

Лекция 12

Spark



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

Но с ростом популярности фреймворка, пользователи предъявляют все большие требования к нему:

- **Выполнение итеративных** задач, например, алгоритмы machine learning
- **Интерактивной** аналитики
- **Realtime** обработки



Для решения этих 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*

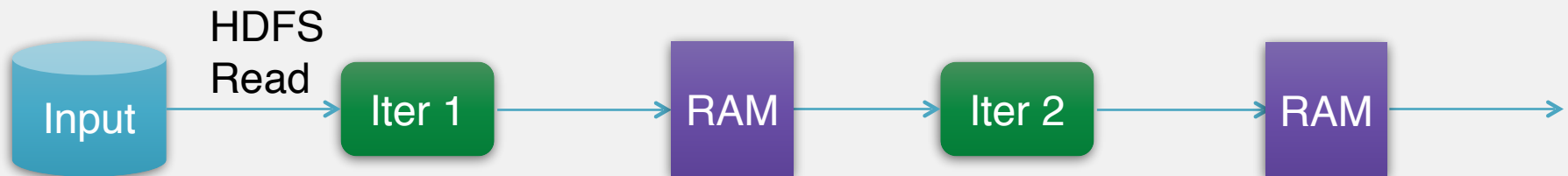
Выполнение последних *map* задач может растягиваться надолго и приводить к замедлению и простоям ресурсов кластера



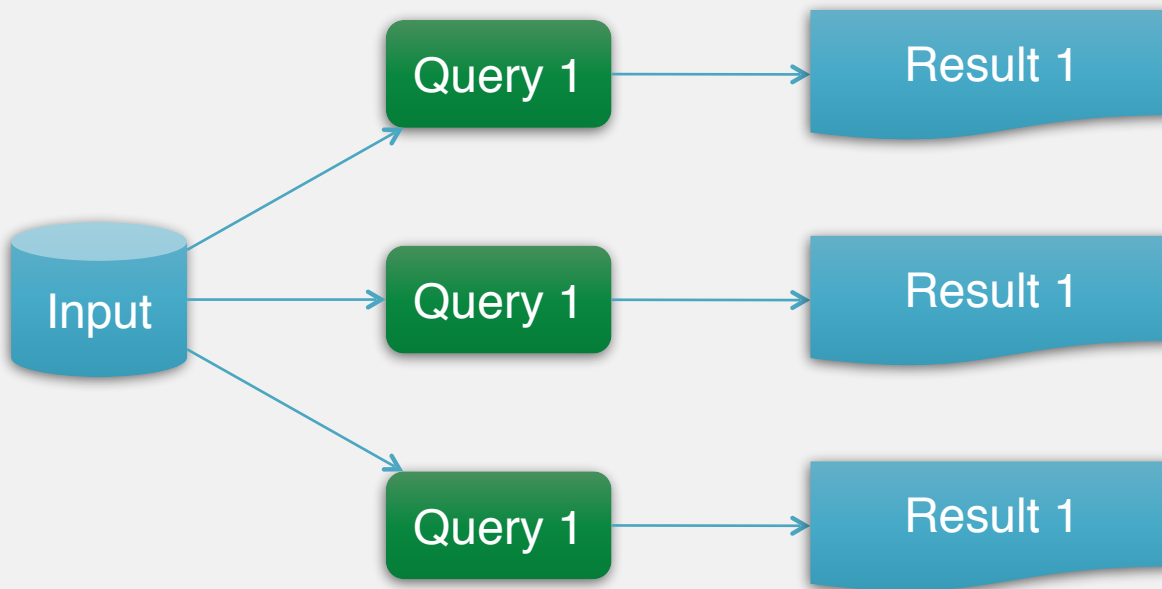
In-Memory Data Processing and Sharing



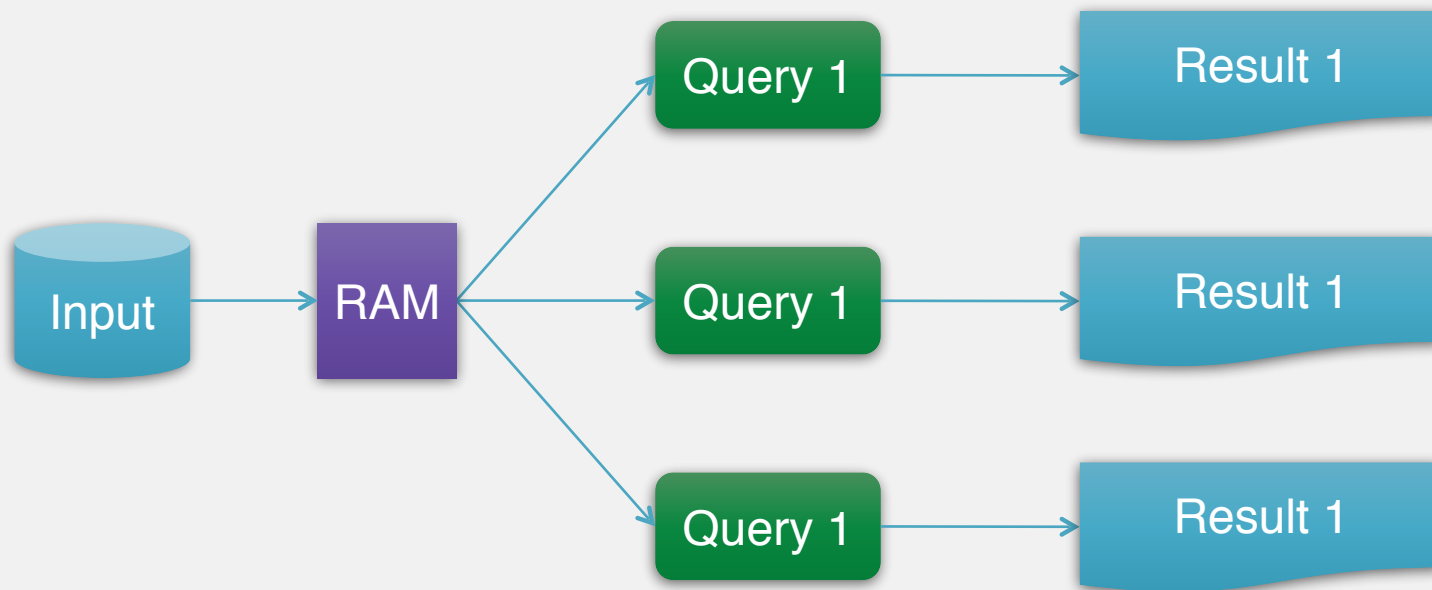
Пример



Пример



Пример





Задача

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

Решение

- *Spark !*
- *Resilient Distributed Datasets (RDD)*

Apache Spark Engine



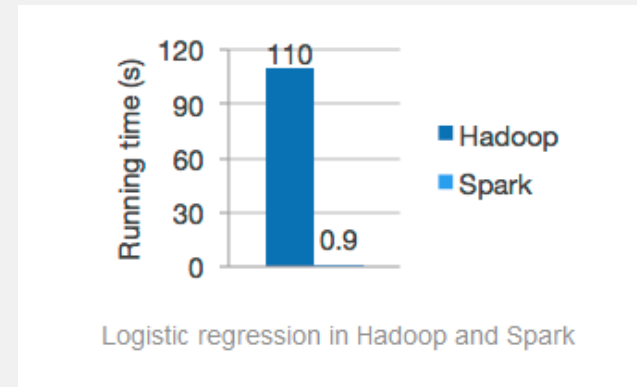
- *Lightning-fast cluster computing!*
- *Apache Spark* – быстрый и многоцелевой «движок» для обработки больших объемов данных
- Предоставляет программный интерфейс на большинстве популярных языков
- Каждый RDD является объектом в Spark





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, Python и даже R!
 - Более 80 высокоуровневых операторов для построения параллельных приложений
 - Их можно использовать интерактивно в spark-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

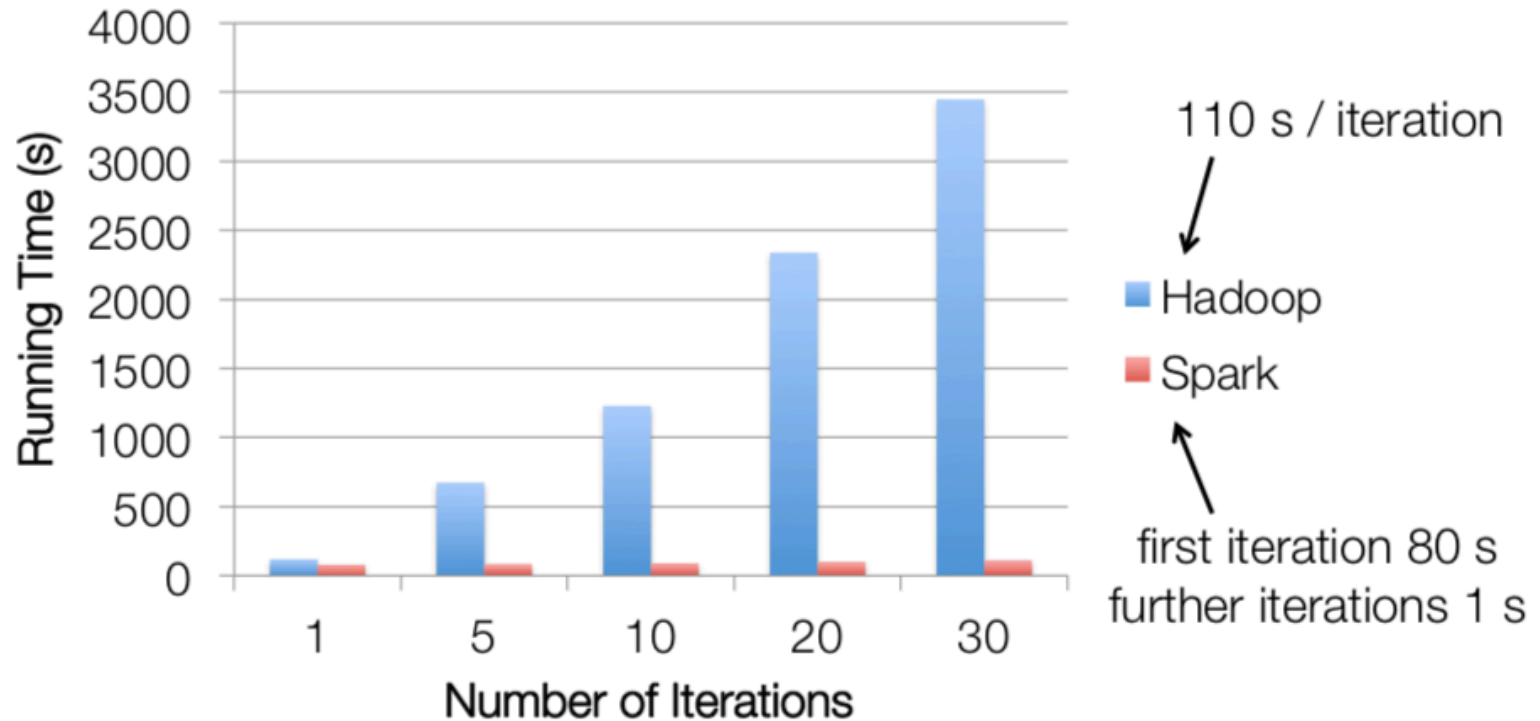


Spark Logistic Regression



```
1. val points = spark.textFile(...).map(parsePoint).cache()
2. var w = Vector.zeros(d)
3. for (i <- 1 to numIterations) {
4.   val gradient = points.map { p =>
5.     (1 / (1 + exp(-p.y * w.dot(p.x))) - 1) * p.y * p.x)
6.     .reduce(_ + _)
7.   w -= alpha * gradient
8. }
9. }
```

Spark Logistic Regression



100 GB of data on 50 m1.xlarge EC2 machines



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

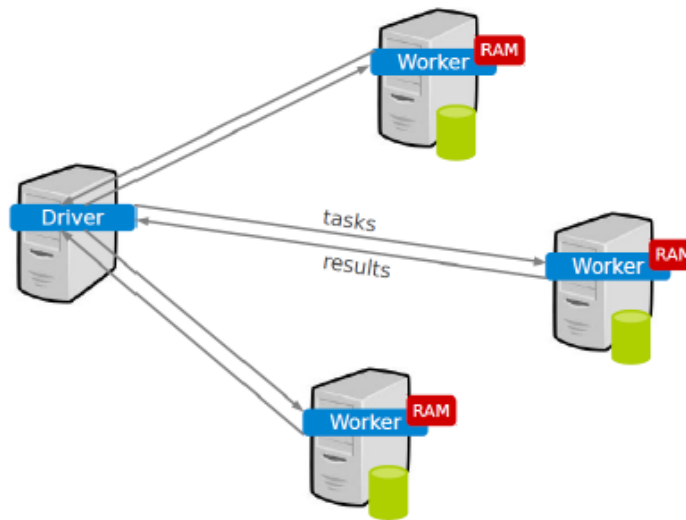
<https://spark.apache.org/>



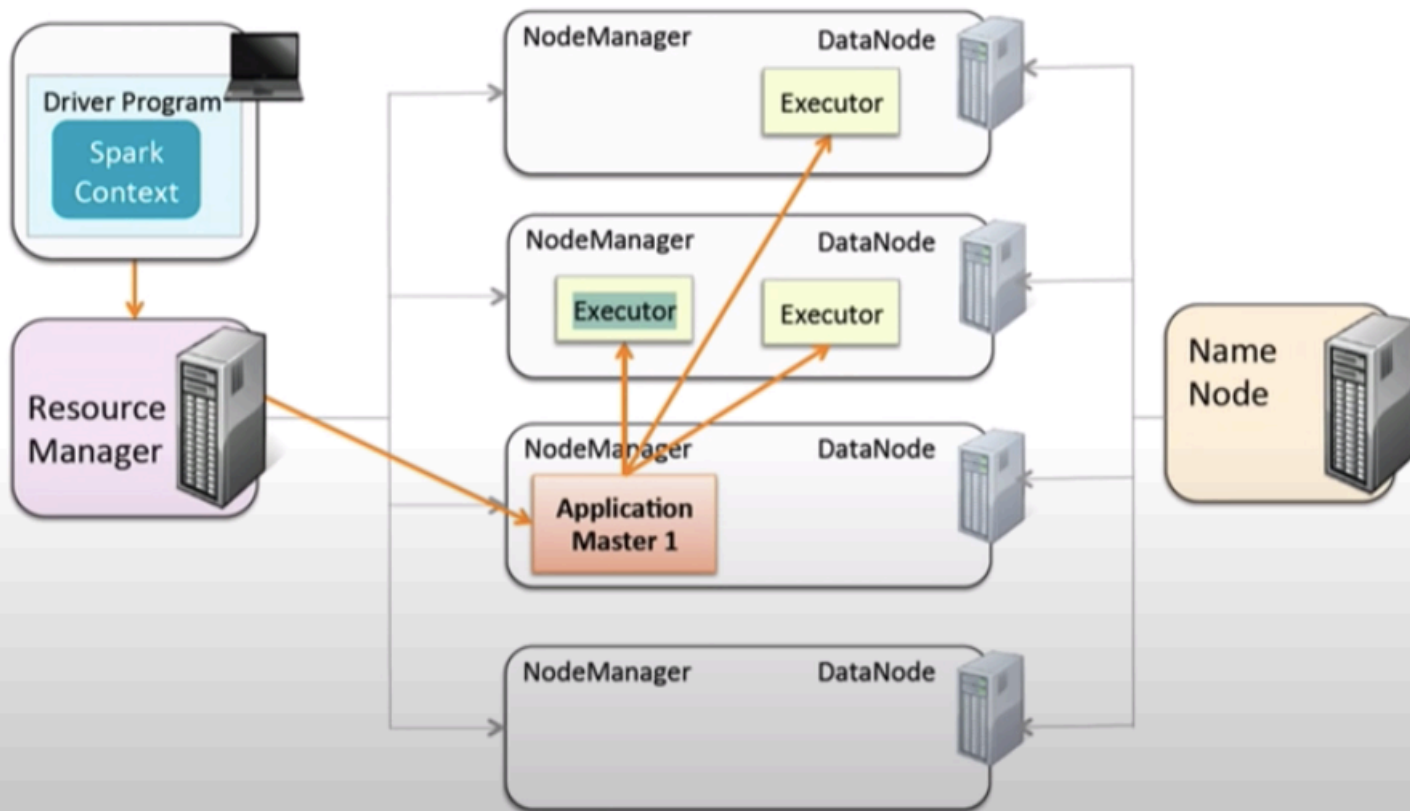
Программная архитектура Spark



Приложение на Spark состоит из **программы-драйвера**, которая запускает пользовательскую функцию *main* и выполняет различные операции параллельно на кластере, и нод-воркеров, на которых непосредственно происходит вся обработка.



Архитектура Spark подробней (YARN)

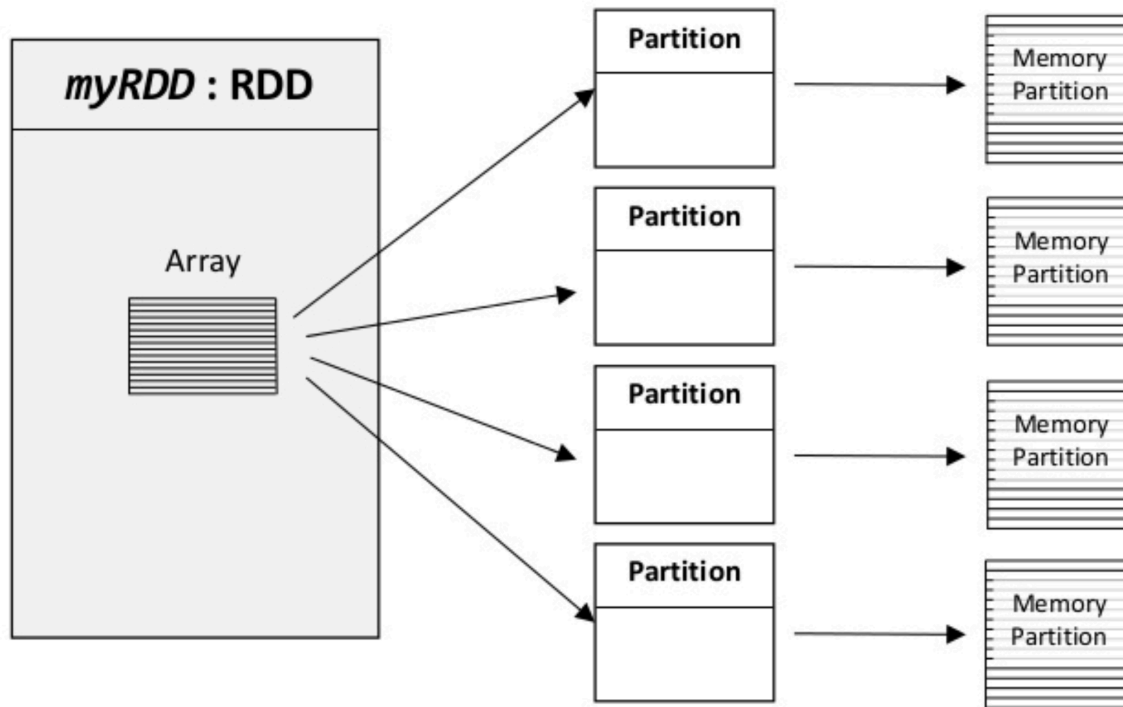


RDD



- Абстрактное представление распределенной RAM
- **Immutable** коллекция объектов распределенных по всему кластеру
- RDD делится на партии(**partiotion**), которые являются атомарными частями информации
- Партии RDD могут хранится на различных нодах кластера

RDD. На пальцах



Some RDD Characteristics

- Hold references to Partition objects
- Each Partition object references a subset of your data
- Partitions are assigned to nodes on your cluster
- Each partition/split will be in RAM (by default)

Программная модель Spark

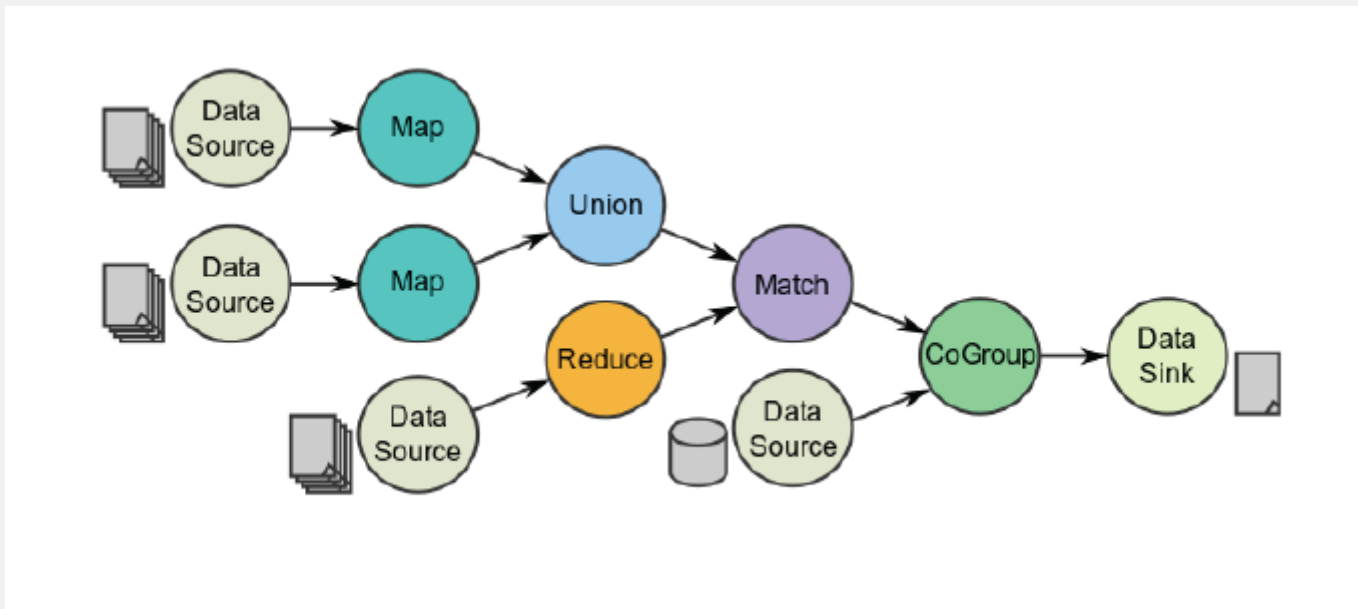


- Основана на **parallelizable operators**
- Эти операторы являются функциями высокого порядка, которые выполняют *user-defined* функции параллельно
- Поток обработки данных состоит из любого числа **data sources, operators** и **data sinks** путем соединения выходов(*outputs*) одних операций со входами(*inputs*) других

Программная модель Spark



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



Программная модель Spark. Lazy Evaluation



Программная модель спарка основана на ленивых вычислениях. В момент выполнения задачи, для конкретного RDD не начнут выполняться преобразования(*transformations*), до тех пор, пока для него не будет вызвано хотя бы одно действие(*action*).



Higher-Order Functions



Higher-order functions: RDD операторы

Существует два типа операторов

- **transformations** и **actions**

Transformations: lazy-операторы, которые создают новые RDD. В каком-то смысле это операторы, выполняющие преобразования вида RDD -> RDD.

Actions: запускают вычисления и возвращают результат в программу или пишут данные во внешнее хранилище. Или это операторы, выполняющие преобразования вида RDD -> реальные данные.

Higher-Order Functions



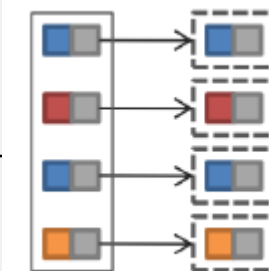
Transformations	<i>map</i> (<i>f</i> : <i>T</i> ⇒ <i>U</i>)	: RDD[<i>T</i>] ⇒ RDD[<i>U</i>]
	<i>filter</i> (<i>f</i> : <i>T</i> ⇒ Bool)	: RDD[<i>T</i>] ⇒ RDD[<i>T</i>]
	<i>flatMap</i> (<i>f</i> : <i>T</i> ⇒ Seq[<i>U</i>])	: RDD[<i>T</i>] ⇒ RDD[<i>U</i>]
	<i>sample</i> (<i>fraction</i> : Float)	: RDD[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



RDD Transformations - Map



Все пары обрабатываются независимо



```
1. SparkConf conf = new SparkConf("Test");
2. JavaSparkContext sc = new JavaSparkContext(conf);

3. // passing each element through a function.
4. List<Integer> data = Arrays.asList(1, 2, 3, 4, 5);
5. JavaRDD<Integer> distData = sc.parallelize(data);

6. // {1, 4, 9}
7. JavaRDD<Integer> squares = nums.map(x -> { return x * x;})
8. // selecting those elements that func returns true.
9. JavaRDD<Integer> even = squares.filter(x -> { return x % 2 ==
0;}) // {4}

10. // mapping each element to zero or more others.
11. JavaRDD<String> textFile = sc.textFile("hdfs://...");
12. JavaPairRDD<String, Integer> counts = textFile.flatMap(s ->
Arrays.asList(s.split(" ")).iterator()) // {word1,word2,..}
```

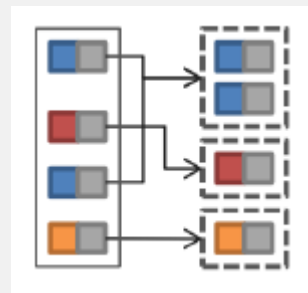


RDD Transformations - Reduce



- Пары с одинаковыми ключами группируются
- Группы обрабатываются независимо

```
1. List<Tuple2<String,Integer>> pairs = ... //  
   создаем список и заполняем парами вида (("cat",  
   1), ("dog", 1), ("cat", 2)  
  
2. JavaPairRDD<String,Integer> pets =  
   sc.parallelizePairs(pairs);  
  
3. JavaPairRDD<String,Integer> petsReduced =  
   pets.reduceByKey((x, y) -> x + y)  
4. // {(cat, 3), (dog, 1)}  
  
5. JavaPairRDD<String,Integer> petsGrouped =  
   pets.groupByKey()  
6. // {(cat, (1, 2)), (dog, (1))}
```





RDD Transformations - Join

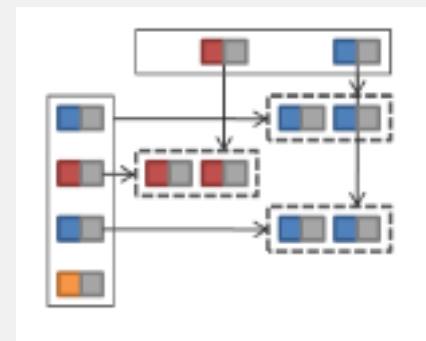


- Выполняется *equal-join* по ключу
- Join-кандидаты обрабатываются независимо

```
1. JavaPairRDD<String,String> visits = sc.parallelize(...);  
2. // ("index.html", "1.2.3.4"),  
3. // ("about.html", "3.4.5.6"),  
4. // ("index.html", "1.3.3.1")
```

```
5. JavaPairRDD<String,String> pageNames =  
  sc.parallelize(...);  
6. // ("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

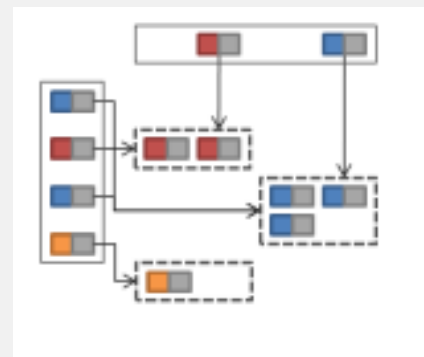


- Каждый *input* группируется **по ключу**
- Группы с одинаковыми ключами обрабатываются вместе

```
1. JavaPairRDD<String,String> visits = sc.parallelize(...);
2. // ("index.html", "1.2.3.4"),
3. // ("about.html", "3.4.5.6"),
4. // ("index.html", "1.3.3.1")

5. JavaPairRDD<String,String> pageNames =
  sc.parallelize(...);
6. // ("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*(можно прокинуть руками) для генератора произвольных чисел для каждой партии чтобы детерминировано выбирать сэмплы записей



Возвращает все элементы RDD в виде массива

```
1. List<Integer> data = Arrays.asList(1, 2, 3, 4, 5);  
2. JavaRDD<Integer> nums = sc.parallelize(data);  
3. nums.collect(); // Array(1, 2, 3)
```

Что если на драйвере не хватит памяти?

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)); // 6
```

Записывает элементы RDD в виде текстового файла

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

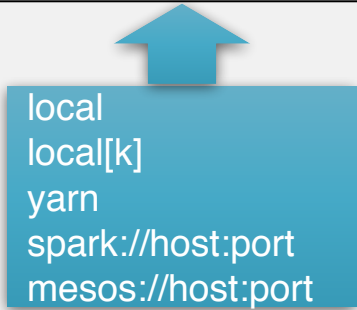
SparkContext



- Основная точка входа для работы со Spark
- В Spark 2 - `SparkSession`
- Доступна в *shell* как переменная **sc**
- В *standalone*-программах необходимо создавать ОТДЕЛЬНО

```
1. import org.apache.spark.api.java.JavaSparkContext

2. JavaSparkContext sc = new
   JavaSparkContext(String master, String appName,
   String sparkHome, String[] jars, ...);
```



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



Преобразовать коллекцию в RDD

```
1. List<Integer> data = Arrays.asList(1, 2, 3, 4, 5);  
2. JavaRDD<Integer> nums = sc.parallelize(data);
```

Загрузить текст из локальной FS, HDFS или S3

```
1. JavaRDD<String> a = sc.textFile("file.txt")  
2. JavaRDD<String> b = sc.textFile("directory/*.txt")  
3. JavaRDD<String> c = sc.textFile("hdfs://host:port/file")
```



Посчитать число строк содержащих **MAIL**

```
1. JavaRDD<String> file = sc.textFile("hdfs://...")
2. JavaRDD<String> sics = file.filter(s ->
   s.contains("MAIL")) // transformation
3. JavaRDD<String> cached = sics.cache()
4. JavaRDD<Integer> ones = cached.map( t -> (1)) //
   transformation
5. Integer count = ones.reduce((i,j) -> (i + j)) // action
```

```
1. JavaRDD<String> file = sc.textFile("hdfs://...")
2. Integer count = file.filter(s ->
   (s.contains("MAIL"))).count()
```



transformation



action

Shared Variables



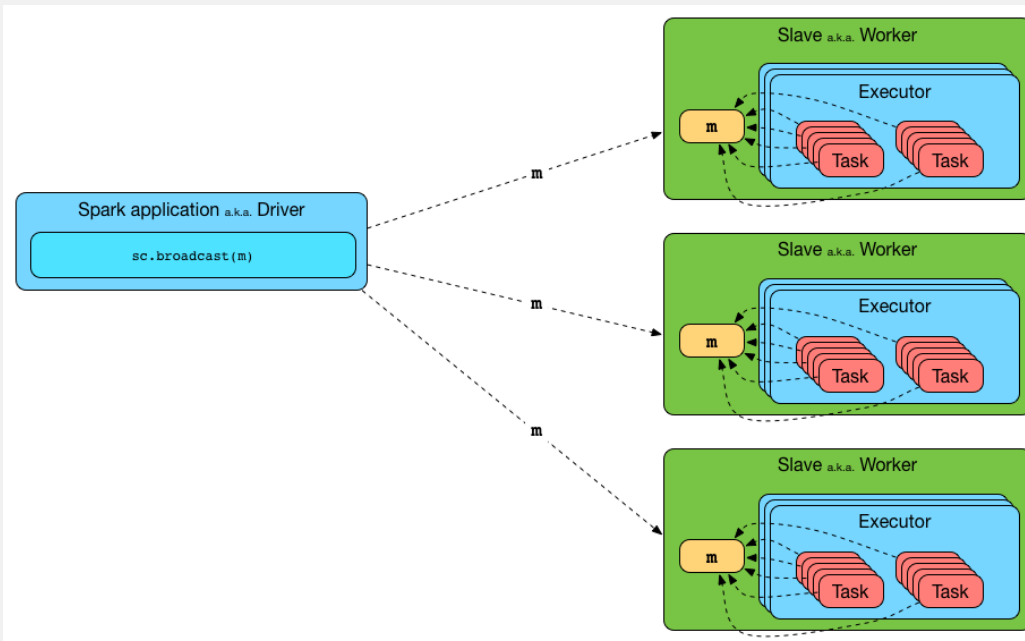
- Когда Spark запускает выполнение функции параллельно как набор задач/операций на различных нодах, то отправляется копия каждой переменной, используемой в функции, на каждый task
- Иногда нужно, чтобы переменная была общая между задачами или между задачами программой-драйвером
- Использование обычных *read-write* общих переменные между задачами 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 каждую операцию

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

```
1. Broadcast<String> broadcastVar = sc.broadcast("Hello Spark");  
2. // код на воркере  
3. broadcastVar.value() // "Hello Spark"
```



Sum Counters



```
1. int counter = 0;
2. JavaRDD<Integer> rdd = sc.parallelize(data);
3.
4.
5. rdd.foreach(x -> counter += x);
6.
7. println("Counter value: " + counter);
8. // What will be printed ???
```

Shared Variables: Accumulators



Могут быть только **инкрементированы**

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

Можно создавать **кастомные** аккумуляторы, имплементируя AccumulatorV2

Применяются **один раз** в actions

Ошибки при вычислении аккумуляторов не убивают задачу



Accumulators: Примеры



```
1. LongAccumulator accum = jsc.sc().longAccumulator();
2. data.map(x -> { accum.add(x); return f(x); });
3. // Here, accum is still 0 because no actions have caused the
   map to be computed.
```

```
1. LongAccumulator accum = jsc.sc().longAccumulator();

3. sc.parallelize(Arrays.asList(1, 2, 3, 4)).foreach(x ->
   accum.add(x));
4. // ...
5. // 10/09/29 18:41:08 INFO SparkContext: Tasks finished in
   0.317106 s

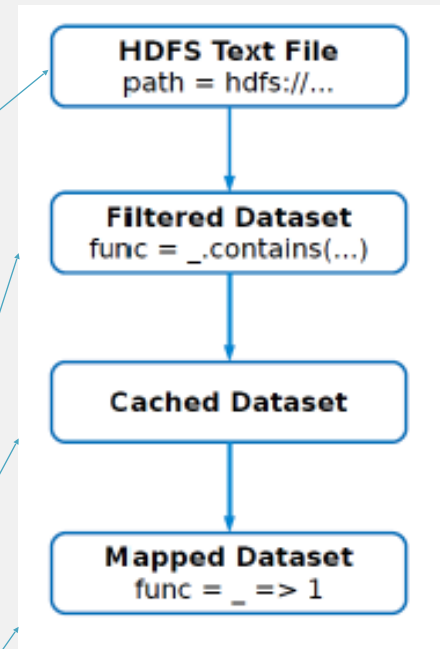
7. accum.value();
8. // returns 10
```



Lineage: это transformations, используемые для построения RDD

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

```
1. JavaRDD<String> file =  
  sc.textFile("hdfs://...")  
2. JavaRDD<String> sics = file.filter(s ->  
  s.contains("MAIL")) // transformation  
3. JavaRDD<String> cached = sics.cache()  
4. JavaRDD<Integer> ones = cached.map( t ->  
  (1)) // transformation  
5. Integer count = ones.reduce((i,j) -> (i  
  + j)) // action
```

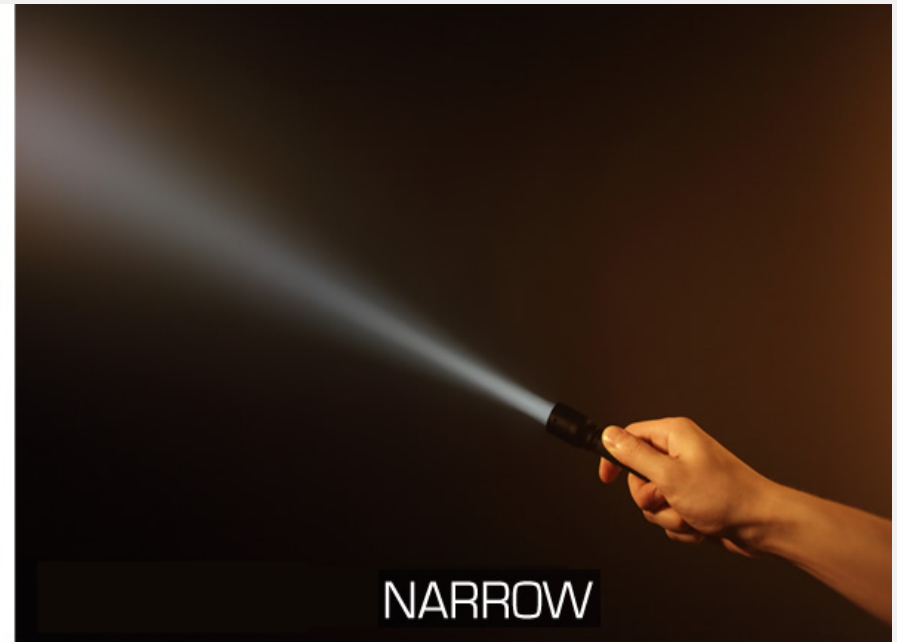


Dependencies RDD



Два типа зависимостей между RDD

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



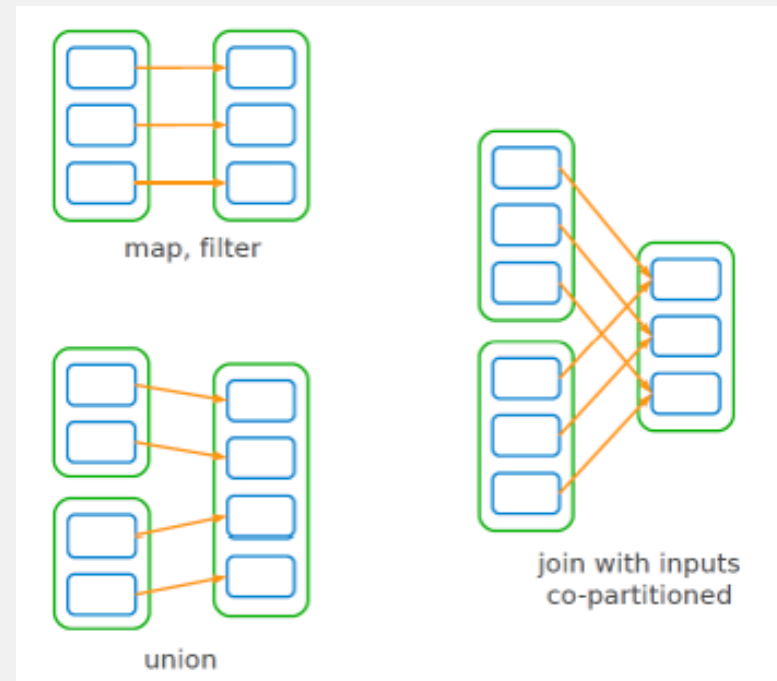
Dependencies RDD: Narrow



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

Narrow dependencies
позволяют выполнять
pipelined execution на
одной ноте кластера:

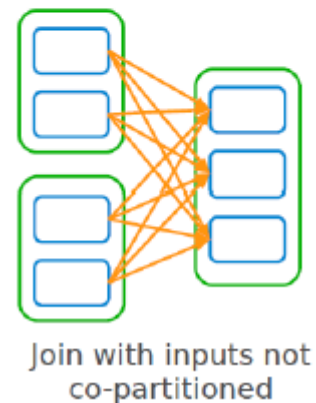
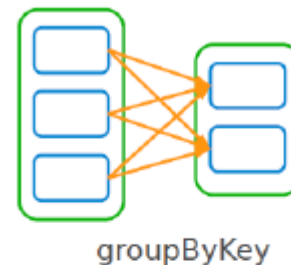
- Напр., фильтр
следуемый за Map



Dependencies RDD: Wide



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



Job Scheduling

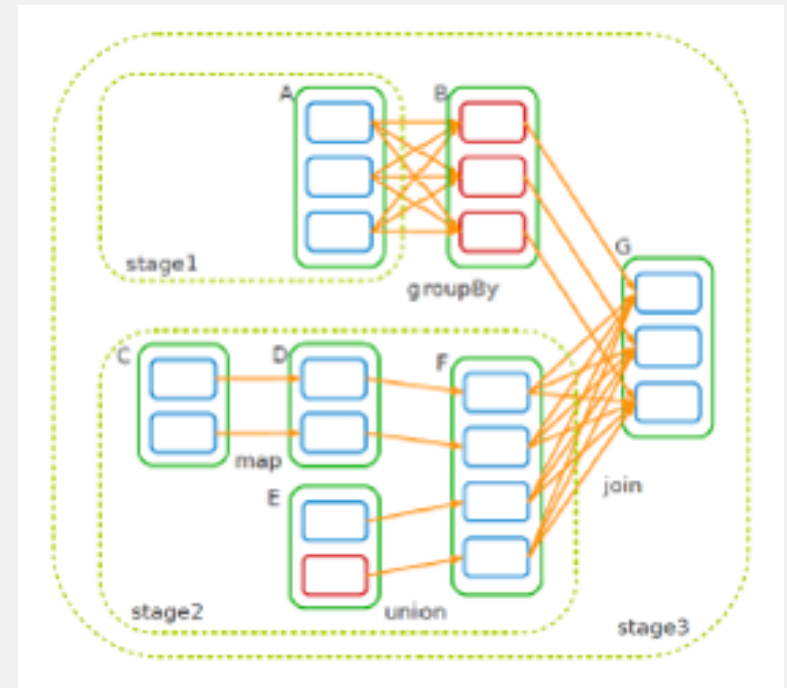


Когда пользователь запускает ***action*** на RDD шедулер строит DAG из ***stages*** графа ***RDD lineage***

Stage содержит различные ***pipelined transformations*** с ***narrow dependencies***

Граница для ***stage***

- *Shuffles* для *wide dependencies*
- Уже обработанные партиции



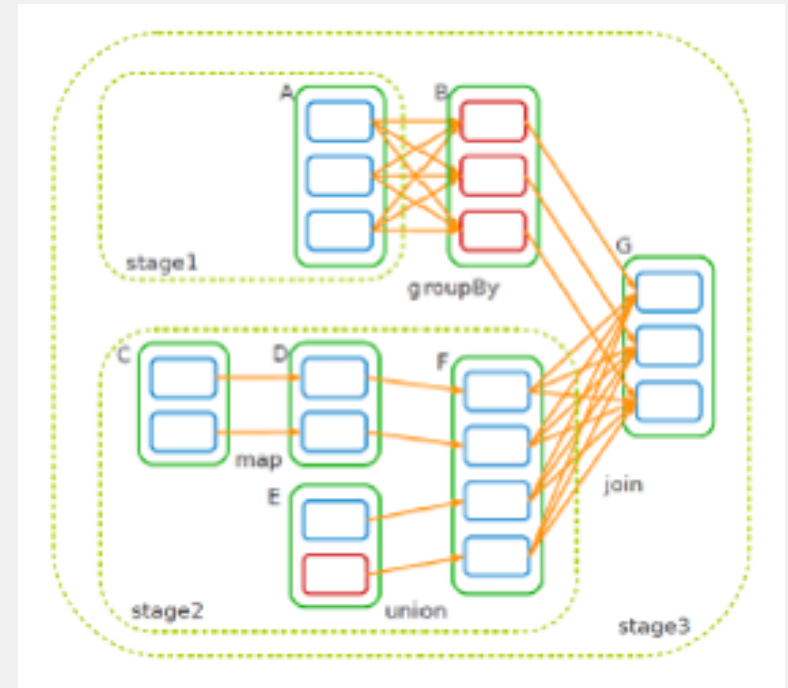
Job Scheduling



Шедулер запускает задачи для обработки оставшихся партиций (***missing partitions***) из каждой ***stage*** пока не обработается **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***
- **Возможность 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 parents*** становятся недоступны, то таски сабмитятся для расчета отсутствующих партиций в параллели

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



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***



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 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)



- ***RDD*** – это распределенная абстракция памяти, которая обладает следующими плюшками:
 - In-memory computation
 - Lazy Evaluation
 - Fault Tolerance
 - Immutability
 - Persistence
 - Partitioning
 - Parallel
 - Location-Stickiness
 - Coarse-grained Operation
 - Typed
 - No limitation
- Два типа операций: ***Transformations*** и ***Actions***
- RDD fault tolerance: ***Lineage***

Spark SQL: DataSets и DataFrames



- С версии 1.3
- Позволяет работать со структурированными данными и обращаться к ним по имени
- Основными структурами данных являются Datasets и DataFrames
- Объединяют возможности RDD (преобразования, функции высших порядков) с возможностями SQL
- Под капотом RDD, но с оптимизациями
- Требуют меньше памяти если по умному их использовать
- Можно создать из Hive таблиц, json подобных файлов, других RDD, внешних баз данных, и из всего имеющего структуру



```
//Читаем из Hive
```

```
1. c = HiveContext(sc)
2. rows = c.sql("select text, year from hive table")
3. rows.filter(lambda r: r.year > 2013).collect()
```

```
1. // Читаем из JSON
```

```
2. c.jsonFile("tweets.json").registerAsTable("tweets")
3. c.sql("select text, user.name from tweets")
```

```
tweets.json
{"text": "hi",
 "user": {
   "name": "matei",
   "id": 123 }}
```



Методы: `agg`, `columns`, `count`, `distinct`, `drop`, `dropDuplicates`, `filter`, `groupBy`, `orderBy`, `registerTable`, `schema`, `show`, `select`, `where`, `withColumns`

```
6. // Run SQL statements
7. val teenagers = context.sql(
8.     "SELECT name FROM people WHERE age >= 13 AND age <= 19")

12. // The results of SQL queries are RDDs of Row objects
13. val names = teenagers.map(t => "Name: " + t(0)).collect()
```

Практика





Примеры: Text Search (Python)



```
1. file = sc.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 = sc.textFile("hdfs://...");
2. JavaRDD<String> errors = file
3.     .filter(s -> s.contains(«ERROR»));
4.
5. // Count all the errors
6. errors.count();

7. // Count errors mentioning MySQL
8. errors.filter(s -> s.contains("MySQL")).count();

11. // Fetch the MySQL errors as an array of strings
12. errors.filter(s -> s.contains("MySQL")).collect();
```



Примеры: Word Count (Python)



```
1. file = sc.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 = sc.textFile("hdfs://...");
2. JavaRDD<String> words = file.flatMap(s ->
3.     Arrays.asList(s.split(" ")));

5. JavaPairRDD<String, Integer> pairs = words.mapToPair(w ->
6.     new Tuple2<String, Integer>(w, 1));

8. JavaPairRDD<String, Integer> counts = pairs
9.     .reduceByKey((a, b) -> a + b);
10. counts.saveAsTextFile("hdfs://...");
```



Спасибо за внимание!

Смирнов Даниил

dr.smirnov@corp.mail.ru

Не забудьте отметить