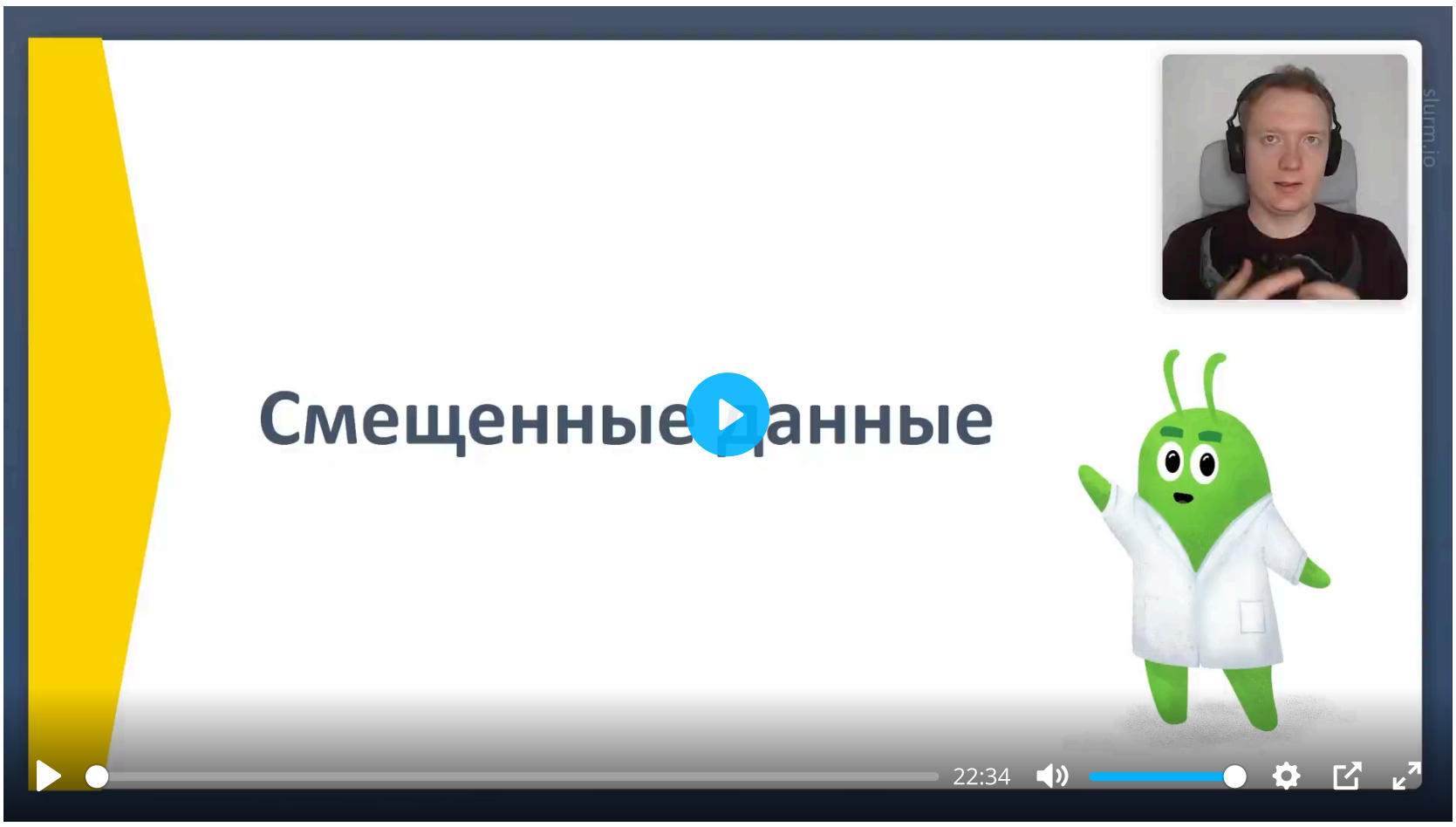
4.4 Партицирование, репартицирование Spark Streaming



Текстовая расшифровка видео:

СМЕЩЕННЫЕ ДАННЫЕ

План:

- Равномерно ли распределены данные;
- Data Skew;
- Виды партиционирования – RoundRobin;
- Виды партиционирования – HashPartitioning;
- Другой вариант – coalesce;
- «Соленые» данные;
- Подытог.

Равномерно ли распределены данные?

Одновременно Spark выполняет N task’ов, которые обрабатывают N партиций, где N – это суммарное число доступных потоков на всех воркерах.

Исходя из этого, важно обеспечивать:

- Достаточное количество партиций для распределения нагрузки по всем воркерам;
- Равномерное распределение данных между партициями.

Data Skew

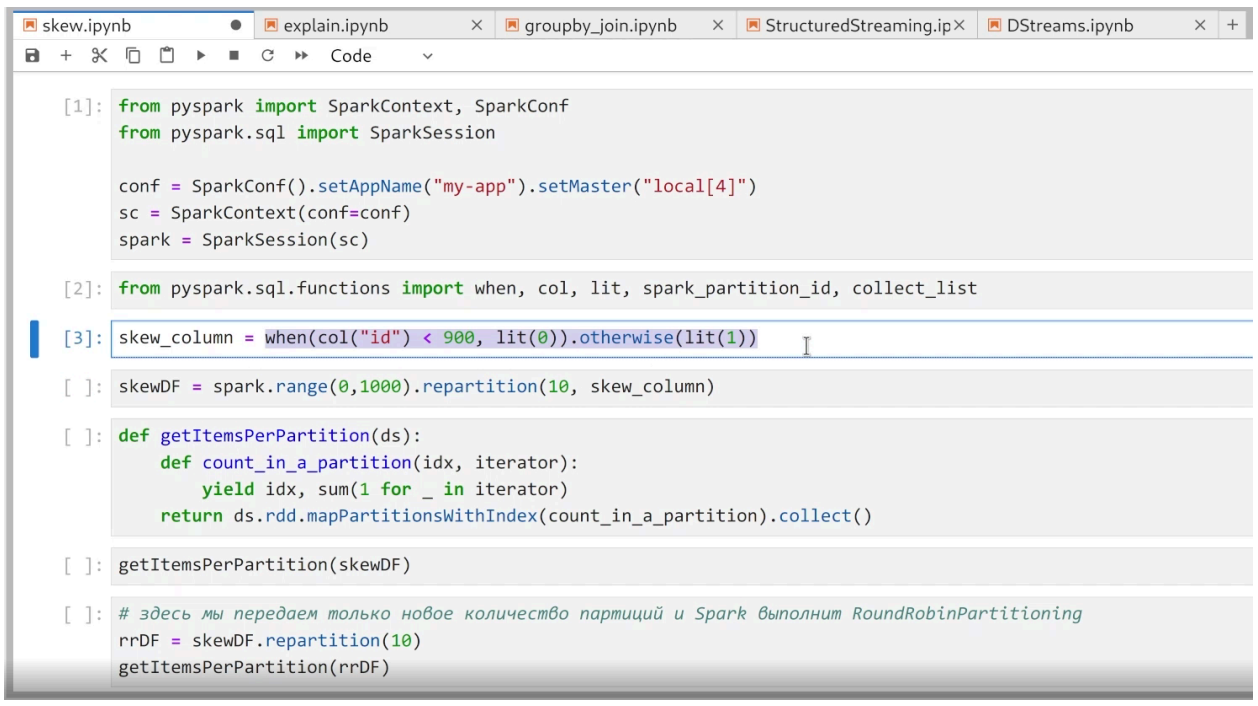
Предположим, что есть датасет, который крайне плохо распределен по кластеру:

```python
from pyspark.sql.functions import when, col, lit

def getItemsPerPartition(ds):
  def count_in_a_partition(idx, iterator):
    yield idx, sum(1 for _ in iterator)
  return ds.rdd.mapPartitionsWithIndex(count_in_a_partition).collect()


 skew_column = when(col("id") < 900, lit(0)).otherwise(lit(1))
skewDF = spark.range(0,1000).repartition(10, skew_column)
getItemsPerPartition(skewDF)
```

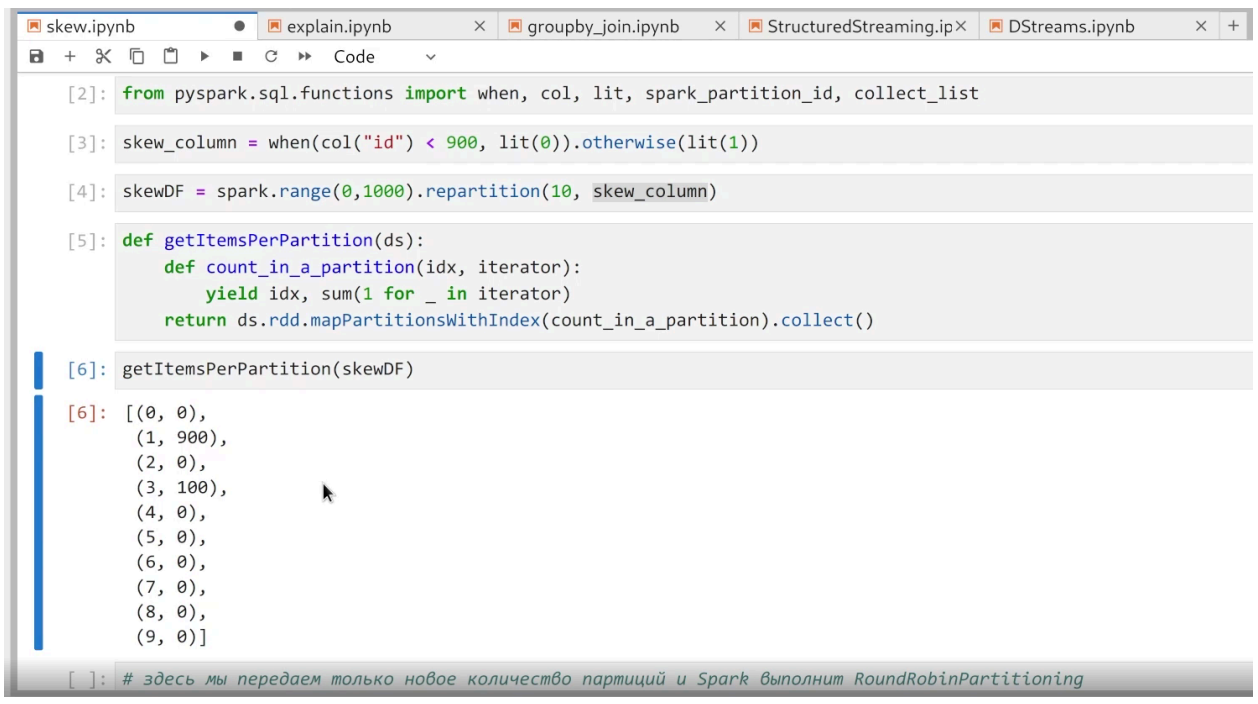
У нас есть и Spark, и функции. Мы делаем dataframe, вызываем repartition и говорим «сделай 10 партиций, но при распределении используй эту колонку»:



Часто там может быть ID и т.д.

Если посмотреть на то, по какому условию мы делаем партиционирование, то увидим, что для чисел от 0 до 1000 значения меньше 900 попадут в одну партицию, а остальные – в другую, так как в колонке два значения. Чувствуется, что уже что-то не так.

Есть функция, которая называется «**mapPartitionsWithIndex**». Мы проходимся по всем партициям и смотрим, сколько записей в каждой партиции оказалось. Мы видим, что несмотря на то, что партиций 10, данные оказались в двух:



Причем в партиции «1» окажется 900 записей, а в партиции «3» – 100. Это крайне нежелательный случай!

Виды партиционирования – RoundRobin

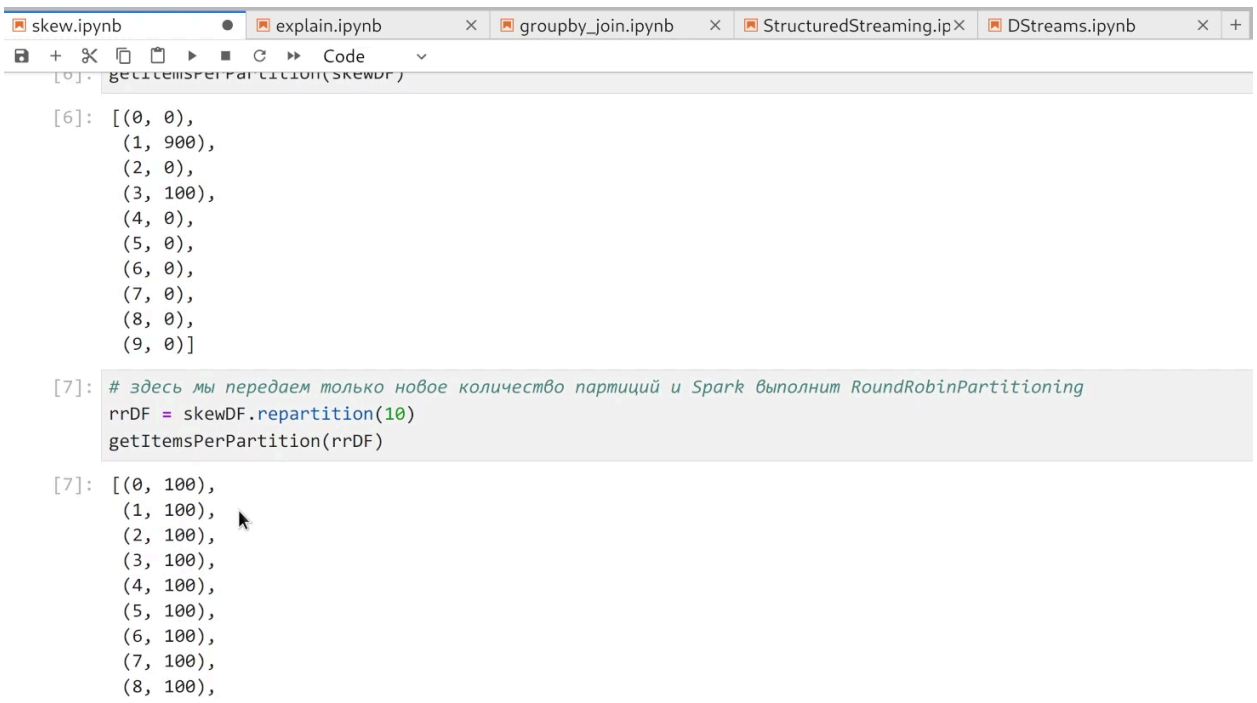
Если мы не будем указывать колонку, а просто скажем «сделай repartition на 10 партиций», то

Дедлайн 07 июля, 23:59 Мск

```
# здесь мы передаем только новое
# количество партиций и Spark выполнит
# RoundRobinPartitioning
rrDF = skewDF.repartition(10)
getItemsPerPartition(rrDF)
```

RoundRobinPartitioning – это подход, при котором Spark просматривает все значения в партициях и каждое следующее значение перемещает на следующую партицию.
RoundRobinPartitioning может быть эффективен, но **он ничего не знает о данных.** Он пересылает **очень** много данных горизонтально.

Выполним код:



RoundRobin’ом данные раскидались.

Виды партиционирования – HashPartitioning

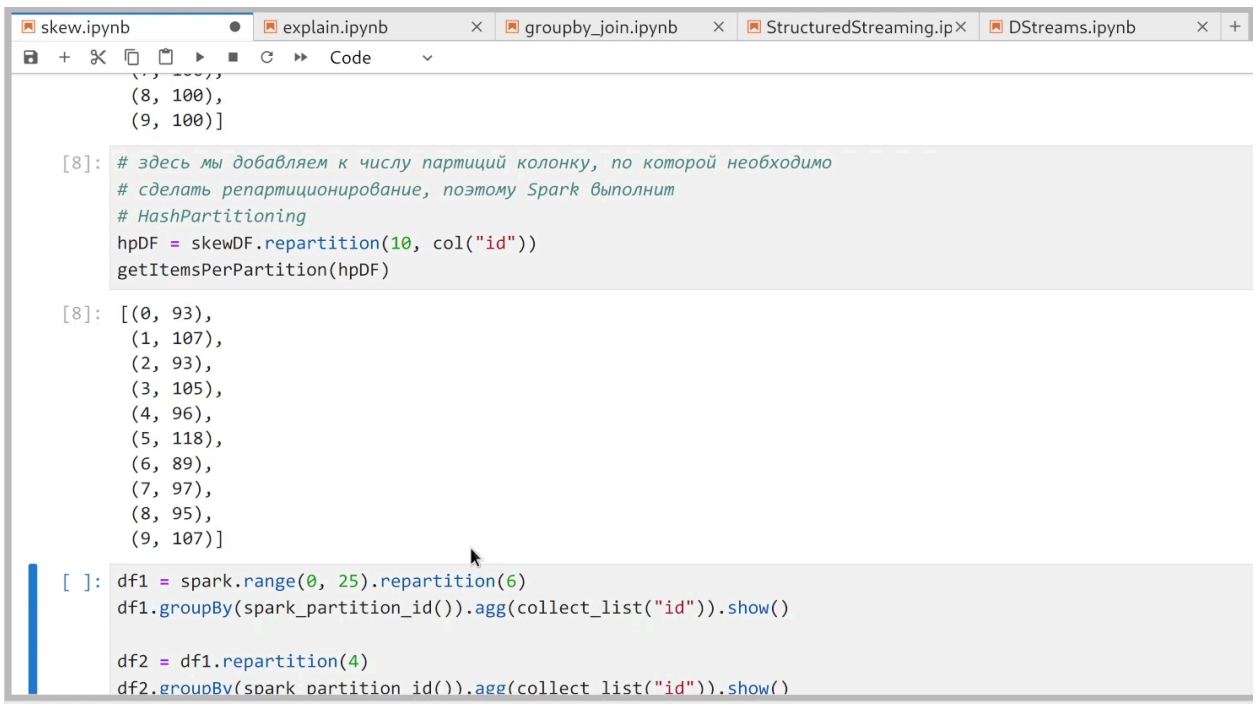
Как мы говорили ранее, имя колонки определяется по значениям. На самом деле там используется тип Partitioning’a, который называется «**HashPartitioning**». От значений, которые есть в колонке, считается хэш. На основе этого хэша принимается решение в какую партицию какие данные закинуть:

```
# здесь мы добавляем к числу партиций колонку, по которой необходимо
# сделать репартиционирование, поэтому Spark выполнит
# HashPartitioning
hpDF = skewDF.repartition(10, col("id"))
getItemsPerPartition(hpDF)
```

```
[(0, 93),
 (1, 107),
 (2, 93),
 (3, 105),
 (4, 96),
 (5, 118),
 (6, 89),
 (7, 97),
 (8, 95),
 (9, 107)]
```

Если мы запустим партиционирование по этой колонке, указав ее, то выполнится HashPartitioning.

Иногда могут быть коллизии, и распределение данных по кластеру будет не таким равномерным. Если мы знаем, что кусочки датасетов, относятся к одному и тому же значению в этой колонке, будем процессить независимо. Подобный метод партиционирования может быть более эффективным при расчетах.



Другой вариант – coalesce

Когда данные распартиционированы, мы хотим их переложить. Это можно сделать двумя способами:

- **Repartition**, когда речь идет об увеличении партиций.
- **Coalesce**, когда речь идет об уменьшении партиций.

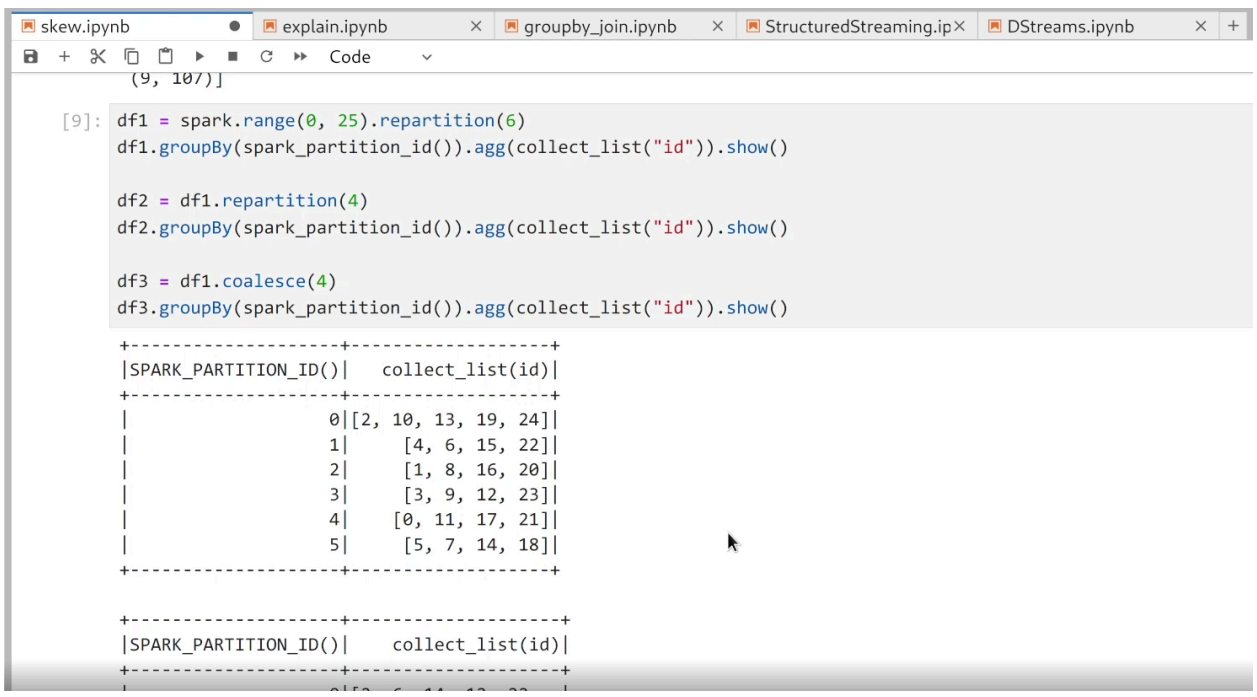
```
df1 = spark.range(0, 25).repartition(6)
df1.groupBy(spark_partition_id()).agg(collect_list("id")).show()

df2 = df1.repartition(4)
df2.groupBy(spark_partition_id()).agg(collect_list("id")).show()

df3 = df1.coalesce(4)
df3.groupBy(spark_partition_id()).agg(collect_list("id")).show()
```

Coalesce пытается минимизировать дополнительные пересылки данных на кластере.

Рассмотрим пример:



Мы видим данные, раскиданные по шести партициям. Вызываем базовый Repartition. Как мы помним, происходит RoundRobinPartitioning, и данные перекладываются по всему кластеру как получится. Если же вызвать Coalesce, мы увидим, что партиций стало меньше, но при этом структура партиций осталась. Некоторые сегменты просто объединились между собой. Например, две партиции лежат на одном воркере, а мы уменьшаем количество партиций. При вызове Coalesce есть серьезный шанс схлопнуть две партиции в одну, при этом дополнительной пересылки по кластеру не будет.

Нюанс: после выполнения Coalesce количество данных по партициям будет неравномерным.

«Соленые» данные

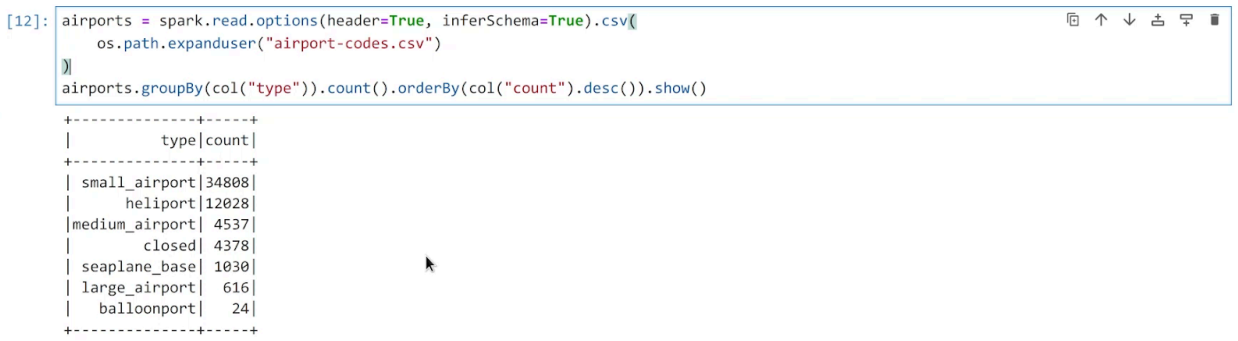
Дедлайн 07 июля, 23:59 Мск

```
airports = spark.read.options(header=True, inferSchema=True).csv(
    os.path.expanduser("~/Experiments/slurm/airport-codes.csv")
)
airports.groupBy(col("type")).count().orderBy(col("count").desc()).show()
```

Мы должны получить следующую таблицу:

```
+--------------+-----+
|          type|count|
+--------------+-----+
| small_airport|34808|
|      heliport|12028|
|medium_airport| 4537|
|        closed| 4378|
| seaplane_base| 1030|
| large_airport|  616|
|   balloonport|   24|
+--------------+-----+
```

Все получилось:



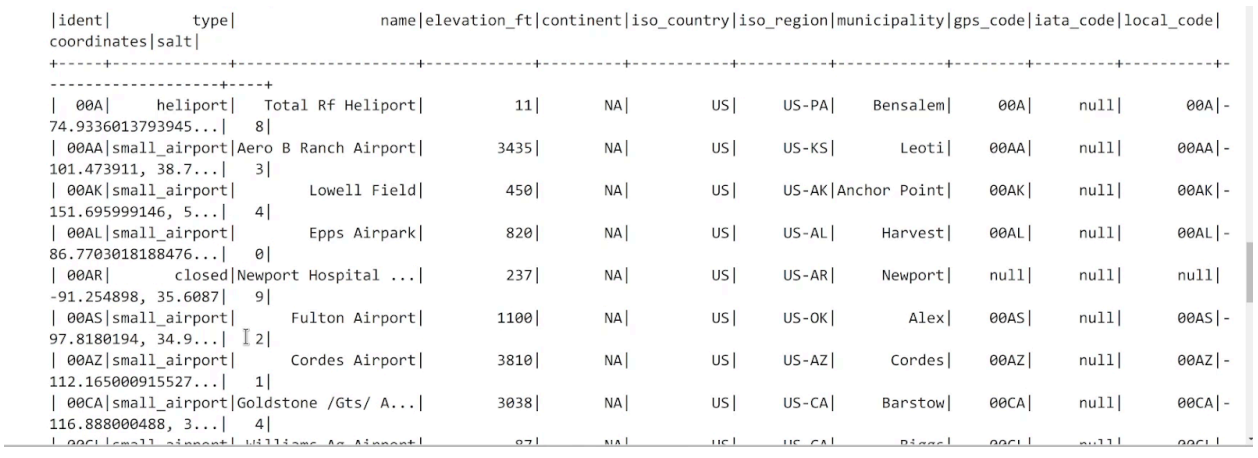
В ней мы видим **нюанс**: мы проводим группировку по колонке с типом. В колонке с типом большой разброс по количеству уникальных значений, например, маленьких аэропортов ≈35 000, а аэропортов для воздушных шаров – 24. Когда происходит подобное, есть риск, что при выполнении GroupBy, Spark может вылететь по памяти. Внутри происходит shuffle, и Spark репроцессионирует данные, пересылая данные, относящиеся к одному значению колонки, на одну ноду. Это одна из самых частых ошибок.

Для решения такой проблемы используется **подход «соление данных»**. Попробуем посчитать ту же задачу. Мы сделаем не одну группировку, а две.

Для начала придумаем колонку, в которой зададим распределение. Здесь примерно прикидываем:

Например, у нас будет 10 партиций. Мы заполняем колонку остатками от деления. Рандомное число умножаем на 100, округляем его до целого и получаем какое-то значение. Считаем остаток от деления на 10. У нас получается выдуманная «соленая» колонка.

Смотрим:



В этой колонке будут числа от 0 до 10. Сделаем первый groupBy не по колонке «Type», а по двум колонкам: «Type», «Salt». Это приведет к тому, что данные по кластеру будут группироваться не

Дедлайн 07 июля, 23:59 Мск

```
[13]: from pyspark.sql.functions import pmod, round, rand, sum
```

```
[14...
# дополнительная колонка с солью
salt_mod_ten = pmod(round((rand() * 100), 0), lit(10)).cast("int")
salted = airports.withColumn("salt", salt_mod_ten)
# агрегируем более взвешенно
first_step = salted.groupBy(col("type"), col("salt")).count()
#second_step = first_step.groupBy(col("type")).agg(sum("count").alias("count"))
#second_step.show()
```

Вторым шагом мы сделаем второй groupBy, в котором уже сможем применить агрегат к колонке «Type». Поскольку данные связаны с конкретным типом, они уже независимо лежат на партиции. Есть серьезный шанс, что это увидит оптимизатор, вследствие чего дополнительной второй пересылки не будет. Мы вызываем ту же операцию, которую считали выше:

```
# дополнительная колонка с солью
salt_mod_ten = pmod(round((rand() * 100), 0), lit(10)).cast("int")
salted = airports.withColumn("salt", salt_mod_ten)
# агрегируем более взвешенно
first_step = salted.groupBy(col("type"), col("salt")).count()
second_step = first_step.groupBy(col("type")).agg(sum("count").alias("count"))
#second_step.show()
```

Получаем тот же результат, но из-за группировки в два этапа мы не вылетим по памяти:

```
[15]: # дополнительная колонка с солью
salt_mod_ten = pmod(round((rand() * 100), 0), lit(10)).cast("int")
salted = airports.withColumn("salt", salt_mod_ten)
# агрегируем более взвешенно
first_step = salted.groupBy(col("type"), col("salt")).count()
second_step = first_step.groupBy(col("type")).agg(sum("count").alias("count"))
second_step.show()
```

```
+--------------+-----+
|          type|count|
+--------------+-----+
| large_airport|  616|
|   balloonport|   24|
| seaplane_base| 1030|
|      heliport|12028|
|        closed| 4378|
|medium_airport| 4537|
| small_airport|34808|
+--------------+-----+
```

Подытог

Партиционирование – важный аспект распределенных вычислений, от которого напрямую зависит стабильность и скорость вычислений.

В Spark работает правило **1 TASK=1 THREAD= 1 PARTITION**

Репартиционирование и соление данных позволяет решить проблему перекоса данных и вычислений.

Важно помнить, что репартиционирование использует дисковую и сетевую подсистемы – обмен данными происходит по сети, а результат часто записывается на диск, что может стать узким местом при выполнении репартиционирования.