

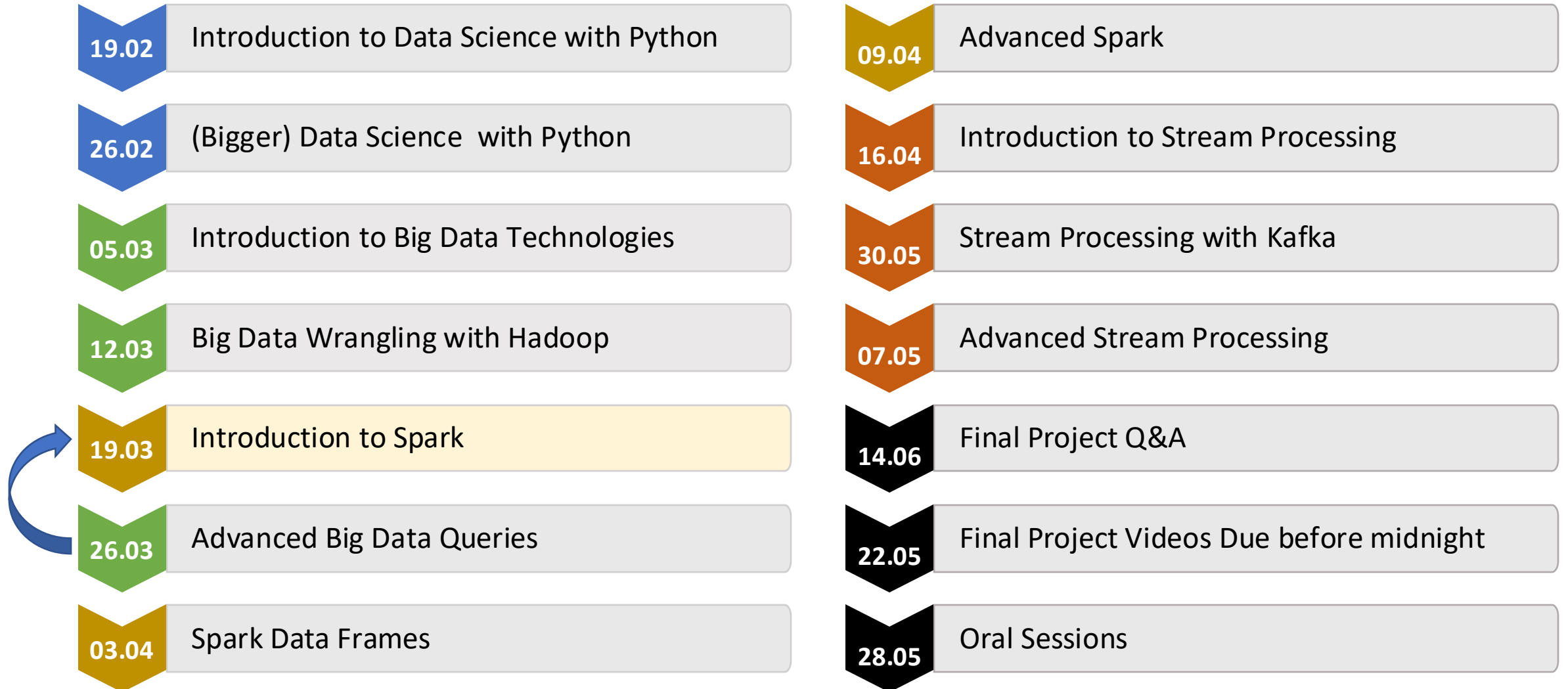
THE DATA SCIENCE LAB

Introduction to Spark + RDDs

COM 490 – Module 3a

Week 5

Agenda 2025 – Module 3a



Announcements

- Homework #1 due today
- Homework #2 out
- Module 3a today! → module 2c next week

Today's Agenda

- Introduction to Spark
- Spark architecture overview
- RDDs
- Getting started with Spark
- Introduction to this week's exercises

Introduction to Spark

Getting started with



What is Spark?

Yet another- distributed computation framework

Apache Spark™ is a unified analytics engine for large-scale data processing.

KEY FEATURES

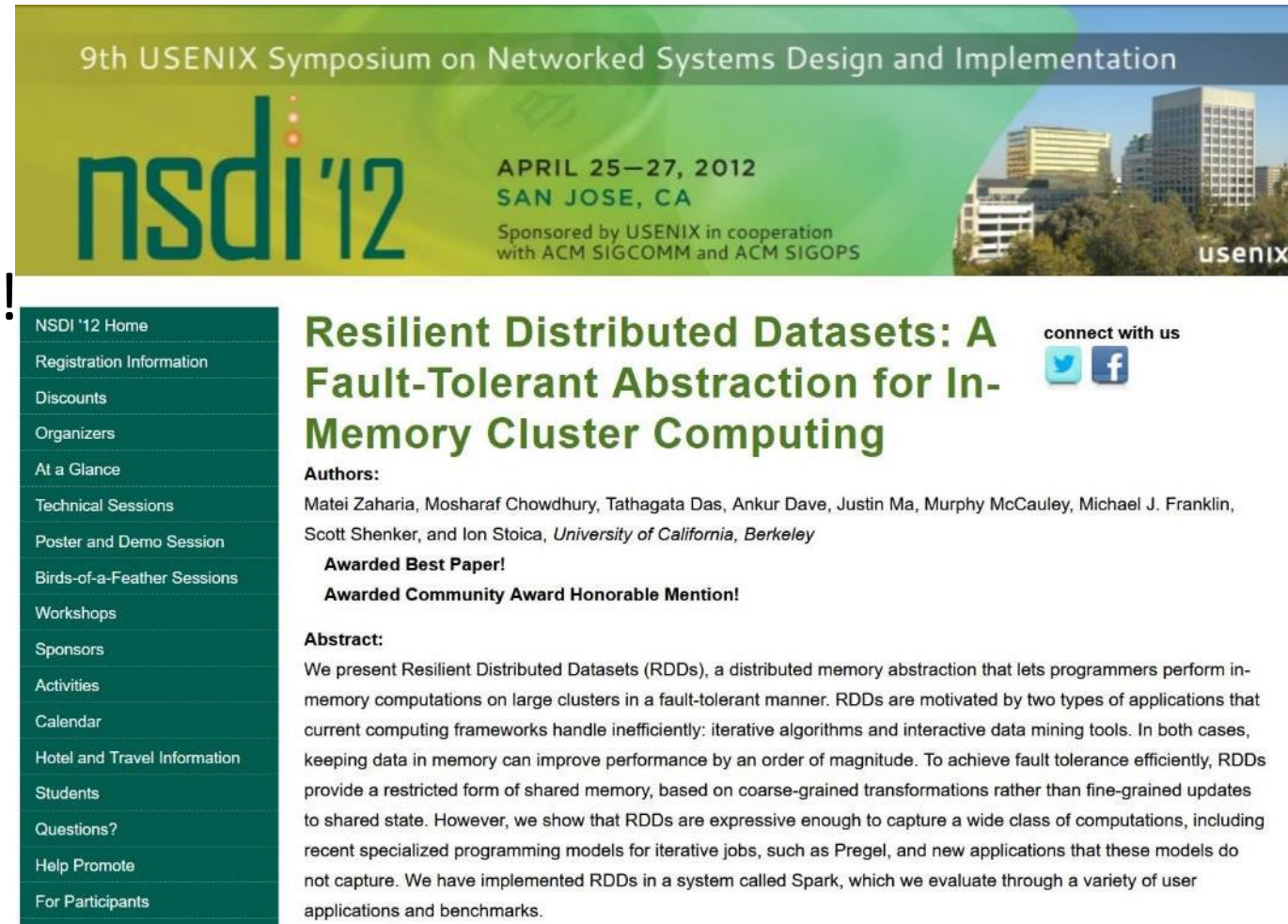
- Interactive data exploration
- In-memory data
- Fault-tolerance, parallelism

USED BY

- Internet applications (recommendation engines, usage analysis)
- Classic big data use cases e.g., text analysis
- Some academia, notably neuroscience

A bit of history

- MapReduce OSDI 2004
- Hadoop 2006, open source MR
- Many applications & frameworks!
- Mesos NSDI 2011
- Spark NSDI 2012
- Yarn SOCC 2013



9th USENIX Symposium on Networked Systems Design and Implementation



nsdi'12

APRIL 25—27, 2012
SAN JOSE, CA

Sponsored by USENIX in cooperation with ACM SIGCOMM and ACM SIGOPS

usenix

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

connect with us  

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

Awarded Best Paper!
Awarded Community Award Honorable Mention!

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 inefficiently: iterative algorithms and interactive data mining tools. In both cases, keeping data in memory can improve performance by an order of magnitude. To achieve fault tolerance efficiently, RDDs provide a restricted form of shared memory, based on coarse-grained transformations rather than fine-grained updates to shared state. However, we show that RDDs are expressive enough to capture a wide class of computations, including recent specialized programming models for iterative jobs, such as Pregel, and new applications that these models do not capture. We have implemented RDDs in a system called Spark, which we evaluate through a variety of user applications and benchmarks.

NSDI '12 Home
Registration Information
Discounts
Organizers
At a Glance
Technical Sessions
Poster and Demo Session
Birds-of-a-Feather Sessions
Workshops
Sponsors
Activities
Calendar
Hotel and Travel Information
Students
Questions?
Help Promote
For Participants

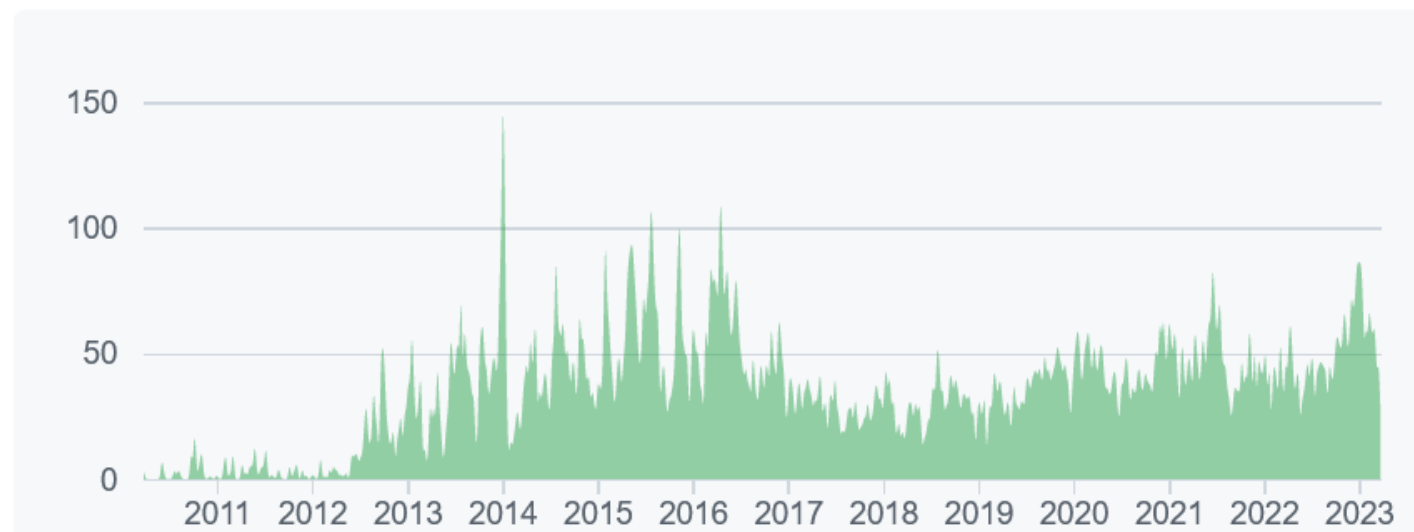
A bit of stats

- Language: Scala
- Since 2009, more than 1200 developers have contributed to Spark!
- The project's [committers](#) come from more than 25 organizations.

Mar 28, 2010 – Mar 29, 2023

Contributions: Commits ▼

Contributions to master, excluding merge commits and bot accounts



Why Spark?

Spark is just one solution that facilitates analysis on large data.

Other options:

- using the Message Passing Interface (MPI) library
- similar frameworks e.g. [Apache Flink](#) (more streaming-specific)
- Python-specific [Dask](#) (nice abstraction for scaling python-native applications)

Flexibility

Spark's flexibility is what makes it so popular.

The spark runtime can be deployed on:

- a single machine (local)
- a set of pre-defined machines (stand-alone)
- a dedicated Hadoop-aware scheduler (YARN/Mesos)
- "cloud", e.g. Amazon EC2
- Kubernetes

Incremental, interactive development

Deployment workflow: start small (local) and scale up to one of the other solutions, depending on needs and resources.

In addition, you can run applications on any of these platforms either

- interactively through a shell (or a Jupyter notebook as we'll see)
- batch mode

No code changes to go between these methods of deployment!

Spark's flexibility

Supported languages

- Scala
- Java
- R
- Python
- .NET
- Julia

Supported data storage

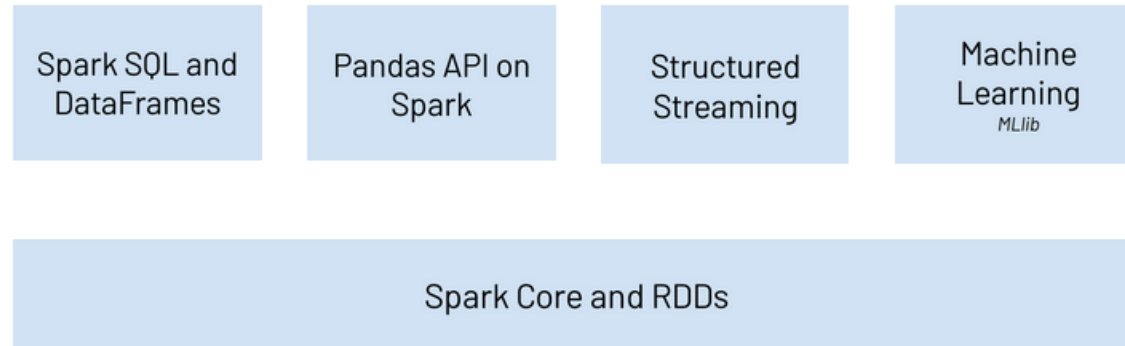
- Cassandra
- HDFS
- Alluxio
- Hbase
- Hive

Specialized libraries

- SparkSQL
- SparkStreaming
- MLlib
- GraphX

PySpark components

- Py

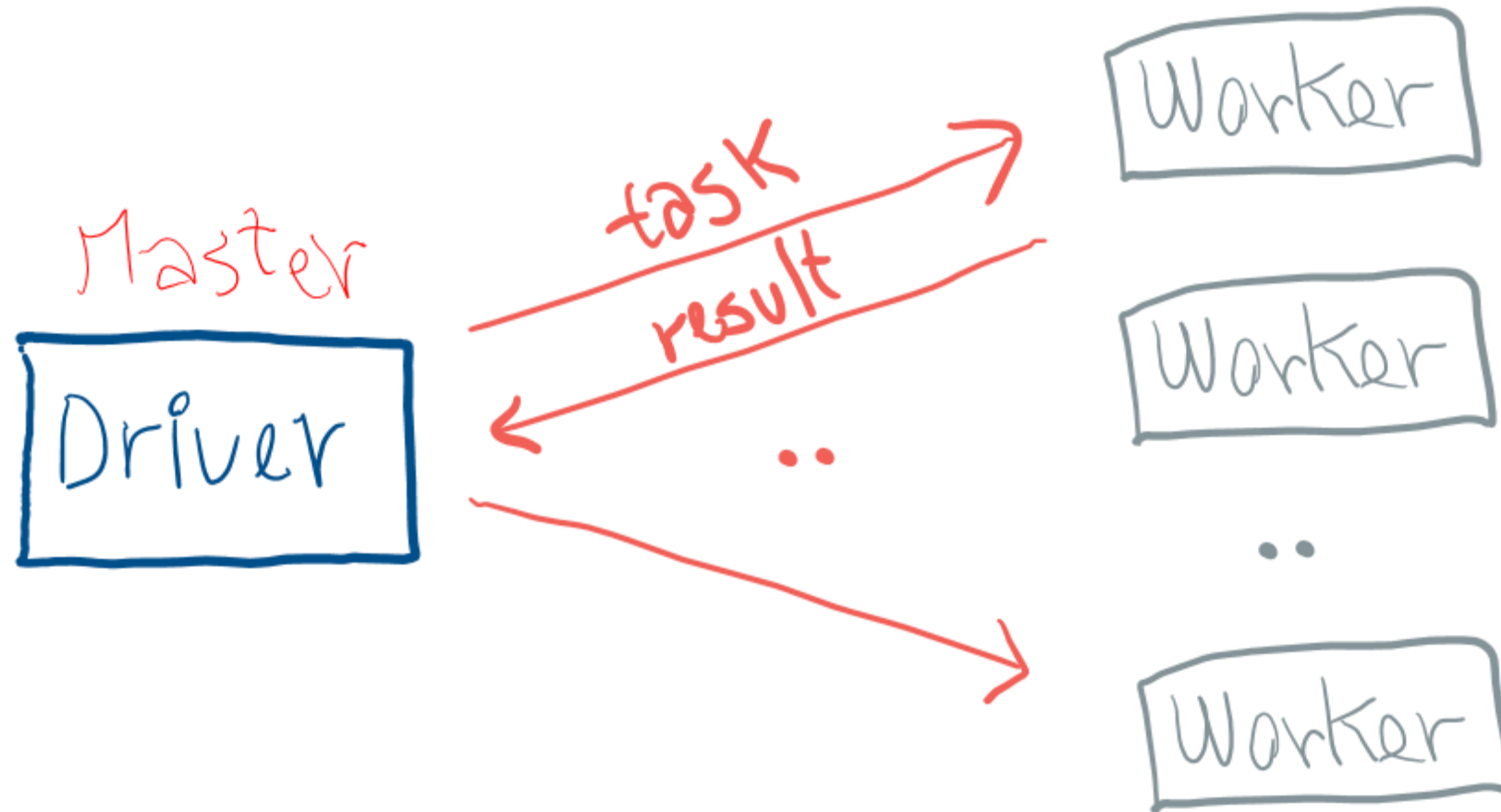


Spark Architecture Overview

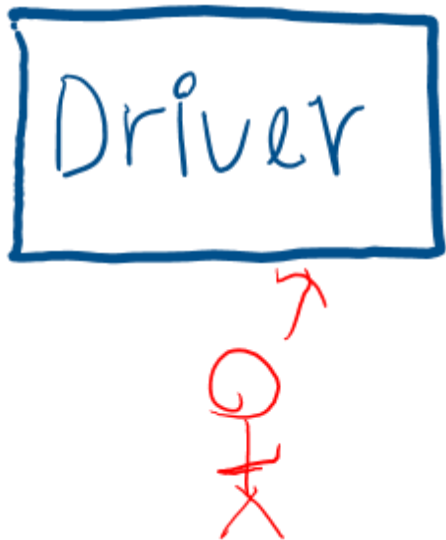
Distributed computing framework

- Revisit
 1. Distribute work
 2. Orchestrate task execution
 3. Collect results
- Fault-tolerant
- Efficient

Spark architecture



Driver



- coordinates the work to be done
- keeps track of tasks
- collects metrics about the tasks (disk IO, memory, etc.)
- communicates with the workers (and the user)

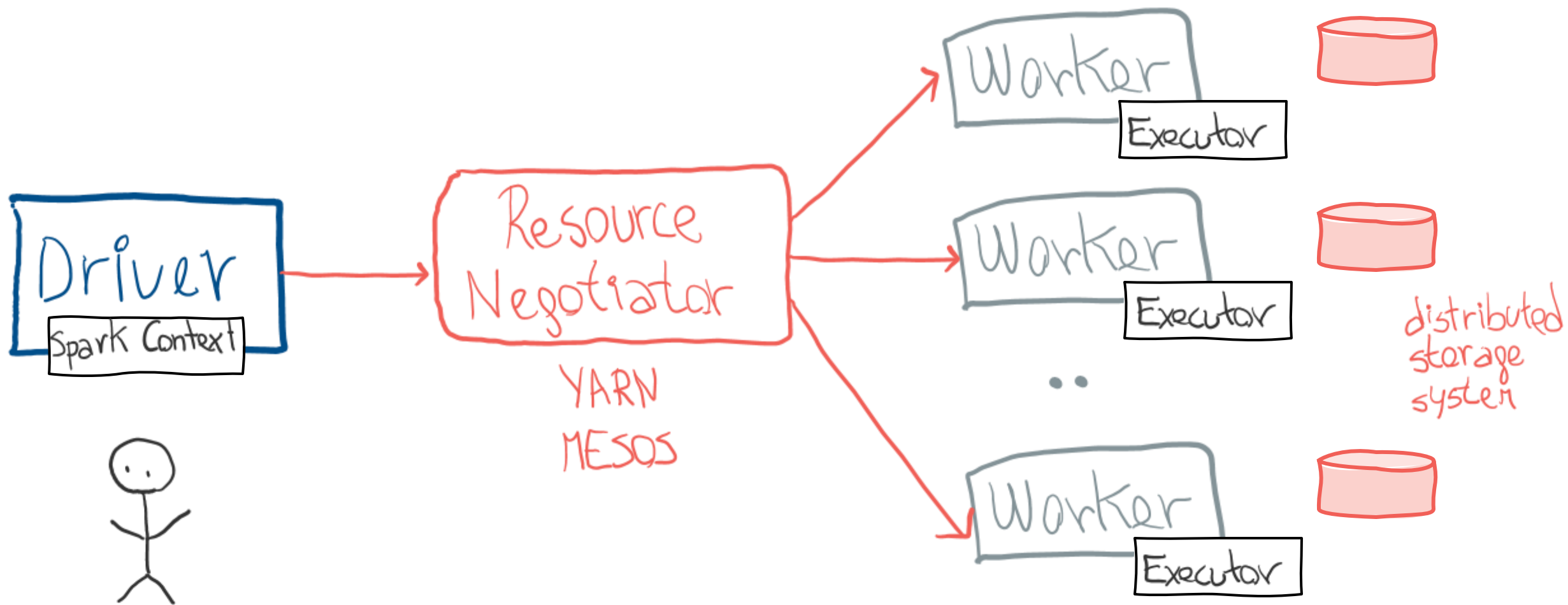
Worker

- receive tasks to be done from the driver
- store data in memory or on disk
- perform calculations
- return results to the driver

The user's access point to this Spark universe is the **Spark Context** which provides an interface to generate RDDs.



Spark architecture



Resilient Distributed Dataset

Basic Data Abstraction

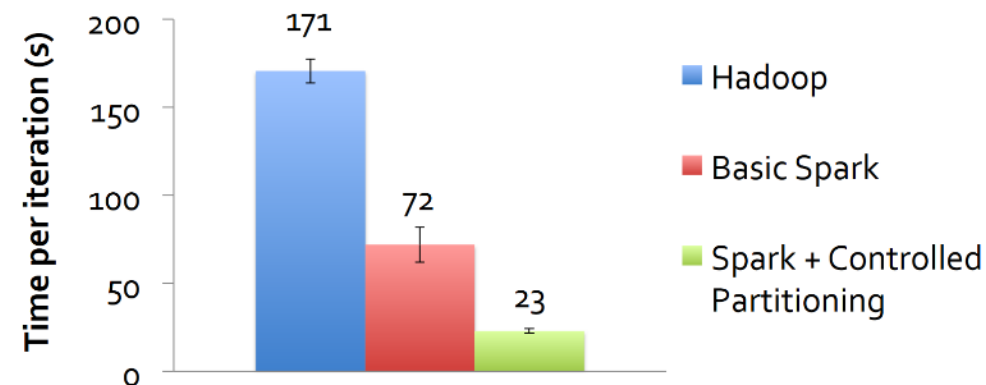
RDD: primary interface of every Spark application

- keeps track of data distribution across the workers
- provides an interface to the user to access and operate on the data
- fault-tolerant and efficient for iterative algorithms

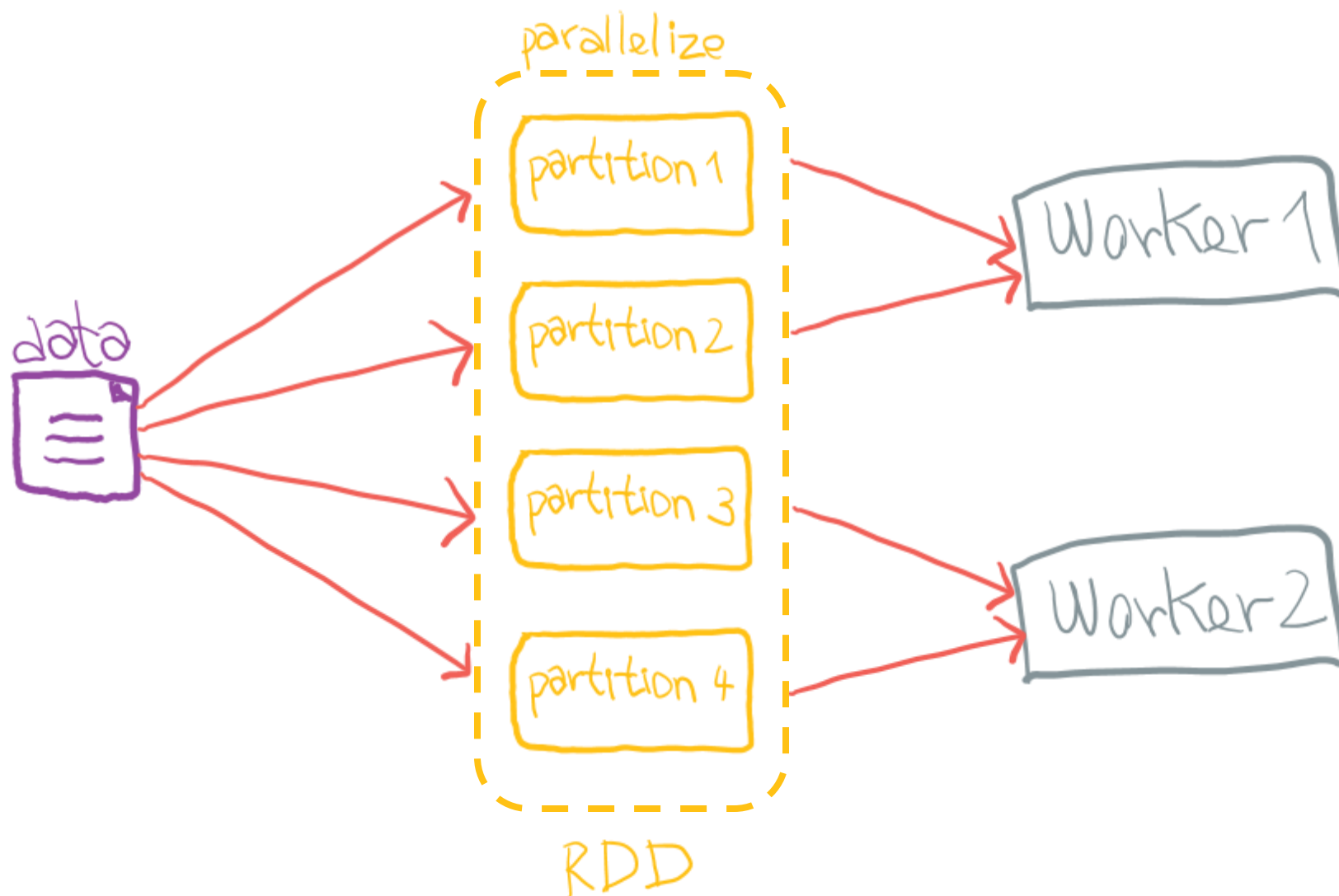
As a Spark user, you write applications that feed data into RDDs and subsequently transform them into something useful

Why RDDs?

- Use case: iterative algorithms ML, graph, ad-hoc queries
- Hadoop MapReduce: stable storage
 - HDFS, disk, replication (disk I/O)
 - for every stage!
- RDD keep data in memory
 - Faster BUT
 - expensive fault-tolerance
- RDD lineage
 - state can be rebuilt with lineage (cheap to replicate)
 - limit operations: immutable, coarse-grained actions



RDDs



RDD operations

Once an RDD is created, it is **immutable** - it can only be transformed into a new RDD via a **transformation**.

A transformation, however, does not trigger any computation, only updates the DAG.

Calculations are triggered by **actions**.

Transformations

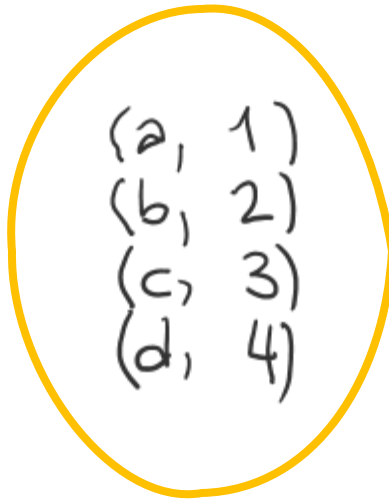
Name	Use
<code>map(func)</code>	the most basic transformation with 1:1 correspondence to original data
<code>filter(func)</code>	only keep those elements for which the filter function evaluates to True
<code>mapPartitions(func)</code>	similar to map but done on a per-partition basis (requires a generator function)
<code>flatMap(func)</code>	similar to map but each element can be mapped to more than one corresponding item (func returns a Seq)
<code>distinct([numPartitions])</code>	only retain the unique elements of the entire RDD
<code>reduceByKey(func, [numPartitions])</code>	group elements by key and keep the data distributed
<code>groupByKey([numPartitions])</code>	sort elements based on keys K, where K implements Ordered

Transformations are evaluated "lazily" - only executed once an action is performed.

Transformations: map

map

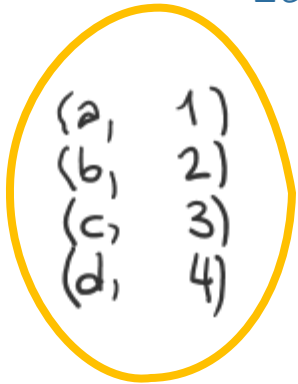
```
lambda (k,v): (k, 2*v)
```



Transformations: flatMap

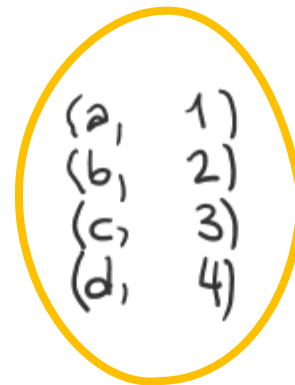
map

```
lambda(k,v): (k, range(v))
```



flatMap

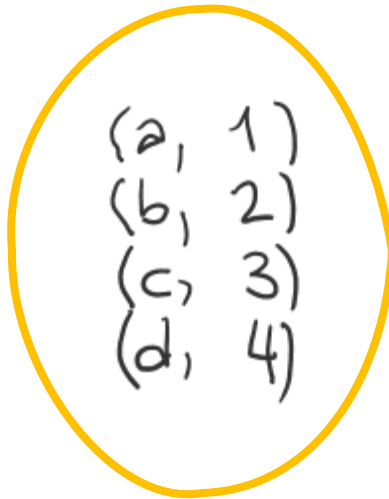
```
lambda(k,v): [(k,v) for v in range(v)]
```



Transformations: filter

filter

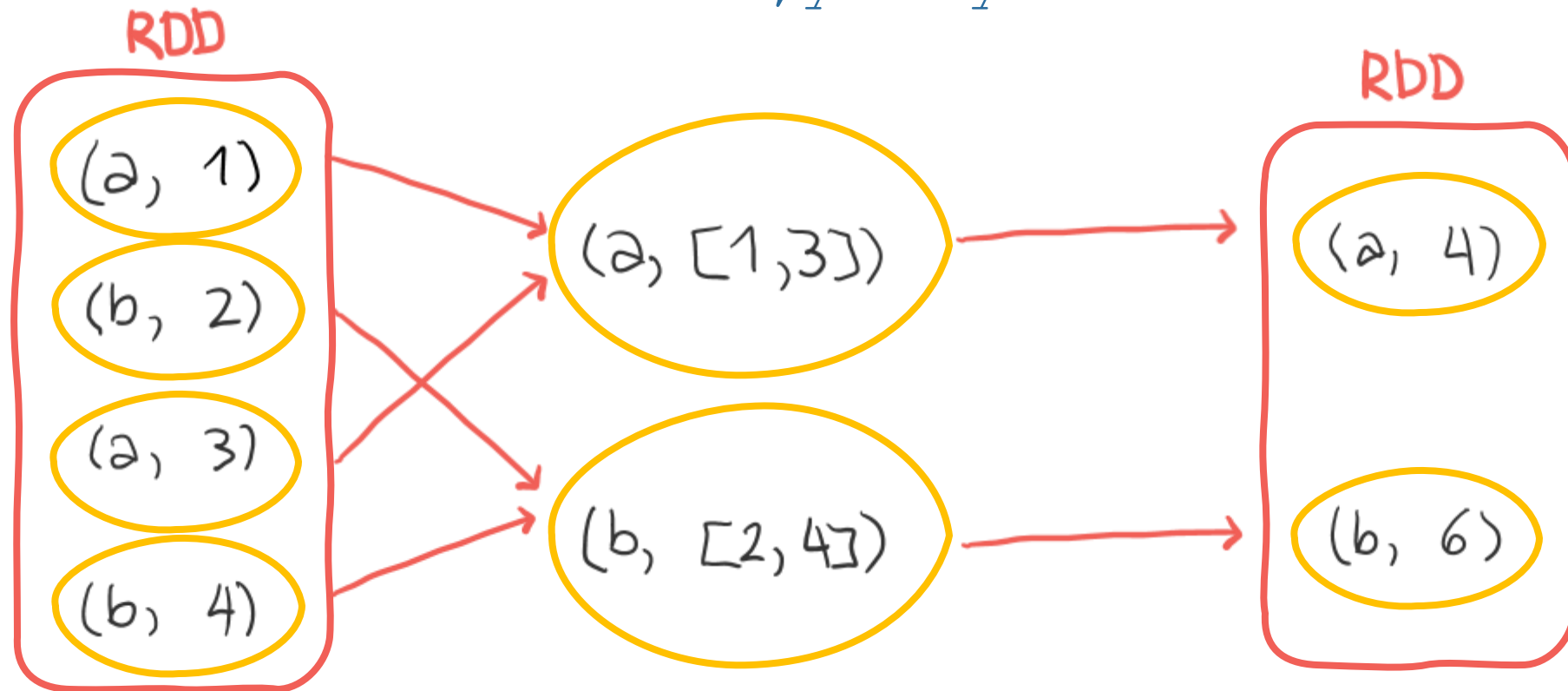
lambda (k, v) : v % 2



Transformations: reduceByKey

reduceByKey

`lambda x, y: x+y`



Actions

Name	Use
<code>reduce(func)</code>	reduces the entire RDD to a single value
<code>collect()</code>	pulls all elements of the RDD to the driver (often a bad idea!!)
<code>first()</code>	returns the first element of the RDD to the driver
<code>take(n)</code>	yields a desired number of items to the driver
<code>countByKey()</code>	yields a hashmap of (K, Int) pairs with the count of each key

Don't worry, you will soon get to practice with most of these!

Lineage

- When an RDD is transformed, this transformation is not automatically carried out.
- Instead, the system remembers how to get from one RDD to another and only executes whatever is needed for the action that is being done.
- this allows one to build up a complex "pipeline" and easily tweak/rerun it in its entirety

Getting started with Spark

Initializing Spark

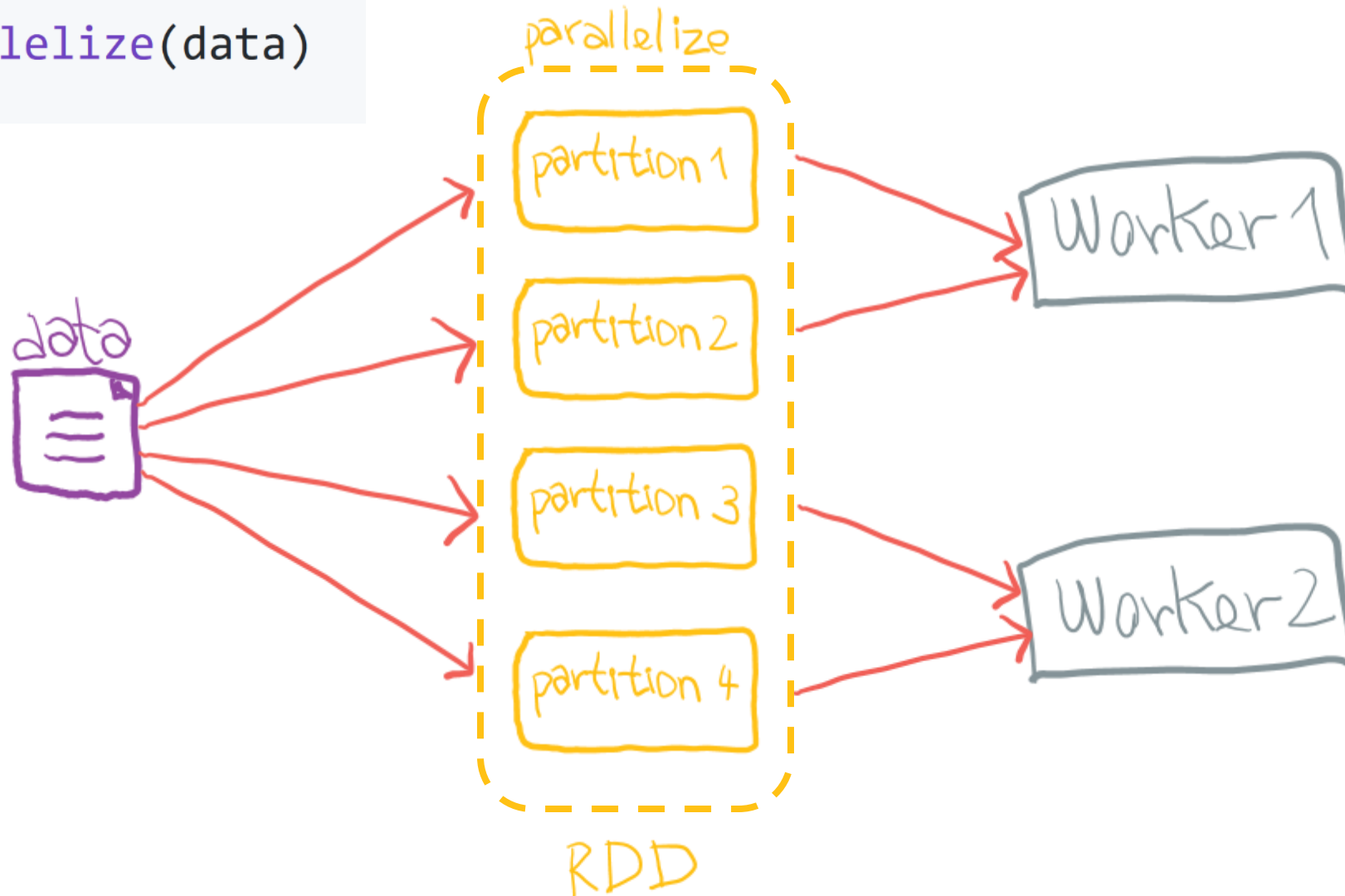
```
import pyspark  
  
sc = pyspark.SparkContext()
```

Launches the Spark runtime and connects the application to the master.

This creates a driver which can now be used to dispatch work to the resources allocated for the application.

Create an RDD: parallelize

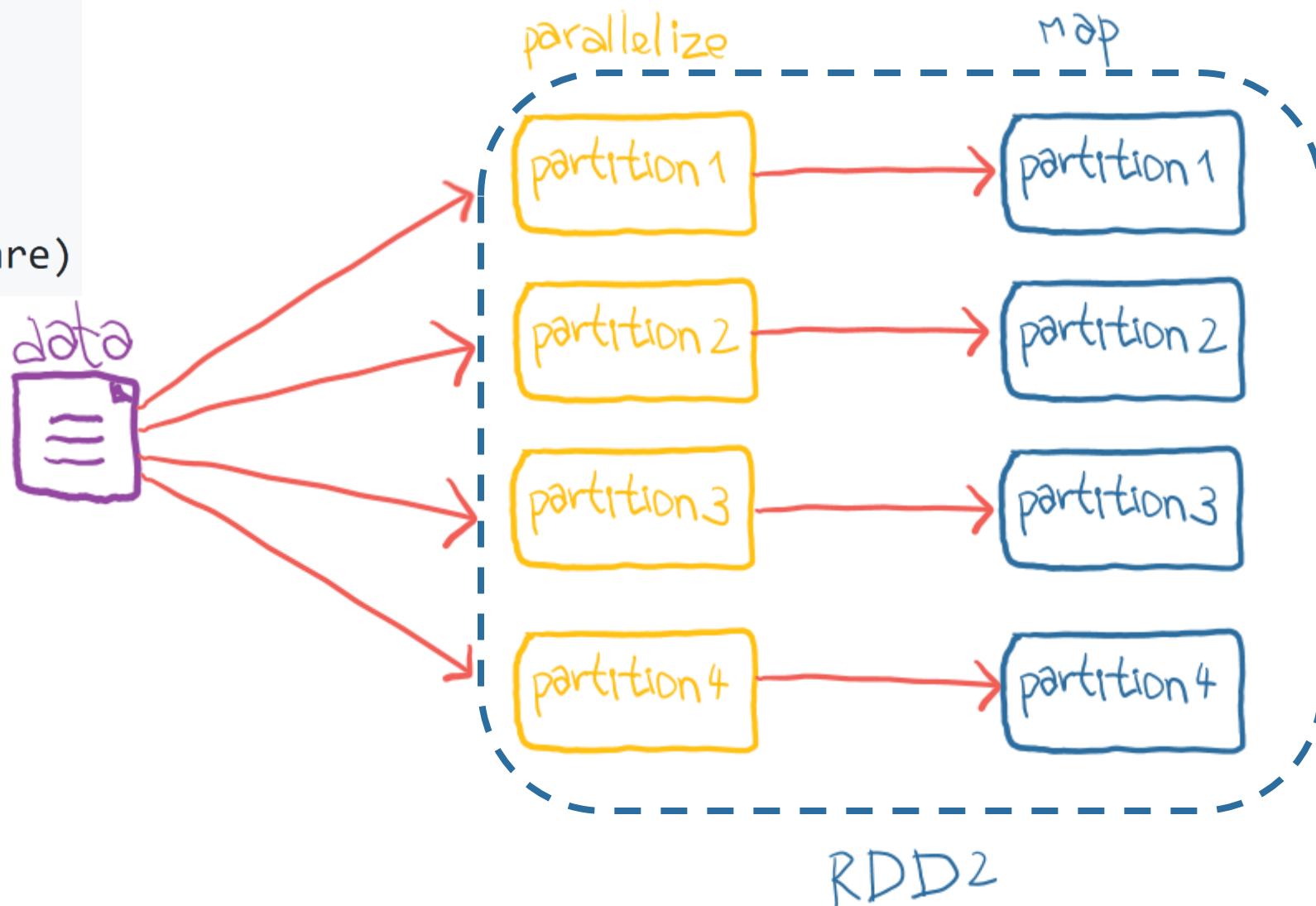
```
rdd = sc.parallelize(data)
```



Transform an RDD: map

```
def square(x):  
    return x*x
```

```
rdd = sc.parallelize(data)  
rdd_squared = rdd.map(square)
```

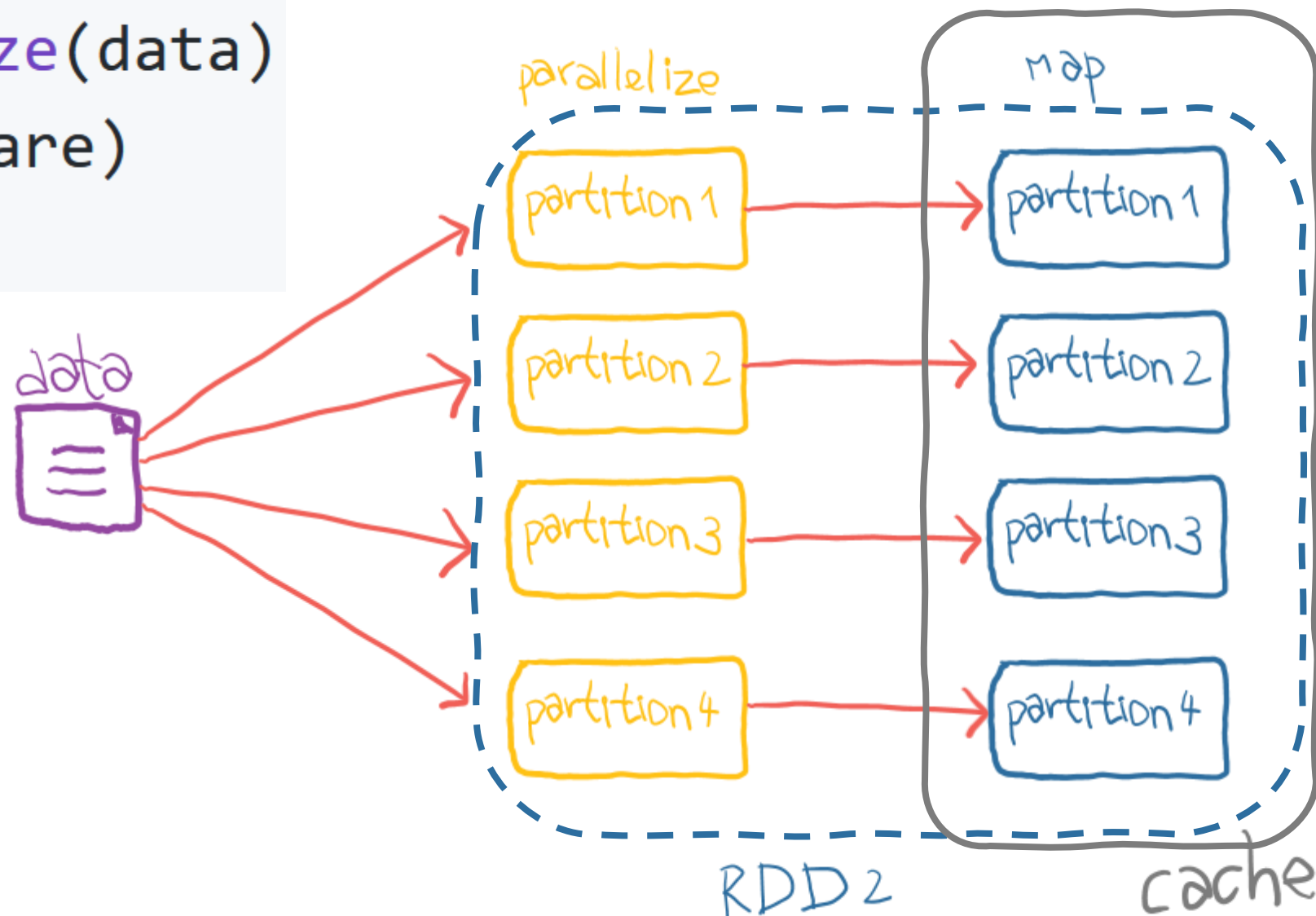


Caching

- RDD evaluations are *lazy*
- whenever an action is performed, the entire lineage graph is recalculated
- unless! an intermediate RDD is cached -- then it is only calculated once and reused from memory each subsequent time
- this allows for good performance when iterating on an RDD is required

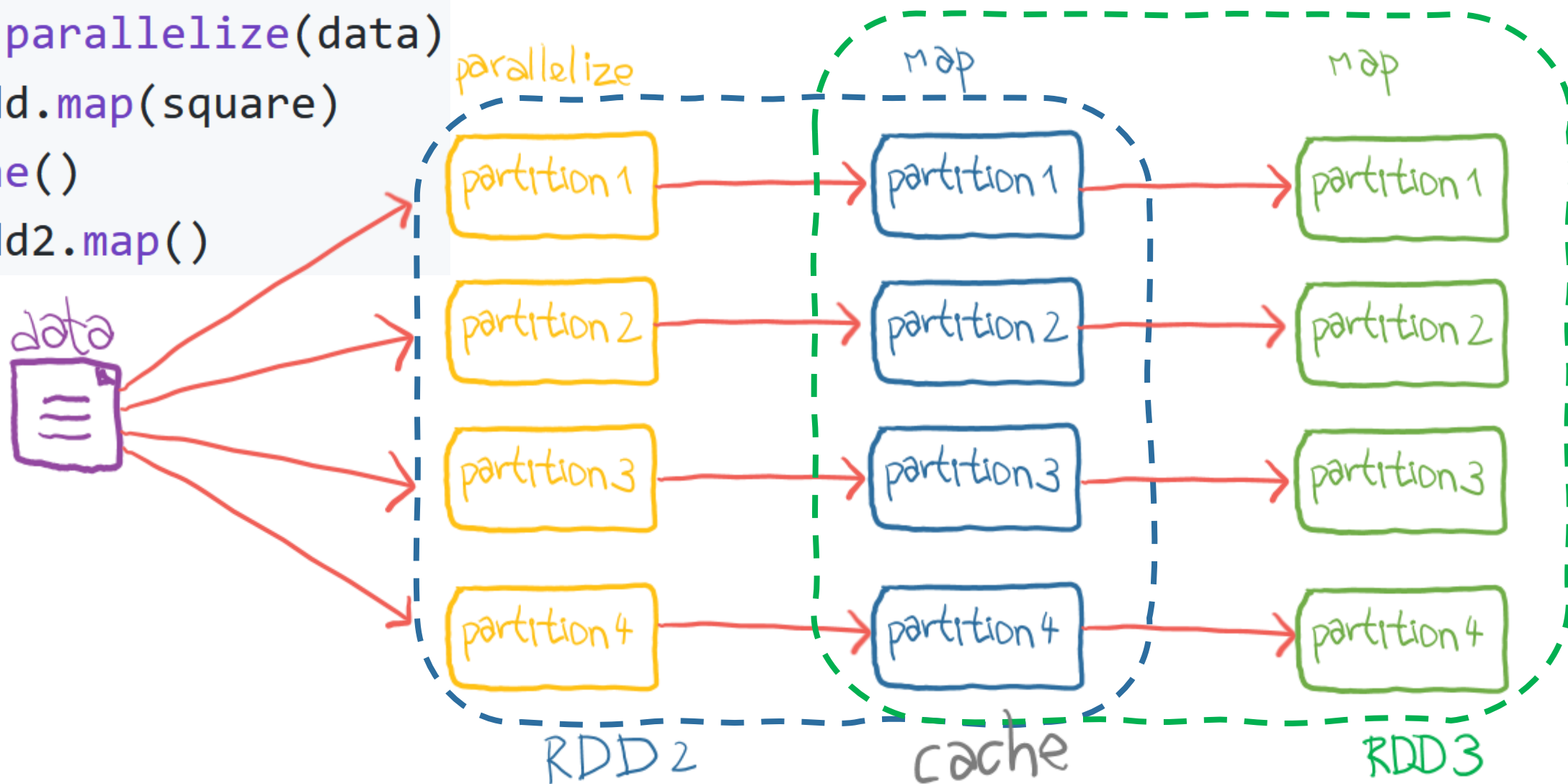
Caching

```
rdd = sc.parallelize(data)
rdd2 = rdd.map(square)
rdd2.cache()
```



Caching example

```
rdd = sc.parallelize(data)
rdd2 = rdd.map(square)
rdd2.cache()
rdd3 = rdd2.map()
```



Partitioning

- data of each RDD is partitioned and each partition is assigned to an executor
- each partition in a transformation results in a task
- there may be many more tasks than cores in the system, which allows for good utilization by fine-graining the overall load.

References

- Spark official website <https://spark.apache.org/>
- Spark RDD documentation <https://spark.apache.org/docs/latest/rdd-programming-guide.html>
- Spark RDD paper <https://www.usenix.org/system/files/conference/nsdi12/nsdi12-final138.pdf>
- Spark github repository <https://github.com/apache/spark/>
- MapReduce paper <https://dl.acm.org/doi/abs/10.1145/1327452.1327492>
- Yarn paper <https://dl.acm.org/doi/10.1145/2523616.2523633>

Start your engines

<https://dslabgit.datascience.ch/course/2025/module-3a>

(fork and git clone)

Time for the basic Spark tutorial!

- Gitlab repo for this week:
- <https://dslabgit.datascience.ch/course/2025/module-3a>
- Spark and RDDs
 - Python refresher
 - Spark introduction
 - Gutenberg

Analyzing the Gutenberg corpus

- The [Gutenberg Project](#) is a large free repository of books and other media in different languages (but primarily English)
- We will use it to do some basic text analysis using key/value PairRDDs in Spark.

The data

- We have pre-processed the data already and created an RDD for you to use. The RDD consisting of (ID, text) key-value pairs can be found in
- `HDFS /data/com-490/gutenberg/rdd`
- This RDD will form the basis of the work in the notebook.

The goal

- The eventual goal is to produce something like the [Google NGram viewer](#), but for the Gutenberg corpus.
- Because the notebook is quite long, the last part of the notebook is already filled in, but feel free to run it!