LSML #2

Введение в Apache Spark

Apache Spark

- Фреймворк для выполнения распределенных задач
- API на нескольких языках: Scala, Java, Python
- Будем рассматривать на примере Python API – PySpark
- Включает полезные модули:
  - MLlib: алгоритмы машинного обучения
  - Spark SQL: выполнение SQL запросов на кластере
  - Алгоритмы на графах, потоковая обработка данных, …

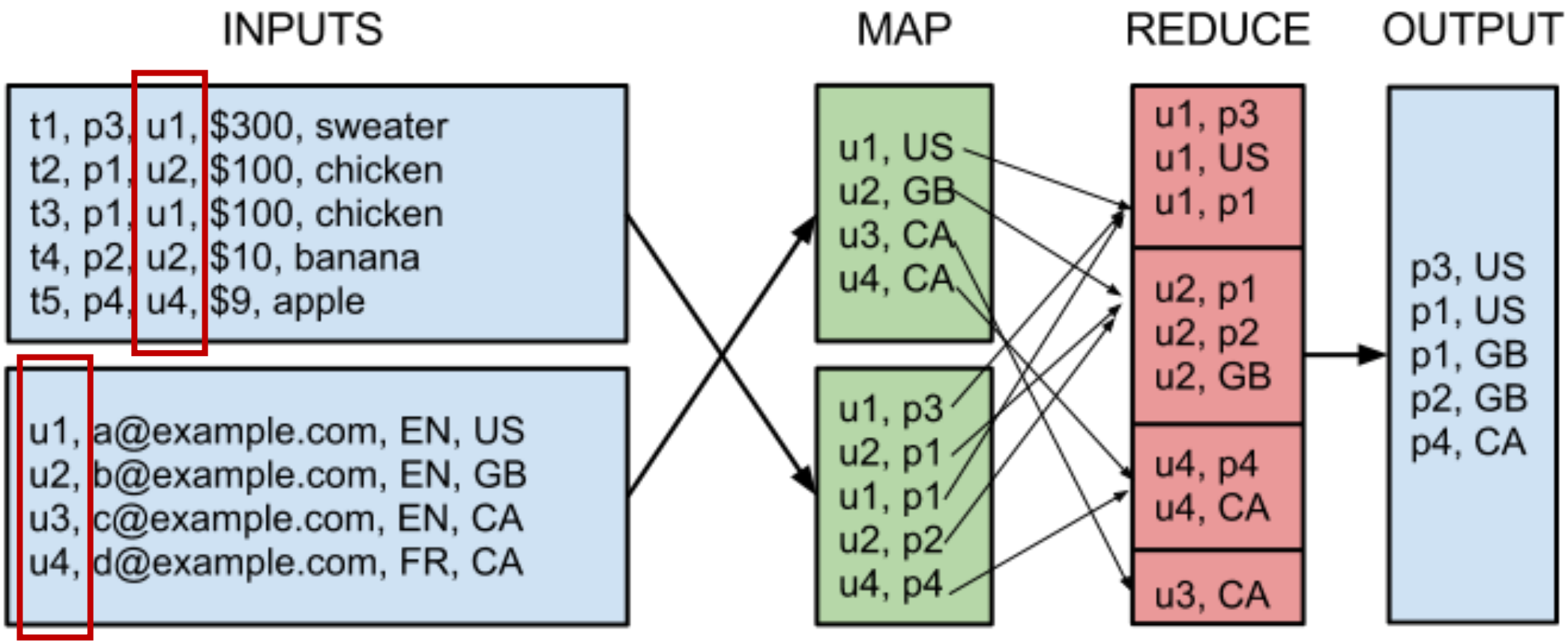
Join на SQL

- Таблица **a** – покупки пользователей (**user**, product, …)
- Таблица **b** – информация о пользователях (**user**, state, …)
- Хотим получить покупки продуктов по штатам
- Нужно сделать join по **user**

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

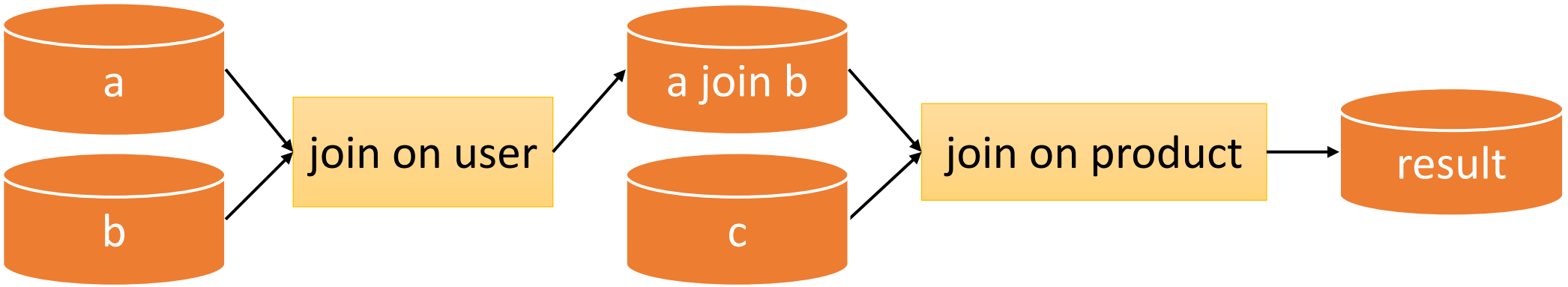
Join на MapReduce

- Этот же запрос на MapReduce:



MapReduce: еще один join

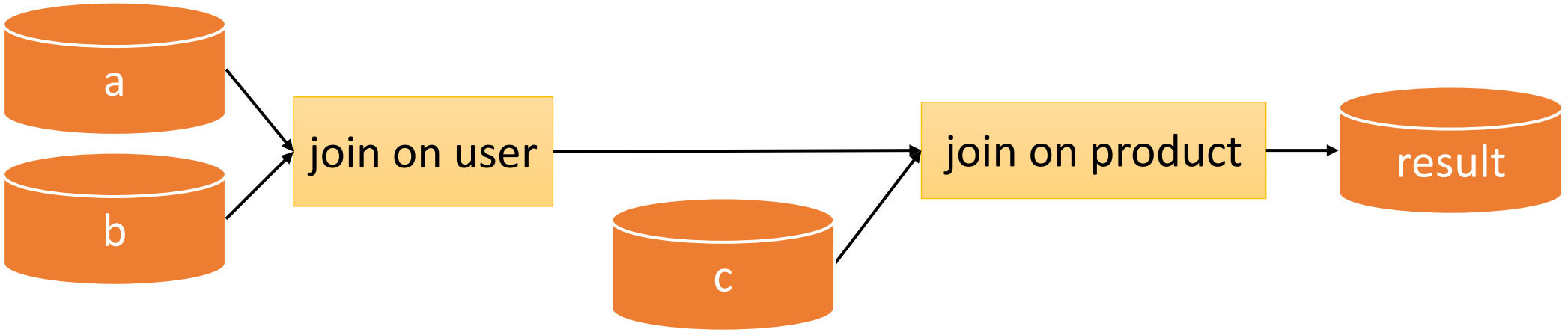
- В таблице **c** лежит информация о продуктах (**product**, type, …)
- Решение на **Hadoop MapReduce**:



- Промежуточные результаты пишутся на диск и читаются с диска
- Это медленно и не всегда необходимо!

Spark: еще один join

- Решение на **Spark**:



- Результаты вычислений передаются по возможности в памяти **без сохранения на диск**
- Промежуточные результаты можно **закэшировать в памяти** и выполнять над ними несколько операций (если хватает суммарной памяти машин)
- Вычисления можно представить в виде графа (**DAG** – направленный граф без циклов)

RDD в Spark

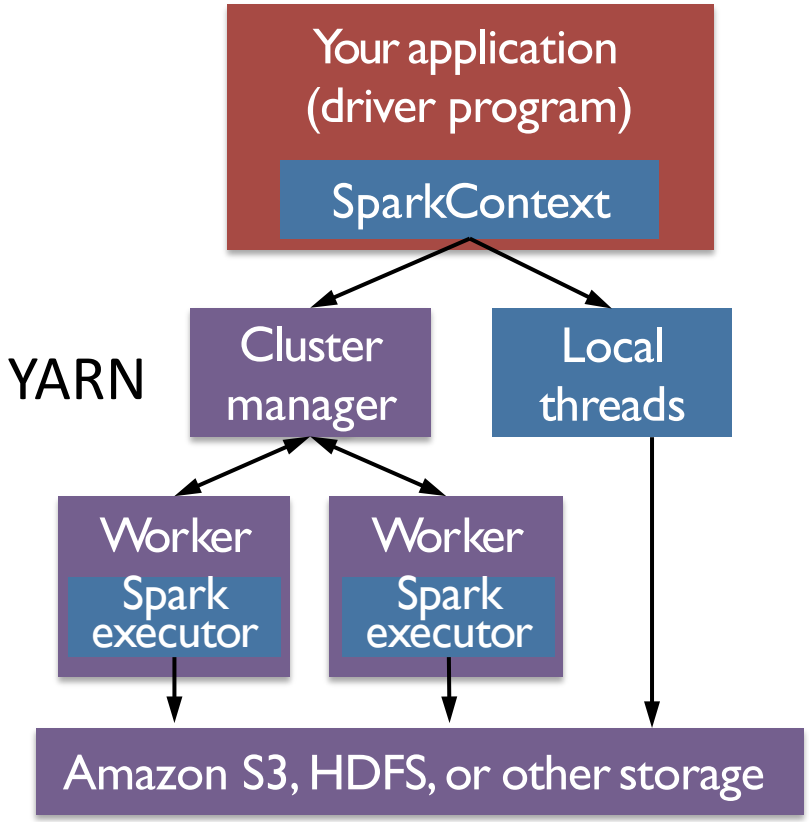
- **RDD** (resilient distributed dataset) – восстанавливаемый распределенный набор данных
  - Набор данных хранится распределенно на разных машинах
  - Потерянные части могут быть восстановлены (известна цепочка вычислений – DAG)

- Как сделать RDD?
  - Из файла (например, в HDFS)
  - **Распараллелив** Python коллекцию (список, итератор, …)
  - **Трансформацией** из другого RDD

Операции над RDD

- Программа на Spark пишется в терминах операций над RDD
- **Трансформации**:
  - RDD → RDD
  - Ленивые – не вычисляются сразу
  - Примеры: **map, reduceByKey, join**
- **Действия**:
  - Приводят к вычислению RDD
  - Примеры: **saveAsTextFile, collect, count**
- **Другие операции**:
  - **persist, cache** – помечают RDD для сохранения в памяти/на диске при первом их вычислении

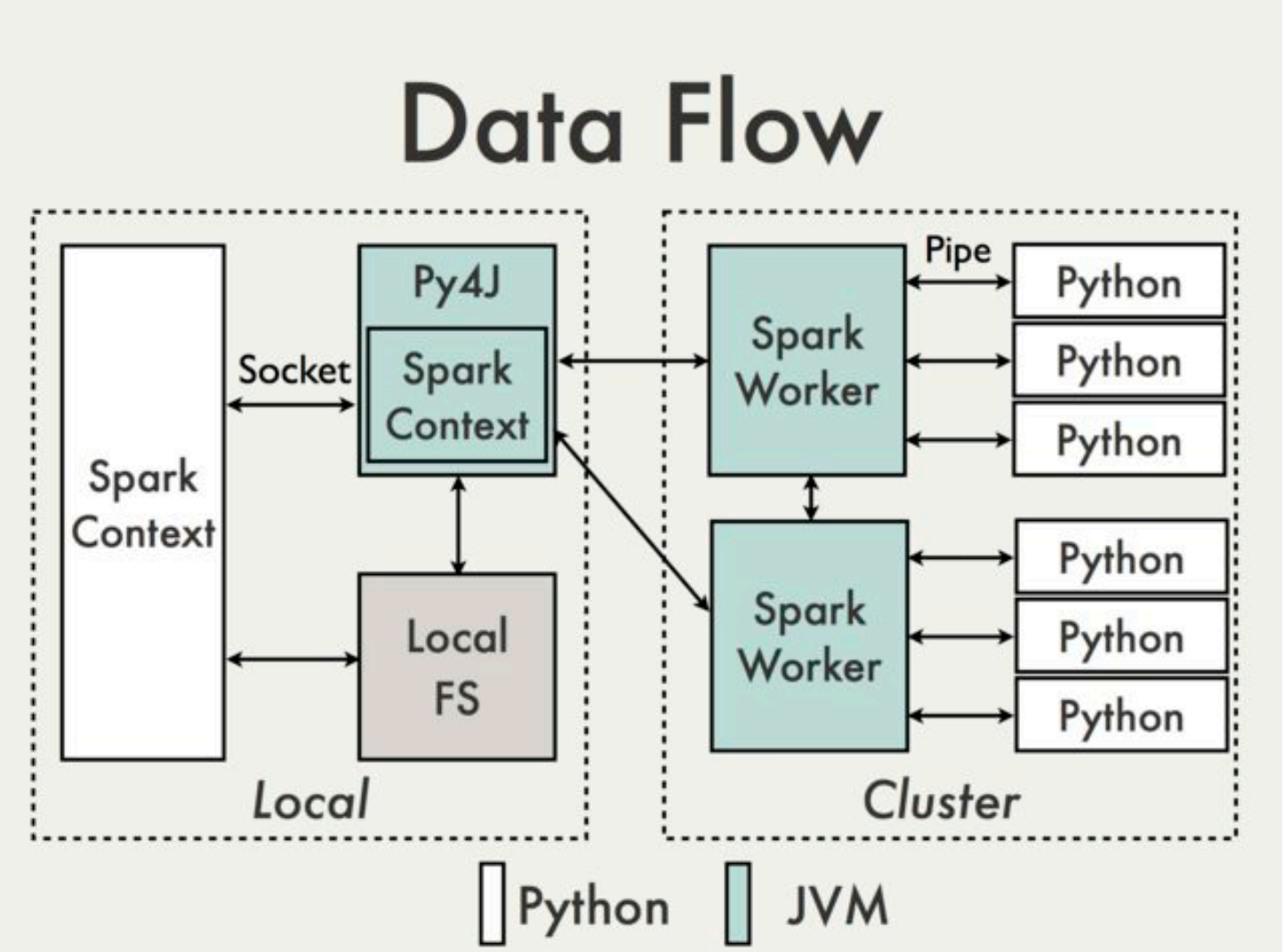
Как устроена программа на Spark



- Программа состоит из **driver** и **workers**.
- **Driver** хранит цепочку вычислений – DAG
- **Workers** запускаются на машинах кластера или как локальные процессы на одной машине с driver
- RDD распределены по **workers**
- **Driver –** это обычная программа на Python, в которой создается **SparkContext (sc)** – объект для работы со Spark

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

Особенности PySpark



- Все Python объекты сериализуются при помощи cPickle:
  - Любое действие с Python объектом требует десериализации в Python
- При работе с PySpark данные хранятся дважды:
  - Сериализованный RDD в JVM у Spark Worker
  - Десериализованные данные в памяти Python

https://cwiki.apache.org/confluence/display/SPARK/PySpark+Internals

Пример трансформации

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

```
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
[1.998738696391860 3, 1.997388155520317]
```

MapReduce как две операции в Spark

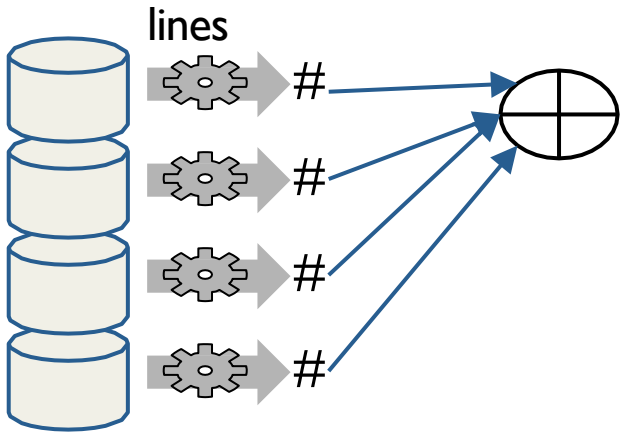
```
rdd = (
    sc
   .parallelize(["this is text", "text too"])
   .flatMap(lambda x: [(w, 1) for w in x.split()])
   .reduceByKey(lambda a, b: a + b))
print rdd
print rdd.collect()
```

```
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()
```

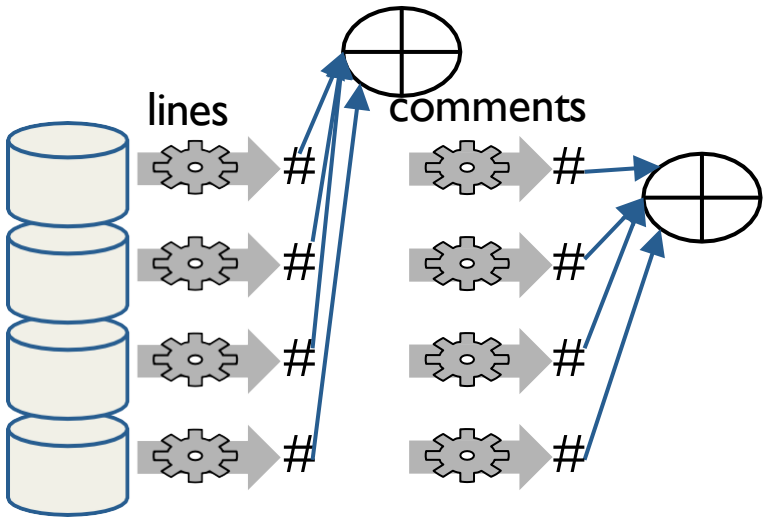


count() causes Spark to:

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

Кэширование в RAM

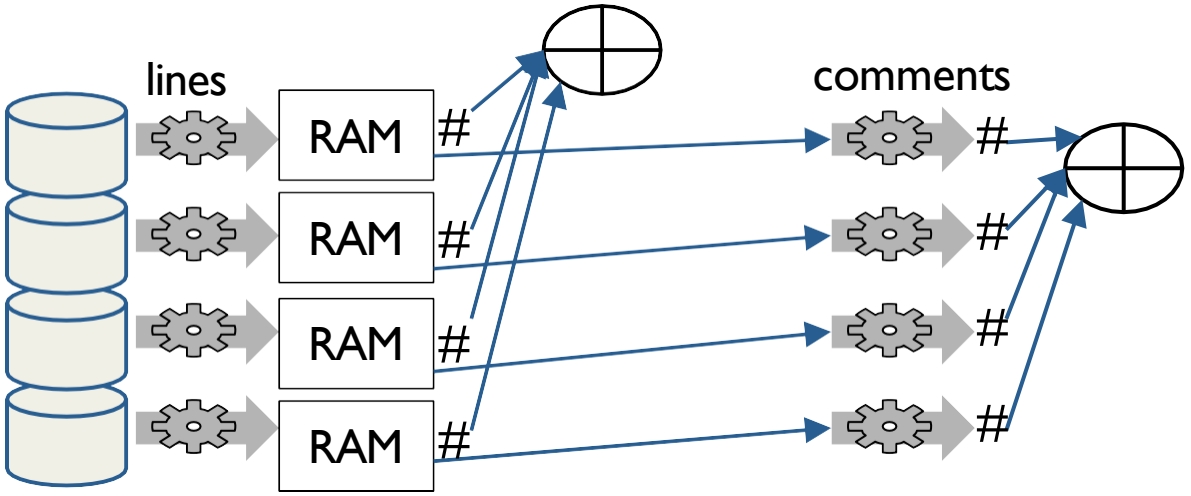
```
lines = sc.textFile("...", 4)
comments = lines.filter(isComment)
print lines.count(), comments.count()
```



Spark recomputes lines:
- read data (again)
- sum within partitions
- combine sums in driver

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 workers
  - Словарь в ML алгоритме
  - Вектор весов в ML алгоритме
- Workers 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 workers:
  - Количество ошибок обработки строк
- Только driver может прочитать итоговые значения
- Для workers 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
```

DataFrame API

- Работает с DataFrame (таблицы)
- Поддерживает SQL запросы
- Умеет конвертировать из/в Pandas DataFrame
- Вычисления производятся на Java (без Python)
  - Для этого нужно использовать базовые типы и коллекции в колонках

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

DataFrame API пример

```
import pandas as pd
from pyspark.sql import SparkSession

ss = (SparkSession
      .builder
      .appName("spark sql example")
      .getOrCreate())
```

```
df = pd.DataFrame(
    [["cat", [1, 1]], ["cat", [2]], ["dog", [1]]],
    columns=["name", "cnt"])
```

DataFrame API пример

```
df
```

	name	cnt
0	cat	[1, 1]
1	cat	[2]
2	dog	[1]

```
sdf = ss.createDataFrame(df)
```

```
sdf.printSchema()
```

```
root
 |-- name: string (nullable = true)
 |-- cnt: array (nullable = true)
 |    |-- element: long (containsNull = true)
```

Native Java Types

DataFrame API пример

```
sdf.registerTempTable("animals")
```

```
ss.sql("""
select name, sum(cnt) as sum
from
    (select name, explode(cnt) as cnt
     from animals)
group by name
""").toPandas()
```

Выполняется на кластере

	name	sum
0	dog	1
1	cat	4

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