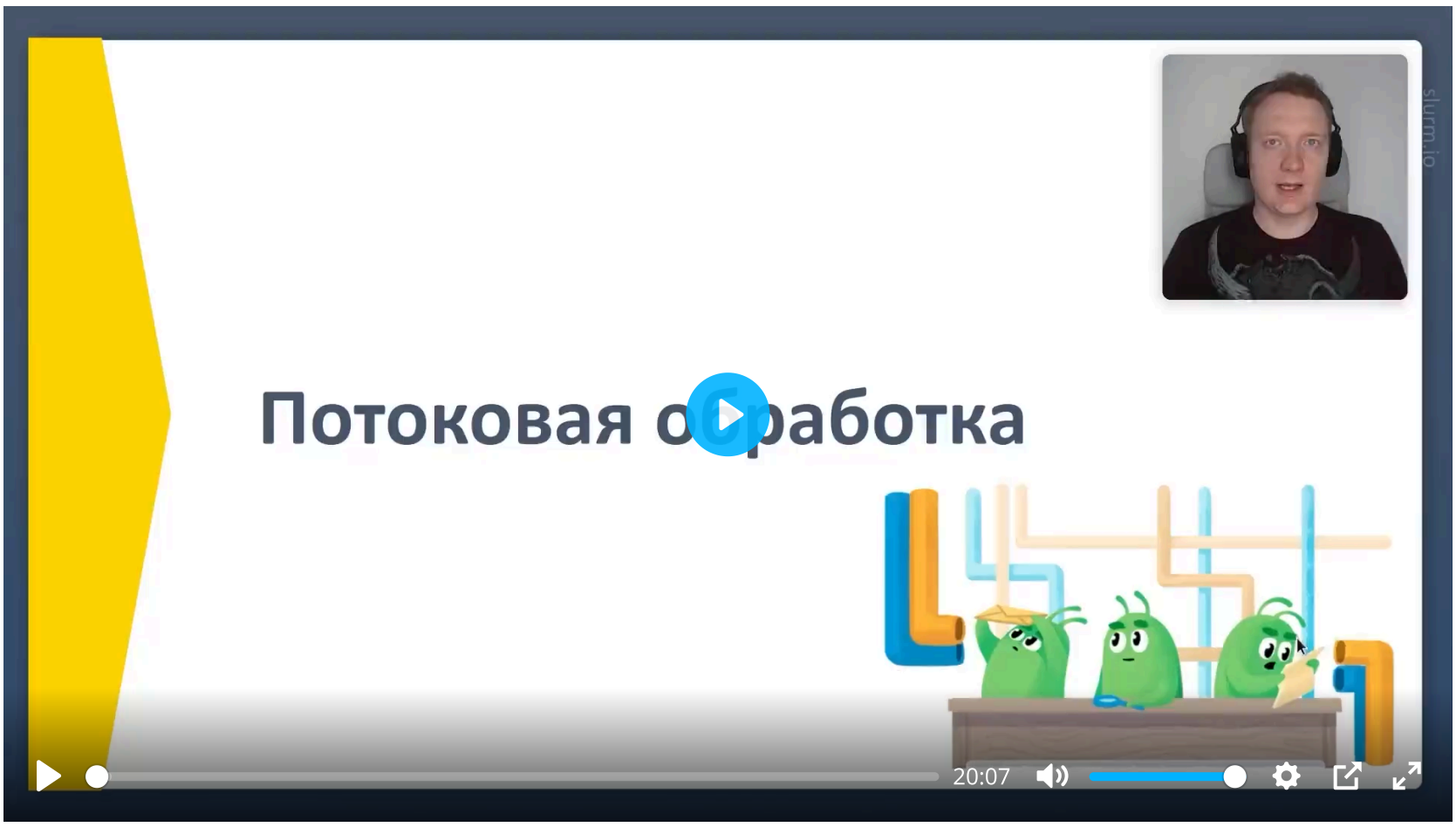
4.4 Партицирование, репартицирование Spark Streaming



Текстовая расшифровка видео:

ПОТОКОВАЯ ОБРАБОТКА

План:

- Discretized Streams;
- Источники данных;
- Простой пример;
- Трансформации;
- Structured Streaming;
- Minibatch WordCount.

Discretized Streams

В Spark существует два варианта подхода к работе с потоковыми данными:

DStreams (Discretized Stream) – это подход, основанный на RDD.

Spark Structured Streaming – это подход, реализующийся поверх DataFrame. Данный подход считается более новым. **Официальная рекомендация:** любой новый стриминг следует писать с помощью Structured Streams.

Как работает DStreams?

Когда мы создаем RDD, мы можем создать целую пачку. Также, когда прилетают данные, мы можем из каждого фрагмента данных создавать новый RDD и каждый из этих RDD независимо процессить в параллель. Эта идея похожа на partitions, только в стриминговой парадигме:

- Каждый кусочек данных на кластере в RDD «размазан», данные прилетают в виде RDD.
- Мы применяем к ним стриминговую обработку.

- Пишем результат дальше в приемник.

Источники данных

Источников для работы с потоками два (условное разделение):

Простые источники – источники, базово поддерживающиеся Spark'ом из коробки (чтение из файлов, чтение из сетевых сокетов).

Сложные источники – источники с более сложной стриминговой системой (Apache Kafka, AWS Kinesist и т.д.).

Простой пример

Рассмотрим простой пример того, как может выглядеть работа в парадигме DStreams:

```
1. from pyspark import SparkContext, SparkConf
2.
3. conf = SparkConf().setAppName("my-app").setMaster("local[4]")
4. sc = SparkContext(conf=conf)
5. ssc = StreamingContext(sc, 10)
6.
7. rdd_q = [
8.   sc.parallelize([j for j in range(1, 1001)], 10)
9. ]
10. input_stream = ssc.queueStream(rdd_q)
11. by_10_stream = input_stream.map(lambda x: (x % 10, 1))
12. sum_stream = by_10_stream.reduceByKey(lambda a, b: a + b)
13. sum_stream.pprint()
14.
15. ssc.start()
16. time.sleep(6)
17. ssc.stop(stopSparkContext=True, stopGraceFully=True)
```

Шаги:

- Создаем пачку RDD;
- Параллелизуем и разбиваем их на 10 кусков;
- В каждом из этих кусков есть числа от 1 до 1000. Для каждых из этих чисел считаем пару, где ключом будет остаток от деления числа на 10;
- В конце делаем reduceByKey, то есть суммируем, сколько раз каждый из результатов встречался в исходных данных.

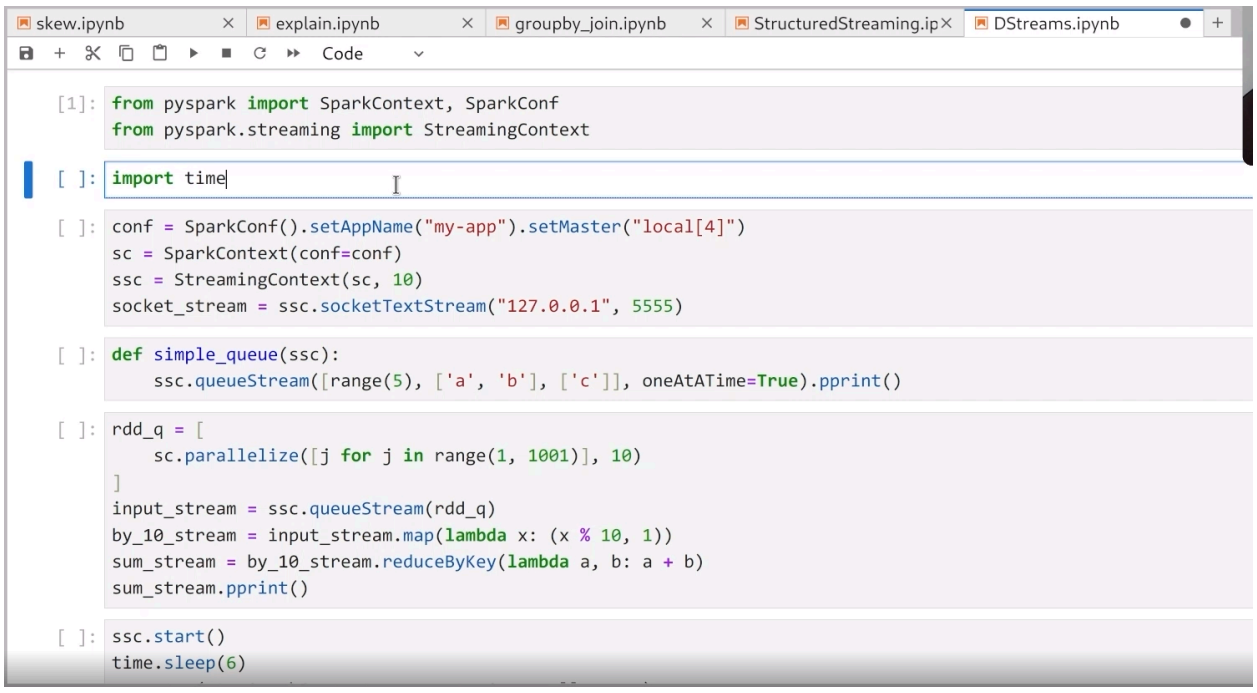
Формально мы можем добавлять дополнительные RDD, они также будут обсчитывать результаты, а результат также будет вводится.

В примере мы вводим результаты, запускаем, ждем выведения результата и полностью останавливаем с закрытием и корректным завершением всех промежуточных механизмов.

Посмотрим, как это может выглядеть.

У нас есть DStreams. Достанем SparkContext.

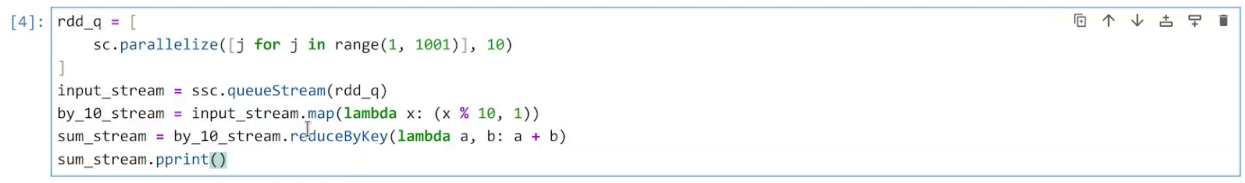
Обратите внимание: помимо стандартного SparkContext, SparkSession появился StreamingContext:



Поскольку в примере использована новая версия Spark, Spark «ругается» (DStreams deprecated) и просит взять новый Structured Streaming:

```
/usr/local/spark/python/pyspark/streaming/context.py:72: FutureWarning: DStream is deprecated as of Spark 3.4.0. Migrate to Structur
ed Streaming.
  warnings.warn(
```

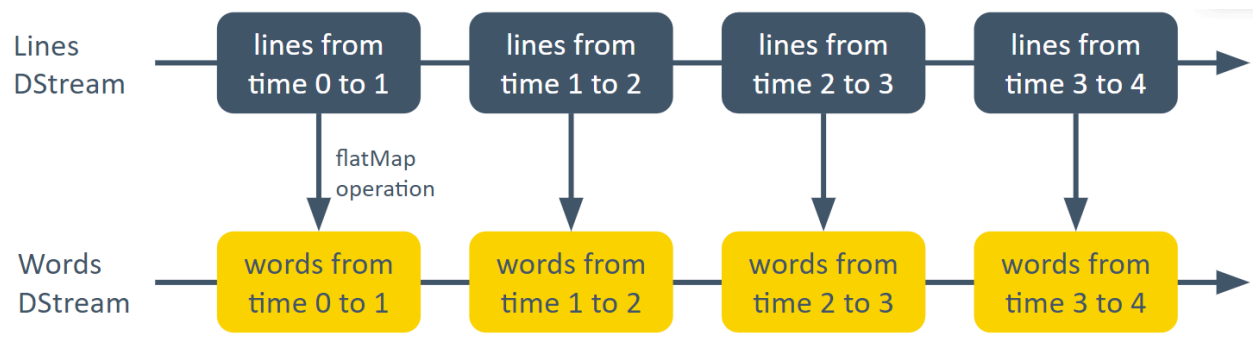
Обратите внимание на pprint:



Он ничего не вывел. Он добавился, как некая трансформация, как команда «Что делать потоку с каждым кусочком данных, который туда прилетает». Когда мы запустим поток и подождем, данные начнут писаться:

Сделаем start. Видим, что порция вывелась. Закончились 6 секунд, мы завершились:

Обратите внимание на схему:



Здесь показан пример, который называется «Разбиение текста на слова». Прилетает текст со строчками, мы используем условный flatMap, нарезая текст по пробельному символу. Далее, в выходной поток летят RDD со словами, а не строками.

Дедлайн 07 июля, 23:59 Мск

Многие из этих операций рассчитаны на применение чего-либо к разным рядам в RDD. Операции, например, map(), filter(), flatMap(), repartition() и т.д. работают для RDD так же, как и для DStreams. Отличие только в том, что они работают на потоке (для каждого из кусочков отдельно).

Отдельно стоит упомянуть оконные функции. Поскольку мы работаем с потоком, стоит воспользоваться поддержкой «окошек», чтобы иметь «скользящее окно», которым мы сможем считать, например, скользящее среднее или среднее количество кликов в логах сайта.

Оконные функции:

- window();
- countByWindow();
- reduceByKeyAndWindow();
- countByValueAndWindow().

Претензия к Spark: это стриминг, а minibatch, потому что данные нарезаются на RDD, с каждым мы работаем, как с RDD и т.д.

Structured Streaming

Можно представить, что это виртуальная таблица, в конец которой постоянно записываются новые данные.

Structured Streaming позволяет использовать гораздо больше подкапотных оптимизаций благодаря Catalyst. На стриме к нему можно применять SQL-запросы, то есть можно использовать SQL даже для потоков.

Лучше интегрируется со сторонними плагинами, например, с MLLib.

Код во многом проще.

Minibatch WordCount

Рассмотрим пример:

```
1. from pyspark.sql import SparkSession
2. from pyspark.sql.functions import explode, split
3.
4. spark = SparkSession.builder.appName("SlurmStructuredStreamingDemo") \
5.     .getOrCreate()
6. lines = spark.readStream.format("socket") \
7.     .option("host", "127.0.0.1").option("port", 5555).load()
8. words = lines.select(explode(split(lines.value, " ")).alias("word"))
9. word_counts = words.groupBy("word").count()
10. query = word_counts.writeStream.outputMode("complete").format("console").start()
11. query.awaitTermination()
```

Мы делаем простейший WordCount, как мы уже делали это при работе с DataFrame. В примере используются похожие функции: разбиваем DataFrame по словам и считаем, сколько раз каждое слово встречалось в исходных данных. Нюанс: мы это делаем на потоке, то есть периодически смотрим на момент прихода очередной порции данных. Также следует обратить внимание на API. Раньше мы вызывали в обычном режиме spark.read и spark.write. Здесь мы вызываем spark.readStream и spark.writeStream. Мы читаем данные из какого-то источника (например, из сокета), пишем данные в какой-то приемник (например, в вывод самого Spark) и запускаем, как стриминговый query.

Чтобы это работало, перезапускаем локальный Spark с указанием NetWare Host, так как мы хотим читать с локального сокета:

```
Alacritty                                                    meow-nofer - Thunar
 meow-nofer@clefairy  ~/Experiments/slurm  docker run --rm --name pyspark -v $(pwd):/home/jo
yter/pyspark-notebook
Entered start.sh with args: jupyter lab
/usr/local/bin/start.sh: running hooks in /usr/local/bin/before-notebook.d as uid / gid: 1000 /
/usr/local/bin/start.sh: running script /usr/local/bin/before-notebook.d/spark-config.sh
/usr/local/bin/start.sh: done running hooks in /usr/local/bin/before-notebook.d
Executing the command: jupyter lab
```

Мы используем стандартную утилиту «NC», которая позволит закинуть чуть больше данных в сокет:

```
[I 2023-07-28 17:33:02.301 ServerApp] Use Control-C to stop this server and shut down all kernels (twice to skip confir
mation).
[C 2023-07-28 17:33:02.307 ServerApp]

    To access the server, open this file in a browser:
        file:///home/jovyan/.local/share/jupyter/runtime/jpserver-7-open.html
    Or copy and paste one of these URLs:
        http://clefairy:8888/lab?token=e87fc85bce6a6dd7a802c5de92cd9e26ebd71376ba3f5358
        http://127.0.0.1:8888/lab?token=e87fc85bce6a6dd7a802c5de92cd9e26ebd71376ba3f5358
[I 2023-07-28 17:33:04.120 ServerApp] Skipped non-installed server(s): bash-language-server, dockerfile-language-server
-nodejs, javascript-typescript-langserver, jedi-language-server, julia-language-server, pyright, python-language-server
, python-lsp-server, r-languageserver, sql-language-server, texlab, typescript-language-server, unified-language-server
, vscode-css-languageserver-bin, vscode-html-languageserver-bin, vscode-json-languageserver-bin, yaml-language-server
[W 2023-07-28 17:33:05.216 ServerApp] 404 GET /api/kernels/85f5c18e-5165-4c80-b894-1a323bd42ee7?1690565585209 (127.0.0.
[0] 0:[tmux]* 1:zsh-                                                    "clefairy" 19:33 28-Jul-23
```

Создаем SparkSession. Здесь нам не нужно создавать StreamingContext и т.д.:

```
[*]: from pyspark.sql import SparkSession

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

[3]: from pyspark.sql.functions import explode, split

     Используем netcat: nc -lk 5555 (в случае Docker потребует --network host) и попечатаем что-нибудь

[4]: lines = spark.readStream.format("socket").option("host", "127.0.0.1").option("port", 5555).load()

[5]: words = lines.select(explode(split(lines.value, " ")).alias("word"))

[6]: word_counts = words.groupBy("word").count()

[7]: query = word_counts.writeStream.outputMode("complete").format("console").start()

[ ]: query.awaitTermination()

[ ]:
```

Теперь мы можем запустить nc с записью «5555». Набираем текст:

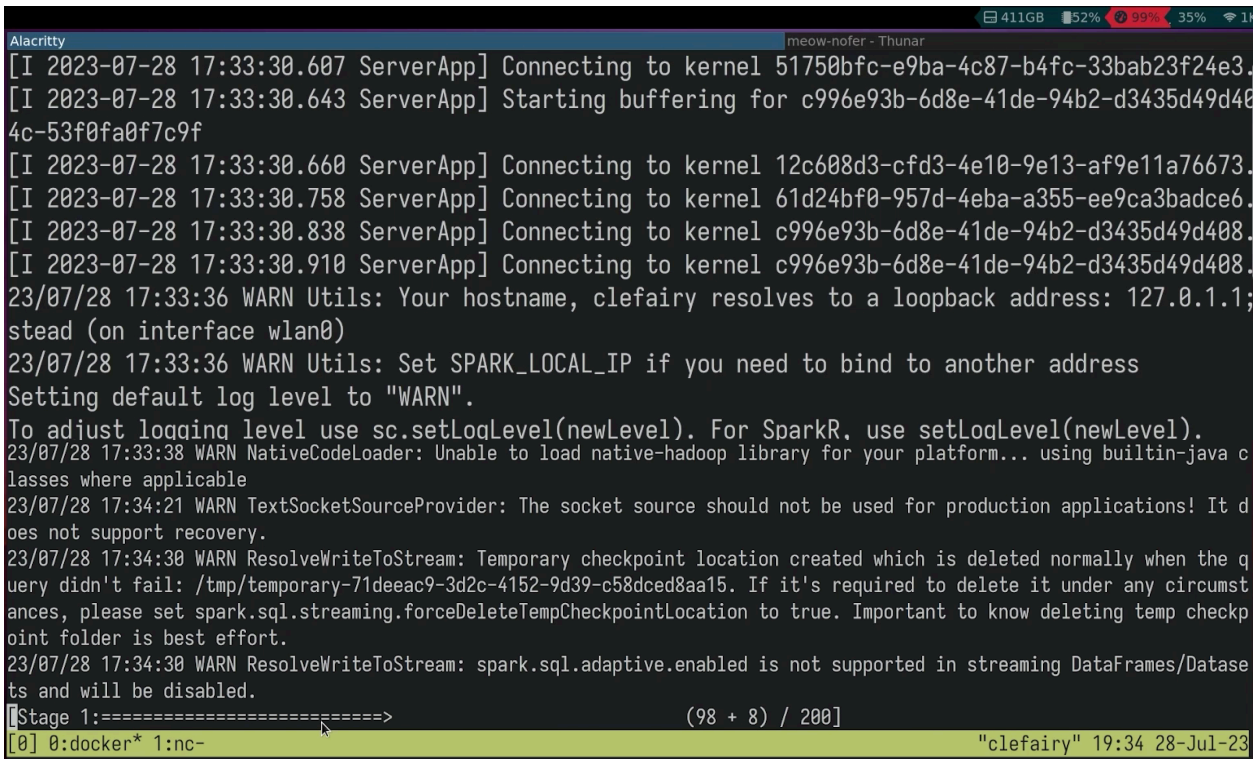
```
Alacritty                                                    meow-nofer - Thunar
 meow-nofer@clefairy  ~/Experiments/slurm  nc -lk 5555
abc abc abc asdasfsdg
aesrga wrgaew raerg dfb
dfb g asd awawee abc
```

Создаем читалку из потока. Для каждой новой порции будем разбивать по словам, группировать, считать word_count. У нас уже запущен query. Выполняем start и ждем завершения:

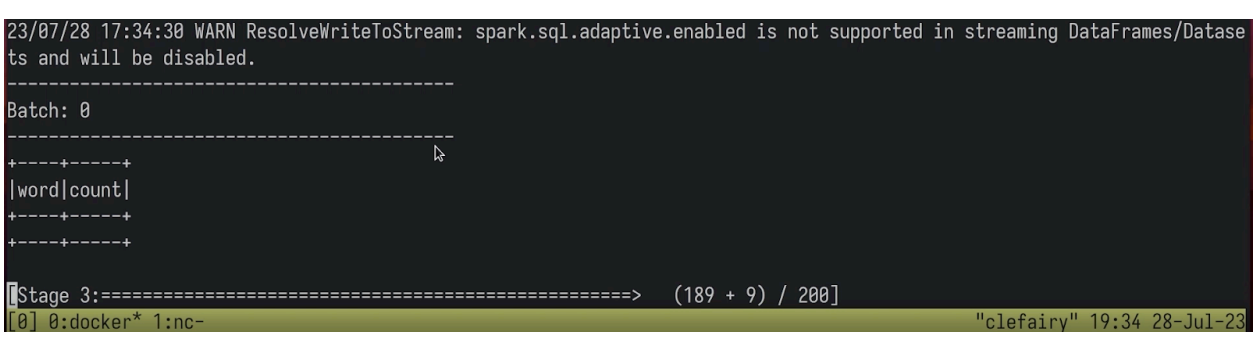
```
skew.ipynb × explain.ipynb × groupby_join.ipynb × StructuredStreaming.ip● × DStreams.ipynb × +

[1]: from pyspark.sql import SparkSession

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

[2]: from pyspark.sql.functions import explode, split

     Используем netcat: nc -lk 5555 (в случае Docker потребует --network host) и попечатаем что-нибудь

[3]: lines = spark.readStream.format("socket").option("host", "127.0.0.1").option("port", 5555).load()

[4]: words = lines.select(explode(split(lines.value, " ")).alias("word"))

[5]: word_counts = words.groupBy("word").count()

[6]: query = word_counts.writeStream.outputMode("complete").format("console").start()

[*]: query.awaitTermination()

[ ]:
```

Если посмотреть на логи Spark, мы увидим, что запустилась Streaming Job'a, пошел Stage, создалась query на потоке:

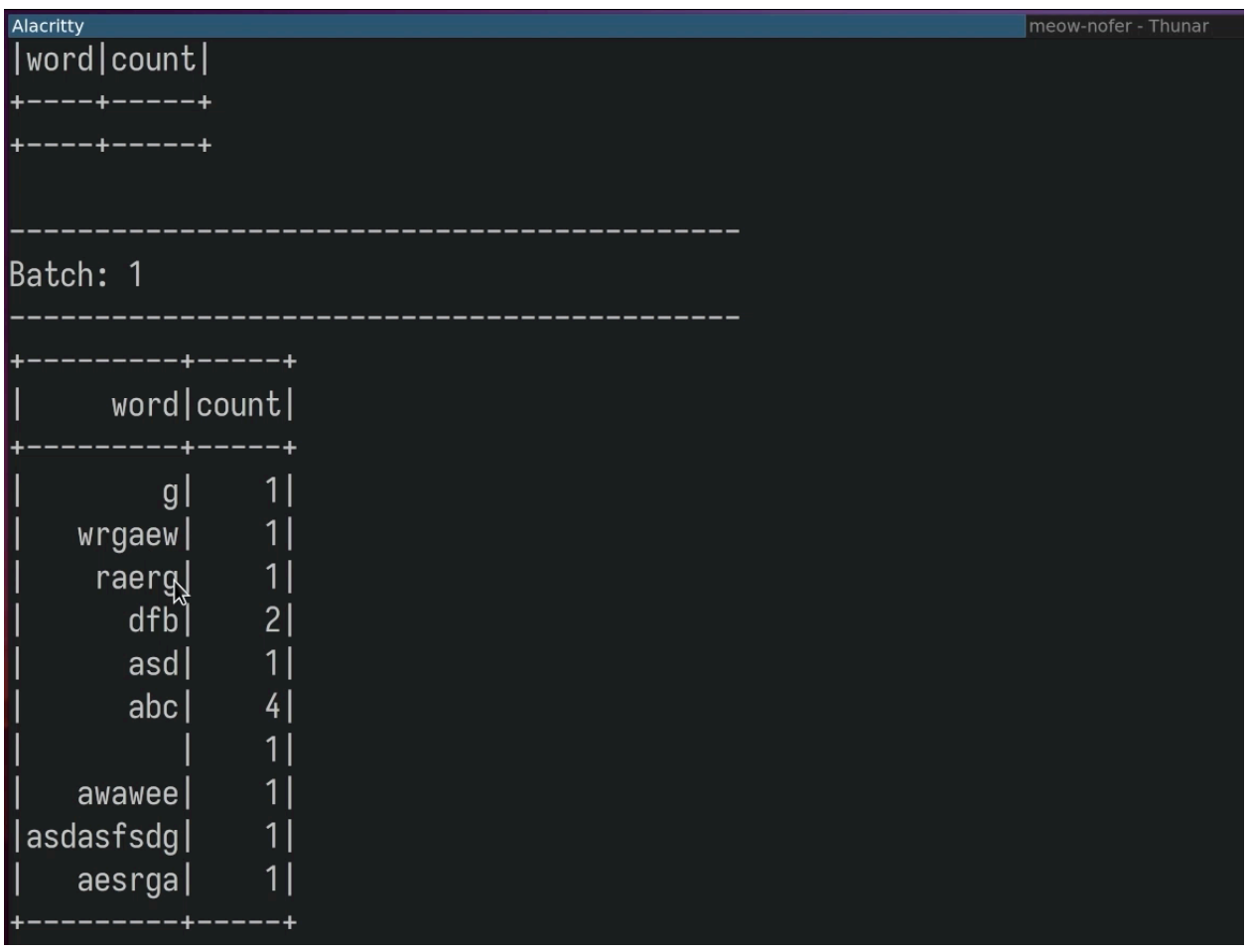
Дедлайн 07 июля, 23:59 Мск



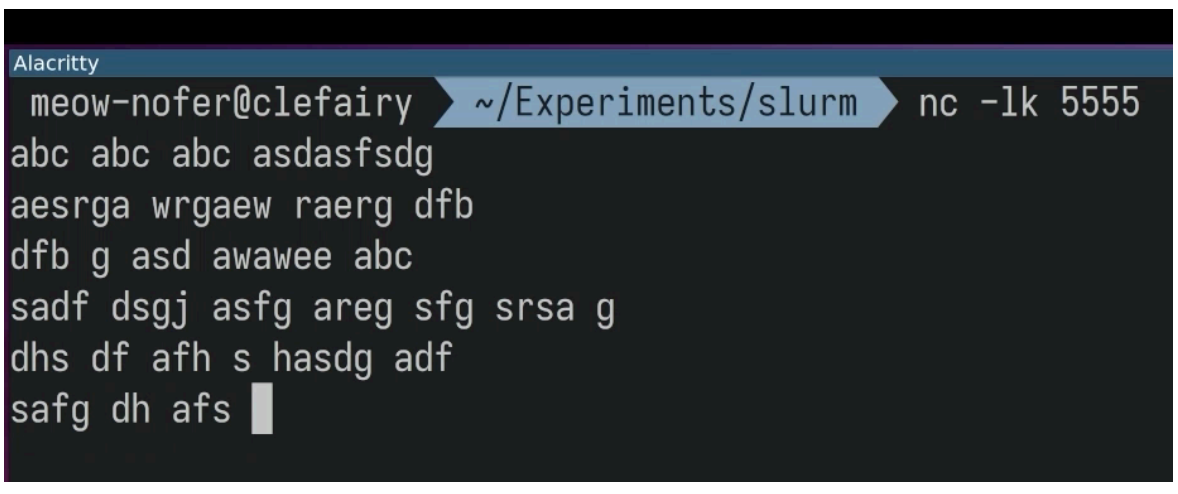
Когда мы будем писать новые данные на nc, они должны появляться здесь:



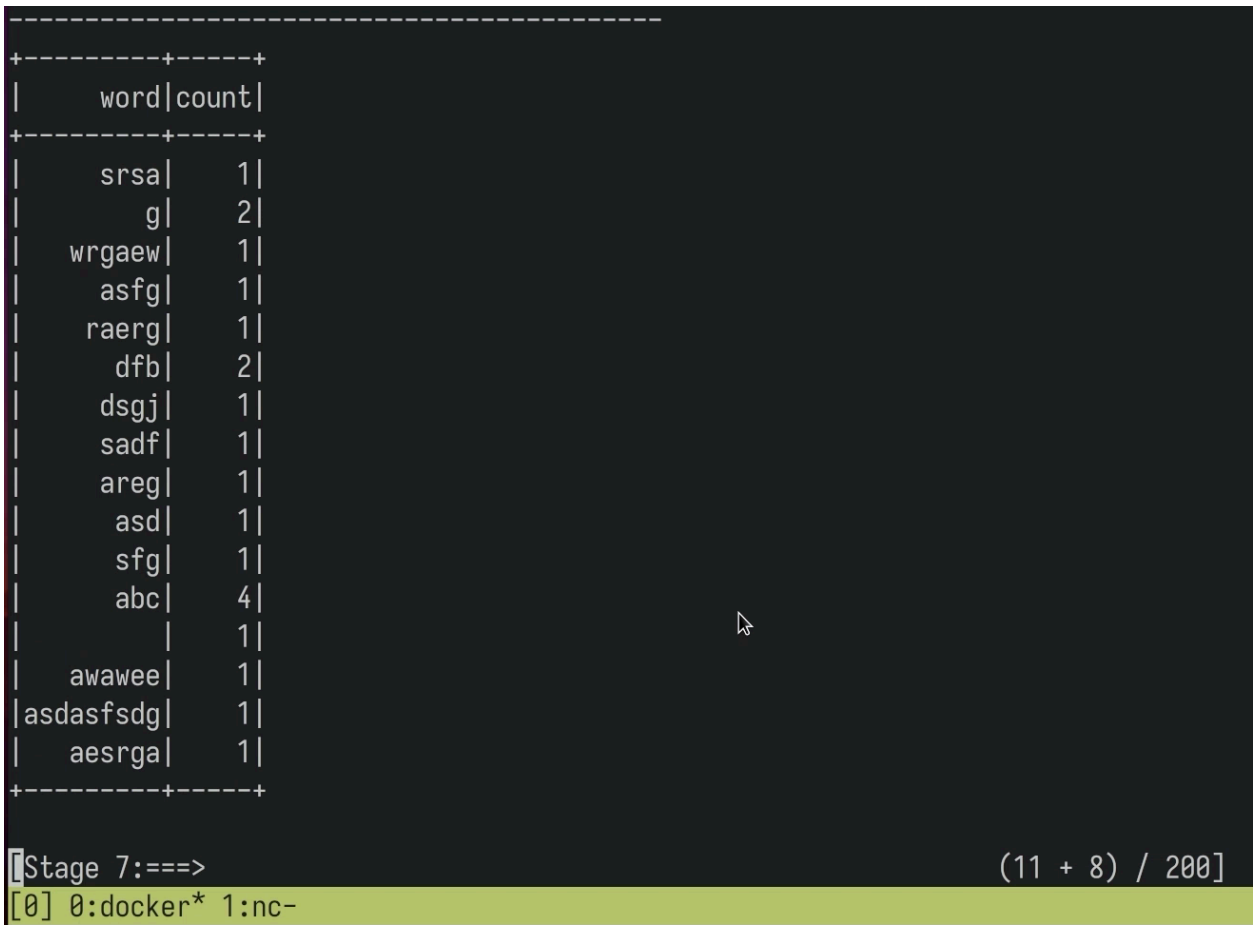
Видим, что он посчитал данные, которые мы закинули ранее:



Можем написать что-либо еще:



Дедлайн 07 июля, 23:59 Мск



Если вы скажете, что это minibatch, то будете правы. Мы действительно обрабатываем данные «батчами». Однако это Stream processing и то, как работает современный Spark Streaming.

Рекомендуем поэкспериментировать с этим самостоятельно.

ИТОГИ

Мы узнали:

- Что такое логический и физический план;
- Как бороться с неравномерным распределением данных по кластеру.
- Как кэшировать промежуточные результаты вычислений.
- Как с помощью spark можно решать задачи потоковой обработки.

Как вам урок?

Изучил, далее

Дедлайн 07 июля, 23:59 Мск