Spark Motivation

- Difficultly of programming directly in Hadoop MapReduce
- Performance bottlenecks, or batch not fitting use cases
- Better support iterative jobs typical for machine learning

Difficulty of Programming in MR

Word Count implementations

- Hadoop MR – 61 lines in Java
- Spark – 1 line in interactive shell

```
sc.textFile('...').flatMap(lambda x: x.split())
    .map(lambda x: (x, 1)).reduceByKey(lambda x, y: x+y)
    .saveAsTextFile('...')
```

VS

```
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<Object, Text, Text, IntWritable>{

    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,IntWritable,Text,IntWritable> {
    private IntWritable result = new IntWritable();

    public void reduce(Text key, Iterable<IntWritable> 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.setCombinerClass(IntSumReducer.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);
  }
}
```

Performance Bottlenecks

How many times the data is put to the HDD during a single MapReduce Job?

- ~~One~~
- ~~Two~~
- ✓ *Three*
- ✓ *More*

Consider Hive as main SQL tool

- Typical Hive query is translated to 3-5 MR jobs
- Each MR would scan put data to HDD 3+ times
- Each put to HDD – write followed by read
- Sums up to 18-30 scans of data during a single Hive query

Performance Bottlenecks

Spark offers you

- Lazy Computations
  - Optimize the job before executing
- In-memory data caching
  - Scan HDD only once, then scan your RAM
- Efficient pipelining
  - Avoids the data hitting the HDD by all means

Spark Pillars

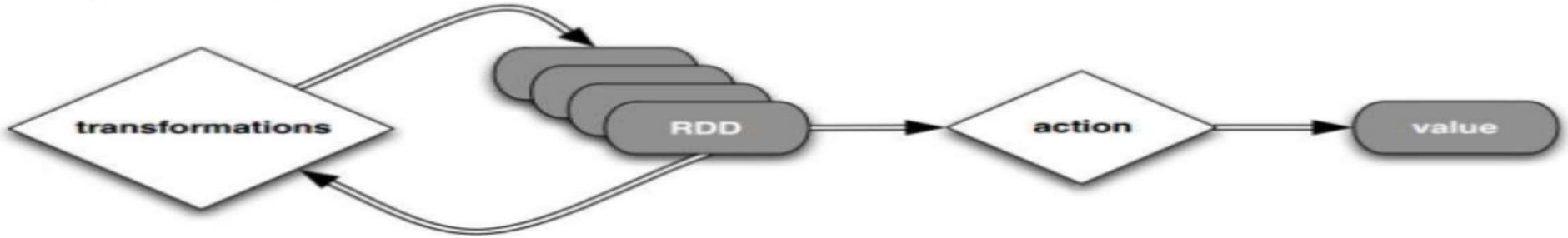
Two main abstractions of Spark

- **RDD** – Resilient Distributed Dataset
- **DAG** – Direct Acyclic Graph

RDD

- Simple view
  - RDD is collection of data items split into partitions and stored in memory on worker nodes of the cluster
- Complex view
  - RDD is an interface for data transformation
  - RDD refers to the data stored either in persisted store (HDFS, Cassandra, HBase, etc.) or in cache (memory, memory+disks, disk only, etc.) or in another RDD

Lazy computations model



Transformation cause only metadata change

DAG

Direct Acyclic Graph – sequence of computations performed on data

- *Node* – RDD partition
- *Edge* – transformation on top of data
- *Acyclic* – graph cannot return to the older partition
- *Direct* – transformation is an action that transitions data partition state (from A to B)

Spark Stack

Spark SQL

For SQL and unstructured data processing

MLib

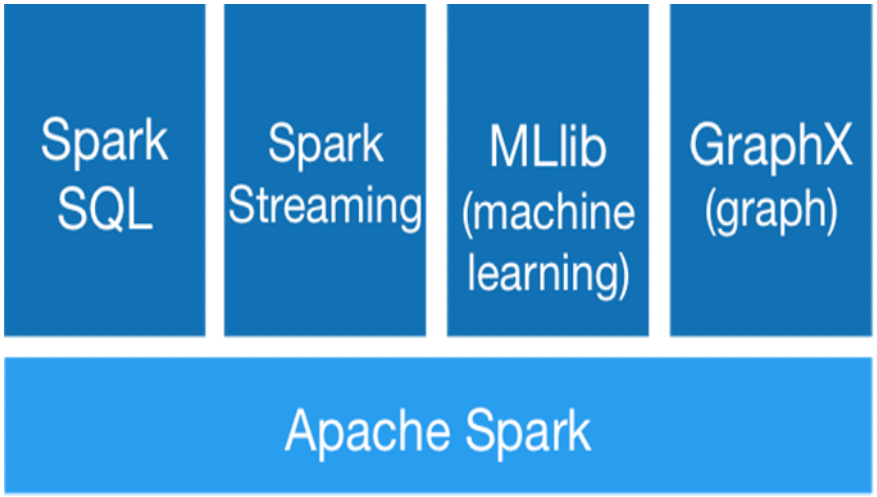
Machine Learning Algorithms

GraphX

Graph Processing

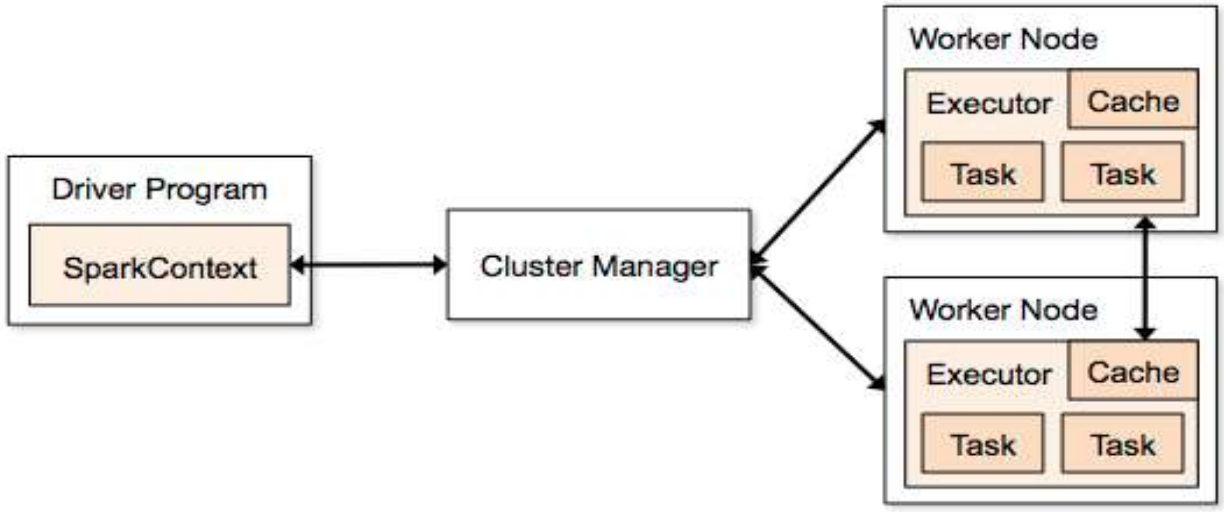
Spark Streaming

stream processing of live data streams



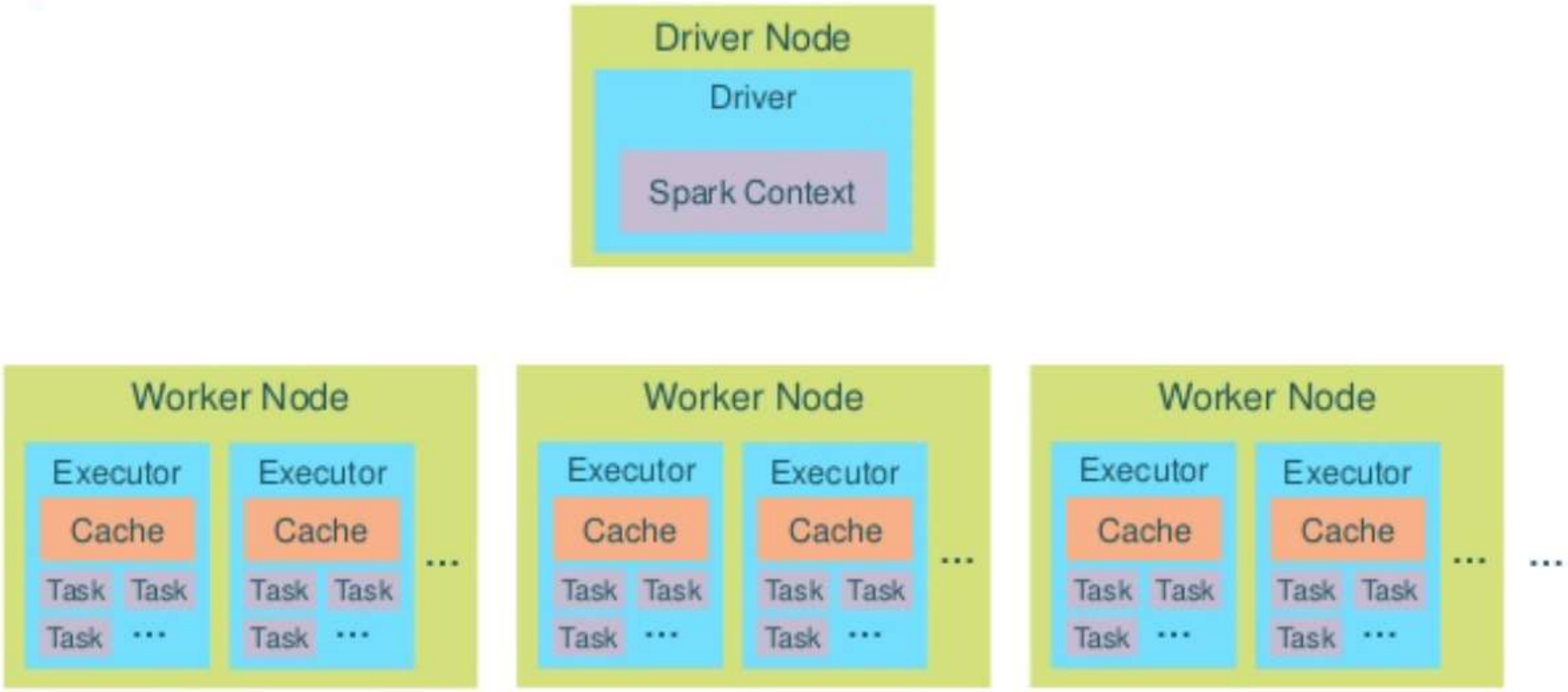
http://spark.apache.org

Execution Flow



http://spark.apache.org/docs/latest/cluster-overview.html

Spark Cluster



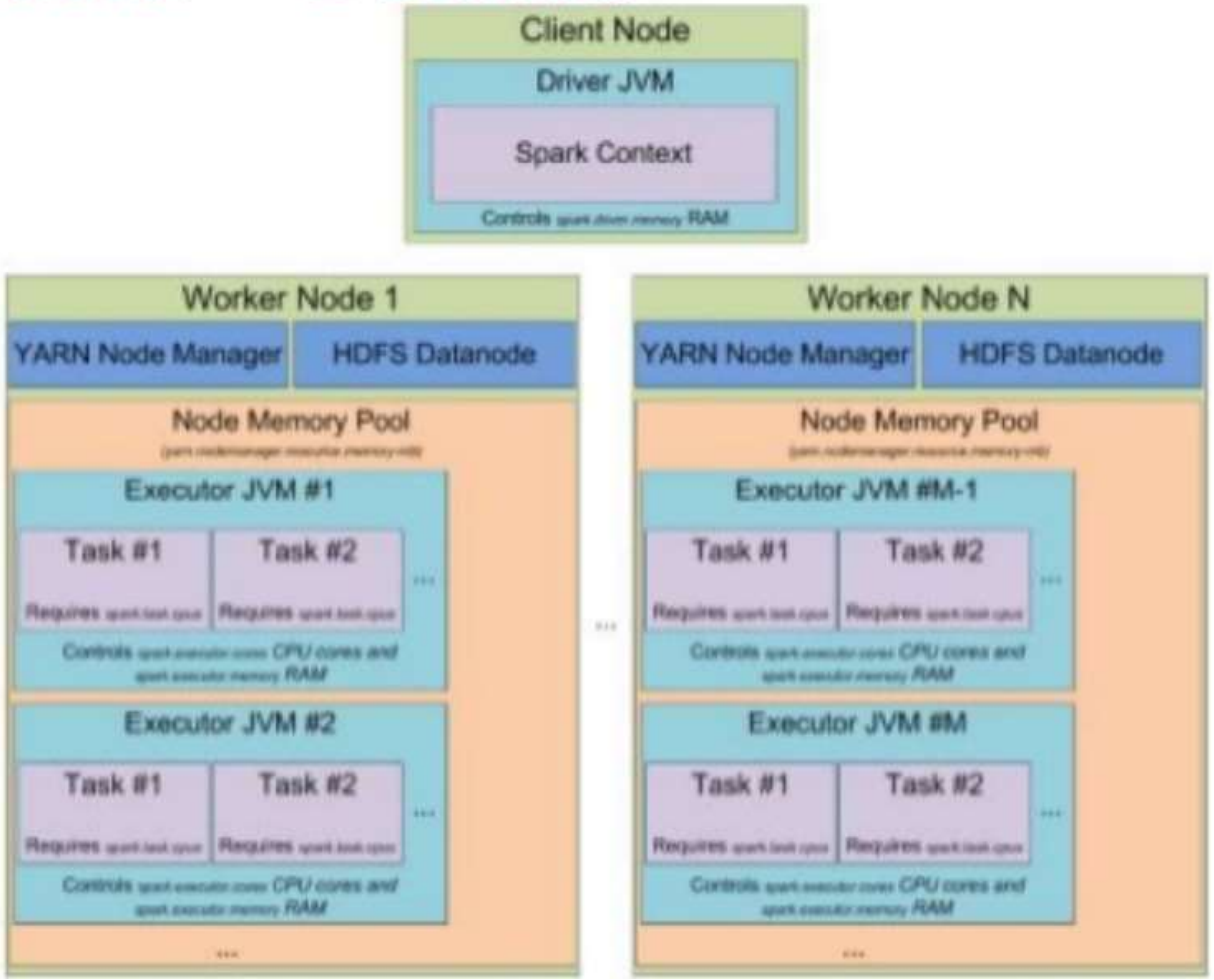
Spark Cluster

- **Driver**
  - Entry point of the Spark Shell (Scala, Python, R)
  - The place where SparkContext is created
  - Translates RDD into the execution graph
  - Splits graph into stages
  - Schedules tasks and controls their execution
  - Stores metadata about all the RDDs and their partitions
  - Brings up Spark WebUI with job information

Spark Cluster

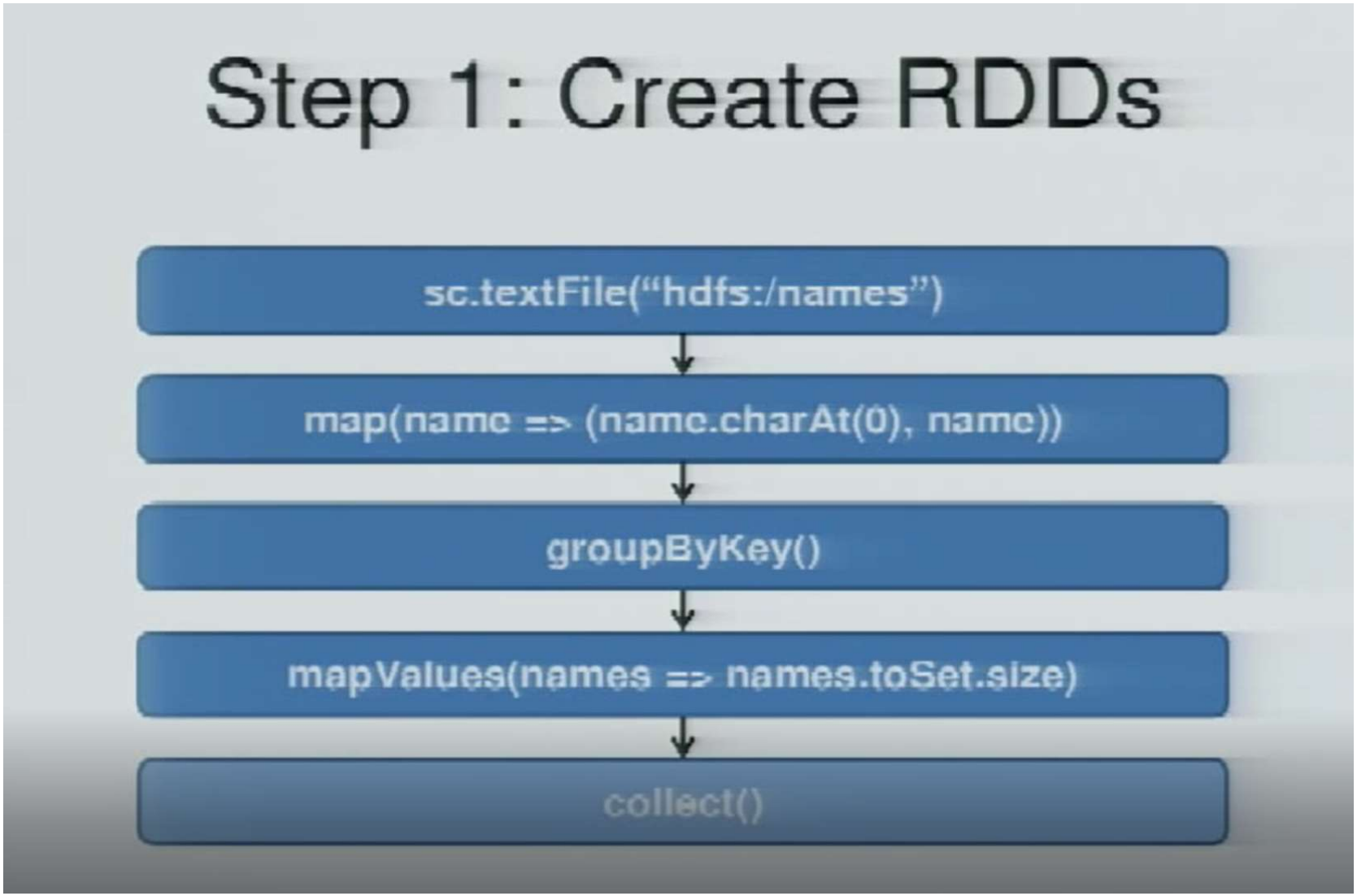
- **Executor**
  - Stores the data in cache in JVM heap or on HDDs
  - Reads data from external sources
  - Writes data to external sources
  - Performs all the data processing

Spark Cluster – Detailed



DEMO

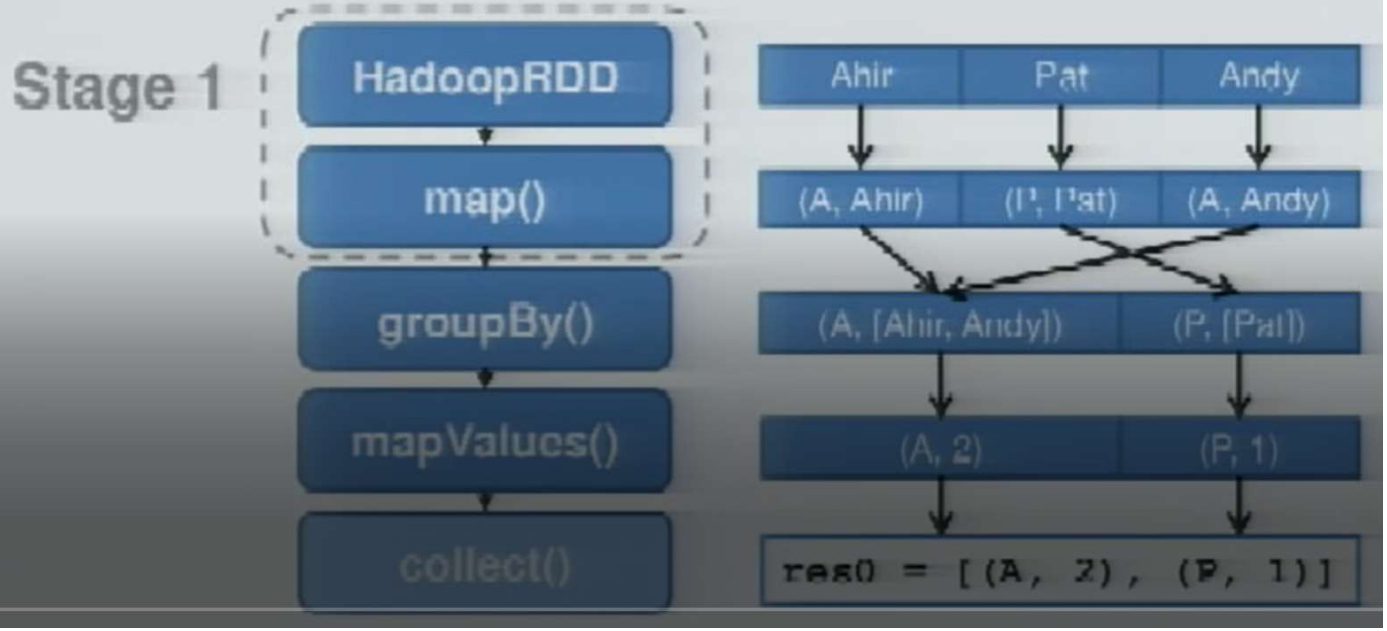
Step 1: Create RDDs

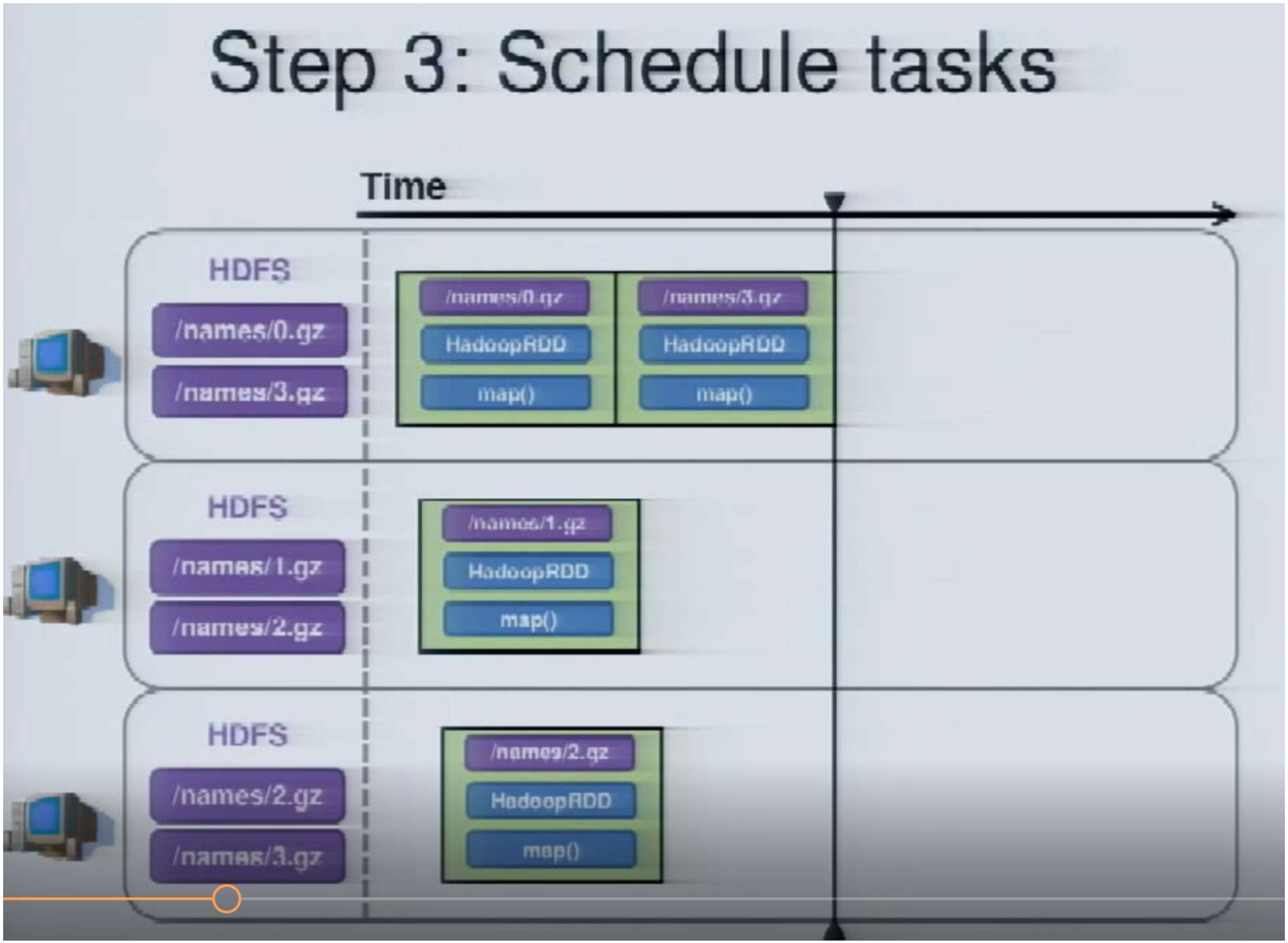


Step 2: Create execution plan

Pipeline as much as possible

Split into "**stages**" based on need to reorganize data




Step 3: Schedule tasks
Time
HDFS
/names/0.gz
/names/3.gz
/names/0.gz
HadoopRDD
map()
/names/3.gz
HadoopRDD
map()
HDFS
/names/1.gz
/names/2.gz
/names/1.gz
HadoopRDD
map()
HDFS
/names/2.gz
/names/3.gz
/names/2.gz
HadoopRDD
map()

The Shuffle

- Redistributes data among partitions
- Hash keys into buckets
- Optimizations:
  - Avoided when possible, if data is already properly partitioned
  - Partial aggregation reduces data movement