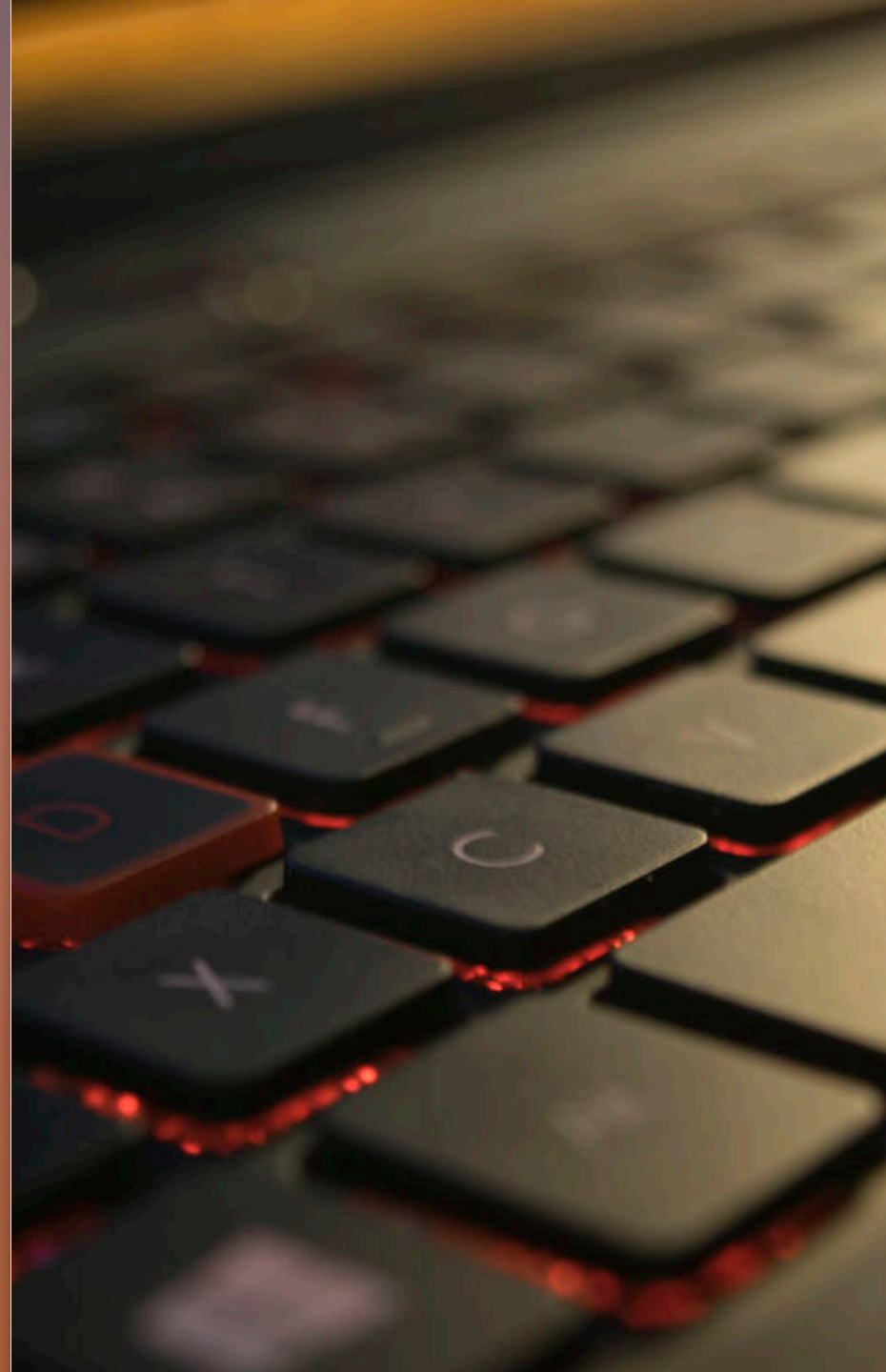
SALK

INTRODUCTION TO SPARK

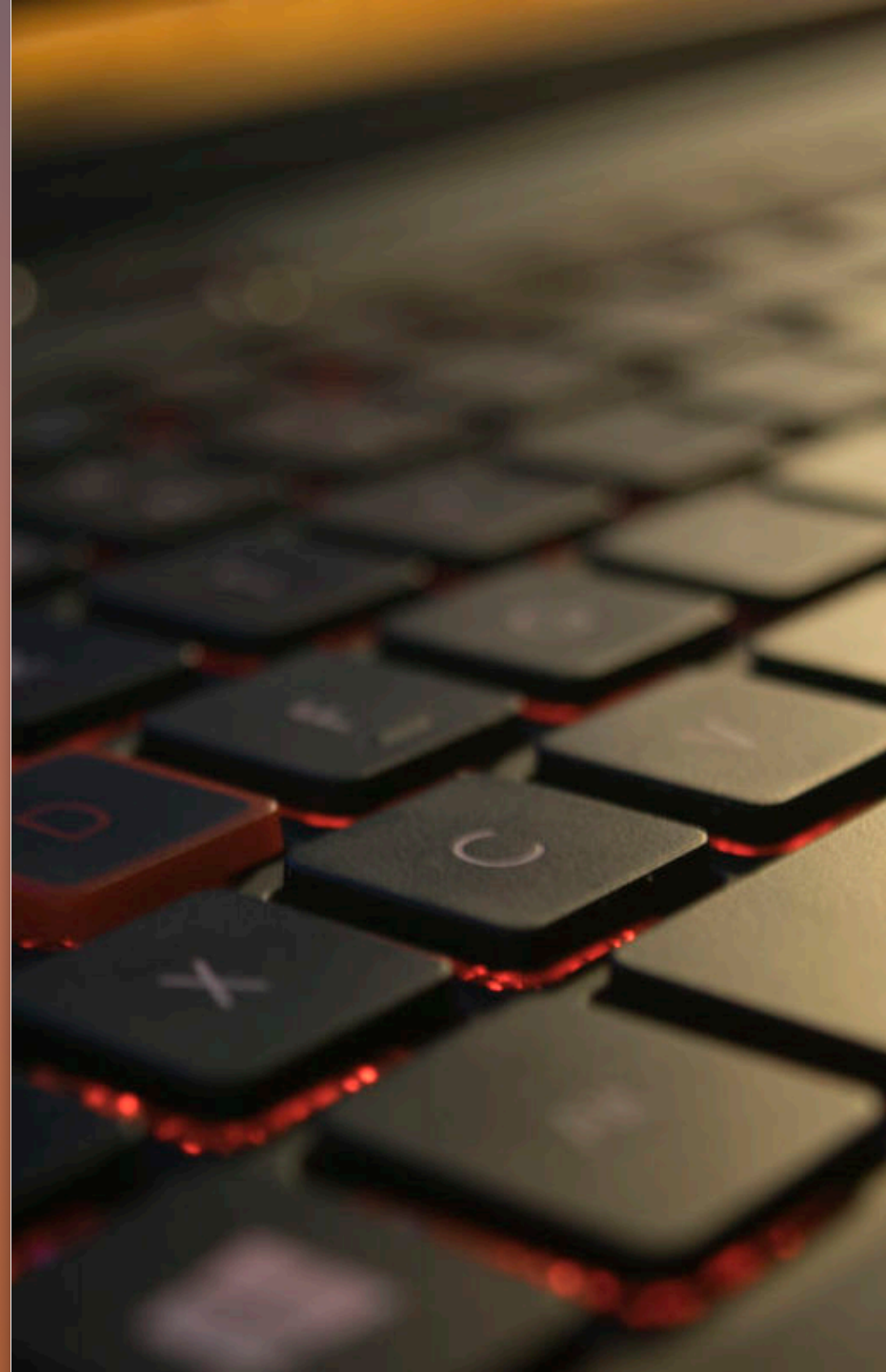
Mehdi LAMRANI

April 2021

SESSION ¼
SPARK FOUNDATIONS



THEORY 1/2



SALK

YOUR TRAINER:

- Mehdi LAMRANI
- Paris Electronic Engineering School
- ESSEC Msc in Quantitative Finance
- Senior Java Architect (20+ Years)
- Algorithmic Trading & Large Scale Computing Specialist
- Big Data Architect (Major French Companies)
- Big Data EMEA Trainer for HortonWorks / Cloudera / Databricks
- Founder of SALK / SYLACE Analytics

GOALS :

- Provide a broad introduction to SPARK Architecture & features
- Get Hands-On experience on many parts of the SPARK ecosystem
- Provide insight on some internal aspects for a better mastery
- See some advanced concepts (Performance Tuning & Monitoring)
- Use an ML example to illustrate & apply these concepts

Foreword (Course Philosophy)

- This course alternates between theory and practice sessions
- During theory Sessions
  - We will NOT focus much on syntax, but rather on concepts
  - Once you have a clear undersanding of concepts,
    syntax follows naturally and is easily searchable
- During practice Sessions :
  - Code examples & guidelines will be provided
  - Comprehensive References to syntax will be provided for you to pick from

Foreword (Course Philosophy)

- What this course is / What to Expect
  - Spark Theory, General Architecture
  - A quick tour of practical APIs
  - Insights into how things really work behind the scenes
- What this course is *not* / What *not* to Expect
  - A Data science course on Spark
  - A practical tutorial on Spark (*there are plenty of available resources for that*)

Foreword (Course Philosophy)

- Before we dive in :
  - Spark Theory has **a LOT of moving pieces** and building blocks
  - We need to go through these abstract layers and tackle them
  - This program purposefully puts an emphasis on these inner aspects
  - Give it time… to absorb and digest all the concepts

Again, practical howto's & tutorials are easily accessible,
but tackling the underlying aspects is less obvious,
and that's what we are here for.

“ Boxing is a science. You don't just walk into a gym and start punching.”

Eddie Futch

FOREWORD

SALK

COURSE OBJECTIVE :

- Spark makes easier what was previsouly much harder to achieve
- It does come however with a load of inherent hardships though, due to is internal complexity by design
- Understanding some of these knots and bolts help develop a healthy mindset when it comes to dealing with some of the frustrations that can arise when dealing with spark computing

  - *What is my job actually doing ?*
  - *Why is it taking so long ?*
  - *Is my job stuck ?*
  - *Should I just relaunch my request ?*
  - *Am I having enough resources ?*
  - *How can I optimize my spark code ?*
  - *etc.*

- ✓ Definition
- History & Context
- Multi Language Support
- Distributed Computing Architecture
- Spark Compionents
- Spark Integration & Access Points
- Resilient Distributed Dataset
- Spark Session
- Lazy Evaluation
- Transformations & Actions
- Narrow & Wide Transformations
- RDD Lineage
- Anatomy of a Spark Application
- Directed Acyclic Graph

SALK

WHAT IS SPARK :

- SPARK is an in-memory dataset-centric distributed computing framework

101 / in a Nutshell :

- Through Spark, Data is input > Manipulated > Spit out
  - ETL Analytics Business as Usual
  - Except data size can be huge & processed in parallel
  - This is specifically where spark shines :
    When your pandas dataframe buddy & monocore python just would'nt take it.

SALK

Think of it as a Data Manipulation API

- What does it actually look like ?

```
val df = spark.read.csv("src/main/resources/zipcodes.csv")
    df.printSchema()
```

SALK

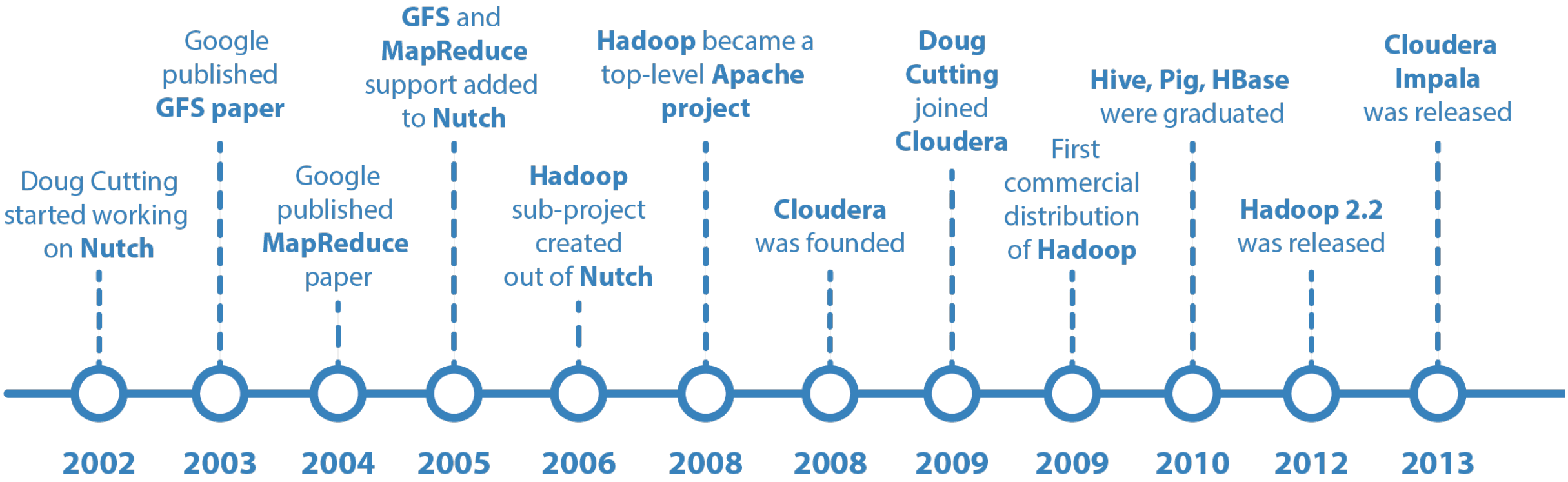
```
emp = [(1,"Smith",-1,"2018","10","M",3000), \
    (2,"Rose",1,"2010","20","M",4000), \
    (3,"Williams",1,"2010","10","M",1000), \
    (4,"Jones",2,"2005","10","F",2000), \
    (5,"Brown",2,"2010","40","",-1), \
      (6,"Brown",2,"2010","50","",-1) \
  ]
empColumns = ["emp_id","name","superior_emp_id","year_joined",
       "emp_dept_id","gender","salary"]

empDF = spark.createDataFrame(data=emp, schema = empColumns)
```

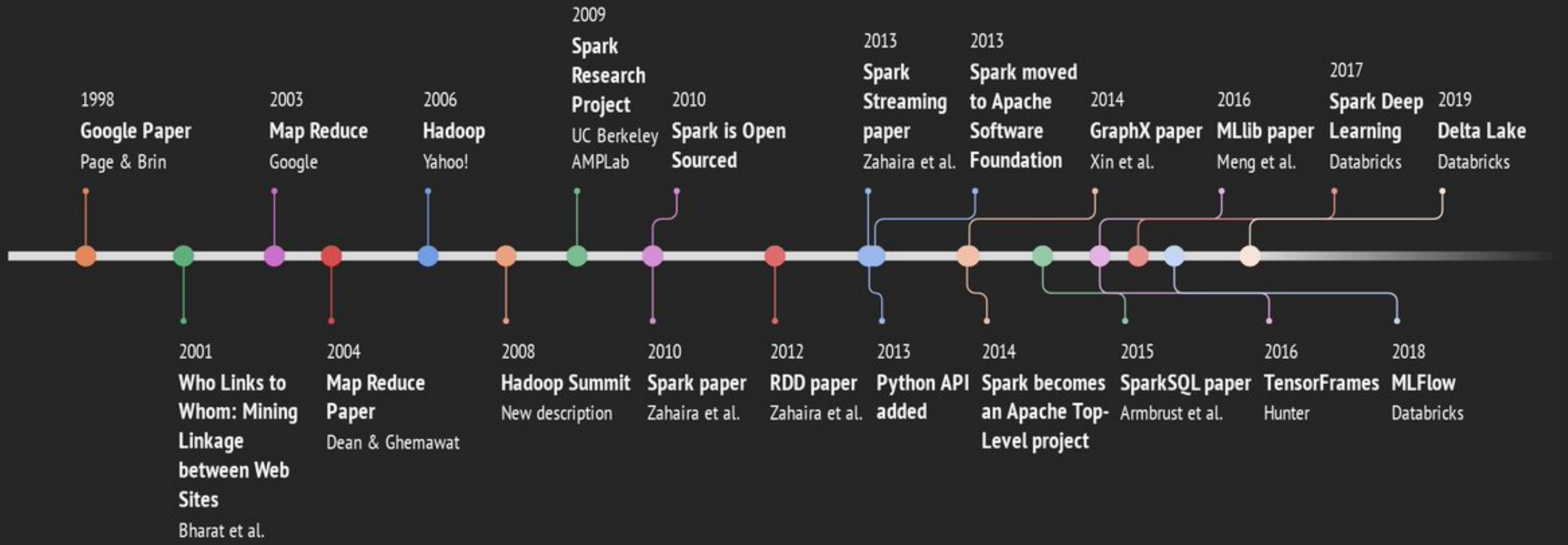
Now you can play with it !

select, combine, join, aggregate…

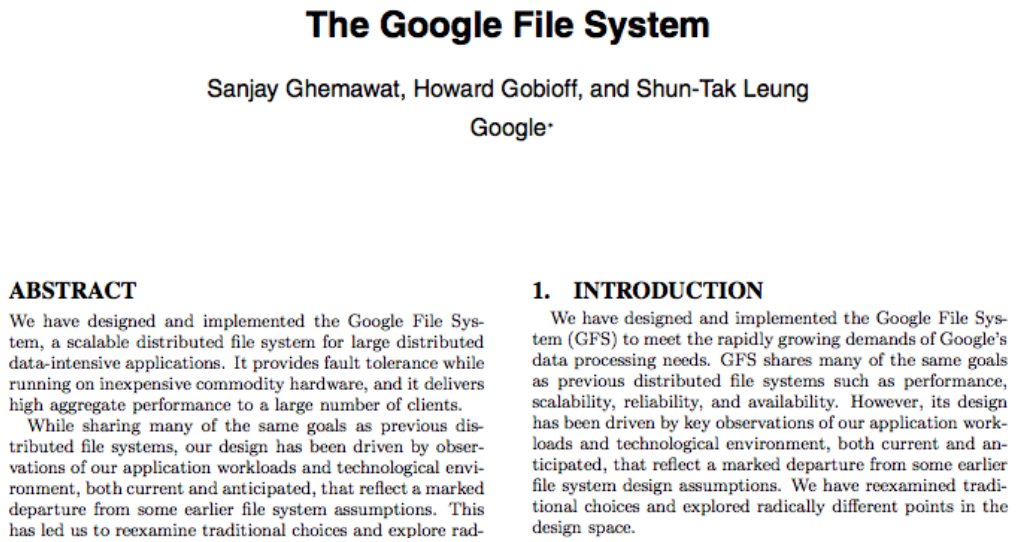
- ✓ Definition
- ✓ History & Context
- Multi Language Support
- Distributed Computing Architecture
- Spark Components
- Spark Integration & Access Points
- Resilient Distributed Dataset
- Spark Session
- Lazy Evaluation
- Transformations & Actions
- Narrow & Wide Transformations
- RDD Lineage
- Anatomy of a Spark Application
- Directed Acyclic Graph


Doug Cutting started working on **Nutch**
2002
Google published **GFS paper**
2003
Google published **MapReduce** paper
2004
GFS and **MapReduce** support added to **Nutch**
2005
Hadoop sub-project created out of **Nutch**
2006
Hadoop became a top-level **Apache project**
2008
Cloudera was founded
2008
Doug Cutting joined **Cloudera**
2009
First commercial distribution of **Hadoop**
2009
Hive, Pig, HBase were graduated
2010
Hadoop 2.2 was released
2012
Cloudera Impala was released
2013

Apache Spark Ecosystem Timeline



By Favio Vázquez @faviovaz



The Google File System

Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung
Google*

ABSTRACT

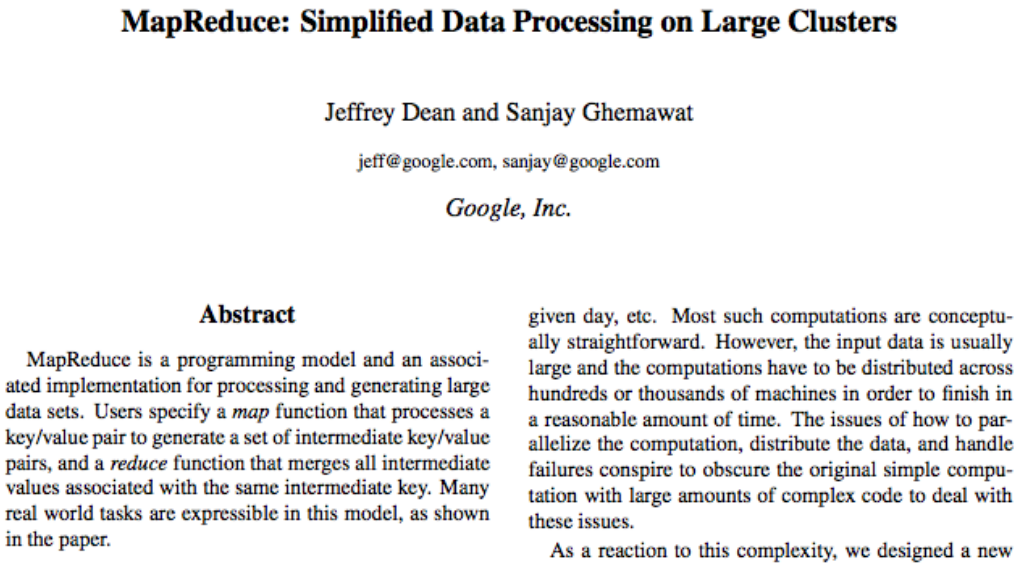
We have designed and implemented the Google File System, a scalable distributed file system for large distributed data-intensive applications. It provides fault tolerance while running on inexpensive commodity hardware, and it delivers high aggregate performance to a large number of clients.

While sharing many of the same goals as previous distributed file systems, our design has been driven by observations of our application workloads and technological environment, both current and anticipated, that reflect a marked departure from some earlier file system assumptions. This has led us to reexamine traditional choices and explore rad-

1. INTRODUCTION

We have designed and implemented the Google File System (GFS) to meet the rapidly growing demands of Google's data processing needs. GFS shares many of the same goals as previous distributed file systems such as performance, scalability, reliability, and availability. However, its design has been driven by key observations of our application workloads and technological environment, both current and anticipated, that reflect a marked departure from some earlier file system design assumptions. We have reexamined traditional choices and explored radically different points in the design space.

- Good Read : https://www.usenix.org/system/files/conference/nsdi12/nsdi12-final138.pdf



MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat

jeff@google.com, sanjay@google.com

Google, Inc.

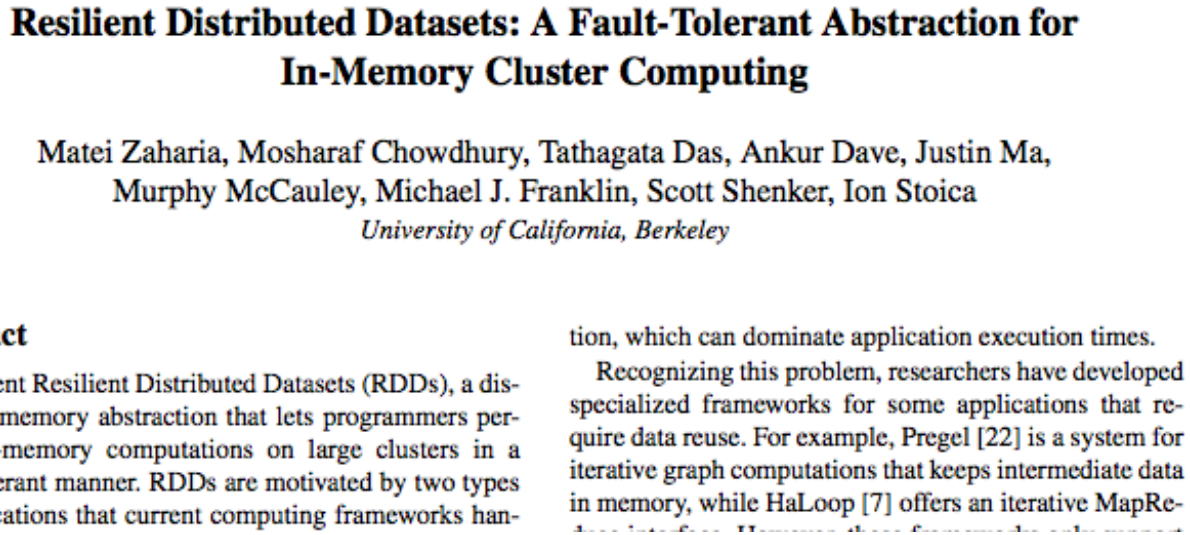
Abstract

MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a *map* function that processes a key/value pair to generate a set of intermediate key/value pairs, and a *reduce* function that merges all intermediate values associated with the same intermediate key. Many real world tasks are expressible in this model, as shown in the paper.

given day, etc. Most such computations are conceptually straightforward. However, the input data is usually large and the computations have to be distributed across hundreds or thousands of machines in order to finish in a reasonable amount of time. The issues of how to parallelize the computation, distribute the data, and handle failures conspire to obscure the original simple computation with large amounts of complex code to deal with these issues.

As a reaction to this complexity, we designed a new

- Good Read : https://static.googleusercontent.com/media/research.google.com/en//archive/mapreduce-osdi04.pdf



Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing

Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma,
Murphy McCauley, Michael J. Franklin, Scott Shenker, Ion Stoica
University of California, Berkeley

Abstract

We present Resilient Distributed Datasets (RDDs), a distributed memory abstraction that lets programmers perform in-memory computations on large clusters in a fault-tolerant manner. RDDs are motivated by two types of applications that current computing frameworks handle, which can dominate application execution times.

Recognizing this problem, researchers have developed specialized frameworks for some applications that require data reuse. For example, Pregel [22] is a system for iterative graph computations that keeps intermediate data in memory, while HaLoop [7] offers an iterative MapRe-

- Good Read : https://www.usenix.org/system/files/conference/nsdi12/nsdi12-final138.pdf

MAPREDUCE QUICK REMINDERS

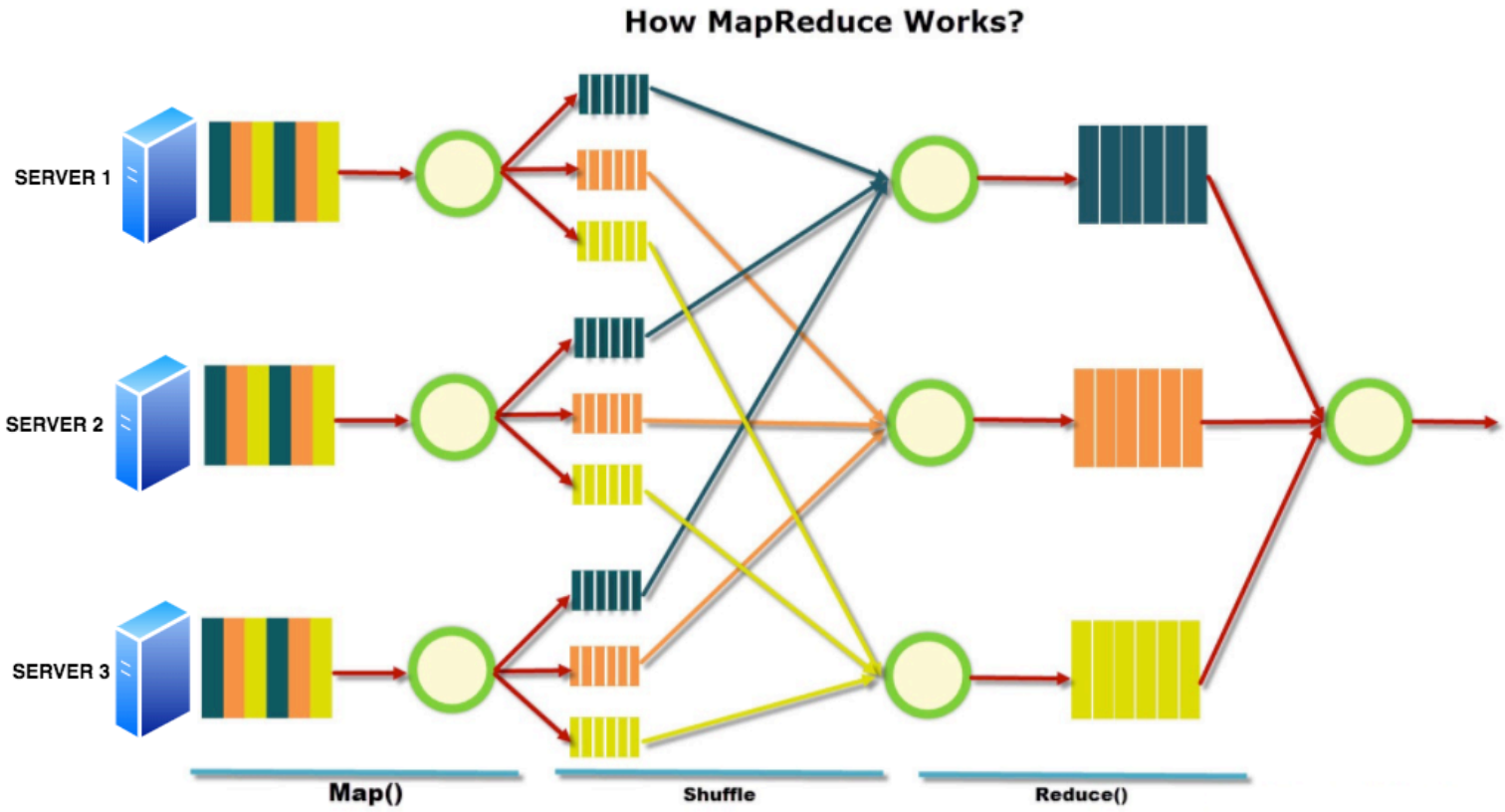
- Why bother ?
  - Spark creators used MR as a starting thinking point
  - The fundamental concepts of MR still apply to Spark
  - Spark is a (huge) improvement of the ideas behind MR and its implementation.

Map Reduce

- framework for processing parallelizable problems across large datasets using multiple computers
- Data is physically stored / distributed over multiple servers
- A Map phase performs local filtering/sorting operations
- A shuffle phase redistributes data across servers accordingly
- A reduce phase performs result grouping tasks

Map Reduce Example

- Let's say our distributed dataset lists customer data
- We want to count customers by continent (NAMR/EUR/APAC)
- Every part of the data on each server has a mix of each
- MAP phase would sort them locally
- SUFFLE would aggregate each customer group together across servers
- REDUCE phase would then perform counts for each group



Pretty clever...
Any limitations ?

SALK

- MAPREDUCE LIMITATION ONE :
  - Cluttered design and boilerplated code model

Source Code

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.net.URI;
import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.Set;
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;
import org.apache.hadoop.mapreduce.Counter;
import org.apache.hadoop.util.GenericOptionsParser;
import org.apache.hadoop.util.StringUtils;

public class WordCount2 {

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

    static enum CountersEnum { INPUT_WORDS }

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

    private boolean caseSensitive;
    private Set<String> patternsToSkip = new HashSet<String>();

    private Configuration conf;
    private BufferedReader fis;

    @Override
    public void setup(Context context) throws IOException,
        InterruptedException {
      conf = context.getConfiguration();
      caseSensitive = conf.getBoolean("wordcount.case.sensitive", true);
      if (conf.getBoolean("wordcount.skip.patterns", false)) {
        URI[] patternsURIs = Job.getInstance(conf).getCacheFiles();
        for (URI patternsURI : patternsURIs) {
          Path patternsPath = new Path(patternsURI.getPath());
          String patternsFileName = patternsPath.getName().toString();
          parseSkipFile(patternsFileName);
        }
      }
    }

    private void parseSkipFile(String fileName) {
      try {
        fis = new BufferedReader(new FileReader(fileName));
        String pattern = null;
        while ((pattern = fis.readLine()) != null) {
          patternsToSkip.add(pattern);
        }
      } catch (IOException ioe) {
        System.err.println("Caught exception while parsing the cached file '"
            + StringUtils.stringifyException(ioe));
      }
    }

    @Override
    public void map(Object key, Text value, Context context
                    ) throws IOException, InterruptedException {
      String line = (caseSensitive) ?
          value.toString() : value.toString().toLowerCase();
      for (String pattern : patternsToSkip) {
        line = line.replaceAll(pattern, "");
      }
      StringTokenizer itr = new StringTokenizer(line);
      while (itr.hasMoreTokens()) {
        word.set(itr.nextToken());
        context.write(word, one);
        Counter counter = context.getCounter(CountersEnum.class.getName(),
            CountersEnum.INPUT_WORDS.toString());
        counter.increment(1);
      }
    }
  }
```

MAPREDUCE WORDCOUNT

```
    private void parseSkipFile(String fileName) {
      try {
        fis = new BufferedReader(new FileReader(fileName));
        String pattern = null;
        while ((pattern = fis.readLine()) != null) {
          patternsToSkip.add(pattern);
        }
      } catch (IOException ioe) {
        System.err.println("Caught exception while parsing the cached file '"
            + StringUtils.stringifyException(ioe));
      }
    }

    @Override
    public void map(Object key, Text value, Context context
                    ) throws IOException, InterruptedException {
      String line = (caseSensitive) ?
          value.toString() : value.toString().toLowerCase();
      for (String pattern : patternsToSkip) {
        line = line.replaceAll(pattern, "");
      }
      StringTokenizer itr = new StringTokenizer(line);
      while (itr.hasMoreTokens()) {
        word.set(itr.nextToken());
        context.write(word, one);
        Counter counter = context.getCounter(CountersEnum.class.getName(),
            CountersEnum.INPUT_WORDS.toString());
        counter.increment(1);
      }
    }
  }

  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();
    GenericOptionsParser optionParser = new GenericOptionsParser(conf, args);
    String[] remainingArgs = optionParser.getRemainingArgs();
    if ((remainingArgs.length != 2) && (remainingArgs.length != 4)) {
      System.err.println("Usage: wordcount <in> <out> [-skip skipPatternFile]");
      System.exit(2);
    }
    Job job = Job.getInstance(conf, "word count");
    job.setJarByClass(WordCount2.class);
    job.setMapperClass(TokenizerMapper.class);
    job.setCombinerClass(IntSumReducer.class);
    job.setReducerClass(IntSumReducer.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);

    List<String> otherArgs = new ArrayList<String>();
    for (int i=0; i < remainingArgs.length; ++i) {
      if ("-skip".equals(remainingArgs[i])) {
        job.addCacheFile(new Path(remainingArgs[++i]).toUri());
        job.getConfiguration().setBoolean("wordcount.skip.patterns", true);
      } else {
        otherArgs.add(remainingArgs[i]);
      }
    }
    FileInputFormat.addInputPath(job, new Path(otherArgs.get(0)));
    FileOutputFormat.setOutputPath(job, new Path(otherArgs.get(1)));

    System.exit(job.waitForCompletion(true) ? 0 : 1);
  }
}
```

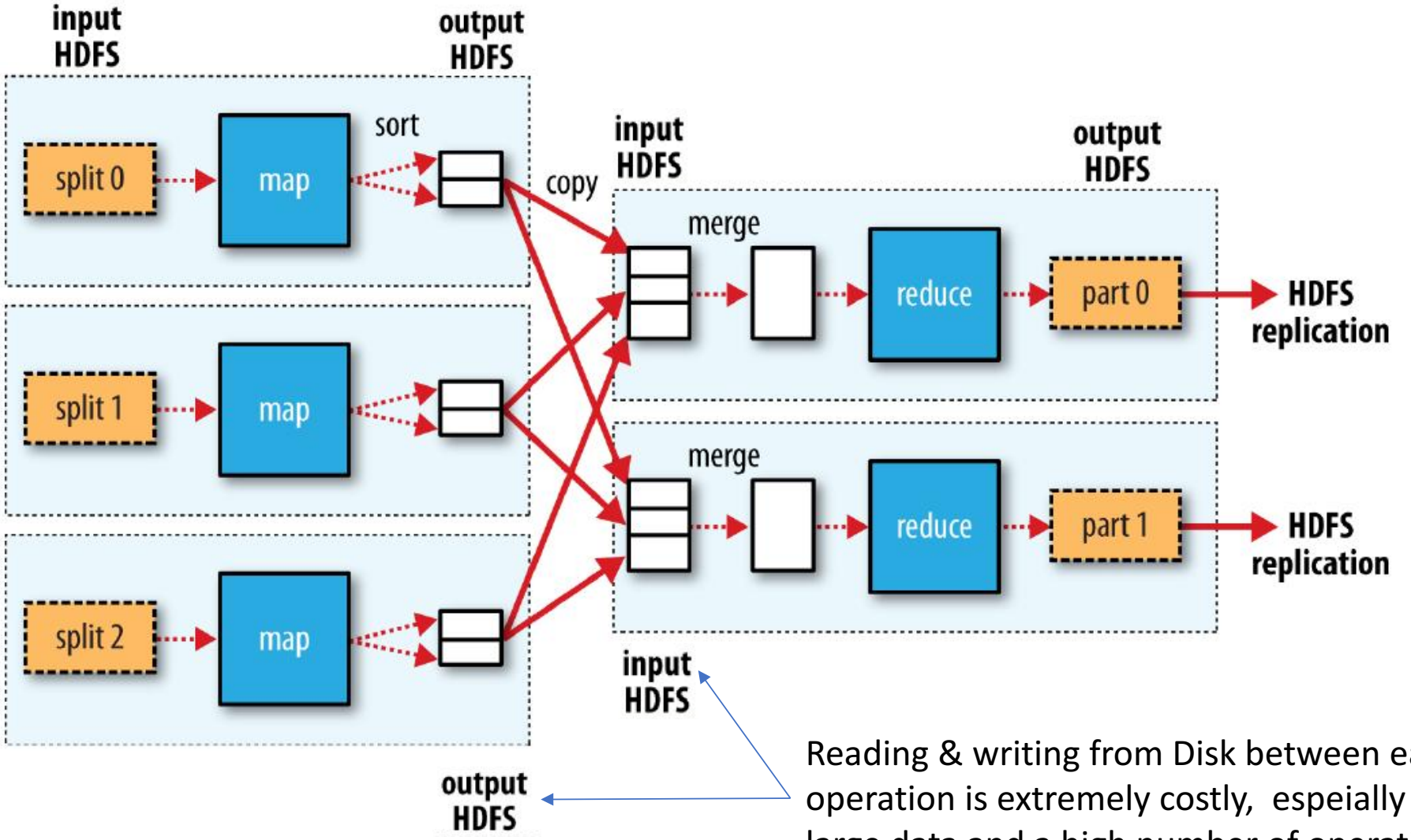
MAPREDUCE WORDCOUNT

●●●

APACHE Spark

SALK

- MAPREDUCE LIMITATION TWO :
  - Intermediate storage has to take place between each phase



Reading & writing from Disk between each operation is extremely costly, espeially we have large data and a high number of operations

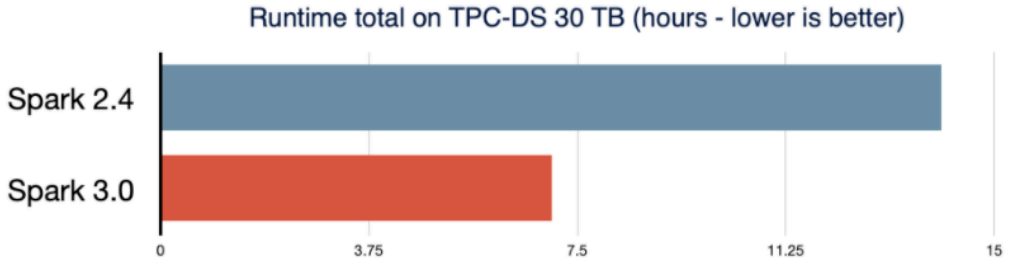
- SPARK Solves this crucial issue by leveraging memory directly.
- Instructions & data are serialized directly from server memory to server memory and do not require (or at least minimize considerably) writing and reading from disk
- Less IO Throughput
- This is a key feature of SPARK that makes it "in-memory" and accounts for much of its speed & performance

→ how spark leverages closures

SALK

- It' open source (for real)
- Backed up by a strong community
- Supported by large industry-leading companies
- Source code diving is rewarding – Not all internal aspects are well documented.
- Advantages : Stop guessing, just read the code !
  - Example Questions that only source code can (really) answer
    - How spark standalone cluster differs from Yarn plugin ?
    - How is data phisically distributed between workers ?
    - How is the the DAG generated ?
- Don't be shy, go on and clone the repo !

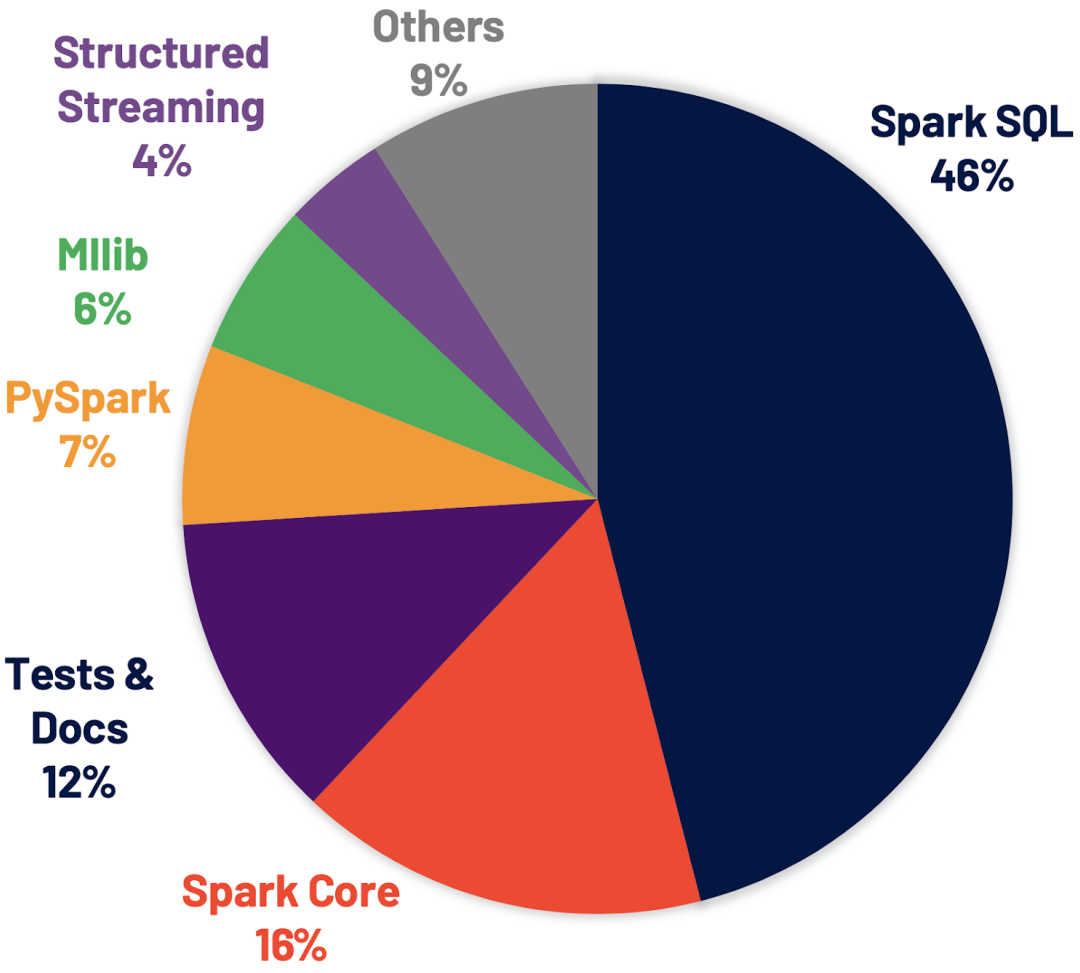
  https://github.com/apache/spark



SPARK 3

- Fresh out of the Box (GA in June 2020)

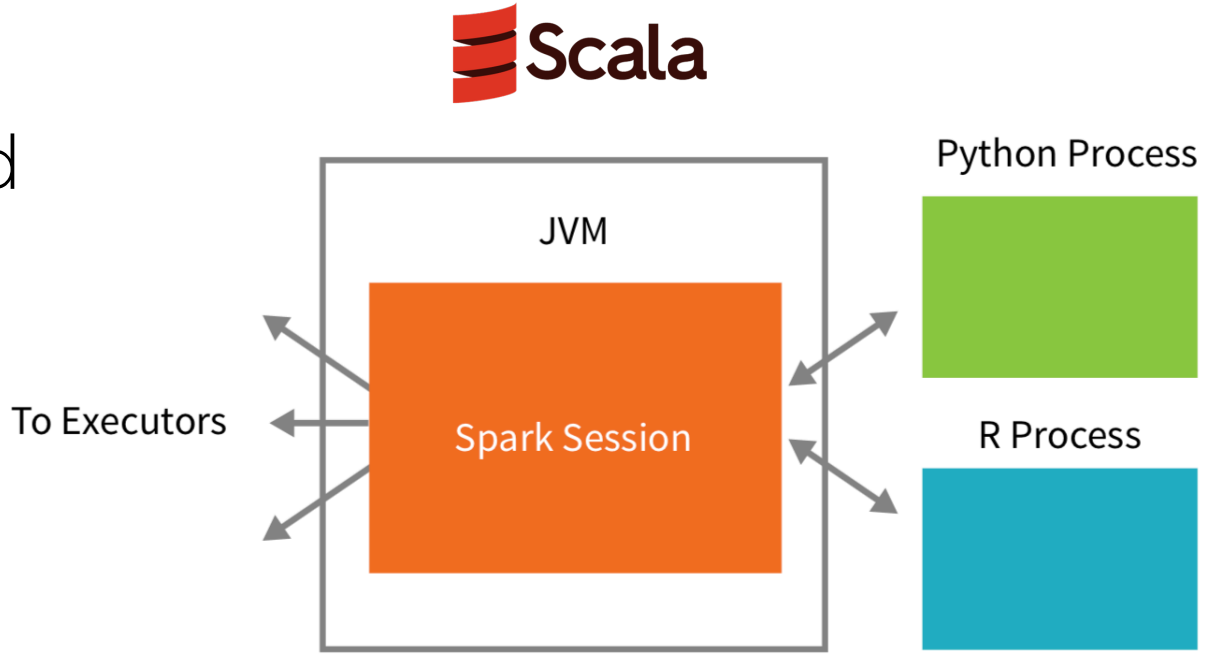
- 2x performance improvement on TPC-DS over Spark 2.4, enabled by many optimizations
- ANSI SQL compliance
- Significant improvements in pandas APIs (Koalas)
- Better Python error handling, simplifying PySpark exceptions
- New UI for structured streaming
- Up to 40x speedups for calling R user-defined functions
- Over 3,400 Jira tickets resolved



- ✓ Definition
- ✓ History & Context
- ✓ Multi Language Support
- Distributed Computing Architecture
- Spark Components
- Spark Integration & Access Points
- Resilient Distributed Dataset
- Spark Session
- Lazy Evaluation
- Transformations & Actions
- Narrow & Wide Transformations
- RDD Lineage
- Anatomy of a Spark Application
- Directed Acyclic Graph

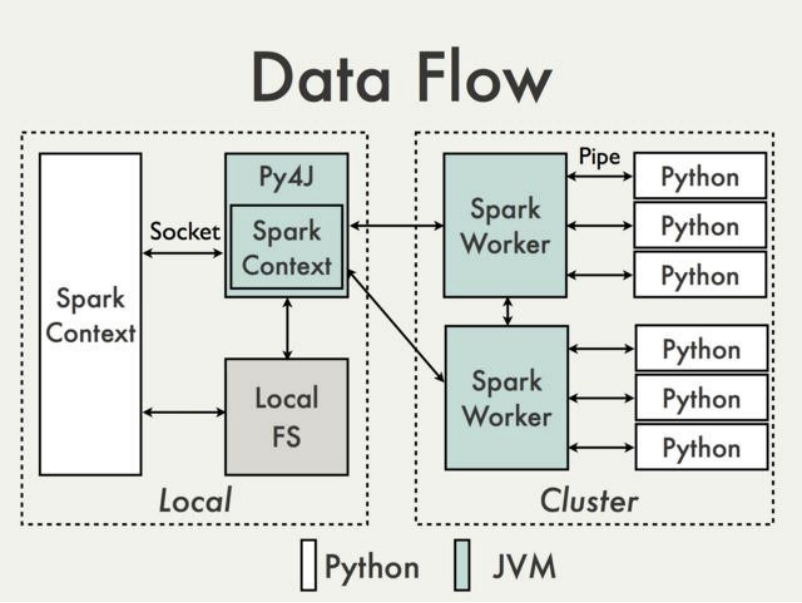
- # Python & R supported

- Everything at the core of Spark is natively SCALA / Java though
- Python API is an additionnal layer on top of Spark Scala
- Utilizes Py4j as a bridge between both worlds



- There is (much) more to the story…

Out of scope
(but interesting…)



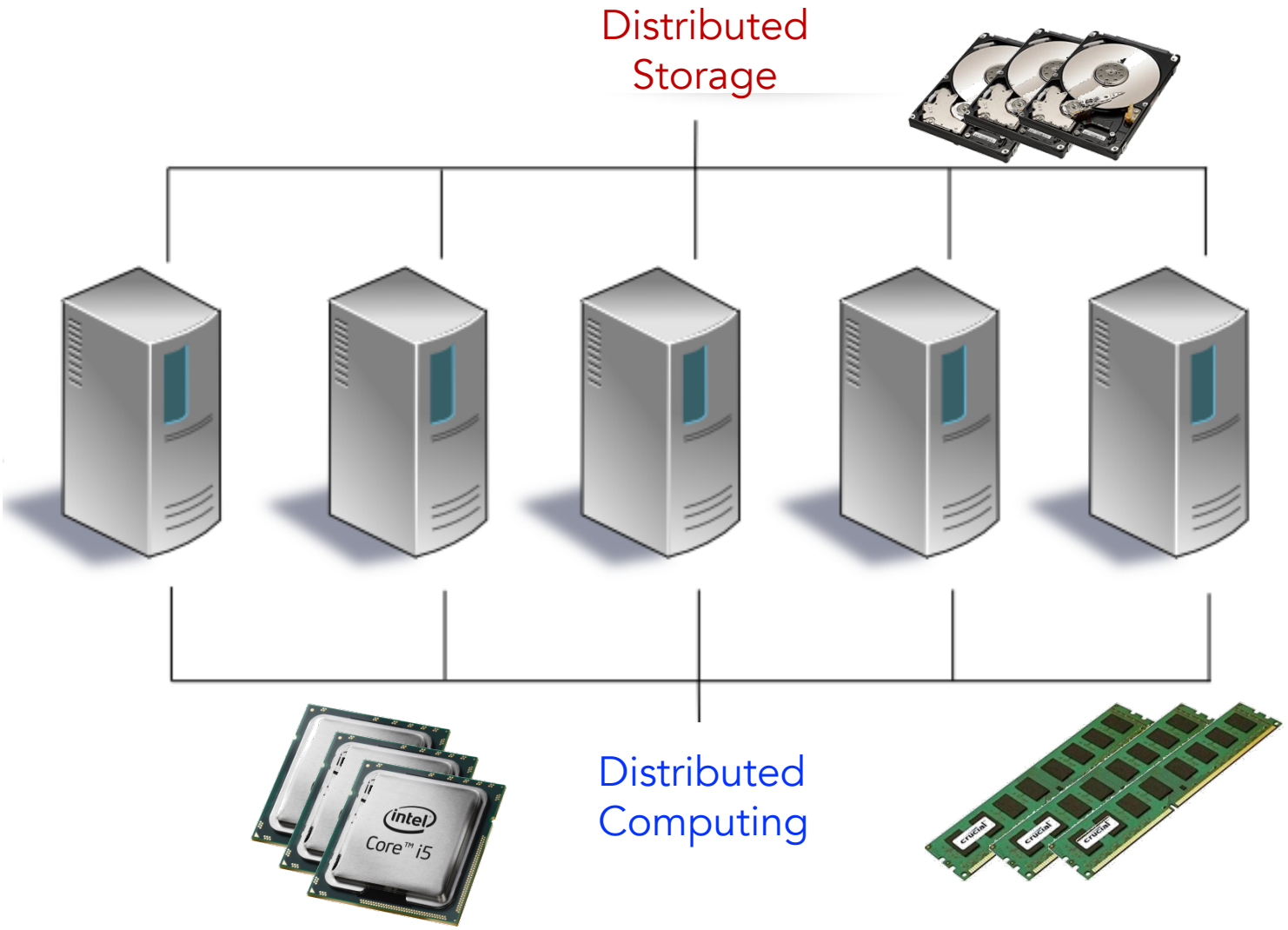
→ pyspark internal architecture

“ Architecture starts when you carefully put two bricks together. There it begins. ”

Ludwig Mies van der Rohe

- ✓ Definition
- ✓ History & Context
- ✓ Multi Language Support
- ✓ Distributed Computing Architecture
- Spark Components
- Spark Integration & Access Points
- Resilient Distributed Dataset
- Spark Session
- Lazy Evaluation
- Transformations & Actions
- Narrow & Wide Transformations
- RDD Lineage
- Anatomy of a Spark Application
- Directed Acyclic Graph

Key Slide !

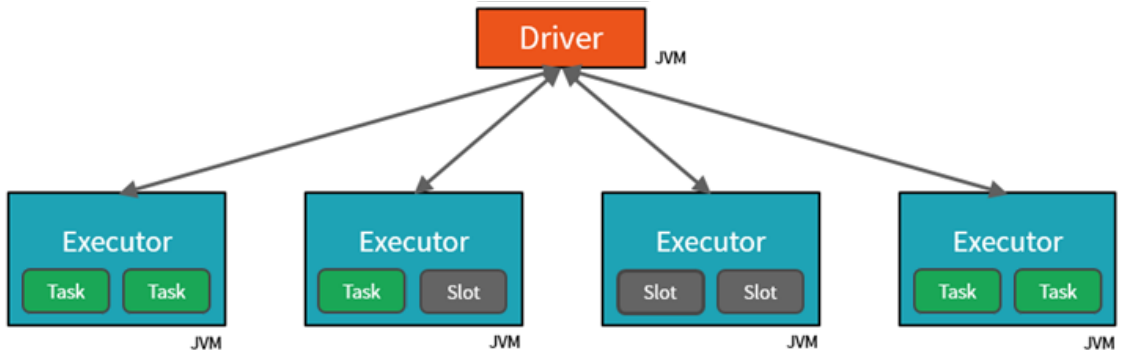


Compute where you store

Spark Architecture
(high level)

- The Driver is a program that runs on a node/client, and coordinates independent sets of processes on a cluster than run on executors

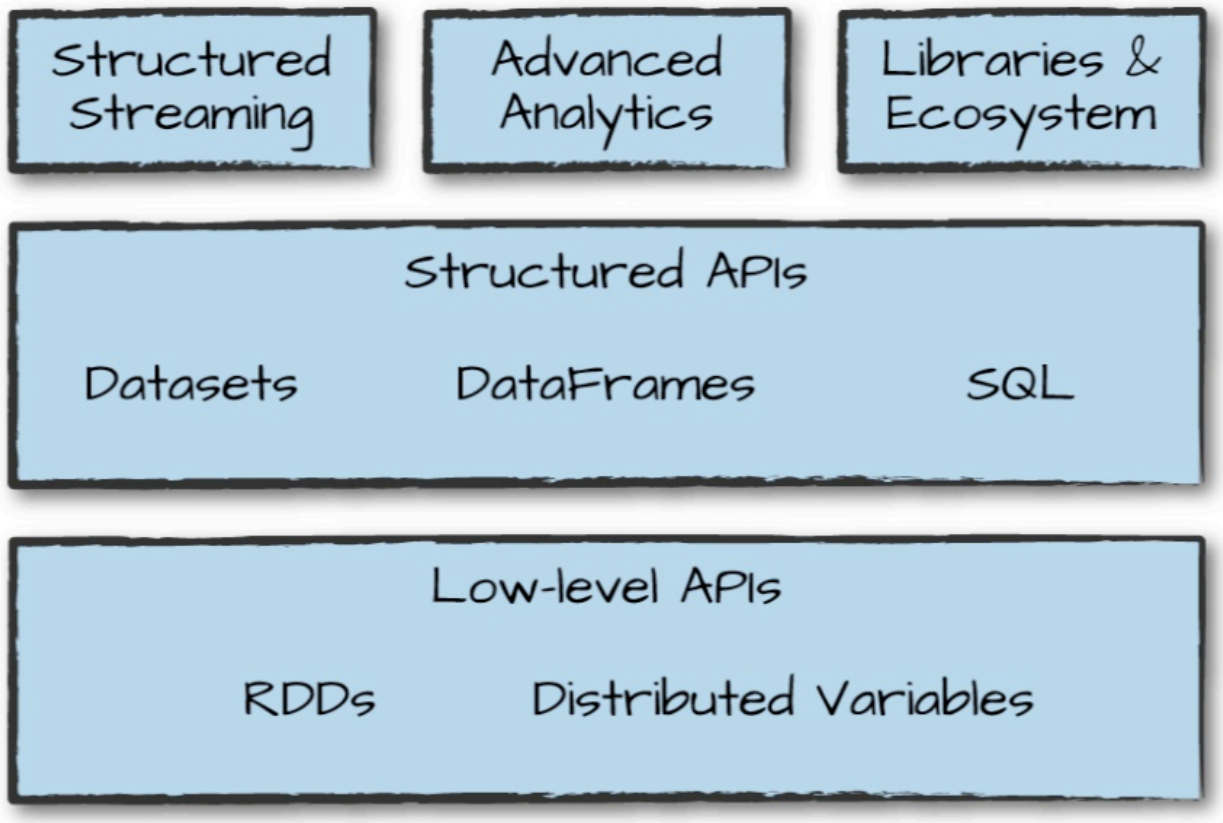
- Each Executor carries out the tasks that the driver assigns to it, and reports the state of the computation on that executor back to the driver node.



more on this later...
(Cluster Managers)

- ✓ Definition
- ✓ History & Context
- ✓ Multi Language Support
- ✓ Distributed Computing Architecture
- ✓ Spark Components
- Spark Integration & Access Points
- Resilient Distributed Dataset
- Spark Session
- Lazy Evaluation
- Transformations & Actions
- Narrow & Wide Transformations
- RDD Lineage
- Anatomy of a Spark Application
- Directed Acyclic Graph

SALK



- Handles Data at Rest (Dataframes)
- Handles Data in Motion (Streaming)

- ✓ Definition
- ✓ History & Context
- ✓ Multi Language Support
- ✓ Distributed Computing Architecture
- ✓ Spark Components
- ✓ Access Points
- Resilient Distributed Dataset
- Spark Session
- Lazy Evaluation
- Transformations & Actions
- Narrow & Wide Transformations
- RDD Lineage
- Anatomy of a Spark Application
- Directed Acyclic Graph

SALK

- Spark engine can be accessed from a variety of clients
  - Spark Console
    - spark-shell (scala)
    - pyspak (python)
  - spark-submit Application (scala / python code file)
  - Notebooks (Jupyter, Zeppelin)
  - Cloud clients (AWS, Azure, Databricks)
  - ...

- Whatever the nature of the client, they are all pointing to a specific installation of spark on a cluster, and thus sharing its resources

more on this later...

(Resource Management)

SALK

databricks

amazon
EMR

“ It's always the small pieces that make the big picture.”

- ✓ Definition
- ✓ History & Context
- ✓ Multi Language Support
- ✓ Distributed Computing Architecture
- ✓ Spark Compionents
- ✓ Access Points
- ✓ Resilient Distributed Dataset
- Spark Session
- Lazy Evaluation
- Transformations & Actions
- Narrow & Wide Transformations
- RDD Lineage
- Anatomy of a Spark Application
- Directed Acyclic Graph

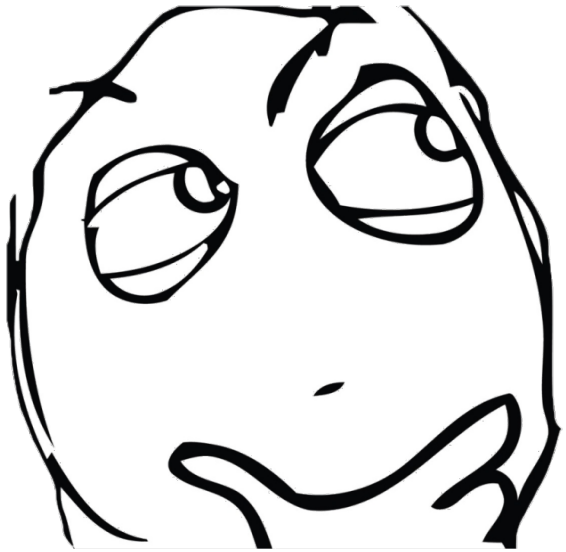
SALK

- RESILIENT
- DISTRIBUTED
- DATASET

APACHE Spark™

SALK

First…What is a DataSet ?



SALK

DATASET

"In mathematics, a **set** is a well-defined collection of distinct objects, considered as an object in its own right"

- When we think of dataset, especially in classic data science, we think of structured data (csv, database extract, etc)
- In Spark, virtually everything can be a dataset, as long as it can be represented in binary format

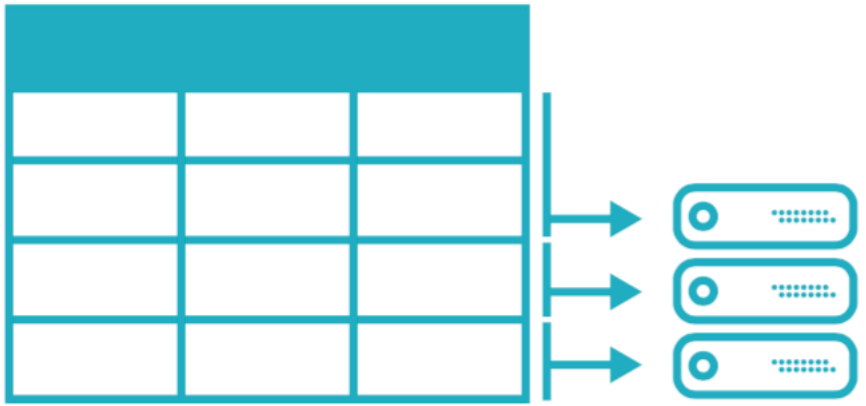
APACHE Spark™

SALK

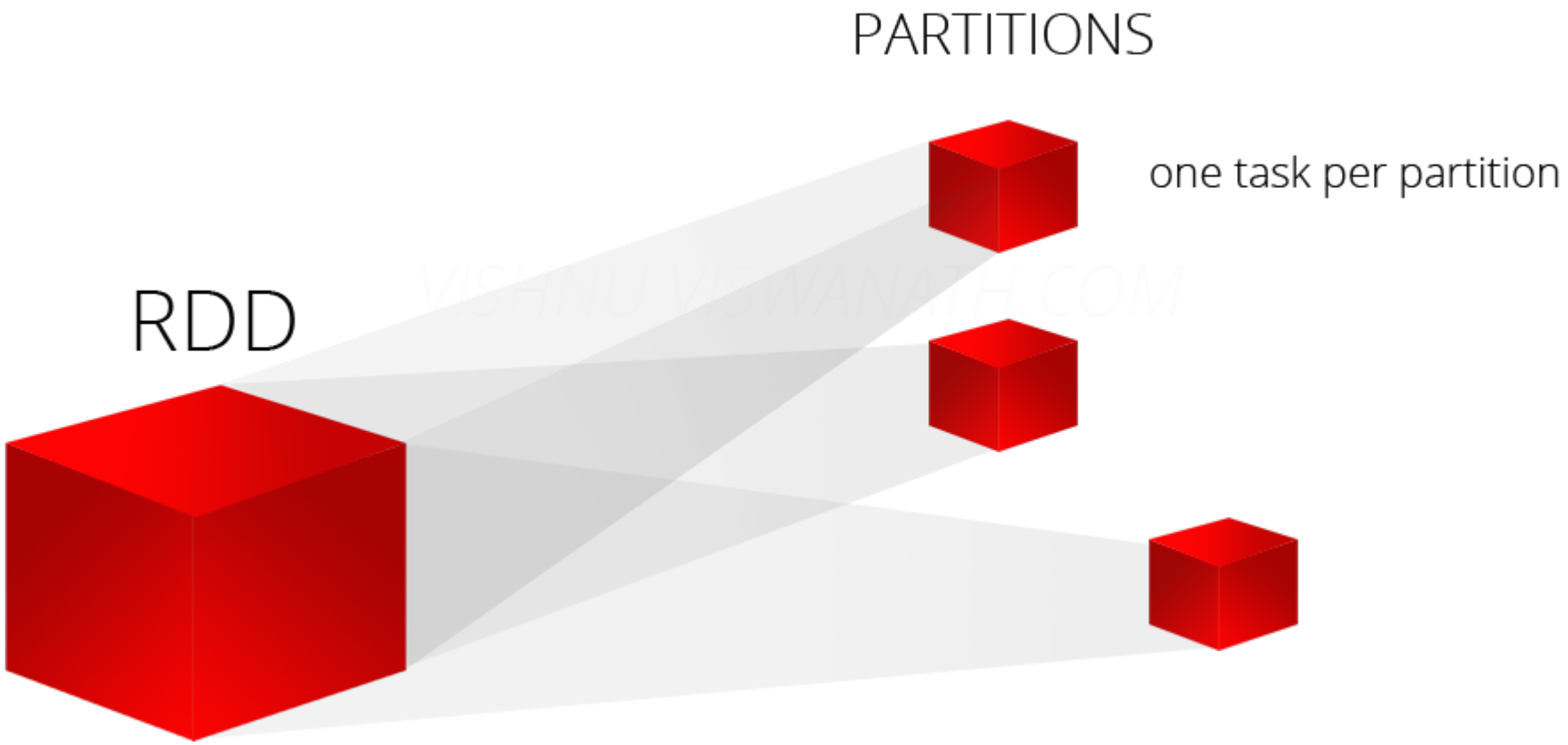
DISTRIBUTED

Spreadsheet on a single machine

Table or DataFrame partitioned across servers in a data center



- RDD is an **abstraction**
- It is an entity that holds "pointers" to where the data actually is, and the operations that need to be performed on it
- In consequence, an RDD does not strictly contain data itself



- RDD isn't *strictly* a collection of data, rather than a recipe for manipulating and transforming data.

SALK

IMMUTABLE

- RDDs are immutable : Once defined, an RDD does not change

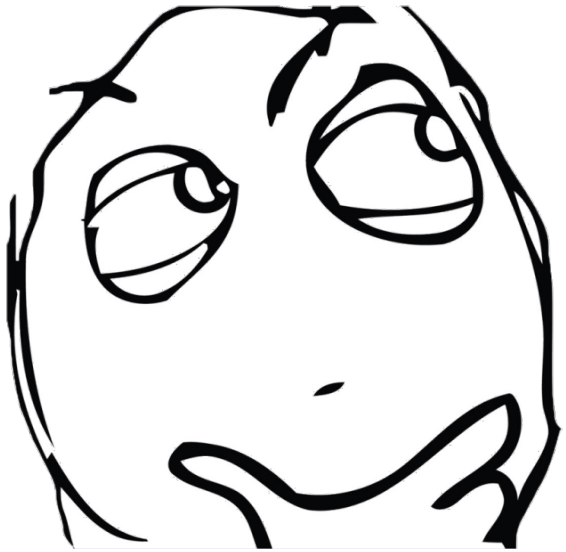
```
df = spark.createDataFrame(data = data, schema = schema)
sub_df = df.filter(col("state") == "NY")
```

- No self reassignement
- Any change results in a new RDD (this in an important detail)

- RDDs are a deterministic function of their input.
- That means an RDD can be recreated at any time
- This is what makes them *Resilient*

SALK

...Why make RDDs Immutable?



SALK

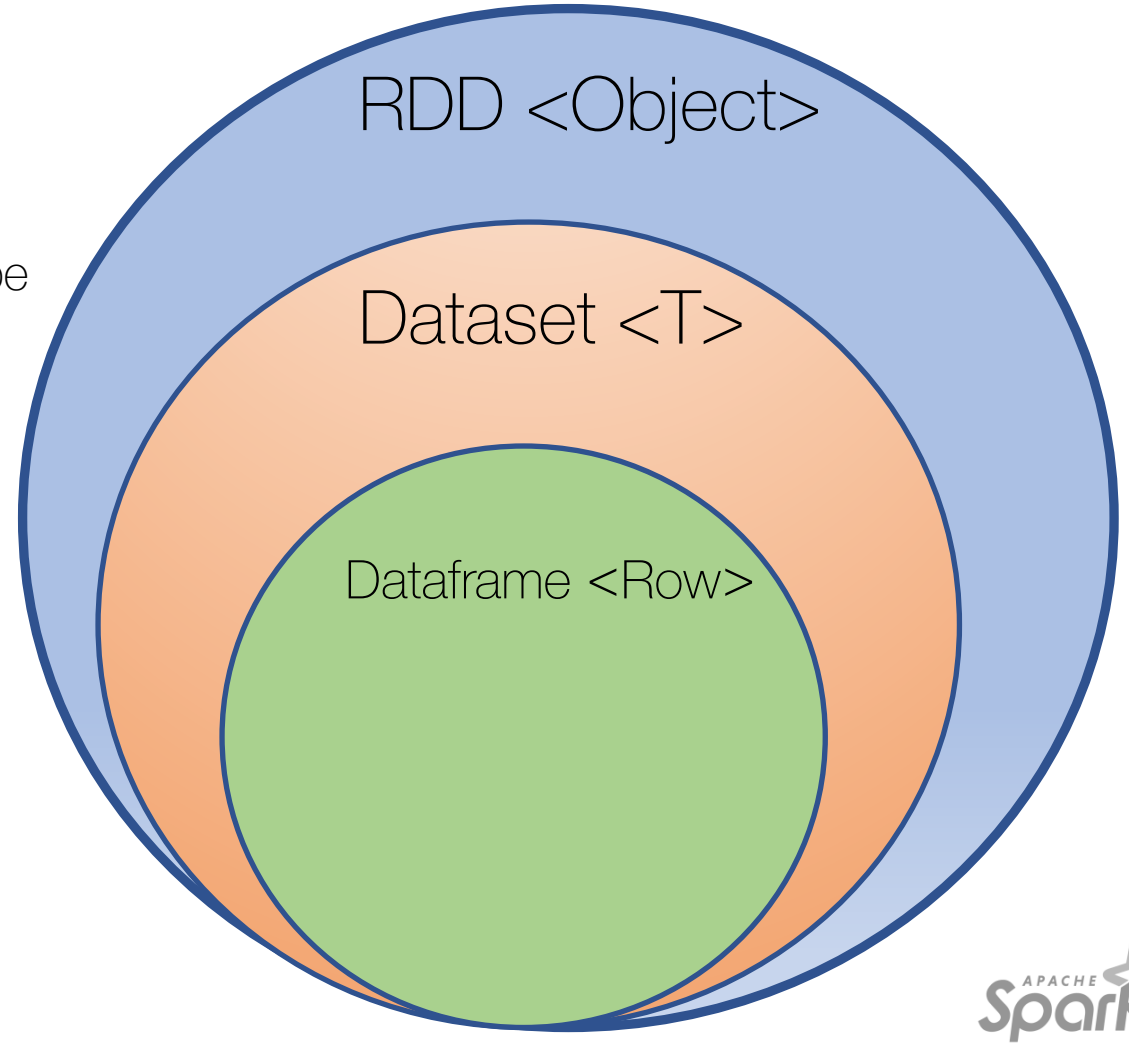
- Immutability rules out a big set of potential problems
  (especially due to updates from multiple threads at once)
- Immutable data
  - is safe to share across processes.
  - can easily live in memory as well as on disk.
- This makes it easy to move operations that hit disk to memory instead

- RDD design is sound, at cost of having to copy data rather than mutate it in place.
- That's a decent tradeoff to make:
  Gaining fault tolerance and correctness by design, with no developer effort
- Worth spending memory and CPU on

more on this later...
(RDD Lineage)

From RDD's to Dataframes

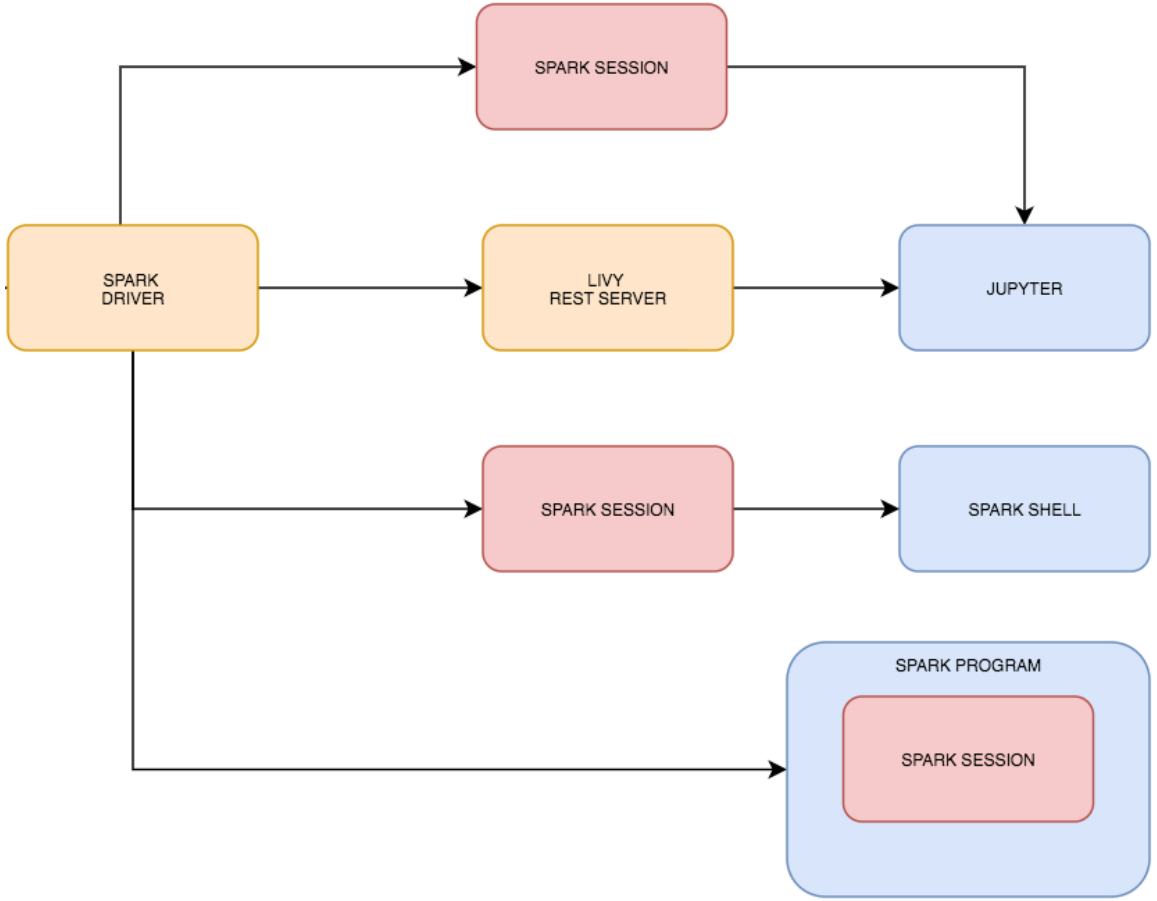
- The more the structures are specialized, the more structured they get
  - RDD is a collection of Objects
    - An object is the most generic structure that can be
  - a Dataset is a collection of Typed object
  - a Dataframe is a set or Row objects.
- Dataframes are the defacto preferred API
  - it is aims toward structured data
  - it accounts for most optimisations in the framework
  - Use dataframes, unless you are pushed out of chartered territories (unstructured data)



- ✓ Definition
- ✓ History & Context
- ✓ Multi Language Support
- ✓ Distributed Computing Architecture
- ✓ Spark Components
- ✓ Access Points
- ✓ Resilient Distributed Dataset
- ✓ Spark Session
- Lazy Evaluation
- Transformations & Actions
- Narrow & Wide Transformations
- RDD Lineage
- Anatomy of a Spark Application
- Directed Acyclic Graph

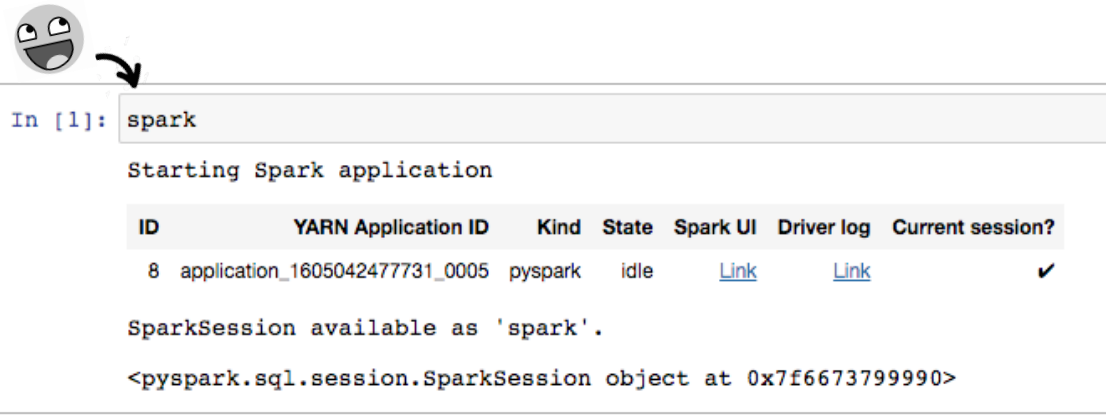
Spark Session

- SparkSession is the driver process controling your spark application
- When you launch a spark client (jupyter notebook, spark shell, spark application), you always instanciate a spark session
- There is a one-to-one correspondence between a SparkSession and a Spark Application.
- The variable is available as spark when you start the console.



- # Spark Session

- 'spark' is the first command you need to lauch in any pyspark jupyter notebook to launch a spark session
- when you're on the spark shell, however, the session is anlready instanciated for you



- There is no magic
  - There is always some kind of plumbing & integration involved
  - Some clients work more "natively" than others (spark shell)
  - You often need to tweak stuff to make things work smoothly (notebooks)
  - findspark is a cool python library that can help you locate the spark path in your host environement.
  - pyspark python library also installs the python API for spark, independently but it does need to be tied to a spark installation (although it comes with lightweight standalone binaries)
    caution : you need to make sure it fits the version of spark that is installed.

PRACTICE TIME
Workshop ½

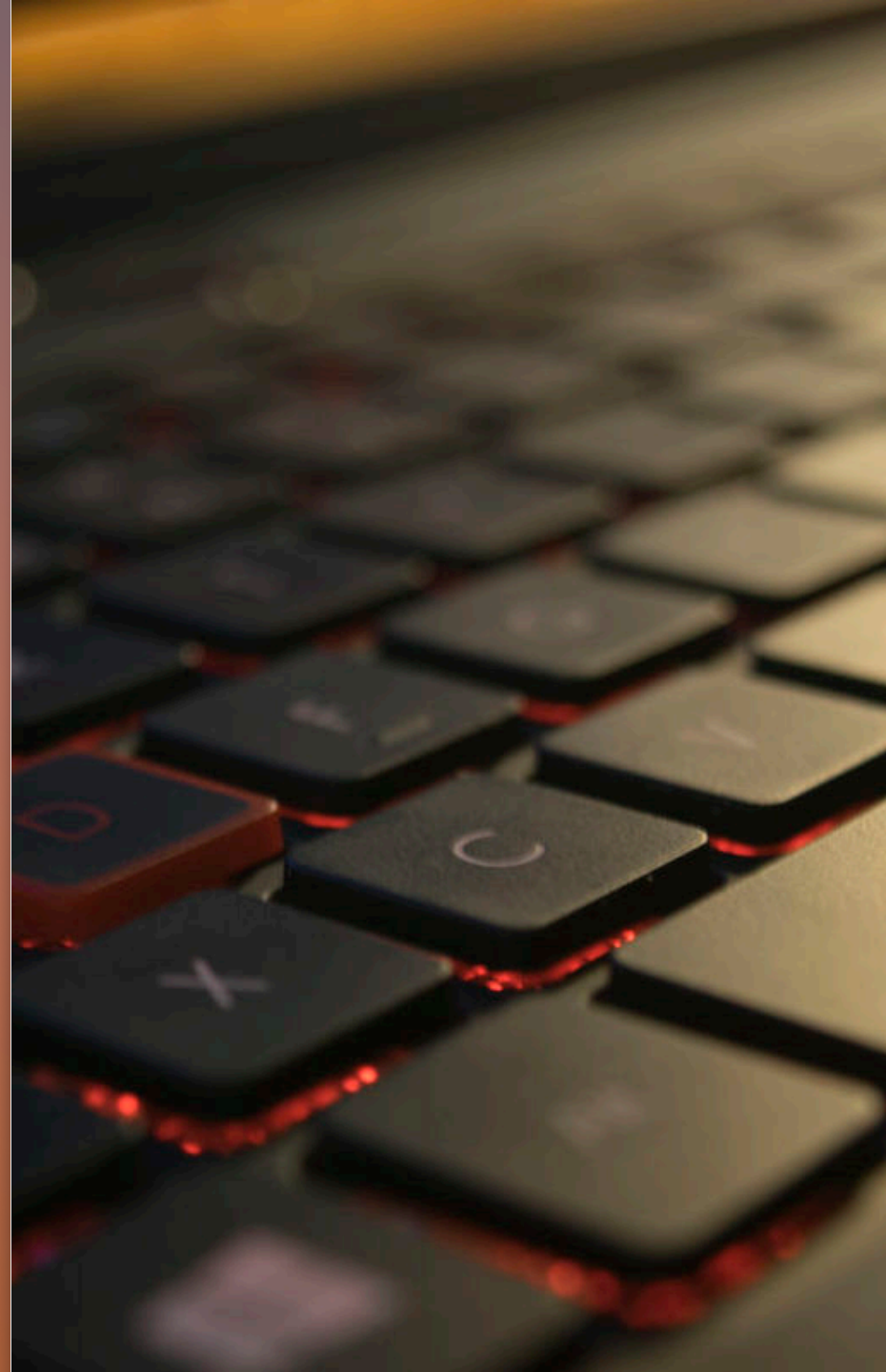
First Hands-on Lab : Onboarding & SSH'ing into EMR (Demo)

PRACTICE TIME
Workshop ½

PySpark Shell Exercices (warmup)

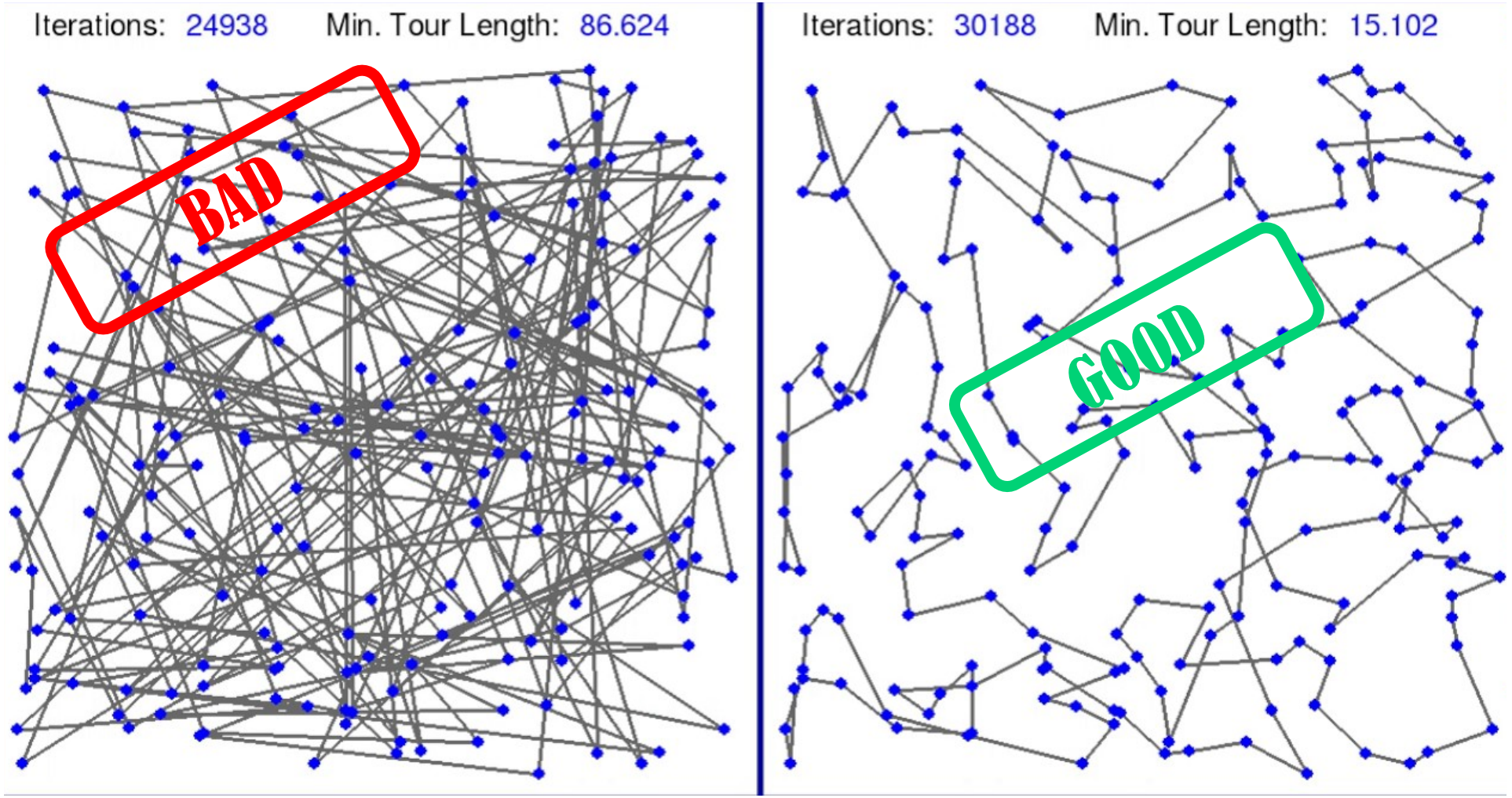
https://github.com/mehdi-lamrani/spark-training/spark-c-training/blob/master/day%201/part%201:2/shell/exercices/02-pyspark-shell.md

THEORY 2/2



- ✓ Definition
- ✓ History & Context
- ✓ Multi Language Support
- ✓ Distributed Computing Architecture
- ✓ Spark Components
- ✓ Access Points
- ✓ Resilient Distributed Dataset
- ✓ Spark Session
- ✓ Lazy Evaluation
- Transformations & Actions
- Narrow & Wide Transformations
- RDD Lineage
- Anatomy of a Spark Application
- Directed Acyclic Graph

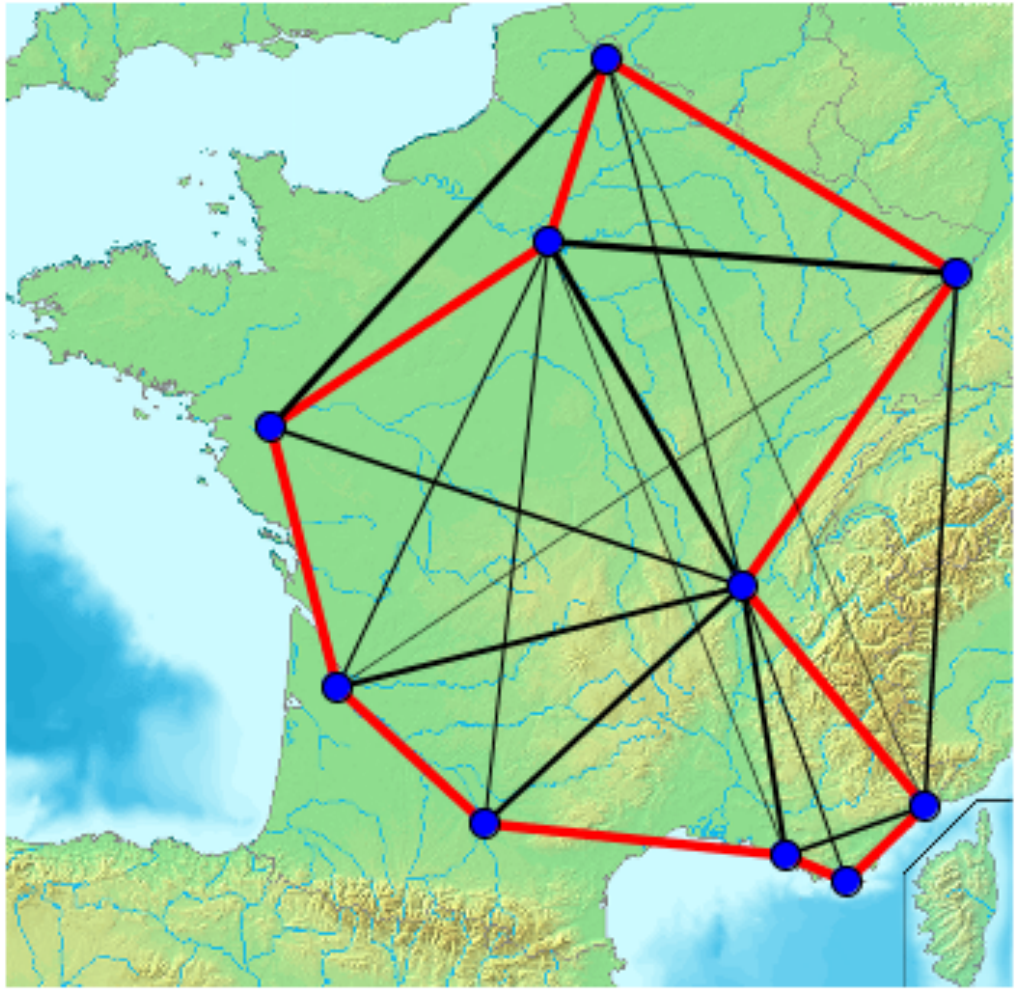
LAZY EVALUATION (Intuition)



SALK

LAZY EVALUATION
Intuition

- Optimization Heuristic
- Traveling Salesman
  - Cities are Storage
  - Goods are Data
  - Roads are Network
- Objective :
  - Minimise workload
  - Eliminate unnecessary effort
  - Order Operations in an effective way
  - Minimal Entropy



SALK

LAZY EVALUATION

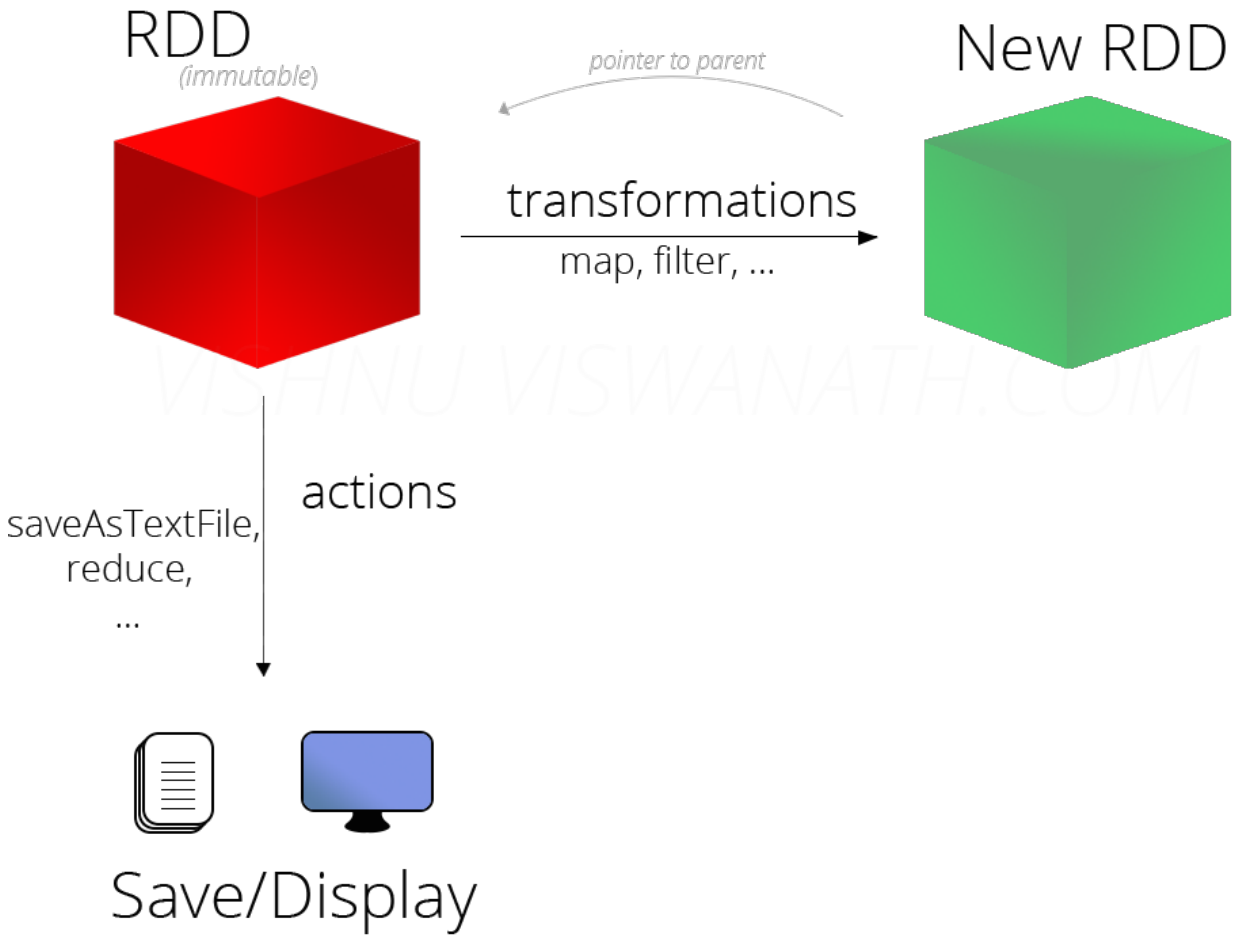
- Spark will wait until the very last moment to execute the graph of computation instructions.
- Instead of modifying the data immediately when you express some operation, you build up a plan of transformations to apply to your source data.
- By waiting until the last minute to execute the code : Spark compiles this plan from your raw DataFrame transformations to an optimized plan that will run as efficiently as possible across the cluster.
- This provides immense benefits because Spark can optimize the entire data flow from end to end.

- An example of this is something called predicate pushdown on DataFrames.
- If we build a large Spark job but specify a filter at the end that only requires us to fetch one row from our source data, the most efficient way to execute this is to access the single record that we need.
- Spark will actually optimize this for us by pushing the filter down automatically.

- ✓ Definition
- ✓ History & Context
- ✓ Multi Language Support
- ✓ Distributed Computing Architecture
- ✓ Spark Components
- ✓ Access Points
- ✓ Resilient Distributed Dataset
- ✓ Spark Session
- ✓ Lazy Evaluation
- ✓ Transformations & Actions
- Narrow & Wide Transformations
- RDD Lineage
- Anatomy of a Spark Application
- Directed Acyclic Graph

- # Transformations
  - Operations that apply modifications to dataframes are called transformations.

- # Actions
  - An action instructs Spark to compute a result from a series of transformations on RDDs/DFs.
  - The simplest action is count, which gives us the total number of records in the DataFrame.



SALK

- Transformations

Transformation	Meaning
map(*func*)	Return a new distributed dataset formed by passing each element of the source through a function *func*.
filter(*func*)	Return a new dataset formed by selecting those elements of the source on which *func* returns true.
flatMap(*func*)	Similar to map, but each input item can be mapped to 0 or more output items (so *func* should return a Seq rather than a single item).
mapPartitions(*func*)	Similar to map, but runs separately on each partition (block) of the RDD, so *func* must be of type Iterator<T> => Iterator<U> when running on an RDD of type T.
mapPartitionsWithIndex(*func*)	Similar to mapPartitions, but also provides *func* with an integer value representing the index of the partition, so *func* must be of type (Int, Iterator<T>) => Iterator<U> when running on an RDD of type T.
sample(*withReplacement*, *fraction*, *seed*)	Sample a fraction *fraction* of the data, with or without replacement, using a given random number generator seed.
union(*otherDataset*)	Return a new dataset that contains the union of the elements in the source dataset and the argument.
intersection(*otherDataset*)	Return a new RDD that contains the intersection of elements in the source dataset and the argument.
distinct([*numPartitions*]))	Return a new dataset that contains the distinct elements of the source dataset.
groupByKey([*numPartitions*])	When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable<V>) pairs. **Note:** If you are grouping in order to perform an aggregation (such as a sum or average) over each key, using `reduceByKey` or `aggregateByKey` will yield much better performance. **Note:** By default, the level of parallelism in the output depends on the number of partitions of the parent RDD. You can pass an optional `numPartitions` argument to set a different number of tasks.
reduceByKey(*func*, [*numPartitions*])	When called on a dataset of (K, V) pairs, returns a dataset of (K, V) pairs where the values for each key are aggregated using the given reduce function *func*, which must be of type (V,V) => V. Like in `groupByKey`, the number of reduce tasks is configurable through an optional second argument.

SALK

- Transformations

aggregateByKey(*zeroValue*)(*seqOp*, *combOp*, [*numPartitions*])	When called on a dataset of (K, V) pairs, returns a dataset of (K, U) pairs where the values for each key are aggregated using the given combine functions and a neutral "zero" value. Allows an aggregated value type that is different than the input value type, while avoiding unnecessary allocations. Like in `groupByKey`, the number of reduce tasks is configurable through an optional second argument.
sortByKey([*ascending*], [*numPartitions*])	When called on a dataset of (K, V) pairs where K implements Ordered, returns a dataset of (K, V) pairs sorted by keys in ascending or descending order, as specified in the boolean `ascending` argument.
join(*otherDataset*, [*numPartitions*])	When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (V, W)) pairs with all pairs of elements for each key. Outer joins are supported through `leftOuterJoin`, `rightOuterJoin`, and `fullOuterJoin`.
cogroup(*otherDataset*, [*numPartitions*])	When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (Iterable<V>, Iterable<W>)) tuples. This operation is also called `groupWith`.
cartesian(*otherDataset*)	When called on datasets of types T and U, returns a dataset of (T, U) pairs (all pairs of elements).
pipe(*command*, *[envVars]*)	Pipe each partition of the RDD through a shell command, e.g. a Perl or bash script. RDD elements are written to the process's stdin and lines output to its stdout are returned as an RDD of strings.
coalesce(*numPartitions*)	Decrease the number of partitions in the RDD to numPartitions. Useful for running operations more efficiently after filtering down a large dataset.
repartition(*numPartitions*)	Reshuffle the data in the RDD randomly to create either more or fewer partitions and balance it across them. This always shuffles all data over the network.
repartitionAndSortWithinPartitions(*partitioner*)	Repartition the RDD according to the given partitioner and, within each resulting partition, sort records by their keys. This is more efficient than calling `repartition` and then sorting within each partition because it can push the sorting down into the shuffle machinery.

SALK

- Actions

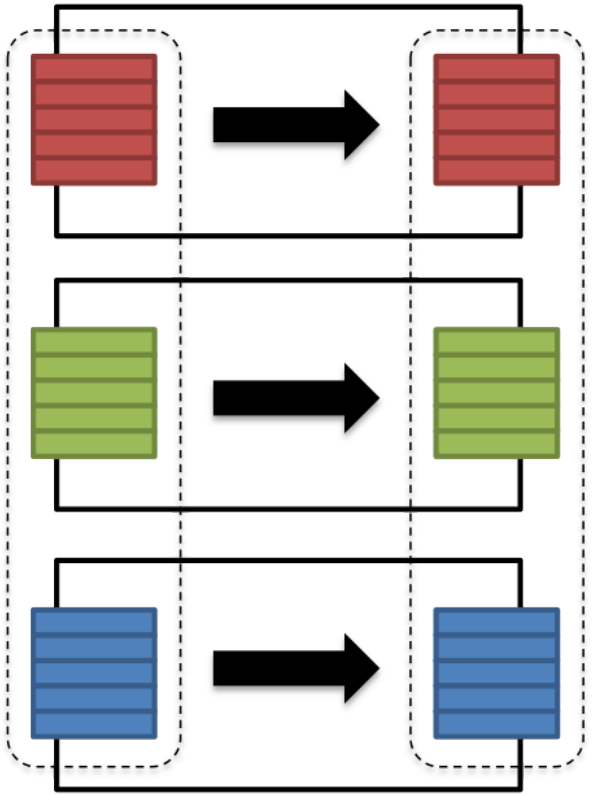
Action	Meaning
reduce(*func*)	Aggregate the elements of the dataset using a function *func* (which takes two arguments and returns one). The function should be commutative and associative so that it can be computed correctly in parallel.
collect()	Return all the elements of the dataset as an array at the driver program. This is usually useful after a filter or other operation that returns a sufficiently small subset of the data.
count()	Return the number of elements in the dataset.
first()	Return the first element of the dataset (similar to take(1)).
take(*n*)	Return an array with the first *n* elements of the dataset.
takeSample(*withReplacement*, *num*, [*seed*])	Return an array with a random sample of *num* elements of the dataset, with or without replacement, optionally pre-specifying a random number generator seed.
takeOrdered(*n*, *[ordering]*)	Return the first *n* elements of the RDD using either their natural order or a custom comparator.
saveAsTextFile(*path*)	Write the elements of the dataset as a text file (or set of text files) in a given directory in the local filesystem, HDFS or any other Hadoop-supported file system. Spark will call toString on each element to convert it to a line of text in the file.
saveAsSequenceFile(*path*) (Java and Scala)	Write the elements of the dataset as a Hadoop SequenceFile in a given path in the local filesystem, HDFS or any other Hadoop-supported file system. This is available on RDDs of key-value pairs that implement Hadoop's Writable interface. In Scala, it is also available on types that are implicitly convertible to Writable (Spark includes conversions for basic types like Int, Double, String, etc).
saveAsObjectFile(*path*) (Java and Scala)	Write the elements of the dataset in a simple format using Java serialization, which can then be loaded using `SparkContext.objectFile()`.
countByKey()	Only available on RDDs of type (K, V). Returns a hashmap of (K, Int) pairs with the count of each key.
foreach(*func*)	Run a function *func* on each element of the dataset. This is usually done for side effects such as updating an Accumulator or interacting with external storage systems. **Note**: modifying variables other than Accumulators outside of the `foreach()` may result in undefined behavior. See Understanding closures for more details.

- ✓ Definition
- ✓ History & Context
- ✓ Multi Language Support
- ✓ Distributed Computing Architecture
- ✓ Spark Compionents
- ✓ Access Points
- ✓ Resilient Distributed Dataset
- ✓ Spark Session
- ✓ Lazy Evaluation
- ✓ Transformations & Actions
- ✓ Narrow & Wide Transformations
- • RDD Lineage
- • Anatomy of a Spark Application
- • Directed Acyclic Graph

Narrow Transformations

- Transformations for which each input partition will contribute to only one output partition

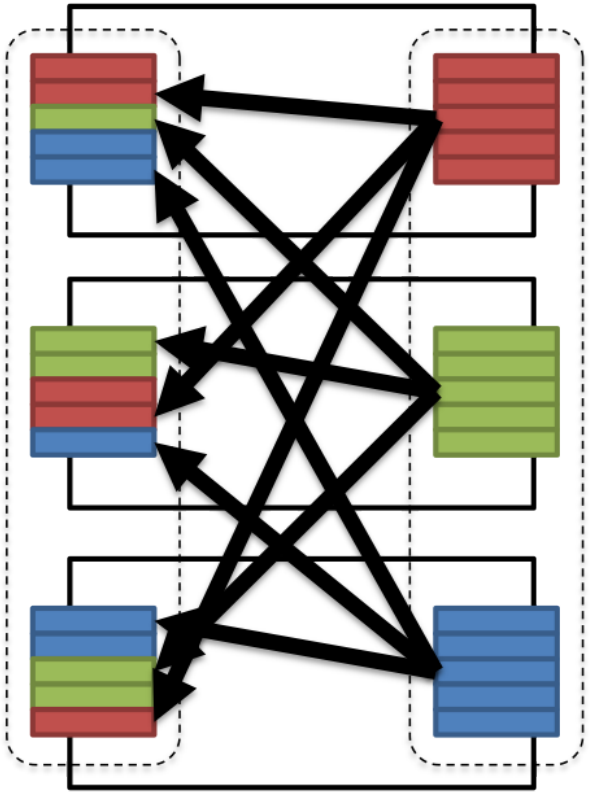
- Functions such as `map(), filter(), union()` are some examples of narrow transformations.



Wide Transformations

- Transformations having input partitions contributing to many output partitions.
- Often referred to as a shuffle, whereby Spark will exchange partitions across the cluster.

- Functions such as
  `groupByKey(), aggregateByKey(), aggregate()`
  `join(), repartition()`
  are some examples of wide transformations.



- ✓ Definition
- ✓ History & Context
- ✓ Multi Language Support
- ✓ Distributed Computing Architecture
- ✓ Spark Compionents
- ✓ Access Points
- ✓ Resilient Distributed Dataset
- ✓ Spark Session
- ✓ Lazy Evaluation
- ✓ Transformations & Actions
- ✓ Narrow & Wide Transformations
- ✓ RDD Lineage
- • Anatomy of a Spark Application
- • Directed Acyclic Graph

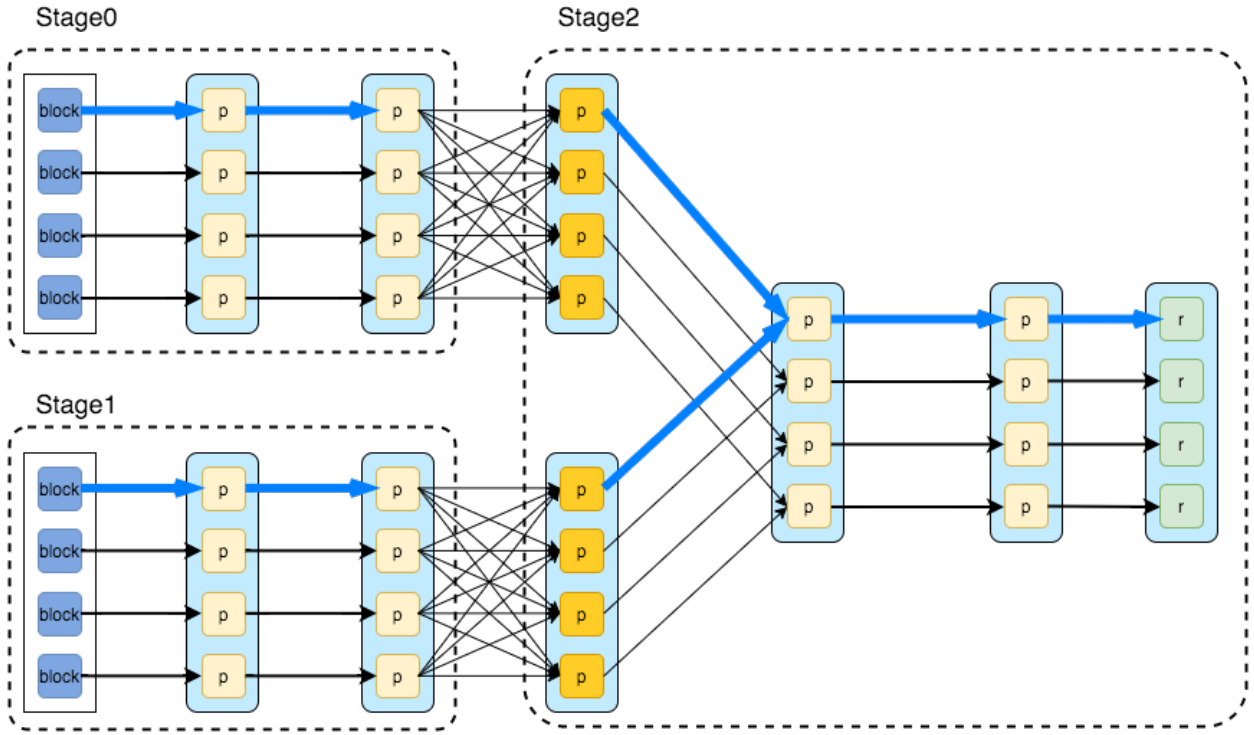
SALK

- *“ Nothing is created, everything is transformed… especially if it's an RDD ”*

Antoine Lavoisier in 1774

RDD Lineage

- When a new RDD has been created from an existing RDD, that new RDD contains a pointer to the parent RDD.
- All the dependencies between the RDDs are logged in a graph *(rather than the actual data)*
- This graph is called the lineage graph



This lineage graph is a big deal in terms of design, as it permits many optimisations and other feature as distribution and resilience.
Immutablity of RDD's is what make this lineage possible.

There is *always* more to Spark than meets the eye

- ✓ Definition
- ✓ History & Context
- ✓ Multi Language Support
- ✓ Distributed Computing Architecture
- ✓ Spark Compionents
- ✓ Access Points
- ✓ Resilient Distributed Dataset
- ✓ Spark Session
- ✓ Lazy Evaluation
- ✓ Transformations & Actions
- ✓ Narrow & Wide Transformations
- ✓ RDD Lineage
- ✓ Anatomy of a Spark Application
- • Directed Acyclic Graph

JOB

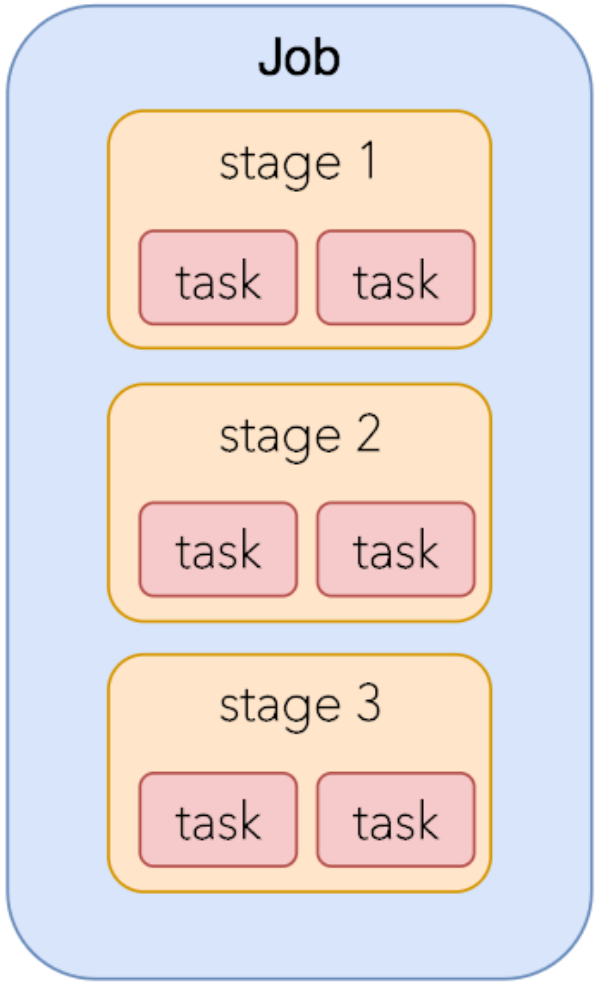
- A parallel computation consisting of multiple tasks that gets spawned in response to a Spark action such as `save()`, `collect()`, or `write()`

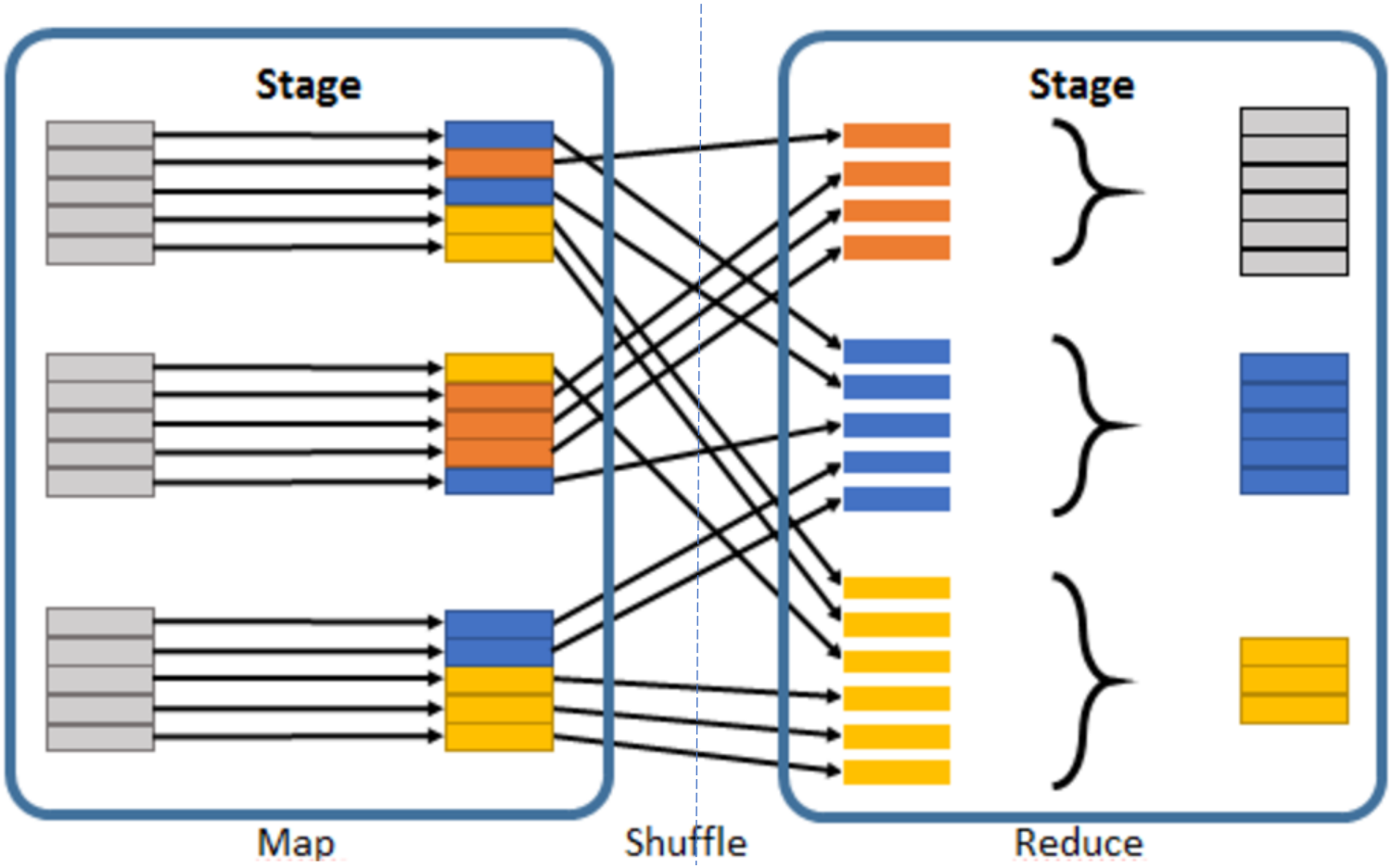
STAGE

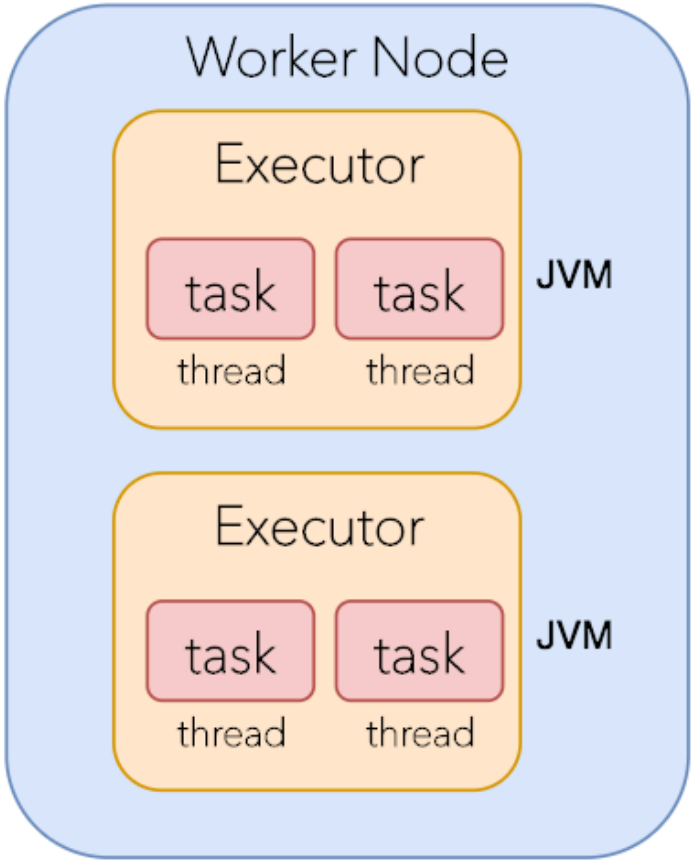
- Each job gets divided into smaller sets of tasks, called stages
- A Stage is a sequence of Tasks that can all be run together, in parallel, without a shuffle.
- *Example: reading a file, then runnning .map and .filter can all be done without a shuffle, so it can fit in a single stage.(narrow transformation)*
- *Each stage can be : shuffle map or result type.*
  shuffle map represents stages whose results are the inputs for next stages.
  result stage represents stages whose results are sent to the driver (= results of Spark actions).

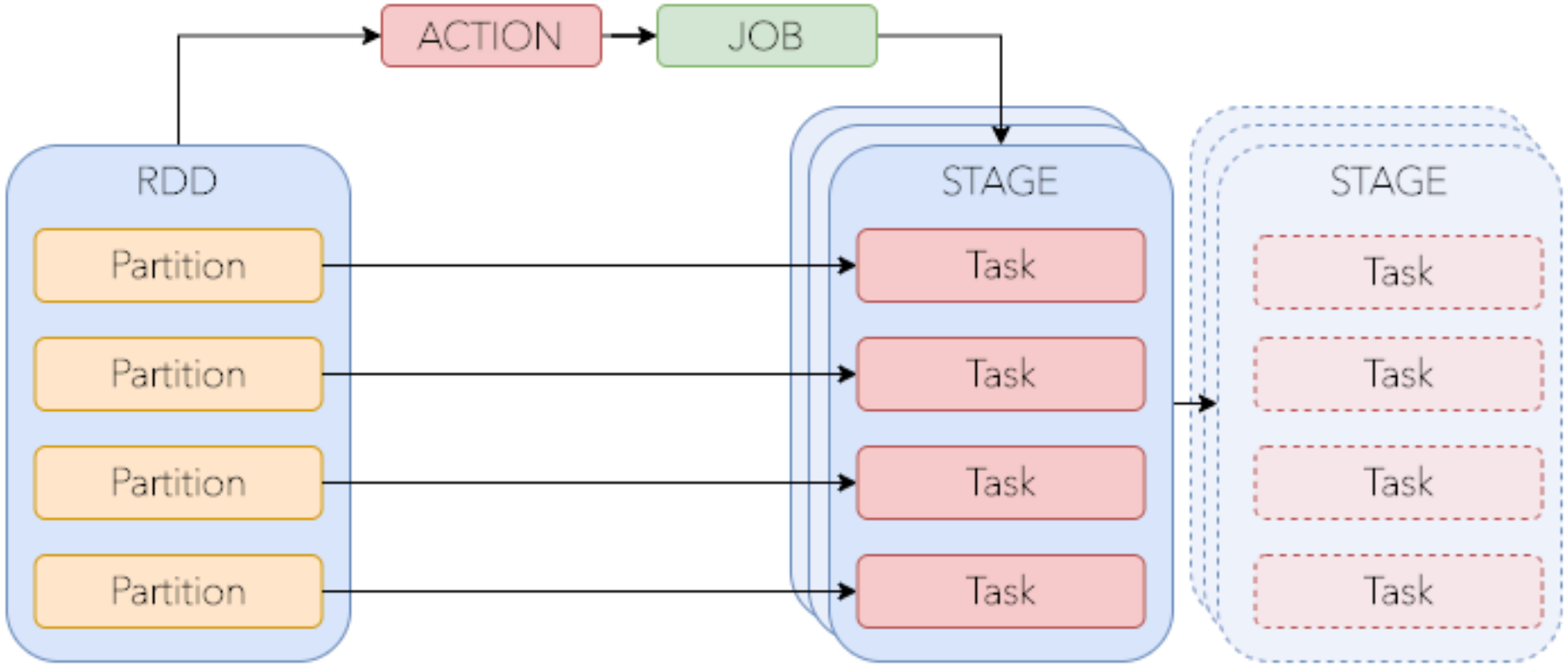
TASK

- A task the smallest unit of execution used to compute a new RDD.
- A Task is a single operation (.map or .filter) applied to a single Partition.


Job
stage 1
task
task
stage 2
task
task
stage 3
task
task


Stage
Stage
Map
Shuffle
Reduce
Stage
Boundary


Worker Node
Executor
task
task
thread
thread
JVM
Executor
task
task
thread
thread
JVM

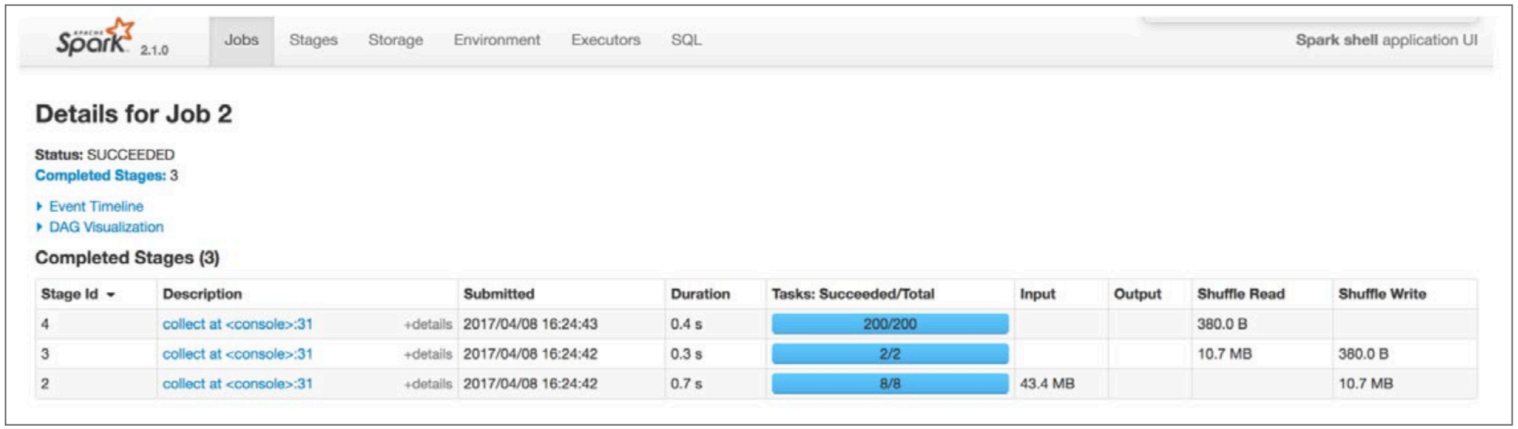


- Good Read 1 : https://stackoverflow.com/questions/41340612/does-stages-in-an-application-run-parallel-in-spark

SALK

SPARK WEB UI

- Execution details (jobs tasks stages) can be seen in the spark web UI during execution, and from the history server after that.



APACHE Spark

Internals of Task *(Advanced)*

- A task is a command sent by the driver to executor in serialized form.
- More precisely, the command is represented as a closure, with all methods and variables needed to make computation.
- Variables are only the copies of objects declared in driver's program
- They are not shared among executors
  (i.e. each executor will operate on different objects).

Out of scope
(but interesting…)

A closure is a persistent scope which holds on to local variables even after the code execution has moved out of that block

A closure is the set of variables and methods which must be visible for the executor to perform its computations on the RDD. This closure is serialized and sent to each executor.

- Good Read 1 : https://stackoverflow.com/questions/36636/what-is-a-closure
- Good Read 2 : https://medium.com/@mycupoftea00/understanding-closure-in-spark-af6f280eebf9

- ✓ Definition
- ✓ History & Context
- ✓ Multi Language Support
- ✓ Distributed Computing Architecture
- ✓ Spark Compionents
- ✓ Access Points
- ✓ Resilient Distributed Dataset
- ✓ Spark Session
- ✓ Lazy Evaluation
- ✓ Transformations & Actions
- ✓ Narrow & Wide Transformations
- ✓ RDD Lineage
- ✓ Anatomy of a Spark Application
- • Directed Acyclic Graph

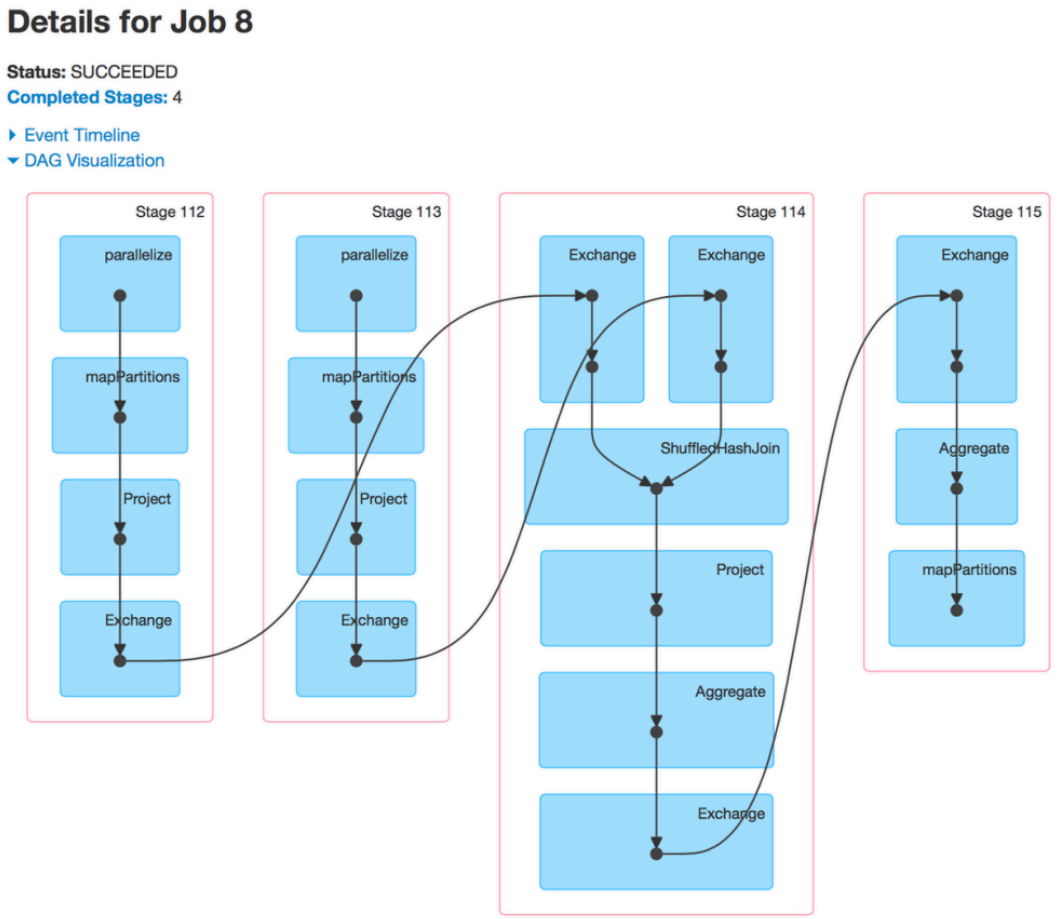
DAG (Directed Acyclic Graph)

- Graph : the structure composed of nodes. Some of them can be connected together through edges.
- Acyclic : the graph doesn't have cycles.
  - No node is visited more than once.
  - The traversal doesn't back to the previous node.
- Directed : Relationships between nodes have a single direction (flowing down)
- Spark's DAG consists on RDDs (nodes) and calculations (edges).
- Basically, it is a logical representation of all operations that need to be performed after an action / job is triggered.

SALK

DAG Representation

- Spark DAG is represented with a set of Vertices and Edges
- Vertices represent the RDDs
- Edges represent the Operation to be applied on RDD.
- When an action is called on Spark RDD at a high level, DAG is created and is submitted for execution



- Stage Boundary = Need for Shuffle

SALK

• SUMMING UP

- Spark is a distributed programming model in which the user specifies transformations.
- Multiple transformations build up a directed acyclic graph of instructions.
- An action begins the process of executing that graph of instructions, as a single job, by breaking it down into stages and tasks to execute across the cluster.
- The logical structures that we manipulate with transformations and actions are DataFrames.
- To create a new DataFrame, you call a transformation.
- To start computation or convert to native language types, you call an action.

Excerpt From: Bill Chambers / Matei Zaharia : “Spark Definitive Guide"

DEMO TIME

Using Jupyter / Zeppelin Notebooks (DEMO)

PRACTICE TIME
Workshop 2/2

Introduction to Dataframe API – Schema management

https://github.com/mehdi-lamrani/spark-training/spark-c-training/tree/master/day%201/part%202:2/notebooks

END OF SESSION 1