с подключеним к worker-0

Выполняем команды:

*export PYSPARK\_PYTHON=python3  
export SPARK\_KAFKA\_VERSION=0.10  
/spark2.4/bin/pyspark --packages org.apache.spark:spark-sql-kafka-0-10\_2.11:2.4.5 --driver-memory 512m --master local[1]***from** pyspark.sql **import** SparkSession, DataFrame  
**from** pyspark.sql **import** functions **as** F  
**from** pyspark.sql.types **import** StructType, StringType, IntegerType, TimestampType  
  
#spark = SparkSession.builder.appName(**"mmingalov\_spark"**).getOrCreate()  
  
kafka\_brokers = **"bigdataanalytics-worker-0.novalocal:6667"**raw\_orders = spark.readStream. \  
 format(**"kafka"**). \  
 option(**"kafka.bootstrap.servers"**, kafka\_brokers). \  
 option(**"subscribe"**, **"orders\_json"**). \  
 option(**"maxOffsetsPerTrigger"**, **"5"**). \  
 option(**"startingOffsets"**, **"earliest"**). \  
 load()  
  
*##разбираем value*schema = StructType() \  
 .add(**"order\_id"**, StringType()) \  
 .add(**"customer\_id"**, StringType()) \  
 .add(**"order\_status"**, StringType()) \  
 .add(**"order\_purchase\_timestamp"**, StringType()) \  
 .add(**"order\_approved\_at"**, StringType()) \  
 .add(**"order\_delivered\_carrier\_date"**, StringType()) \  
 .add(**"order\_delivered\_customer\_date"**, StringType()) \  
 .add(**"order\_estimated\_delivery\_date"**, StringType())  
  
value\_orders = raw\_orders \  
 .select(F.from\_json(F.col(**"value"**).cast(**"String"**), schema).alias(**"value"**), **"offset"**)  
  
value\_orders.printSchema()

  
  
extended\_orders = value\_orders.select(**"value.order\_id"**, **"value.order\_status"**, **"value.order\_purchase\_timestamp"** ) \  
 .withColumn(**"order\_receive\_time"**, F.current\_timestamp())  
  
  
*#чекпоинт нужен для переноса кэша из оперативки в hdfs***def** console\_output(df, freq):  
 **return** df.writeStream \  
 .format(**"console"**) \  
 .trigger(processingTime=**'%s seconds'** % freq ) \  
 .option(**"checkpointLocation"**, **"checkpoints/duplicates\_console\_chk"**) \  
 .options(truncate=**False**) \  
 .start()  
  
stream = console\_output(extended\_orders , 20)



stream.stop()  
  
*##WATERMARK должна очищать чекпоинт, но не гарантирует удаление точно вовремя. Возможно, удалит позже, но не раньше*waterwarked\_orders = extended\_orders.withWatermark(**"order\_receive\_time"**, **"1 minute"**)  
  
*#получаем дедупликацию на каждый "тик"  
#бизнес-логика вымышленная для наглядности примера: 1 заказ с уникальным статусом (хотя реальная ситуация -- уникальный order\_id + order\_status)*deduplicated\_orders = waterwarked\_orders.drop\_duplicates([**"order\_status"**, **"order\_receive\_time"**])

**важно:**

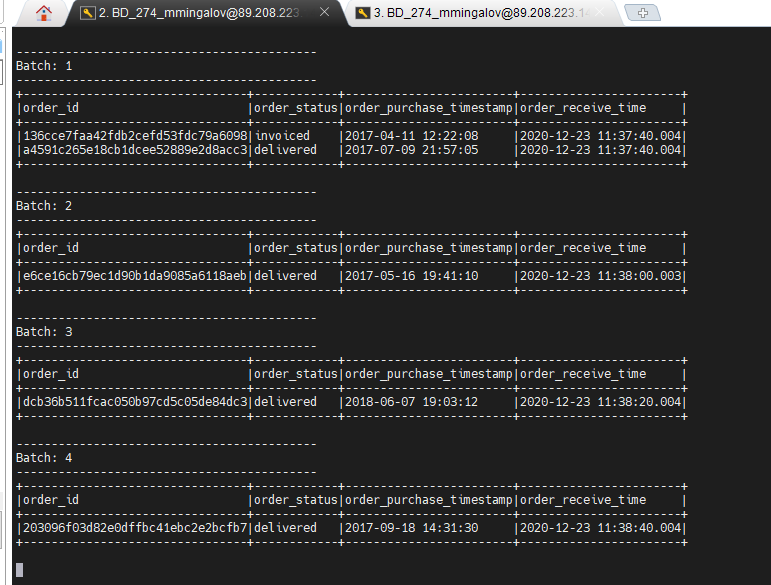
**сперва удалим директорию checkpoint**

в отдельной вкладке mobaXterm выполним команду:

**hdfs dfs -rmr checkpoints/duplicates\_console\_chk**

далее запускаем stream:

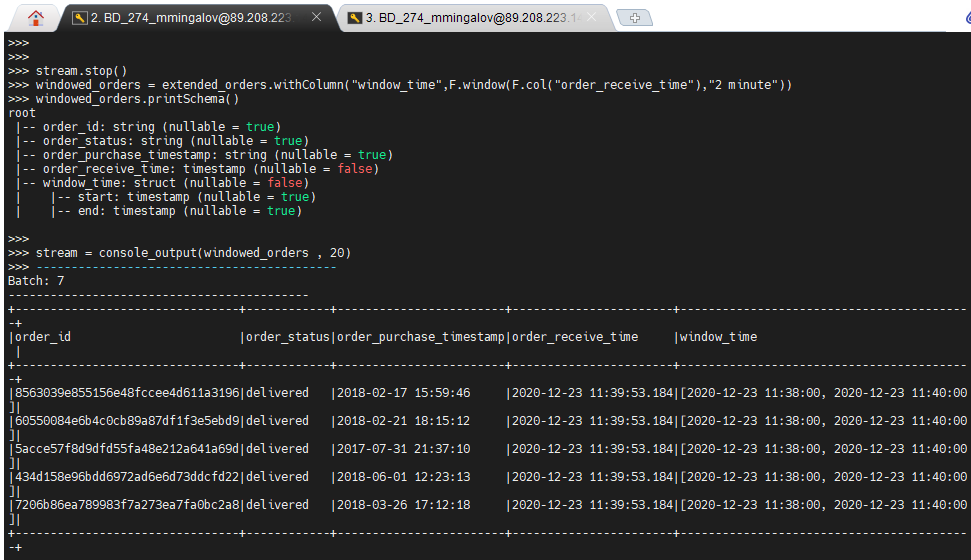
stream = console\_output(deduplicated\_orders , 20)



stream.stop()

# **WINDOW**

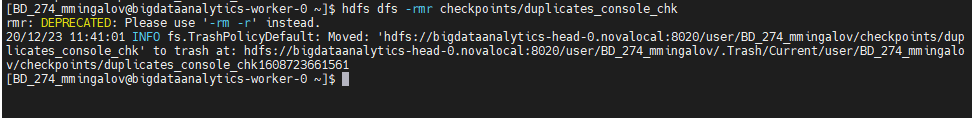
*#WINDOW - получаем дедупликацию в промежуток времени  
#создаем временное окно, объединяющее несколько микро-батчей (в зависимости от размера окна конечно же)*windowed\_orders = extended\_orders.withColumn(**"window\_time"**,F.window(F.col(**"order\_receive\_time"**),**"2 minute"**))  
windowed\_orders.printSchema()  
  
stream = console\_output(windowed\_orders , 20)



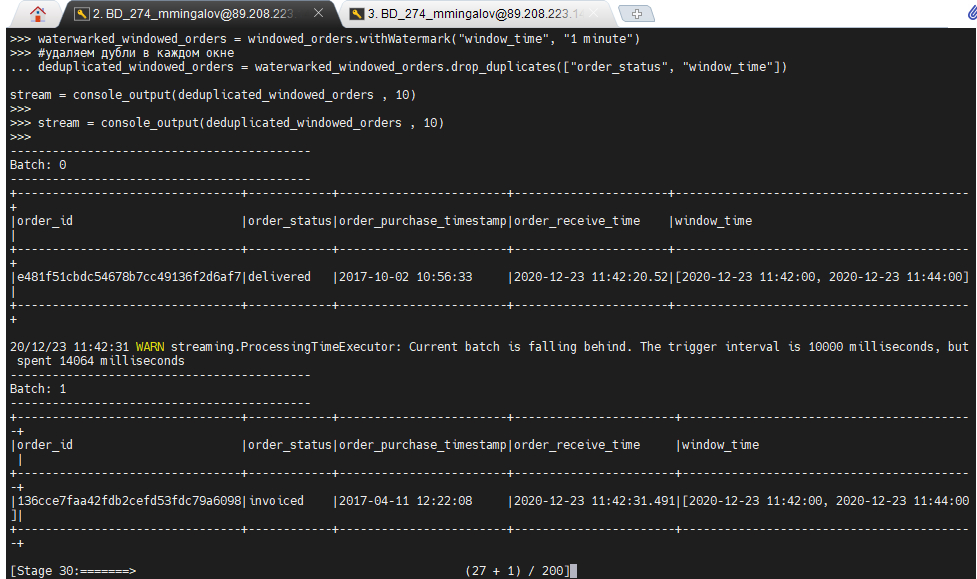
stream.stop()  
*#устанавливаем вотермарк для очистки чекпоинта*

**снова удалим директорию checkpoint**

в отдельной вкладке mobaXterm выполним команду:



waterwarked\_windowed\_orders = windowed\_orders.withWatermark(**"window\_time"**, **"1 minute"**)  
*#удаляем дубли в каждом окне*deduplicated\_windowed\_orders = waterwarked\_windowed\_orders.drop\_duplicates([**"order\_status"**, **"window\_time"**])  
  
stream = console\_output(deduplicated\_windowed\_orders , 10)



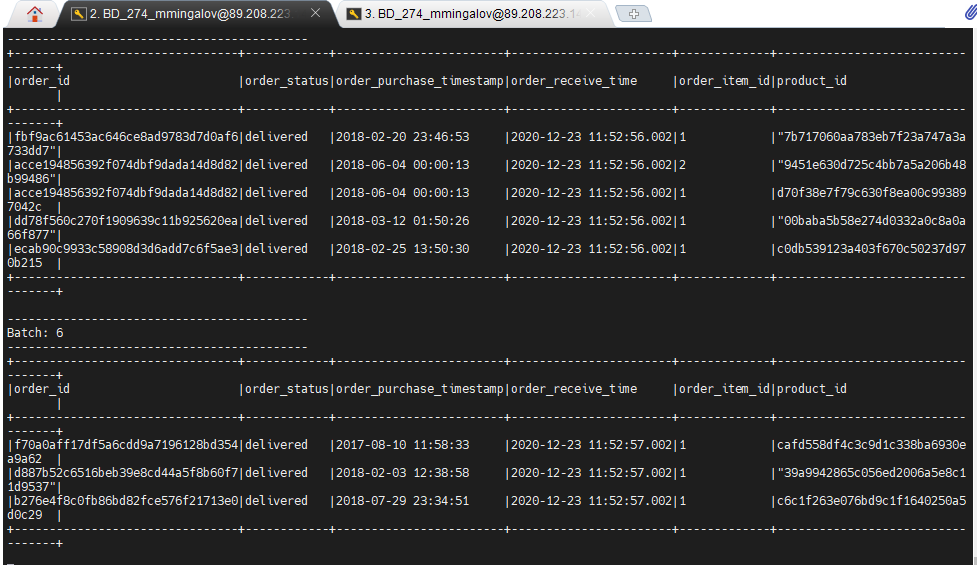
stream.stop()

# **JOINS**

#переопределим функцию

**def** console\_output(df, freq, out\_mode):  
 **return** df.writeStream.format(**"console"**) \  
 .trigger(processingTime=**'%s seconds'** % freq ) \  
 .options(truncate=**False**) \  
 .option(**"checkpointLocation"**, **"checkpoints/my\_watermark\_console\_chk2"**) \  
 .outputMode(out\_mode) \  
 .start()

*#stream-hdfs join*static\_df = spark.table(**"spark\_streaming\_course.order\_items"**)  
static\_joined = waterwarked\_orders.join(static\_df, **"order\_id"**, **"inner"** )  
static\_joined.isStreaming  
static\_joined.printSchema *#колонка order\_id только одна*selected\_static\_joined = static\_joined.select(**"order\_id"**, **"order\_status"**, **"order\_purchase\_timestamp"**, **"order\_receive\_time"**, **"order\_item\_id"**, **"product\_id"**)  
  
stream = console\_output(selected\_static\_joined , 1, **"update"**)



stream.stop()