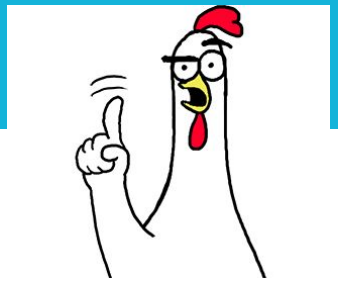
# topic 35:
# big data + pyspark

FLATIRON SCHOOL

# Agenda

- An overview of big data
- Distributed systems with MapReduce
- The Hadoop ecosystem
- Spark and PySpark
  - Spark Context
  - Resilient Distributed Datasets (RDD) in Spark
    - What is RDD?
    - RDD transformations and actions
  - Spark use cases

# You WON’T be able to...

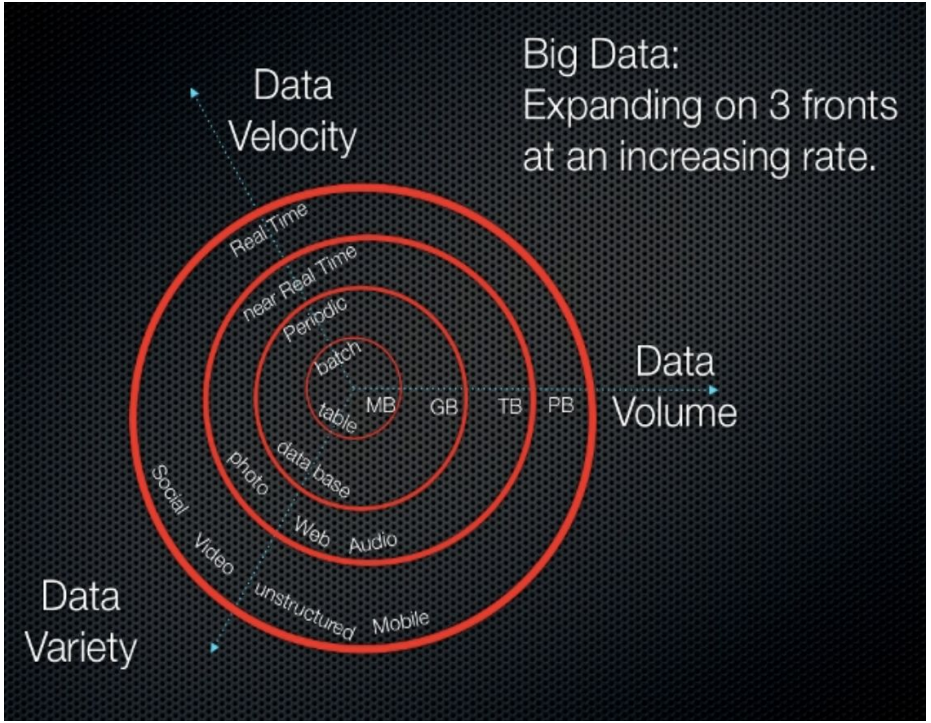


- Write and produce MapReduce jobs in Java
- Become proficient in end-to-end ETL with data in the hadoop ecosystem
- Working with AWS instances and productionizing your ml models made with pyspark

The frameworks involving MapReduce, Docker, PySpark, and distributed systems in general are highly technical and specialized, so we will only learn a high level overview of it

# What is Big Data?

- Three V's of big data
  - Volume
  - Velocity
  - Variety



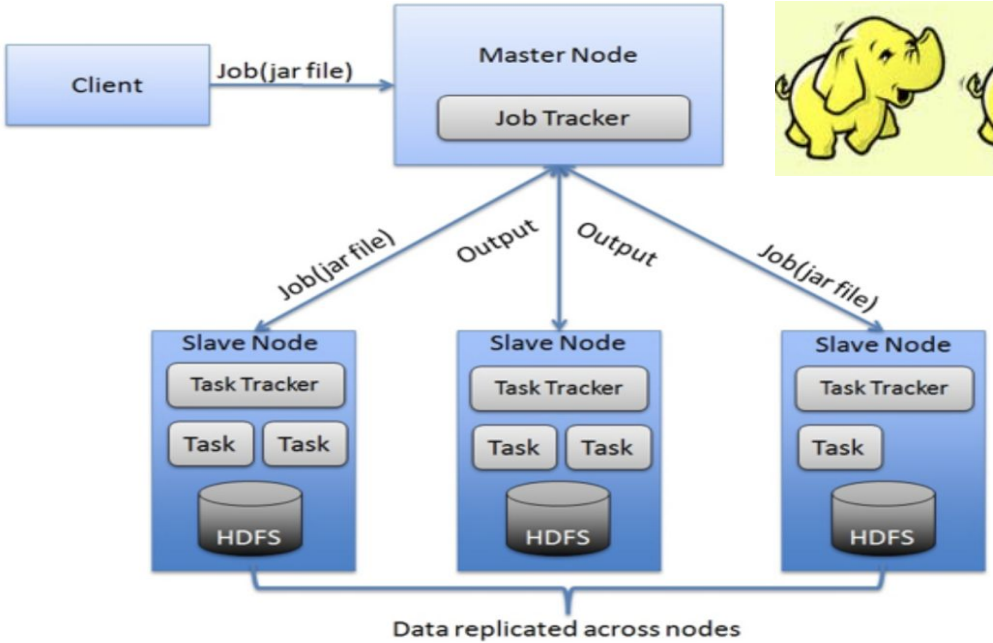
# Why do we need distributed systems?...

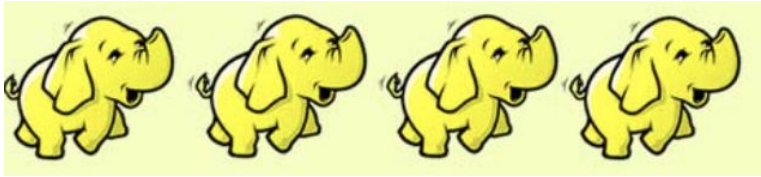
- How long would it take for 1TB of data to be read on a hard drive?
  - 1 TB = 1024 GB = 1048576 MB
  - 1048576 / 100 MB/s  = 2.91 hours

  - But with 100 MB/s across 10 systems:
  - 1 TB/ 1000  MB/s = 17 min
  - 1 TB/  10,000 MB/s = 105 seconds

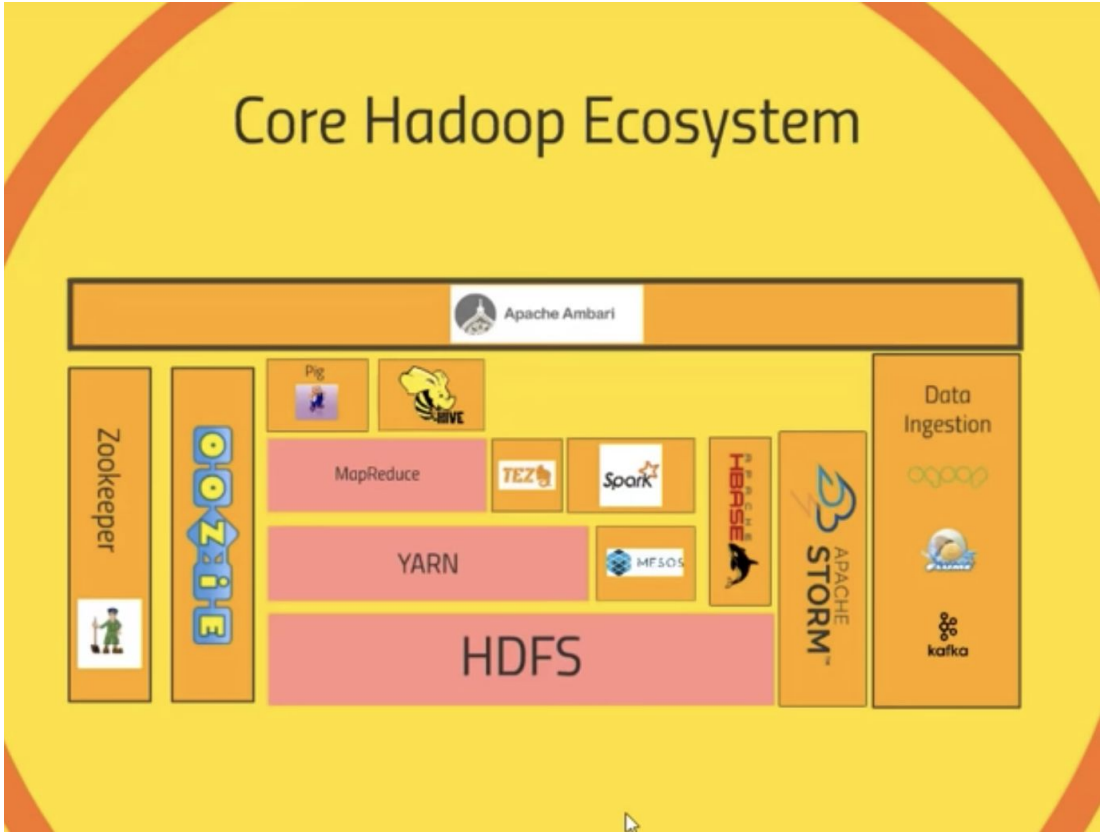
# Distributed system - Hadoop

Hadoop is a distributed computing system and environment for very large datasets on computer clusters



# Hadoop Ecosystem




**HDFS** stands for Hadoop Distributed File System: allows us to distribute data across a cluster of computers, making multiple hard drives in our cluster function like a singular file system

**YARN** (Yet Another Resource Negotiator) that deals with data processing. It decides what each node runs, and when. Thought of as the executive and the heartbeat the keeps the cluster running

Other resources like **MapReduce** can be built on top of **YARN**

# Divide & Conquer

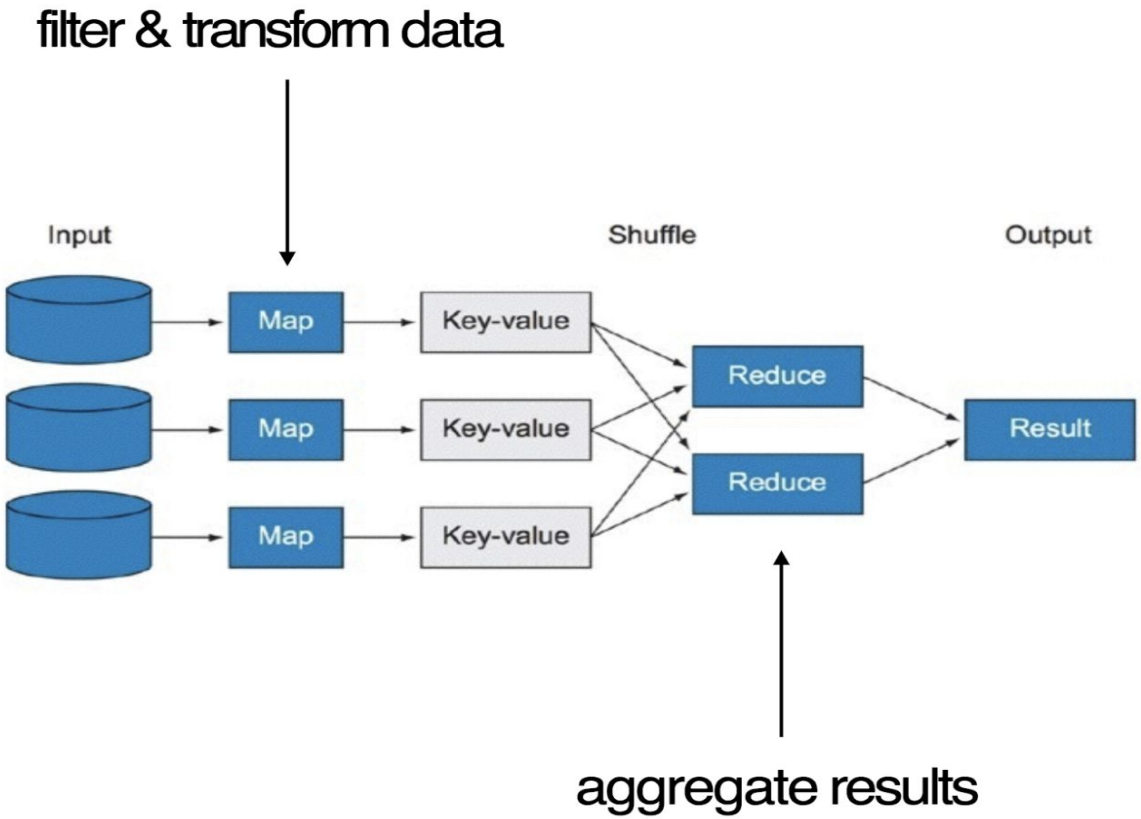
- Instead of using just one system to perform a task, Hadoop and **MapReduce** allow tasks to be divided across multiple systems:
  a. Tasks are split into many subtasks
  b. Subtasks are solved individually
  c. Subtasks are recombined into a final result

# MapReduce

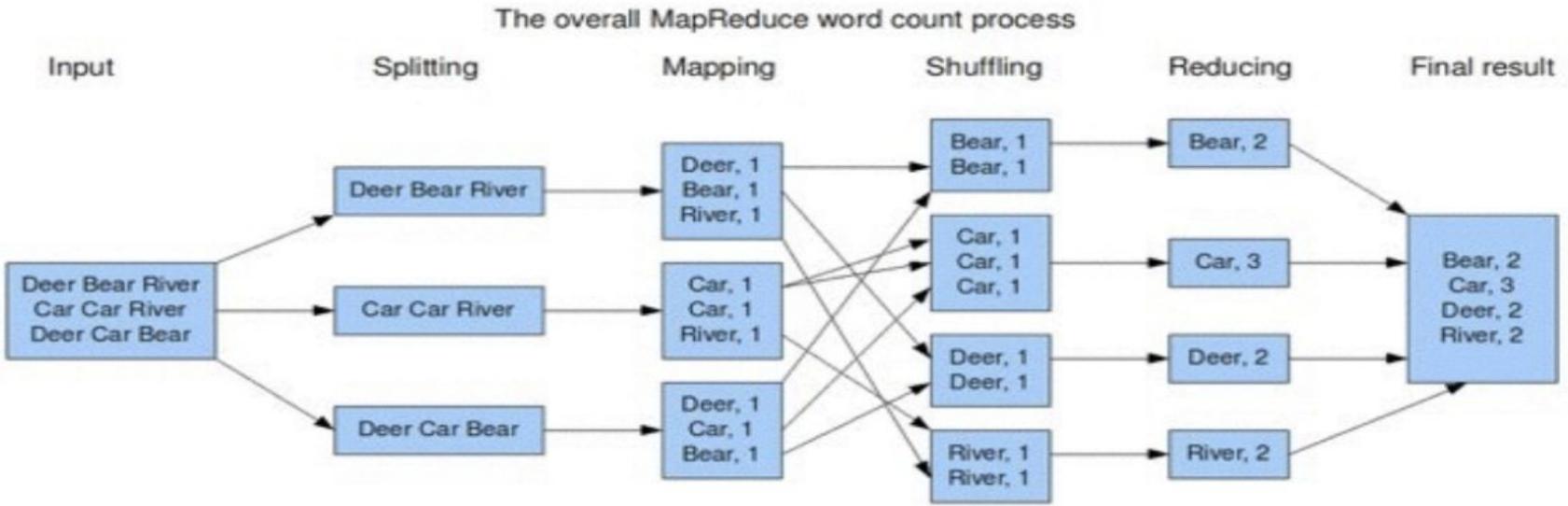
- MapReduce splits tasks into subtasks and allows them to be processed in parallel
- This happens in two phases:
  - The mapper phase - transforms your data in parallel across all nodes in your computing cluster in an efficient manner
  - The reducer phase - aggregate your entire data and return a final result

```
1  class Mapper
2     method Map(docid id, doc d)
3        for all term t in doc d do
4           Emit(term t, count 1)
5
6  class Reducer
7     method Reduce(term t, counts [c1, c2,...])
8        sum = 0
9        for all count c in [c1, c2,...] do
10           sum = sum + c
11       Emit(term t, count sum)
```

# MapReduce


filter & transform data
Input
Shuffle
Output
Map
Map
Map
Key-value
Key-value
Key-value
Reduce
Reduce
Result
aggregate results

# Example: Word Count with MapReduce



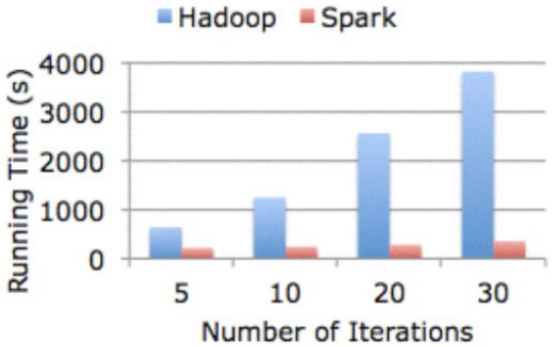
# Beyond Hadoop

- Even though Hadoop speeds up processing tremendously, there are a few problems...
  - Batch processing for iterative algorithms
  - Hadoop jobs are mostly written in Java, which can be painful and error-prone without expertise - making it less accessible

# Beyond MapReduce


```
file = spark.textFile("hdfs://...")

file.flatMap(line => line.split(" "))
    .map(word => (word, 1))
    .reduceByKey(_ + _)
```



# Spark

- Advantages of Spark
  - More fault-tolerant
  - Higher-level query language to implement SQL-like semantics on top of MapReduce
  - Useful for iterative algorithms
  - Some Spark jobs can be up to 100x faster than traditional MapReduce

# PySpark

- Apache **Spark** is written in **Scala**
- Spark is a general purpose data-processing software that offers high level APIs written in Scala, Java, R, and Python

- To support Python with Spark, Apache Spark Community released **PySpark**
- Similar computation speed and power as Scala
- PySpark APIs are similar to Pandas and Scikit-learn

# SparkContext & RDDs

- PySpark has **SparkContext**, an entrypoint into the world of Spark via Python, similar to using a cursor to connect to a SQL database
- Resilient Distributed Datasets (RDDs) are the fundamental data structures of Spark
- RDDs are essentially the Spark representation of a set of data, spread across multiple machines, with APIs to allow you to act on it
- RDDs are to PySpark what Pandas DataFrames are to Jupyter Notebooks
- RDDs can come from any data source
    - e.g. text files, a database, a JSON file etc.

# Spark RDDs

- Features of RDDs:
  - Resilient: Ability to withstand failures
  - Distributed: Spanning across multiple machines
  - Datasets: Collection of partitioned data e.g, Arrays, Tables, Tuples etc

# Spark RDDs

- How to create and work with an RDD?
  a. Parallelize an existing collection of objects

**RDD by paralellizing**

```
from pyspark.sql import SparkSession

spark = SparkSession \
    .builder \
    .appName("Python Spark create RDD example") \
    .config("spark.some.config.option", "some-value") \
    .getOrCreate()

df = spark.sparkContext.parallelize([(1, 2, 3, 'a b c'),
                (4, 5, 6, 'd e f'),
                (7, 8, 9, 'g h i')]).toDF(['col1', 'col2', 'col3','col4'])
```

```
df.show()
```

```
+----+----+----+-----+
|col1|col2|col3| col4|
+----+----+----+-----+
|   1|   2|   3|a b c|
|   4|   5|   6|d e f|
|   7|   8|   9|g h i|
+----+----+----+-----+
```

# Spark RDDs

- How to create and work with an RDD?
  a. Parallelize
  b. Work with external data
     - Files in HDFS
     - Objects in Amazon S3 bucket
     - lines in a text file

# RDD Operations

- Transformations - transform the given RDD and return a new one
  a. Basic RDD transformations:
     - map()
     - flatmap()

```
RDD = sc.parallelize(["hello world", "how are you"])
RDD_flatmap = RDD.flatMap(lambda x: x.split(" "))
```

     - filter()

# RDD Operations

- Actions - perform operations that return a value after running a computation on the RDD
- Basic RDD actions:
  a. collect() - returns all the elements of the dataset as an array
  b. take(N) - returns an array with the first N elements of the dataset
  c. first() - returns first element of RDD
  d. count() - returns total number of elements in RDD

# Spark Use Cases

- Streaming data and analyzing data in real time
  - Data ETL
  - Data enrichment - combining streamed data with static data
- Machine Learning  - mllib in PySpark
  - Perform tasks such as ML modeling, market segmentation and sentiment analysis
  - Recommendation engines
- Interactive Analysis

# Spark Use Cases

### Train and test a decision tree classifier

Now we train a DecisionTree model. We specify that we're training a boolean classifier (i.e., there are two outcomes). We also specify that all of our features are categorical and the number of possible categories for each.

In [21]:
```
model = DecisionTree.trainClassifier(training_rdd,
                                     numClasses=2,
                                     categoricalFeaturesInfo={
                                        0: 3,
                                        1: 2,
                                        2: 2
                                     })
```

We now apply the trained model to the feature values in the test set to get the list of predicted outcomines.

In [22]:
```
predictions_rdd = model.predict(test_rdd.map(lambda x: x.features))
```

We bundle our predictions with the ground truth outcome for each passenger in the test set.

In [23]:
```
truth_and_predictions_rdd = test_rdd.map(lambda lp: lp.label).zip(predictions_rdd)
```

Now we compute the test error (% predicted survival outcomes == actual outcomes) and display the decision tree for good measure.

In [24]:
```
accuracy = truth_and_predictions_rdd.filter(lambda v_p: v_p[0] == v_p[1]).count() / float(test_cou
nt)
print('Accuracy =', accuracy)
print(model.toDebugString())
```

```
Accuracy = 0.8020304568527918
DecisionTreeModel classifier of depth 4 with 21 nodes
  If (feature 2 in {0.0})
   If (feature 1 in {0.0})
    If (feature 0 in {0.0,1.0})
     Predict: 1.0
    Else (feature 0 not in {0.0,1.0})
     Predict: 0.0
   Else (feature 1 not in {0.0})
    If (feature 0 in {1.0})
     Predict: 0.0
    Else (feature 0 not in {1.0})
     If (feature 0 in {0.0})
      Predict: 0.0
     Else (feature 0 not in {0.0})
      Predict: 0.0
  Else (feature 2 not in {0.0})
```