1. Использовал docker-образы Bitnami для установки обеих систем: Kafka и Spark, т.к. установить непосредственно по инструкции, показанной на лекции, не получилось.

|  |  |  |  |
| --- | --- | --- | --- |
| **ПО** | **Версия ПО** | **Ссылка на Git** | **Команда docker** |
| Kafka | 3.8.0 | https://github.com/bitnami/containers/tree/main/bitnami/kafka | docker pull bitnami/kafka |
| Spark | 3.5.2 | https://github.com/bitnami/containers/tree/main/bitnami/spark | docker pull bitnami/spark |

1. Создал **docker-compose.yml** файл для запуска контейнеров в единой сети:

services: zookeeper: image: bitnami/zookeeper ports: - "2181:2181" environment: ALLOW\_ANONYMOUS\_LOGIN: "yes" networks: - kafka-network kafka: image: bitnami/kafka environment: KAFKA\_CFG\_ZOOKEEPER\_CONNECT: zookeeper:2181 KAFKA\_LISTENERS: INSIDE://PLAINTEXT://0.0.0.0:29092,OUTSIDE://PLAINTEXT://0.0.0.0:9092 KAFKA\_ADVERTISED\_LISTENERS: INSIDE://kafka:29092,OUTSIDE://172.21.182.15:9092 KAFKA\_LISTENER\_SECURITY\_PROTOCOL\_MAP: INSIDE:PLAINTEXT,OUTSIDE:PLAINTEXT KAFKA\_INTER\_BROKER\_LISTENER\_NAME: INSIDE ports: - "9092:9092" depends\_on: - zookeeper networks: - kafka-network spark-master: image: bitnami/spark ports: - "8080:8080" - "7077:7077" volumes: - /mnt/d/Курсы и тренинги/Data Engineer Нетология/Spark Structured Streaming:/opt/spark/work-dir environment: - SPARK\_MODE=master networks: - kafka-network spark-worker: image: bitnami/spark volumes: - /mnt/d/Курсы и тренинги/Data Engineer Нетология/Spark Structured Streaming:/opt/spark/work-dir environment: - SPARK\_MODE=worker - SPARK\_WORKER\_MASTER=spark://spark-master:7077 depends\_on: - spark-master networks: - kafka-network networks: kafka-network: driver: bridge

Здесь добавлена сеть **kafka-network** для того, чтобы оба контейнера видели друг друга и могли обмениваться данными.

Добавление Zookeeper было необязательным, просто «из коробки» Kafka запускалась с ошибками и, потратив много сил, нашел в сети решение, основанное на Zookeeper.

Сейчас понимаю, что можно было запустить без Zookeper, в режиме KRaft.

Что важно: пришлось переписать настройки Listeners, т.к. исходные Listeners от Bitnami настроены так, что не допускают подключения внешних клиентов:

- KAFKA\_CFG\_LISTENERS=PLAINTEXT://:9092,CONTROLLER://:9093

- KAFKA\_CFG\_ADVERTISED\_LISTENERS=PLAINTEXT://:9092

В **docker-compose.yml** я разделил Listeners на внутренних и внешних, через указание разных портов, на которых Kafka слушает внутренних и внешних клиентов. В качестве IP для внешнего хоста использовал IP WSL своего ПК ('172.21.182.15' определил командой ip addr в Bash)

Также в контейнеры добавлена опция volumes с указанием пути до python и spark-скриптов на моем локальном ПК для монтирования томов и обеспечения возможности запуска скриптов непосредственно внутри контейнеров.

1. Выполнил в терминале docker-compose up, развернул контейнеры в единой сети
2. В своем Python-окружении установил необходимые библиотеки:

* pip install kafka
* pip install pyspark

1. Поскольку Kafka установлена в контейнере, пришлось внести правки в скрипт **producer.py**:

* вместо строки KafkaProducer(bootstrap\_servers=['localhost:9092']   
  прописал KafkaProducer(bootstrap\_servers=['172.21.182.15:9092'],  
  где '172.21.182.15' – мой IP WSL (определил командой ip addr в Bash)
* в строке producer.send  
  добавил методы .add\_callback и .add\_errback, вызывающие соответствующие функции для формирования логов и получения обратной связи о том, было ли сообщение успешно отправлено:

# Функция для обратного вызова при успешной отправке сообщения

def on\_send\_success(record\_metadata):

print(f"Message sent to {record\_metadata.topic} partition {record\_metadata.partition}with offset {record\_metadata.offset}")

# Функция для обратного вызова при ошибке

def on\_send\_error(excp):

print(f"Message delivery failed: {excp}")

1. Конечная версия скрипта **producer.py**, генерирующего входной поток для Kafka:

from time import sleep

from json import dumps

from kafka import KafkaProducer

from random import randrange, choices

import string

# Функция для обратного вызова при успешной отправке сообщения

def on\_send\_success(record\_metadata):

print(f"Message sent to {record\_metadata.topic} partition {record\_metadata.partition} with offset {record\_metadata.offset}")

# Функция для обратного вызова при ошибке

def on\_send\_error(excp):

print(f"Message delivery failed: {excp}")

producer = KafkaProducer(bootstrap\_servers=['172.21.182.15:9092'],

value\_serializer=lambda x: dumps(x).encode('utf-8'),

retries=5)

def push():

for e in range(50):

text = ''.join(choices(string.ascii\_uppercase + string.digits, k=20))

user = {'id': randrange(5), 'action': text}

# Отправляем сообщение с обработчиками

producer.send('netology-spark', value=user).add\_callback(on\_send\_success).add\_errback(on\_send\_error)

sleep(randrange(3))

print("Produced message ", e)

try:

while True:

push()

except KeyboardInterrupt:

producer.close()

print("Exit")

1. Запуск скрипта генерирует принты вида:

Message sent to netology-spark partition 0 with offset 255

Produced message 0

Produced message 1

Message sent to netology-spark partition 0 with offset 256

Message sent to netology-spark partition 0 with offset 257

Produced message 2

Message sent to netology-spark partition 0 with offset 258

Produced message 3

Message sent to netology-spark partition 0 with offset 259

Produced message 4

Produced message 5

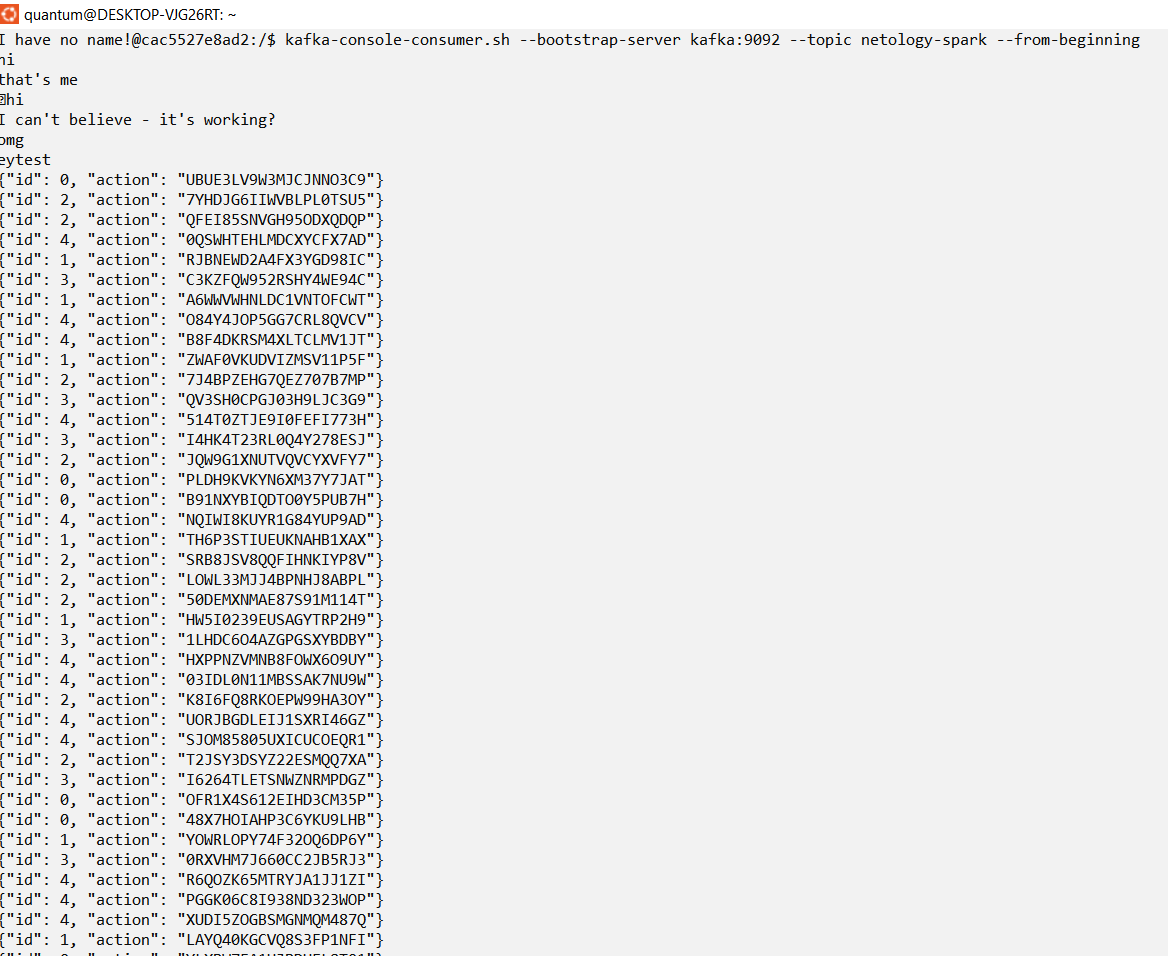
Message sent to netology-spark partition 0 with offset 260

Message sent to netology-spark partition 0 with offset 261

которые свидетельствует об успешной отправке сообщений в Kafka.

1. Заходим в контейнер Kafka. Для этого:

* командой docker ps смотрим id контейнера Kafka
* вводим команду docker exec -it <id контейнера> /bin/bash
* для чтения входящего потока внутри контейнера вводим kafka-console-consumer.sh --bootstrap-server kafka:9092 --topic netology-spark --from-beginning
* видим поступающие от продюсера сообщения в формате бинарных данных:



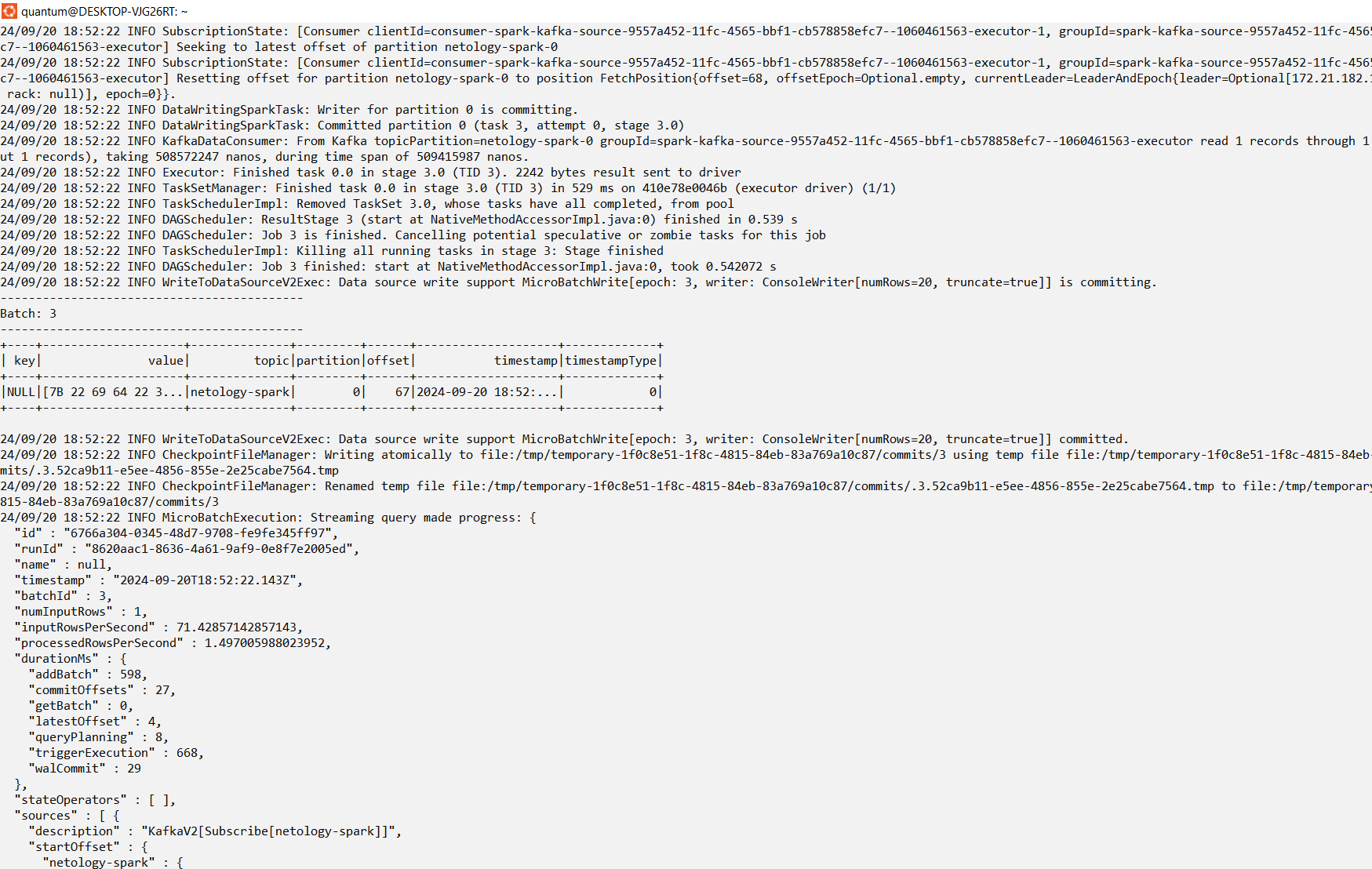
1. Дальше заходим в контейнер Spark с помощью команды docker exec - t <id контейнера Spark> /bin/bash
2. Для получения входного потока из Kafka внутри контейнера Spark выполняем /opt/bitnami/spark/conf$ spark-submit --packages org.apache.spark:spark-sql-kafka-0-10\_2.12:3.5.2 /opt/spark/work-dir/structure\_streaming\_kafka\_test.py

Я создал бэкап версию Spark-скрипта (structure\_streaming\_kafka\_test.py) и работал с ней.

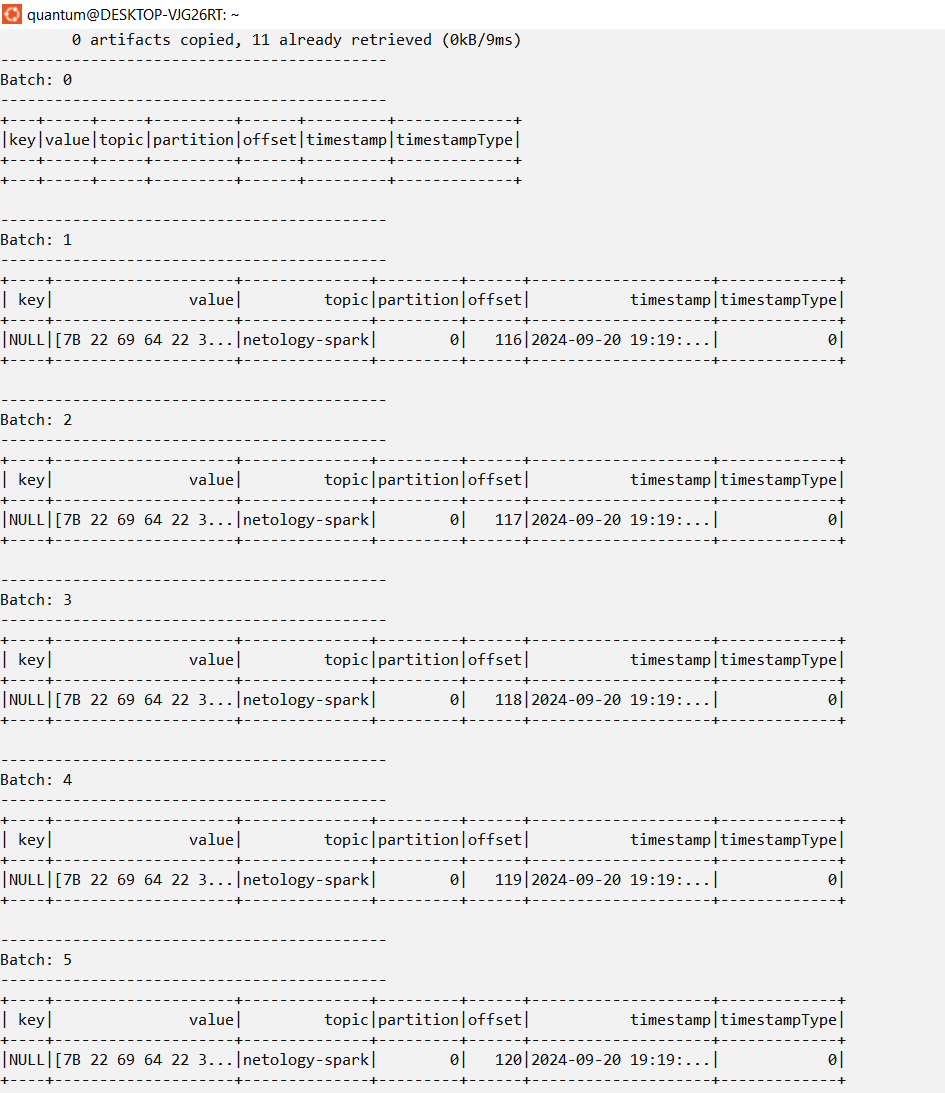
Данный скрипт хранится в директории, к которой был смонтирован том в разделе volumes docker-compose.yml

1. Дальнейшие манипуляции и распаковка данных из Kafka осуществлялись исключительно через редактирование файла **structure\_streaming\_kafka\_test.py:**

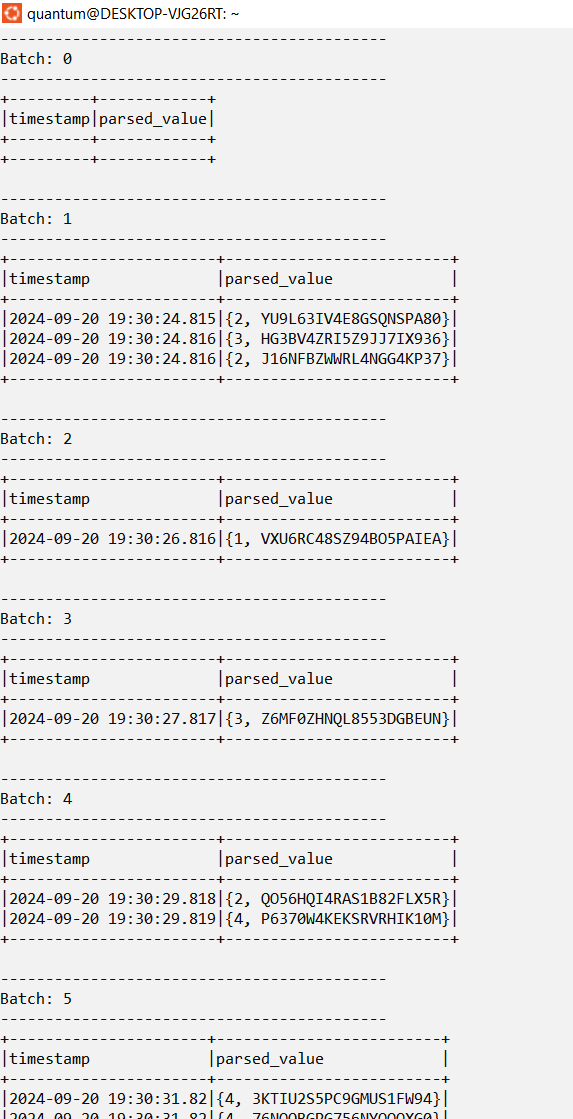
* Чтение бинарных данных с логированием на уровне **info**:



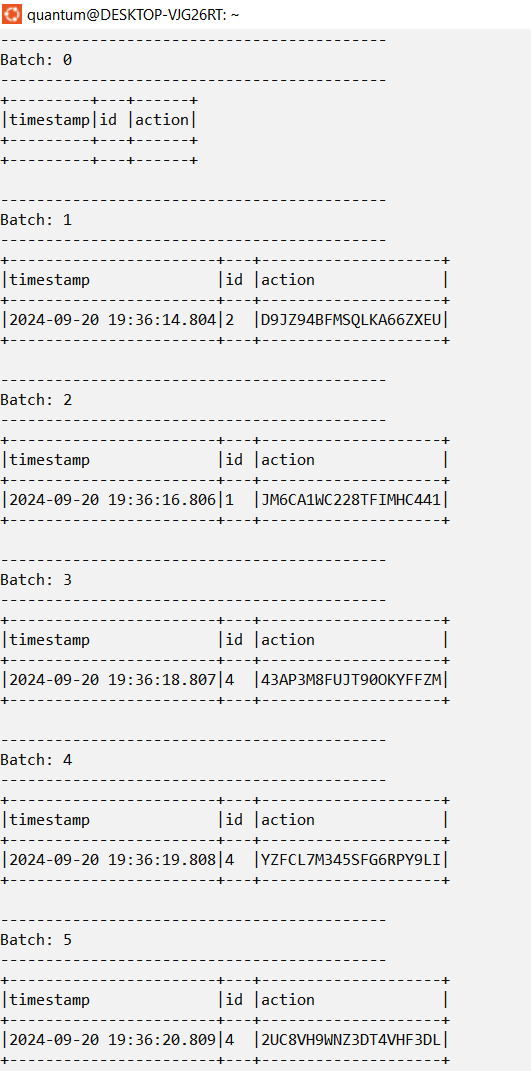
* Чтение бинарных данных с логированием на уровне **error**:



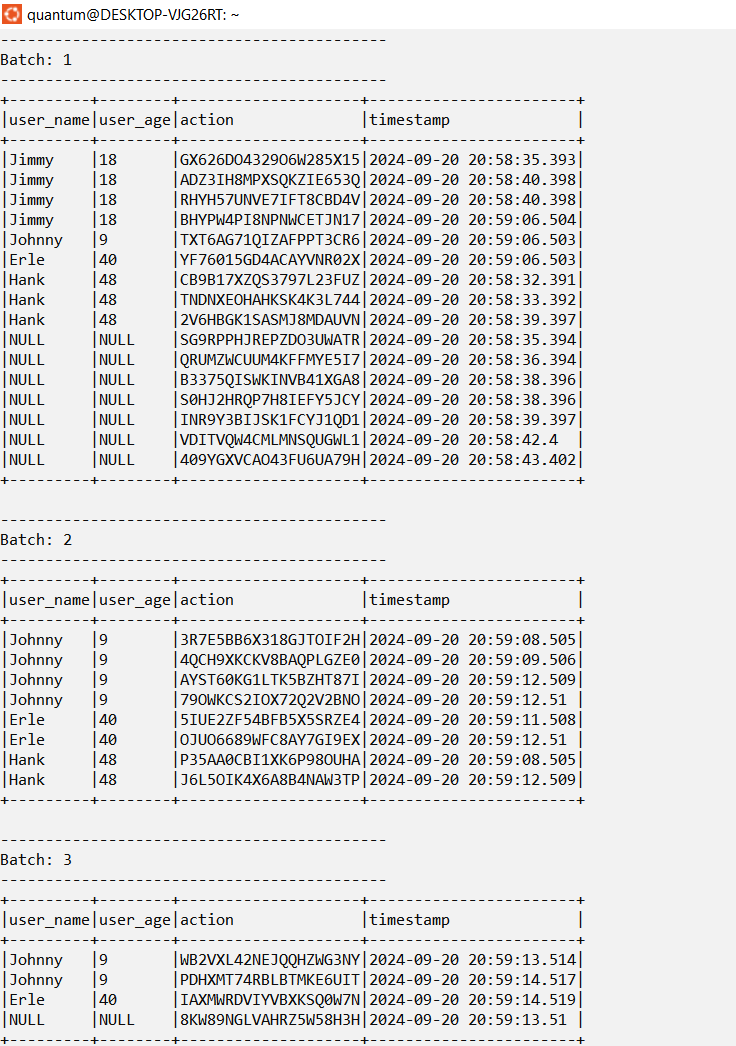
* Распакованные данные после применения схемы и перевода в json:



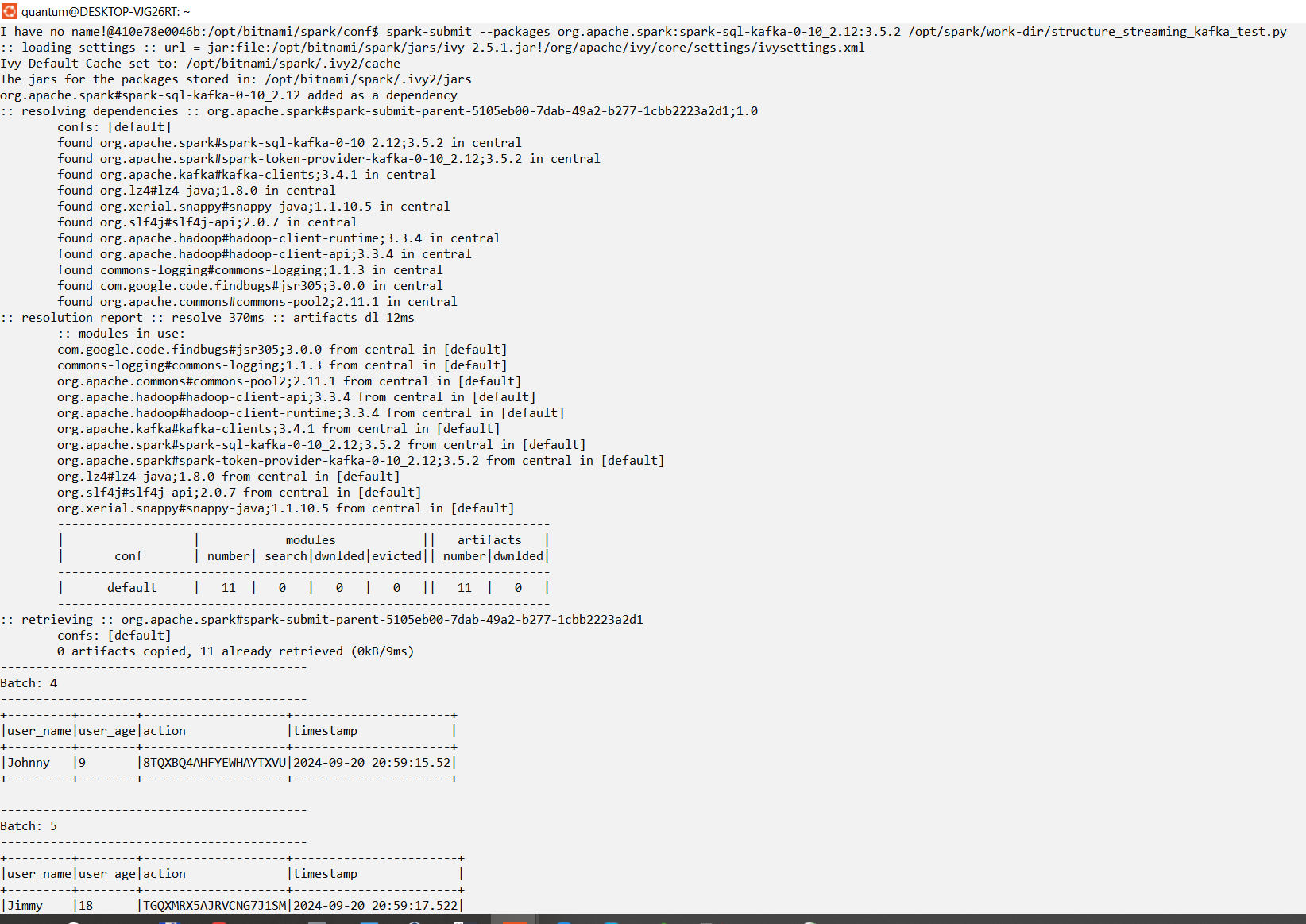
* Распакованные данные до уровня user\_id и action:



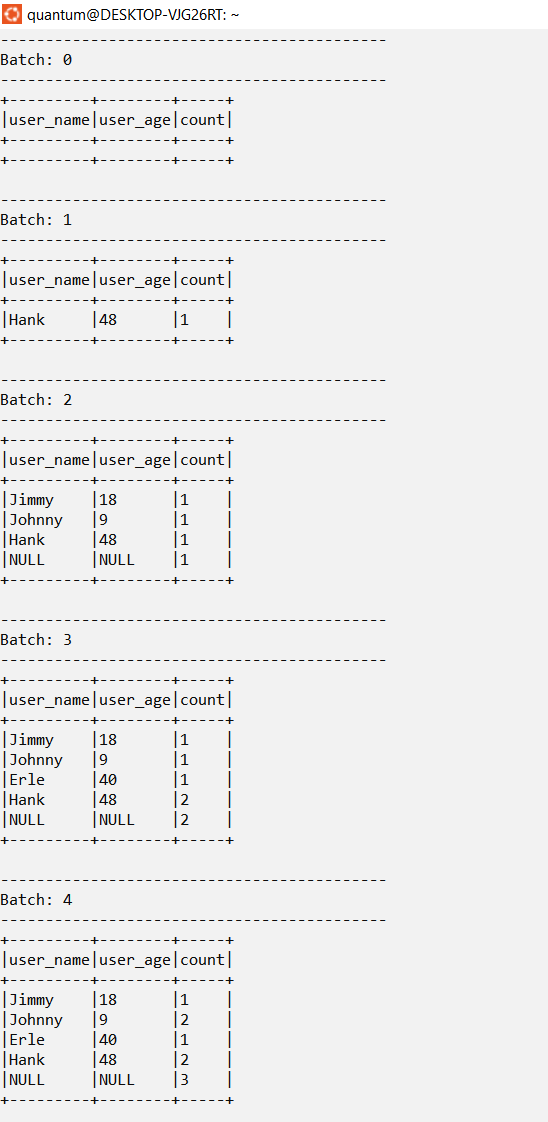
* Данные после джойна входящего и статического потоков:



* После добавления чекпойнтов фиксируем, что чтение данных из входного потока возобновляется не с 0, а с последнего прочитанного батча:



* Чтение данных после агрегации:



1. Итоговый Spark-скрипт:

from pyspark.sql import SparkSession

from time import sleep

from pyspark.sql.functions import col,from\_json

from pyspark.sql.types import StructType, StringType, IntegerType

# явным образом задаем структуру json-контента

schema = StructType().add("id",IntegerType()).add("action", StringType())

users\_schema = StructType().add("id",IntegerType()).add("user\_name", StringType()).add("user\_age", IntegerType())

spark = SparkSession.builder.appName("SparkStreamingKafka").getOrCreate()

input\_stream = spark \

.readStream \

.format("kafka") \

.option("kafka.bootstrap.servers", "kafka:29092") \

.option("subscribe", "netology-spark") \

.option("failOnDataLoss", False) \

.load()

#покажем входящий контент

#input\_stream.writeStream.format("console").outputMode("append").start().awaitTermination()

#input\_stream = input\_stream.writeStream.format("console").outputMode("append").start()

json\_stream = input\_stream.select(col("timestamp").cast("string"), from\_json(col("value").cast("string"), schema).alias("parsed\_value"))

#json\_stream.writeStream.format("console").outputMode("append").option("truncate", False).start().awaitTermination()

#выделим интересующие элементы

clean\_data = json\_stream.select(col("timestamp"), col("parsed\_value.id").alias("id"), col("parsed\_value.action").alias("action"))

#clean\_data.writeStream.format("console").outputMode("append").option("truncate", False).start().awaitTermination()

#добавим join со статическим dataset - создаем данные

users\_data = [(1,"Jimmy",18),(2,"Hank",48),(3,"Johnny",9),(4,"Erle",40)]

users = spark.createDataFrame(data=users\_data,schema=users\_schema)

#делаем join

#join\_stream = clean\_data.join(users, clean\_data.id == users.id, "left\_outer").select(users.user\_name, users.user\_age, clean\_data.action, clean\_data.timestamp)

join\_stream = clean\_data.join(users, clean\_data.id == users.id, "left\_outer").select(users.user\_name, users.user\_age, clean\_data.action, clean\_data.timestamp)

#join\_stream.writeStream.format("console").outputMode("append").option("truncate", False).start().awaitTermination()

#убираем terminate

#res\_checkpoints= join\_stream.writeStream.\

#format("console").\

#outputMode("append").\

#option("truncate", False).\

#option("checkpointLocation", "checkpoint/target").\

#start()

#sleep(10)

#res\_checkpoints.stop()

#добавим агрегат - отображать число уникальных айдюков

stat\_stream = clean\_data.groupBy("id").count()

#stat\_stream.writeStream.format("console").outputMode("complete").option("truncate", False).start().awaitTermination()

join\_stream\_agg = stat\_stream.join(users, stat\_stream.id == users.id, "left\_outer").select(users.user\_name, users.user\_age, col('count'))

res\_checkpoints\_agg= join\_stream\_agg.writeStream.\

format("console").\

outputMode("complete").\

option("truncate", False).\

option("checkpointLocation", "checkpoint/target").\

start()\

sleep(10)

res\_checkpoints\_agg.stop()

1. Для сокращения количества логов внесена правка в конфигурационный файл **log4j2.properties** в контейнере Spark:

* Оригинальный файл **/opt/bitnami/spark/conf/log4j2.properties.template** скопирован; в копии убрано окончание **.template**
* В новом файле найден параметр rootLogger.level = info;   
  ‘info’ заменен на ‘error’

