

L3: APACHE SPARK ВВЕДЕНИЕ

Подготовил: Алексей Попов БН BigData Solution

Читает: Андрей Журлов

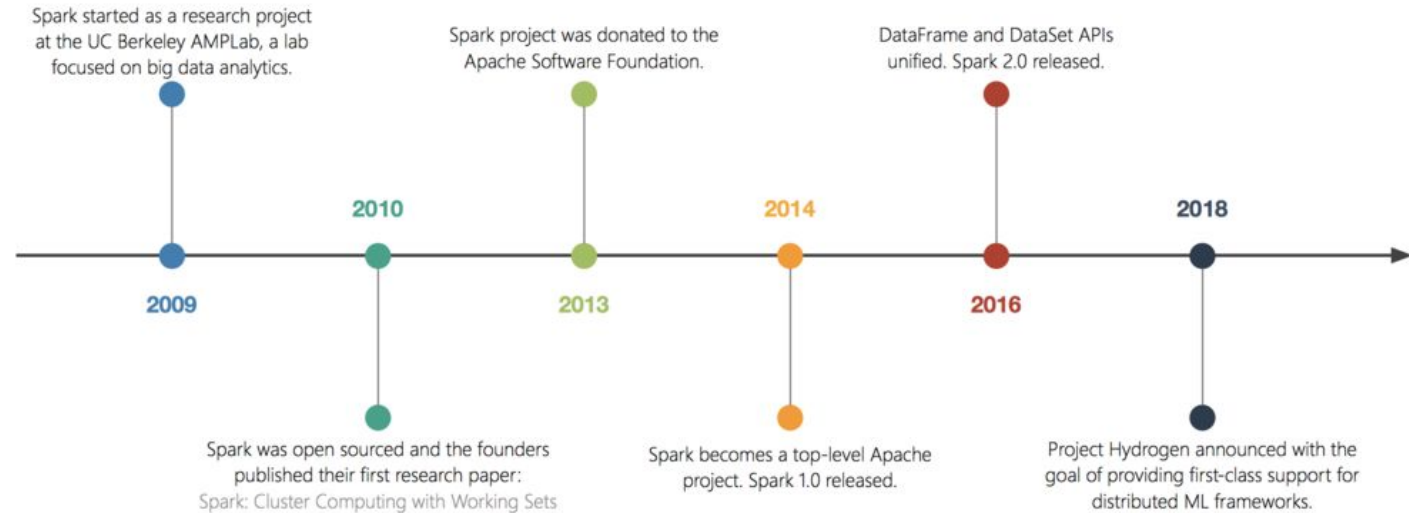
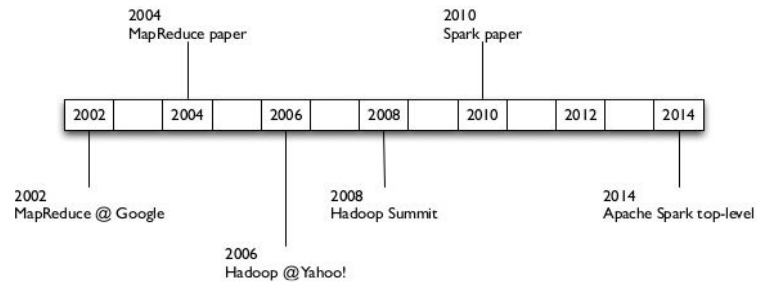
План презентации

- Apache Spark обзор
- Как работает Spark
- RDD
- Трансформация и действие
- Структура задания Spark

Краткая история Spark



Spark: A Brief History



Что такое Apache Spark

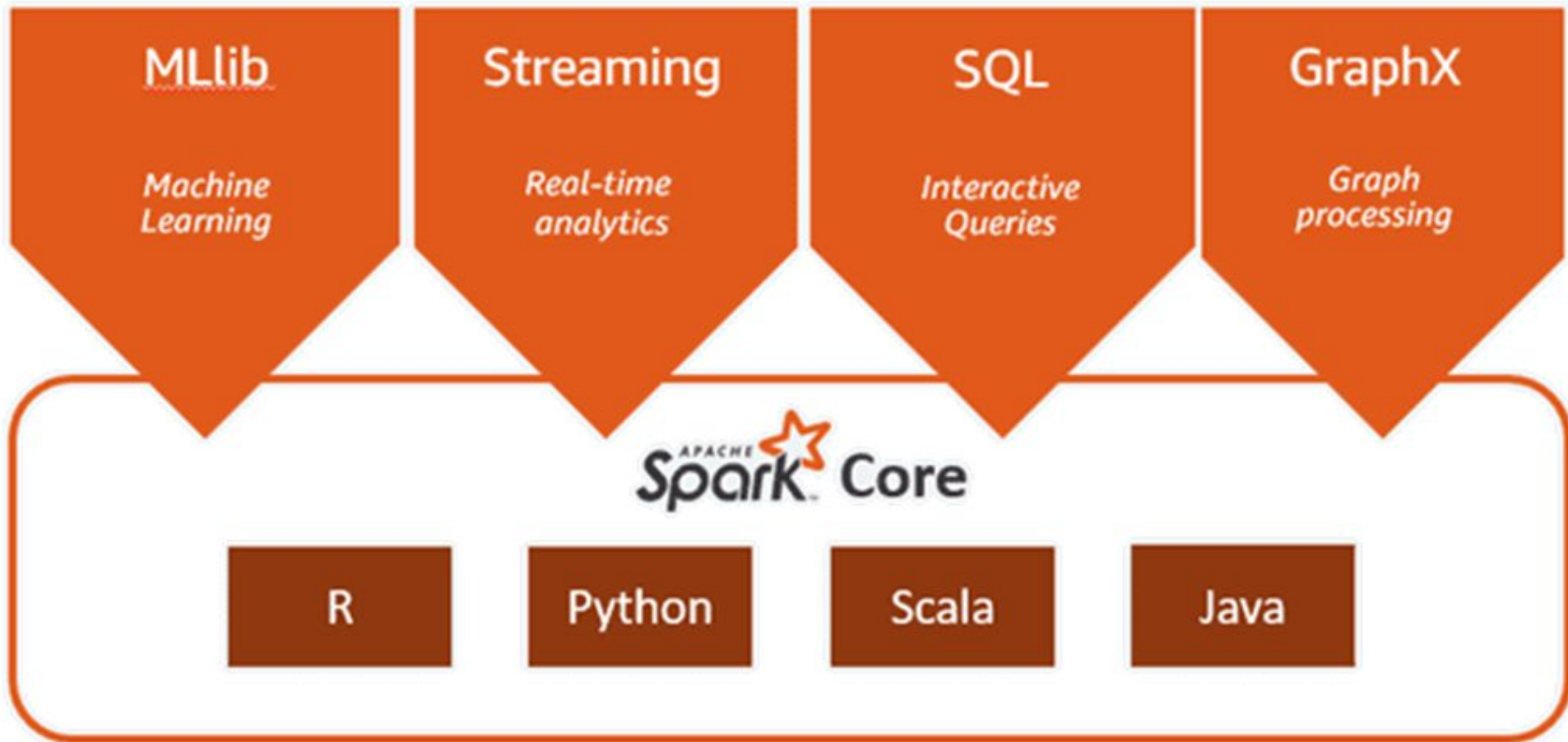
Apache Spark – это BigData **фреймворк с открытым исходным кодом** для распределённой **пакетной и потоковой** обработки неструктурированных и слабоструктурированных данных, входящий в экосистему проектов Hadoop.

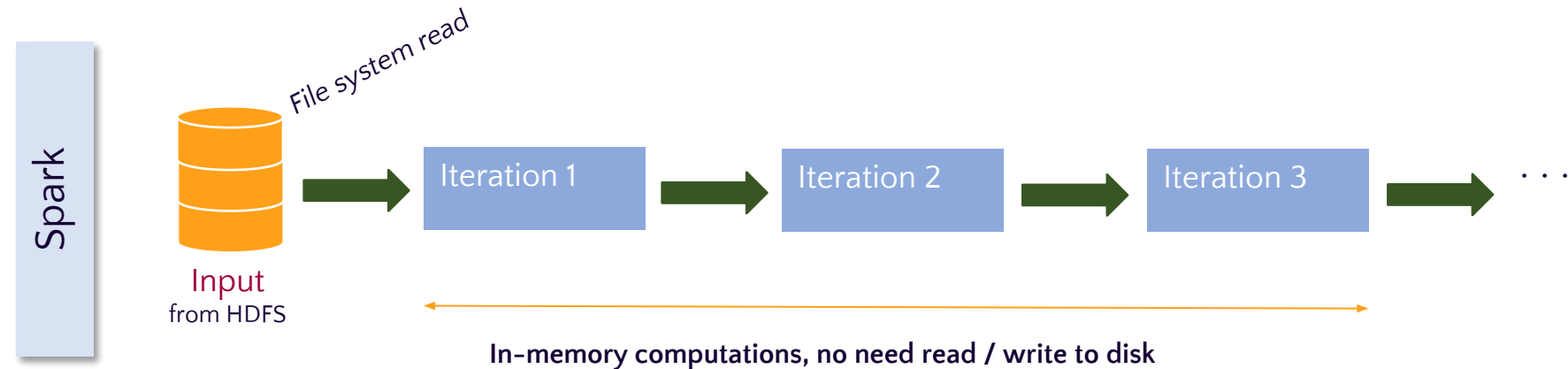
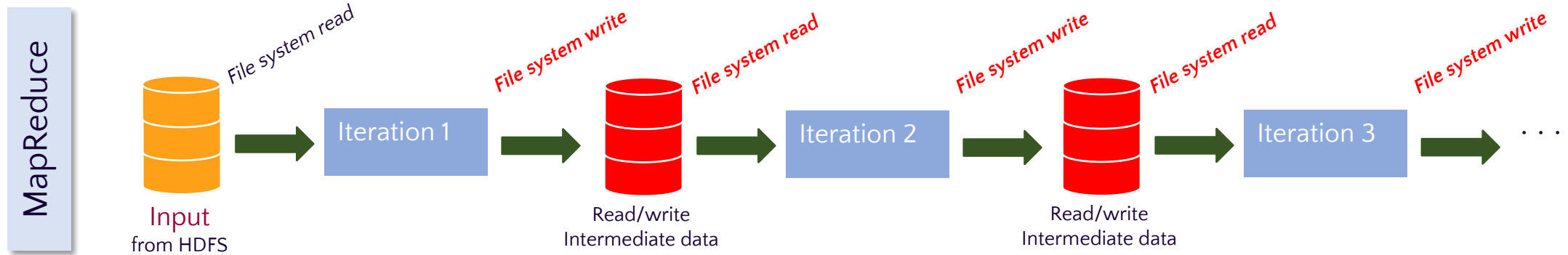
Основным автором Apache Spark считается Матей Захария (Matei Zaharia), румынско-канадский учёный в области информатики. Он начал работу над проектом в 2009 году, будучи аспирантом Университета Калифорнии в Беркли. В 2010 году проект опубликован под лицензией BSD, в 2013 году передан фонду Apache Software Foundation и переведён на лицензию Apache 2.0, а в 2014 году принят в число проектов верхнего уровня Apache. **Изначально Спарк написан на Scala.**

Преимущества и особенности Apache Spark

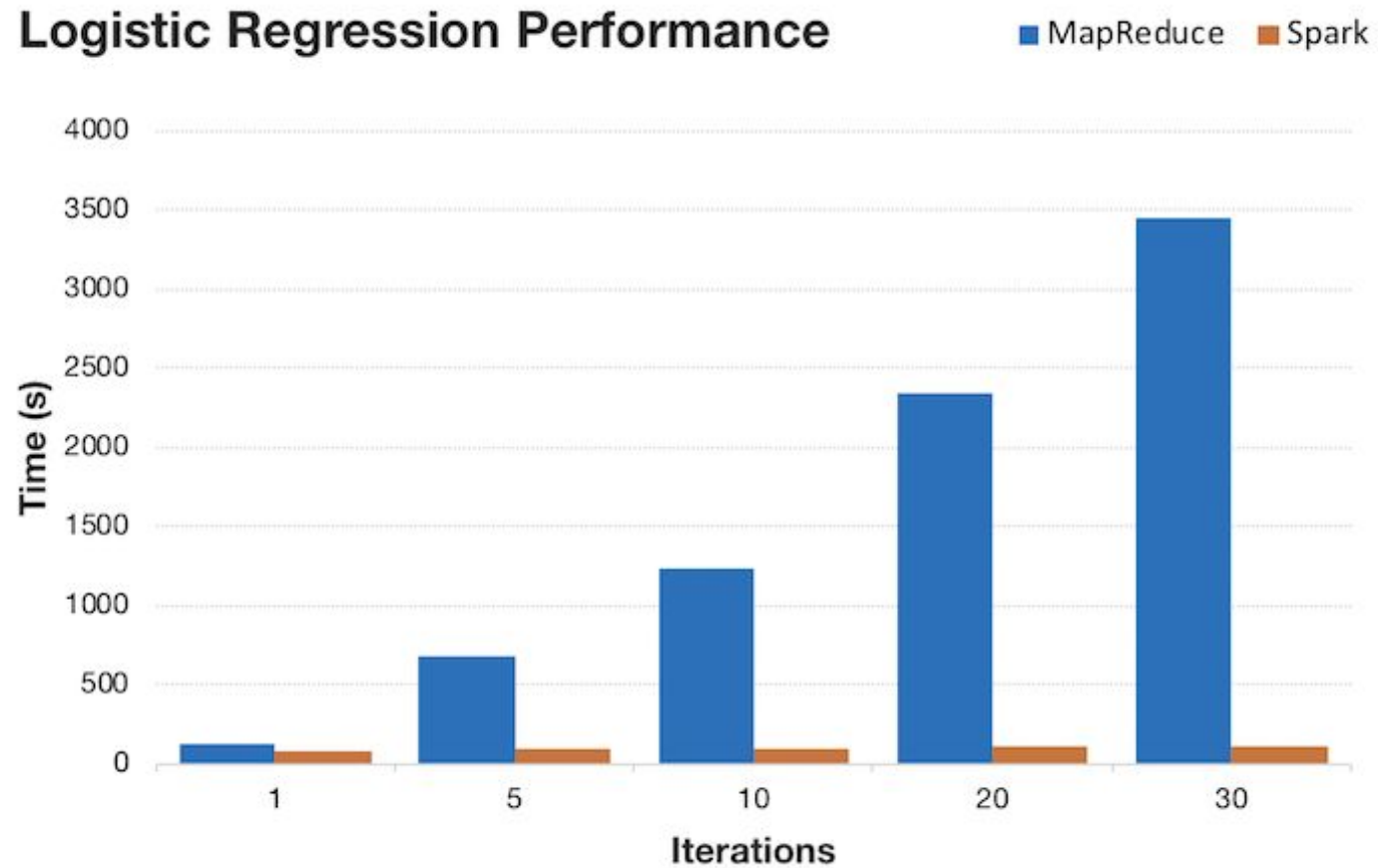
- **Spark — всё-в-одном для работы с большими данными** - Spark создан для того, чтобы помогать решать широкий круг задач по анализу данных, начиная с простой загрузки данных и SQL-запросов и заканчивая машинным обучением и потоковыми вычислениями, при помощи одного и того же вычислительного инструмента с неизменным набором API.
- **Spark оптимизирует своё машинное ядро для эффективных вычислений** — то есть Spark только управляет загрузкой данных из систем хранения и производит вычисления над ними, но сам не является конечным постоянным хранилищем.
- **Библиотеки Spark дарят очень широкую функциональность** — сегодня стандартные библиотеки Spark являются главной частью этого проекта с открытым кодом. Ядро Spark само по себе не слишком сильно изменялось с тех пор, как было выпущено, а вот библиотеки росли, чтобы добавлять ещё больше функциональности. И так Spark превратился в multifunctional инструмент анализа данных. В Spark есть библиотеки для SQL и структурированных данных (Spark SQL), машинного обучения (MLlib), потоковой обработки (Spark Streaming) и аналитики графов (GraphX).
- **Поддержка нескольких языков разработки** - Scala, Java, Python и R

Преимущества и особенности Apache Spark





MapReduce и Spark



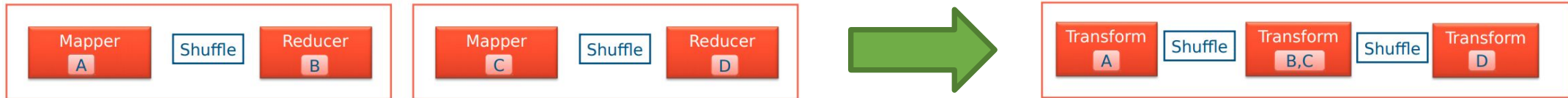
- Преимущество Spark особенно проявляется если необходимо выполнить цепочку задач или итераций

MapReduce и Spark

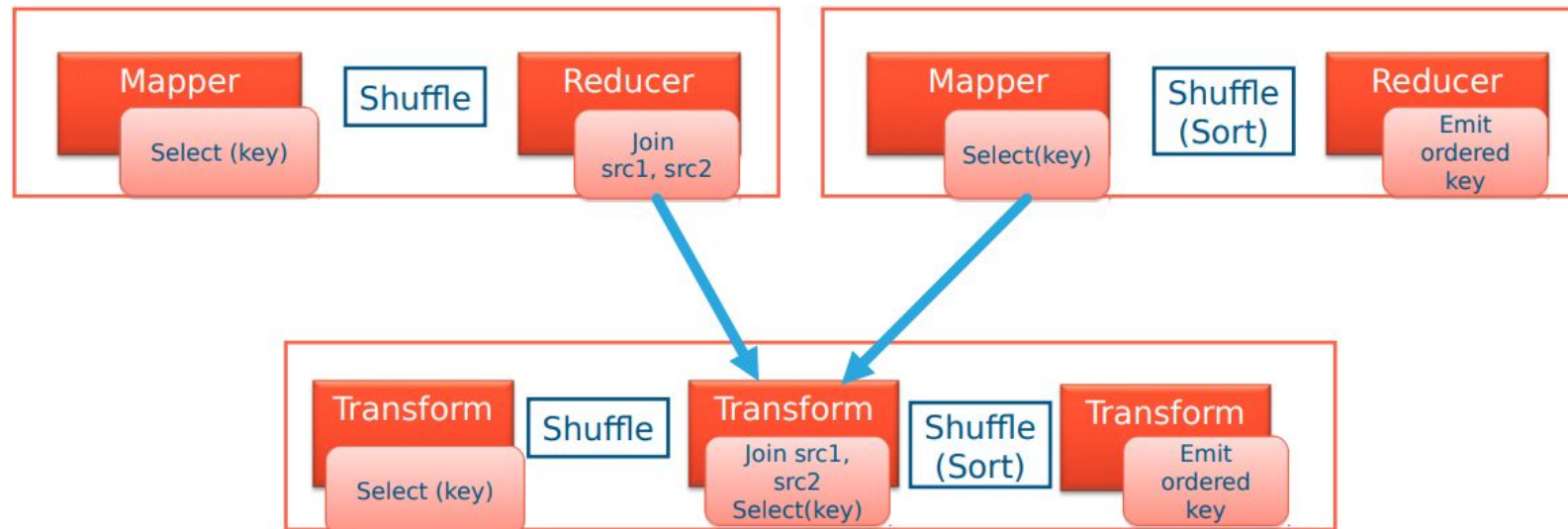
	MapReduce	Spark
Данные	Файл	RDD сохраняемые в памяти узлов
Программа	Map, Shuffle, Reduce в заданном порядке	Трансформации в любом заданном порядке, нет деления на виды.
Жизненный цикл	Задача - Java процессы, которые запускаются и выгружаются для каждого шага	Задача – выполняется на доступных, долгоживущих процессах Executors

MapReduce и Spark

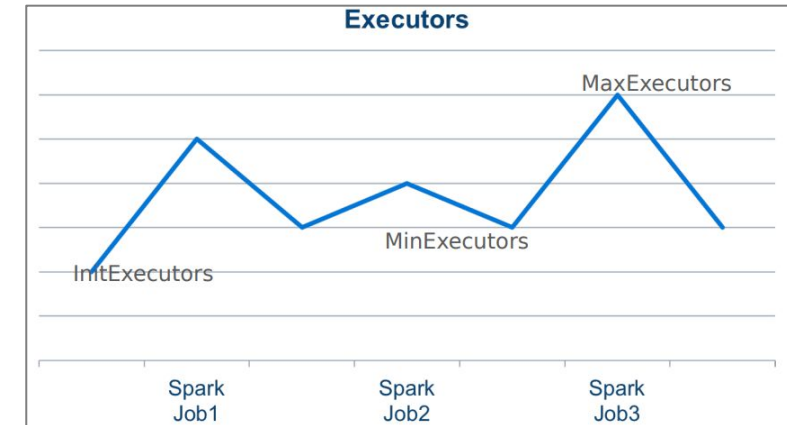
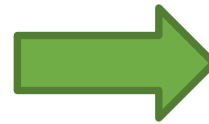
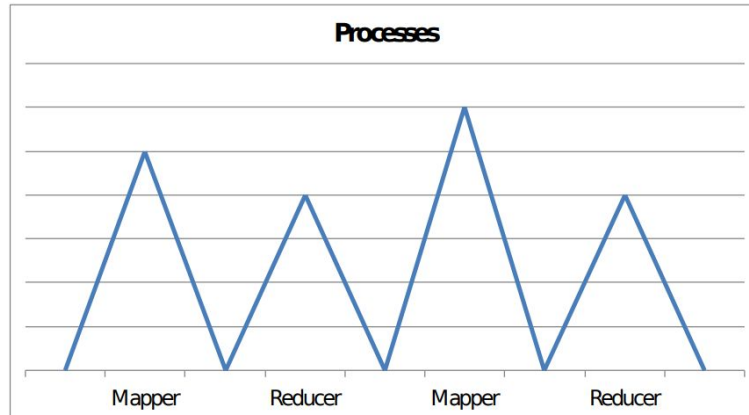
Меньше шагов – Spark job это набор трансформаций (без разделения Mapper – Reducer) разделенных Shuffle.



```
SELECT src1.key FROM
(SELECT key FROM src1 JOIN src2 ON src1.key = src2.key)
ORDER BY src1.key;
```



Жизненный цикл процессов



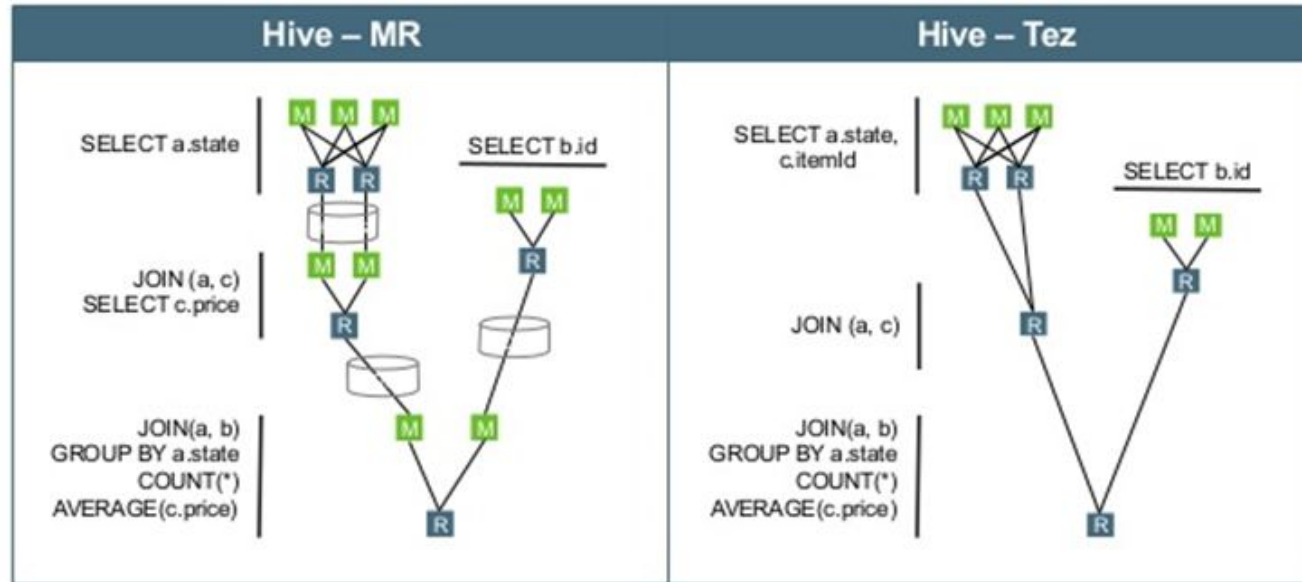
- **MapReduce** – каждый шаг запускает и удаляет процессы Mapper и Reducer
- **Spark** – каждый Executor (исполнитель) является долгоживущим процессом и может в течение жизни исполнять одну или несколько задач последовательно и параллельно (executor cores)

Развитие MapReduce – Tez

Hive-on-MR vs. Hive-on-Tez

```
SELECT a.x, AVERAGE(b.y) AS avg
FROM a JOIN b ON (a.id = b.id) GROUP BY a
UNION SELECT x, AVERAGE(y) AS AVG
FROM c GROUP BY x
ORDER BY AVG;
```

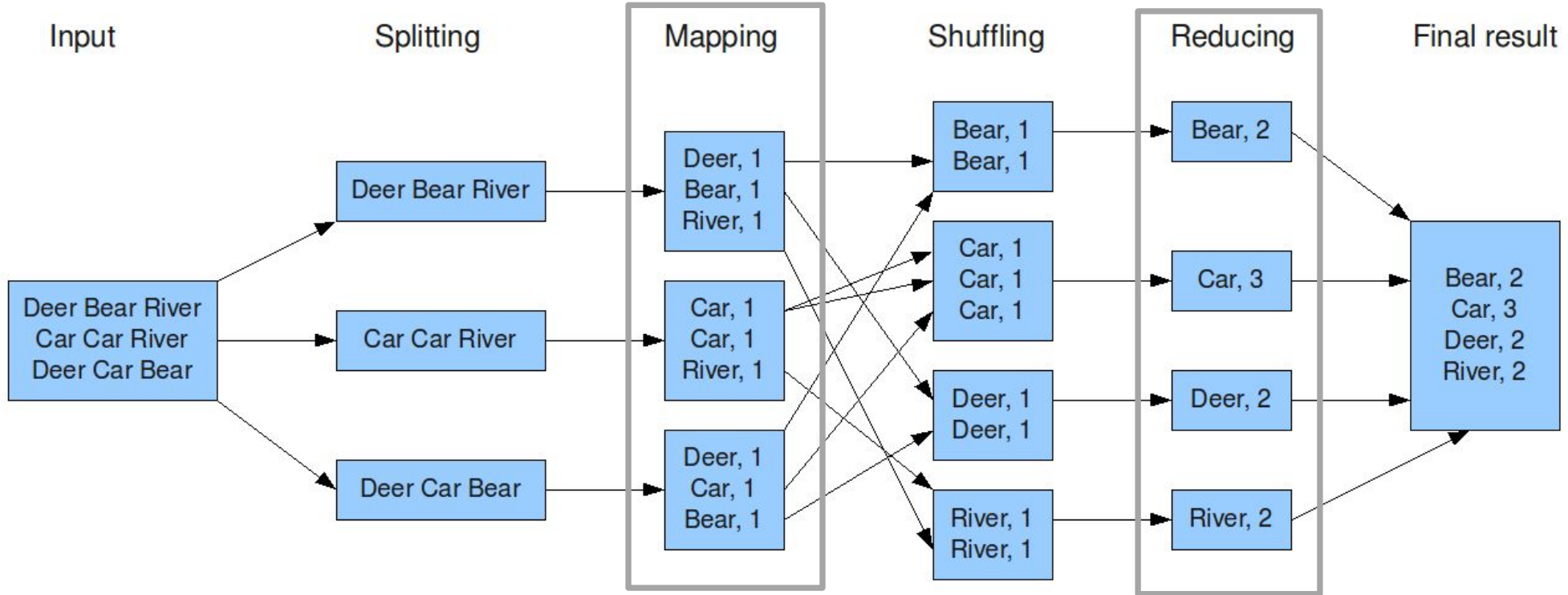
Tez avoids
unneded writes to
HDFS



- **Tez** – позволяет запустить цепочку MR на выполнение без промежуточной записи в HDFS

MapReduce: word count

The overall MapReduce word count process



Необходимо написать Mapper и Reducer все остальное обеспечивает MapReduce фреймворк

MapReduce Java

```
// Word Count Example - MapReduce (Java)
import java.io.IOException;
import java.util.StringTokenizer;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class WordCount {

    public static class TokenizerMapper
        extends Mapper<
            Text, Text, Text, Text> {

        private final static IntWritable one = new IntWritable(1);
        private Text word = new Text();

        public void map(Object key, Text value, Context context
            ) throws IOException, InterruptedException {
            StringTokenizer itr = new StringTokenizer(value.toString());
            while (itr.hasMoreTokens()) {
                word.set(itr.nextToken());
                context.write(word, one);
            }
        }
    }

    public static class IntSumReducer
        extends Reducer<
            Text, Text, Text, Text> {

        private IntWritable result = new IntWritable();

        public void reduce(Text key, Iterable values,
            Context context
            ) throws IOException, InterruptedException {

            int sum = 0;
            for (IntWritable val : values) {
                sum += val.get();
            }
            result.set(sum);
            context.write(key, result);
        }
    }

    public static void main(String[] args) throws Exception {
        Configuration conf = new Configuration();
        Job job = Job.getInstance(conf, "word count");
        job.setJarByClass(WordCount.class);
        job.setMapperClass(TokenizerMapper.class);
        job.setReducerClass(IntSumReducer.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);
        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));
        System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
}
```



Word Count для Spark на Scala

```
val sc = new SparkContext(new SparkConf().setAppName("Spark Count"))
val textFile = sc.textFile("hdfs://...")
val counts = textFile.flatMap(line => line.split(" "))
                    .map(word => (word, 1))
                    .reduceByKey(_ + _)
counts.saveAsTextFile("hdfs://...")
```



Особенности Spark

Устойчивость к сбоям

Для каждого набора данных Spark ведет Lineage и может пересчитать данные с любого момента при сбое/потере узла

Lazy Evaluation (ленивое исполнение)

Реальная работа начинается только тогда, когда требуются данные (сохранение файл, count, collect, ...)

Потоковая обработка в реальном времени

Возможность как Batch так и Streaming обработки данных



Скорость работы

Spark – eng. «ИСКРА»
Различные архитектурные решения для увеличения скорости (кеширование, долгоживущие executors, ...)

Универсальность

Универсальный фреймворк для разработки широкого спектра задач: batch, streaming, ML, GraphX, SparkSQL. Возможность разработки своих модулей

Поддержка нескольких языков

Scala, Java, Python, R

Основные концепции Spark

Работаем с коллекцией
как с единым целым

Id	Name
1	A
2	B
3	C
4	D
5	E
6	F
7	G
8	H
9	K

На самом деле внутри
это набор партиций...

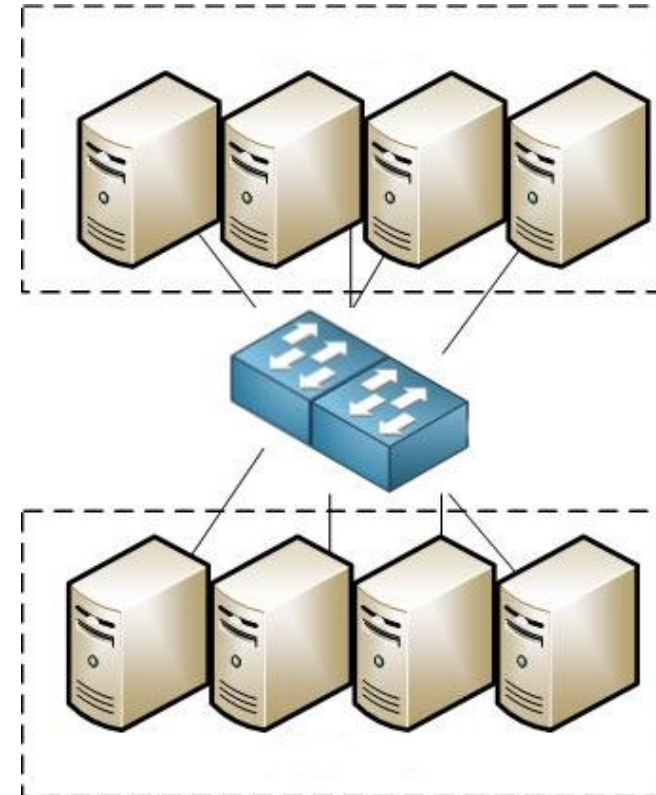
Id	Name
3	C
6	F
1	A
8	H
9	K

Id	Name
2	B

Id	Name
4	D
5	E

Id	Name
7	G

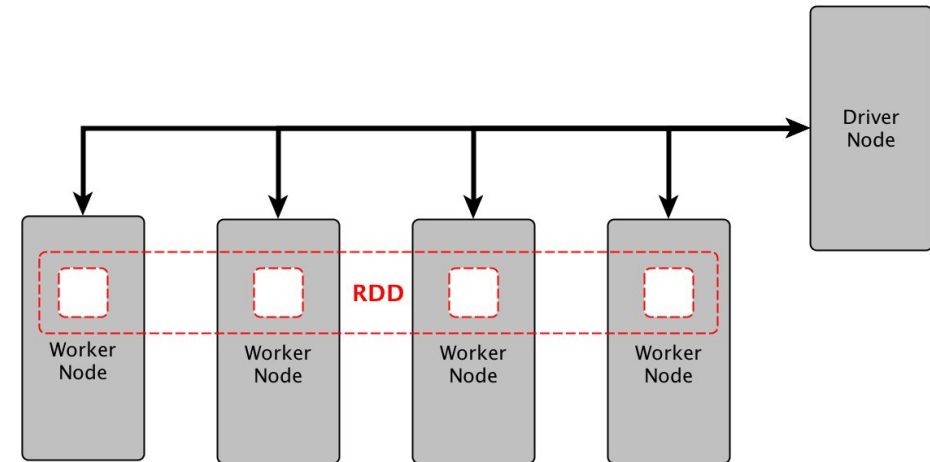
... распределенных на рабочих узлах
(в памяти, в кеше, на диске, может и не существовать физически)



```
val textFile = sc.textFile("hdfs://...")
```

RDD - Resilient Distributed Dataset:

- **Неизменяемая** распределенная коллекция (таблица)
- **Отказоустойчивая** - для RDD ведется **Lineage** – Spark всегда знает как восстановить RDD в случае сбоя
- Внутри RDD разбита на **партиции** — это минимальный объем RDD, который будет обработан каждым рабочим узлом.
- RDD распределена по узлам Executors



Трансформация и действие

```
val textFile = sc.textFile("hdfs://...")
val splits = textFile.flatMap(line => line.split(" "))

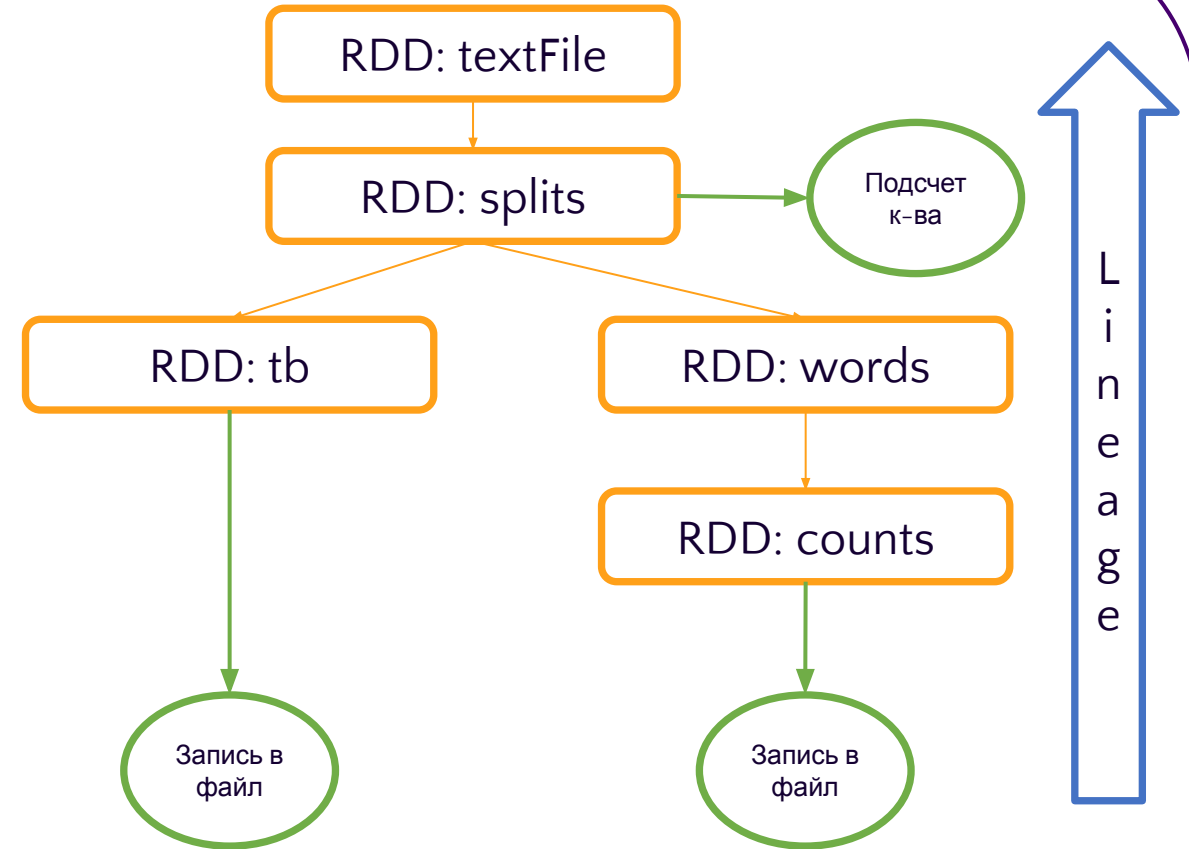
val allwords = splits.count()

val tb = splits.filter(_.startsWith("b"))
tb.saveAsTextFile("hdfs://...")

val words = splits.map(word => (word, 1))
val counts = words.reduceByKey(_ + _)
counts.saveAsTextFile("hdfs://...")
```

Трансформация
не приводит к запуску вычислений

Действие
запускает цепочку вычислений



Lazy Evaluation

Трансформация и действие

Трансформация

не приводит к запуску вычислений

Примеры:

- `map(func)`
- `filter(func)`
- `union(otherDataset)`
- `reduceByKey(func)`
- `join(otherDataset)`

Действие

запускает цепочку вычислений

Примеры:

- `collect()`
- `count()`
- `take(n)`
- `saveAsTextFile(path)`



Плюсы и минусы Lazy Evaluation

Удобство написания программ

Улучшает читаемость кода, можно разбивать на небольшие куски, потом все соберется в единый DAG.

Избежание ненужных вычислений и трафика между драйвером и узлами

Обрабатываются только те данные, которые реально нужны. (take(10))
Строится единый план выполнения.

Оптимизация

Построенный план запроса оптимизируется Spark, сдвигая например некоторые фильтры ближе к началу



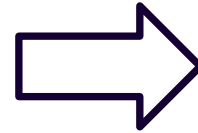
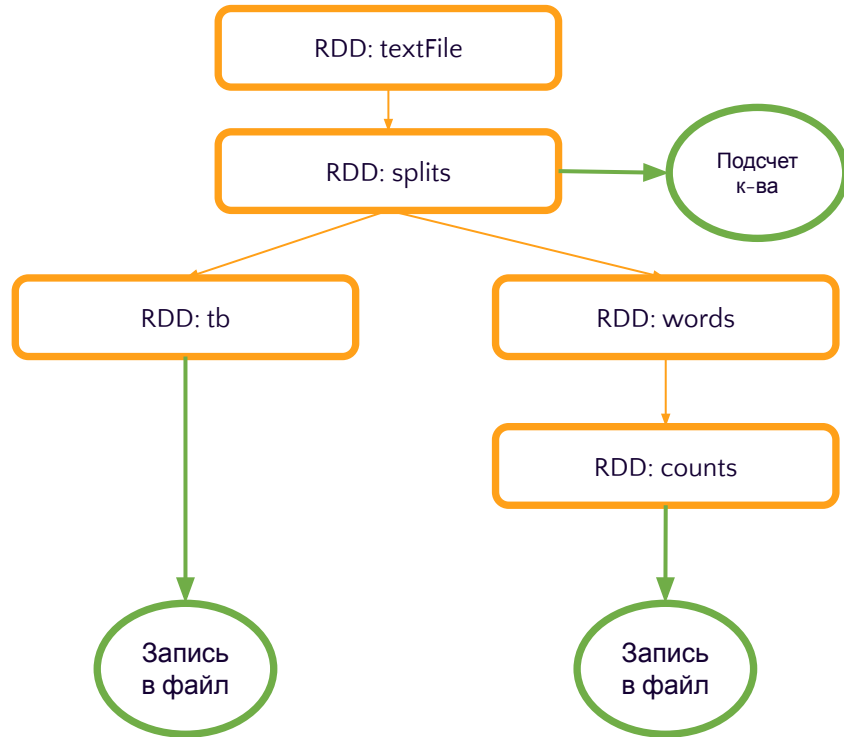
Необходимо заботиться о повторном вычислении

Каждое действие выполняется без оглядки на другое. Необходимо заботиться об избегании повторных вычислений.
cache(), persist()

Разрастание плана запросов

Особенно в итерационных алгоритмах. Здесь может помочь savepoint(), который сохраняет данные на диск и очищает lineage.

Lazy Evaluation кэширование



Lazy Evaluation кэширование

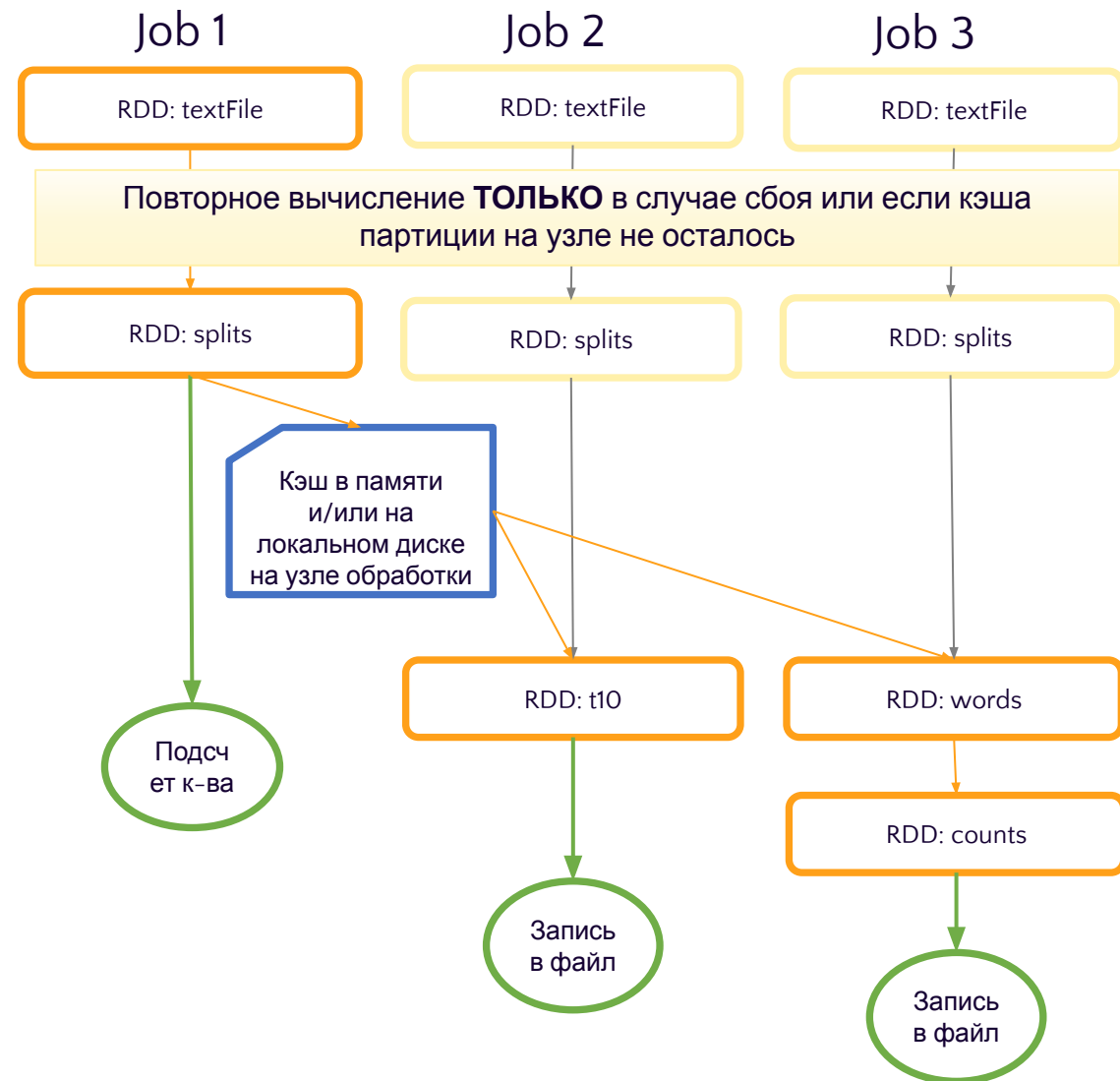
```
val textFile = sc.textFile("hdfs://...")
val splits = textFile.flatMap(line => line.split(" ")).cache()

val allwords = splits.count()

val tb = splits.filter(_.startsWith("b"))
tb.saveAsTextFile("hdfs://...")

val words = splits.map(word => (word, 1))
val counts = words.reduceByKey(_ + _)
counts.saveAsTextFile("hdfs://...")
```

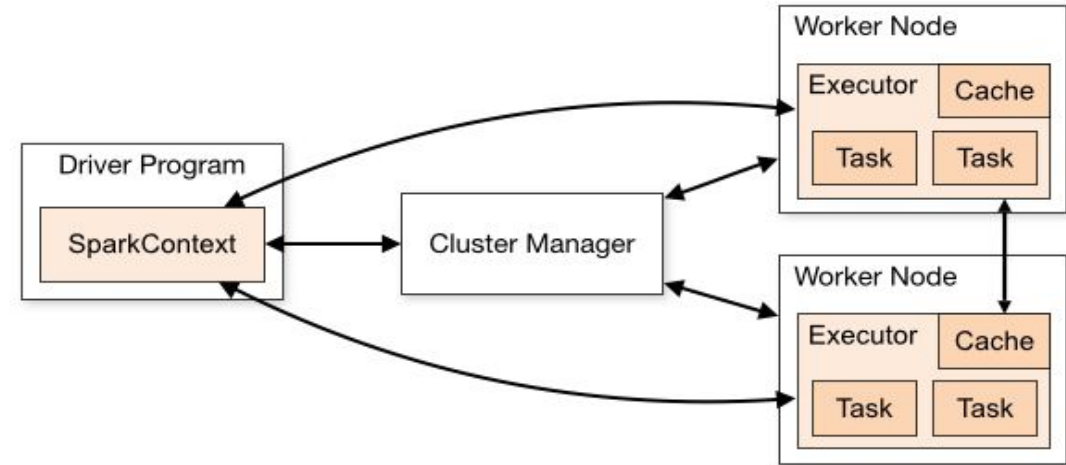
- Кэширование позволяет избежать **повторного вычисления** ветки графа
- Иногда кэширование может занять много памяти и времени и **быстрее будет повторно произвести вычисления**



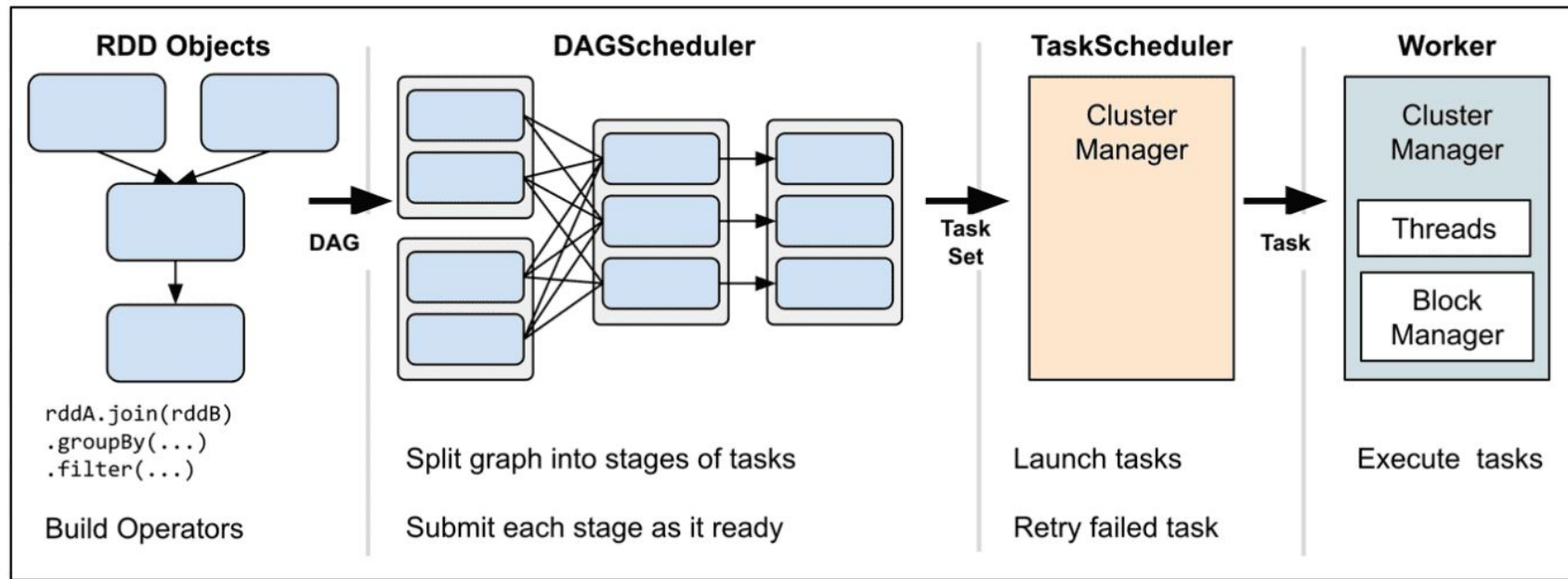
Как устроено приложение Spark

Каждая задача получает для выполнения:

- **num_executors** – к-во отдельных процессов JVM, в которых будут запущена потоки обработки данных(они могут быть расположены как на одном узле, так и на разных). Процессы будут работать до конца работы приложения.
- **executor_cores** – к-во параллельных потоков выполняемых в каждом ехесутор. Обработка данных идет в потоках.
- **executor-memory** – к-во памяти выделяемое каждому Executor
- **driver-memory** – к-во памяти выделяемое драйверу.



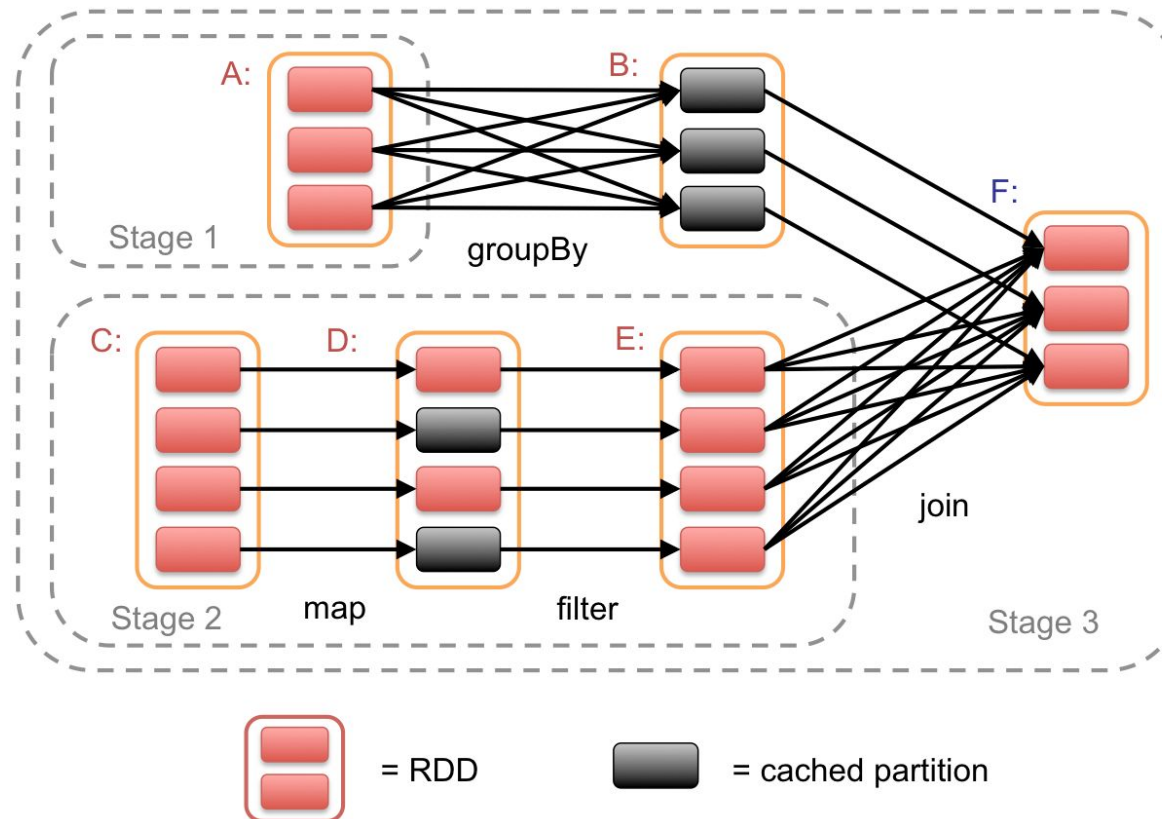
Приложение Spark



- Для **каждого действия** строится DAG выполнения
- DAG отправляется в DAGScheduler
- DAGScheduler разбивает его на этапы (stages) и отправляет на выполнение на TaskScheduler
- TaskScheduler использует менеджер кластера (Yarn, Mesos, Spark Standalone) для выделения ресурсов
- Каждый Executor получает от Driver задание (Tasks) и выполняет его над своей порцией данных
- Данные отсылаются на Driver или сохраняются в файл или кэшируются в памяти Executor

Приложение Spark

Этап это последовательность трансформаций разделенных Shuffle



**Звучит интересно, хочу
попробовать !!!**

1. Установить Java и Python, если будете работать в PySpark
2. Скачать Spark: <https://spark.apache.org/downloads.html>



Download Libraries Documentation Examples Community Developers

Download Apache Spark™

1. Choose a Spark release:
2. Choose a package type:
3. Download Spark: [spark-3.0.1-bin-hadoop2.7.tgz](#)
4. Verify this release using the 3.0.1 [signatures](#), [checksums](#) and [project release KEYS](#).

Note that, Spark 2.x is pre-built with Scala 2.11 except version 2.4.2, which is pre-built with Scala 2.12. Spark 3.0+ is pre-built with Scala 2.12.

3. Распаковать архив в любую папку
4. Установить переменную окружения SPARK_HOME на эту папку

...



5. Запустить:

bin/spark-shell – интерпретатор Scala

или

bin/pyspark – интерпретатор Python

```
20/09/17 19:57:50 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java
asses where applicable
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
Spark context Web UI available at http://ws36-044.neoflex.ru:4040
Spark context available as 'sc' (master = local[*], app id = local-1600361878150).
Spark session available as 'spark'.
Welcome to

      /--\
     /    \
    /      \
   /        \
  /          \
 /            \
/              \
\              /
 \            /
  \          /
   \        /
    \      /
     \    /
      --\

version 2.4.6

Using Scala version 2.11.12 (OpenJDK 64-Bit Server VM, Java 12.0.1)
Type in expressions to have them evaluated.
Type :help for more information.

scala> val x=sc.textFile("c:/test.txt")
x: org.apache.spark.rdd.RDD[String] = c:/test.txt MapPartitionsRDD[1] at textFile at <console>:24

scala> x.count()
res0: Long = 4

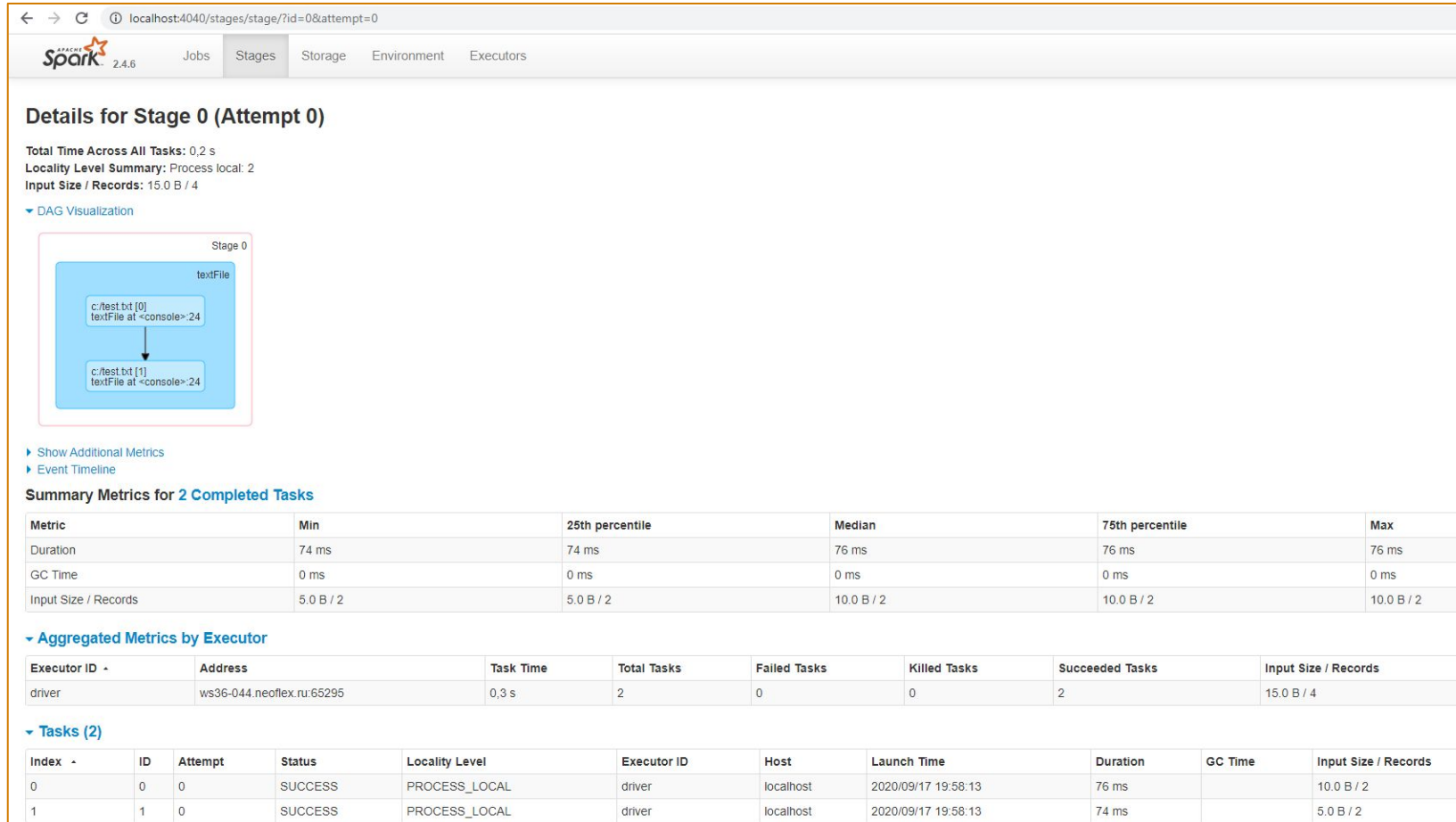
scala>
```

:q для выхода



Как можно попробовать Spark

6. Во время работы интерпретатора будет доступен Spark History Server <http://localhost:4040/>, где можно изучить как работает приложение Spark



Details for Stage 0 (Attempt 0)

Total Time Across All Tasks: 0,2 s
 Locality Level Summary: Process local: 2
 Input Size / Records: 15,0 B / 4

▼ DAG Visualization

```

  graph TD
    A["c:/test.txt [0]  
textFile at <console>:24"] --> B["c:/test.txt [1]  
textFile at <console>:24"]
  
```

► Show Additional Metrics
 ► Event Timeline

Summary Metrics for 2 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	74 ms	74 ms	76 ms	76 ms	76 ms
GC Time	0 ms	0 ms	0 ms	0 ms	0 ms
Input Size / Records	5,0 B / 2	5,0 B / 2	10,0 B / 2	10,0 B / 2	10,0 B / 2

▼ Aggregated Metrics by Executor

Executor ID	Address	Task Time	Total Tasks	Failed Tasks	Killed Tasks	Succeeded Tasks	Input Size / Records
driver	ws36-044.neoflex.ru:65295	0,3 s	2	0	0	2	15,0 B / 4

▼ Tasks (2)

Index	ID	Attempt	Status	Locality Level	Executor ID	Host	Launch Time	Duration	GC Time	Input Size / Records
0	0	0	SUCCESS	PROCESS_LOCAL	driver	localhost	2020/09/17 19:58:13	76 ms		10,0 B / 2
1	1	0	SUCCESS	PROCESS_LOCAL	driver	localhost	2020/09/17 19:58:13	74 ms		5,0 B / 2

**СПАСИБО
ЗА ВНИМАНИЕ!**

Подготовил: Алексей Попов БН BigData Solution

Читает: Андрей Журлов