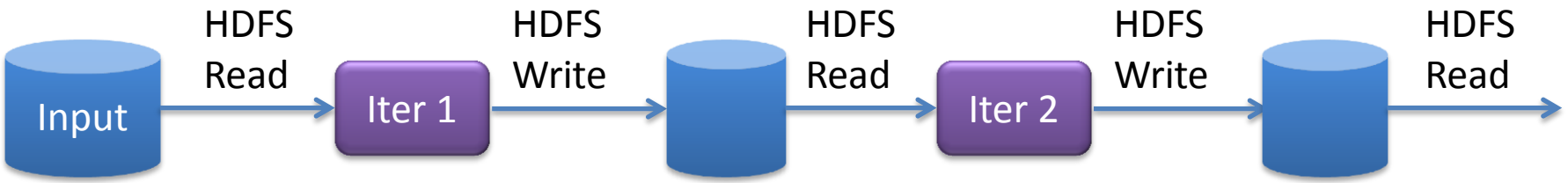

Spark

Преимущества Spark

- Следующая ступень в обработке BigData:
  - **Итеративные** задачи
  - **Интерактивная** аналитика
- Может работать с разными типами данных (текст, графы, базы данных)
- Может обрабатывать данные по частям (batch) и в потоке (streaming)
- Имеет 80 высокоуровневых функций для обработки данных (кроме map и reduce)

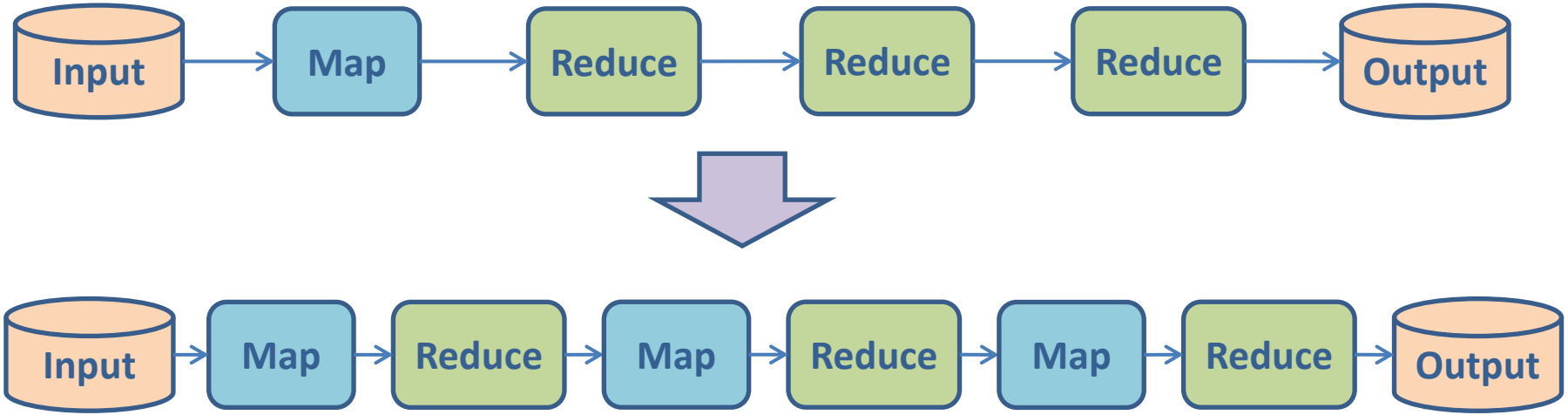
Недостатки MapReduce

Нет эффективных примитивов для общих данных

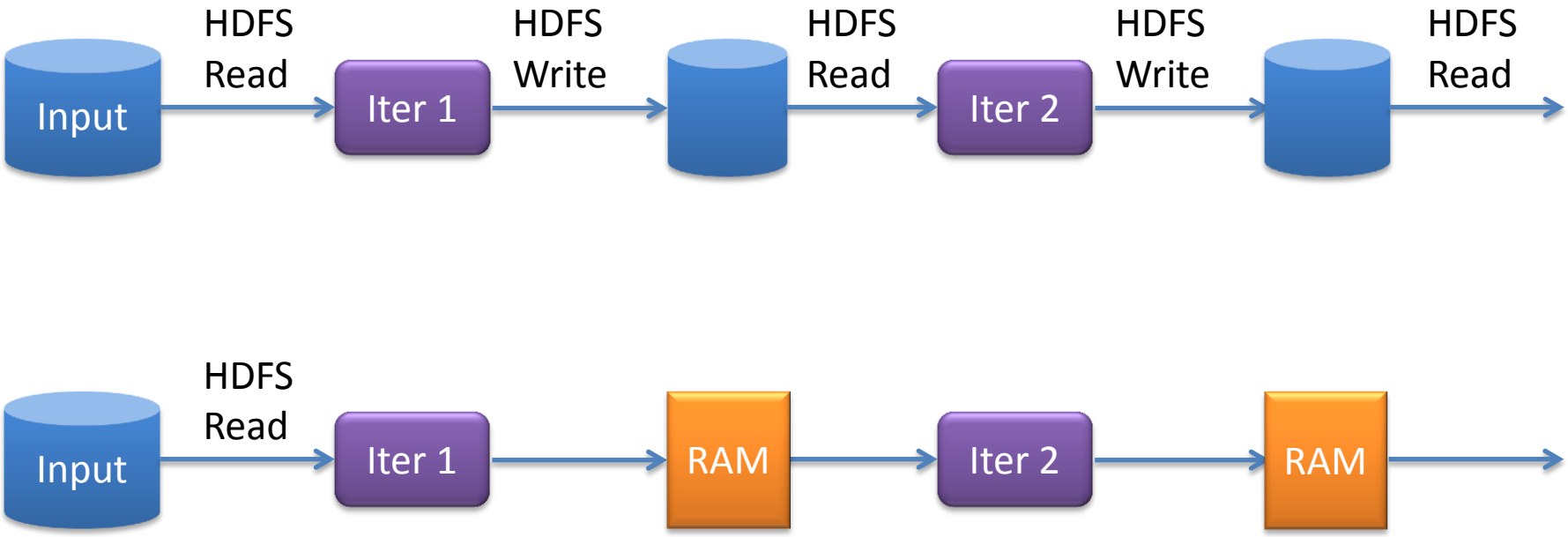


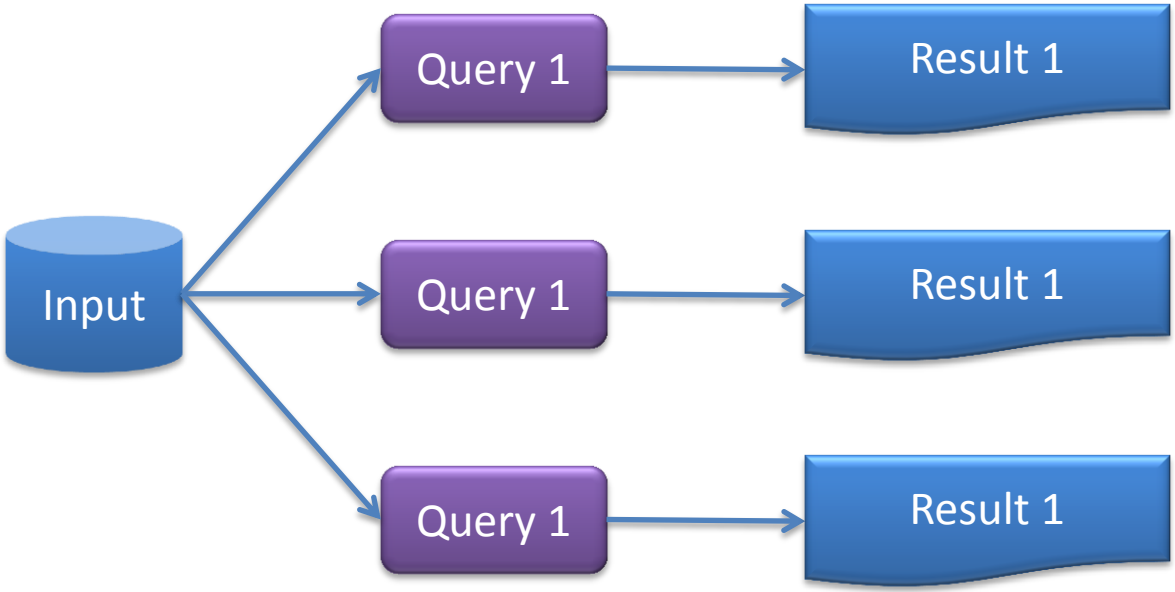
Недостатки MapReduce

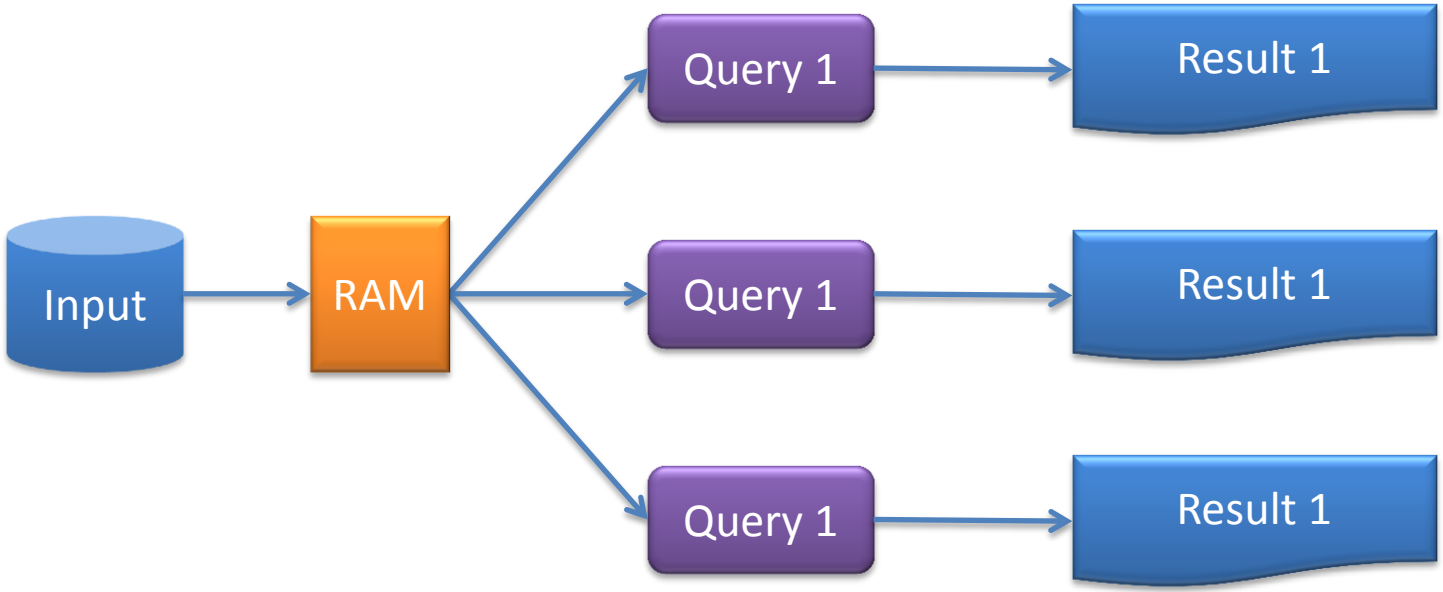
Необходимость применять шаблон MapReduce



In-Memory Data Processing and Sharing




Input
Query 1
Query 1
Query 1
Result 1
Result 1
Result 1


Input
RAM
Query 1
Query 1
Query 1
Result 1
Result 1
Result 1

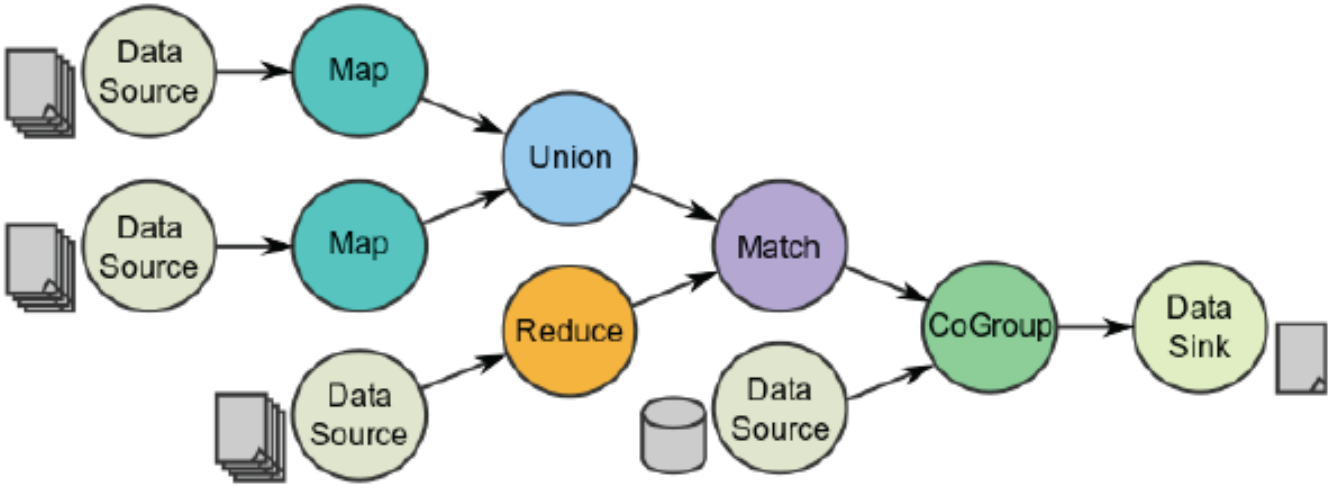
Resilient Distributed Datasets (RDD)

- 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)



Higher-Order Functions

- 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

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}
```

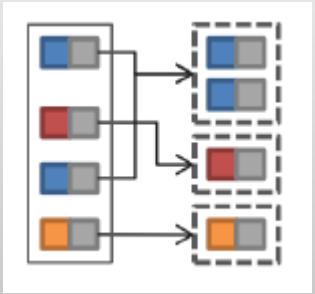
Все пары обрабатываются независимо

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

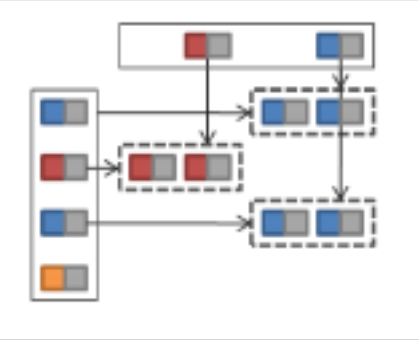
Группы обрабатываются независимо

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"))
```

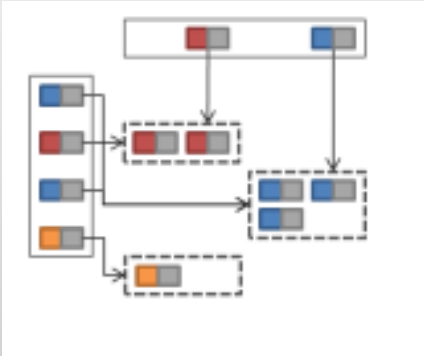


RDD Transformations - **CoGroup**

```
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 - **Union** и **Sample**

Union: объединяет два RDD, дубликаты не удаляются

Sample: выбирает произвольную часть данных (детерминировано)

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 *Java* необходимо создавать отдельно

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

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

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"))
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()
```

Shared Variables

Есть два типа *shared variables*

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

Shared Variables: **Broadcast Variables**

- 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: **Accumulators**

- 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
```