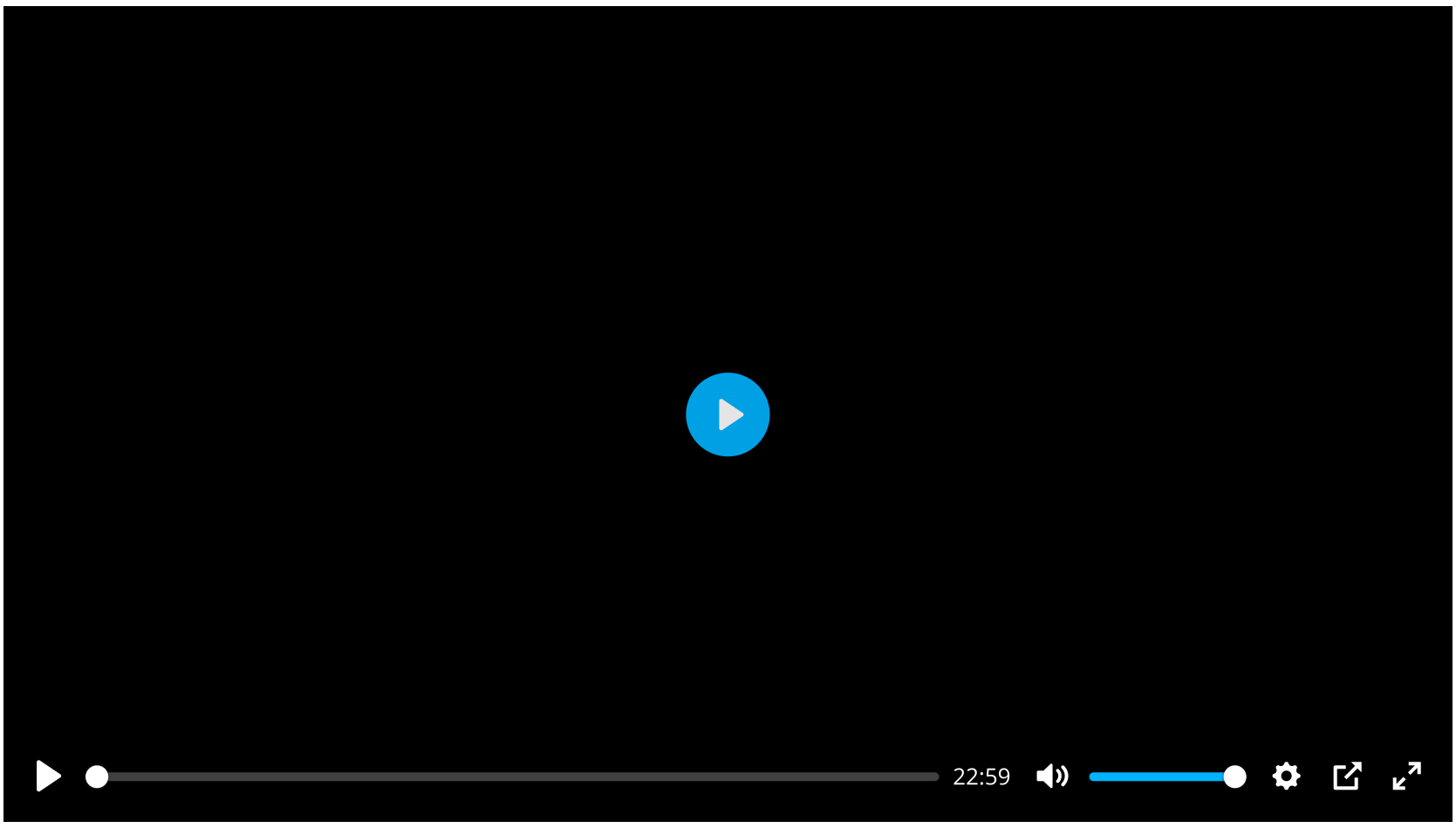
4.2 Что такое Spark и зачем он нужен DE. Введение в Resilient Distributed Datasets



Текстовая расшифровка видео:

RESILIENT DISTRIBUTED DATASETS

План:

- Базовое API;
- Что делать с RDD;
- Простейшие трансформации;
- Распределенка;
- Как все сломать;
- Стадии вычислений;
- Действия (action);
- Частичная сортировка и «разравнивание»;
- Промежуточные итоги.

Базовое API

У нас есть данные, которые мы закидываем на кластер. Эти данные нарезаются на «кусочки» – Partition’ы. На каждый Partition попадает порция из этих изначальных данных. Когда мы запускаем распределенные вычисления, каждый из этих кусочков обрабатывается независимо (если получается; если не получается, то можно использовать горизонтальные пересылки данных).

RDD – это датасет, «размазанный» по кластеру в виде Partition’ов.

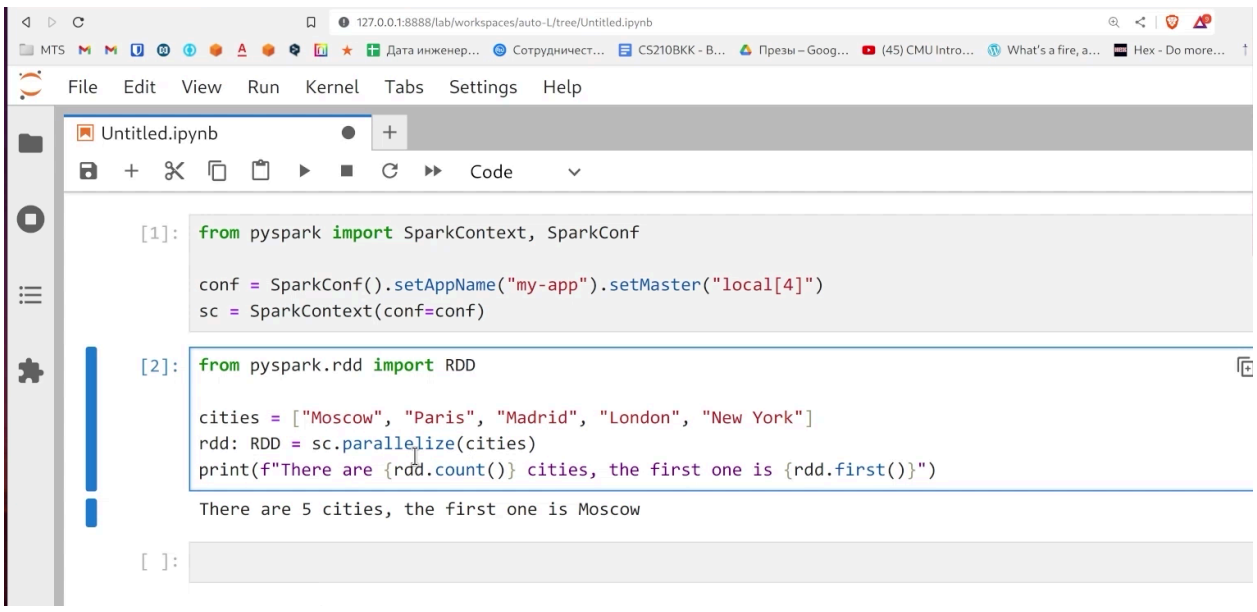
Рассмотрим код:

```
from pyspark.rdd import RDD

cities = ["Moscow", "Paris", "Madrid", "London", "New York"]
rdd: RDD = sc.parallelize(cities)
print(f"There are {rdd.count()} cities, the first one is {rdd.first()}")
```

Здесь специально добавлен импорт RDD для того, чтобы показать из каких вызовов возвращается объект типа RDD. Это важно, так как иногда может возвращаться RDD при запуске кода, а может возвращаться нечто иное (это довольно опасная ситуация).

Ранее мы создали notebook и запустили контекст. Теперь мы можем скопировать код, вставить в notebook и запустить:



Основной метод, который эту коллекцию превращает в RDD, – **sc.parallelize**.

SC – это SparkContext.

Когда вызываем sc.parallelize, мы получаем RDD. Поверх RDD можем вызывать разные методы для того, чтобы что-либо посчитать. Например, мы можем посмотреть, сколько всего объектов в этом RDD, достать первый объект и тд.

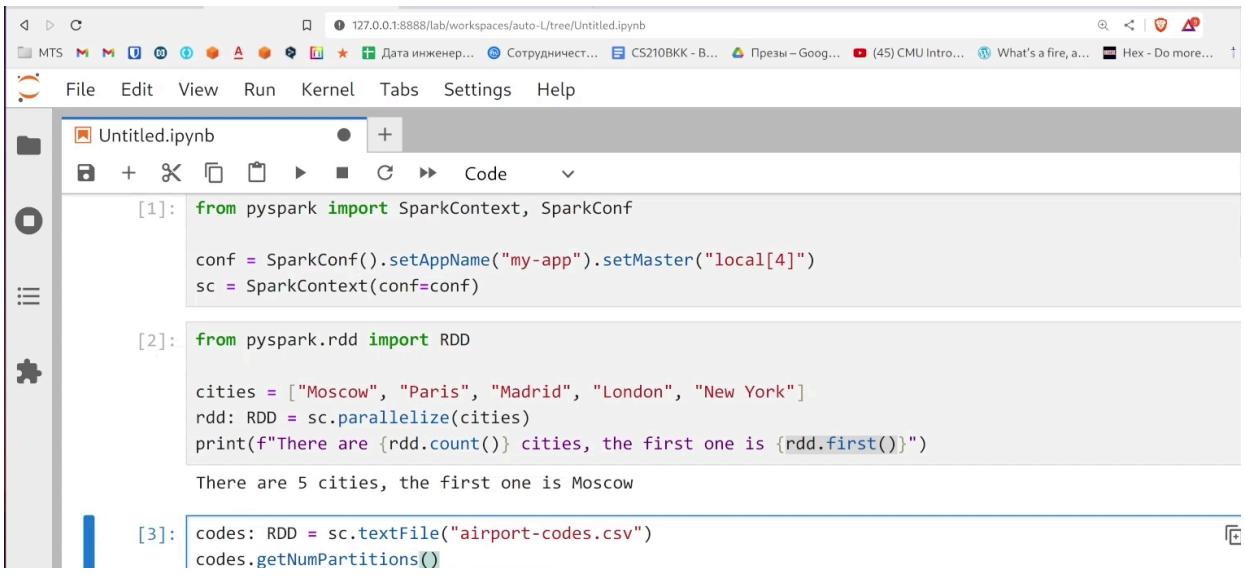
Если мы хотим залить данные не из стандартного контейнера Python, а из файла, например, HDFS, то можно использовать путь к файлу (можно его сначала прочитать, а потом «скормить»):

```
codes: RDD = sc.textFile("/tmp/data/airport-codes.csv")
codes.getNumPartitions()
# 2, может быть другим у вас
```

Можно использовать **sc.textFile**. Он прочитает нужный файл, а потом превратит его в RDD.

В качестве примера будем использовать тот же файл, который мы использовали на прошлых занятиях, а именно – коды аэропортов.

Метод **getNumPartitions** покажет на сколько «кусочков» разделился файл:



Дедлайн 07 июля, 23:59 Мск

Видим, что весь файл разбит на два Partition’a.

У нас есть переменная RDD с типом RDD из нашей распределенной коллекции, и есть codes, которые тоже RDD, но другого типа. Мы добавили данные в кластер. Что мы можем с ними сделать?

Что делать с RDD?

Есть два термина:

- Трансформация (Transformations);
- Действия (Actions).

Хитрость RDD:

Слово «Resilient» в аббревиатуре переводится как «устойчивый», «неизменный». Вспомним, что Spark написан на Scala с применением функционального подхода. Как мы знаем, все объекты неизменны. Здесь во многом используется тот же подход.

Трансформация – это набор методов, который мы можем применить к объекту RDD. Результатом выполнения мы не поменяем существующий RDD, а получим новый, который будет представлять из себя «ленивую» обертку.

У нас есть **способ вычисления**: для того, чтобы получить новый RDD на основе предыдущего, нужно к предыдущему применить какое-то преобразование.

Трансформации не запускают никакие дополнительные вычисления на кластере. Яркий пример: мы хотим привести все слова к верхнему регистру. Мы добавляем трансформацию, однако пока не будет запроса на развертывание графа вычислений, никаких реальных вычислений не будет.

Кратко о Трансформации: обычно ничего не считают, просто добавляют в граф вычислений. Результат – новый RDD.

Действия (Actions) используются для развертывания графа. Это другой набор вызовов, приводящий к тому, что цепочка трансформаций, написанная нами, раскручивается и схлопывается к какому-то результату.

Нюанс: это может быть опасно, так как при выполнении действия данные, которые мы преобразовали, посчитали и т.д. пересылаются на нашу машину, с которой мы запускаем обработку. Если этих данных много, то они могут упасть. **Принимайте решения взвешенно и осознавайте возможные последствия.**

Кратко о Действиях: раскручивают граф вычислений и запускают их распределённо. Результат зависит от того, что попросили.

Простейшие трансформации

Рассмотрим на примере:

```
def plus_one(i):
        return i + 1


plus_one = lambda i: i + 1
```

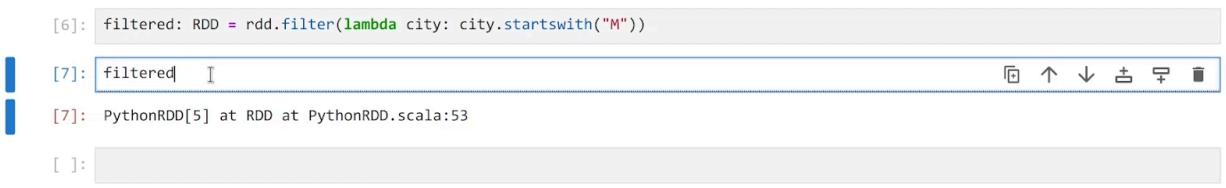
У нас есть RDD, к которому мы применяем примитивную функцию с помощью **map()/filter()**.

Map()/Filter() – это уже готовые и уже имеющиеся в стандартной библиотеке Python методы. С «**Map()**» мы «лениво» применяем одну и ту же функцию ко всем элементам. С помощью «**Filter()**» мы выкидываем те элементы, для которых предикат возвращает false, оставляя те, для которых он

Поработаем с данным кусочком кода:

```
filtered: RDD = rdd.filter(lambda city: city.startswith("M"))
```

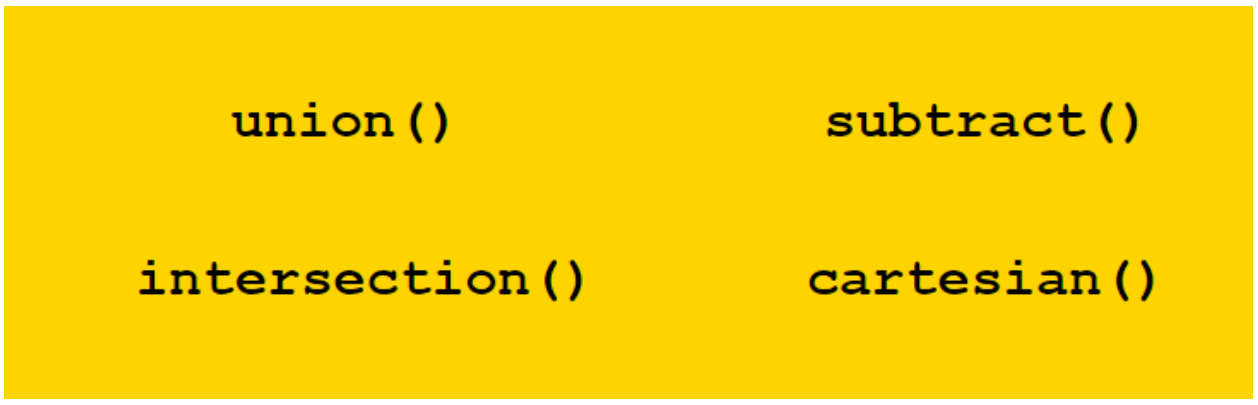
У нас есть города. Мы можем применить фильтр и отфильтровать те города, названия которых начинаются на букву «М». Обратите внимание, что вычисления не запустились:



Переменная «Filtered» все еще представляет из себя просто RDD и для того, чтобы получить результат, нужно вызвать action. Забегая наперед скажем, что можно вызвать collect (самый базовый, но и самый опасный action, вызывающий раскручивание всей цепочки и возврат всего результата). Если мы вызовем collect, у нас вернутся два города, начинающихся на букву «М»:

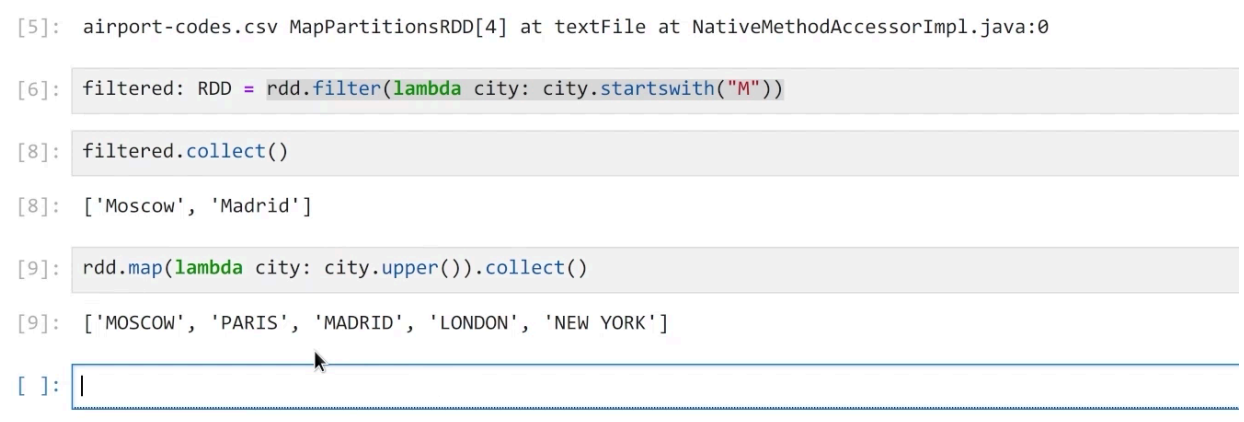
Помимо базовых операций map()/filter() есть большое количество других, которые тоже представляют собой трансформации, то есть никакие вычисления на кластере по умолчанию не запускают. Например:

- **Distinct()**, когда мы выкидываем все дубликаты;
- **Sample()**, когда мы берем по определенной логике из всех элементов какую-то подгруппу;
- **Операции по работе с сетами** (с множествами), когда мы считаем объединение множеств, пересечение множеств и т.д.:



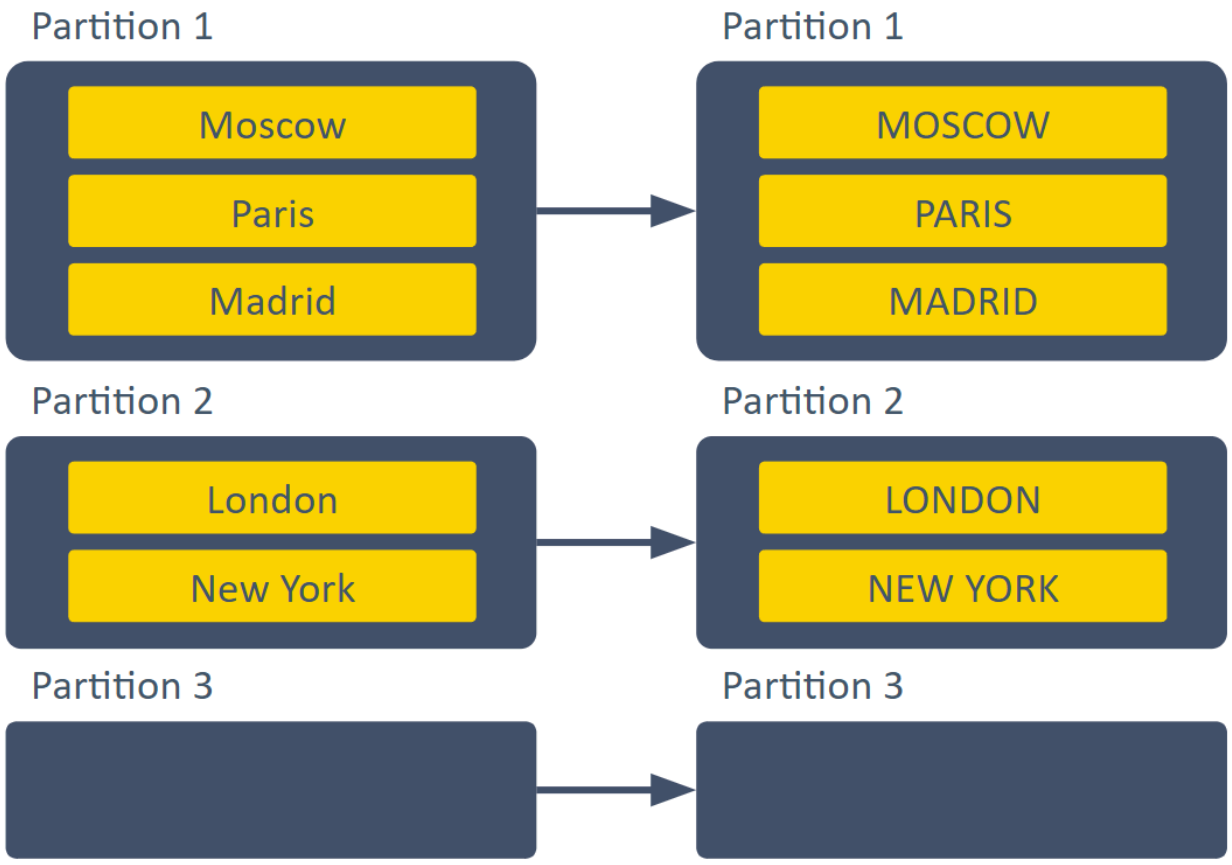
Распределенка

Если мы хотим в примере привести города к верхнему регистру, то в коде мы можем сделать следующее:



Когда мы вызвали parallels, данные по кластеру уже были «размазаны». Мы предположили, что три города из пяти попали в один partition, два других – в другой:

Дедлайн 07 июля, 23:59 Мск



В зависимости от того, как настроено биение по partitions, partition может создаваться больше, часть из которых могут быть пустыми (ничего страшного в этом нет, они не будут считаться).

Поскольку операция «to upper» довольно простая, мы, вызывая action, данные никуда не пересылаем.

Как все сломать

Представим, что где-то в коде возникла ошибка. Например, мы делим на ноль:

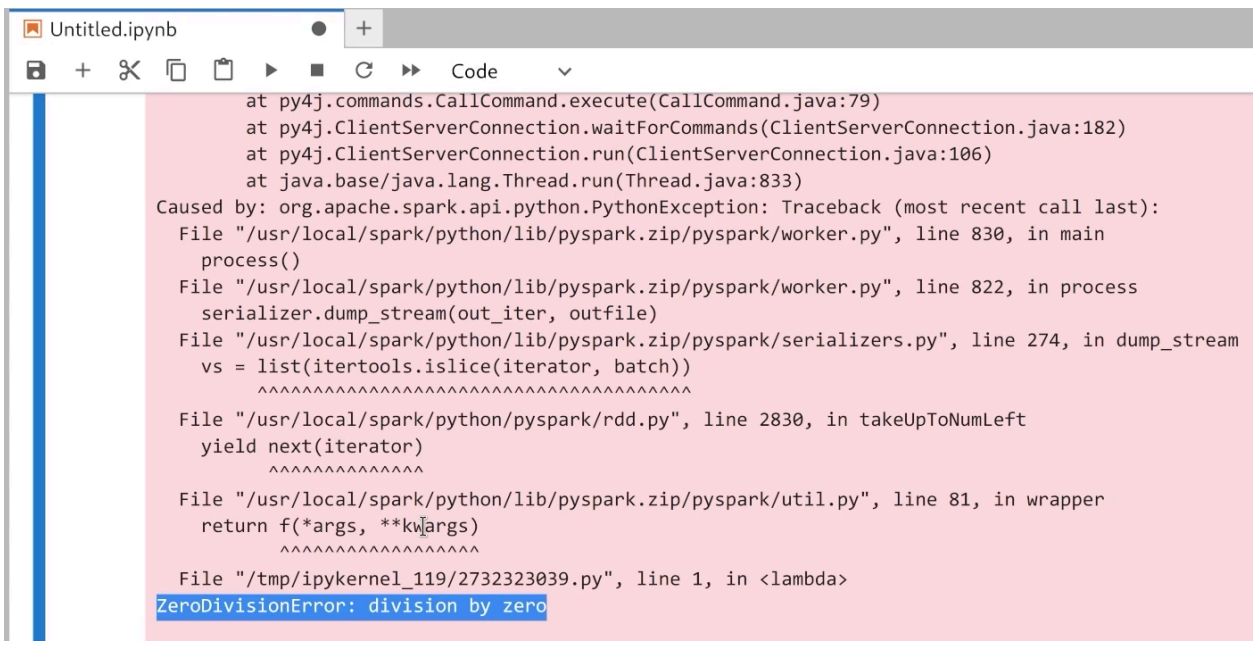
```
[ ]: a: RDD = sc.parallelize([1,2,3,4,5]).map(lambda num: num / 0)
```

Нюанс: поскольку map – «ленивая» трансформация, мы не узнаем об ошибке пока не вызовем action, так как ошибка возникнет в процессе вычислений.

Для того, чтобы все упало, достаточно вызвать action, сказав «.first». Он попытается прочитать первый элемент из RDD, закинет его в lambla, та попытается разделить на ноль и упадет с ошибкой:

```
rdd.map(lambda i: i / 0).first()
```

Если мы сделаем «a.first», то получим следующее:



Стадии вычислений

Поскольку мы никогда не меняем начальный RDD, вычисления могут быть не линейными, а разрозненными (подсчет с разных сторон). Вычисления будут представлять из себя граф (дерево).

Обратим внимание на пример:

Дедлайн 07 июля, 23:59 Мск

```python
# Трансформация map: не запускает вычислений,
# не изменяет изначальный RDD
upper_rdd: RDD[str] = rdd.map(lambda city: city.upper())
# Метод take возвращает N первых элементов RDD
upper_lst: list = upper_rdd.take(3)
old_lst: list = rdd.take(3)
print(f"New RDD: {','.join(upper_lst)}")
# New RDD: MOSCOW, PARIS, MADRID
print(f"Old RDD: {','.join(old_lst)}")
# Old RDD: Moscow, Paris, Madrid
```

Это иллюстрация в коде, где мы приходим к верхнему регистру и берем три элемента в новом RDD, который уже «лениво» посчитан. Нам ничего не мешает достать старый RDD, спросить у него. Там все также будут неизмененные элементы.

Нюанс: стадии по умолчанию будут пересчитываться.

Действия (actions)

Когда мы хотим получить какой-либо результат, мы хотим схлопнуть RDD к какому-то конкретному значению и переместить данные на основную ноду, на которой мы все запускаем.

Одним из типичных actions является **reduse()**, который вынесли в functools:

```python
from functools import reduce
reduce(lambda a, b: a + b, [21, 7, 14])
```

Он берет элемент и аккумулятор по умолчанию и применяет функцию до тех пор, пока не схлопнет весь массив. Каждый раз **в первый аргумент попадает следующий элемент из изначального контейнера, а во второй – результат применения предыдущей операции.**

В Spark мы можем использовать action reduse(). Обратите внимание, что **он возвращает уже не RDD, а int:**

```python
# Действие reduce применяет функцию f к промежуточному результату
# от предыдущей итерации со следующим элементом коллекции
count: int = rdd.map(lambda x: len(x)).reduce(lambda a, b: a + b)
print(f"The RDD contains {count} letters")
# The RDD contains 31 letters
```

Таким образом мы пройдемся по всем городам, у каждого посчитаем длину и посмотрим общее количество символов в датасете:

```
[12]: count: int = rdd.map(lambda x: len(x)).reduce(lambda a, b: a + b)
      print(f"The RDD contains {count} letters")

      The RDD contains 31 letters
```

Action’ов не так много, но все они достаточно простые, так, например, take() берет один или несколько элементов, стараясь не раскручивать по максимуму граф вычисления для остальных.

Count() – это внезапный подсчет количества элементов. Это тоже action, так как в случае с RDD мы можем не знать точное количество записей.

Top() – это action, позволяющий доставать элемент по функции сравнения.

Частичная сортировка и «разравнивание»

TakeOrdered – комбинация из sort и take:

```
sc.parallelize(
  [10, 1, 2, 9, 3, 4, 5, 6, 7]
).takeOrdered(6, key=lambda x: -x)
# [10, 9, 7, 6, 5, 4]
```

Рассмотрим пример:

```
mapped_rdd = rdd.map(lambda x: list(x))
print(mapped_rdd.take(2))
# [['M', 'o', 's', 'c', 'o', 'w'], ['P', 'a', 'r', 'i', 's']]
flatmapped_rdd = rdd.flatMap(lambda x: x.lower())
print(flatmapped_rdd.take(4))
# ['m', 'o', 's', 'c']
```

Представим, что каждый элемент RDD – список. Нас интересует возможность пройтись по всем элементам, распаковав все списки. Это позволяет сделать специальная версия map'a – **flatMap**, он разравнивает промежуточные результаты.

Мы берем датасет с городами, каждый город мы разбиваем по буквам. Мы можем захотеть составить RDD, которая будет представлять из себя коллекцию букв в нижнем регистре. Мы применяем flatMap, вызываем метод приведения к нижнему регистру и получаем набор RDD, где каждый элемент – буква.

Промежуточные итоги

RDD – неизменяемый (иммутабельный) распределенный набор данных.

Трансформации «лениво» создают новый RDD и не изменяют существующий.

Только **действия** приводят к запуску реальных вычислений.

Как вам урок? 😠 😐 🔥

Изучил, далее ›

Дедлайн 07 июля, 23:59 Мск