Интеллектуальные информационные системы

Фреймворк Spark

Кафедра управления и интеллектуальных технологий НИУ «МЭИ»

2023 г.

Фреймворк Spark

Lightning-fast cluster computing

Фреймворк для обработки больших объемов данных.

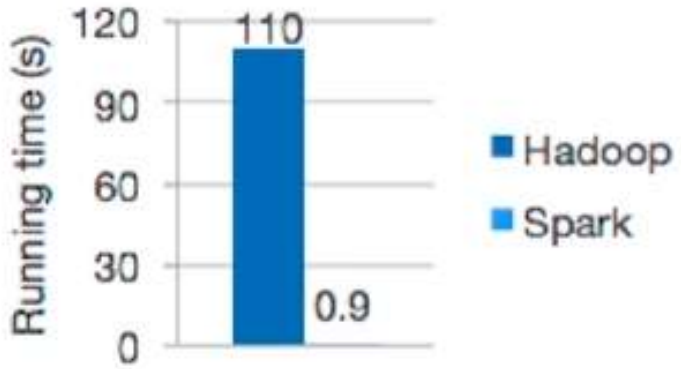
Имеет парадигму отличную от MapReduce

Написан на языке Scala

Разработан в 2009 году Matei Zaharia в UoC Berkley

https://spark.apache.org/

Фреймворк Spark



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

Фреймворк Spark

Подходит для итеративных задач
и интерактивных вычислений.

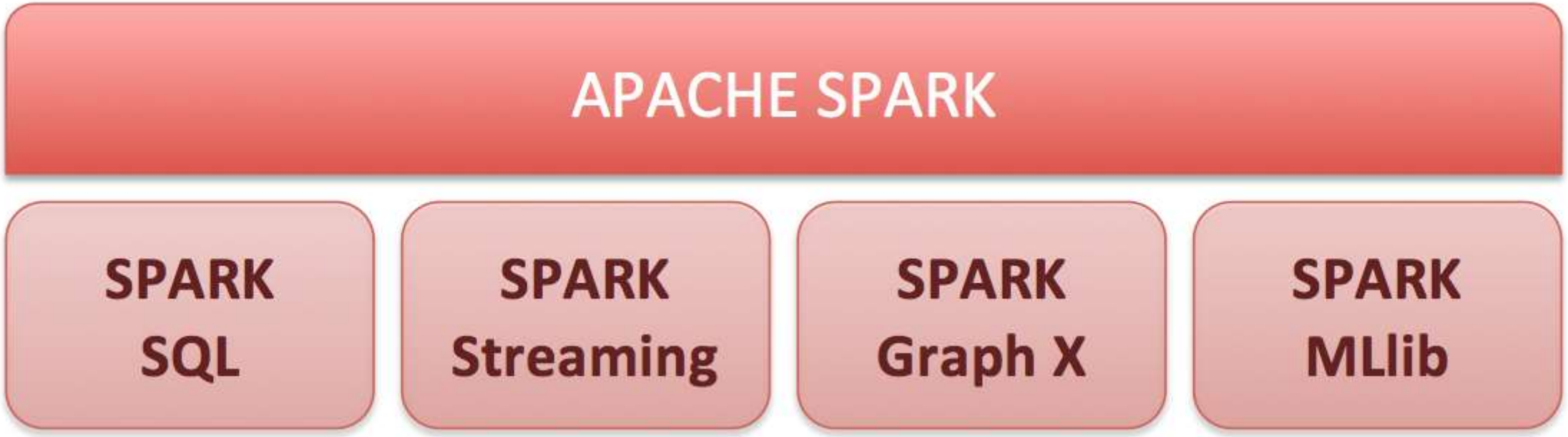
Позволяет обрабатывать данные как по частям (batch), так и потоковые (streaming)

Встроенные инструменты для работы с разными типами данных – текст, графы, базы данных

Имеет более 80 встроенных высокоуровневых функций для обработки данных (кроме map и reduce)

https://spark.apache.org/

Фреймворк Spark

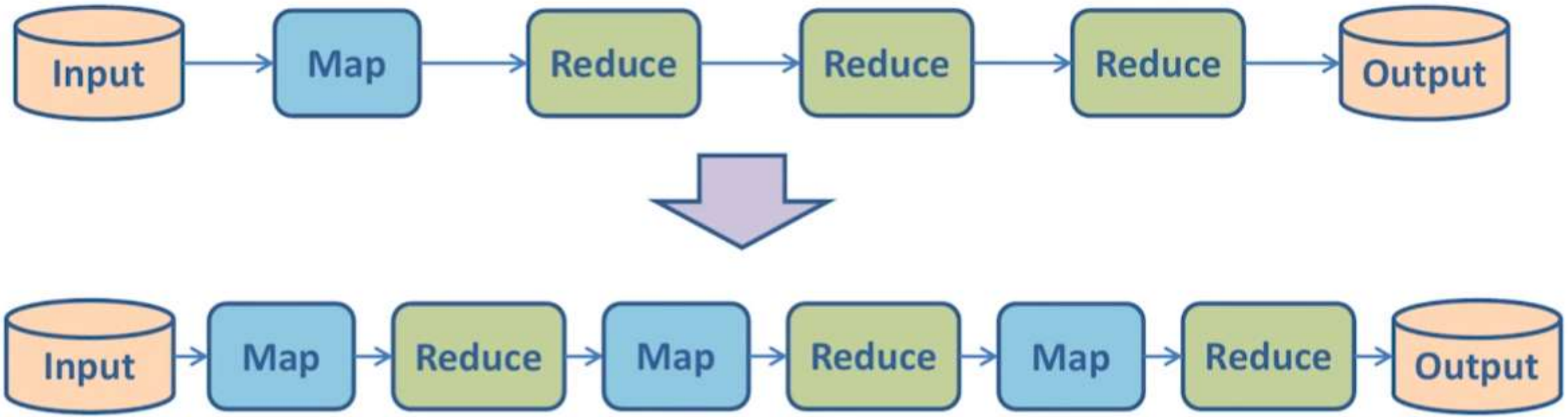


Недостатки MapReduce

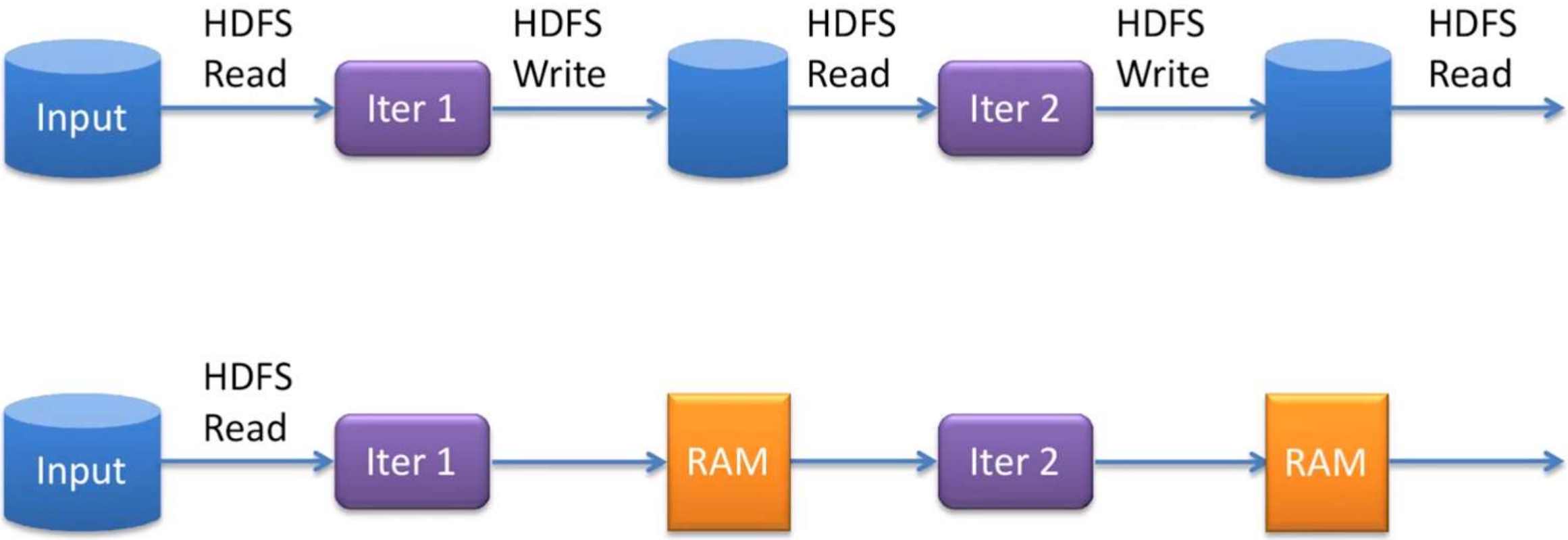
Нет эффективных примитивов для общих данных

Недостатки MapReduce

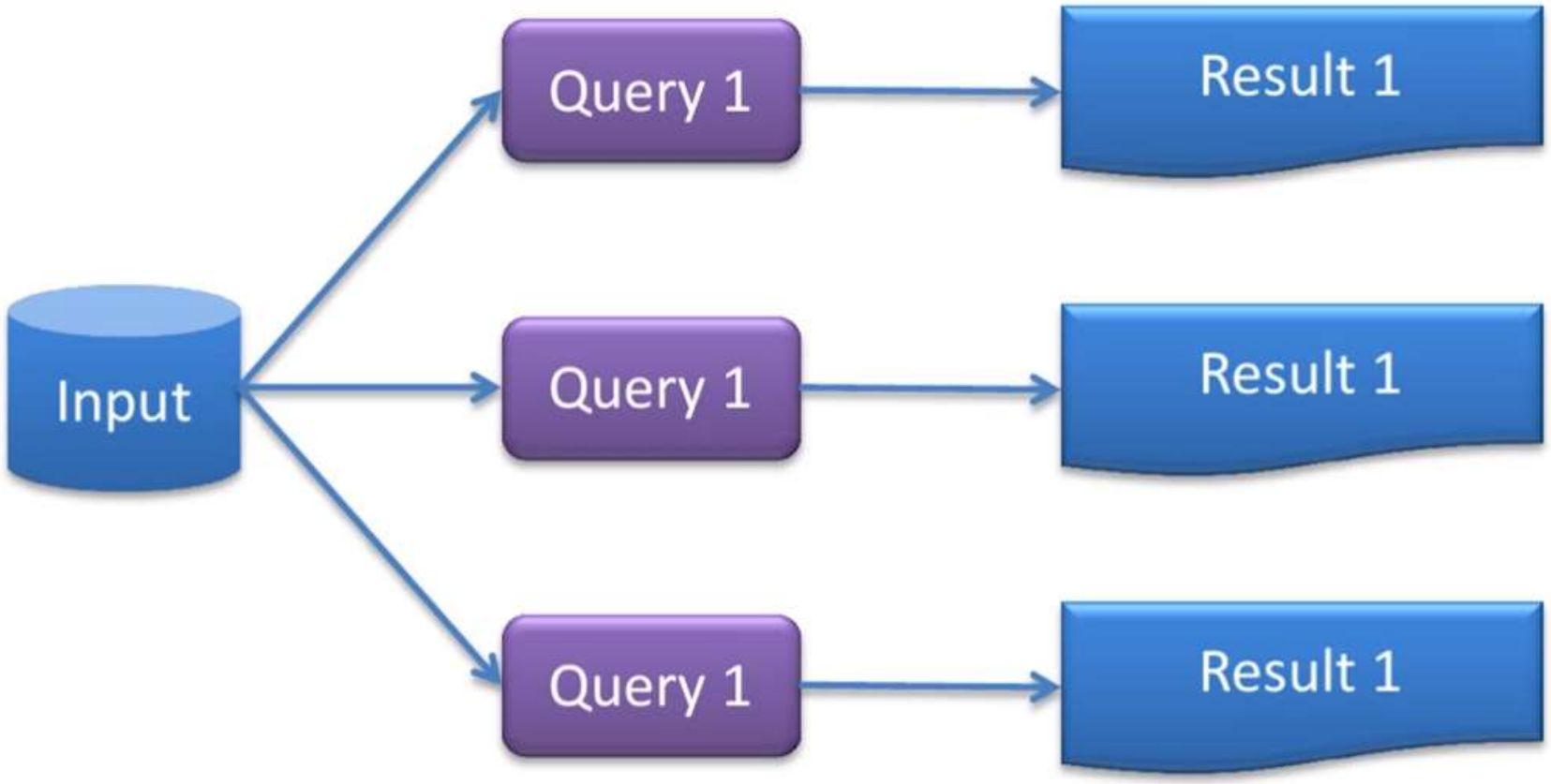
Необходимость применять шаблон MapReduce



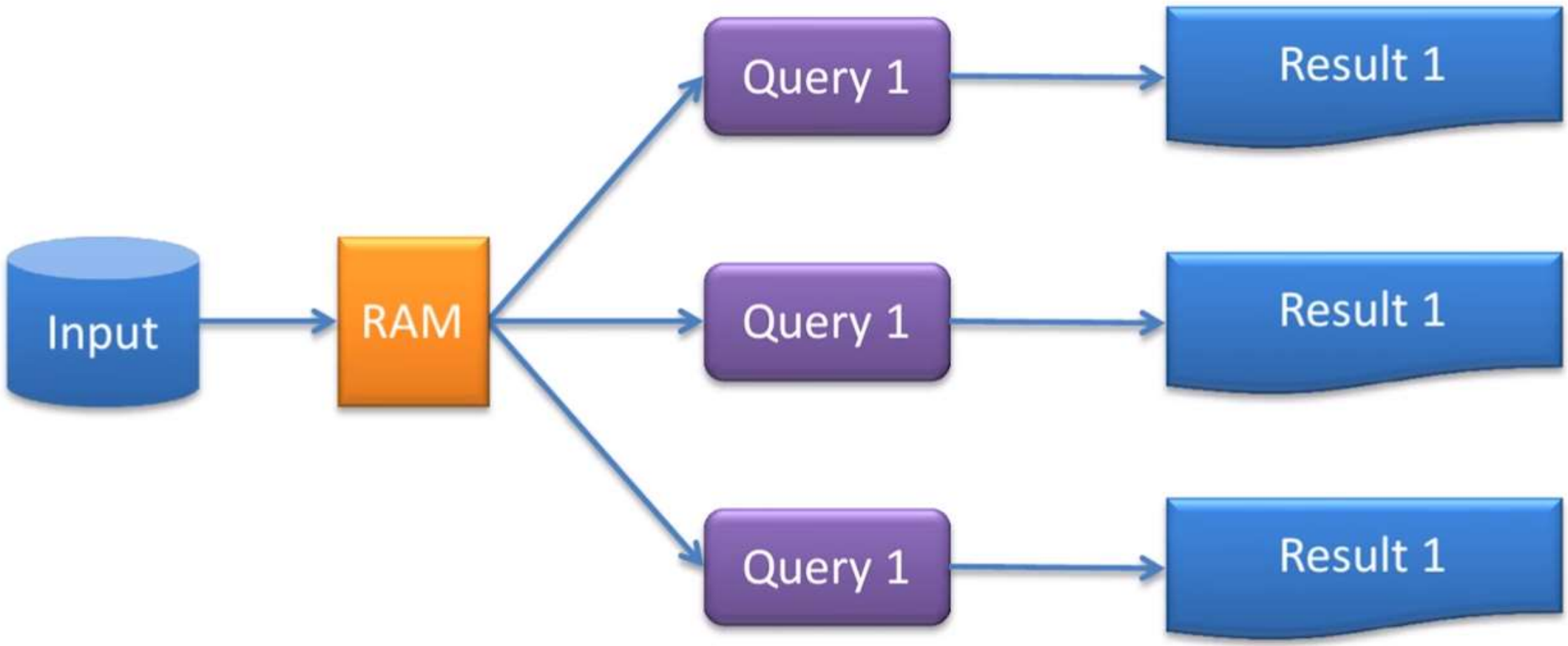
In-Memory Data Processing & Sharing



In-Memory Data Processing & Sharing



In-Memory Data Processing & Sharing



Resilient Distributed Datasets (RDD)

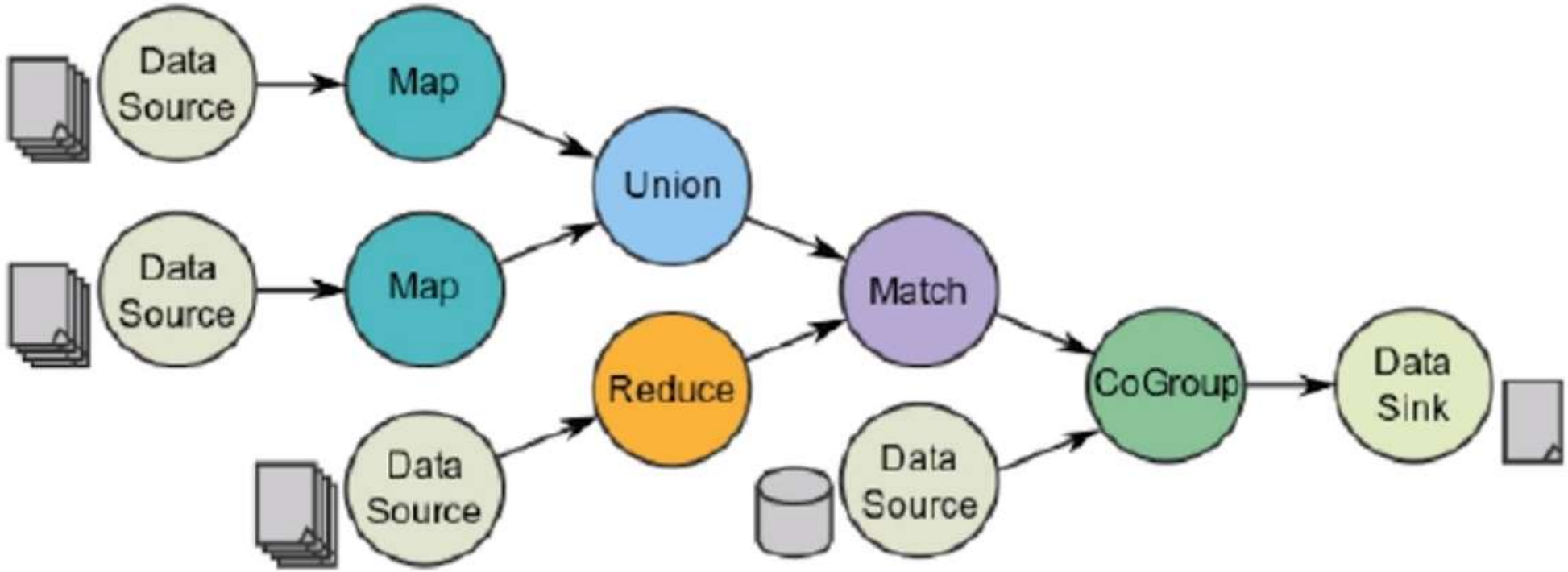
- Абстрактное представление распределенной RAM
- RDD делится на партиции, которые являются атомарными частями информации
- Партиции RDD хранятся на различных серверах
- Над RDD можно производить операции
- При этом сами RDD остаются неизменными (immutable)

Программная модель Spark

- Основана на **parallelizable operators**
- Поток обработки данных состоит из любого числа **data sources, operators** и **data sinks** путем соединения их *inputs* и *outputs*

Directed Acyclic Graph (DAG)

Любая программа может быть представлена в виде DAG



Высокоуровневые операции

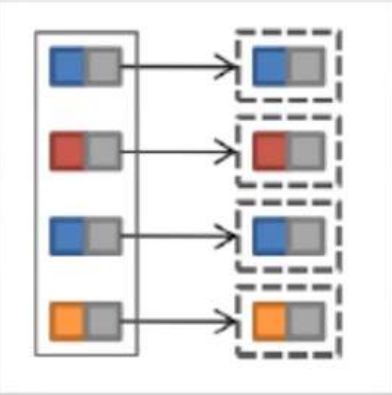
- Существует два типа RDD операторов:
  - **transformations**
  - **actions**
- **Transformations:** lazy-операторы, которые создают новые RDD
- **Actions:** запускают вычисления и возвращают результат в программу или во внешнее хранилище

Высокоуровневые операции

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

RDD transformations - Map

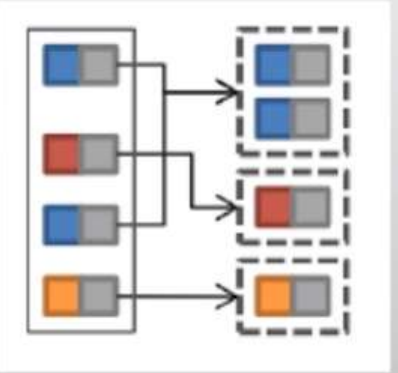
```
// passing each element through a function.
val nums = sc.parallelize(Array(1, 2, 3))
val squares = nums.map(x => x * x)  // {1, 4, 9}

// 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}
```



Все пары обрабатываются независимо

RDD transformations - Reduce

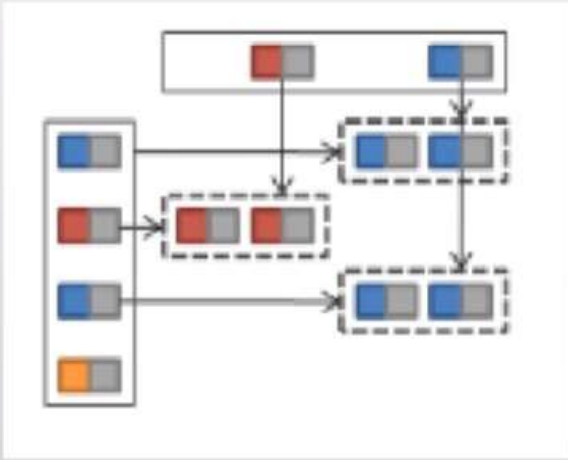
```
val pets = sc.parallelize(
    Seq(("cat", 1), ("dog", 1), ("cat", 2)))

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

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



Пары с одинаковыми ключами группируются
Группы обрабатываются независимо

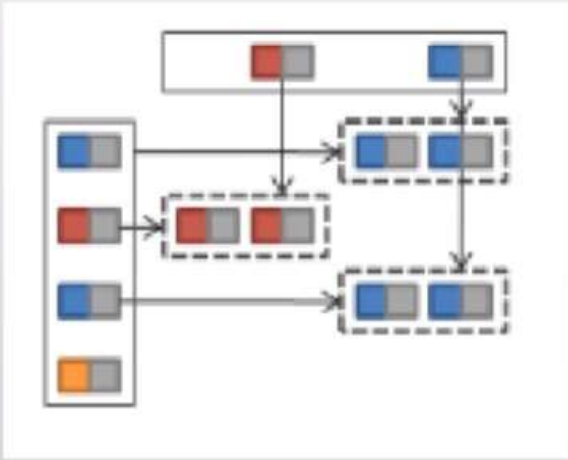
RDD transformations - JOIN

```
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"))
```



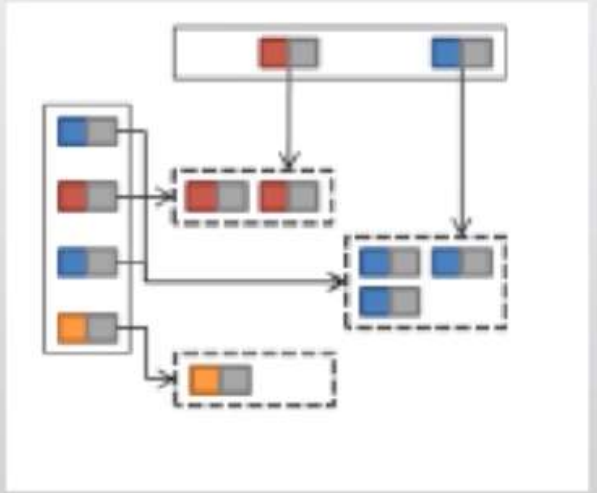
RDD transformations - JOIN

```
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

```
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")))
```

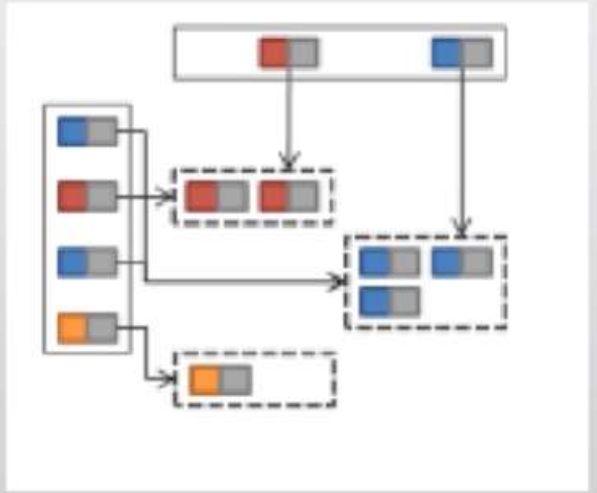


Каждый *input* группируется по ключу

Группы с одинаковыми ключами обрабатываются вместе

RDD transformations - CoGroup

```
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")))
```



Каждый *input* группируется по ключу

Группы с одинаковыми ключами обрабатываются вместе

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")
```

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 Scala

Посчитать число строк содержащих **MAIL**

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

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

Что выведет программа?

Input:
Jingle bells
Jingle bells
Jingle all the way

```
val text = sc.textFile("latin.txt")
text.flatMap(line => line.split(" ")) .map(x => (x, 1)) .count()
```

Что выведет программа?

Input:
Jingle bells
Jingle bells
Jingle all the way

```
val text = sc.textFile("latin.txt")
text.flatMap(line => line.split(" ")) .map(x => (x, 1)) .count()
```

Shared variables

Два типа общих переменных:

- Broadcast variable
- Accumulator

Shared variables - broadcast

- Read-only переменные **кешируются** на каждой машине
- 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 - accumalator

- Могут быть только **добавлены**
- Могут использоваться для реализации **счетчиков**

```
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
```

Shared variables - accumalator

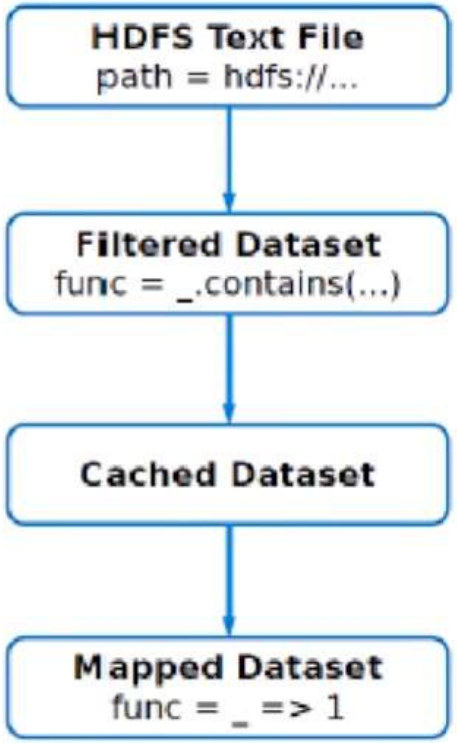
- Могут быть только **добавлены**
- Могут использоваться для реализации **счетчиков**

```
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
```

Lineage – цепочка преобразований для RDD

- **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(_+_)
```

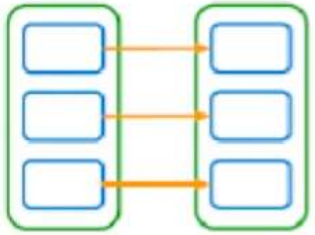


Dependencies - зависимости

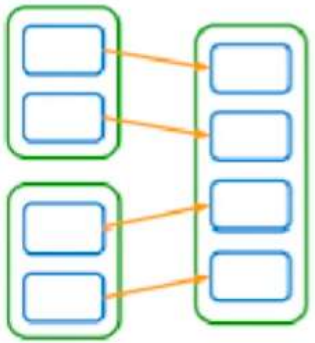
- Два типа зависимостей между RDD
  - **Narrow**
  - **Wide**

Narrow dependencies

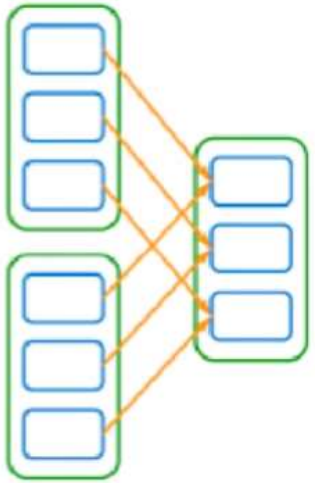
- **Narrow:** каждая партиция родительского RDD используется максимум в одной дочерней партиции RDD
- *Narrow dependencies* позволяют выполнять **pipelined execution** на одной ноде кластера:
  - Например, фильтр следуемый за Map



map, filter



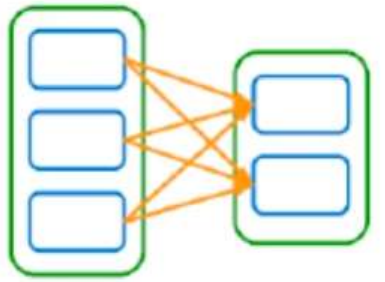
union



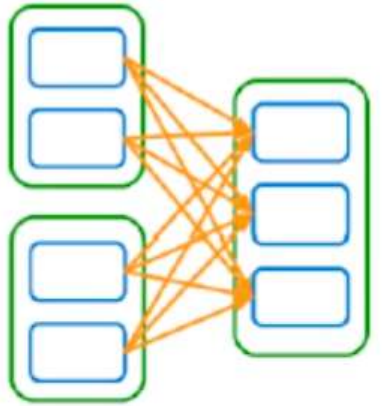
join with inputs co-partitioned

Wide dependencies

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

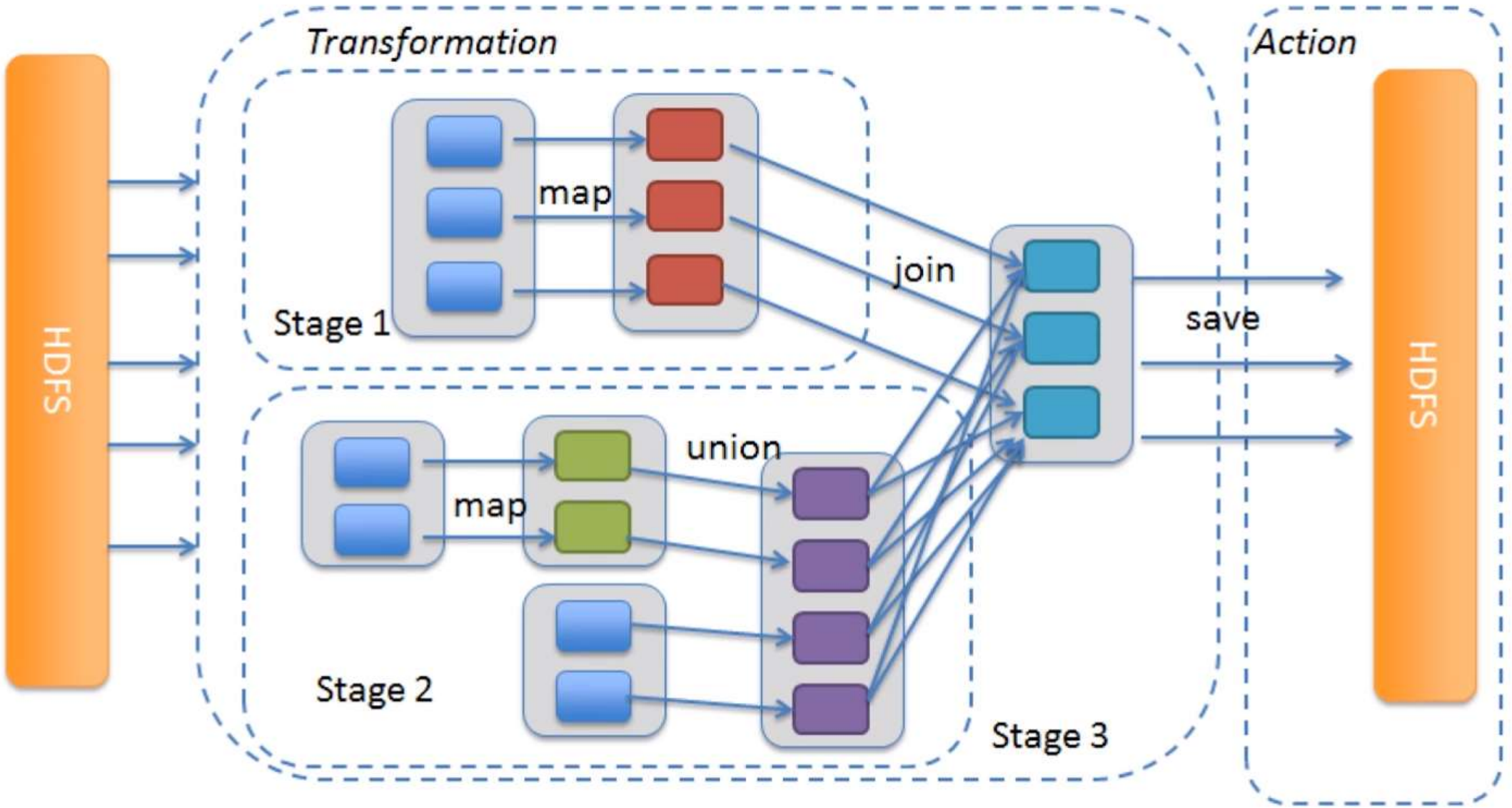


groupByKey



Join with inputs not co-partitioned

Пример lineage задачи



Управление памятью

- Если недостаточно памяти для ***новых*** партиций RDD, то будет использоваться механизм вытеснения ***LRU*** (*least recently used*)
- Spark предоставляет 3 опции для хранения *RDD*
  - В ***memory storage*** в виде ***deserialized Java objects***
  - В ***memory storage*** в виде ***serialized Java objects***
  - На ***disk storage***
- Функция *cache()* явным образом указывает, что RDD нужно хранить в памяти