СЛЁРМ Обучение для инженеров и технических лидеров

Дата-инженер

Что такое Spark и зачем он нужен DE. Введение в RDD

Николай Марков

Цели урока. Что вы узнаете:

1. В чем плюсы Spark’а и как он работает
2. Что такое RDD и зачем их использовать
3. Что такое трансформации (transformations) и действия (actions)
4. Решение базовых задач с использованием RDD

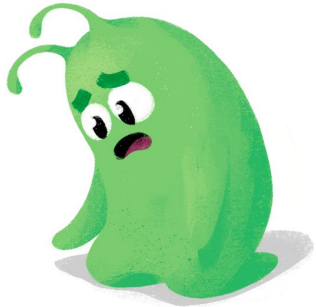
Зачем нужен Spark

Проблемы с MapReduce

Вместо написания бизнес-логики приходится концентрироваться на том, как ее натянуть на MapReduce

Все промежуточные результаты вычислений сохраняются на диск, который намного медленнее памяти

Даже самые базовые операции (join, aggregate и т.д.) требуют написания руками довольно большого количества кода



Что предлагает Spark

Проблема упрощения кода для **параллельных вычислений** актуальна как **в рамках одной машины**, так и **на гигантских кластерах**

Иногда данные поступают **потоком** и надо реагировать на них **как можно быстрее**

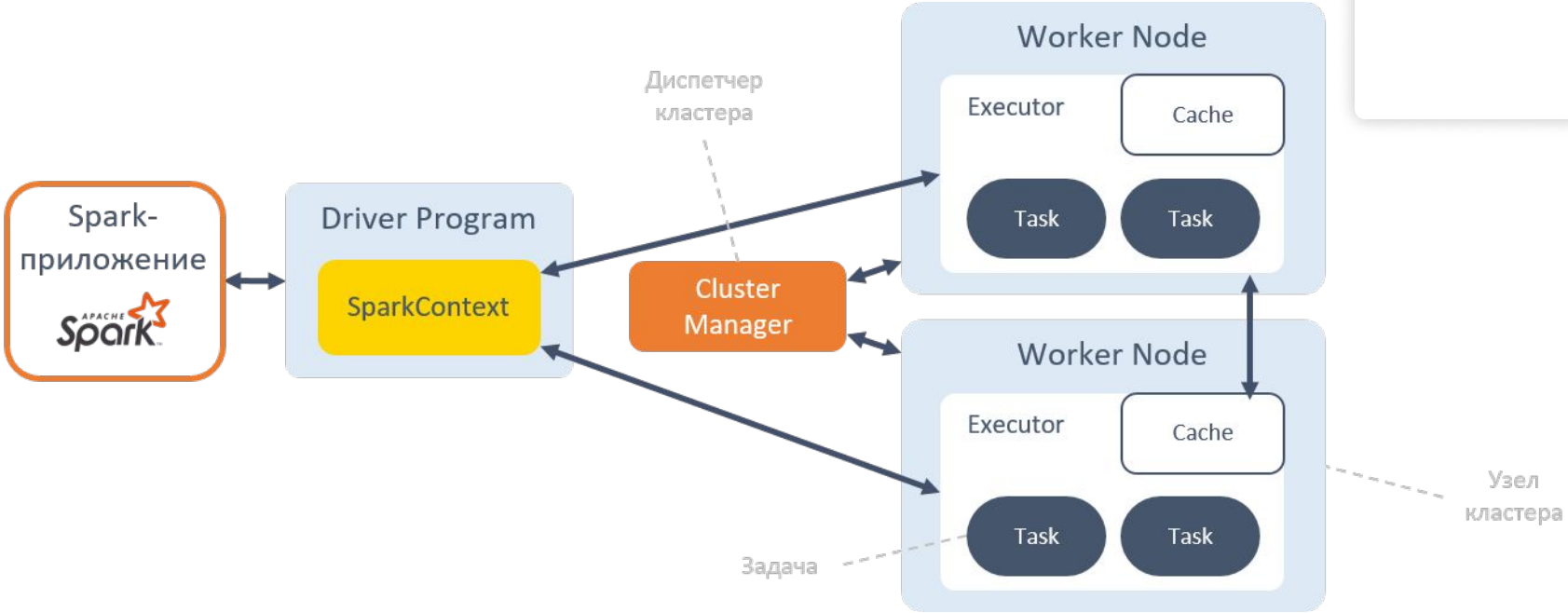
Требуется удобно взаимодействовать с **разными источниками и приемниками**

Как насчет продвинутых **средств для отладки** и оптимизированной **поддержки SQL?**

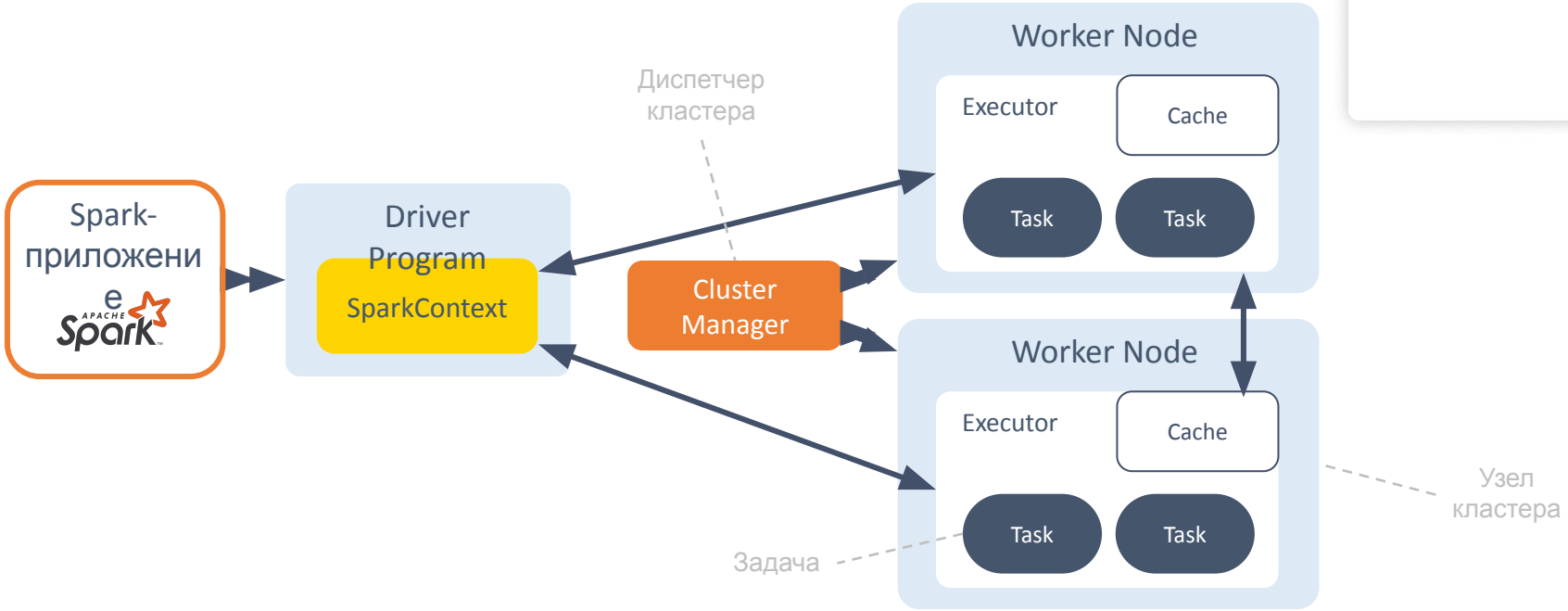
Расширенная **стандартная библиотека** и поддержка дополнительных плагинов для **работы с графами** и **машинного обучения**

Как устроен Spark

Упрощенная схема работы



Упрощенная схема работы



Основные компоненты

Driver

- Управляет информацией о приложении
- Откликается на команды пользователя
- Анализирует и распределяет работу для Worker’ов

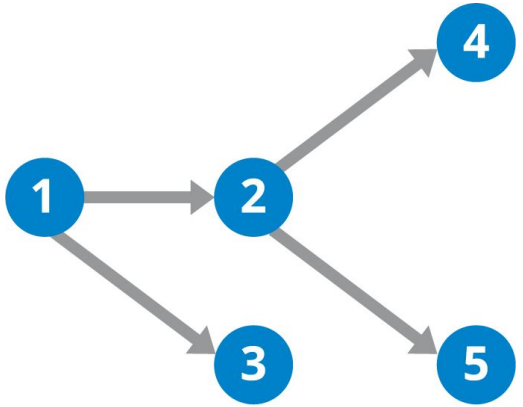
Worker(s) aka Executor(s)

- Обрабатывают данные
- Получают задачи не напрямую, а через Driver
- Работают с одним или несколькими **Partition**’ами

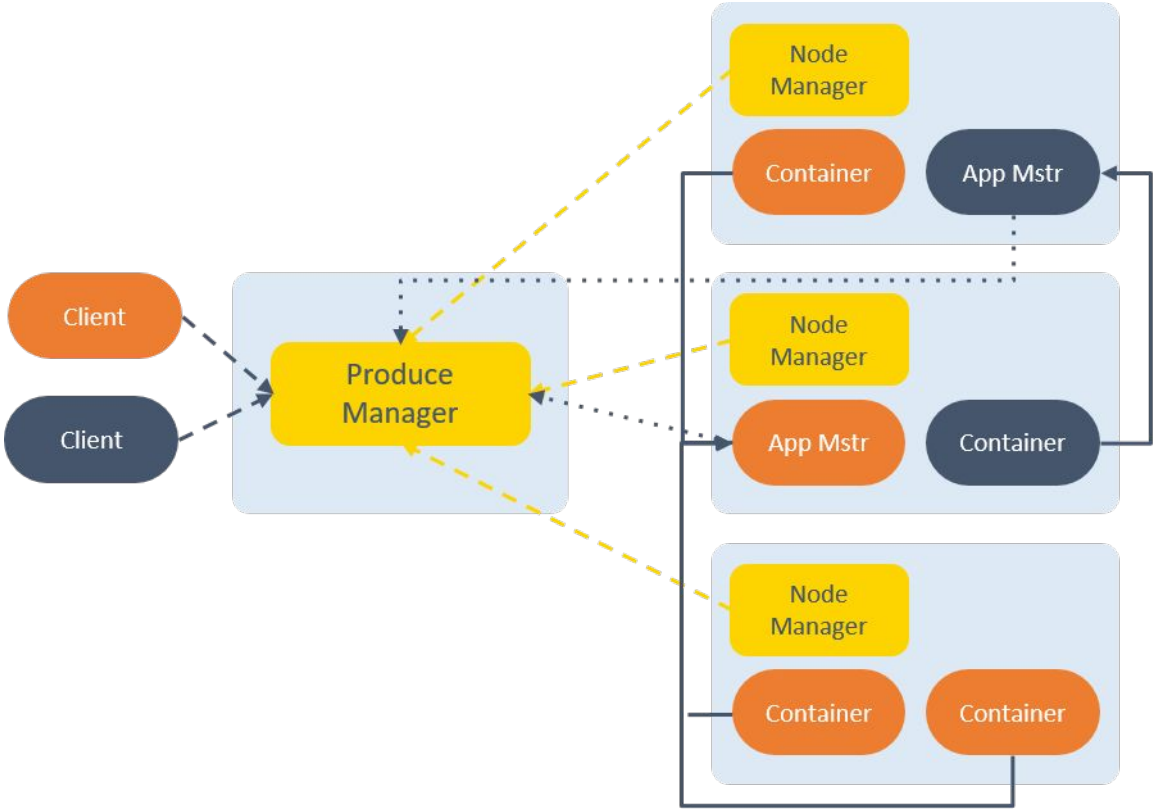
Логистический граф

- **Driver** строит направленный ациклический граф (DAG) из задач на основе кода пользователя
- Но проблема не только в параллелизме, но и в балансировке задач на **Executor**'ах
- Проще говоря, при равном объеме работы на всех исполнителях кластер будет работать эффективнее всего
- Это касается и кода, и данных

[Закон Амдала]

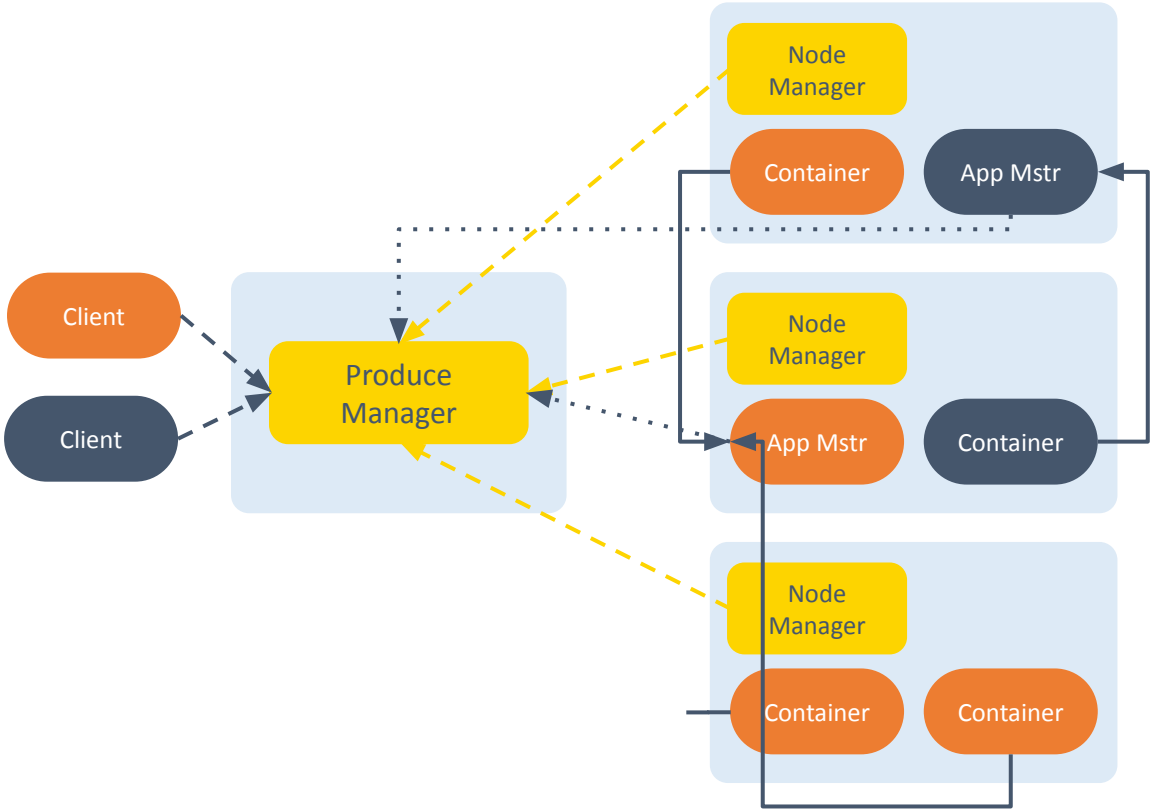


Менеджер кластера - YARN



- Отвечает за поиск ресурсов для запуска и Driver’а, и Worker’ов
- Квотирует ресурсы между пользователями
- Отслеживает статус выполнения

Менеджер кластера - YARN



- Отвечает за поиск ресурсов для запуска и Driver’а, и Worker’ов
- Квотирует ресурсы между пользователями
- Отслеживает статус выполнения

Начало работы

Самодостаточная система

Скачать с сайта, аналогично Hadoop

Установить пакет для Python:
~$ pip install pyspark

Jupyter в docker-образе:
~$ docker pull jupyter/pyspark-notebook
~$ docker run --name pyspark -p 8888:8888 -p 4040:4040 jupyter/pyspark-notebook

Запуск приложений

spark-shell / pyspark

spark-submit / standalone

http://127.0.0.1:4040/jobs/

С точки зрения кода

SparkContext и **SparkSession**

- Появился в Spark 1.x
- Исторически использовался для создания RDD и размазывания данных по кластеру
- Доступен в spark-shell, как **sc**

- Появилась в Spark 2.x
- Фактически заменила собой большую часть SparkContext
- Включает в себя обертки для **Hive**, streaming'а и прочего
- Позволяет работать как с RDD, так и с DataFrame/Dataset
- Доступна в spark-shell, как **spark**

```
1. from pyspark import SparkContext, SparkConf
2.
3. conf = SparkConf().setAppName("my-app").setMaster("local[4]")
4. sc = SparkContext(conf=conf)
```

Источники, приемники, форматы

Local FS

HDFS S3

JDBC

Redis Elasticsearch

MongoDB

CSV

JSON XML

RCFile

Parquet ORC

Avro

Resilient Distributed Datasets

Базовое API

RDD — это просто датасет, «размазанный» по кластеру в виде **Partition**'ов

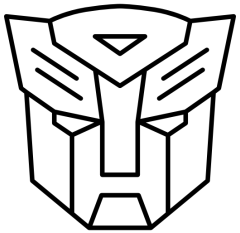
```
1. from pyspark.rdd import RDD
2.
3. cities = ["Moscow", "Paris", "Madrid", "London", "New York"]
4. rdd: RDD = sc.parallelize(cities)
5. print(f"There are {rdd.count()} cities, the first one is {rdd.first()}" )
```

Кроме локальной коллекции можно использовать файл как локальный, так и из S3 (*s3n://, s3a://*), HDFS (*hdfs://*) или, например, выборку из БД (через *JDBC*)

```
1. codes: RDD = sc.textFile("/tmp/data/airport-codes.csv" )
2. codes.getNumPartitions()
3. # 2, может быть другим у вас
```

(пример – [база кодов аэропортов])

Что делать с RDD?



Трансформации
(Transformations)

map, filter, etc.

Обычно ничего не считают, просто добавляют шаг в граф вычислений. **Результат** — новый RDD

Действия
(Actions)

reduce, collect, foreach, etc.

Раскручивают граф вычислений
и запускают их распределенно.
Результат зависит от того, что попросили

Простейшие трансформации

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

```
plus_one = lambda i: i + 1
```

map()

filter()

distinct()

sample()

union()

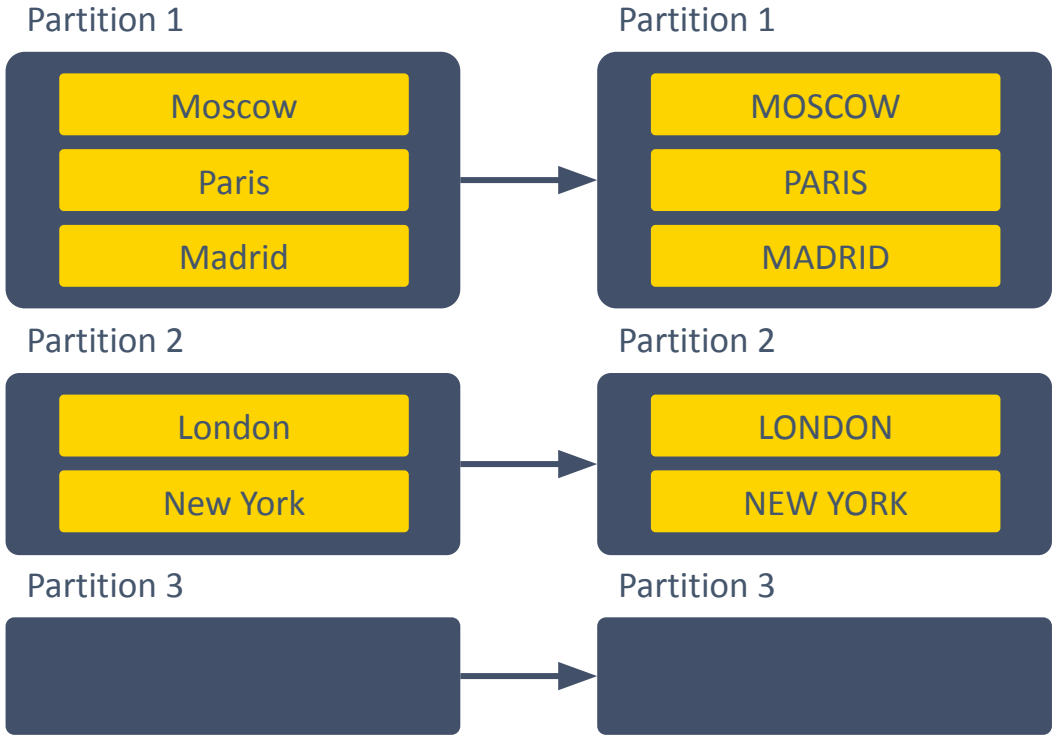
intersection()

subtract()

cartesian()

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

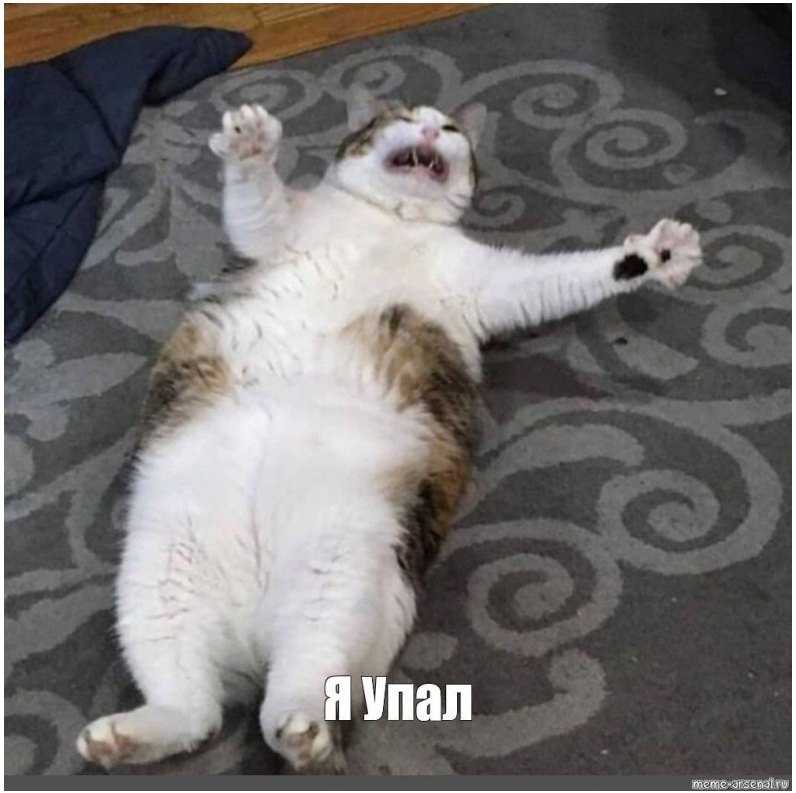
Распределенка



Как все сломать

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

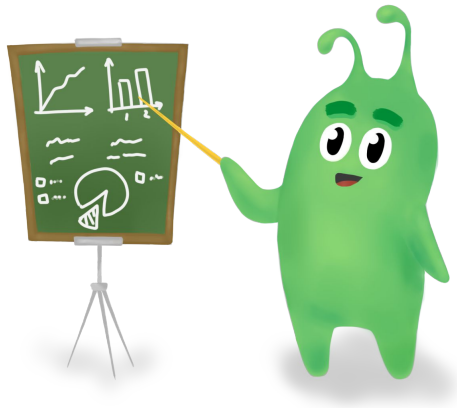
Как все сломать




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

Стадии вычислений

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



Действия (actions)

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

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

reduce() **collect()** **take()** **count()** **top()**

Выкачивает результат на Master-ноду!

Частичная сортировка и «разравнивание»

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

Распаковка вложенных контейнеров

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

Промежуточные итоги

RDD – неизменяемый (иммутабельный) распределенный набор данных

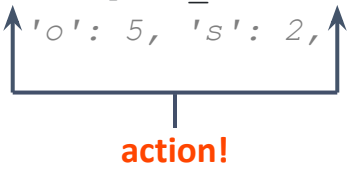
Трансформации (map, filter, flatMap etc.) «лениво» создают новый RDD и не изменяют существующий

Только действия (count, reduce, collect, take) приводят к запуску реальных вычислений

Ключи и значения

Старые-добрые ключи-значения

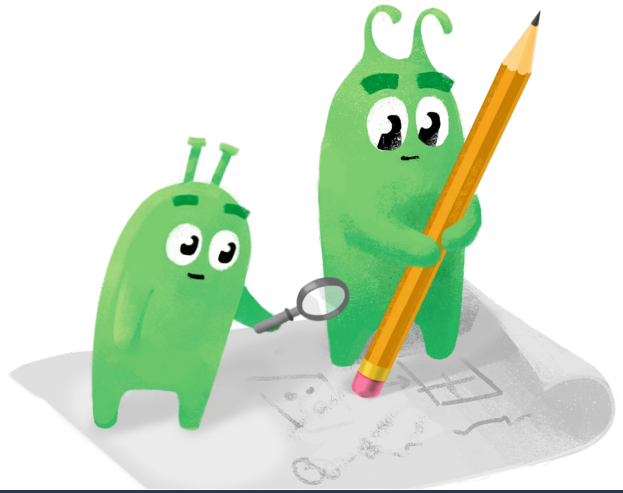
```
1. pair_rdd = rdd.flatMap(lambda x: x.lower()).map(lambda x: (x, 1))
2. print(pair_rdd.take(4))
3. # [('m', 1), ('o', 1), ('s', 1), ('c', 1)]
4. counts: dict = pair_rdd.countByKey()
5. # {'m': 2, 'o': 5, 's': 2, 'c': 1, 'w': 2, 'p': 1, 'a': 2, ... }
```




```
1. letter_count: RDD = pair_rdd.reduceByKey(lambda x, y: x + y)
2. print(letter_count.take(3))
3. # [('p', 1), ('d', 3), ('l', 1)]
```

WordCount

```
1. text_rdd = sc.textFile("Experiments/slurm/76-0.txt")
2. text_rdd.flatMap(
3.   lambda l: l.split(" ")
4. ).map(lambda w: (w, 1)).reduceByKey(
5.   lambda x, y: x + y
6. ).top(10, key=lambda i: i[1])
```



Join’ы

```
favorite_letters = ['a', 'd', 'o']
fav_let_rdd = sc.parallelize(favorite_letters).map(lambda x: (x,1))
joined = letter_count.leftOuterJoin(fav_let_rdd)
```

Почему бы не взять паттерн матчинг?

```
def get_favorite(letter_info):
  letter, (left_count, right_count) = letter_info
  match right_count:
    case None:
      return f"The letter {letter} is not my favorite"
    case _:
      return f"The letter {letter} is my favorite and appears in text {left_count} times"

joined.map(get_favorite).collect()
```

Итоги. О чем поговорили:

1. Как попробовать Spark у себя
2. Как устроено низкоуровневое API в лице RDD
3. В чем отличие между трансформациями (transformations) и действиями (actions)
4. Как делать распределенные операции с ключами и значениями

Спасибо
за внимание!