



# ТЕХНОСФЕРА

## Методы распределенной обработки больших объемов данных в Hadoop

Лекция 12: Spark

# 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

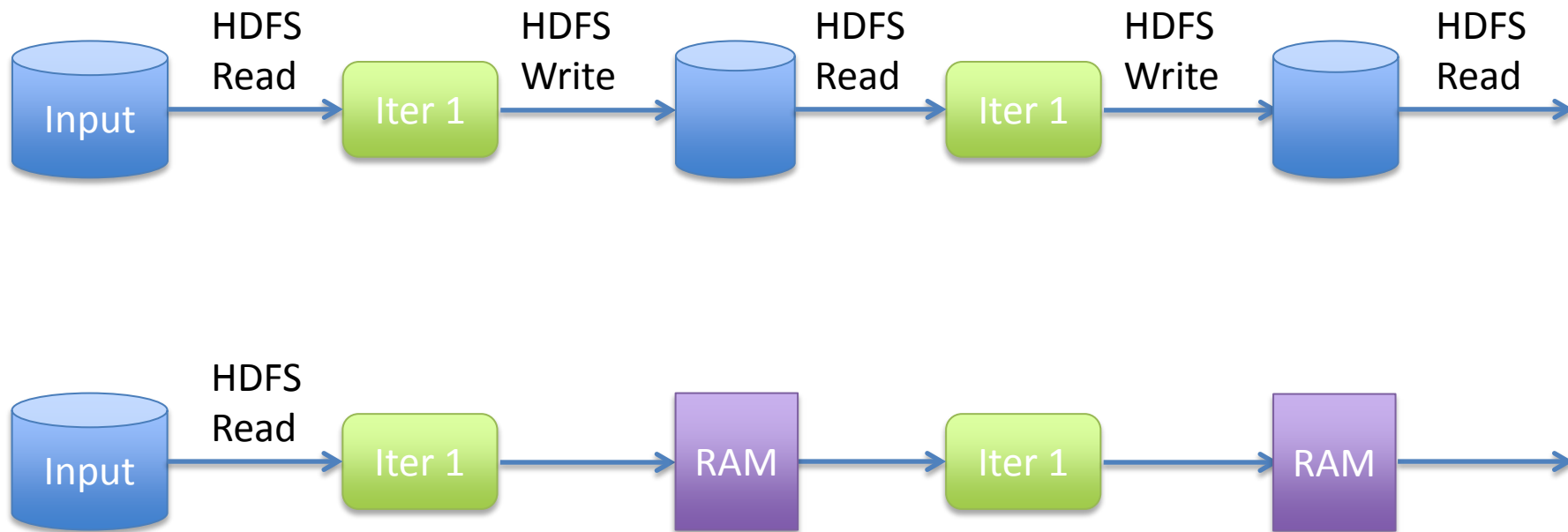
# Мотивация

- 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*

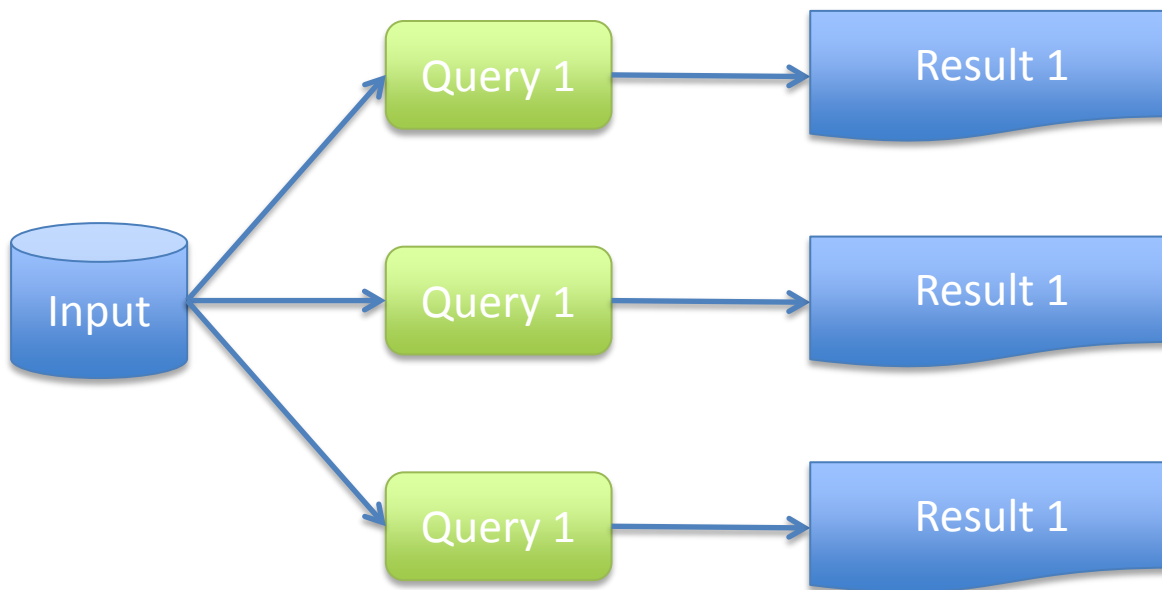
# Решение

## In-Memory Data Processing and Sharing

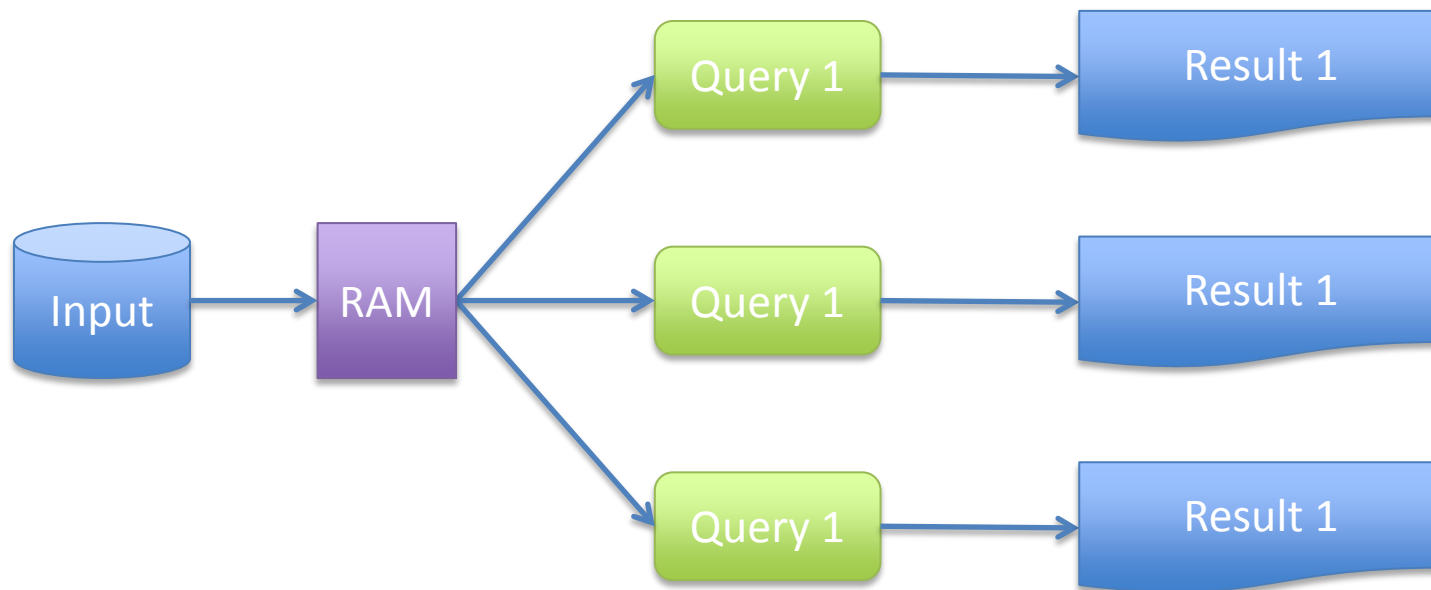
# Пример



# Пример



# Пример



- Задача
  - Разработать дизайн абстракции распределенной памяти с поддержкой **fault tolerant** и **эффективности**
- Решение
  - *Resilient Distributed Datasets (RDD)*



# RDD

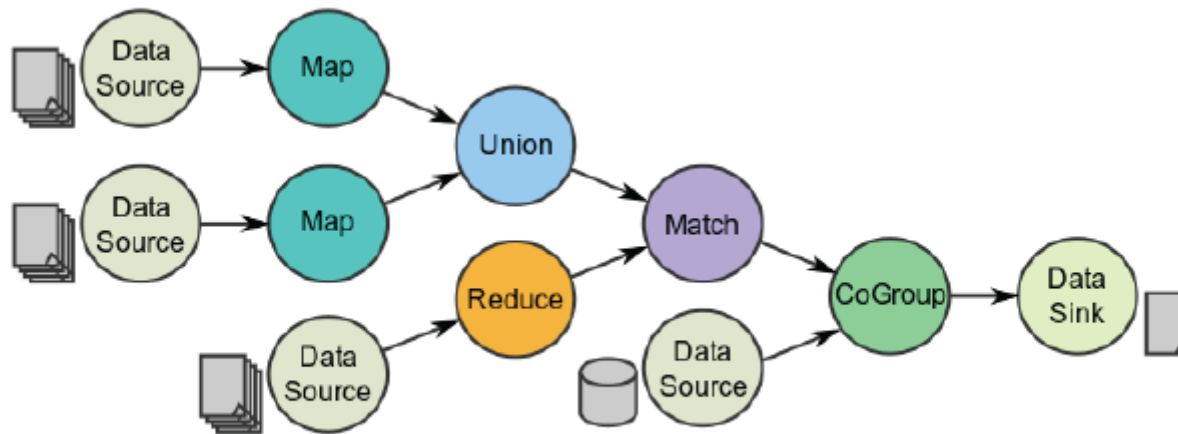
- Абстрактное представление распределенной RAM
- Immutable коллекция объектов распределенных по всему кластеру
- RDD делится на партиции, которые являются атомарными частями информации
- Партиции RDD могут храниться на различных нодах кластера

# 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

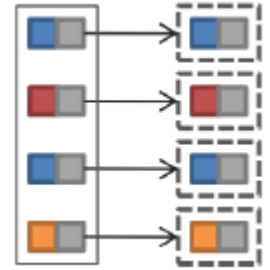
- **Higher-order** functions: RDD операторы
- Существует два типа операторов
  - **transformations** и **actions**
- **Transformations:** lazy-операторы, которые создают новые RDD
- **Actions:** запускают вычисления и возвращают результат в программу или пишут данные во внешнее хранилище

# Higher-Order Functions

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

# RDD Transformations - Map

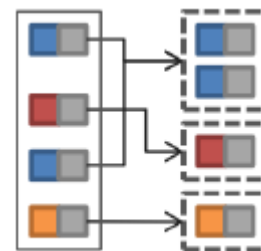
- Все пары обрабатываются  
независимо



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

# RDD Transformations - Reduce

- Пары с одинаковыми ключами группируются
- Группы обрабатываются независимо



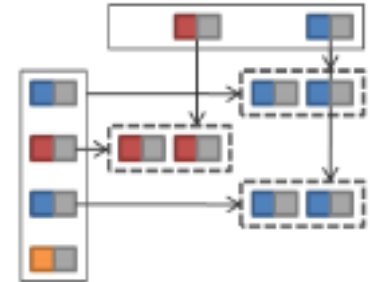
```
val pets = sc.parallelize(Seq(("cat", 1), ("dog", 1), ("cat", 2)))

pets.reduceByKey((x, y) => x + y)
// {(cat, 3), (dog, 1)}

pets.groupByKey()
// {(cat, (1, 2)), (dog, (1))}
```

# RDD Transformations - Join

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

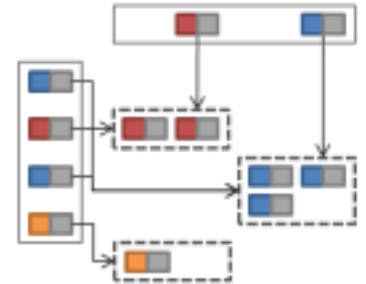


```
val visits = sc.parallelize(Seq(("index.html", "1.2.3.4"),  
                                ("about.html", "3.4.5.6"),  
                                ("index.html", "1.3.3.1")))  
  
val pageNames = sc.parallelize(Seq(("index.html", "Home"),  
                                   ("about.html", "About")))  
  
visits.join(pageNames)  
// ("index.html", ("1.2.3.4", "Home"))  
// ("index.html", ("1.3.3.1", "Home"))  
// ("about.html", ("3.4.5.6", "About"))
```



# RDD Transformations - CoGroup

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



```
val visits = sc.parallelize(Seq(("index.html", "1.2.3.4"),  
                                ("about.html", "3.4.5.6"),  
                                ("index.html", "1.3.3.1")))  
  
val pageNames = sc.parallelize(Seq(("index.html", "Home"),  
                                    ("about.html", "About")))  
  
visits.cogroup(pageNames)  
// ("index.html", (("1.2.3.4", "1.3.3.1"), ("Home")))  
// ("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 сэмплы записей

# RDD Actions

- Возвращает все элементы RDD в виде массива

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

- Возвращает массив с первыми n элементами RDD

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

- Возвращает число элементов в RDD

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

# RDD Actions

- Агрегирует элементы RDD используя заданную функцию

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

- Записывает элементы RDD в виде текстового файла


```
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

```
import org.apache.spark.SparkContext  
import org.apache.spark.SparkContext._
```

```
val sc = new SparkContext(master, appName, [sparkHome], [jars])
```



```
local  
local[k]  
spark://host:port  
mesos://host:port
```

# Создание RDD

- Преобразовать коллекцию в RDD

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

- Загрузить текст из локальной FS, HDFS или S3

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

# Пример

- Посчитать число строк содержащих **MAIL**

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

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



# Shared Variables

- Когда Spark запускает выполнение функции параллельно как набор задач на различных нодах, то отправляется копия каждой переменной, используемой в функции, на каждый task
- Иногда нужно, чтобы переменная была общая между задачами или между задачами программой-драйвером



# Shared Variables

- Обновления переменных не распространяются обратно в программу-драйвер
- Использование обычных *read-write* общих переменные между задачами неэффективно
  - К примеру, необходимо отправить на каждую ноду большой датасет
- 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 task
- *Broadcast Variables* не отсылаются на ноду больше **одного раза**

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

scala> broadcastVar.value
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 **+=**

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

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

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

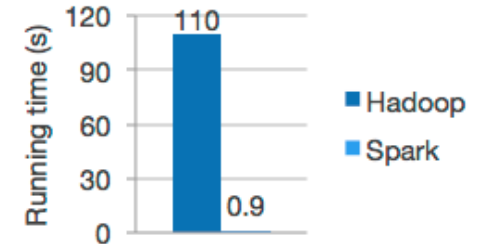
# Apache Spark Engine



- *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



Logistic regression in Hadoop and Spark

```
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

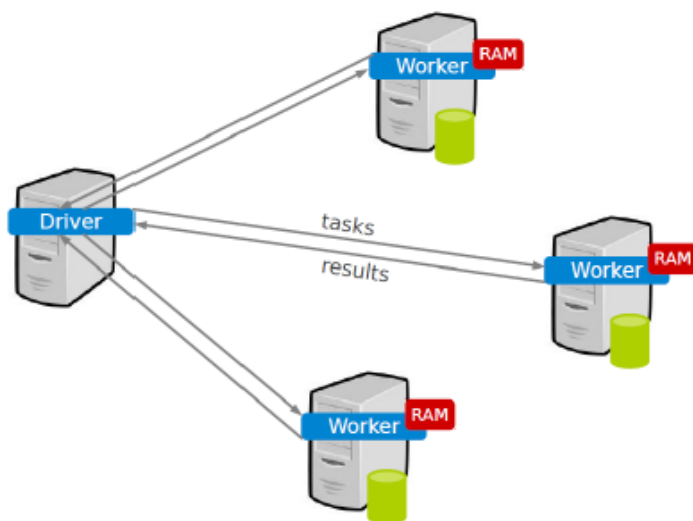
# Apache Spark

- 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

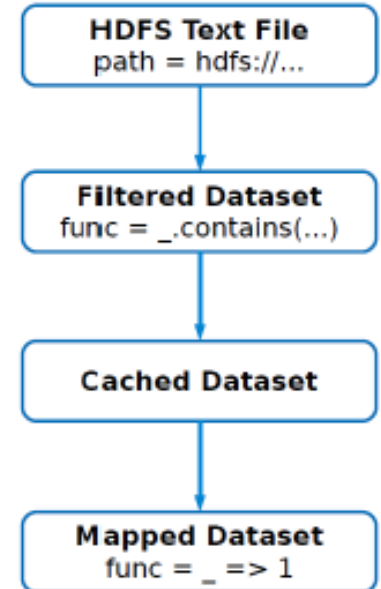
- Приложение на Spark состоит из **программы-драйвера**, которая запускает пользовательскую функцию *main* и выполняет различные операции параллельно на кластере



# Lineage

- **Lineage:** это transformations, используемые для построения RDD
- **RDD** сохраняются как цепочка объектов, охватывающих весь **lineage** каждого RDD

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



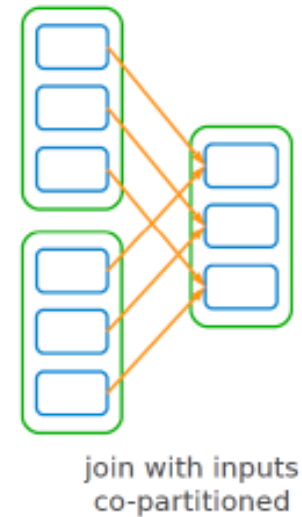
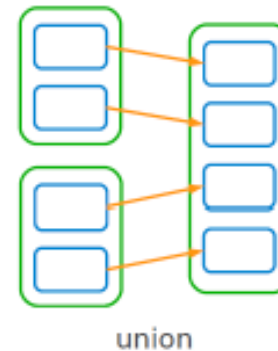
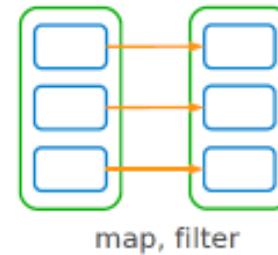


# Dependencies RDD

- Два типа зависимостей между RDD
  - **Narrow**
  - **Wide**

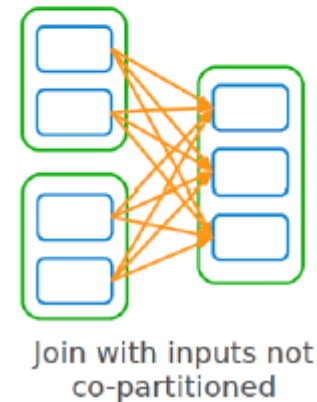
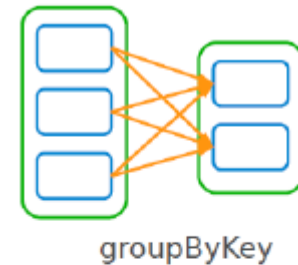
# Dependencies RDD: Narrow

- **Narrow:** каждая партиция родительского RDD используется как минимум в одной дочерней партиции RDD
- *Narrow dependencies* позволяют выполнять **pipelined execution** на одной ноде кластера:
  - Напр., фильтр следующий за Map



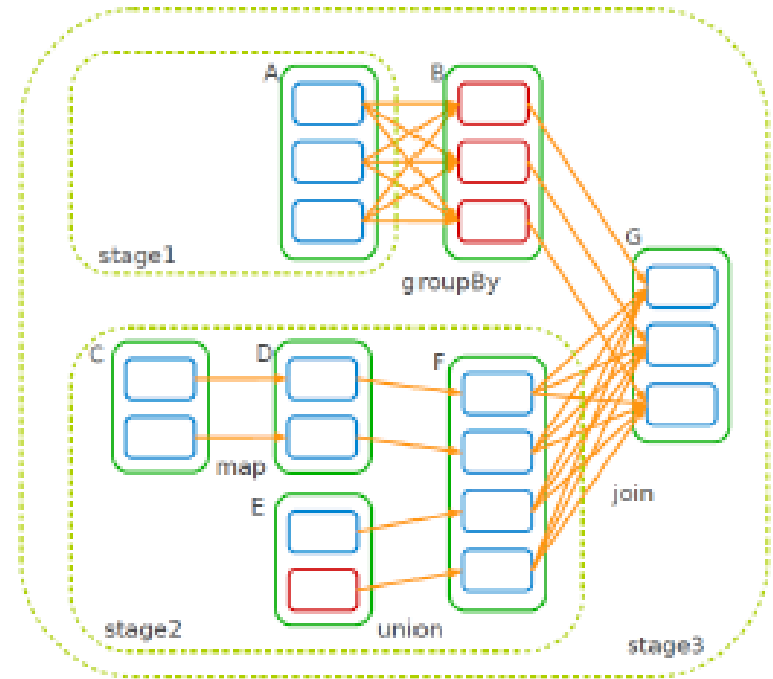
# Dependencies RDD: **Wide**

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



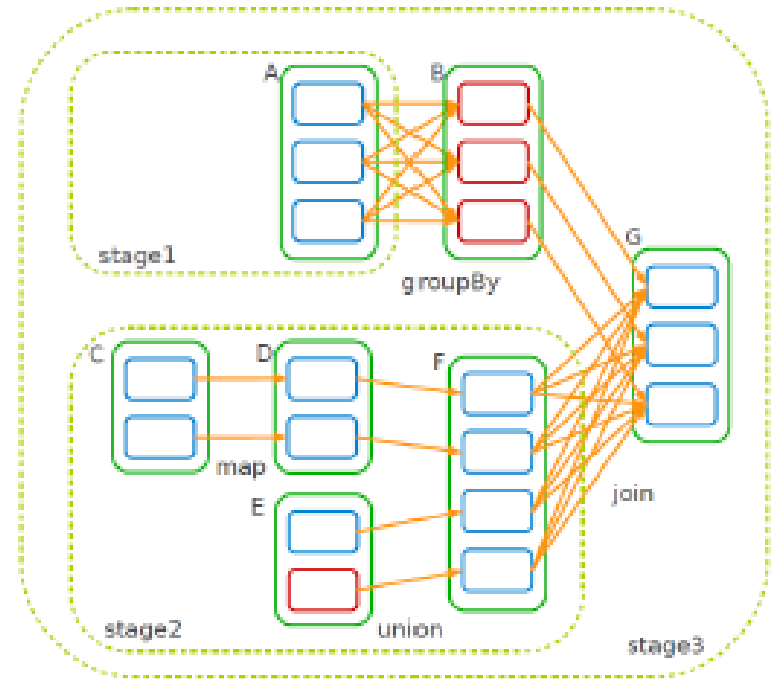
# Job Scheduling

- 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 задачи для обработки оставшихся партиций (***missing partitions***) из каждой ***stage*** пока не обрабатается **целевая (*target*) RDD**
- Задачи назначаются машинам на основе **локальности**
  - Если задаче требуется партиция, которая доступна в памяти на ноду, то задача 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 партии 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 все элемента из набора данных
- Приложения, которые **не подходят** для RDD
  - Приложения, которые выполняют ***asynchronous ne-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)

```
val file = spark.textFile("hdfs://...")
val errors = file.filter(line => line.contains("ERROR"))

// Count all the errors
errors.count()

// Count errors mentioning MySQL
errors.filter(line => line.contains("MySQL")).count()

// Fetch the MySQL errors as an array of strings
errors.filter(line => line.contains("MySQL")).collect()
```

# Примеры: Text Search (Python)

```
file = spark.textFile("hdfs://...")
errors = file.filter(lambda line: "ERROR" in line)

# Count all the errors
errors.count()

# Count errors mentioning MySQL
errors.filter(lambda line: "MySQL" in line).count()

# Fetch the MySQL errors as an array of strings
errors.filter(lambda line: "MySQL" in line).collect()
```

# Примеры: Text Search (Java)

```
JavaRDD<String> file = spark.textFile("hdfs://...");  
JavaRDD<String> errors = file.filter(new Function<String, Boolean>() {  
    public Boolean call(String s) { return s.contains("ERROR"); }  
});
```

```
// Count all the errors  
errors.count();
```

```
// Count errors mentioning MySQL  
errors.filter(new Function<String, Boolean>() {  
    public Boolean call(String s) { return s.contains("MySQL"); }  
}).count();
```

```
// Fetch the MySQL errors as an array of strings  
errors.filter(new Function<String, Boolean>() {  
    public Boolean call(String s) { return s.contains("MySQL"); }  
}).collect();
```

# Примеры: Word Count (Scala)

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



# Примеры: Word Count (Python)

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

# Примеры: Word Count (Java)

```
JavaRDD<String> file = spark.textFile("hdfs://...");  
JavaRDD<String> words = file.flatMap(new FlatMapFunction<String, String>() {  
    public Iterable<String> call(String s) { return Arrays.asList(s.split(" ")); }  
});
```

```
JavaPairRDD<String, Integer> pairs = words.mapToPair(new  
PairFunction<String, String, Integer>() {  
    public Tuple2<String, Integer> call(String s) { return new Tuple2<String,  
Integer>(s, 1); }  
});
```

```
JavaPairRDD<String, Integer> counts = pairs.reduceByKey(new  
Function2<Integer, Integer>() {  
    public Integer call(Integer a, Integer b) { return a + b; }  
});  
counts.saveAsTextFile("hdfs://...");
```

# Вопросы?

