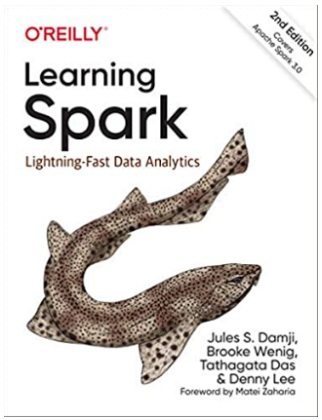
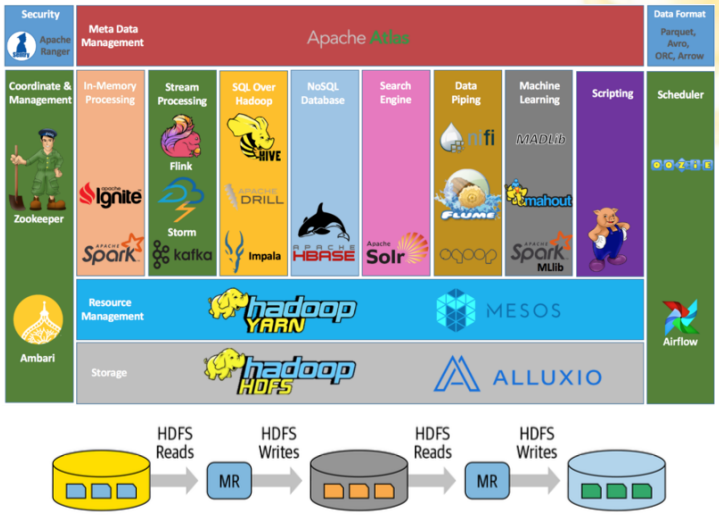
9.1: Apache Spark: Principles

- **Instructor**: Dr. GP Saggese, gsaggese@umd.edu
- **References**:
  - Concepts in the slides
  - Academic paper
    - "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing", 2012
  - Mastery
    - "Learning Spark: Lightning-Fast Data Analytics" (2nd Edition)
    - Not my favorite, but free here



Hadoop MapReduce: Shortcomings

- **Hadoop is hard to administer**
  - Many layers (HDFS, Yarn, Hadoop, . . .)
  - Extensive configuration
- **Hadoop is hard to use**
  - Verbose API
  - Limited language support (e.g., Java is native)
  - MapReduce jobs read / write data on disk



- **Large but fragmented ecosystem**
  - No native support for:
    - Machine learning
    - SQL, streaming
    - Interactive computing
  - New systems developed on Hadoop for new workloads
  - E.g., Apache Hive, Storm, Impala, Giraph, Drill

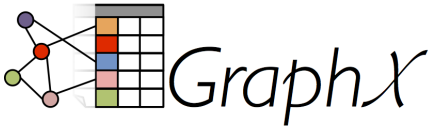
(Apache) Spark

- **Open-source**
  - DataBrick monetizes it ($40B startup)

- **General processing engine**
  - Large set of operations beyond `Map()` and `Reduce()`
  - Combine operations in any order
  - Transformations vs Actions
  - Computation organized as a DAG
    - DAGs decomposed into parallel tasks
  - Scheduler/optimizer for parallel workers
- **Supports several languages**
  - Java, Scala (preferred), Python supported through bindings
- **Data abstraction**
  - Resilient Distributed Dataset (RDD)
  - DataFrames, Datasets built on RDDs
- **Fault tolerance through RDD lineage**
- **In-memory computation**

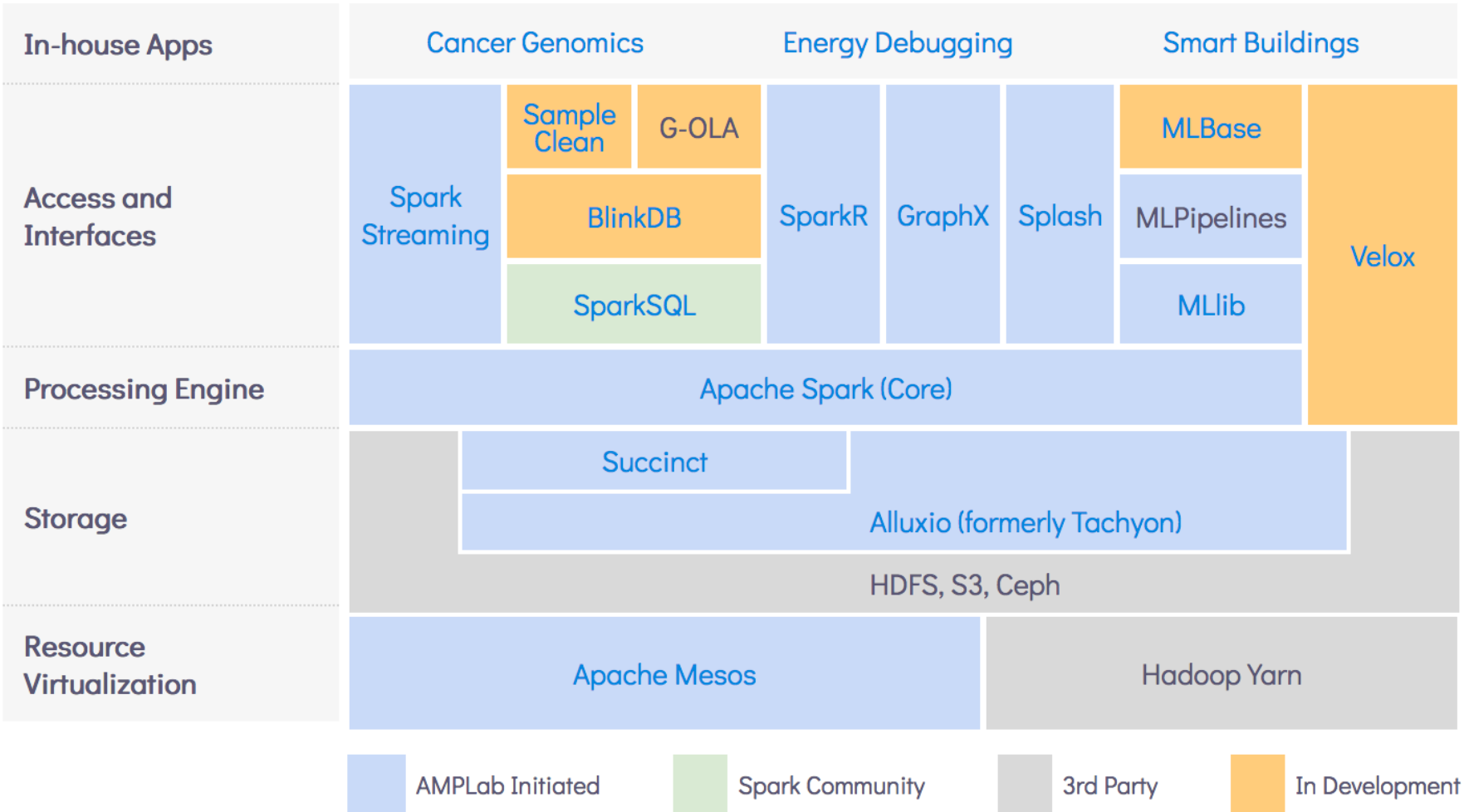
Berkeley: From Research to Companies

- Amplab
- Rise lab



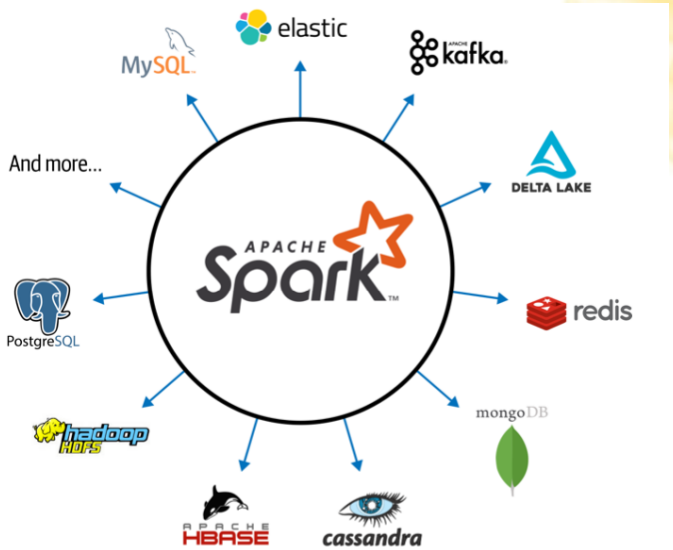
Berkeley AMPLab Data Analytics Stack

- So many tools that they have their own big data stack! https://amplab.cs.berkeley.edu/software/

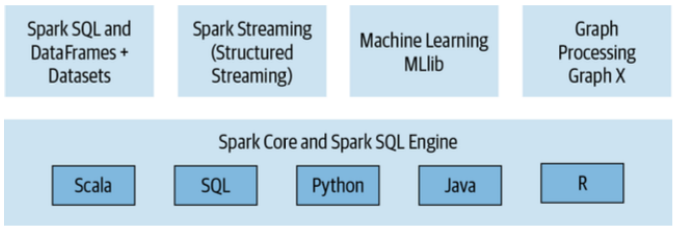


Apache Spark

- **Unified stack**
  - Different computation models in a single framework
- **Spark SQL**
  - ANSI SQL compliant
  - Work with structured relational data
- **Spark MLlib**
  - Build ML pipelines
  - Support popular ML algorithms
  - Built on top of Spark DataFrame
- **Spark Streaming**
  - Handle continually growing tables
  - Tables are treated as static table
- **GraphX**
  - Manipulate graphs
  - Perform graph-parallel computation
- **Extensibility**
  - Read from a many sources
  - Write to many backends



One computation engine

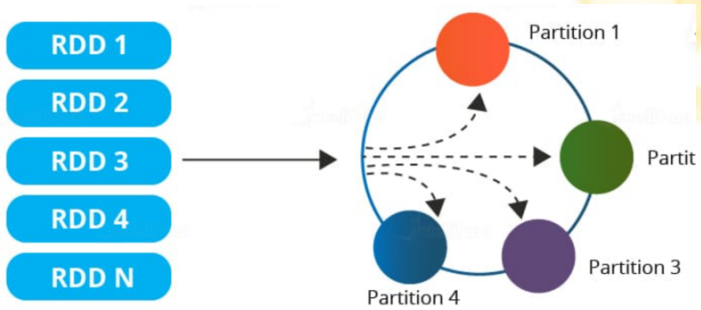


General purpose applications

SCIENCE ACADEMY

Resilient Distributed Dataset (RDD)

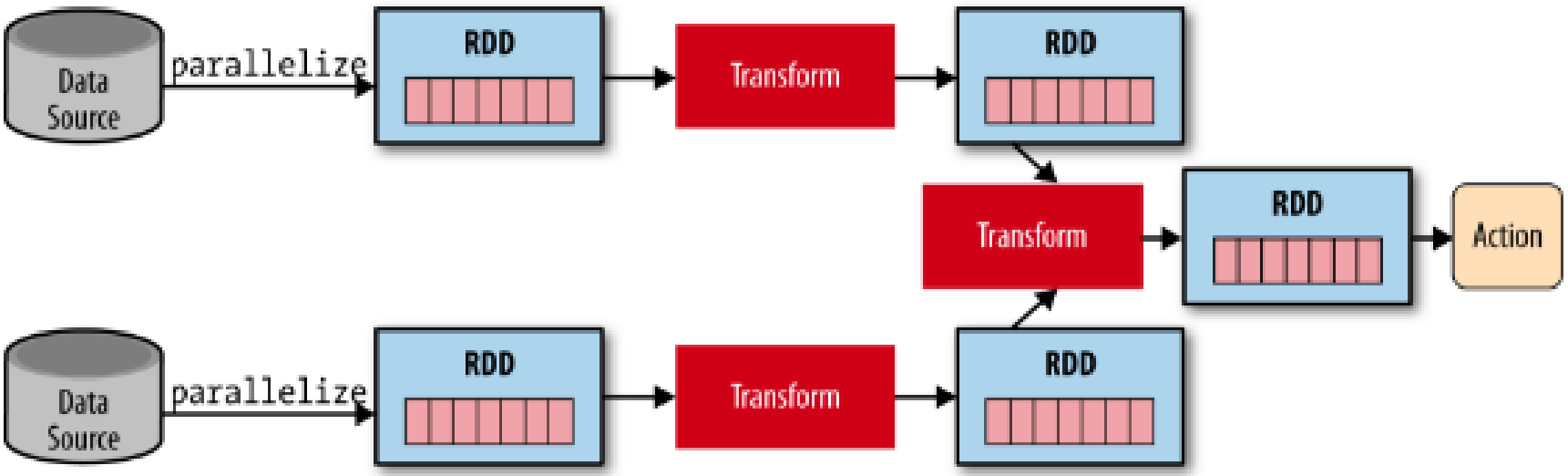
- **A Resilient Distributed Dataset (RDD)**
  - Collection of data elements
  - Partitioned across nodes
  - Operated on in parallel
  - Fault-tolerant
  - In-memory / serializable



- **Applications**
  - Best for applications applying the same operation to all dataset elements (vectorized)
  - Less suitable for asynchronous fine-grained updates to shared state
    - E.g., updating one value in a dataframe
- **Ways to create RDDs**
  - *Reference* data in external storage
    - E.g., file-system, HDFS, HBase
  - *Parallelize* an existing collection in your driver program
  - *Transform* RDDs into other RDDs

Transformations vs Actions

- **Transformations**
  - Lazy evaluation
  - Nothing computed until an Action requires it
  - Build a graph of transformations
- **Actions**
  - When applied to RDDs force calculations and return values
  - Aka Materialize



Spark Example: Estimate Pi

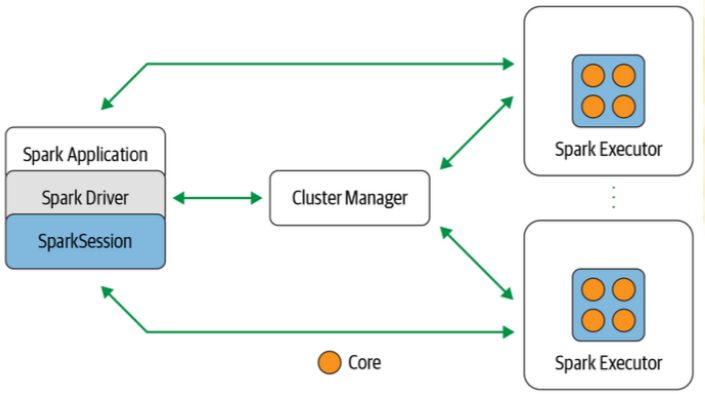
```
# Estimate π (compute-intensive task).
# Pick random points in the unit square [(0,0)-(1,1)].
# See how many fall in the unit circle center=(0, 0), radius=1.
# The fraction should be π / 4.

import random
random.seed(314)

def sample(p):
    x, y = random.random(), random.random()
    in_unit_circle = 1 if x*x + y*y < 1 else 0
    return in_unit_circle

# "parallelize" method creates an RDD.
NUM_SAMPLES = int(1e6)
count = sc.parallelize(range(0, NUM_SAMPLES)) \
           .map(sample) \
           .reduce(lambda a, b: a + b)
approx_pi = 4.0 * count / NUM_SAMPLES
print("pi is roughly %f" % approx_pi)
```

executed in 386ms, finished 04:27:53 2022-11-23

```
pi is roughly 3.141400
```

Spark: Architecture

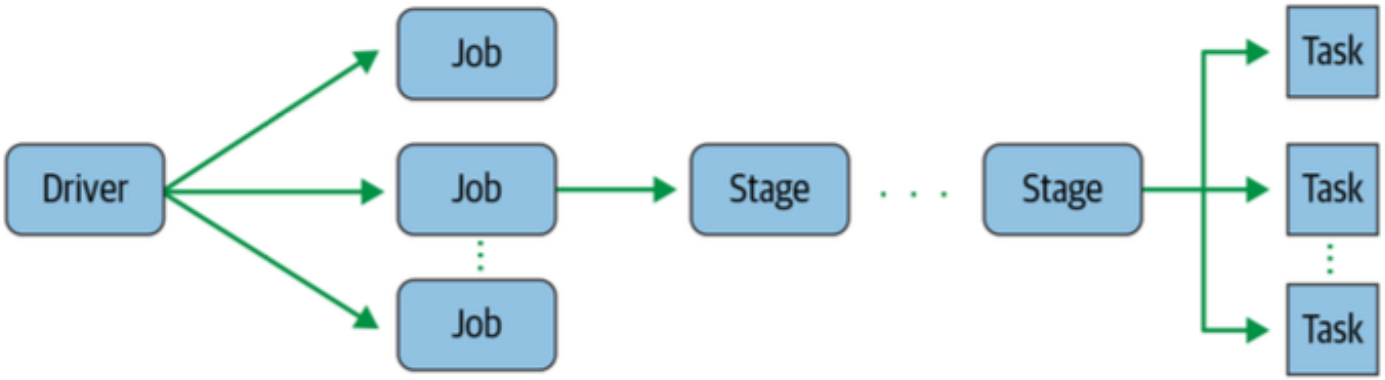
- **Architecture**
  - Who does what
  - Responsibilities of each piece
- **Spark Application**
  - Code describing computation
  - E.g., Python code calling Spark



- **Spark Driver**
  - Instantiate *SparkSession*
  - Communicate with *Cluster Manager* for resources
  - Transform operations into DAG computations
  - Distribute task execution across *Executors*
- **Spark Session**
  - Interface to Spark system
- **Cluster Manager**
  - Manage and allocate resources
  - Support Hadoop, YARN, Mesos, Kubernetes
- **Spark Executor**
  - Run worker node to execute tasks
  - Typically one executor per node
  - JVM

Spark: Computation Model

- **Architecture** = who does what
- **Computational model** = how are things done



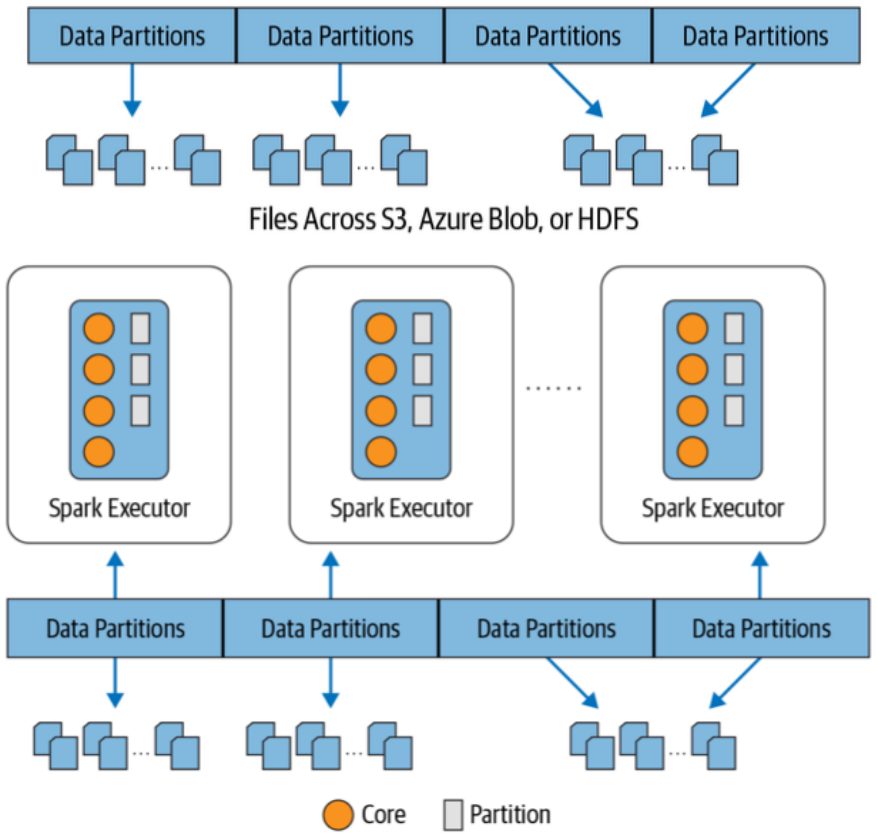
- **Spark Driver**
  - Converts Spark application into one or more Spark *Jobs*
  - Describes computation with *Transformations* and triggers with *Actions*
- **Spark Job**
  - Parallel computation in response to a Spark *Action*
  - Each *Job* is a DAG with one or more dependent *Stages*
- **Spark Stage**
  - Smaller operation within a *Job*
  - *Stages* can run serially or in parallel
- **Spark Task**
  - Each *Stage* has multiple *Tasks*

Deployment Modes

- Spark can run on several different configurations
  - **Local**
    - E.g., run on your laptop
    - Driver, Cluster Manager, Executors all run in a single JVM on the same node
  - **Standalone**
    - Driver, Cluster Manager, Executors run in different JVMs on different nodes
  - **YARN** or **Kubernetes**
    - Driver, Cluster Manager, Executors run on different pods (i.e., containers)

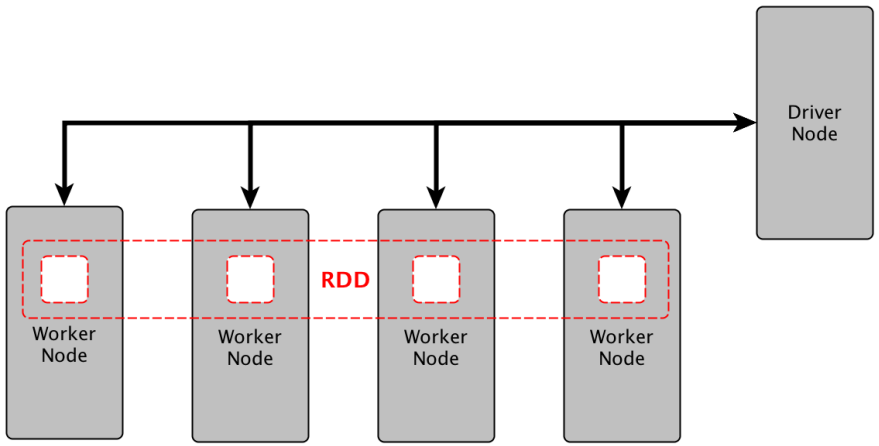
Distributed Data and Partitions

- **Data is distributed** as partitions across different physical nodes
  - Store each partition in memory
  - Enable efficient parallelism
- **Spark Executors** process data "close" to them
  - Minimize network bandwidth
  - Ensure data locality
  - Similar approach to Hadoop



Parallelized Collections

- Parallelized collections created by calling *SparkContext* `parallelize()` on an existing collection



- Data spread across nodes
- Number of *partitions* to cut dataset into
  - Spark runs one *Task* per partition
  - Aim for 2-4 partitions per CPU
    - Spark sets partitions automatically based on your cluster
    - Set manually by passing as a second parameter to `parallelize()`