



Лекция 10

Spark

Клюкин Денис

# Мотивация

---



*MapReduce* отлично упрощает анализ *big data* на больших, но ненадежных, кластерах

Но с ростом популярности фреймворка пользователи хотят большего:

- **Итеративных** задач, например, алгоритмы machine learning
- **Интерактивной** аналитики



## Мотивация

---



Для решения обоих типов проблем требуются одна вещь, которой нет в MapReduce...

- Эффективных примитивов для общих данных  
*(Efficient primitives for data sharing)*

В MapReduce единственный способ для обмена данными между задачами (*jobs*), это надежное хранилище (*stable storage*)

Репликация также замедляет систему, но это необходимо для обеспечения *fault tolerance*

## Решение

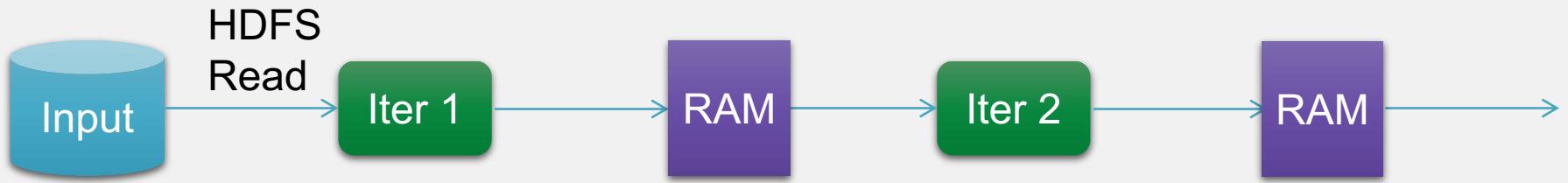
---



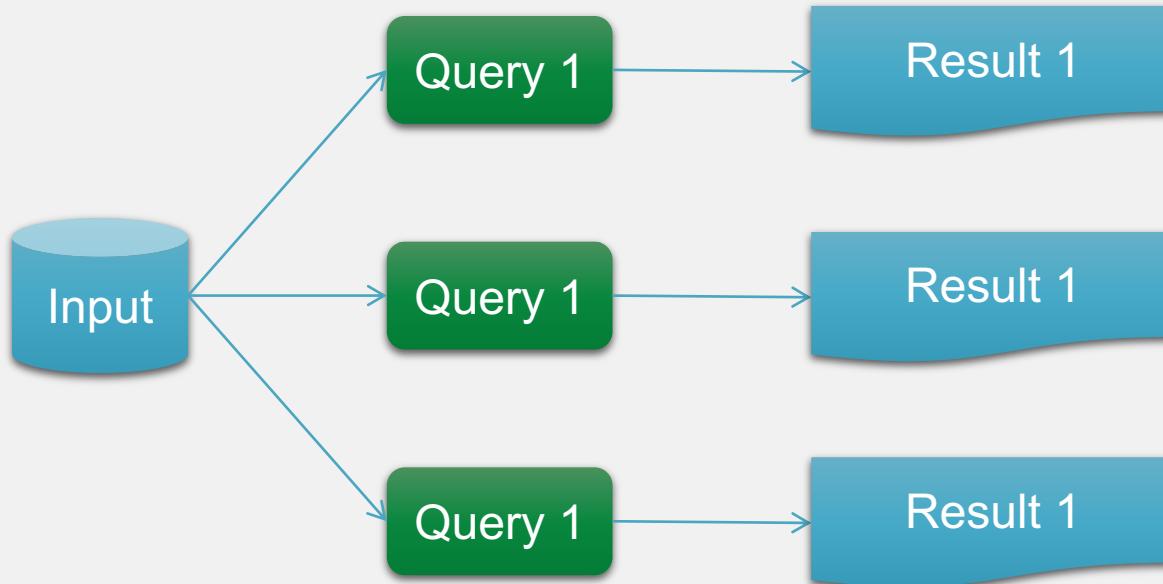
# In-Memory Data Processing and Sharing



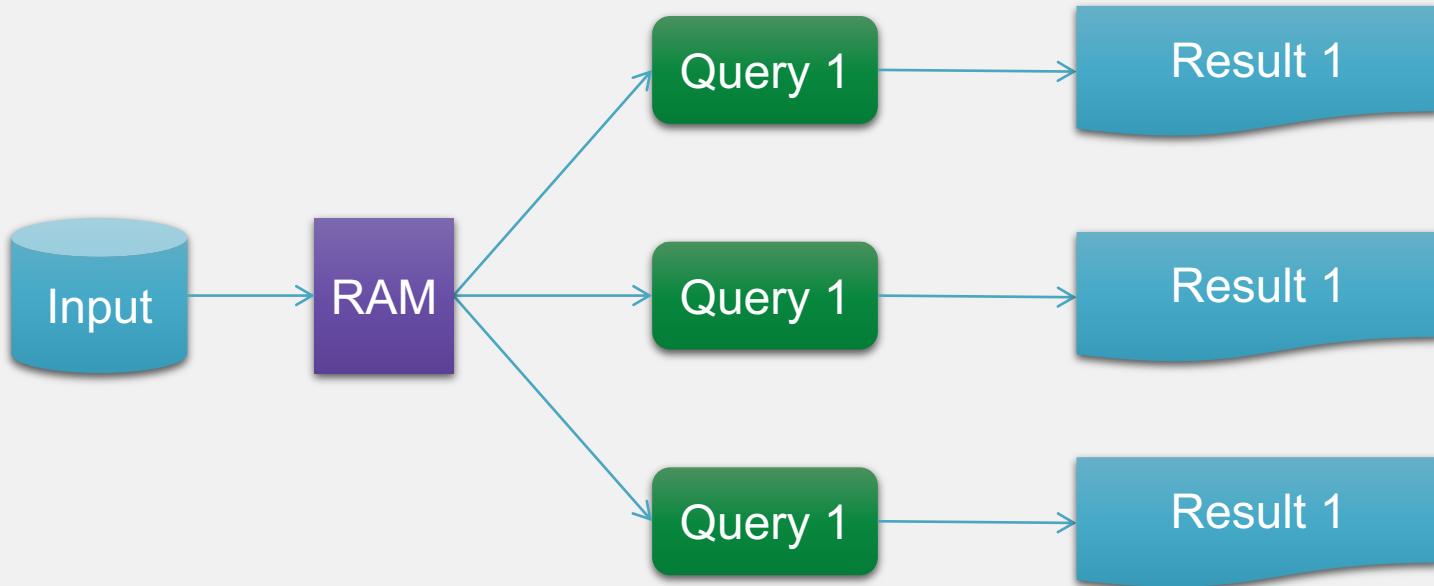
# Пример



# Пример



# Пример





---

•

## Задача

- Разработать дизайн абстракции распределенной памяти с поддержкой **fault tolerant** и **эффективности**

## Решение

- *Resilient Distributed Datasets (RDD)*

# Apache Spark Engine

---



- *Lightning-fast cluster computing!*
- Apache Spark – быстрый и многоцелевой «движок» для обработки больших объемов данных
- Предоставляет программный интерфейс на Scala
- Каждый RDD является объектом в Spark

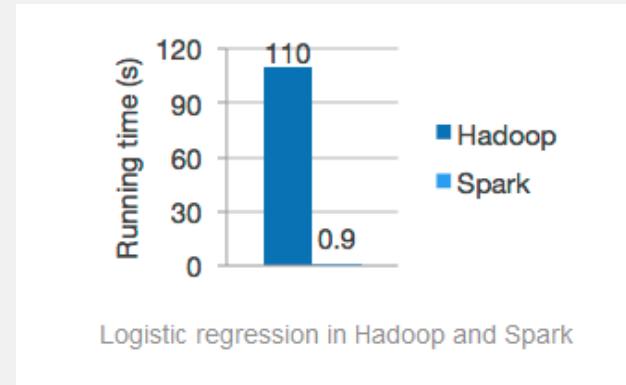


# Apache Spark



## Скорость

- Работает быстрее чем Hadoop MapReduce в 100 раз, если данные в памяти, и в 10 раз, если данные на диске
  - В Spark есть продвинутый DAG-механизм выполнения задач, которые поддерживает cyclic data flow и in-memory computing



## Легкость использования

- Просто писать приложения на Java, Scala и Python
  - Более 80 высокоуровневых операторов для построения параллельных приложений
  - Их можно использовать интерактивно в shell на Scala и Python

```
file = spark.textFile("hdfs://...")  
  
file.flatMap(lambda line: line.split())  
    .map(lambda word: (word, 1))  
    .reduceByKey(lambda a, b: a+b)
```

Word count in Spark's Python API

# Apache Spark



## Обобщенность

- Комбинирование SQL, streaming и комплексной аналитики в рамках одного приложения
- Spark SQL, Mlib, GraphX и Spark Streaming

## Работает везде

- Hadoop, Mesos, standalone или в облаке
- Доступ к данным из различных источников, включая HDFS, Cassandra, HBase, S3

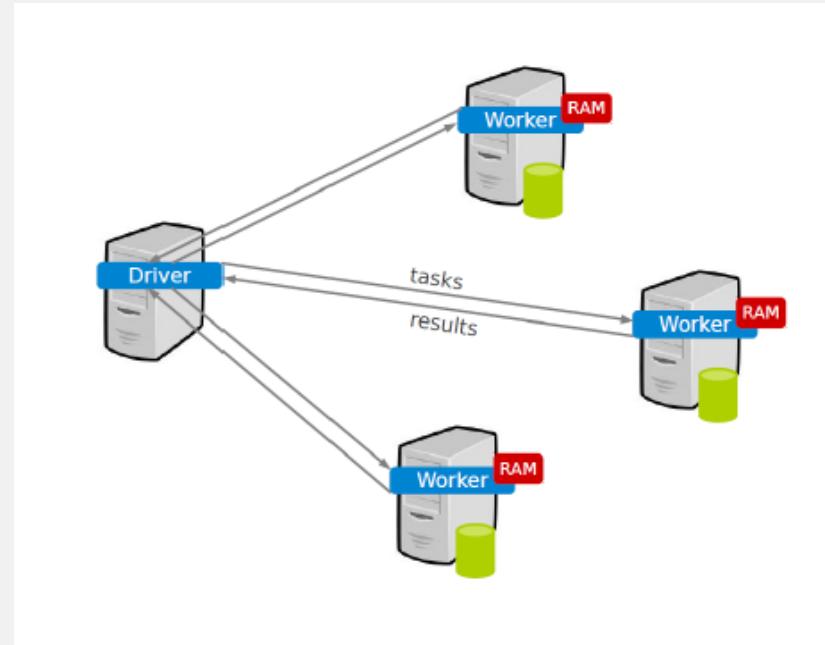
<https://spark.apache.org/>



# Программный интерфейс Spark



Приложение на Spark состоит из **программы-драйвера**, которая запускает пользовательскую функцию *main* и выполняет различные операции параллельно на кластере





- Абстрактное представление распределенной RAM
- Immutable коллекция объектов распределенных по всему кластеру
- RDD делится на партиции, которые являются атомарными частями информации
- Партиции RDD могут храниться на различных нодах кластера

# Программная модель Spark

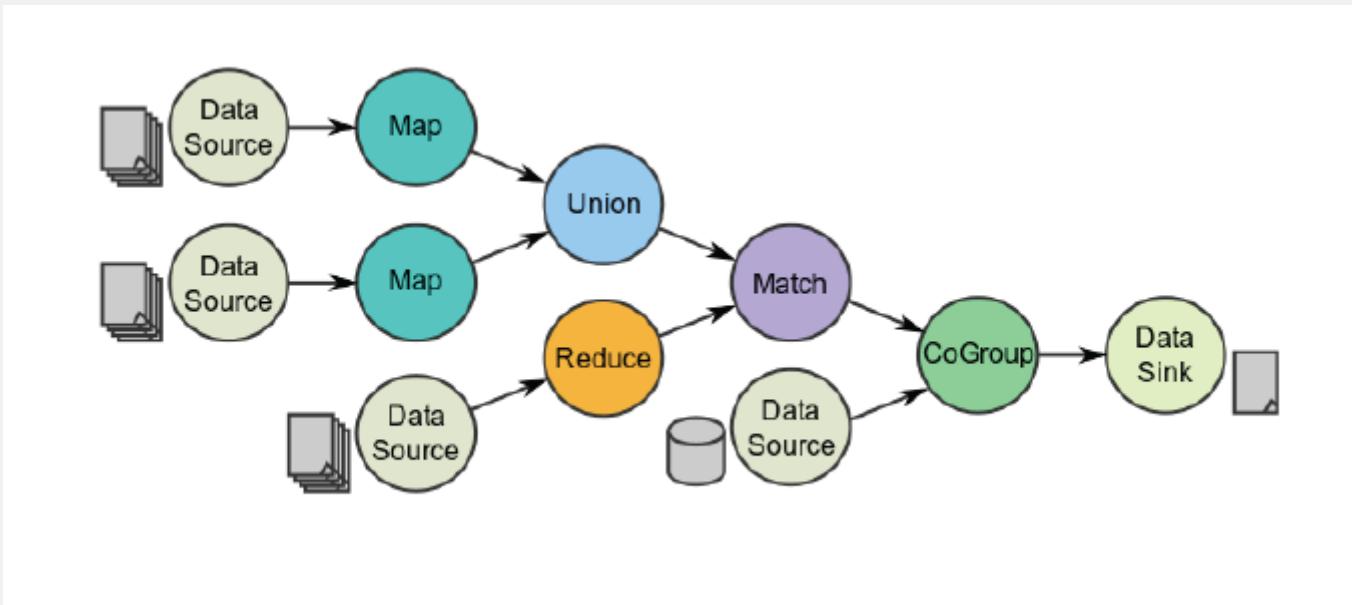


- Основана на **parallelizable operators**
- Эти операторы являются функциями высокого порядка, которые выполняют *user-defined* функции параллельно
- Поток обработки данных состоит из любого числа **data sources, operators и data sinks** путем соединения их *inputs* и *outputs*

# Программная модель Spark



Задача описывается с помощью **directed acyclic graphs (DAG)**



# Higher-Order Functions



**Higher-order functions:** RDD операторы

Существует два типа операторов

- **transformations** и **actions**

**Transformations:** lazy-операторы, которые создают новые RDD

**Actions:** запускают вычисления и возвращают результат в программу или пишут данные во внешнее хранилище

# Higher-Order Functions



Transformations	<i>map</i> ( $f : T \Rightarrow U$ )	: $RDD[T] \Rightarrow RDD[U]$
	<i>filter</i> ( $f : T \Rightarrow \text{Bool}$ )	: $RDD[T] \Rightarrow RDD[T]$
	<i>flatMap</i> ( $f : T \Rightarrow Seq[U]$ )	: $RDD[T] \Rightarrow RDD[U]$
	<i>sample</i> ( $\text{fraction} : \text{Float}$ )	: $RDD[T] \Rightarrow RDD[T]$ (Deterministic sampling)
	<i>groupByKey()</i>	: $RDD[(K, V)] \Rightarrow RDD[(K, Seq[V])]$
	<i>reduceByKey</i> ( $f : (V, V) \Rightarrow V$ )	: $RDD[(K, V)] \Rightarrow RDD[(K, V)]$
	<i>union</i> ()	: $(RDD[T], RDD[T]) \Rightarrow RDD[T]$
	<i>join</i> ()	: $(RDD[(K, V)], RDD[(K, W)]) \Rightarrow RDD[(K, (V, W))]$
	<i>cogroup</i> ()	: $(RDD[(K, V)], RDD[(K, W)]) \Rightarrow RDD[(K, (Seq[V], Seq[W]))]$
	<i>crossProduct</i> ()	: $(RDD[T], RDD[U]) \Rightarrow RDD[(T, U)]$
	<i>mapValues</i> ( $f : V \Rightarrow W$ )	: $RDD[(K, V)] \Rightarrow RDD[(K, W)]$ (Preserves partitioning)
	<i>sort</i> ( $c : \text{Comparator}[K]$ )	: $RDD[(K, V)] \Rightarrow RDD[(K, V)]$
Actions	<i>partitionBy</i> ( $p : \text{Partitioner}[K]$ )	: $RDD[(K, V)] \Rightarrow RDD[(K, V)]$
	<i>count</i> ()	: $RDD[T] \Rightarrow \text{Long}$
	<i>collect</i> ()	: $RDD[T] \Rightarrow Seq[T]$
	<i>reduce</i> ( $f : (T, T) \Rightarrow T$ )	: $RDD[T] \Rightarrow T$
	<i>lookup</i> ( $k : K$ )	: $RDD[(K, V)] \Rightarrow Seq[V]$ (On hash/range partitioned RDDs)
	<i>save</i> ( $path : \text{String}$ )	: Outputs RDD to a storage system, e.g., HDFS

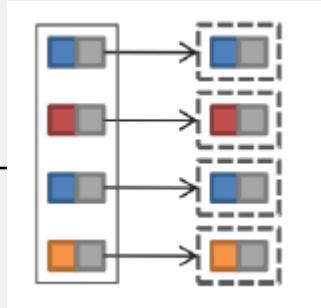


# RDD Transformations - Map



Все пары обрабатываются независимо

```
1. // passing each element through a function.  
2. val nums = sc.parallelize(Array(1, 2, 3))  
3. val squares = nums.map(x => x * x)    // {1, 4, 9}  
  
4. // selecting those elements that func returns true.  
5. val even = squares.filter(x => x % 2 == 0) // {4}  
  
6. // mapping each element to zero or more others.  
7. nums.flatMap(x => Range(0, x, 1)) // {0, 0, 1, 0, 1, 2}
```



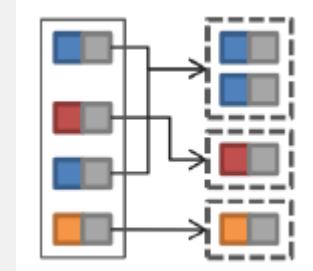


# RDD Transformations - Reduce



- Пары с одинаковыми ключами группируются
- Группы обрабатываются независимо

```
1. val pets = sc.parallelize(  
2.     Seq(("cat", 1), ("dog", 1), ("cat", 2)))  
  
3. pets.reduceByKey((x, y) => x + y)  
4. // {(cat, 3), (dog, 1)}  
  
5. pets.groupByKey()  
6. // {(cat, (1, 2)), (dog, (1))}
```



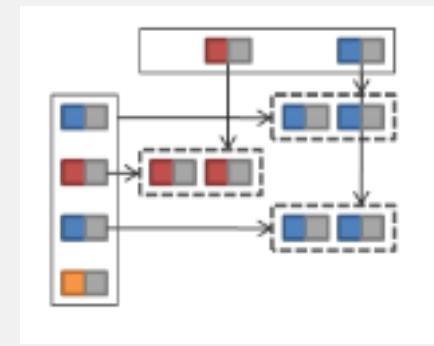


# RDD Transformations - Join



- Выполняется *equi-join* по ключу
- Join-кандидаты обрабатываются независимо

```
1. val visits = sc.parallelize(  
2.                 Seq(("index.html", "1.2.3.4"),  
3.                     ("about.html", "3.4.5.6"),  
4.                     ("index.html", "1.3.3.1")))  
  
5. val pageNames = sc.parallelize(  
6.                 Seq(("index.html", "Home"),  
7.                     ("about.html", "About")))  
  
8. visits.join(pageNames)  
9. // ("index.html", ("1.2.3.4", "Home"))  
10. // ("index.html", ("1.3.3.1", "Home"))  
11. // ("about.html", ("3.4.5.6", "About"))
```



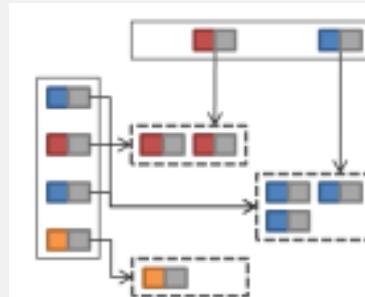


# RDD Transformations - CoGroup



- Каждый *input* группируется **по ключу**
- Группы с одинаковыми ключами обрабатываются вместе

```
1. val visits = sc.parallelize(  
2.                 Seq(("index.html", "1.2.3.4"),  
3.                           ("about.html", "3.4.5.6"),  
4.                           ("index.html", "1.3.3.1")))  
  
5. val pageNames = sc.parallelize(  
6.                 Seq(("index.html", "Home"),  
7.                           ("about.html", "About")))  
  
8. visits.cogroup(pageNames)  
9. // ("index.html", ((1.2.3.4, "1.3.3.1"), ("Home")))  
10. // ("about.html", ((3.4.5.6), ("About")))
```



# RDD Transformations - Union и Sample

---



**Union:** объединяет два RDD и возвращает один RDD используя **bag**-семантику, т.е. дубликаты не удаляются

**Sample:** похоже на *map*, за исключением того, что RDD сохраняет *seed* для генератора произвольных чисел для каждой партии чтобы детерминировано выбирать сэмпл записей

# RDD Actions



Возвращает все элементы RDD в виде массива

```
1. val nums = sc.parallelize(Array(1, 2, 3))  
2. nums.collect() // Array(1, 2, 3)
```

Возвращает массив с первыми n элементами RDD

```
1. nums.take(2) // Array(1, 2)
```

Возвращает число элементов в RDD

```
1. nums.count() // 3
```

# RDD Actions



Агрегирует элементы RDD используя заданную функцию

1. `nums.reduce((x, y) => x + y)`
2. // или
3. `nums.reduce(_ + _) // 6`

Записывает элементы RDD в виде текстового файла

1. `nums.saveAsTextFile("hdfs://file.txt")`

# SparkContext



- Основная точка входа для работы со Spark
- Доступна в *shell* как переменная **sc**
- В *standalone*-программах необходимо создавать отдельно

```
1. import org.apache.spark.SparkContext  
2. import org.apache.spark.SparkContext._  
  
3. val sc = new SparkContext(master, appName, [sparkHome],  
[jars])
```

local  
local[k]  
spark://host:port  
mesos://host:port  
yarn-client  
yarn-cluster



# Создание RDD



Преобразовать коллекцию в RDD

```
1. val a = sc.parallelize(Array(1, 2, 3))
```

Загрузить текст из локальной FS, HDFS или S3

```
1. val a = sc.textFile("file.txt")
2. val b = sc.textFile("directory/*.txt")
3. val c = sc.textFile("hdfs://namenode:9000/path/file")
```

# Пример



## Посчитать число строк содержащих MAIL

```
1. val file = sc.textFile("hdfs://...")  
2. val sics = file.filter(_.contains("MAIL")) // transformation  
3. val cached = sics.cache()  
4. val ones = cached.map(_ => 1) // transformation  
5. val count = ones.reduce(_+_)
```

```
1. val file = sc.textFile("hdfs://...")  
2. val count = file.filter(_.contains("MAIL")).count()
```



transformation



action

# Shared Variables



- Когда Spark запускает выполнение функции параллельно как набор тасков на различных нодах, то отправляется копия каждой переменной, используемой в функции, на каждый таск
- Иногда нужно, чтобы переменная была общая между тасками или между тасками программой-драйвером
- Использование обычных *read-write* общих переменные между тасками неэффективно

# Shared Variables

---



Есть два типа *shared variables*

- **broadcast variables**
- **accumulators**

# Shared Variables: Broadcast Variables



Read-only переменные **кешируются** на каждой машине вместо того, чтобы отправлять копию на каждый таск

*Broadcast Variables* не отсылаются на ноду больше **одного раза**

```
1. scala> val broadcastVar = sc.broadcast(Array(1, 2, 3))
2. broadcastVar: spark.Broadcast[Array[Int]] =
spark.Broadcast(b5c40191-...)
3. scala> broadcastVar.value
4. res0: Array[Int] = Array(1, 2, 3)
```

# Shared Variables: Accumulators



Могут быть только **добавлены**

Могут использоваться для реализации **счетчиков**  
**и сумматоров**

Таски, работающие на кластере, могут затем  
добавлять значение используя оператор **+=**

```
1. scala> val accum = sc.accumulator(0)
2. accum: spark.Accumulator[Int] = 0

3. scala> sc.parallelize(Array(1, 2, 3, 4)).foreach(x =>
  accum += x)
4. ...

5. scala> accum.value
6. res2: Int = 10
```

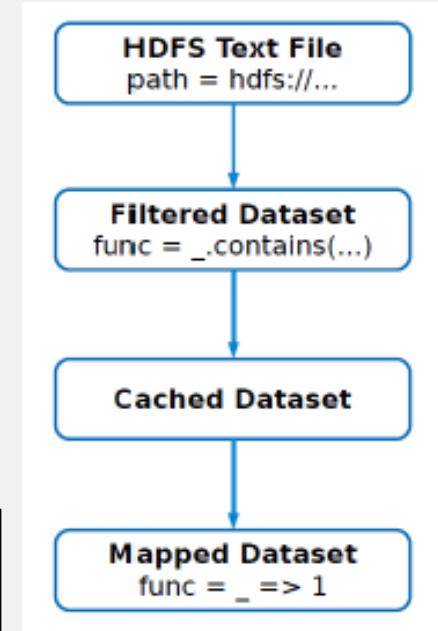
# Lineage



**Lineage:** это transformations, используемые для построения RDD

**RDD** сохраняются как цепочка объектов, охватывающих весь **lineage** каждого RDD

```
1. val file = sc.textFile("hdfs://...")  
2. val mail =  
   file.filter(_.contains("MAIL"))  
3. val cached = mail.cache()  
4. val ones = cached.map(_ => 1)  
5. val count = ones.reduce(_+_)
```



# Dependencies RDD

---



Два типа зависимостей между RDD

- **Narrow**
- **Wide**

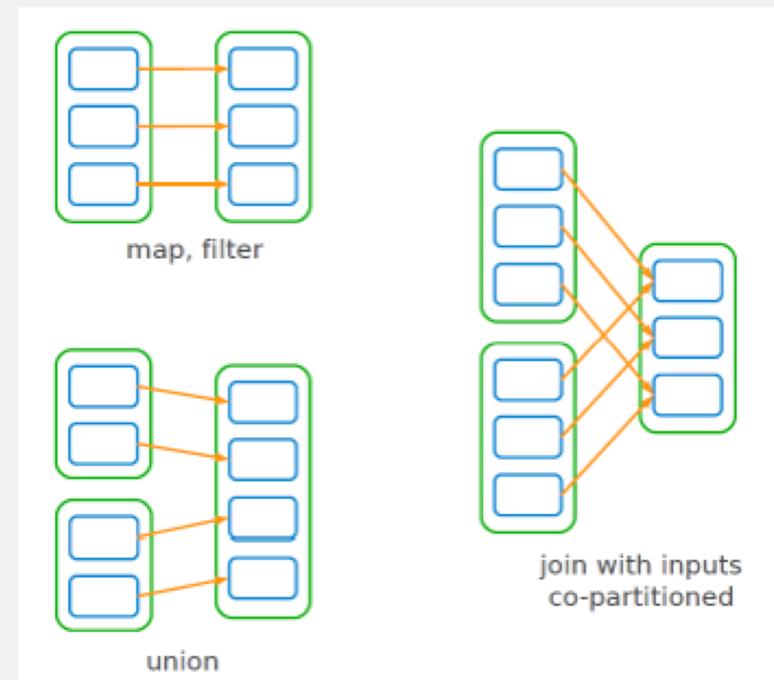
# Dependencies RDD: Narrow



**Narrow:** каждая партиция родительского RDD используется максимум в одной дочерней партиции RDD

*Narrow dependencies* позволяют выполнять **pipelined execution** на одной ноде кластера:

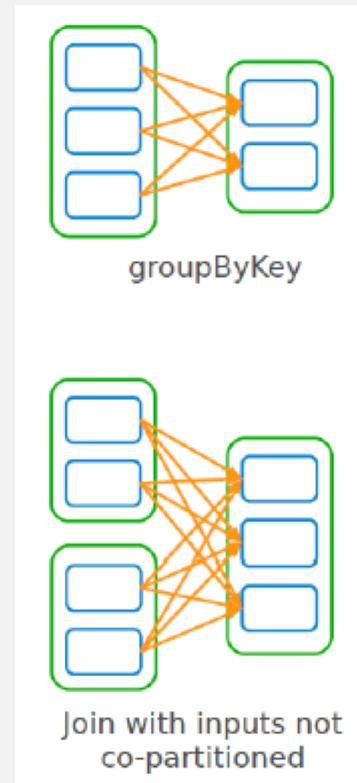
- Напр., фильтр следуемый за Map



# Dependencies RDD: Wide



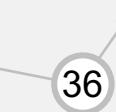
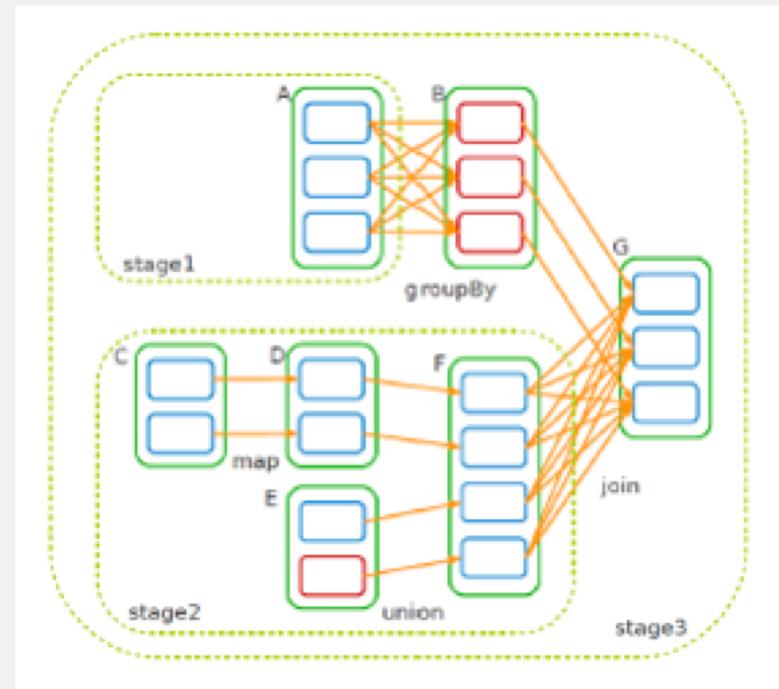
**Wide:** каждая партиция  
родительского RDD  
используется в  
множестве дочерних  
партиций RDD



# Job Scheduling



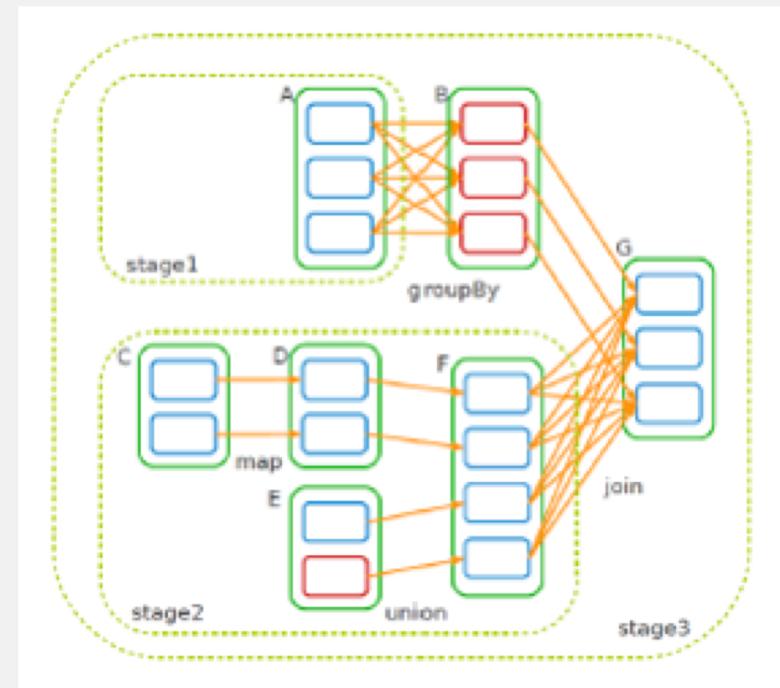
- Когда пользователь запускает ***action*** на RDD шедулер строит DAG из ***stages*** графа ***RDD lineage***
- **Stage** содержит различные ***pipelined transformations*** с ***narrow dependencies***
- Граница для **stage**
  - ***Shuffles*** для ***wide dependencies***
  - Уже обработанные партиции



# Job Scheduling



- Шедулер запускает таски для обработки оставшихся партиций (*missing partitions*) из каждой *stage* пока не обрабатывается целевая (*target*) RDD
- Таски назначаются машинам на основе локальности
  - Если таску требуется партиция, которая доступна в памяти на ноде, то таск отправляется на эту ноду



# RDD Fault Tolerance



- RDD поддерживает **информацию о *lineage***, которая может быть использована для **восстановления** потерянных партиций
- **Логгирование *lineage***
- **Отсутствие репликации**
- Пересчет только **потерянных партиций (*lost partitions*)** RDD

# RDD Fault Tolerance



Промежуточные результаты из *wide dependencies* материализуются на нодах, отвечающих за родительские партиции (для упрощения *fault recovery*)

Если таск фейлится, то он будет перезапущен на другой ноде пока доступны ***stages parents***

Если некоторые ***stages*** становятся недоступны, то таски сабмитятся для расчета отсутствующих партиций в паралели

# RDD Fault Tolerance



- Восстановление может **затратным по времени** для RDD с длинными цепочками *lineage* и wide dependencies
- Может быть полезным сохранять состояния некоторых RDD в надежное хранилище
- Решение, что сохранять, остается за разработчиком

# Memory Management

---



Если недостаточно памяти для **новых** партиций RDD, то будет использоваться механизм вытеснения **LRU** (*least recently used*)

Spark предоставляет три опции для хранения **persistent RDD**

- В *memory storage* в виде **deserialized Java objects**
- В *memory storage* в виде **serialized Java objects**
- На *disk storage*

# Memory Management



В случае ***persistent RDD*** каждая нода хранит любые партиции RDD в RAM

Это позволяет новым *actions* выполняться **намного быстрее**

Для этого используются методы *persist()* или *cache()*

Различные уровни хранения:

- MEMORY ONLY
- MEMORY AND DISK
- MEMORY ONLY SER
- MEMORY AND DISK SER
- MEMORY ONLY 2, MEMORY AND DISK 2 и т.д..

# RDD Applications

---



Приложения, которые **подходят** для RDD

- ***Batch applications***, которые применяют одну операцию ко всем элементам из набора данных

Приложения, которые **не подходят** для RDD

- Приложения, которые выполняют ***asynchronous fine-grained updates***, изменяя общее состояние (например, *storage system* для веб-приложений)

## Итог

---



- **RDD** – это распределенная абстракция памяти, которая является ***fault tolerant*** и **эффективной**
- Два типа операций: ***Transformations*** и ***Actions***
- RDD fault tolerance: ***Lineage***



# Примеры: Text Search (Scala)



```
1. val file = spark.textFile("hdfs://...")  
2. val errors = file.filter(line => line.contains("ERROR"))  
  
3. // Count all the errors  
4. errors.count()  
  
5. // Count errors mentioning MySQL  
6. errors.filter(line => line.contains("MySQL")).count()  
  
7. // Fetch the MySQL errors as an array of strings  
8. errors.filter(line => line.contains("MySQL")).collect()
```



# Примеры: Text Search (Python)



```
1. file = spark.textFile("hdfs://...")  
2. errors = file.filter(lambda line: "ERROR" in line)  
  
3. # Count all the errors  
4. errors.count()  
  
5. # Count errors mentioning MySQL  
6. errors.filter(lambda line: "MySQL" in line).count()  
  
7. # Fetch the MySQL errors as an array of strings  
8. errors.filter(lambda line: "MySQL" in line).collect()
```



# Примеры: Text Search (Java)



```
1. JavaRDD<String> file = spark.textFile("hdfs://...");  
2. JavaRDD<String> errors = file.filter(new Function<String,  
Boolean>() {  
3.     public Boolean call(String s) { return s.contains("ERROR");  
4. }  
5. });  
  
5. // Count all the errors  
6. errors.count();  
  
7. // Count errors mentioning MySQL  
8. errors.filter(new Function<String, Boolean>() {  
9.     public Boolean call(String s) { return s.contains("MySQL");  
10. }}).count();  
  
11. // Fetch the MySQL errors as an array of strings  
12. errors.filter(new Function<String, Boolean>() {  
13.     public Boolean call(String s) { return s.contains("MySQL");  
14. }}).collect();
```



# Примеры: Word Count (Scala)



```
1. val file = spark.textFile("hdfs://...")  
2. val counts = file.flatMap(line => line.split(" "))  
3.           .map(word => (word, 1))  
4.           .reduceByKey(_ + _)  
5. counts.saveAsTextFile("hdfs://...")
```



# Примеры: Word Count (Python)



```
1. file = spark.textFile("hdfs://...")  
2. counts = file.flatMap(lambda line: line.split(" ")) \  
3.           .map(lambda word: (word, 1)) \  
4.           .reduceByKey(lambda a, b: a + b)  
5. counts.saveAsTextFile("hdfs://...")
```



# Примеры: Word Count (Java)



```
1. JavaRDD<String> file = spark.textFile("hdfs://...");  
2. JavaRDD<String> words = file.flatMap(new  
   FlatMapFunction<String, String>() {  
3.     public Iterable<String> call(String s) { return  
   Arrays.asList(s.split(" ")); }  
4. });  
  
5. JavaPairRDD<String, Integer> pairs = words.mapToPair(new  
   PairFunction<String, String, Integer>() {  
6.     public Tuple2<String, Integer> call(String s) { return new  
   Tuple2<String, Integer>(s, 1); }  
7. });  
  
8. JavaPairRDD<String, Integer> counts = pairs.reduceByKey(new  
   Function2<Integer, Integer>() {  
9.     public Integer call(Integer a, Integer b) { return a + b; }  
10.});  
11. counts.saveAsTextFile("hdfs://...");
```



Спасибо за  
внимание!

Клюкин Денис

[d.klyukin@corp.mail.ru](mailto:d.klyukin@corp.mail.ru)

Не забудьте отметиться.