

# LSML #2

Apache Spark: RDD API, DataFrame API

# Apache Spark



Фреймворк для распределенных вычислений

API на многих языках: Scala, Java, **Python (PySpark)**

Внутри много всего: Spark ML, Spark SQL, Spark Streaming

# SQL join запрос

Таблица **a** – покупки пользователей (**user**, product, ...)

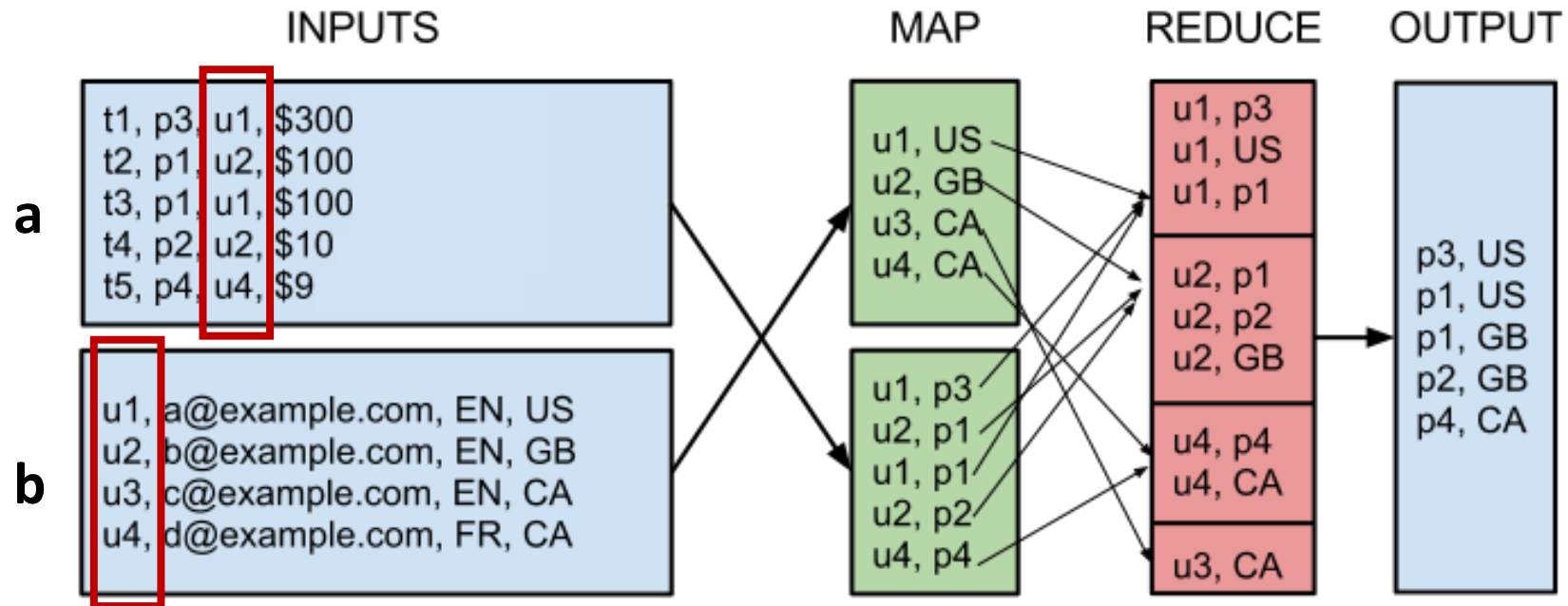
Таблица **b** – информация о пользователях (**user**, country, ...)

Хотим получить покупки продуктов по странам

Нужно сделать join по **user**

```
select
  a.product,
  b.country
from
  a join b on a.user = b.user
```

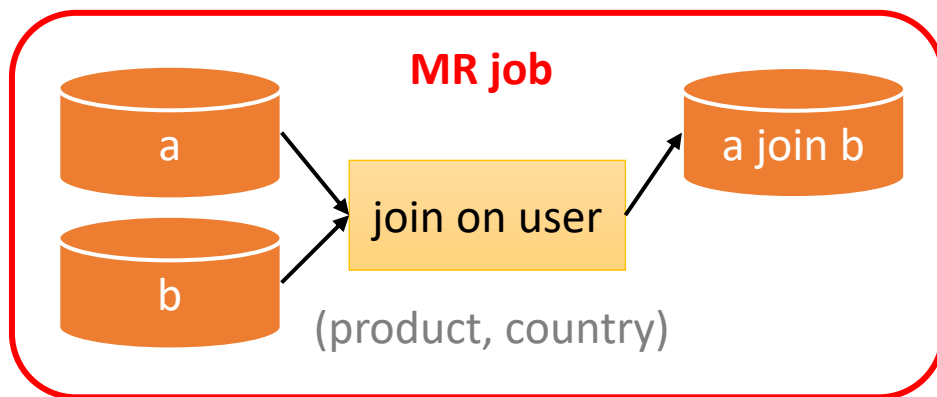
# SQL join на MapReduce



# Еще один join на MapReduce

В таблице **c** лежит информация о продуктах (**product**, category, ...)

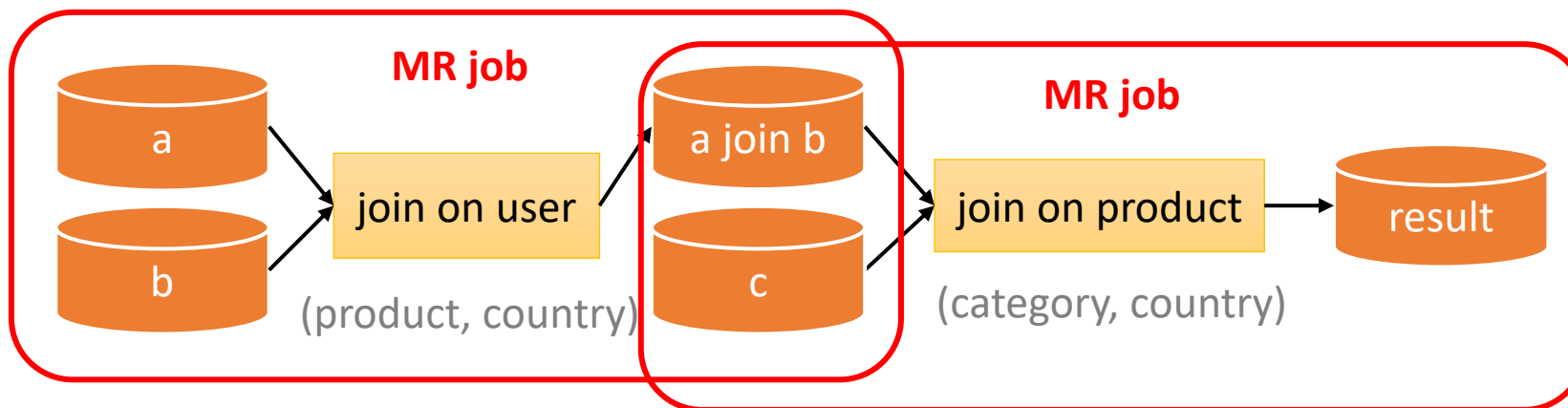
Два join на MapReduce:



# Еще один join на MapReduce

В таблице **c** лежит информация о продуктах (**product**, category, ...)

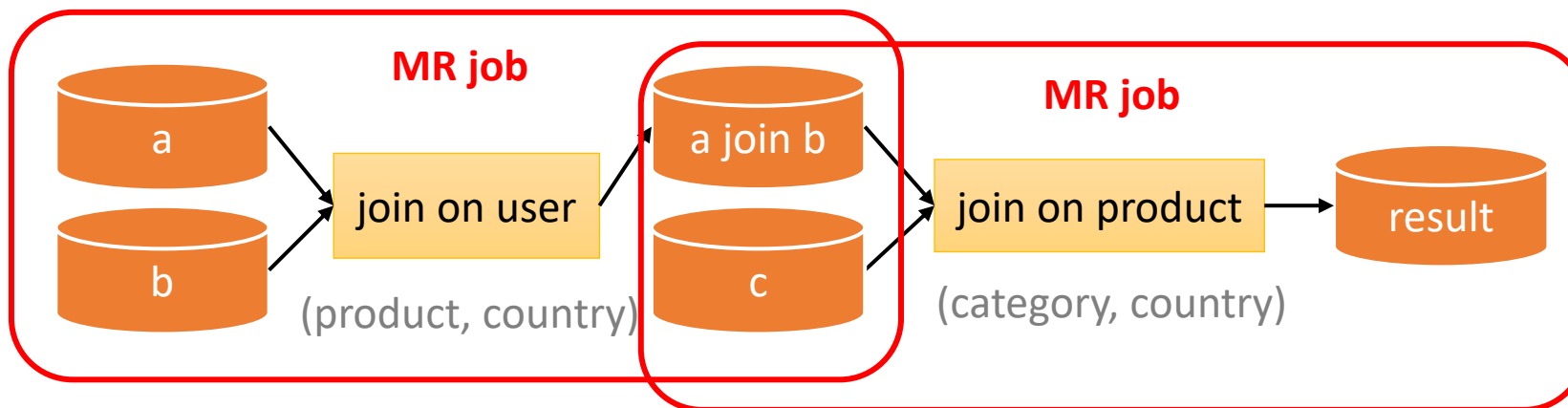
Два join на MapReduce:



# Еще один join на MapReduce

В таблице **c** лежит информация о продуктах (**product**, category, ...)

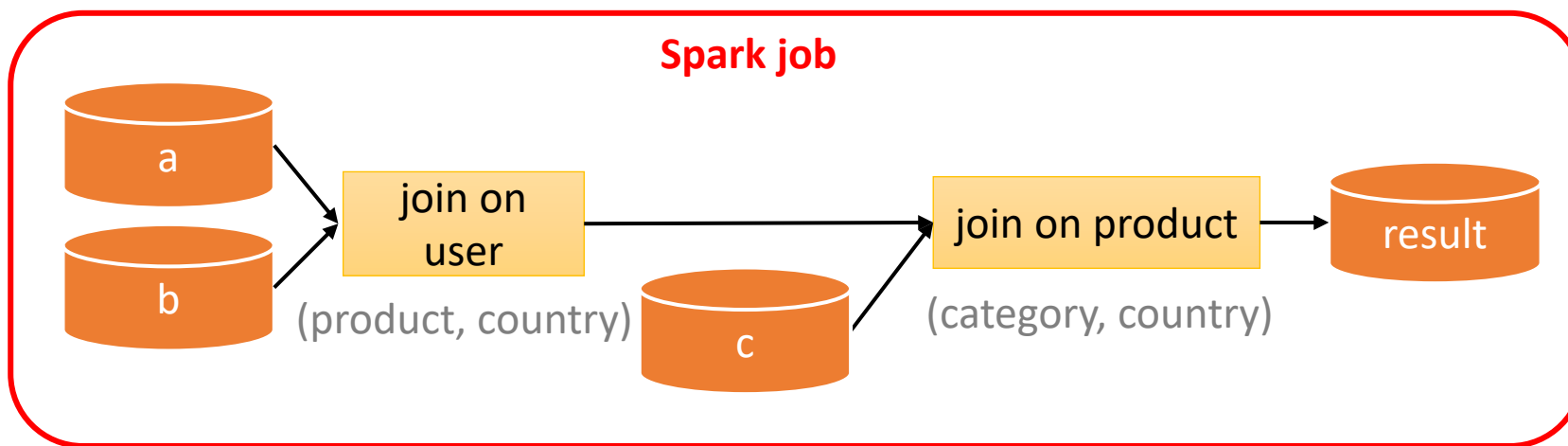
Два join на MapReduce:



- MapReduce хранит результаты в HDFS
- Поэтому для “a join b” мы тратим время на запись в HDFS и тут же читаем эти данные обратно

# Вот тут-то и поможет Spark

Два join на Spark:



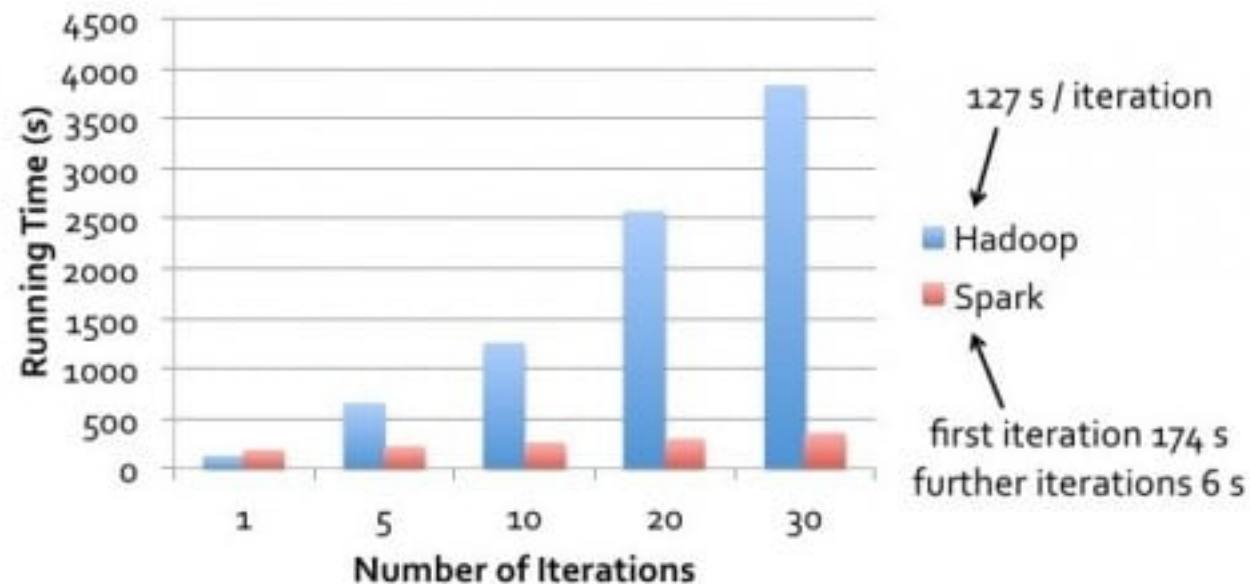
- Вычисления описываются как DAG (Directed Acyclical Graph)
- Промежуточные результаты хранятся в памяти или на диске минуя HDFS



# Итерационные алгоритмы

- В MapReduce есть overhead на запись/чтение в HDFS, запуск каждого шага в YARN
- В ML много итерационных алгоритмов и Spark идеально подходит для таких задач

## Logistic Regression Performance



# Spark vs MapReduce

	Spark	MapReduce
<i>Область применения</i>	Итерационные, интерактивные вычисления	Тяжелая пакетная обработка данных
<i>Простота использования</i>	Удобное API на Python	Hadoop streaming с неудобным интерфейсом
<i>Утилизация RAM</i>	Хранит данные в памяти, когда может	Все данные хранятся в HDFS

# Spark RDD API

В Spark вычисления описываются операциями над RDD

## **Абстракция RDD (resilient distributed dataset):**

Восстанавливаемый распределенный набор данных

Входы операций должны быть RDD

Все промежуточные результаты будут RDD. Так как известна цепочка вычислений (DAG) и потерянные части легко восстановить из входных данных.

# Spark RDD API

В Spark вычисления описываются операциями над RDD

## **Абстракция RDD (resilient distributed dataset):**

Восстанавливаемый распределенный набор данных

Входы операций должны быть RDD

Все промежуточные результаты будут RDD. Так как известна цепочка вычислений (DAG) и потерянные части легко восстановить из входных данных.

## **Как сделать RDD:**

Файл из HDFS (уже восстанавливаемый и распределенный)

Распараллелив Python коллекцию (список, итератор, ...)

Трансформацией из другого RDD

# Операции над RDD

## Трансформации (RDD → RDD):

Трансформации ленивые (вычисляются, когда будут нужны)

*Пример:* **map** применяет преобразование к каждому элементу RDD и возвращает новый RDD с результатом

*Еще примеры:* **reduceByKey, join**

# Операции над RDD

## Трансформации (RDD → RDD):

Трансформации ленивые (вычисляются, когда будут нужны)

*Пример:* **map** применяет преобразование к каждому элементу RDD и возвращает новый RDD с результатом

*Еще примеры:* **reduceByKey, join**

## Действия:

Действия приводят к запуску DAG для расчета RDD

*Примеры:* **saveAsTextFile, collect, count**

# Операции над RDD

## Трансформации (RDD → RDD):

Трансформации ленивые (вычисляются, когда будут нужны)

*Пример:* **map** применяет преобразование к каждому элементу RDD и возвращает новый RDD с результатом

*Еще примеры:* **reduceByKey, join**

## Действия:

Действия приводят к запуску DAG для расчета RDD

*Примеры:* **saveAsTextFile, collect, count**

## Другие операции:

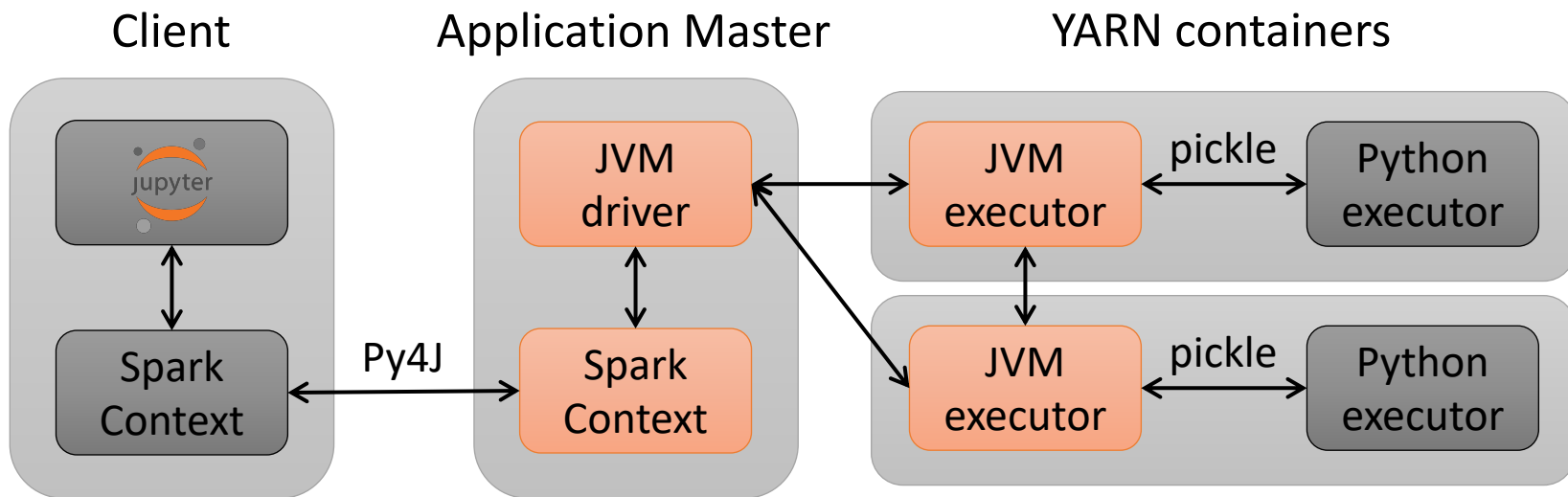
*Примеры:* **persist, cache** заставляют Spark сохранить RDD в памяти для последующего быстрого доступа

# Как устроена программа на PySpark (yarn cluster)

При создании в Python **SparkContext** запускается YARN приложение.

В Application Master запускается **driver**, который создает JVM версию **SparkContext** (хранит конфигурацию, DAG для всех RDD).

**Driver** координирует работу **Executors** (вычисляют RDD).





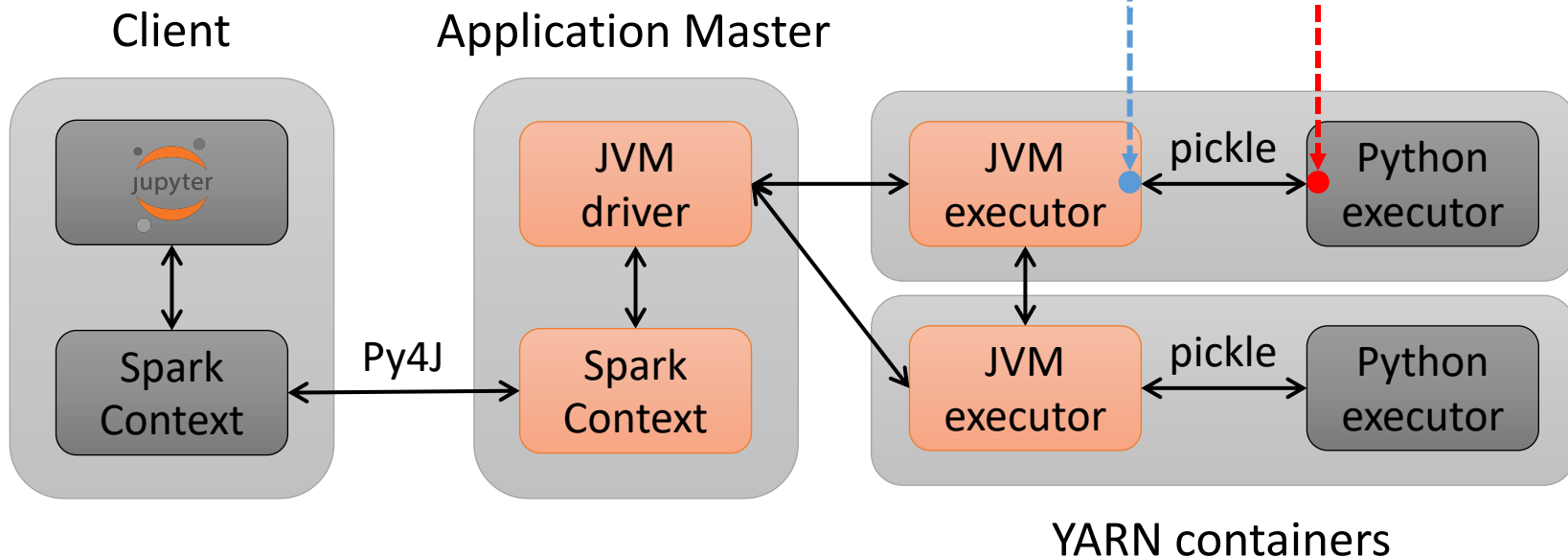
# Как устроена программа на PySpark (yarn cluster)

Spark работает с Python объектами в сериализованном виде (pickle), они будут десериализованы для обработки в Python.

Поэтому элементы RDD занимают память **дважды**:

В сериализованном виде в JVM (например, кэш в RAM)

Десериализованный объект в Python



# Простейшая программа на PySpark

```
rdd = (sc                                     # SparkContext
      .parallelize([1, 2, 3, 4])             # создаем RDD
      .map(lambda x: x * 2))                 # трансформируем RDD
print rdd                                     # ленивые вычисления
print rdd.collect()                         # запускаем DAG
```

```
PythonRDD[17] at RDD at PythonRDD.scala:48
[2, 4, 6, 8]
```

# Пример трансформации

```
rdd = (sc
        .parallelize([1, 2, 3, 4])
        .flatMap(lambda x: [x, x * 2]))
print rdd
print rdd.collect()
```

```
PythonRDD[19] at RDD at PythonRDD.scala:48
[1, 2, 2, 4, 3, 6, 4, 8]
```

## Пример действия

```
rdd = (sc
        .parallelize(np.random.random((1000,)))
        .flatMap(lambda x: [x, x * 2]))
print rdd
print rdd.takeOrdered(2, lambda x: -x)
```

```
PythonRDD[29] at RDD at PythonRDD.scala:48
[1.9987386963918603, 1.997388155520317]
```

# MapReduce как две операции в Spark

```
rdd = (  
    sc  
    .parallelize(["this is text", "text too"])  
    .flatMap(lambda x: [(w, 1) for w in x.split()])  
    .reduceByKey(lambda a, b: a + b))  
print rdd  
print rdd.collect()
```

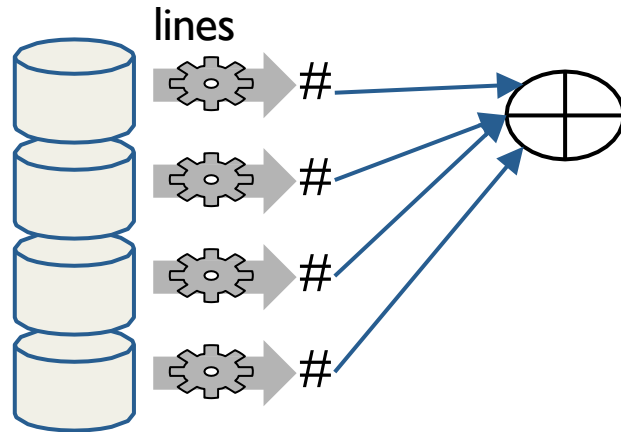
PythonRDD[61] at RDD at PythonRDD.scala:48

[('text', 2), ('too', 1), ('is', 1), ('this', 1)]

# Кэширование в RAM

```
lines = sc.textFile("...", 4)
```

```
print lines.count()
```

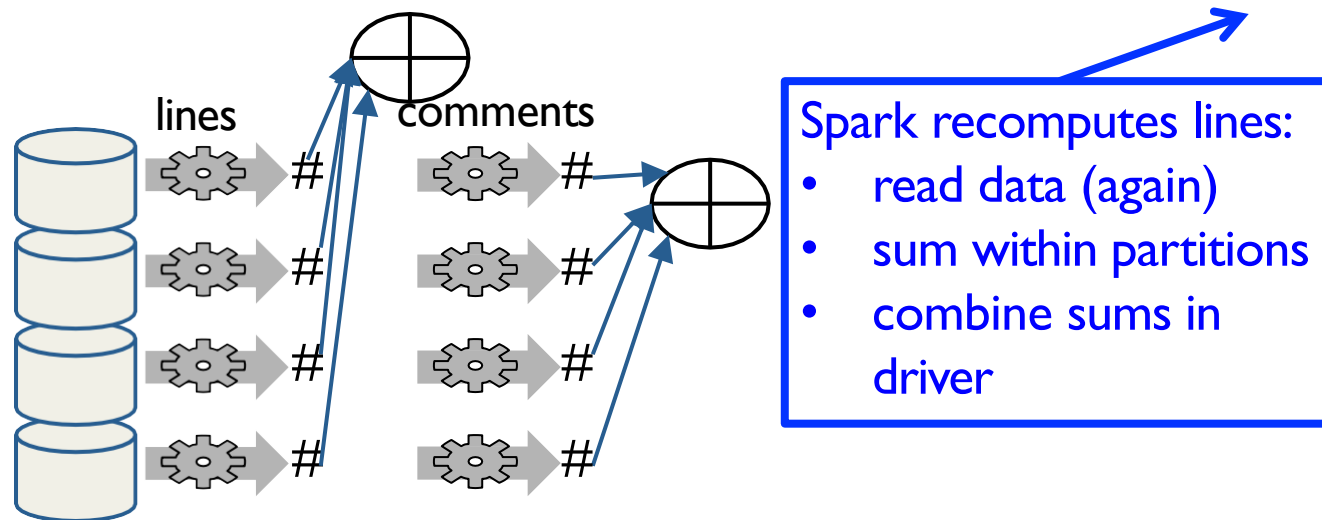


`count()` causes Spark to:

- read data
- sum within partitions
- combine sums in driver

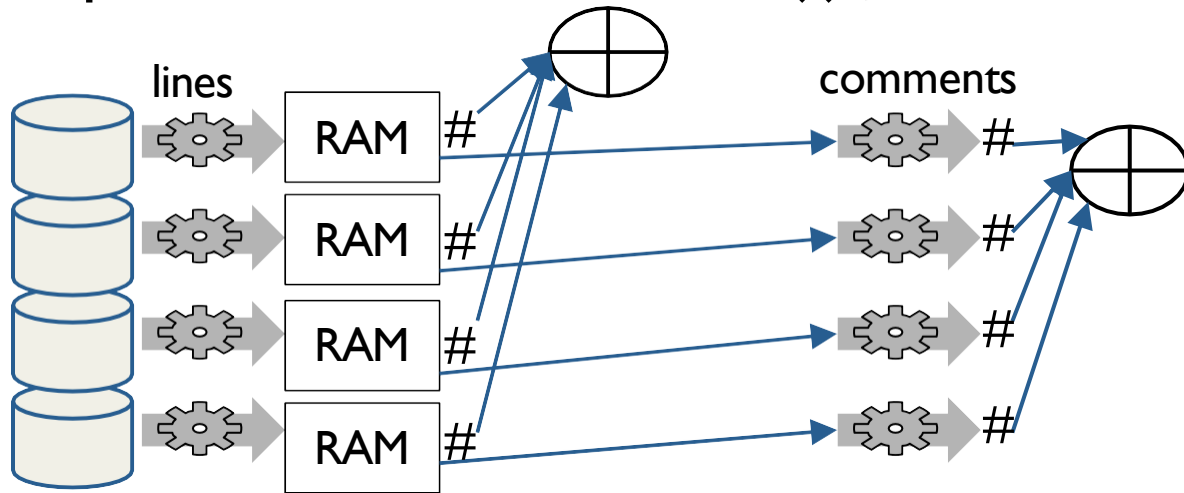
# Кэширование в RAM

```
lines = sc.textFile("...", 4)
comments = lines.filter(isComment)
print lines.count(), comments.count()
```



# Кэширование в RAM

```
lines = sc.textFile("...", 4)
lines.cache() # save, don't recompute!
comments = lines.filter(isComment)
print lines.count(), comments.count()
```





# Broadcast переменные

Когда нужно разослать одни и те же данные на все **executors**

- Словарь в ML алгоритме
- Вектор весов в ML алгоритме

**Executors** имеют **read-only** доступ к этим данным

Отсылаются один раз и могут быть использованы во многих операциях

## Пример с broadcast переменной

```
mapping = {"this": 0, "is": 1, "text": 2, "too": 3}
bc = sc.broadcast(mapping)

rdd = (
    sc
    .parallelize(["this is text", "text too"])
    .flatMap(lambda x: [(bc.value[w], 1) for w in x.split()])
    .reduceByKey(lambda a, b: a + b)
)
print rdd
print rdd.collect()
```

```
PythonRDD[157] at RDD at PythonRDD.scala:48
[(0, 1), (1, 1), (2, 2), (3, 1)]
```

# Accumulator переменные

Когда нужен счетчик (сумматор) при выполнении задач на **executors**:

- Количество ошибок обработки строк
- Суммарный loss

Только **driver** может прочесть итоговые значения

Для **executors** доступ к счетчику **write-only**

# Пример с accumulator переменной

```
bc = sc.broadcast({"this": 0, "is": 1, "text": 2})
errors = sc.accumulator(0)
```

```
def mapper(x):
    global errors
    for w in x.split():
        if w in bc.value:
            yield (bc.value[w], 1)
        else:
            errors += 1
```

```
rdd = (
    sc
    .parallelize(["this is text", "text too"])
    .flatMap(mapper)
    .reduceByKey(lambda a, b: a + b))
print rdd
print rdd.collect()
print "errors:", errors.value
```

```
PythonRDD[187] at RDD at PythonRDD.scala:48
[(0, 1), (1, 1), (2, 2)]
errors: 1
```

# Spark DataFrame API



- В Spark кроме RDD API есть еще DataFrame API
- DataFrame хранит табличные данные (как в Pandas)
- DataFrame поддерживает SQL запросы (на кластере)
- Можно конвертировать из/в Pandas DataFrame
- DataFrame обрабатывается целиком в JVM  
(> 10x быстрее Python)

# Создание DataFrame

Создаем Spark SQL сессию:

```
sc = pyspark.SparkContext()  
se = SparkSession(sc)
```

Создаем RDD:

```
rdd = sc.parallelize([("a", 1), ("a", 2), ("b", 3), ("b", 4)])
```

Конвертируем в DataFrame (вывод типов через Py4J):

```
df = se.createDataFrame(rdd)  
df.printSchema()  
root  
|-- _1: string (nullable = true)  
|-- _2: long (nullable = true)
```

# Запросы к DataFrame

Присваиваем DataFrame имя для запросов:

```
df.registerTempTable("table")
```

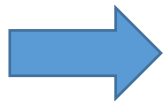
Эквивалентные запросы (выполняются в JVM):

```
se.sql("select key from table where value > 1")
```

```
df.select("key").where("value > 1")
```

```
df.select(df.key).where(df.value > 1)
```

key	value
a	1
a	2
b	3
b	4



key
a
b
b

# Spark DataFrame API

## Плюсы:

- Запросы выполняются в JVM (быстрее чем в Python)
- Spark оптимизирует запросы (превращает запрос в оптимальный план выполнения)

## Минусы:

- В колонках можно хранить только структуры с простыми типами (int, str, float, ...)
- Не все хорошо описывается SQL запросом (например, токенизация текста)



# Заключение

- Spark описывает вычисления в виде графа (DAG) и может оптимизировать хранение промежуточных результатов
- Программа на Spark – это набор операций над RDD
- Вычисления в Spark ленивые, только действия над RDD приводят к запуску вычислений DAG
- DataFrame API позволяет быстрее исполнять запросы к табличным данным, но не такое гибкое как RDD API

# Ссылки

<http://spark.apache.org/docs/latest/programming-guide.html>

<http://spark.apache.org/docs/latest/api/python/index.html>

<http://spark.apache.org/docs/latest/sql-programming-guide.html>

<https://0x0fff.com/wp-content/uploads/2015/11/Spark-Architecture-JD-Kiev-v04.pdf>

<https://spark-summit.org/2014/wp-content/uploads/2014/07/A-Deeper-Understanding-of-Spark-Internals-Aaron-Davidson.pdf>