Introduction to Spark

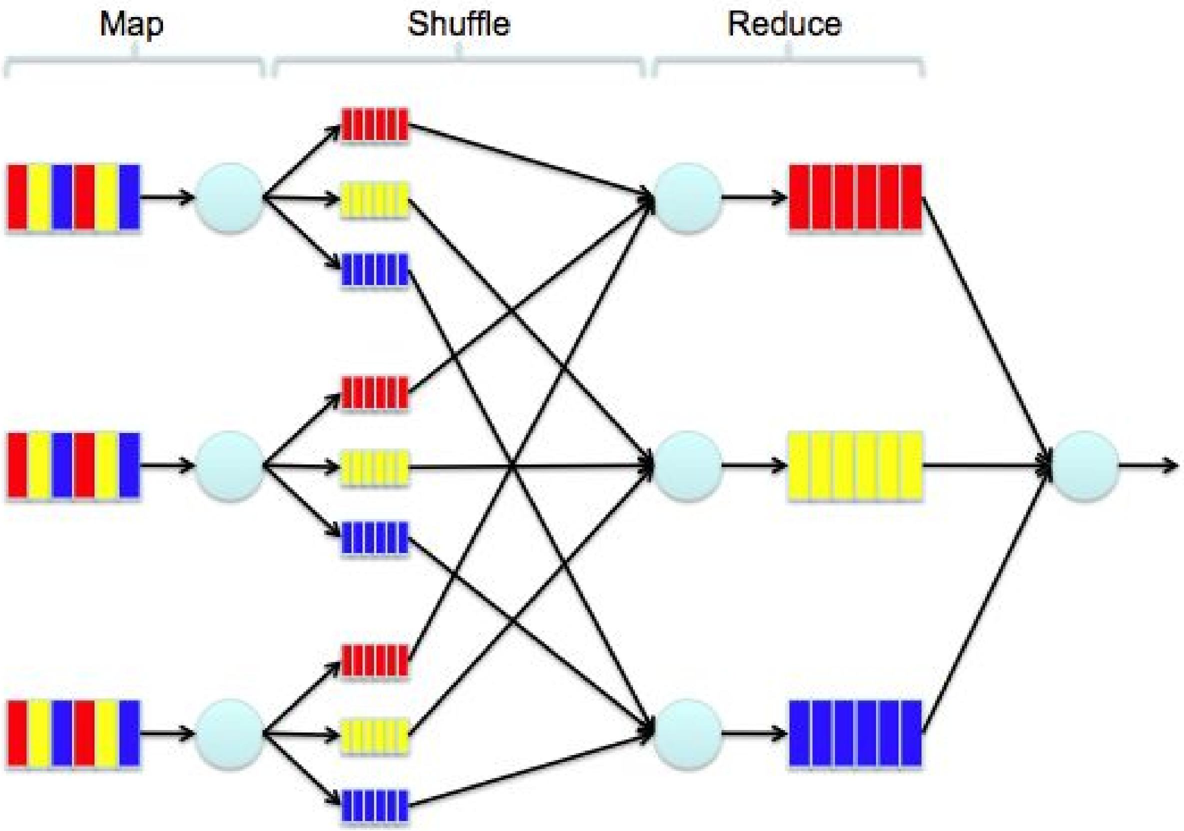
Andrey Ustyuzhanin, andrey.u@gmail.com
Maxim Borisyak, maxim.borisyak@gmail.com

Big Data

- Data is too big for one machine
- distributed computations as requirement
- MPI (?), **Hadoop, Spark**

Map Reduce

- Map
  - `Map(f)`: apply `f` to each element of collection
- Shuffle
  - rearranges data
- Reduce
  - `Reduce(g)`: convolution by associative `g(x, y)`



Hadoop Map Reduce

Немного FP

```
trait Collection[A] {
 def map[B](f: A => B): Collection[B]
 def flatMap[B](f: A => Collection[B]): Collection[B]
 def filter(f: A => Boolean): Collection[A]

 def foldLeft[B](zero: B)(f: (B, A) => B): B
 def foldRight[B](zero: B)(f: (A, B) => B): B
 def reduce[B](zero: B)(f: (A, B) => B): B
 def reduce(f: (A, A) => A): A
}
```

см. секцию A little bit of FP

Spark

- Distributed computations
- Runs on Hadoop, Mesos, …
- High computational speed:

  (0.5-1.0 x binary)

- Exploits functional principles
- Written in **Scala**, supports Java, **Python**

Word count

```
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://...")
```

Logistic regression

```
points = spark.textFile(...).map(parsePoint).cache()
w = numpy.random.ranf(size = D) # current separating plane
for i in range(ITERATIONS):
    gradient = points.map (lambda p:\
      (1 / (1 + exp( -p.y * (w.dot(p.x))) ) - 1) * p.y * p.x
    ).reduce(lambda a, b: a + b)
    w -= gradient
print "Final separating plane: %s" % w
```

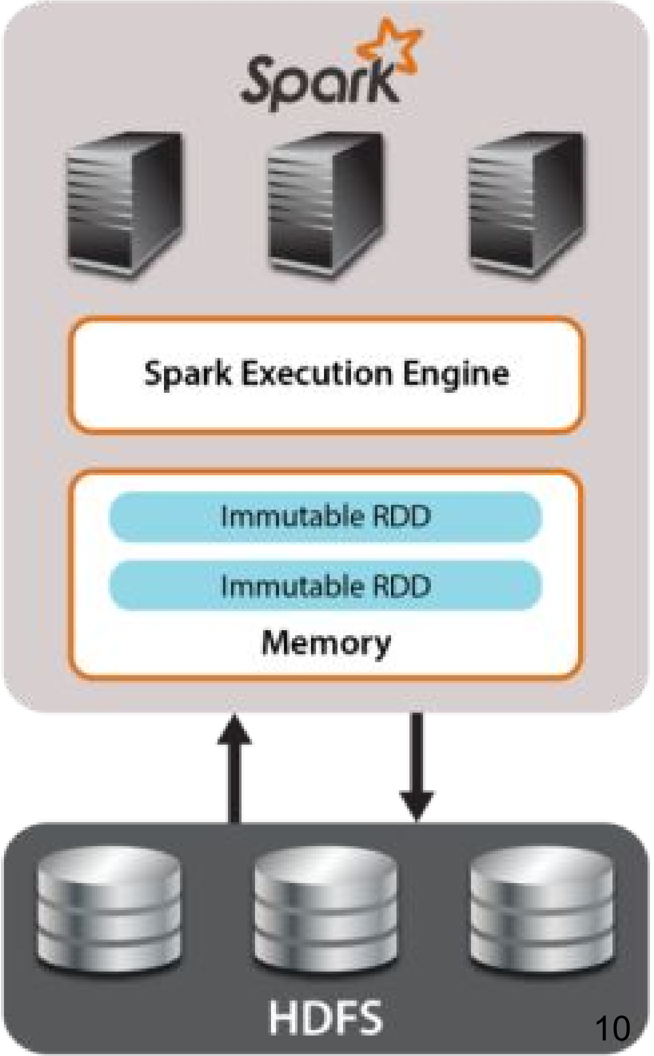
Logistic regression (scala)

```
val points = spark.textFile(...).map(parsePoint).cache()
var w = Vector.random(D) // current separating plane
for (i <- 1 to ITERATIONS) {
  val gradient = points.map { p =>
    (1 / (1 + exp(-p.y*(w dot p.x))) - 1) * p.y * p.x
  }.reduce { _ + _ }
  w -= gradient
}
println("Final separating plane: " + w)
```

Resilient Distributed Datasets

Basic Spark abstraction:

- distributed collection
- **immutable**
- fault tolerant
- can be cached



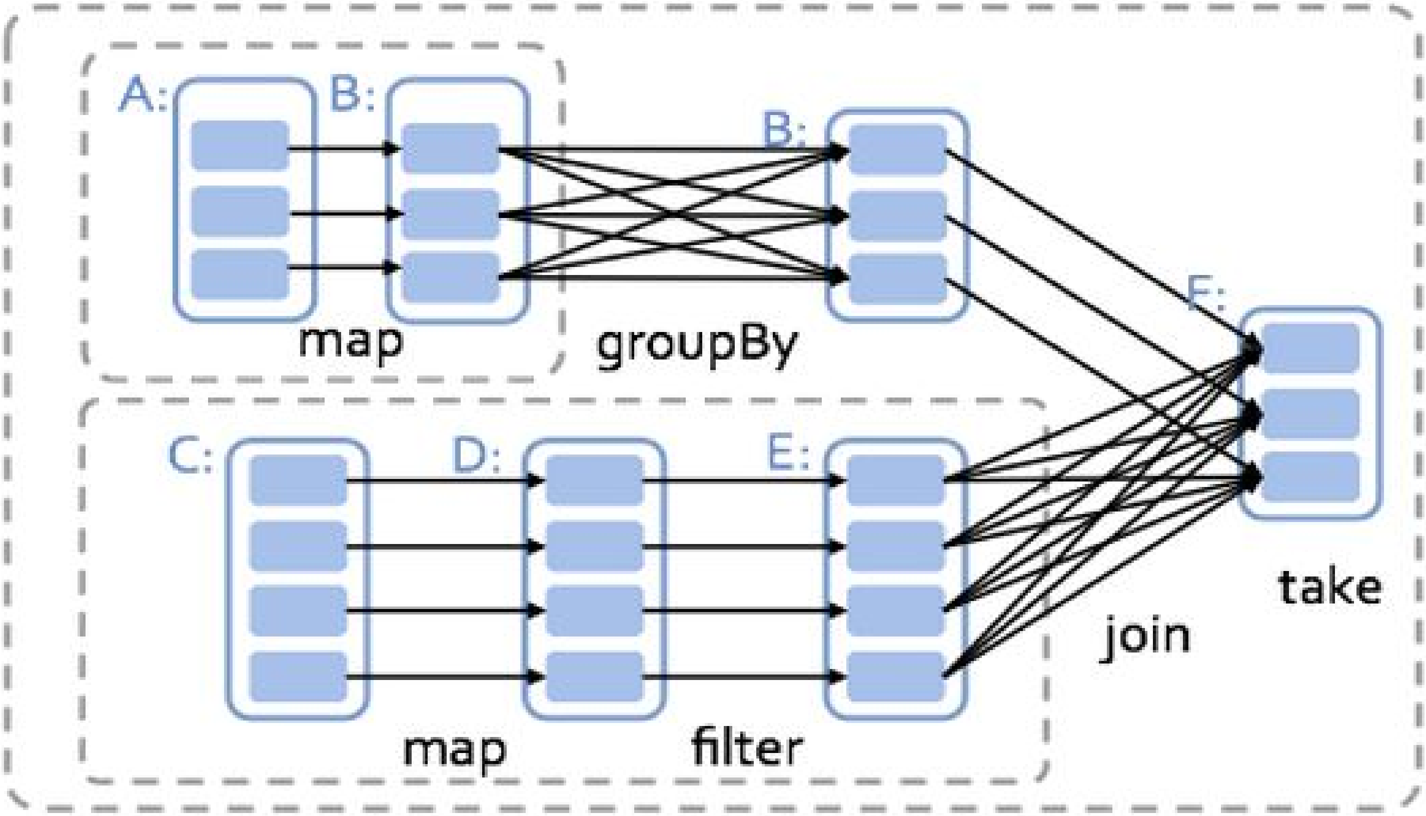
Spark computational model

- Two basic kinds of operations with RDD:
  - action: `RDD[A] => B`
  - transformation: `RDD[A] => RDD[B]`
- Transformations - lazy (!)
- Actions - toggle computation (IO action)

Spark computational model

- Driver / workers
- Driver forms **definition** of Spark program in terms of **Directed Acyclic Graph**.
- An action toggles execution of the DAG

DAG



Creation and saving

```
trait SparkContext {
 def textFile(path: String): RDD[String]
 def objectFile[T](path: String): RDD[T]
 def parallelize[T](seq: Seq[T]): RDD[T]
 def union[T](rdds: Seq[RDD[T]]): RDD[T]
}
trait RDD[T] {
 def saveAsTextFile(path: String): Unit
 def saveAsObjectFile(path: String): Unit
}
```

Creation and saving

Python:

```
text = spark.textFile("hdfs://...")

nums = spark.parallelize(xrange(0, N))
```

Scala:

```
val text: RDD[String] = spark.textFile("hdfs://...")

val count: RDD[Int] = spark.parallelize(1 to N)
```

Basic RDD operations

See full list in [Spark API](#) documentation.

```scala
trait RDD[T] {
 def map[U](f: T => U): RDD[U]
 def flatMap[U](f: T => Seq[U]): RDD[U]
 def filter(p: T => Boolean): RDD[T]

 def aggregate[U](zero: U)(f: (U, T) => U, g: (U, U) => U): U
 def fold(zero: T)(f: (T, T) => T): T
 def reduce(op: (T, T) => T): T
}
```

Basic RDD operations

```
trait RDD[T] {
 def count(): Long
 def max(): T
 def min(): T

 def sample(fraction: Double): RDD[T]
 def take(n: Int): Array[T]
 def collect(): Array[T]
}
```

Operations on PairRDD

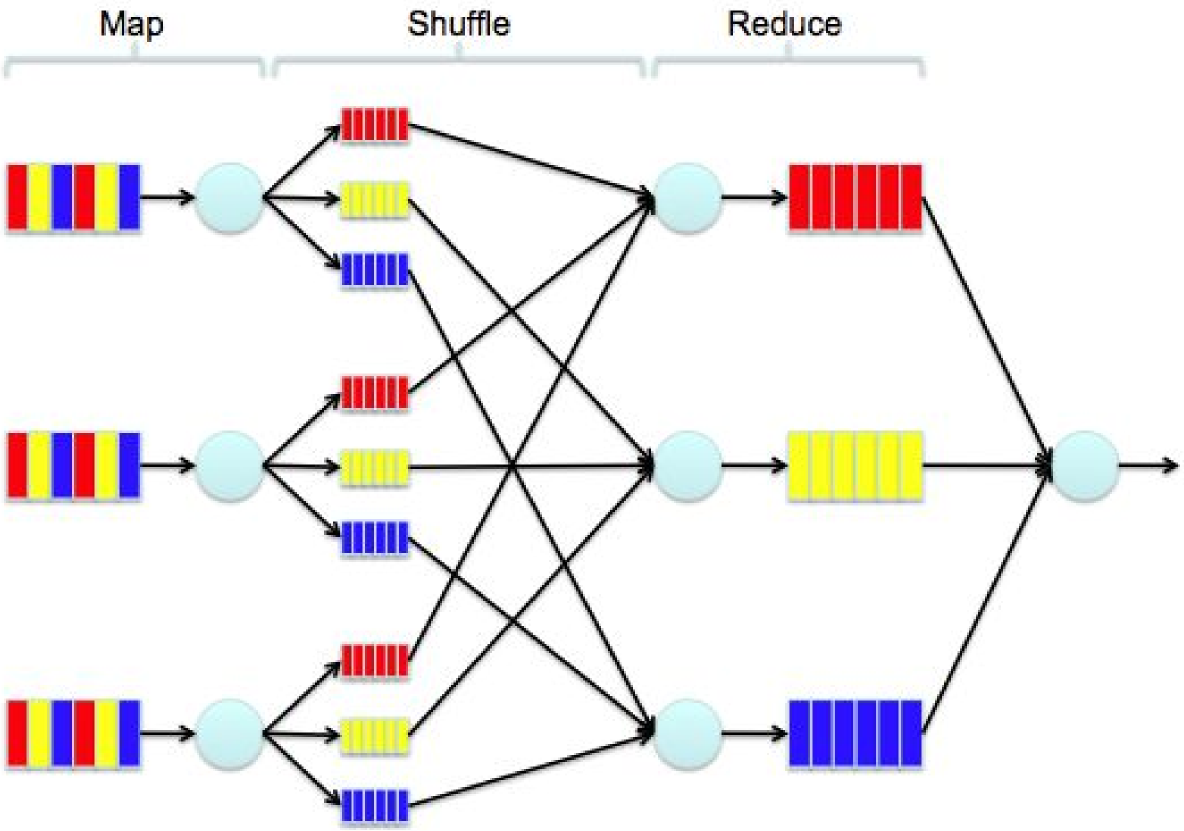
```
trait PairRDD[K, V] extends RDD[(K, V)] {
 def aggregateByKey[U](zero: U)
                          (f: (U, V) => U, g: (U, U) => U): RDD[(K, U)]

 def foldByKey(zero: V)(f: (V, V) => V): RDD[(K, V)]
 def reduceByKey(op: (V, V) => V): RDD[(K, V)]
 def groupByKey(): RDD[(K, Iterable[V])]
}
```

Operations on PairRDD

```scala
trait PairRDD[K, V] extends RDD[(K, V)] {
 def cogroup[U](other: RDD[(K, U)]): RDD[(K, (Iterable[V], Iterable[U]))]

 def fullOuterJoin[U](other: RDD[(K, U)]): RDD[(K, (Option[V], Option[U]))]

 def leftOuterJoin[U](other: RDD[(K, U)]): RDD[(K, (V, Option[U]))]

 def join[U](other: RDD[(K, U)]): RDD[(K, (V, U))]
}
```

Map Reduce (again)

- Map
  - `Map(f)`: apply `f` to each element of collection
  - `RDD[(K, V)]` as the result
- **Shuffle**
  - rearranges data to place **all** entries with the same key are on one node
- Reduce
  - `Reduce(g)`: convolution by associative `g(x, y)`
  - Separately for each key



Hadoop Map Reduce

Latency Comparison Numbers

L1 cache reference	0.5	ns			
Branch mispredict	5	ns			
L2 cache reference	7	ns			14x L1 cache
Mutex lock/unlock	25	ns			
Main memory reference	100	ns			20x L2, 200x L1
Compress 1K bytes with Zippy	3,000	ns			
Send 1K bytes over 1 Gbps network	10,000	ns	0.01	ms	
Read 4K randomly from SSD*	150,000	ns	0.15	ms	
Read 1 MB sequentially from memory	250,000	ns	0.25	ms	
Round trip within same datacenter	500,000	ns	0.5	ms	
Read 1 MB sequentially from SSD*	1,000,000	ns	1	ms	4X memory
Disk seek	10,000,000	ns	10	ms	
Read 1 MB sequentially from disk	20,000,000	ns	20	ms	80x memory, 20X SSD
Send packet CA->Netherlands->CA	150,000,000	ns	150	ms	

Shuffle

- Shuffle is very expensive!
- Data locality => max
- Avoid keys with possibly great number of values

What is wrong in the code?

```
val ys = xs.map(f1)
val zs = ys.map(f2)

println {
 zs.count()
}

val ws = xs.map(f3)
```

Cache

```
val ys = xs.map(f1).cache()
val zs = ys.map(f2)

println {
 zs.count()
}

val ws = xs.map(f3)
```