



# Apache Spark Boot Camp

# About the Trainer

- Who am I?
- Experience with Big Data?
- Use of Apache Spark

# Introductions - Students

- Name
- Organization
- Role
- Expectations

# Agenda

1. Introduction to Big Data
2. Yarn and Running Spark on Yarn
3. Spark Core, RDDs
4. Spark for Structured Data Processing and Data Sources
5. DataFrame Operations
6. Spark accumulators, broadcast, shared memory, SerDe
7. Spark Streaming
8. Spark MLlib and ML
9. GraphX

**Understanding Spark or any other  
Distributed Computing System as a Tool  
is not sufficient.**

**It is important to understand how  
distributed computing works !!**

# **1. Introduction to Big Data**

If an application is **data intensive** its primary challenge is the quantity of data.

As opposed to compute intensive where the main challenge is CPU speed.

# DIA Examples

- Messaging queues
- Caches
- Search indexes
- Frameworks for batch - Spark
- Frameworks for stream - Spark
- NoSQL DB's

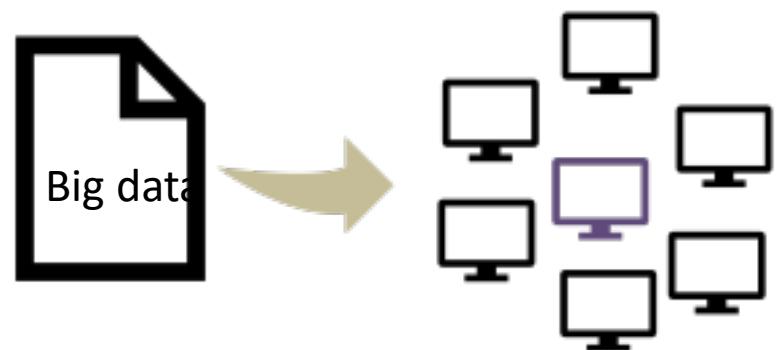
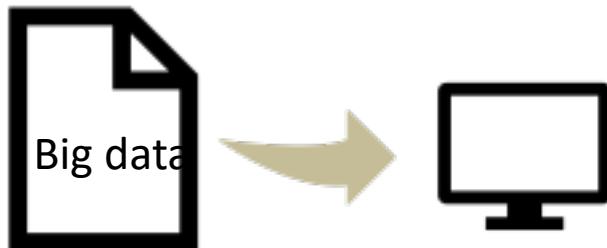
# Data Intensive Applications

Distributed Computing is designed for DIA

A **distributed system** serves to co-ordinate the actions of several computers/systems. This co-ordination is achieved by exchanging **messages**.

# Distributed Computing

vs



# Properties of Distributed System

- Reliability
- Scalability
- Availability
- Efficiency
- Partition Tolerant
- Data Replication
- Consistency
- Failure recovery
- Distributed Transactions

CAP Theorem (Consistency, Availability, Partition Tolerant)

# Properties of Distributed System

- CAP Theorem (consistency, availability, partition tolerant)
- Hashing
- Consistent Hashing
- Consensus Algorithms

Many more .....

# Data Replication and Consistency

- Loss of server holding unique copy of data is an unrecoverable damage
- Important to distribute read/write operations

## Issues with replication:

- Performance: needs more disk, writing to several copies takes more time
- Consistency: same copy of information should be present in all replicas

# Types of Consistency

## Strong Consistency

- Data will get passed to all replicas as soon as a write request comes to one of the replicas, but during this time when the replicas are being updated with new data, response to any subsequent read/write requests by any replicas will get delayed as all replicas are busy in maintaining consistency

# Types of Consistency

## Eventual Consistency

- Make sure that data of each node of the database gets consistent eventually, but will take some time.

Thus offer low latency at a risk of returning stale data

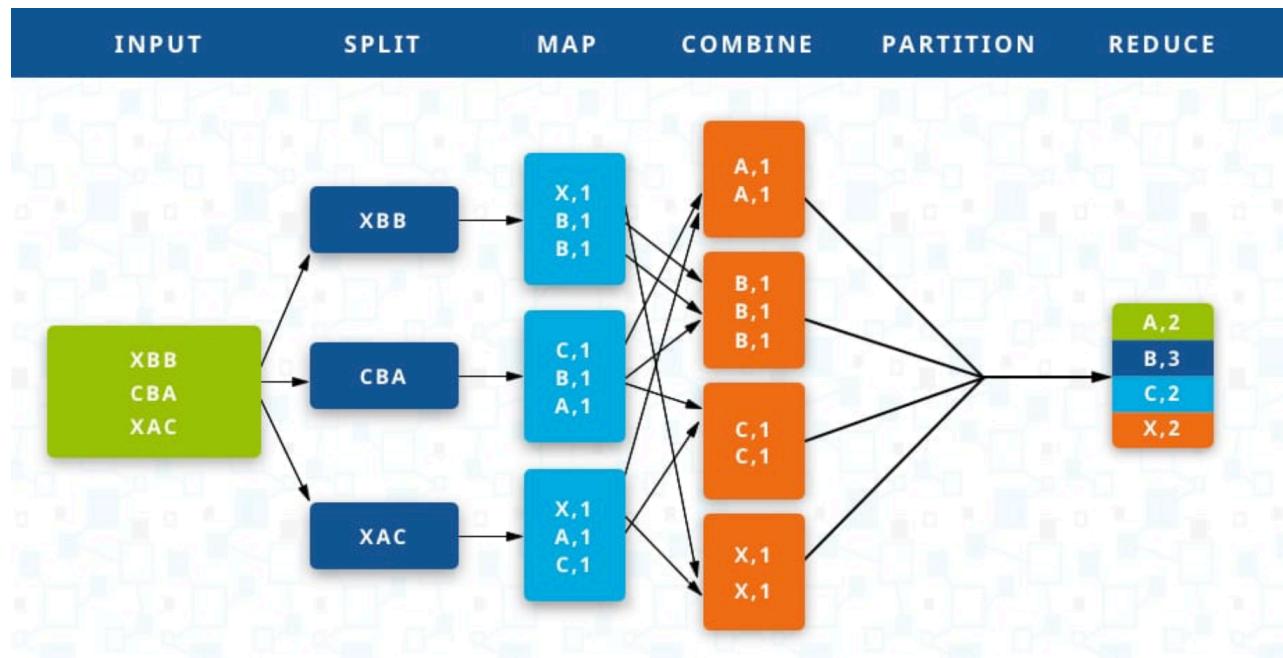
# Failure Management and Recovery

In large distributed systems, failures are inevitable due to network errors, program bugs, out of disk | memory, etc.

- Independence – task handled by one node is independent of others
- Replication – Same data is replicated in multiple nodes
- Multiple master – master is single point of failure, so have multiple masters
- Database writing logs to centralized servers

# Distributed Transactions

**MapReduce** – one of the most common  
algorithm, invented and used by Google in Big  
Table.



# Reliable System

Ability of a distributed system to deliver its services even when one or several of its software or hardware components fail

## Goals

- Performs functions as expected
- Can tolerate user making mistakes
- Performance is sufficient under given load and data volume

## Pre-requisites

- Fault Tolerance
- Redundancy
- Eliminate single point of failure

# Scalability

Ability of a system to continuously evolve in order to support a growing amount of tasks, so to respond to increases in load

- What does your systems current load look like?
- How does your system perform as load increases?
- What are your approaches to coping with load?

# Scalable Applications

## Vertical Scaling



## Horizontal Scaling



# Fault Tolerance

- Fault vs. Failure vs. Error

**Fault** - when a component of a system deviates from expected behavior

**Error** - the part of a system state that deviates from expected output

**Failure** - when the system quits providing the required function or service.

Designing fault tolerant systems become even more important now that we are working in service oriented architecture where we may be using API's and services that our organization has very little control over.

# Fault Tolerance

We assume that systems will never be 100% fault proof so we design our systems to be fault tolerant. These systems are designed to fail safely when faults occur. This can be achieved in different ways.

- Fail backwards
- Fail forward
- Redundant Backups

# Fault Tolerance

Optimal fault tolerant systems often have a mechanism that randomly injects faults into the system to test how the system responds to those faults, and help build confidence in the system.

## Netflix Chaos Monkey

<https://www.gremlin.com/blog/adrian-cockroft-chaos-engineering-what-it-is-and-where-its-going-chaos-conf-2018/>

# Fault Tolerant Systems

## **Not Fault Tolerant**

We were not able to save files (error) to the AWS Simple Storage Service(failure) today due to an error in the code(fault) that retrieve info from the DNS service. This resulted in our application being down for half the day(failure). This made our team rethink the fault tolerance of our application.

# Fault Tolerant Systems

## Fault Tolerant

We have a two tier system that helped our application to respond and recover once S3 came back online causing no downtime in our application. The files were saved to a drive in our corporate cloud and our Kafka cluster had all the S3 request queued and ready to continue where we left off, writing to S3 as soon as S3 came back on line.

Netflix once faced the same problem

<https://www.techrepublic.com/article/aws-outage-how-netflix-weathered-the-storm-by-preparing-for-the-worst/>

""In that instance, Netflix was able to rapidly redirect traffic from the impacted AWS region to datacenters in an unaffected area.""

# Availability

Capacity of a system to limit as much latency as possible, caused by unavailable servers (like crashed servers)

- Failures (crash, unavailability, etc.) must be detected as soon as possible
  - Needs periodically monitoring of system heartbeat  
(typically done by Master node)
- Failover - Quick recovery mechanism
  - Replication and redundancy

# Partition Tolerant

The cluster continues to function even if there is a "partition" (communication break) between two nodes (both nodes might be up, but can't communicate)

## Example:- Gossip Protocol (used by Cassandra)

- Node added to the cluster gets registered with the **gossiper** to receive communication
- The gossiper selects a random node to check if it is alive
- Differentiate between failure detection and long running transactions,  
Cassandra implemented {*Phi Accrual Failure Detection Algorithm*}

# Putting it all together

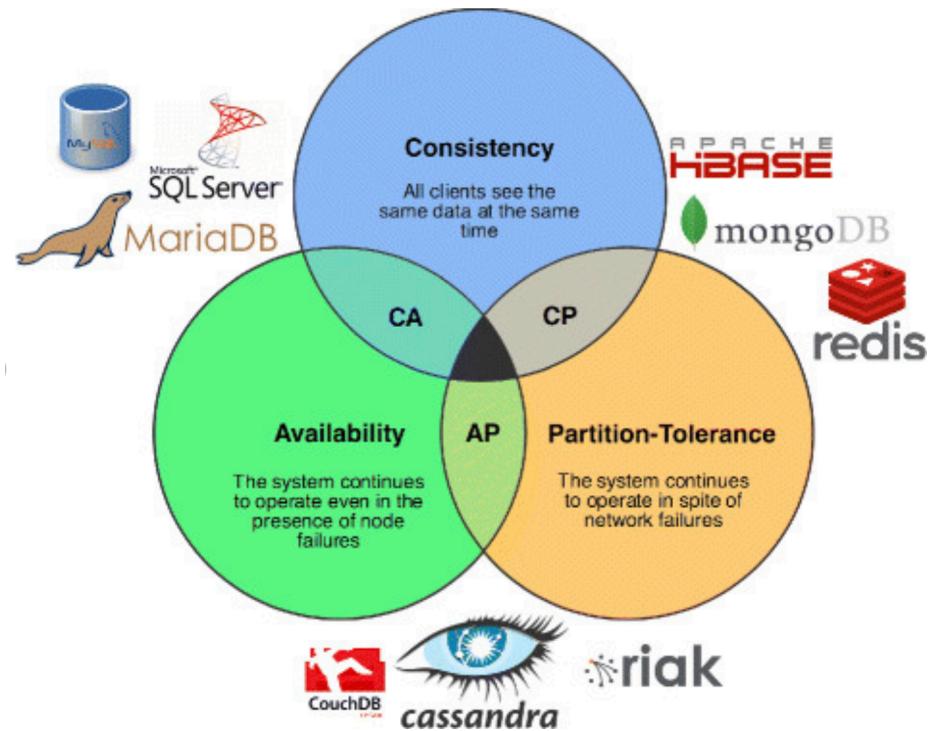
## – CAP Theorem

**CAP** – Consistency, Availability

Partition Tolerant

YOU **CAN'T GET ALL THREE !!!!**

Trade-off between consistency and availability, partition-tolerant has to be dealt anyway



# Ultimately - Efficiency

How do you measure efficiency of a distributed system.

- Response Time or latency
  - Needs periodically monitoring of system heartbeat  
(typically done by Master node)
- Throughput (bandwidth)
  - Replication and redundancy

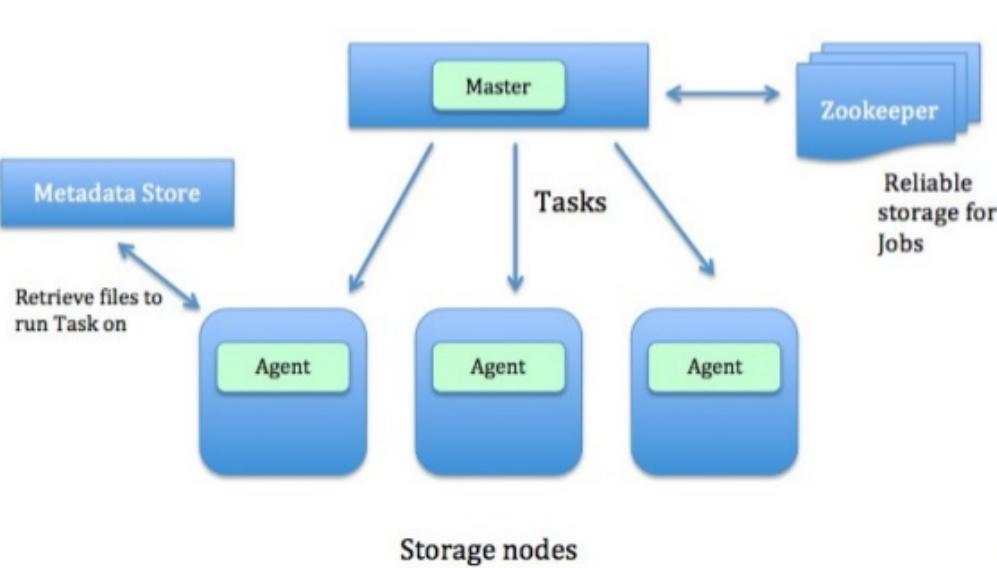
# More on Distributed System

- Master Slave Architecture
- Partitioning
- Consistent Hashing
- Distributed Consensus

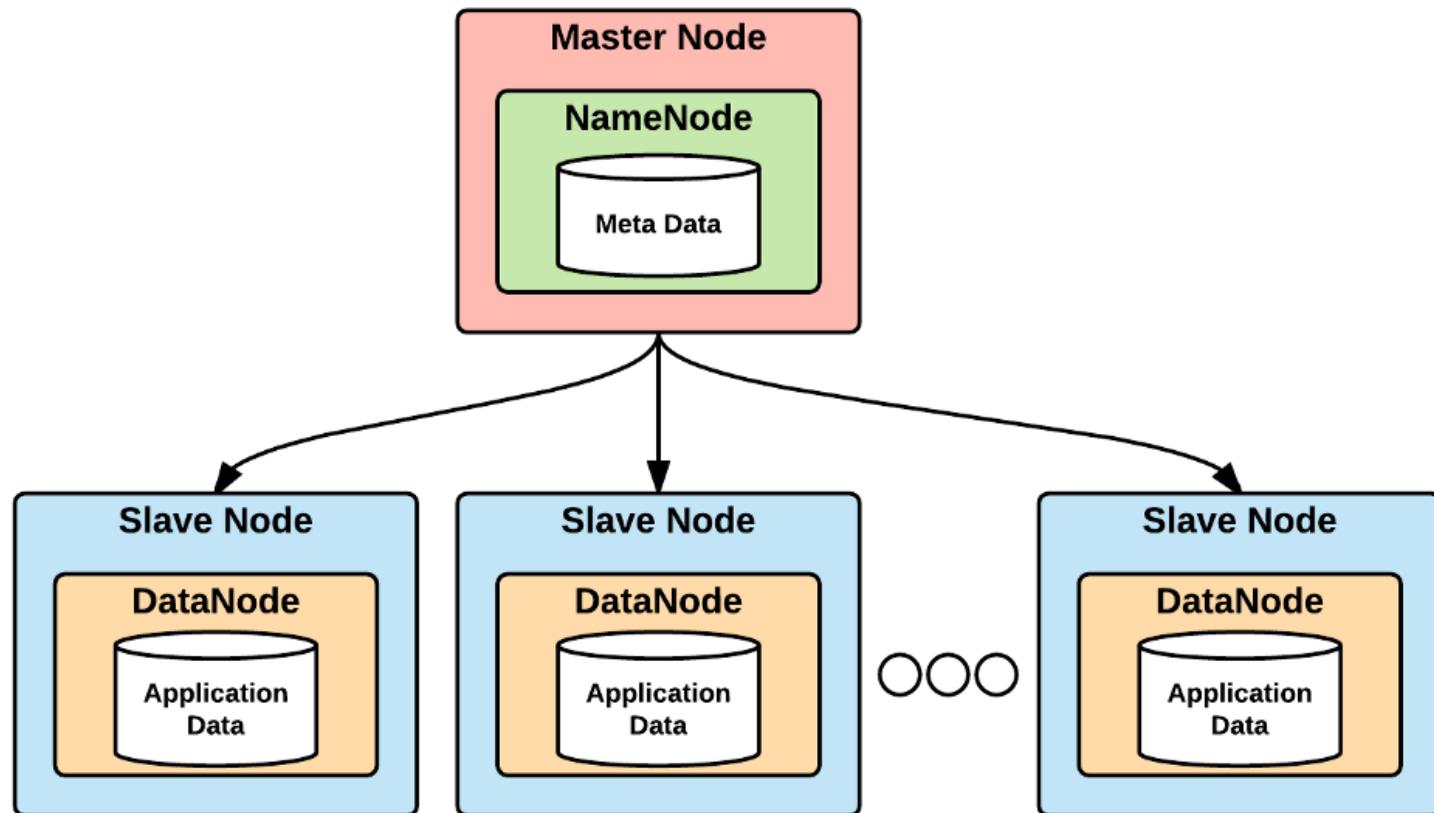
# Master Slave Architecture

# Storage

## Master – slave architecture



# Compute



# Partitioning

# Partitioning

## Basics

- Divide data into multiple chunks
- Place chunks on different nodes
- Both read and write load gets distributed
- Chunks are called shards or partitions or vnodes etc.

## When you add more nodes in the cluster:

- New node will hold a new partitions with new data

# Partitioning

## Pre-requisites of partitioning

- **Fair partition** – to ensure uniform load
- In skewed partitioning, a few partitions will handle most of the requests
- Highly loaded partitions becomes bottleneck for the system – called **hotspots**
- Can't be round-robin – loses track of where data goes
- quick lookup capability

*Each data record has an **Id** or **Key***

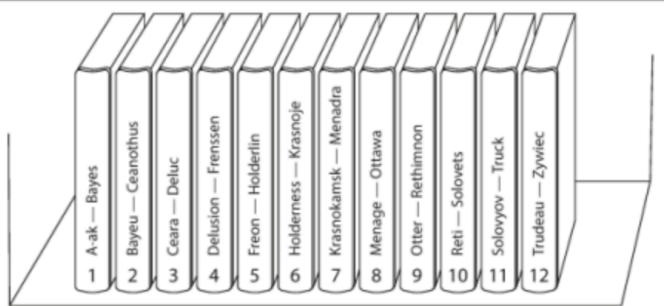
# Partitioning Strategies

## 1. Key Range Partitioning

Divide the entire keyset into continuous ranges and assign each range to a partition.

**Advantage:** keys can be kept in a sorted manner within a partition. So, range queries like time series queries would have a better performance with this strategy.

**Downside:** Most data distribution is non-uniform, thus partitions doesn't get uniform range leading to hotspots



E.g. a bookshelf

# Partitioning Strategies

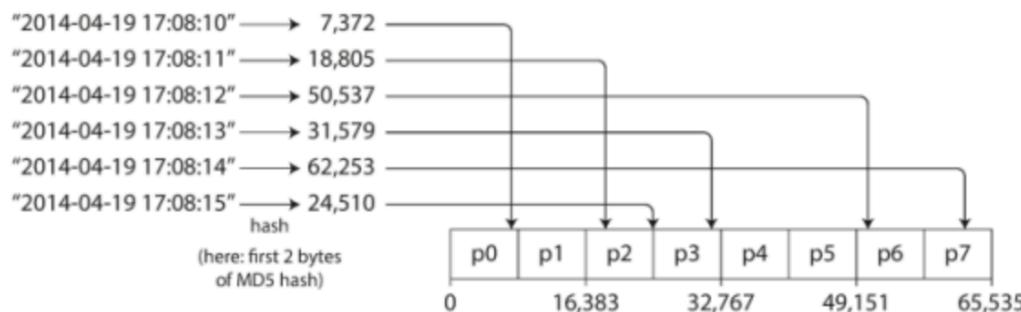
## 2. Key Hash Partitioning

Find hash function of the key and take “mod N” where N is total number of partitions

Partition number =  $\text{Hash(key)} \text{ (Mod } N\text{)}$

**Advantage:** quick O(1) retrieval of keys; solves hotspot problems by randomness.

**Downside:** What happens when number of partitions are changed ? – “mod N” changes and all data previously assigned to “i” partition is no longer in “i” partition. Thus there is a need of re-partition m



# Consistent Hashing

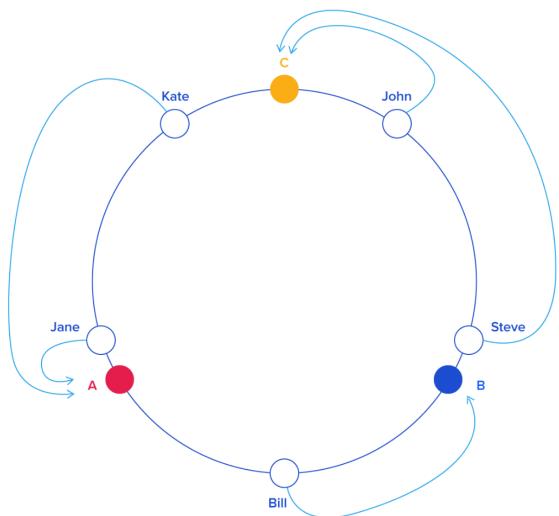
# Consistent Hashing

## How to rebalance partitions when a new node is added/removed

Use a distribution scheme that does not depend directly on the number of partitions

- Assign partitions (servers) a position on an abstract circle, or hash ring by pseudo-randomly assigning them angles (from 0-360)
- Calculate “hash(key)” in “radians or degrees” that has a range from 0-360
- Keys for both data and partitions are on the same circle
- Define an association rule – data will belong to the partition whose key is closest, in a counterclockwise | clockwise direction

# Consistent Hashing



## Minimize re-partition

### 1. Operation - Add a node

If a 4<sup>th</sup> node D is added between A and C, only node A needs to be re-partitioned. Rest un-touched

### 2. Operation - Delete a node

If node A dies, only node B needs to be re-partitioned. Rest un-touched

\*\* Trick for uniform load - Assign not one, but many labels (angles) to each server

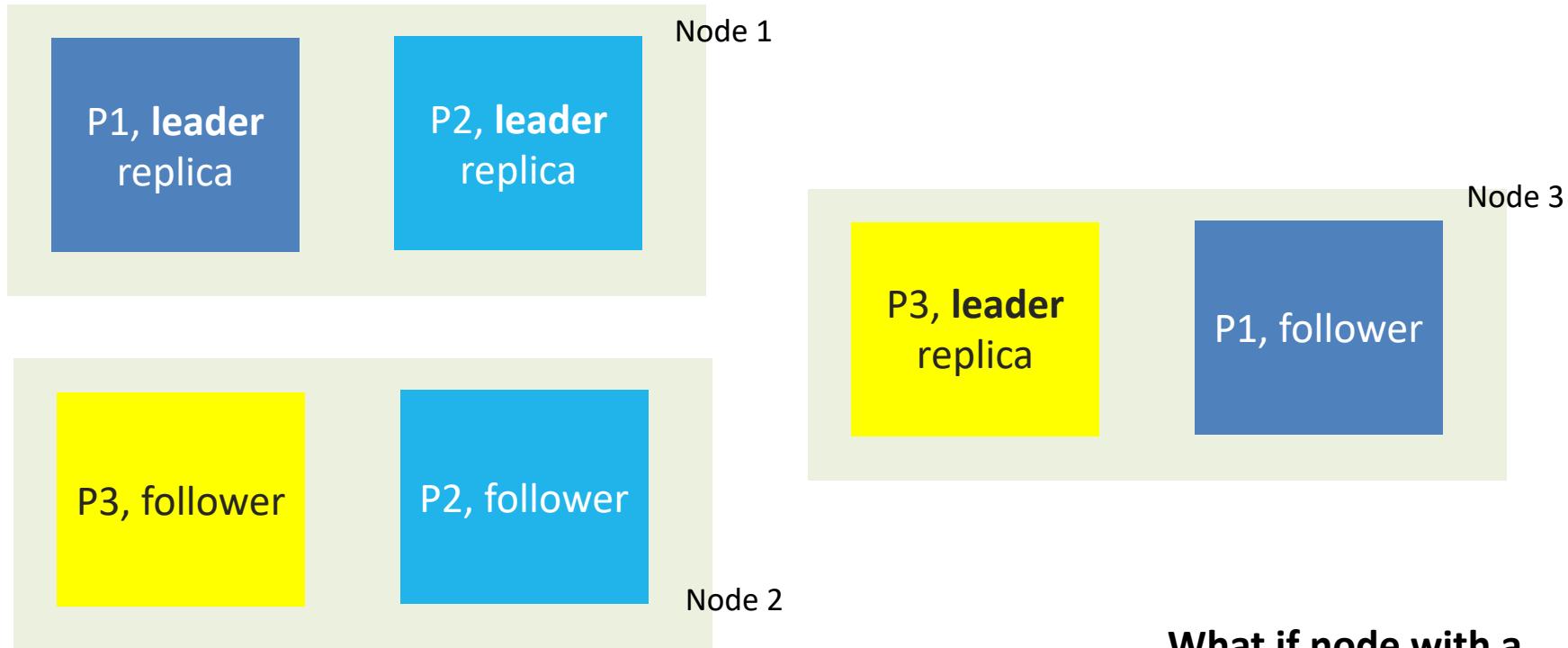
KEY	HASH	ANGLE (DEG)	LABEL	SERVER
"john"	1632929716	58.7	"C"	C
"kate"	3421831276	123.1	"A"	A
"jane"	5000648311	180	"A"	A
"bill"	7594873884	273.4	"B"	B
"steve"	9786437450	352.3	"C"	C

# Distributed Consensus

All distributed system use consensus algorithm

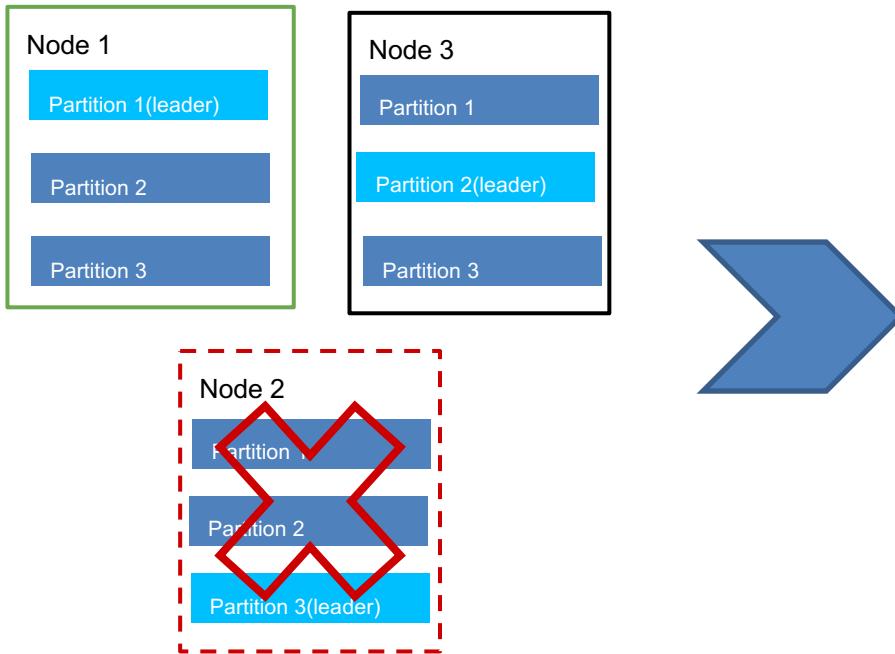
# Example: Kafka

Data has 3 partitions; and 2 replicas (1 leader and 1 follower)  
Leader is important because you want eventual consistency



What if node with a  
leader dies ??

# Leader Replica Election



Node 2 dies  
With leader of partition 3

- Need a new leader of partition 3
- Could be Node 1 or Node 3
- But which one is the in-sync replica
- You do not want to **elect** a stale replica as the leader



Solved by **distributed consensus**  
**Raft, Paxos, Quorum**

# Distributed Consensus

- **PAXOS**

Sequence of proposals (sent by proposer and received by acceptor), which may or may not be accepted by acceptors. If a proposal isn't accepted, it fails

E.g. Cassandra

- **RAFT**

Alternative to Paxos, by separation of logic. Distribute a state machine across cluster, ensuring each machine agrees upon same state machine

E.g. blockchain, etcd

- **QUORUM**

Take vote

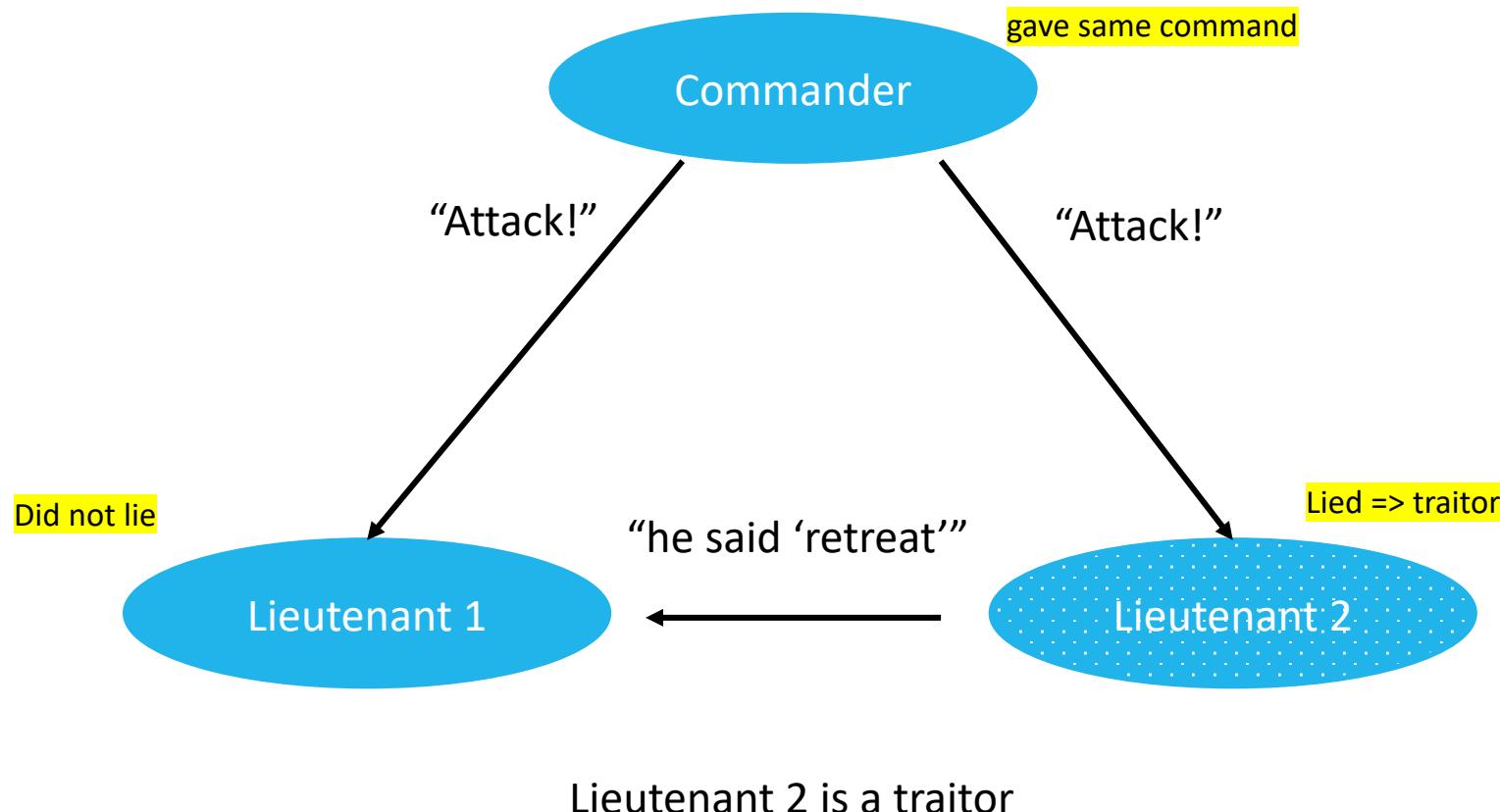
E.g. Kafka, Zookeeper

# Choose # of Replicas

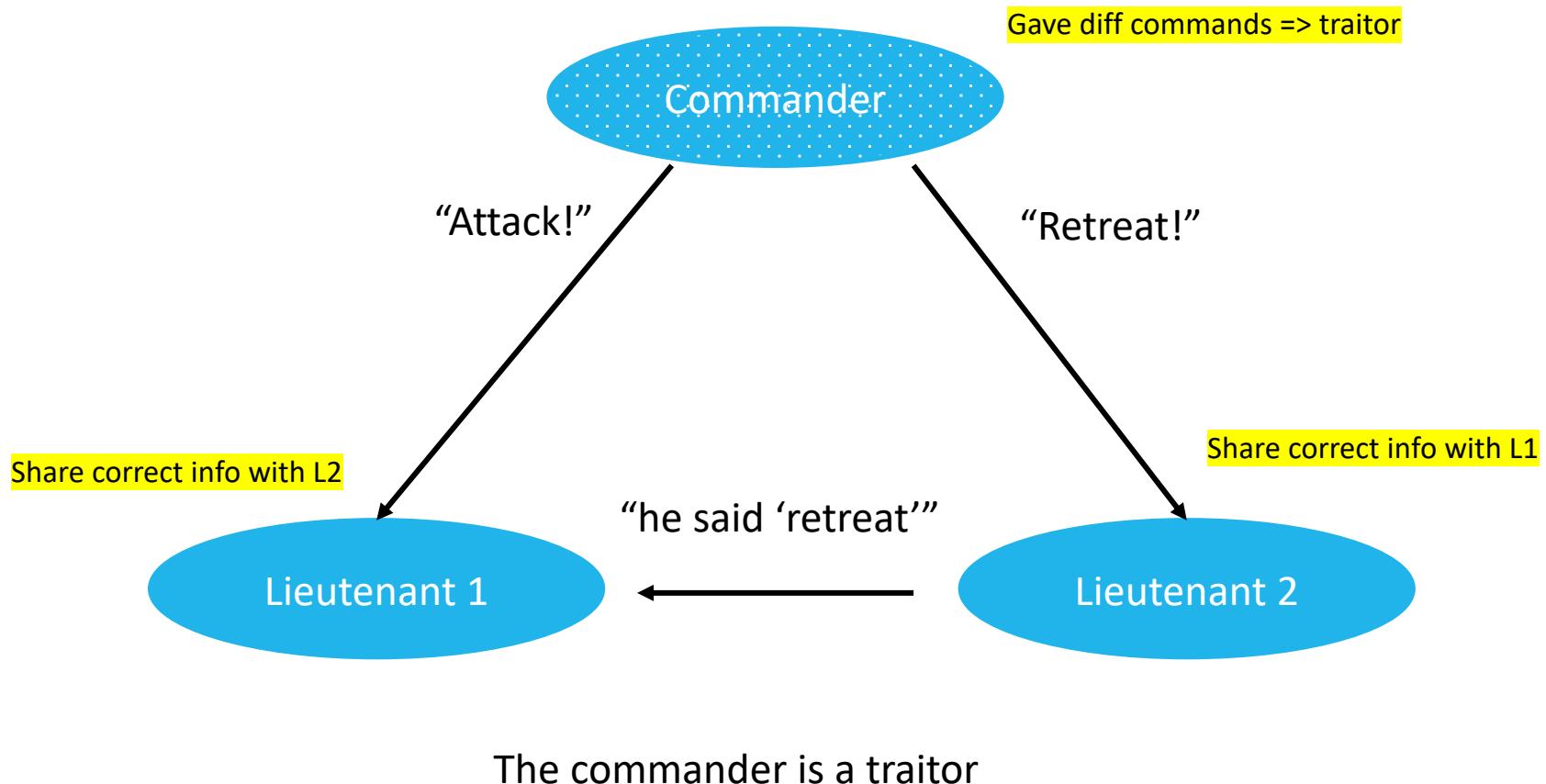
- Minimum: 3
  - Most consensus-based system relies on majority quorum.
    - To tolerate  $f$  failures,  $R = 2f+1$
    - To tolerate **Byzantine fault tolerance**,  $R = 3f+1$
- \*\* Byzantine fault tolerance – nodes can give false information
- \*\* Yet another consensus algorithm – used heavily by **blockchain**

# The Byzantine Generals Problem

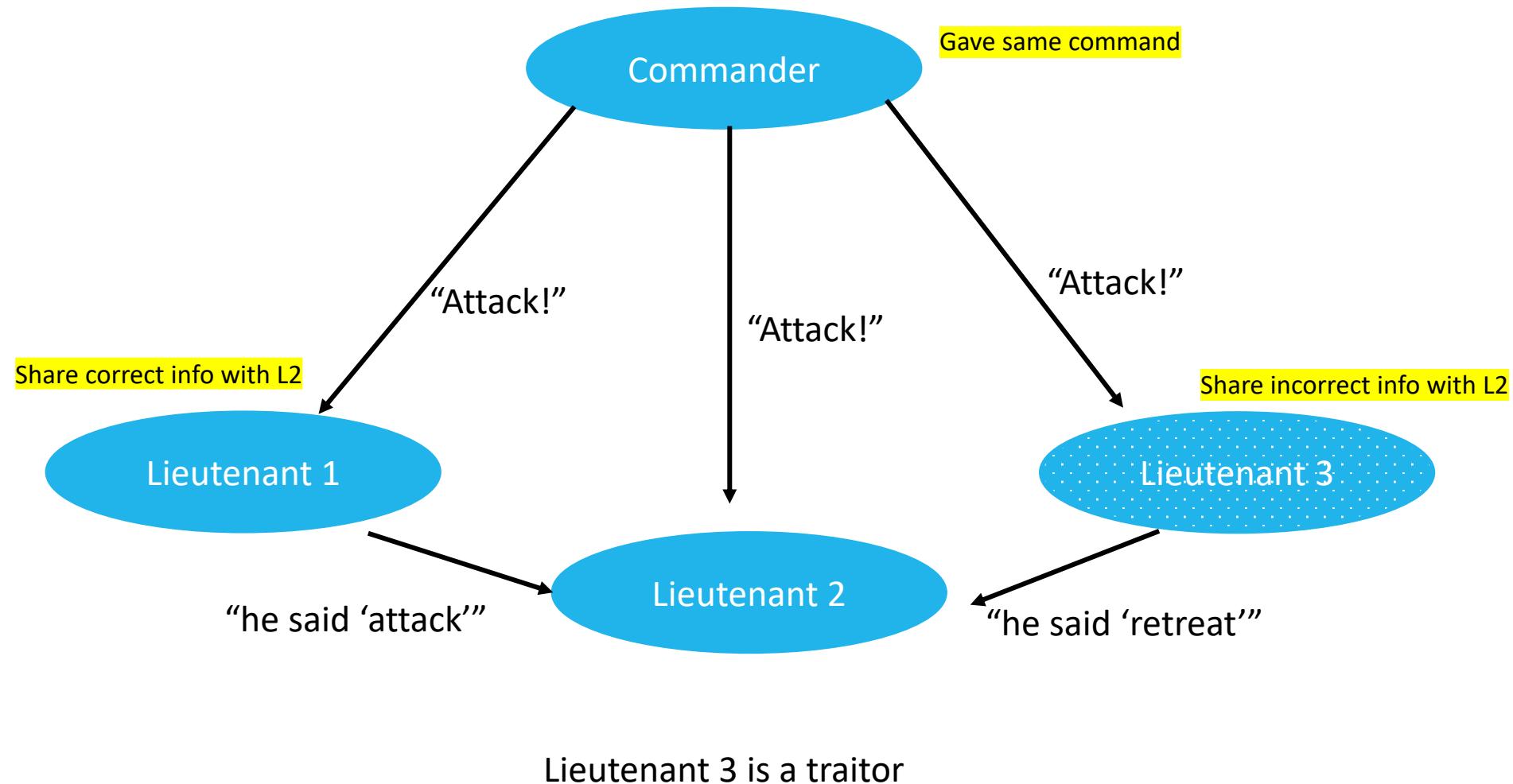
Find who is the traitor by consensus



# The Byzantine Generals Problem



# The Byzantine Generals Problem



# Distributed Consensus for Reliability

Disaster happens

- Hard drive fails
- Datacenter down
- Processes fails

Group of process reliably agree on questions such as:

- Which process is the leader of a group of processes?
- What is the set of processes in a group?
- Has a message been successfully committed to a distributed queue?
- What is a value in a datastore for a given key?

# Distributed Consensus for Reliability

Several processes in a distributed system need to be able to form a consistent view of a critical piece of configuration

- This is one of the most important concepts in distributed computing

# Resource

One of the best books on Distributed Computing Concepts

Designing Data Intensive Applications

<http://shop.oreilly.com/product/0636920032175.do>

Available in Amazon for \$35

# Understanding

- Data
- Data Analytics
- Hadoop

# Data Analytics

- Process of applying qualitative and quantitative techniques
- Goal of providing valuable insights
- Exploratory Data Analysis (EDA)
  - explore data with the intention of finding patterns in the data and relationships between various elements of the data
- Confirmatory Data Analysis (CDA)
  - provide an insight or conclusion on a specific question based on hypothesis and statistical techniques

# Example: Super Market

- A Super Market having a lot of cereals, cookies, vegetables, sodas

# Super Market



Cereal

Soda

Pickles

Cat Food

- Record all transactions into a database or maybe an excel spreadsheet

# Daily report of transactions



Receipt No	Date	ItemId	Quantity	Cost\$	Sale\$
123	03/01	24	3	9	12
123	03/01	25	2	6	7
124	03/02	12	4	7	9
125	03/02	25	1	3	4

# Some reports

Item	Day Of Week	Quantity
Milk	Sunday	1244
Bread	Monday	245
Milk	Monday	190

Item1	Item2	Quantity
Milk	Eggs	360
Bread	Cheese	335
Onions	Tomatoes	310

# The two conclusions we have are:

- Milk is bought more on weekends, so it's better to increase the stock in quantity and variety of Milk products over weekends.
- Milk and Eggs are bought by more customers in one purchase followed by Bread and Cheese. So, we could recommend that the store realigns the aisles and shelves to move the Milk and Eggs closer to each other.

# Inside the Data analytics process

- Data analytics applications involve more than just analyzing data.
  - Before any analytics can be planned, there is also a need to invest time and effort in collecting, integrating, and preparing data, checking the quality of the data and then developing, testing, and revising analytical methodologies.

# Inside the Data analytics process

- Data analysts and scientists can explore and analyze the data using statistical methods like SAS or machine learning models using SparkML.
- Data governance becomes a factor too to ensure the proper collection and protection of the data.
- Data quality processes must be established to make sure the data being collected and engineered is correct and will match the needs of the Data scientists

# 4 Vs of Big Data

## Variety of Data

+4 Billion Videos  
400 Million tweets  
500 Million wearables  
30 Billion pieces of content

## Velocity of Data

TBs of data from financial firms  
20 Billion connections  
Millions of Weather sensors  
100 sensors per car

Big  
Data

## Volume of Data

2.5 EBs per day  
average of 100TBs per company  
+40,000 EBs of data

## Veracity of Data

33% dont trust the data  
Trillions Lost due to bad analytics

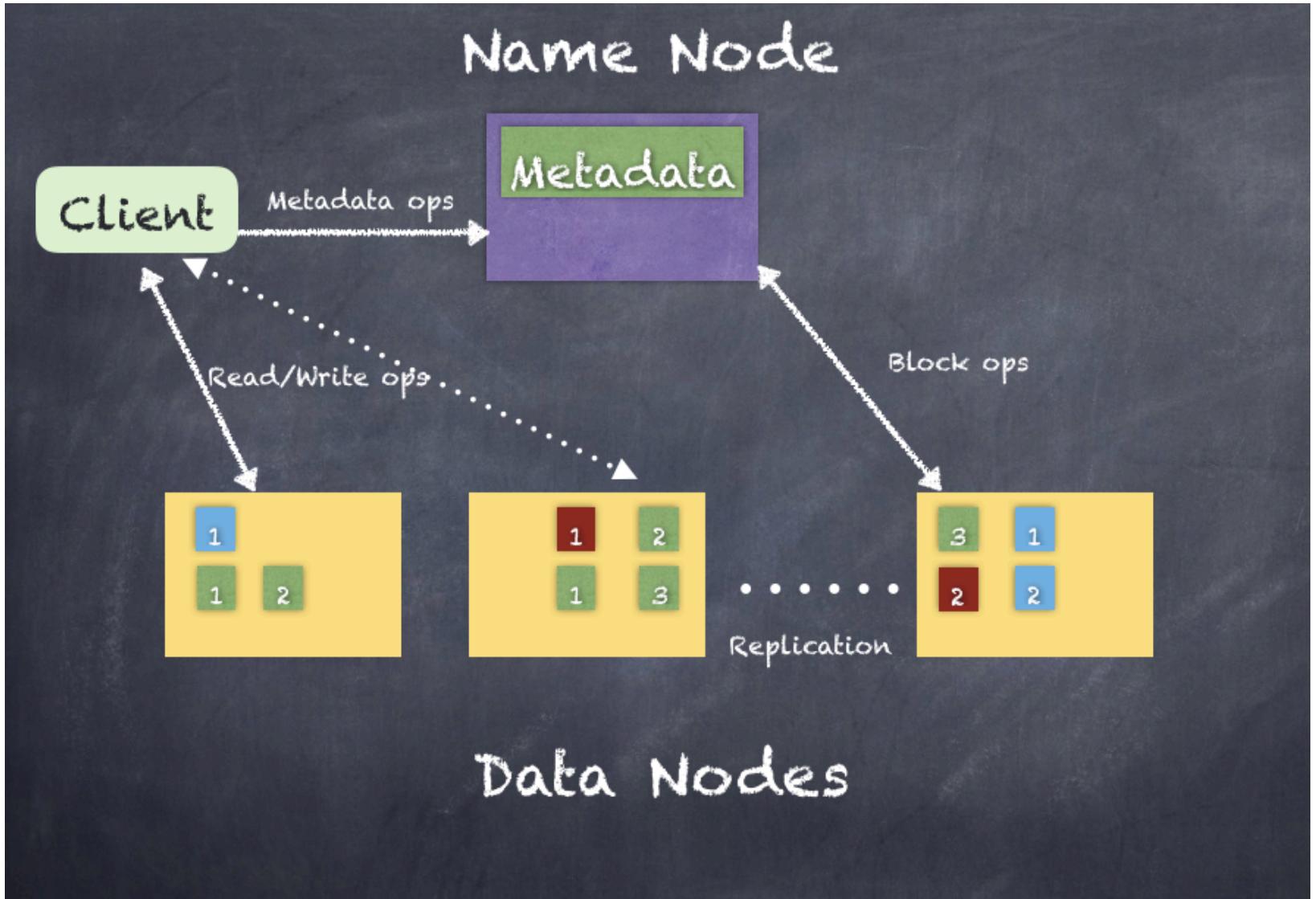
# Distributed computing using Apache Hadoop

- Distributed Storage
  - Hadoop Distributed File System (HDFS)
- Distributed Computing
  - Map Reduce

# HDFS

- Name Node
  - Master Node which manages all metadata of the file system and the data blocks across the Data Nodes
- Data Node
  - Worker Nodes which contain the data in the form of blocks and constantly report up to the Name Node.

# Distributed Storage (HDFS)

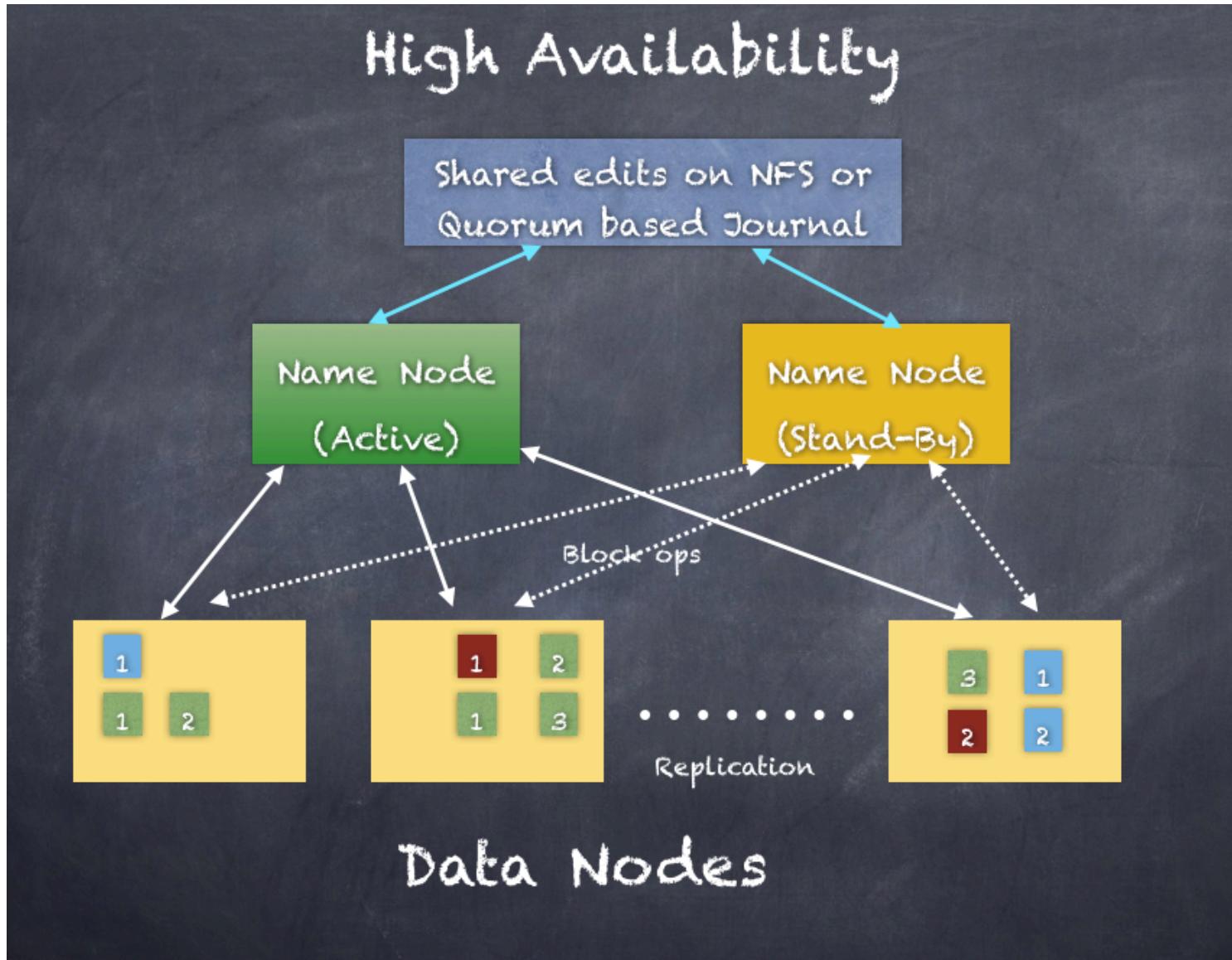


# HDFS High Availability

- HA works by having two Name Nodes in an active-passive mode
- One Name Node is active and other is passive.

When the primary NameNode has a failure, the passive Name Node will take over the role of the Master Node

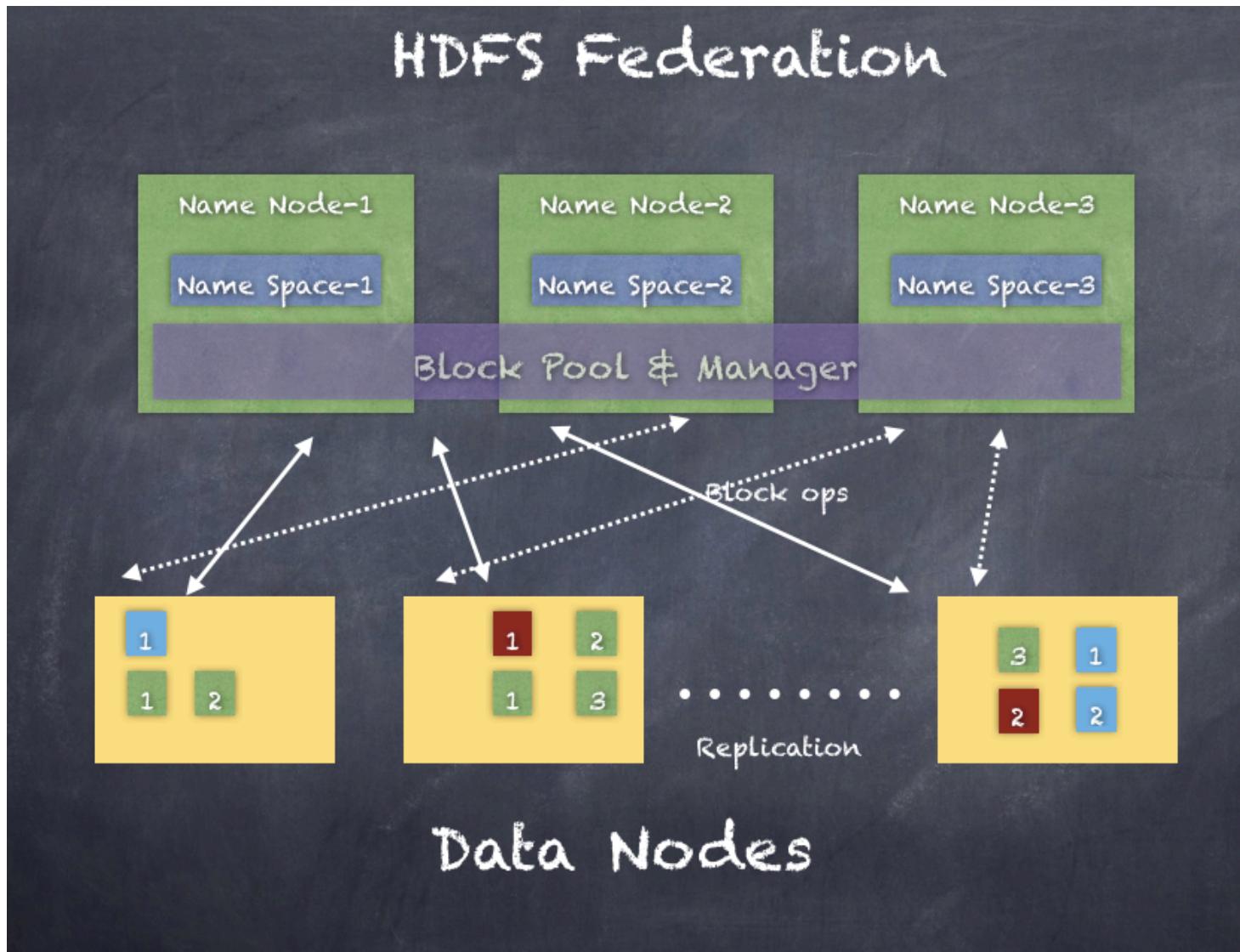
# High Availability



# HDFS Federation

- The HDFS Federation is a way of using multiple Name Nodes to spread the filesystem namespace over. Unlike the first HDFS versions, which simply managed entire cluster using a single NameNode which does not scale that well as the size of the cluster grows.
- HDFS Federation can support significantly larger clusters and horizontally scales the name node or name service using multiple federated name nodes

# HDFS Federation

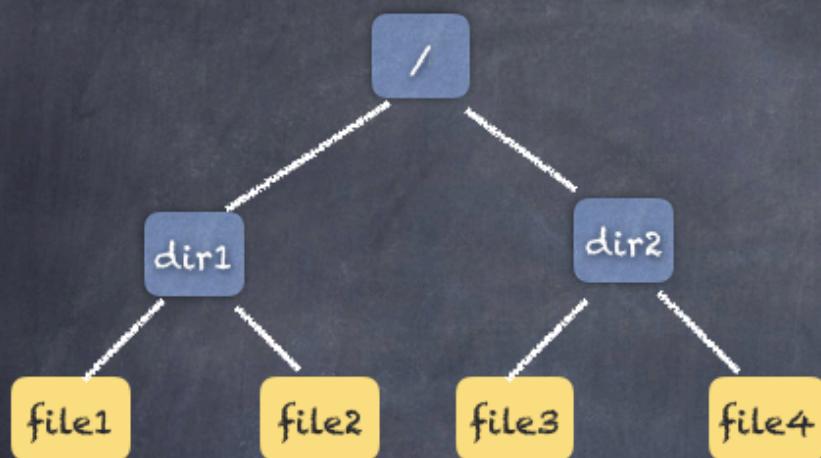


# HDFS Snapshots

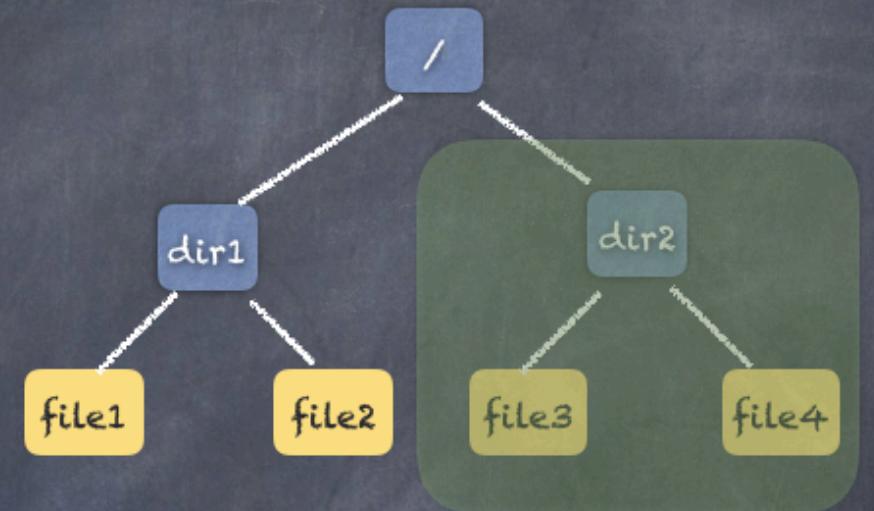
- Using Snapshots, you can take a copy of directories seamlessly using the NameNode's metadata of the data blocks.
- Snapshot creation is instantaneous and doesn't require interference with other regular HDFS operations.

# HDFS Snapshots

## HDFS Snapshot



File System of dir1 and dir2

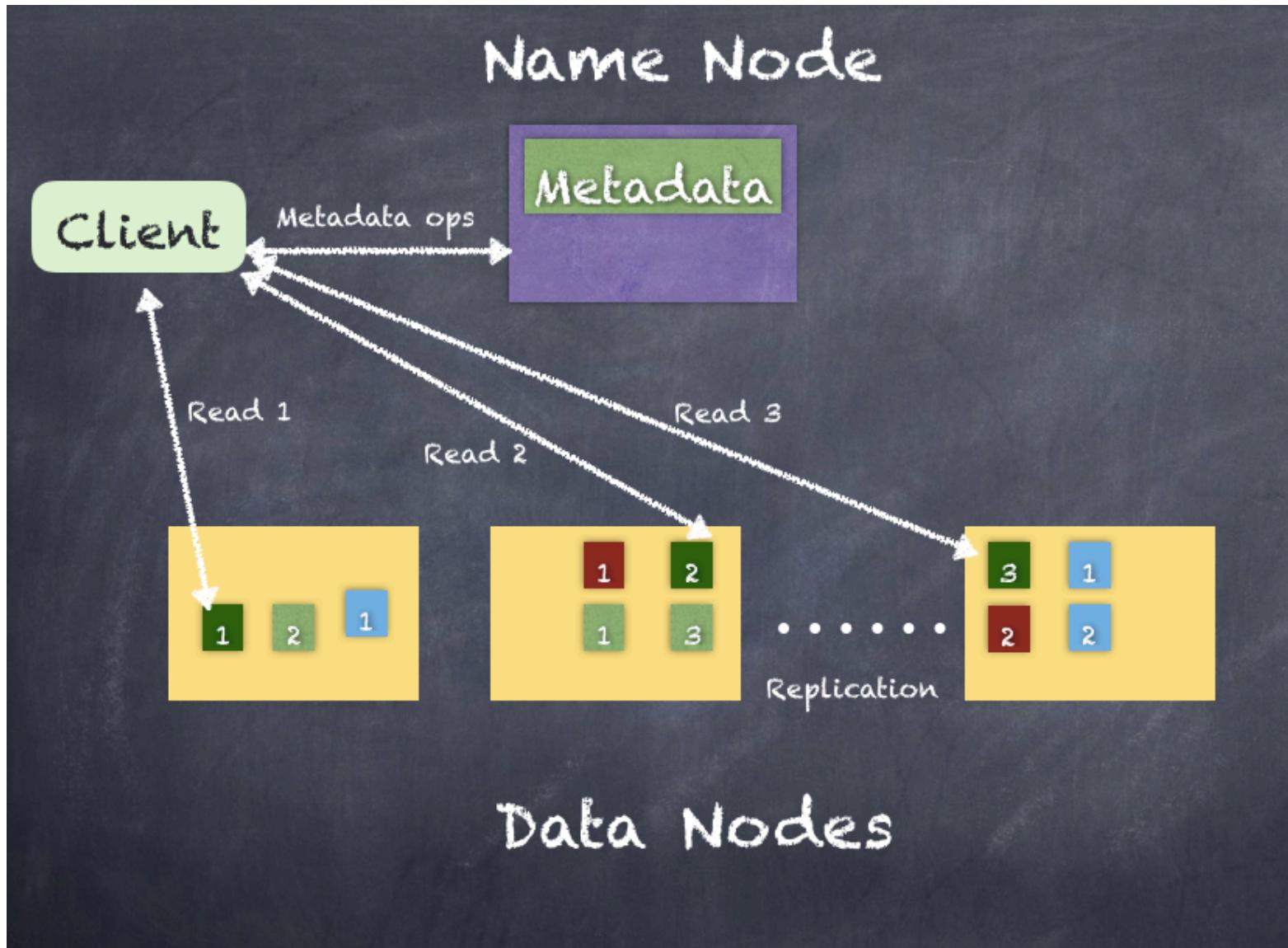


Snapshot of dir2

# HDFS Reads

- Client connects to the NameNode and asks about a file using the name of the file.
- NameNode looks up the block locations for the file and returns the same to the client.
- The client can then connect to the DataNodes and read the blocks needed. NameNode does not participate in the Data transfer.
- If a DataNode fails in the middle, then client gets the replica of the block from another DataNode.

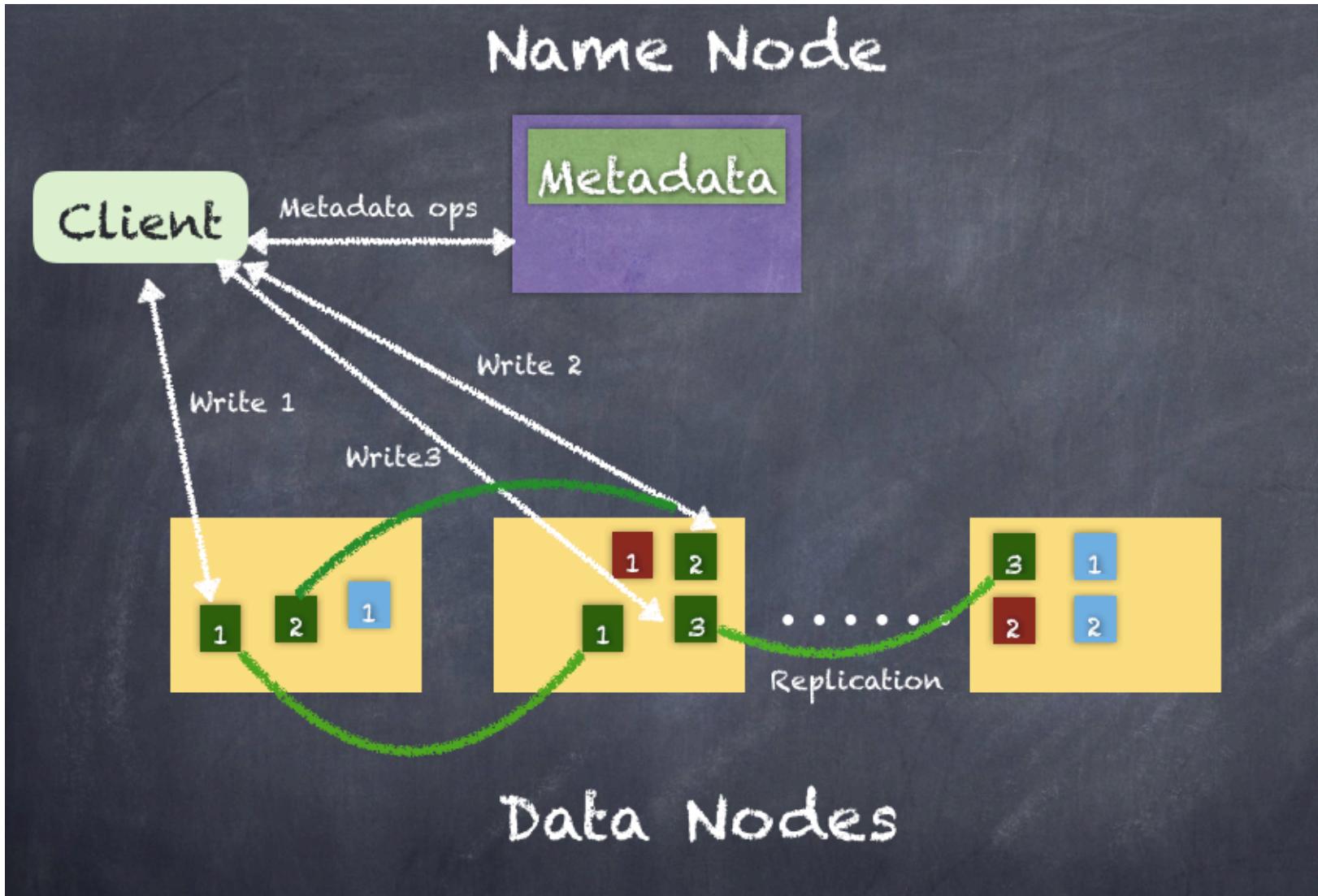
# HDFS Reads



# HDFS Writes

- Client connects to the NameNode and asks the NameNode to let it write to the HDFS.
- NameNode looks up information and plans the blocks, the Data Nodes to be used to store the blocks and the replication strategy to be used. NameNode only tells the client where to write.
- First, the client gets the locations and then writes to the first DataNode. The DataNode that receives the block replicates the block to the DataNodes that should hold the replica copy of the block.
- If a DataNode fails in the middle, then the block gets replicated to another DataNode as determined by the NameNode

# HDFS Writes



# Map Reduce Framework (MR)

- Map Reduce framework enables you to write distributed application to process large amounts of data from a file-system such as HDFS in a reliable and fault-tolerant manner.
- When you want to use the Map Reduce Framework to process data, it works through the creation of a job, which then runs on the framework to perform the tasks needed

# Example

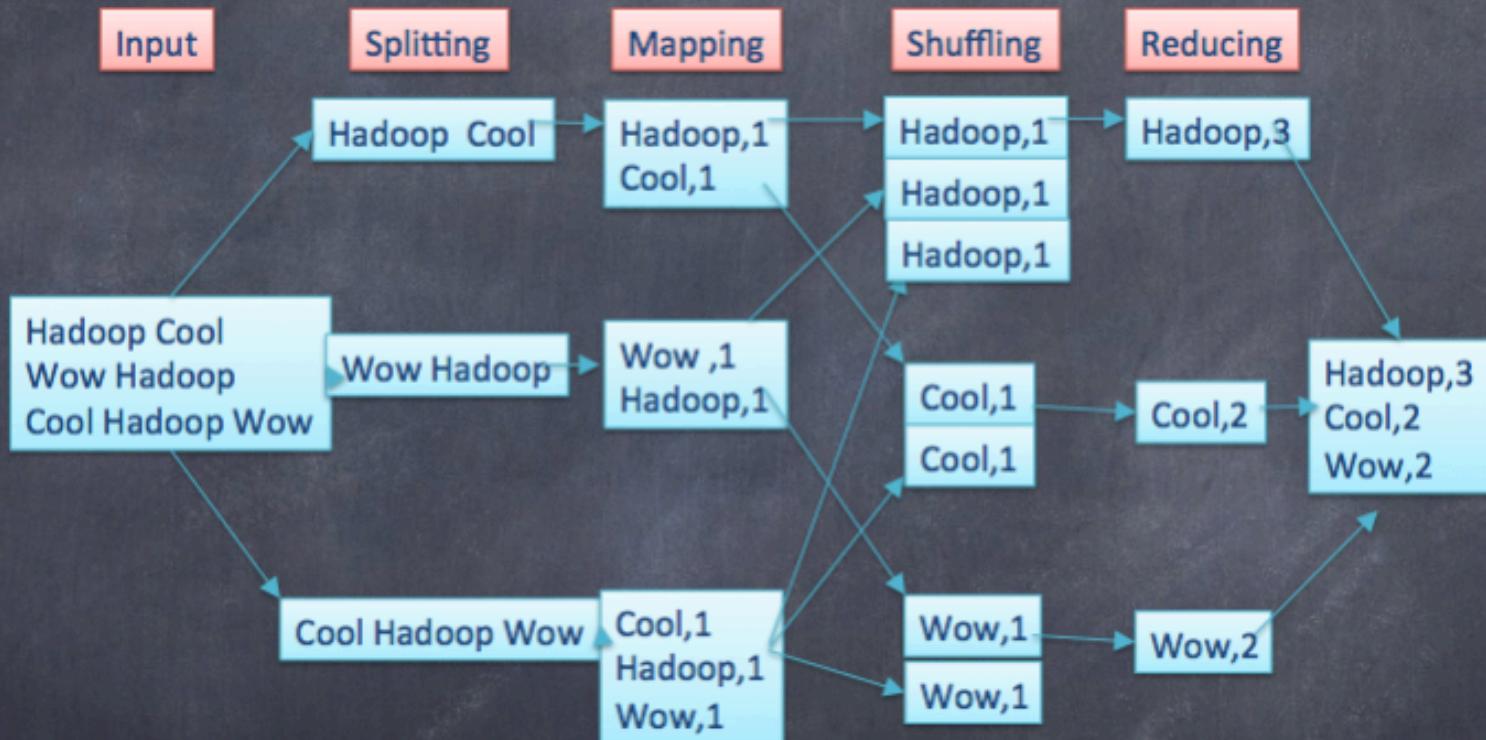
- Let us sort the fruits in a fruit market by type and put them into separate baskets

# Sorting Fruits

Sorting the fruits by type

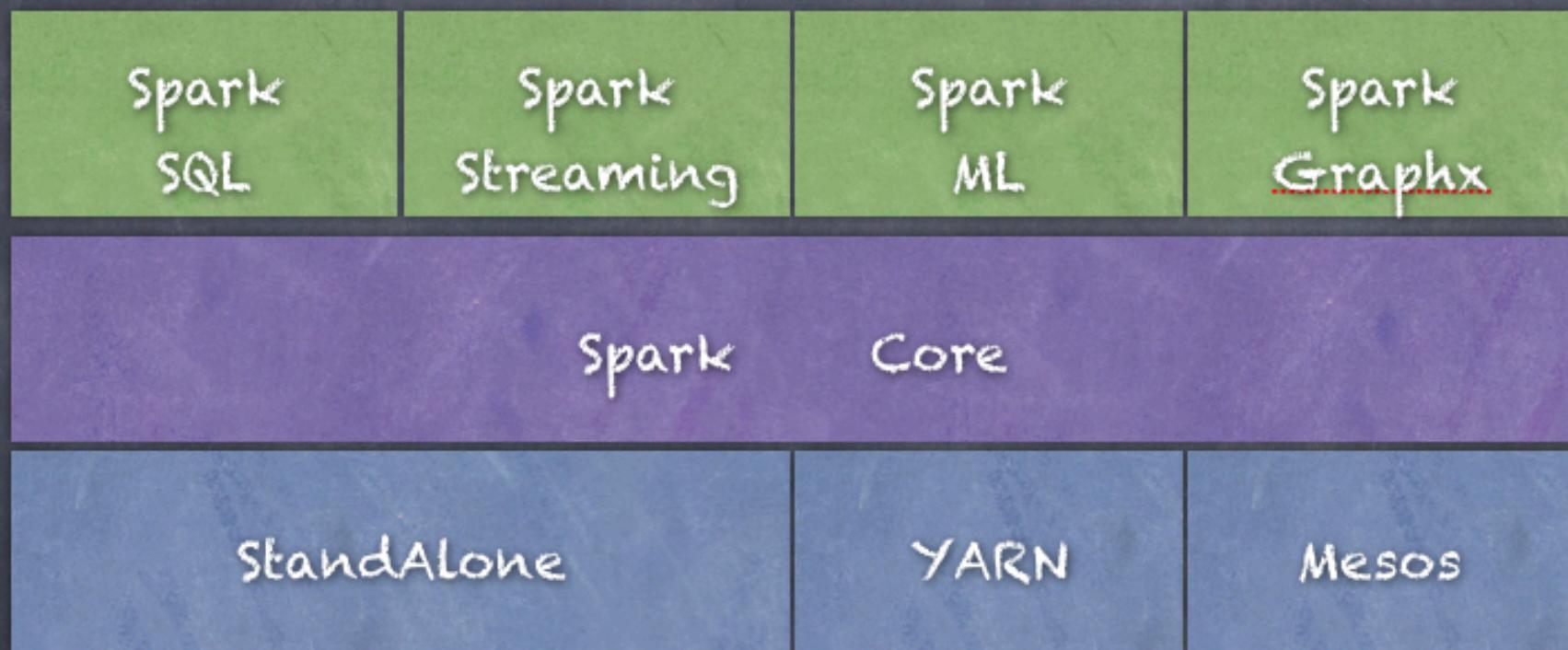


# Map Reduce



# Spark

# Components



# Spark Core

- Spark Core is the underlying general execution engine for spark platform that all other functionality is built upon.
- Spark Core contains **basic Spark functionalities** required for running jobs and needed by other components. It provides in-memory computing and referencing datasets in external storage systems, the most important being the **Resilient**
- **Distributed Dataset (RDD).**
- Spark Core contains logic for accessing various filesystems, such as HDFS, Amazon S3, HBase, Cassandra, Relational Databases, and so on.
- Spark Core also provides fundamental functions to support networking, security, scheduling, and data shuffling

# Spark SQL

- Spark SQL is a component on top of Spark Core
- **SchemaRDD**, provides support for structured data.
- Support of SQL by Spark and Hive QL
- **DataFrames**
- Spark SQL supports reading and writing data to and from various structured formats and data sources, files, parquet, orc, relational databases, Hive, HDFS, S3, and so on.
- Spark SQL provides a query optimization framework called Catalyst to optimize all operations to boost the speed
- Spark SQL also includes a Thrift server, which can be used by external systems to query data through Spark SQL using classic JDBC and ODBC protocols.

# Spark Streaming

- Performs **streaming analytics** by ingesting real-time streaming data from various sources such as HDFS, Kafka, Flume, Twitter, ZeroMQ, Kinesis, and so on.
- Uses micro-batches of data to process in chunks and using a concept known as **Dstreams** (discretized streams)
- Spark Streaming **operates on RDDs** applying transformations and actions as regular RDDs in Spark Core API.
- Spark Streaming can be combined with other Spark components in a single program, unifying real-time processing with machine learning, SQL, and graph operations.

# Spark GraphX

- GraphX is a **distributed graph processing framework** on top of Spark.
- Graphs are data structures comprising vertices and edges
- GraphX provides functions for building graphs, represented as **Graph RDDs**. It provides an API for expressing graph computation that can model the user-defined graphs by using Pregel abstraction API.
- GraphX contains implementations of the most important algorithms of graph theory, such as page rank, connected components, shortest paths, SVD++, and others.

# Spark ML

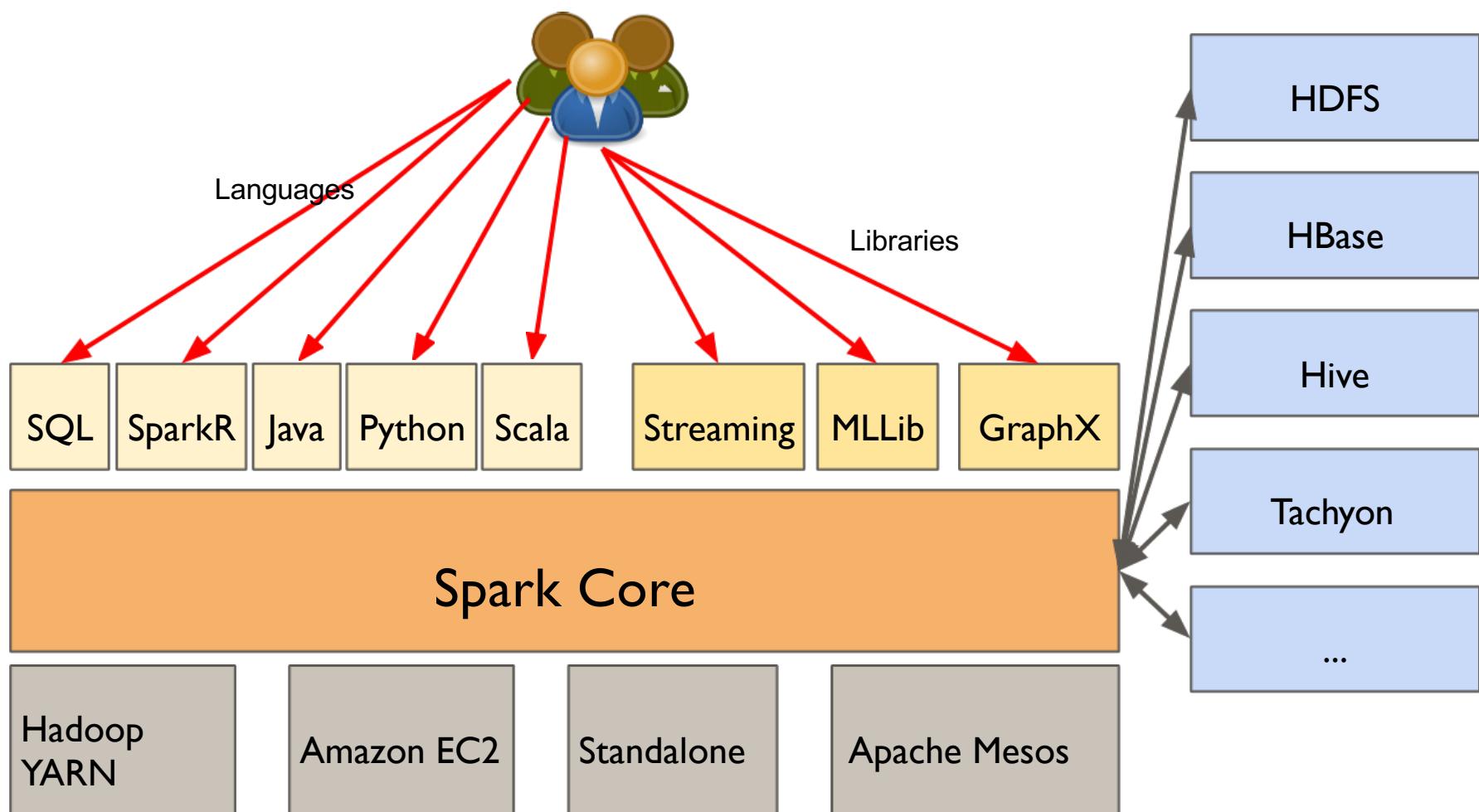
- MLlib is a **distributed machine learning framework** above Spark Core and handles machine-learning models
- Spark MLlib is a library of machine-learning algorithms providing various algorithms such as logistic regression, naïve Bayes classification, Support Vector Machines (SVMs), decision trees, random forests, linear regression, Alternating Least Squares (ALS) and k-means clustering.
- Spark ML integrates very well with Spark Core, Spark Streaming, Spark SQL, and GraphX

# Apache

A fast and general engine for large-scale data  
processing.

- Really fast MapReduce
  - 100x faster than Hadoop MapReduce in memory,
  - 10x faster on disk.
- Builds on similar paradigms as MapReduce
- Integrated with Hadoop

# Spark Architecture



# **2. Deploying & Understanding Apache Spark Architecture**

# YARN

YARN (Yet Another Resource Negotiator)

General Purpose, Distributed, Application Management Framework

- Resource Manager
- Node Manager

# Need for YARN

## Hadoop 1.0

- Single use system
- Capable of running only MR

**HADOOP 1.0**

**MapReduce**

(cluster resource management  
& data processing)

**HDFS**

(redundant, reliable storage)

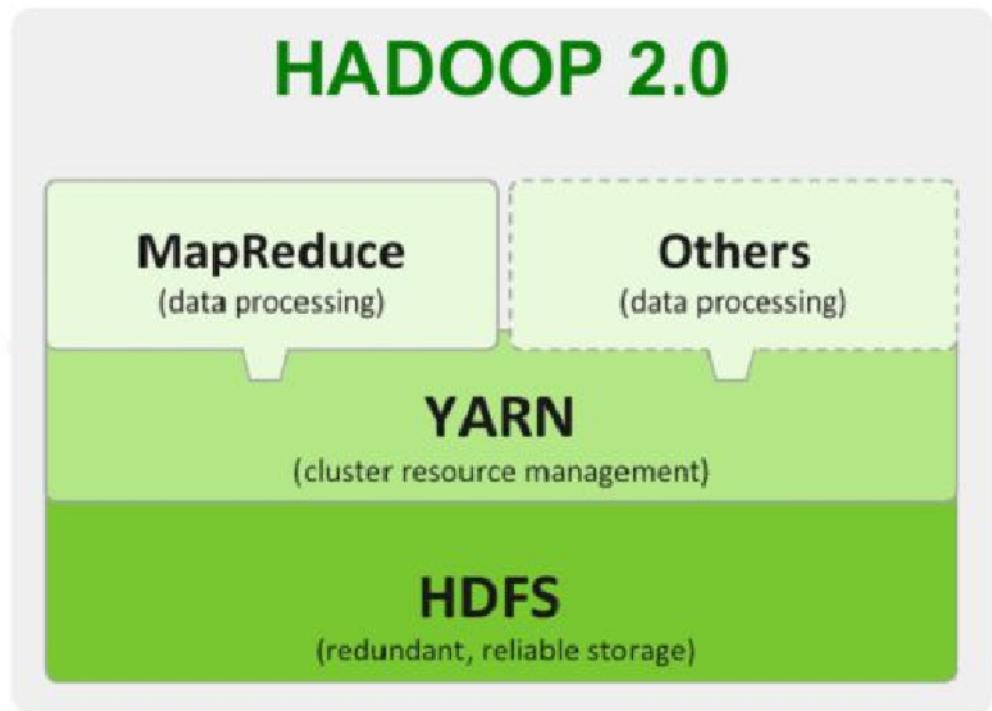
# Need for YARN

- **Scalability**
  - 2009 – 8 cores, 16GB of RAM, 4x1TB disk
  - 2012 – 16+ cores, 48-96GB of RAM, 12x2TB or 12x3TB of disk.
- **Cluster utilization**
  - distinct map slots and reduce slots
- **Supporting workloads other than MapReduce**
  - MapReduce is great for many applications, but not everything.

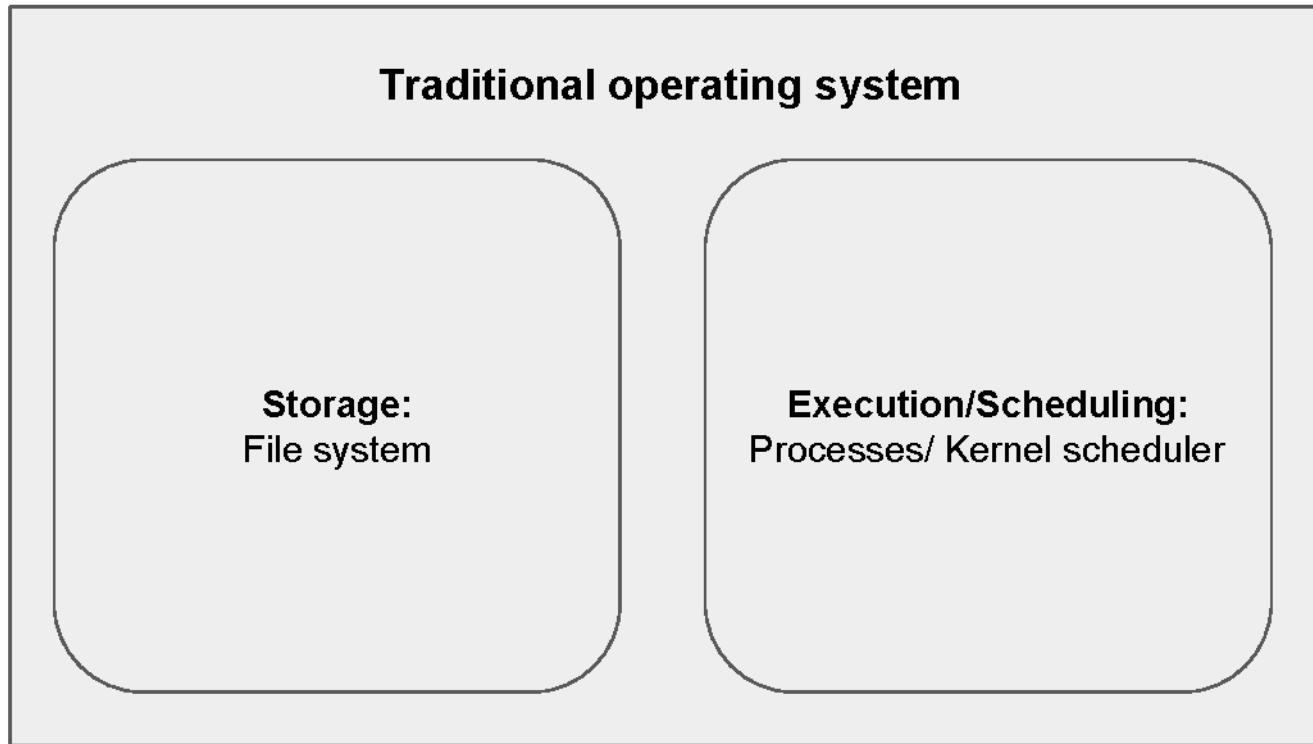
# Need for YARN

## Hadoop 2.0

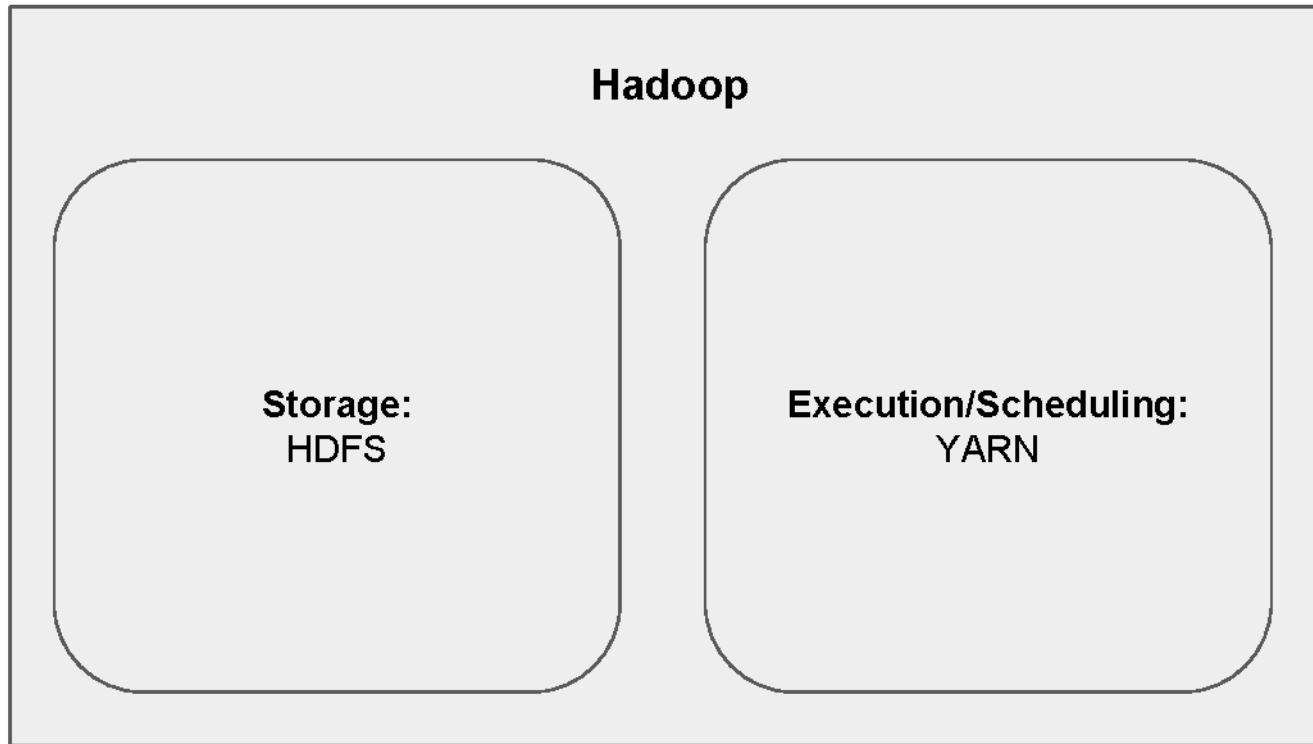
- Multi purpose platform
- Capable of running apps other than MR



# OS analogy



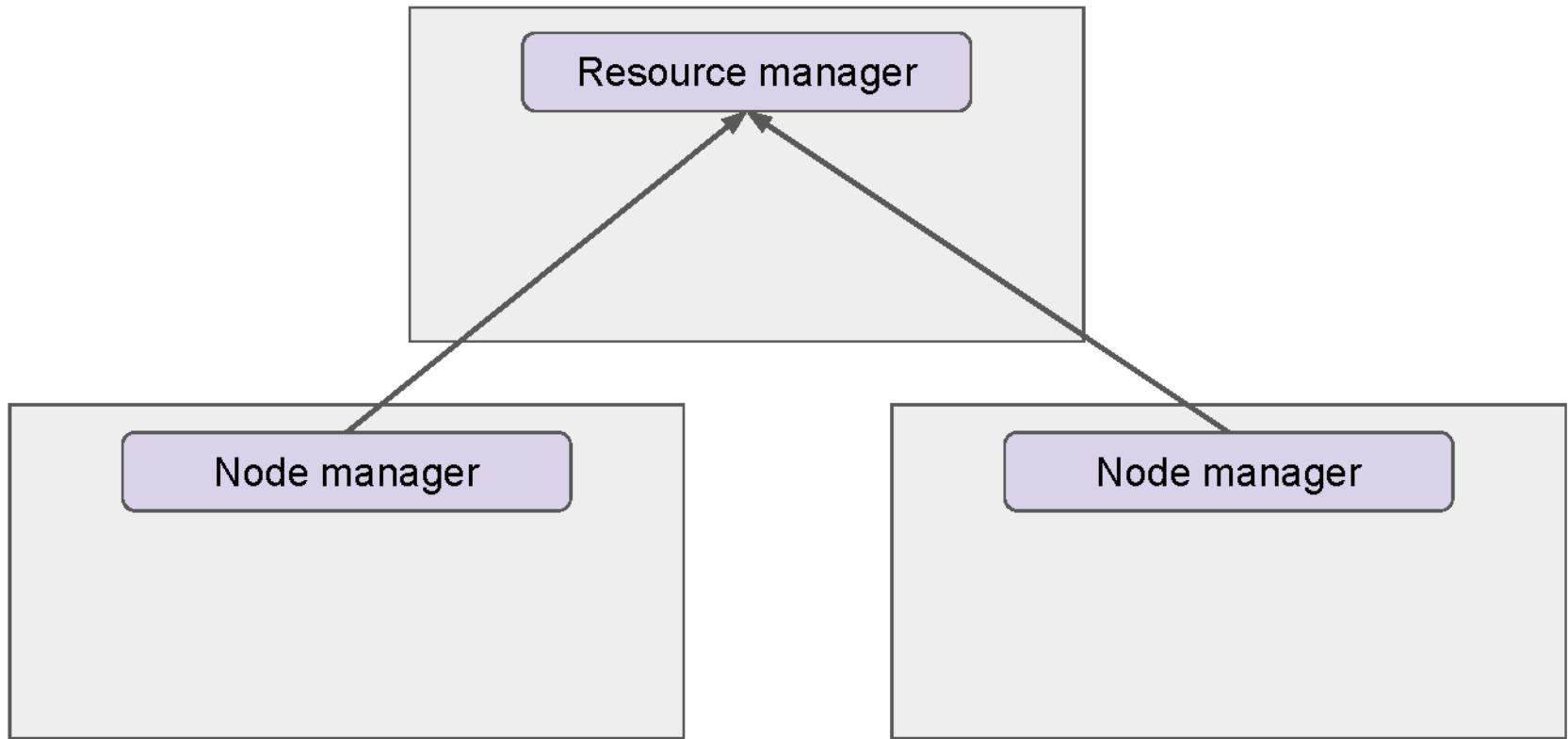
# OS analogy



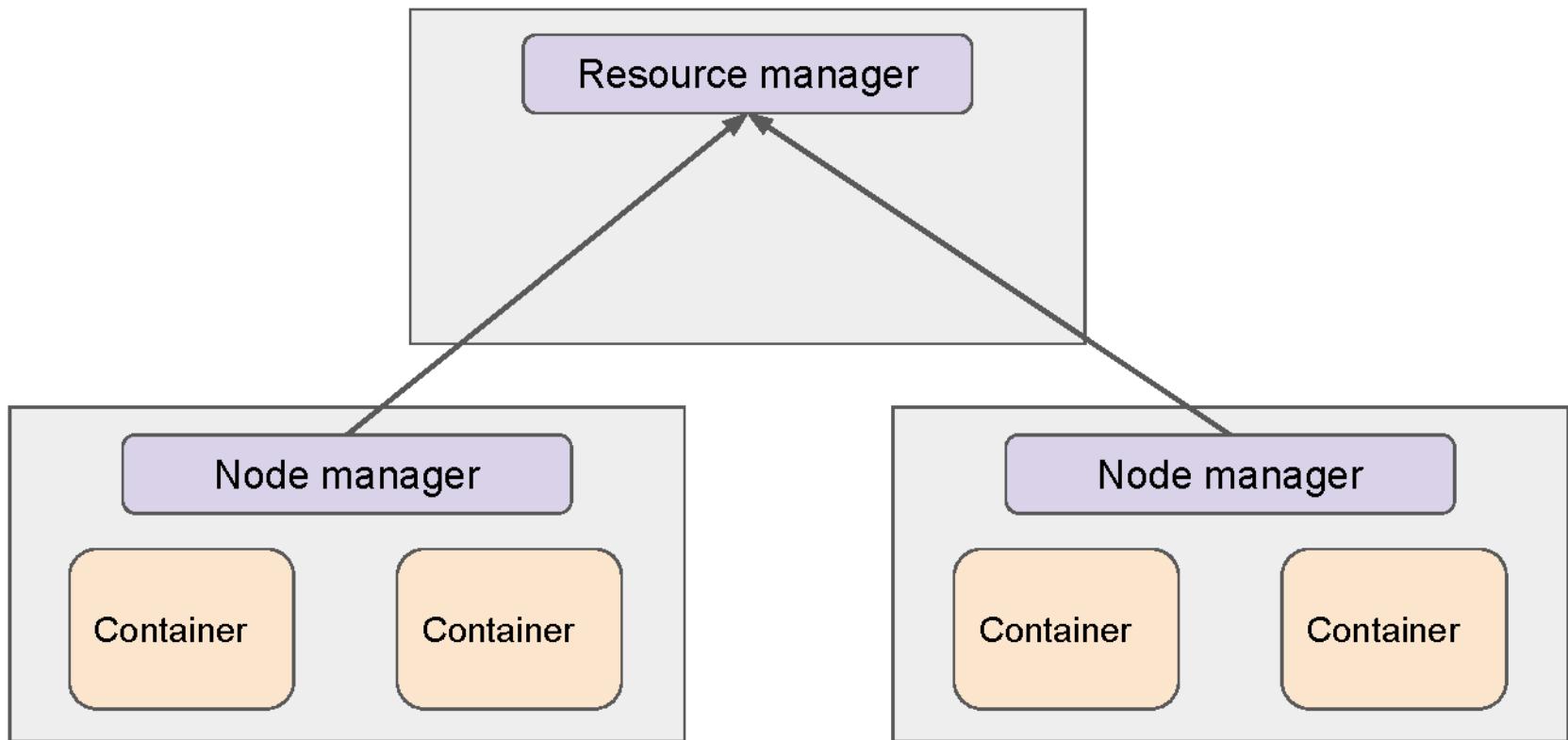
# Why run Spark on YARN

- Leverage existing clusters
- Data locality
- Dynamically sharing the cluster resources between different frameworks.
- YARN schedulers can be used for categorizing, isolating, and prioritizing workloads.
- Only cluster manager for Spark that supports security

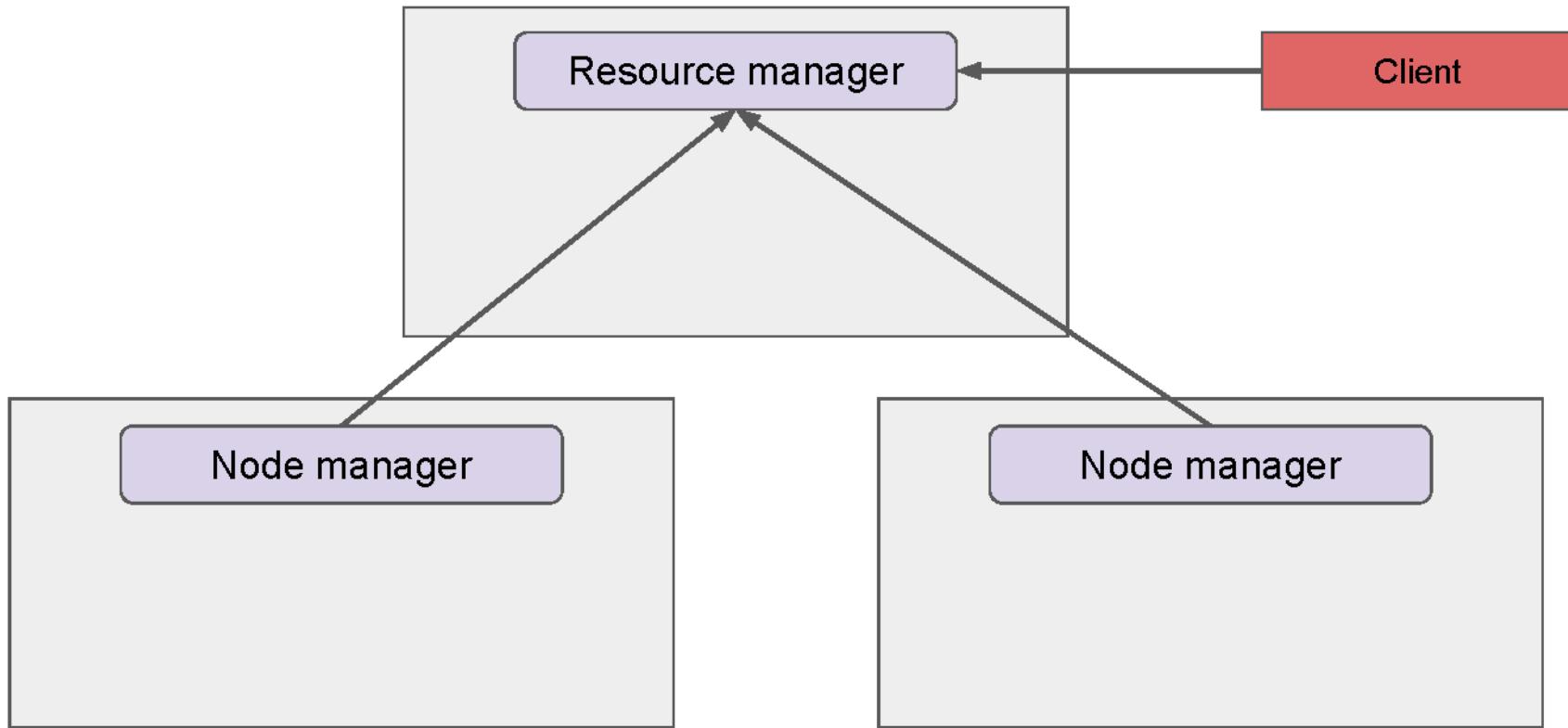
# YARN architecture



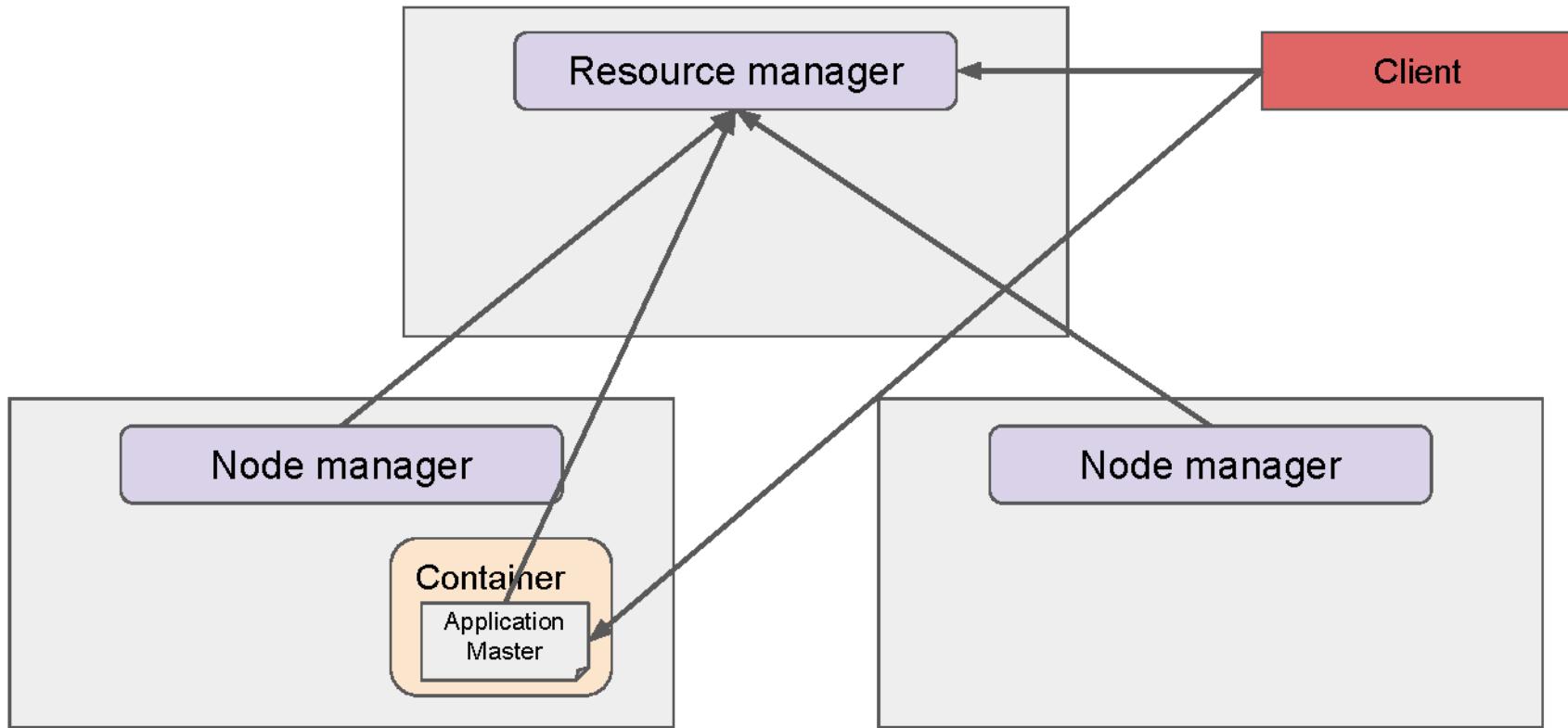
# YARN architecture



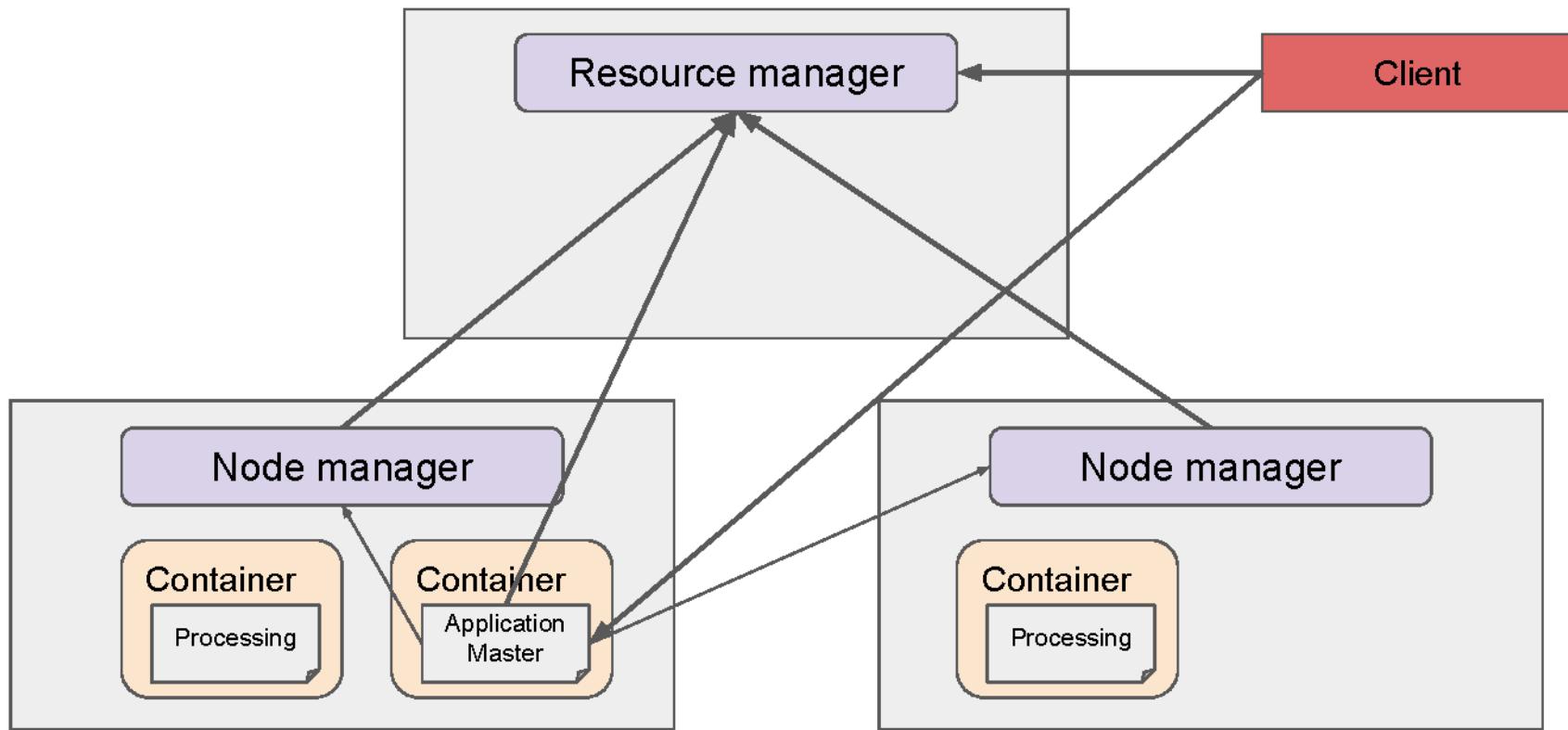
# YARN architecture



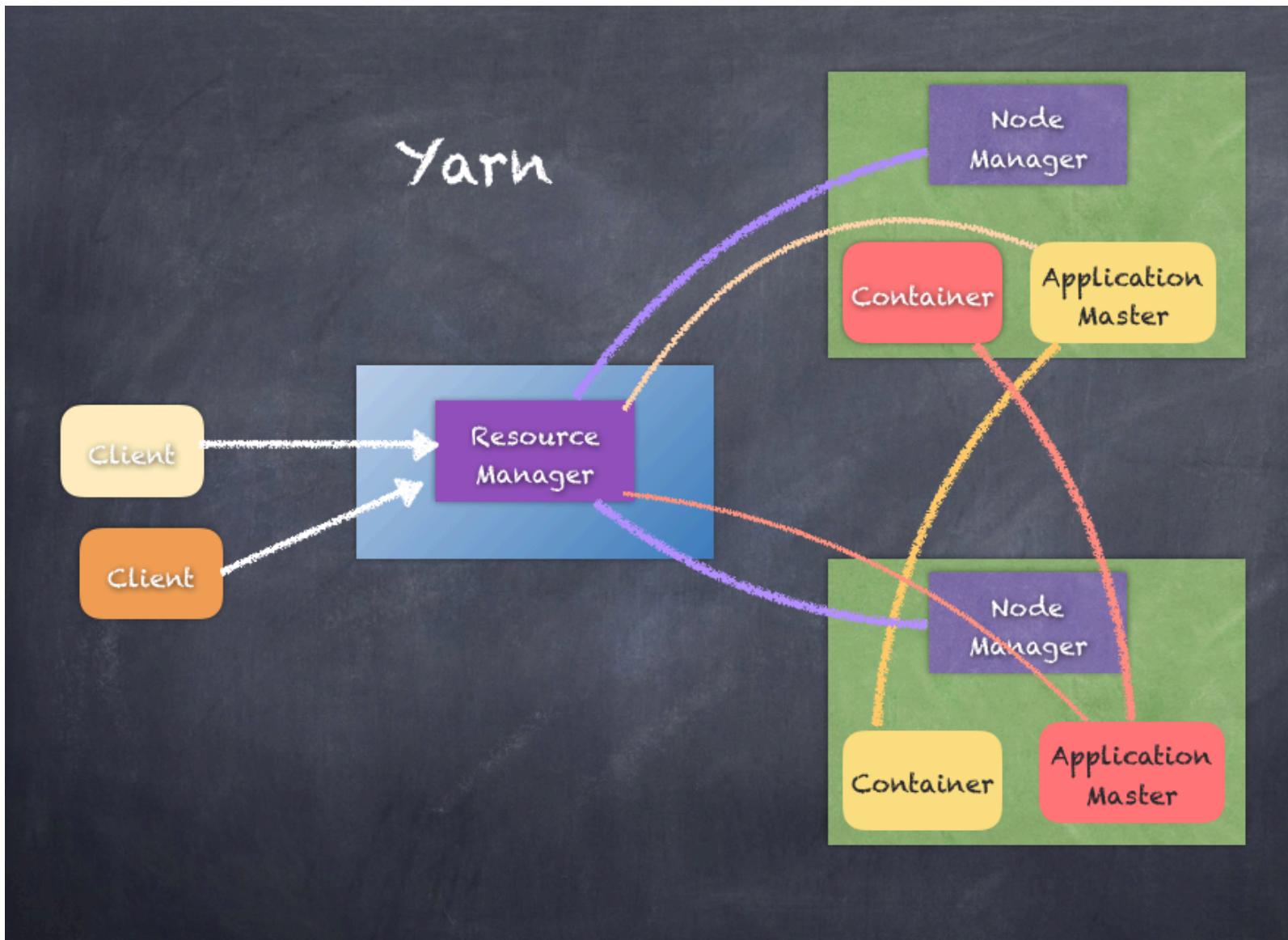
# YARN architecture



# YARN architecture

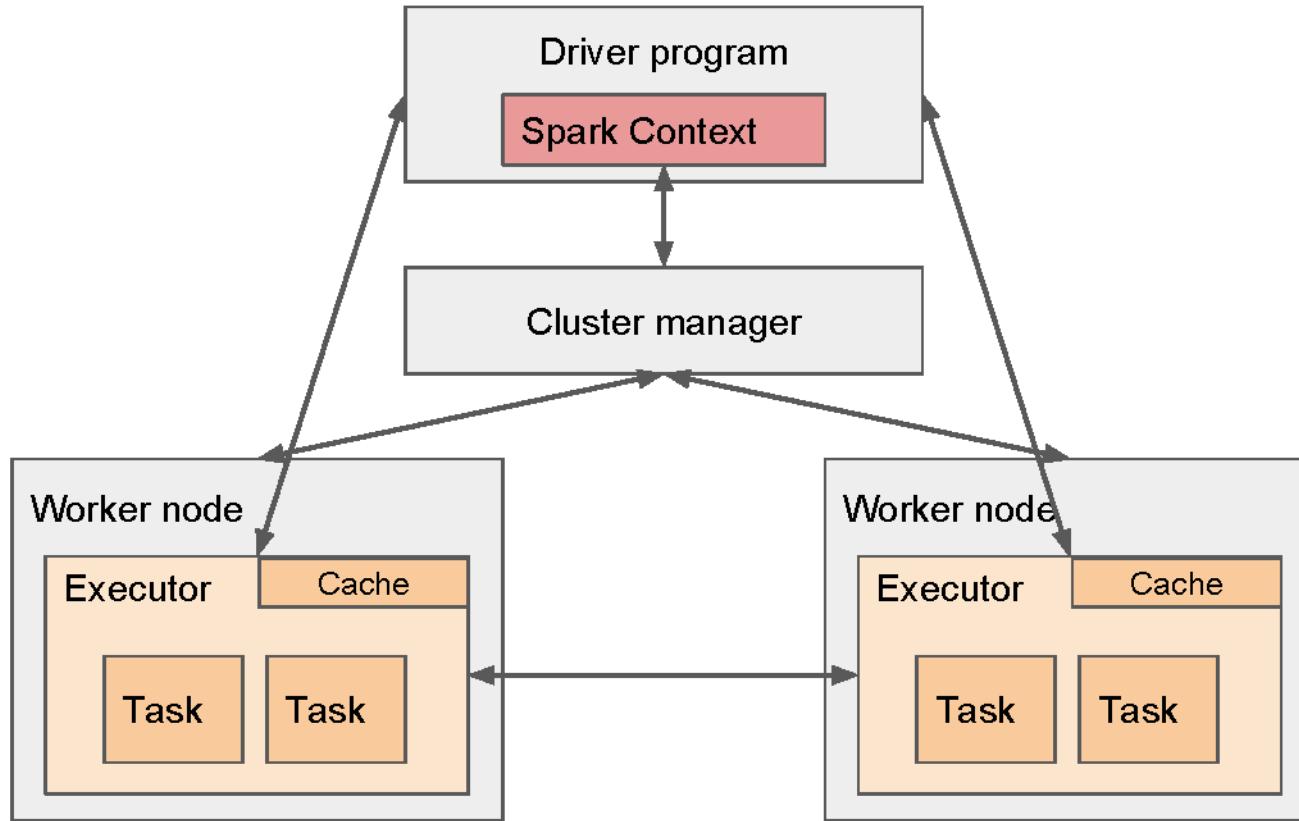


# YARN



# Running Spark on YARN

# Spark architecture



# Spark architecture

- **Driver Program** is responsible for managing the job flow and scheduling tasks that will run on the executors.
- **Executors** are processes that run computation and store data for a Spark application.
- **Cluster Manager** is responsible for starting executor processes and where and when they will be run. Spark supports pluggable cluster manager, it supports

Example: YARN, Mesos and “standalone” cluster manager

# Modes on Spark on YARN

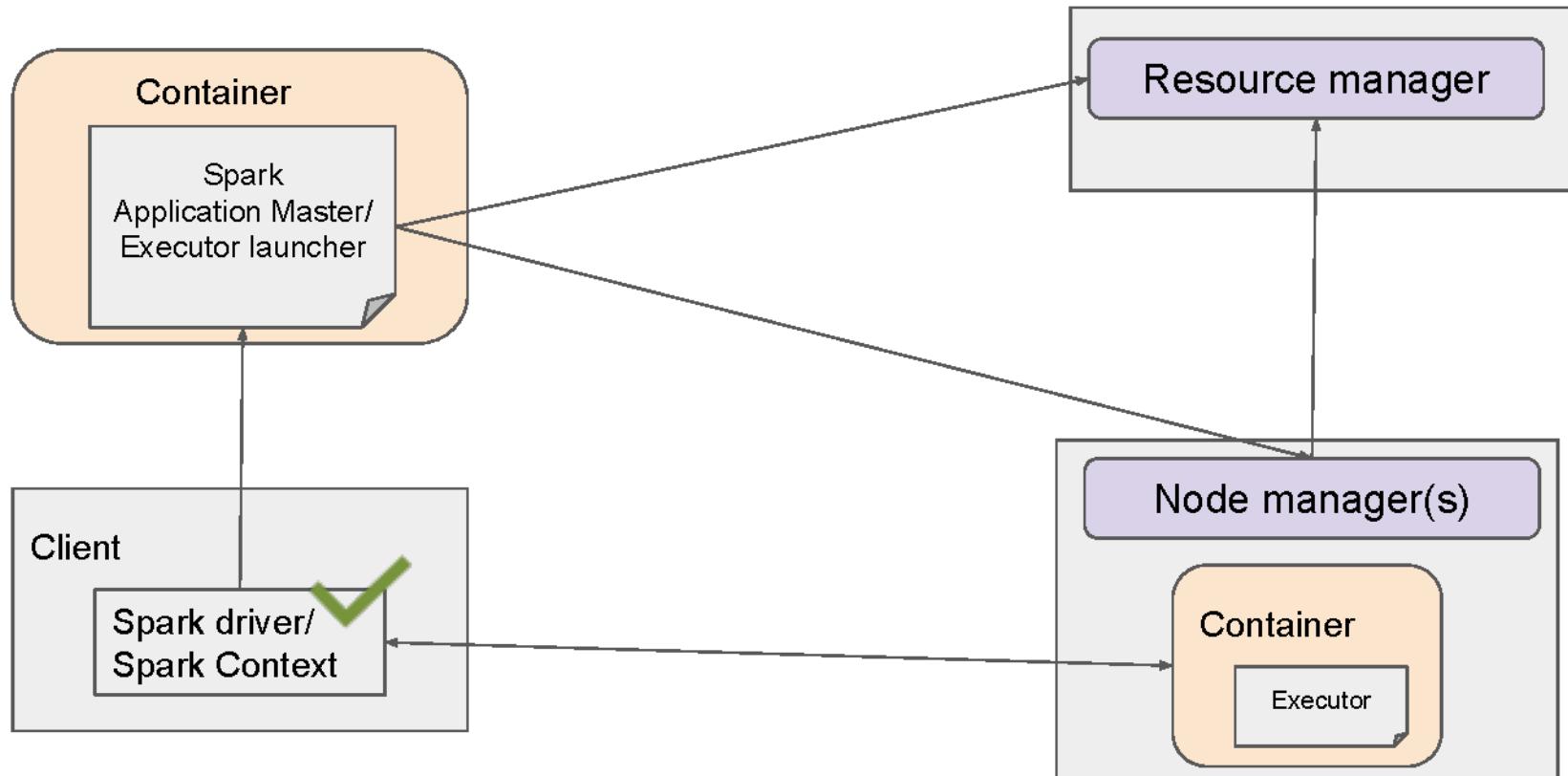
- YARN-Client Mode **Where does driver runs ???**
- YARN-Cluster Mode

# YARN client mode

- Driver runs in the client process, and the application master is only used for requesting resources from YARN.
- Used for interactive and debugging uses where you want to see your application's output immediately (on the client process side).

**So if the client goes away, the job is killed**

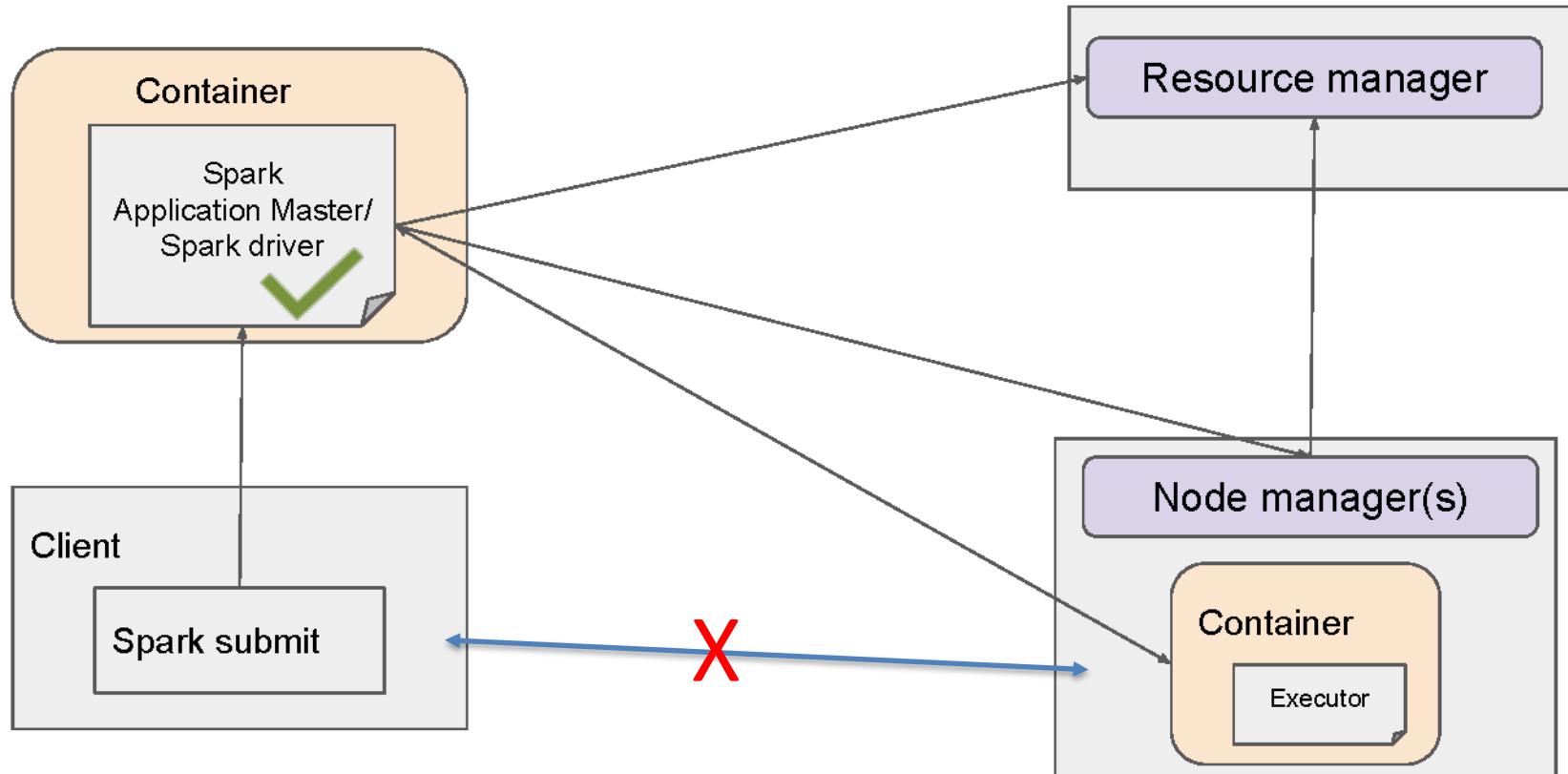
# YARN client mode



# YARN cluster mode

- In yarn-cluster mode, the Spark driver runs inside an application master process which is managed by YARN on the cluster, and the client can go away after initiating the application.
- Yarn-cluster mode makes sense for production jobs.

# YARN cluster mode



# Concurrency vs Parallelism

- Concurrency is about dealing with lots of things at once.
- Parallelism is about doing lots of things at once.

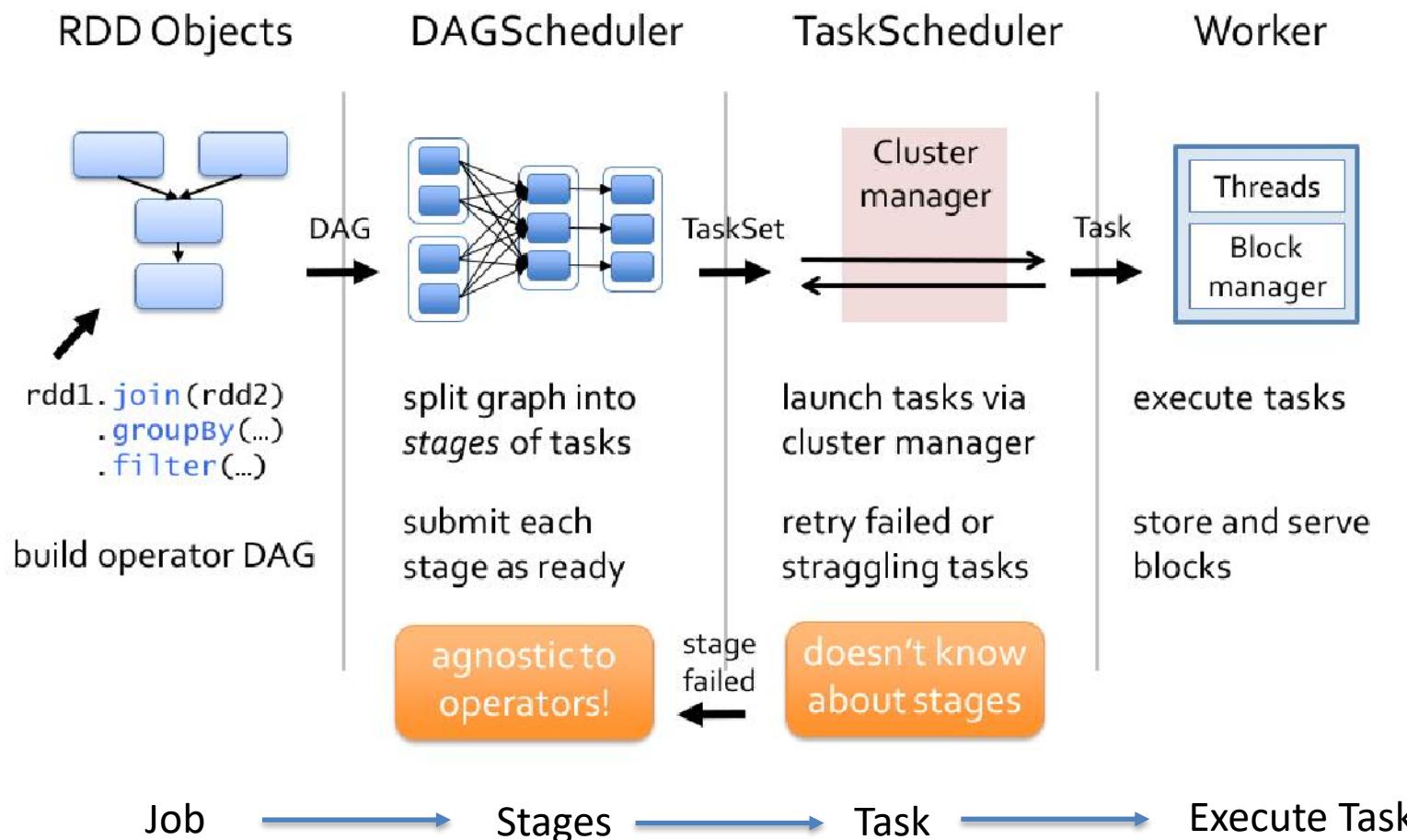
# Akka

- Follows Actor model
  - Keep mutable state internally and communicate through async messages
- Actors

Treat Actors like People. People who don't talk to each other in person. They just talk through mails.

- Object
- Runs in its own thread
- Messages are kept in queue and processed in order

# Internals of Spark



# Installing Apache Spark

- Download Spark
- Download Java (at least 8)
- Install standalone cluster

Downloads | Apache Spark   X   Sridhar

Secure | https://spark.apache.org/downloads.html

APACHE  Spark™ *Lightning-fast cluster computing*

Download   Libraries   Documentation   Examples   Community   Developers   Apache Software Foundation

## Download Apache Spark™

1. Choose a Spark release: 2.2.0 (Jul 11 2017)
2. Choose a package type: Pre-built for Apache Hadoop 2.7 and later
3. Choose a download type: Direct Download
4. Download Spark: [spark-2.2.0-bin-hadoop2.7.tgz](#)
5. Verify this release using the [2.2.0 signatures and checksums](#) and [project release KEYS](#).

Note: Starting version 2.0, Spark is built with Scala 2.11 by default. Scala 2.10 users should download the Spark source package and build with Scala 2.10 support.

### Link with Spark

Spark artifacts are [hosted in Maven Central](#). You can add a Maven dependency with the following coordinates:

Latest News

- Spark 2.2.0 released (Jul 11, 2017)
- Spark 2.1.1 released (May 02, 2017)
- Spark Summit (June 5-7th, 2017, San Francisco) agenda posted (Mar 31, 2017)
- Spark Summit East (Feb 7-9th, 2017, Boston) agenda posted (Jan 04, 2017)

[Archive](#)

# http://IPADDRESS:4040/jobs/

APACHE Spark 2.1.1

Jobs Stages Storage Environment Executors SQL

Spark shell application UI

## Spark Jobs (?)

User: salla  
Total Uptime: 33.0 h  
Scheduling Mode: FIFO  
Completed Jobs: 61

▶ Event Timeline

### Completed Jobs (61)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
60	take at <console>:34	2017/05/22 21:56:46	6 ms	1/1	3/3
59	take at <console>:34	2017/05/22 21:56:46	22 ms	1/1	4/4
58	take at <console>:34	2017/05/22 21:56:46	6 ms	1/1	1/1
57	take at <console>:34	2017/05/22 21:56:18	12 ms	1/1	3/3
56	take at <console>:34	2017/05/22 21:56:18	22 ms	1/1	4/4
55	take at <console>:34	2017/05/22 21:56:18	5 ms	1/1	1/1
54	take at <console>:34	2017/05/22 21:56:03	5 ms	1/1	3/3
53	take at <console>:34	2017/05/22 21:56:03	7 ms	1/1	4/4
52	take at <console>:34	2017/05/22 21:56:03	19 ms	1/1	1/1
51	take at <console>:32	2017/05/22 21:52:45	20 ms	1/1	3/3
50	take at <console>:32	2017/05/22 21:52:45	21 ms	1/1	4/4
49	take at <console>:32	2017/05/22 21:52:45	12 ms	1/1	1/1

# Getting Started - Launching the console

## Login to AWS machine

Username: ec2-user

Password: pem/ppk file

The machine is created from a custom image having

- Java 8
- Spark 2.4.5
- Anaconda with Python 3
- Jupyter Notebooks
- Necessary settings for accessing notebooks in local browser

**Start a notebook** in the AWS machine

\$ jupyter notebook &

**Access notebook** in your local browser

[https://<aws\\_ip\\_address>:8888](https://<aws_ip_address>:8888)

Password for notebook - 12345

# Running Spark

- **Console (available for Scala, Python and R)**

- Easiest and used to Learn, debug, experiment
- {Scala: spark-shell; Python: pyspark}
- pyspark --jars <comma-separated-jars>

Include external dependencies

- **Submit**

- Production jobs
- spark-submit --master <master> --jars <comma-separated-jars> --conf <spark-properties> --name <job\_name> <python\_file> <argument\_1> <argument\_2>

# **3. Spark - RDD**

# RDDs - Resilient Distributed Datasets

## What is RDD?

**Dataset:**  
*Collection of data elements.*  
e.g. Array, Tables, Data frame (R)

**Distributed:**  
*Parts Multiple machines*

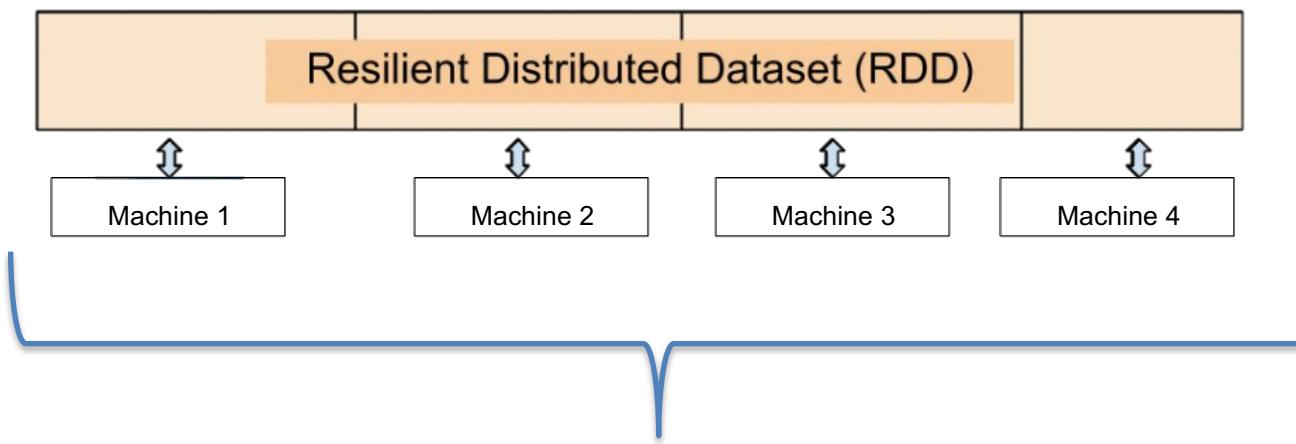
**Resilient:**  
*Recovers on Failure*



Partitioned

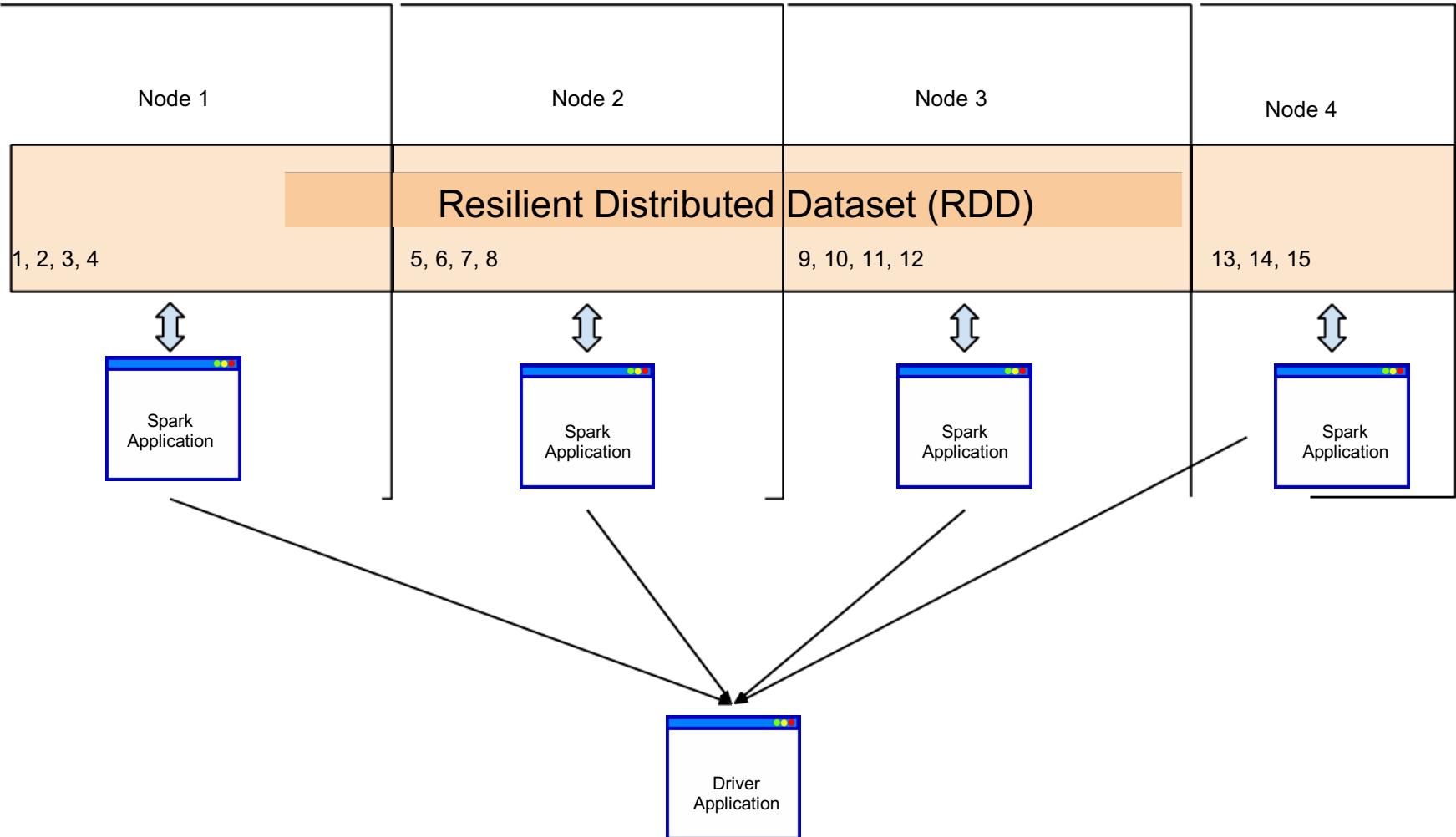
# RDDs - Resilient Distributed Datasets

A collection of elements partitioned across cluster



Actually blocks, not machines..  
One machine has multiple blocks

# RDDs - Resilient Distributed Datasets



# RDDs - Resilient Distributed Datasets

---

**A collection of elements partitioned across cluster**

- An immutable distributed collection of objects.
- Split in partitions which may be on multiple nodes
- Can contain any data type:
  - Python,
  - Java,
  - Scala objects
  - including user defined classes

# RDDs - Resilient Distributed Datasets

---

- RDD Can be persisted in memory
- RDD Auto recover from node failures
- Can have any data type but has a special dataset type for key-value
- Supports two type of operations:
  - Transformation
  - Action

# Creating RDD - Python

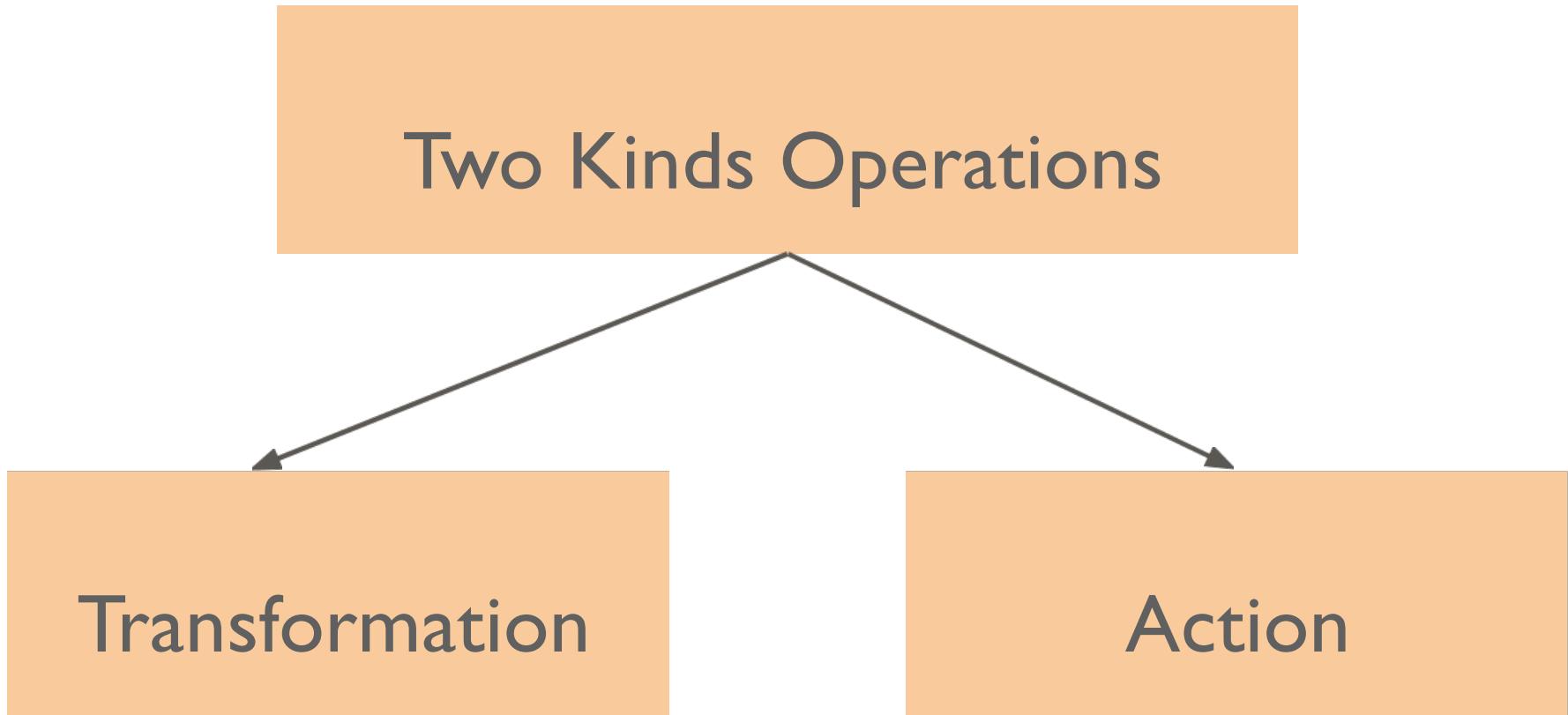
## Method 1: By Directly Loading a file from remote

```
rdd = sc.textFile('/home/ec2-user/data/wordcount.file1')
print(rdd.first())
rdd.take(2)
```

## Method 2: By distributing existing object

```
data = [1, 2, 3, 4, 5]
rdd = sc.parallelize(data)
print(rdd.first())
rdd.take(2)
```

# RDD Operations



# RDD Transformations

Transformation	Meaning
<code>map(func)</code>	Return a new distributed dataset formed by passing each element of the source through a function <i>func</i> .
<code>filter(func)</code>	Return a new dataset formed by selecting those elements of the source on which <i>func</i> returns true.
<code>flatMap(func)</code>	Similar to map, but each input item can be mapped to 0 or more output items (so <i>func</i> should return a Seq rather than a single item).
<code>mapPartitions(func)</code>	Similar to map, but runs separately on each partition (block) of the RDD, so <i>func</i> must be of type <code>Iterator&lt;T&gt; =&gt; Iterator&lt;U&gt;</code> when running on an RDD of type T.
<code>mapPartitionsWithIndex(func)</code>	Similar to mapPartitions, but also provides <i>func</i> with an integer value representing the index of the partition, so <i>func</i> must be of type <code>(Int, Iterator&lt;T&gt;) =&gt; Iterator&lt;U&gt;</code> when running on an RDD of type T.
<code>sample(withReplacement, fraction, seed)</code>	Sample a fraction <i>fraction</i> of the data, with or without replacement, using a given random number generator seed.
<code>union(otherDataset)</code>	Return a new dataset that contains the union of the elements in the source dataset and the argument.
<code>intersection(otherDataset)</code>	Return a new RDD that contains the intersection of elements in the source dataset and the argument.

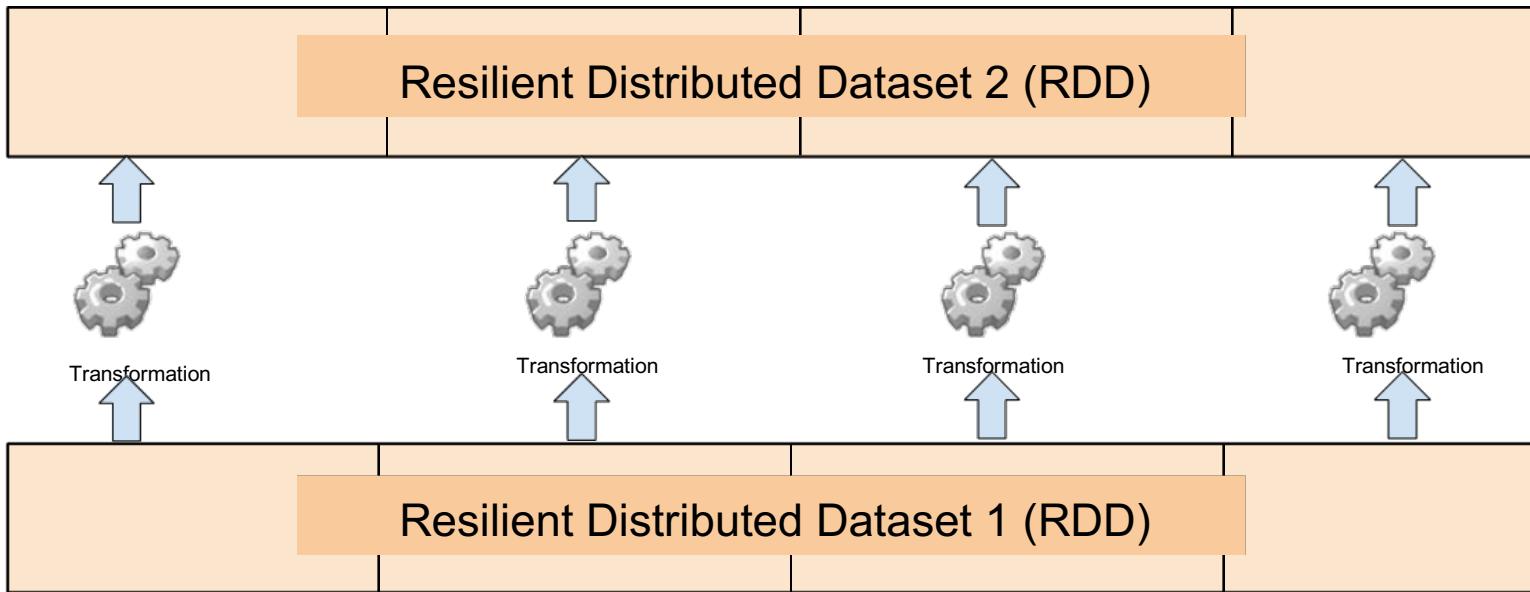
# RDD Transformations

Transformation	Meaning
<code>distinct([numPartitions])</code>	Return a new dataset that contains the distinct elements of the source dataset.
<code>groupByKey([numPartitions])</code>	When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable<V>) pairs. <b>Note:</b> If you are grouping in order to perform an aggregation (such as a sum or average) over each key, using <code>reduceByKey</code> or <code>aggregateByKey</code> will yield much better performance. <b>Note:</b> By default, the level of parallelism in the output depends on the number of partitions of the parent RDD. You can pass an optional <code>numPartitions</code> argument to set a different number of tasks.
<code>reduceByKey(func, [numPartitions])</code>	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 <code>func</code> , which must be of type (V,V) => V. Like in <code>groupByKey</code> , the number of reduce tasks is configurable through an optional second argument.
<code>aggregateByKey(zeroValue)(seqOp, combOp, [numPartitions])</code>	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 <code>groupByKey</code> , the number of reduce tasks is configurable through an optional second argument.
<code>sortByKey([ascending], [numPartitions])</code>	When called on a dataset of (K, V) pairs where K implements <code>Ordered</code> , returns a dataset of (K, V) pairs sorted by keys in ascending or descending order, as specified in the boolean <code>ascending</code> argument.
<code>join(otherDataset, [numPartitions])</code>	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 <code>leftOuterJoin</code> , <code>rightOuterJoin</code> , and <code>fullOuterJoin</code> .

# RDD Transformations

Transformation	Meaning
<code>cogroup(otherDataset, [numPartitions])</code>	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.
<code>cartesian(otherDataset)</code>	When called on datasets of types T and U, returns a dataset of (T, U) pairs (all pairs of elements).
<code>pipe(command, [envVars])</code>	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.
<code>coalesce(numPartitions)</code>	Decrease the number of partitions in the RDD to numPartitions. Useful for running operations more efficiently after filtering down a large dataset.
<code>repartition(numPartitions)</code>	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.
<code>repartitionAndSortWithinPartitions(partitioner)</code>	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.

# RDD - Operations : Transformation

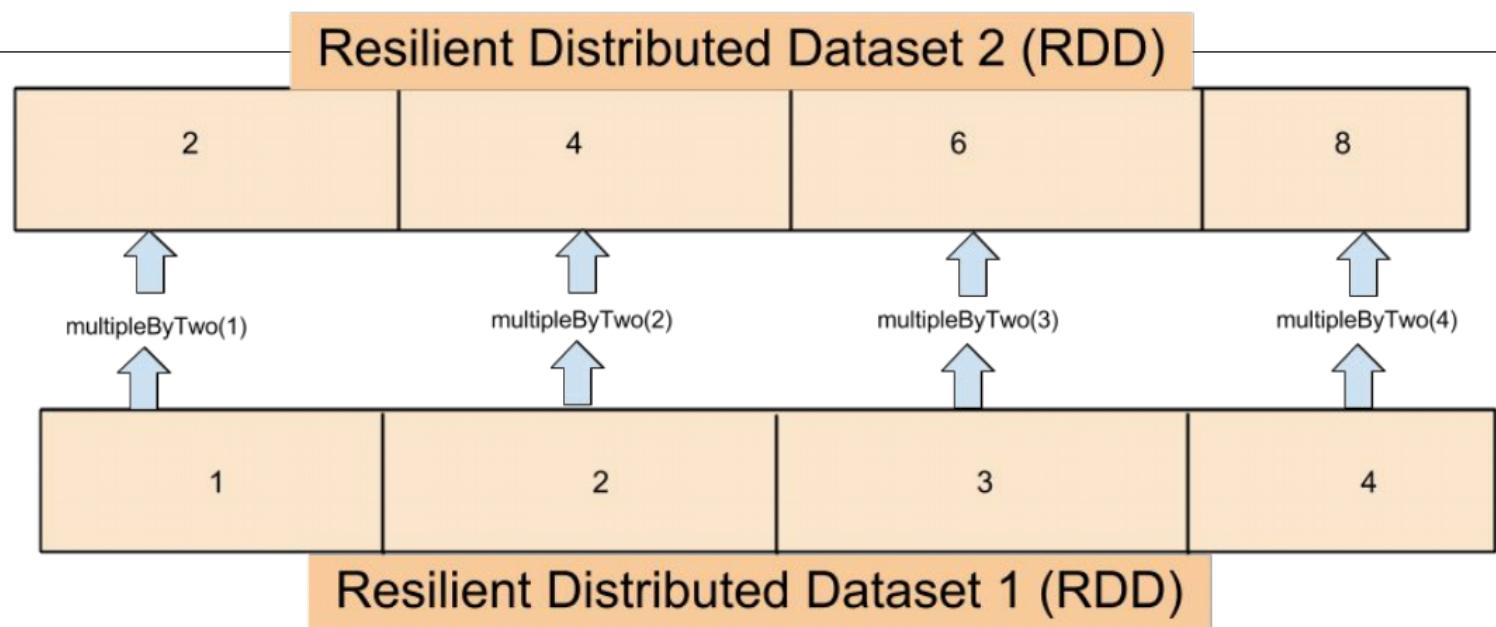


- **Transformations are operations on RDDs**
- **return a new RDD**
- **such as map() and filter()**



# Map Transformation

- Map is a transformation
- That runs provided function against each element of RDD
- And creates a new RDD from the results of execution function





# Map Transformation

## Map Transformation

```
In [5]: print ('Map')
def multiply_by_two(x:int) -> int:
    return x*2

data = range(0,100)
rdd = sc.parallelize(data)
print ('input rdd: ', rdd.take(5))

rdd = rdd.map(multiply_by_two)
print ('output rdd: ', rdd.take(5))
```

```
Map
input rdd:  [0, 1, 2, 3, 4]
output rdd:  [0, 2, 4, 6, 8]
```



# Filter Transformation

## Filter Transformation

```
In [6]: print ('Filter')
def filter_by_even(x:int) -> int:
    return x%2==0

data = range(0,100)
rdd = sc.parallelize(data)
print ('input rdd: ', rdd.take(10))

rdd = rdd.filter(filter_by_even)
print ('output rdd: ', rdd.take(10))
```

```
Filter
input rdd:  [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
output rdd:  [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

# Transformations:: flatMap()

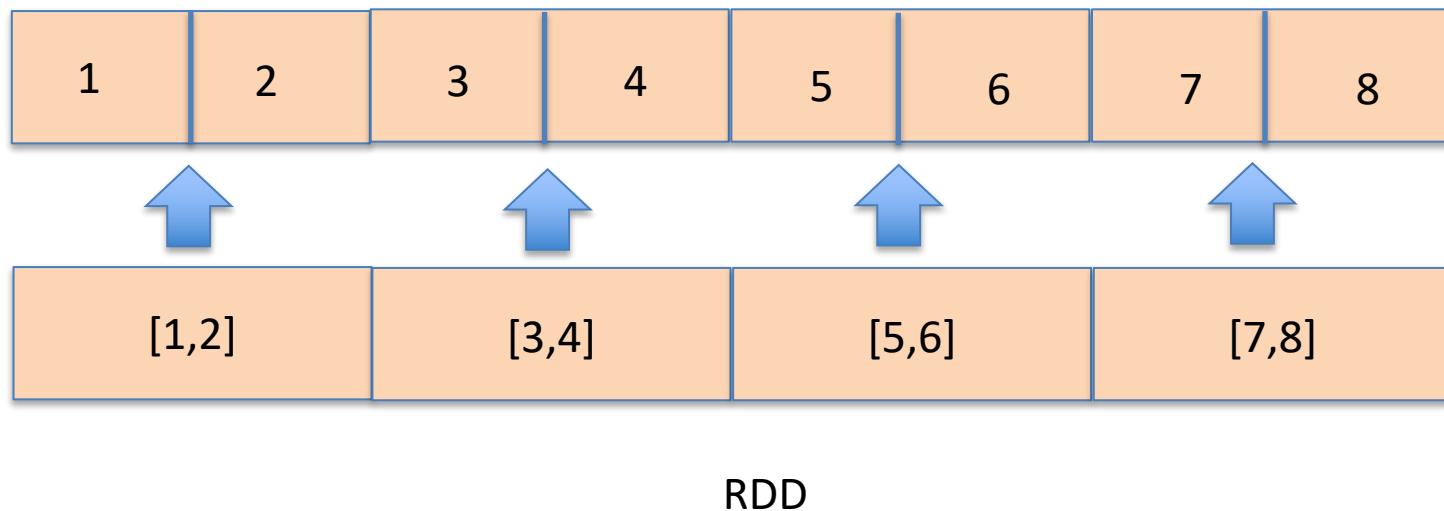
To convert one record of an RDD into multiple records.

# How is it different from Map()?

- In case of map() the resulting rdd and input rdd having same number of elements.
- map() can only convert one to one while flatMap could convert one to many.



# FlatMap Transformation





# FlatMap Transformation

## FlatMap Transformation

```
print ('FlatMap in function')
def flatten(x):
    return x[1].split(',')

data = [('A', '1,2,3'), ('B', '5,6,7'), ('C', '8,9,10')]
rdd = sc.parallelize(data)
print ('input rdd: ', rdd.take(3))

rdd = rdd.flatMap(flatten)
print ('output rdd: ', rdd.take(6))
```

```
FlatMap in function
input rdd:  [('A', '1,2,3'), ('B', '5,6,7'), ('C', '8,9,10')]
output rdd:  ['1', '2', '3', '5', '6', '7']
```

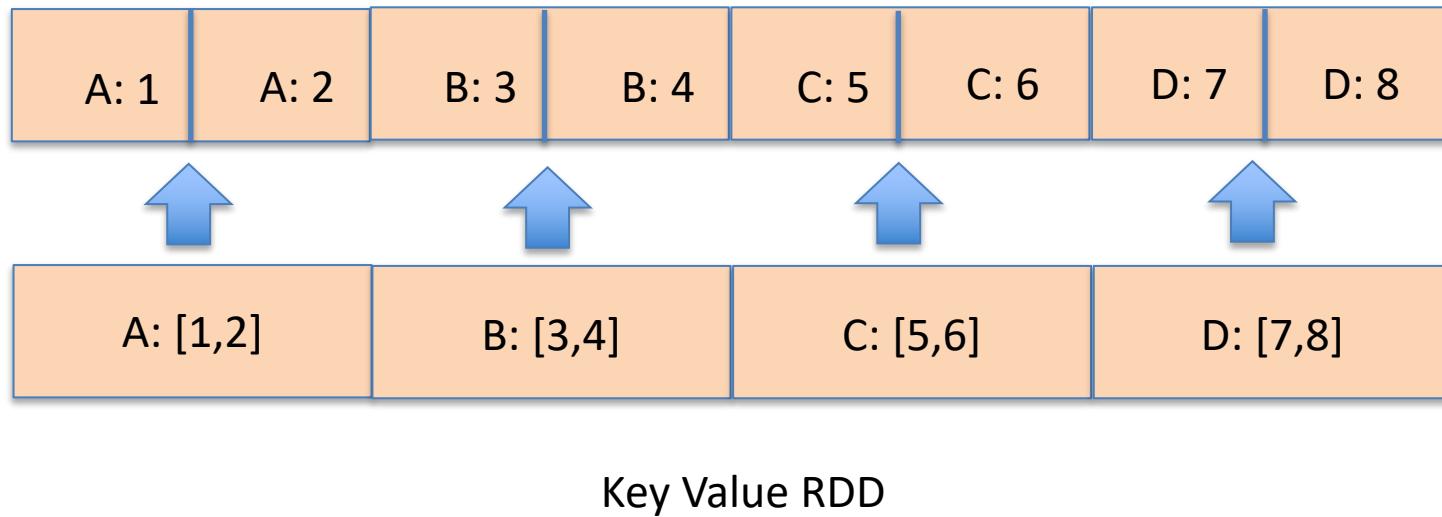
```
print ('FlatMap in lambda')
data = [('A', '1,2,3'), ('B', '5,6,7'), ('C', '8,9,10')]
rdd = sc.parallelize(data)
print ('input rdd: ', rdd.take(3))

rdd = rdd.flatMap(lambda x: (x[1].split(',')))
print ('output rdd: ', rdd.take(6))
```

```
FlatMap in lambda
input rdd:  [('A', '1,2,3'), ('B', '5,6,7'), ('C', '8,9,10')]
output rdd:  ['1', '2', '3', '5', '6', '7']
```



# FlatMapValues Transformation





# FlatMapValues Transformation

## FlatMapValues Transformation

```
print ('FlatMapValues in function')
def flatten(x):
    return x.split(' ')
data = [('A', '1,2,3'), ('B', '5,6,7'), ('C', '8,9,10')]
rdd = sc.parallelize(data)
print ('input rdd: ', rdd.take(3))

rdd = rdd.flatMapValues(flatten)
print ('output rdd: ', rdd.take(6))
```

```
FlatMapValues in function
input rdd:  [('A', '1,2,3'), ('B', '5,6,7'), ('C', '8,9,10')]
output rdd:  [('A', '1,2,3'), ('B', '5,6,7'), ('C', '8,9,10')]
```

```
print ('FlatMapValues in lambda')
data = [('A', '1,2,3'), ('B', '5,6,7'), ('C', '8,9,10')]
rdd = sc.parallelize(data)
print ('input rdd: ', rdd.take(3))

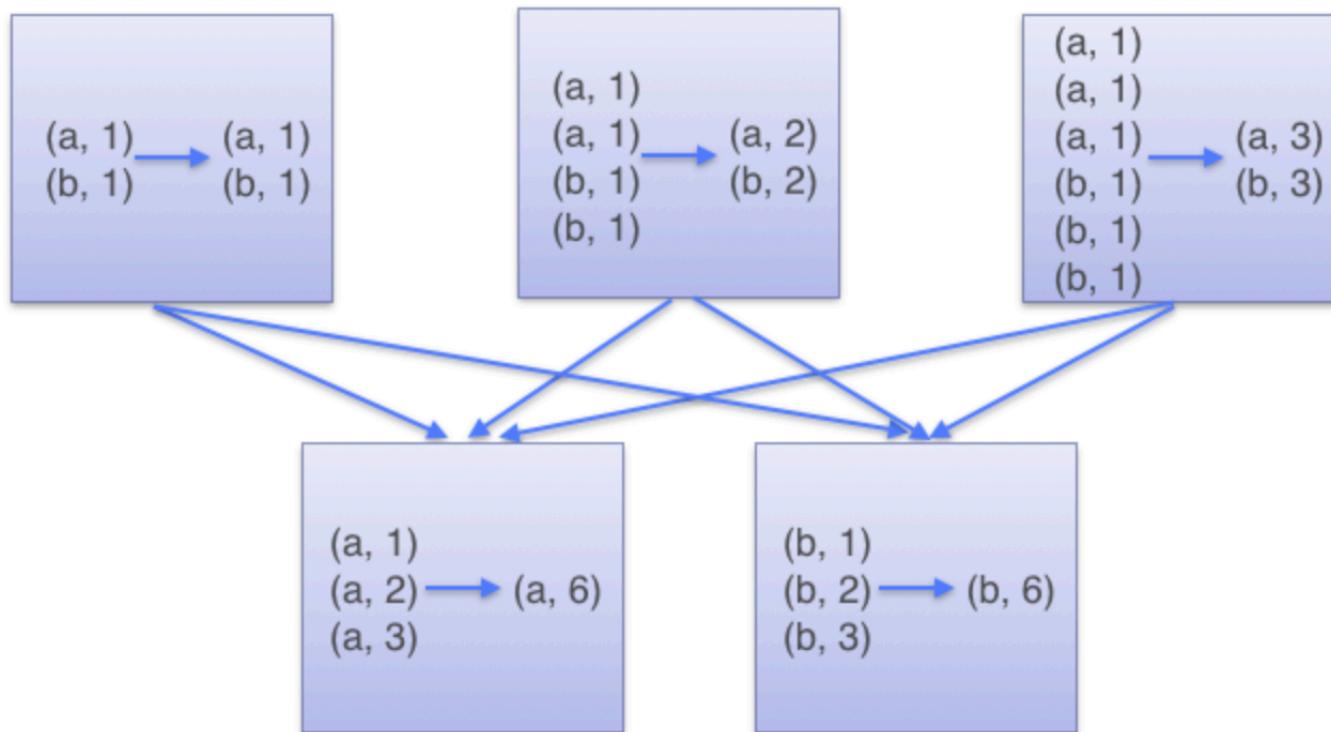
rdd = rdd.flatMapValues(lambda x: x.split(','))
print ('output rdd: ', rdd.take(6))
```

```
FlatMapValues in lambda
input rdd:  [('A', '1,2,3'), ('B', '5,6,7'), ('C', '8,9,10')]
output rdd:  [('A', '1'), ('A', '2'), ('A', '3'), ('B', '5'), ('B', '6'), ('B', '7')]
```

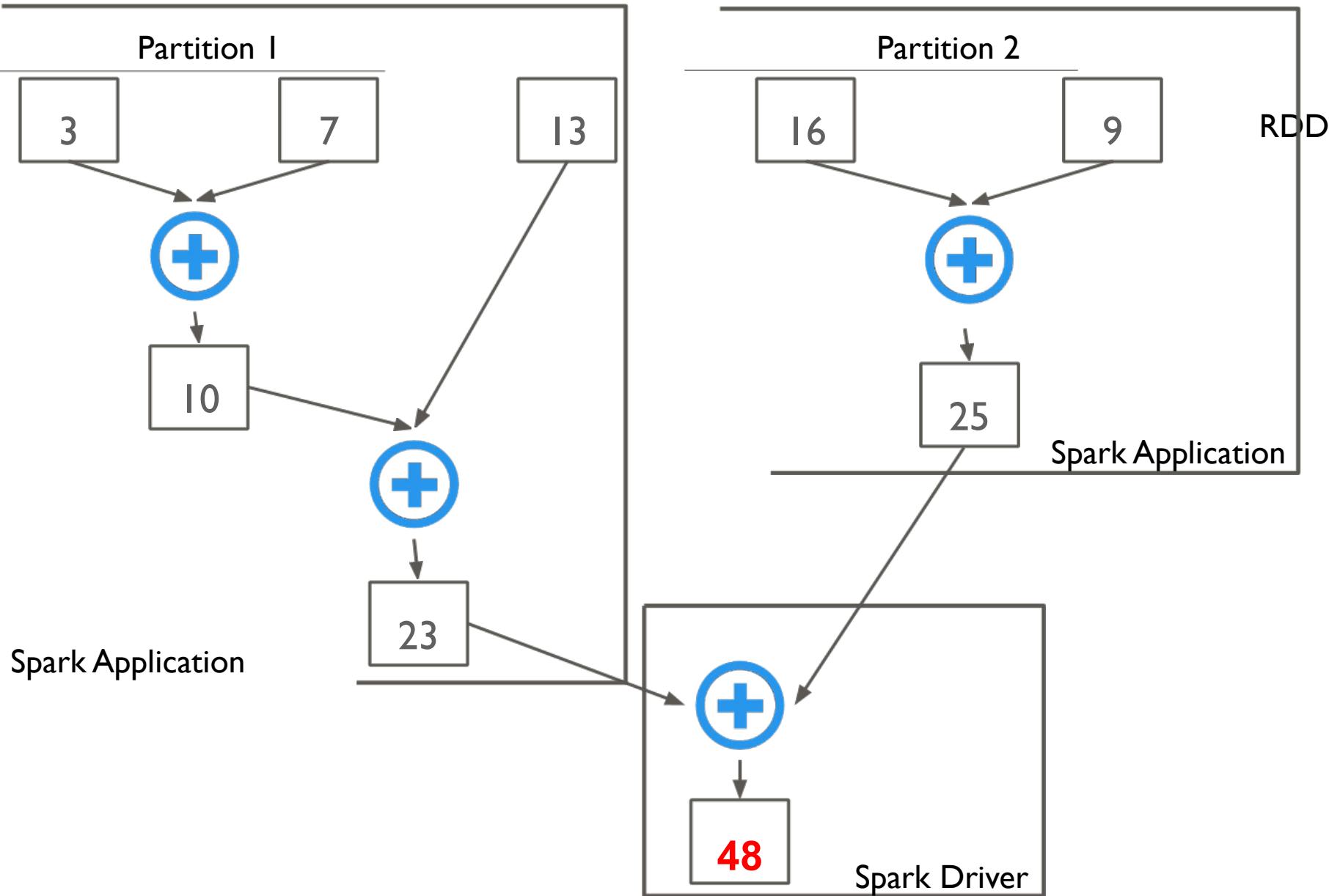


# ReduceByKey Transformation

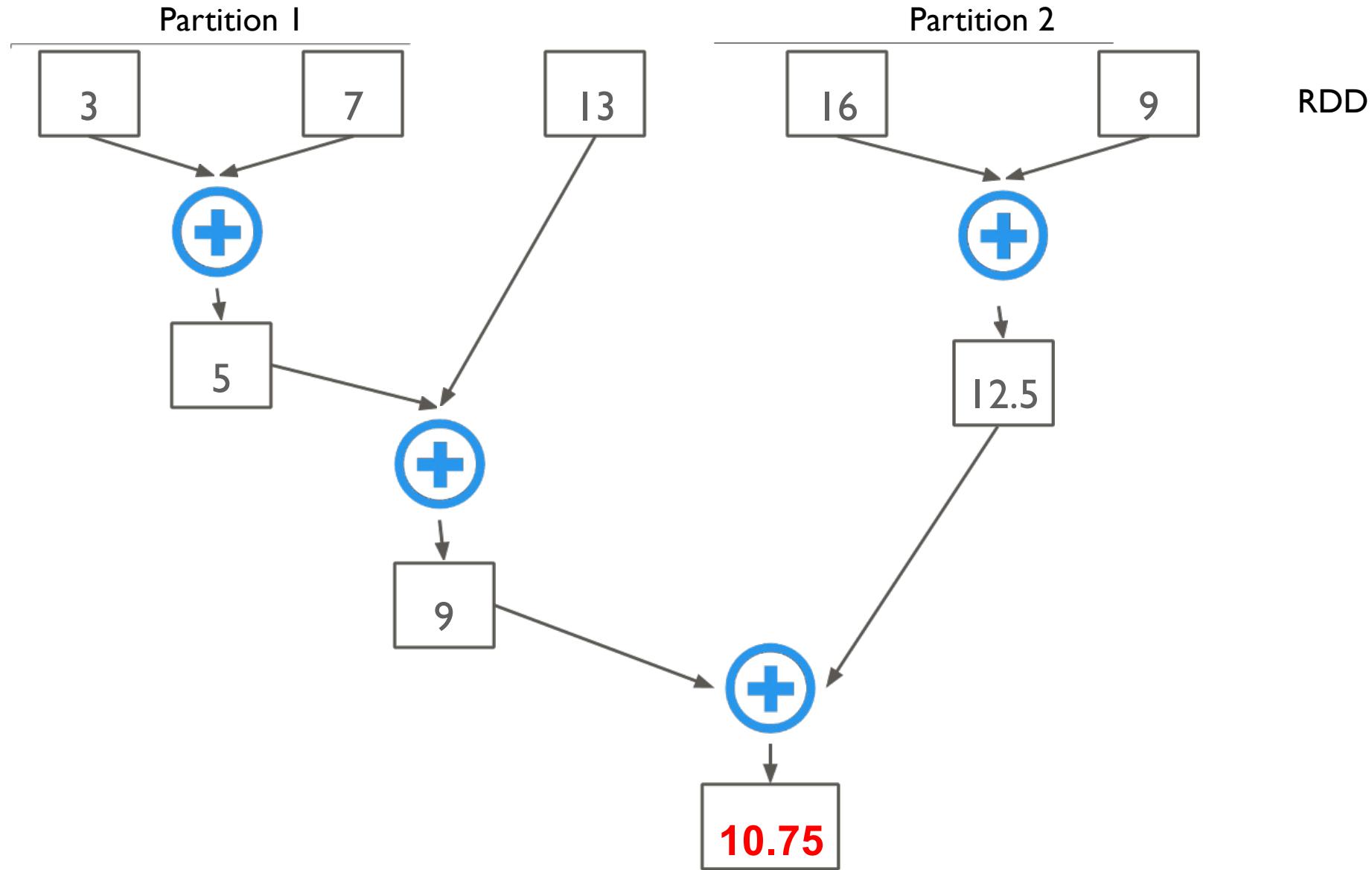
## ReduceByKey



# How does reduce work?



# Why average with reduce is wrong?



# Why average with reduce is wrong?

$$\frac{(3 + 7 + 13)}{3} \neq \frac{\frac{3+7}{2} + 13}{2}$$



# “ReduceByKey” Transformation

## ReduceByKey Transformation

```
rdd_mapped = rdd.map(lambda x: (x[0], 1))
print ('Mapped First: ', rdd_mapped.first())
print ('Mapped Count: ', rdd_mapped.count())

# count occurrence of "artist"
rdd_reduced = rdd_mapped.reduceByKey(lambda x,y: x+y)
print ('\nReduced First: ', rdd_reduced.first())
print ('Reduced Count: ', rdd_reduced.count())
```

Mapped First: ('Future', 1)

Mapped Count: 2017

Reduced First: ('Future', 8)

Reduced Count: 1369

# Approach I - So, how to compute average?

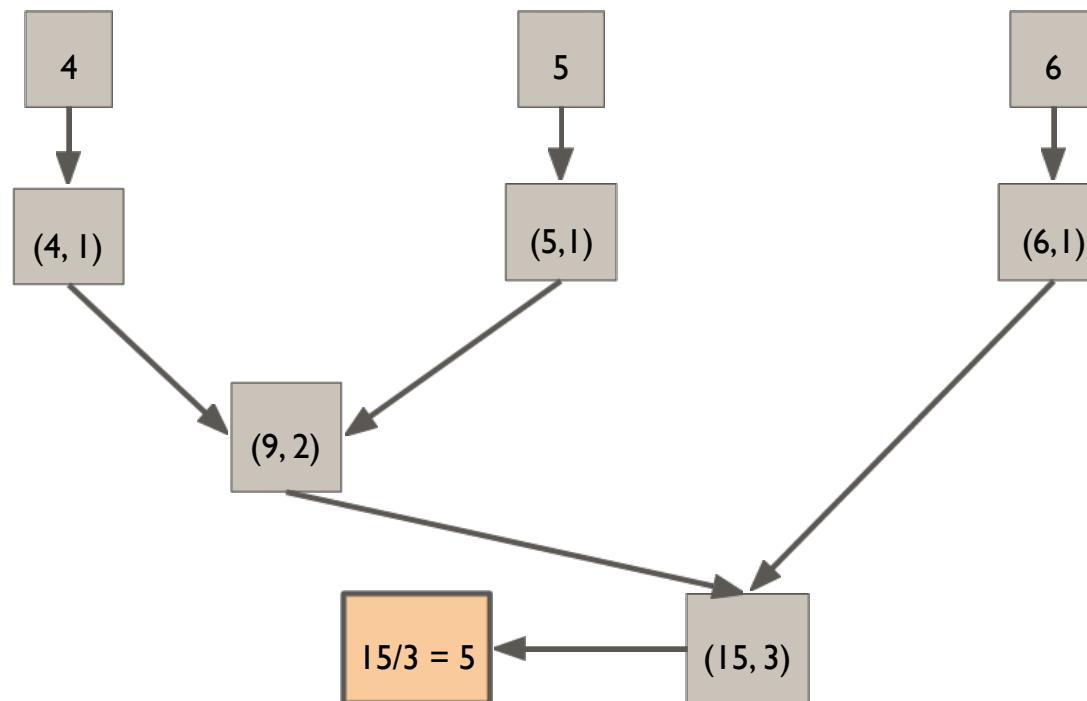
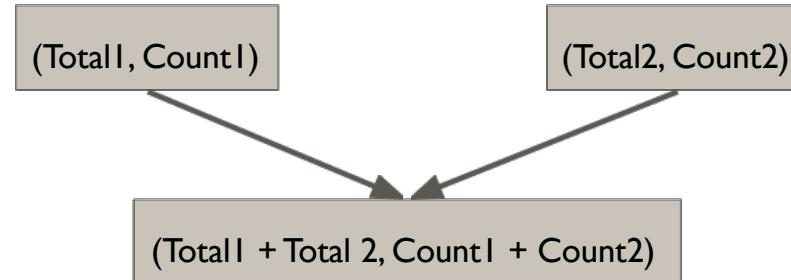
## Approach 1

```
➤ var rdd = sc.parallelize(Array(1.0,2,3, 4, 5 , 6, 7), 3);  
➤ var avg = rdd.reduce(_ + _) / rdd.count();
```

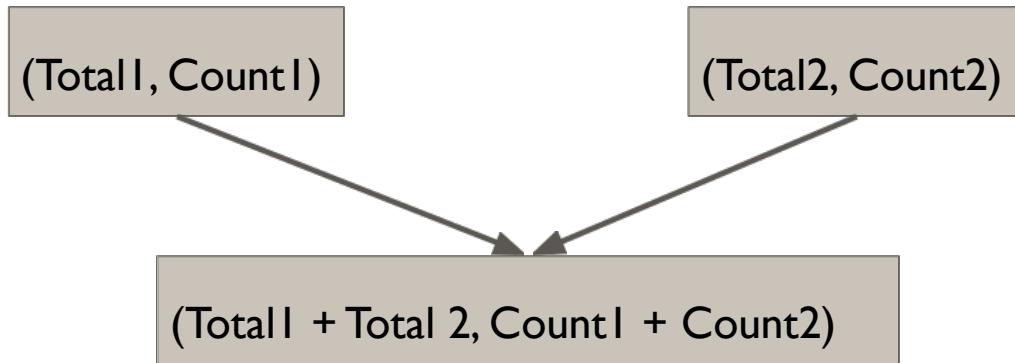
## What's wrong with this approach?

We are computing RDD twice - during **reduce** and during **count**.  
Can we compute sum and count in a single reduce?

# Approach 2 - So, how to compute average?



# Approach 2 - So, how to compute average?



```
> var rdd = sc.parallelize(Array(1.0,2,3, 4, 5 , 6, 7), 3);
> var rdd_count = rdd.map(_ , 1)
> var (sum, count) = rdd_count.reduce((x, y) => (x._1 + y._1, x._2 + y._2))
> var avg = sum / count
```

avg: Double = 4.0

# Comparision of the two approaches?

Approach1:

$$0.023900 + 0.065180$$

= **0.08908** seconds ~ 89 ms

Approach2:

0.058654 seconds ~ 58 ms

Approximately 2X difference.

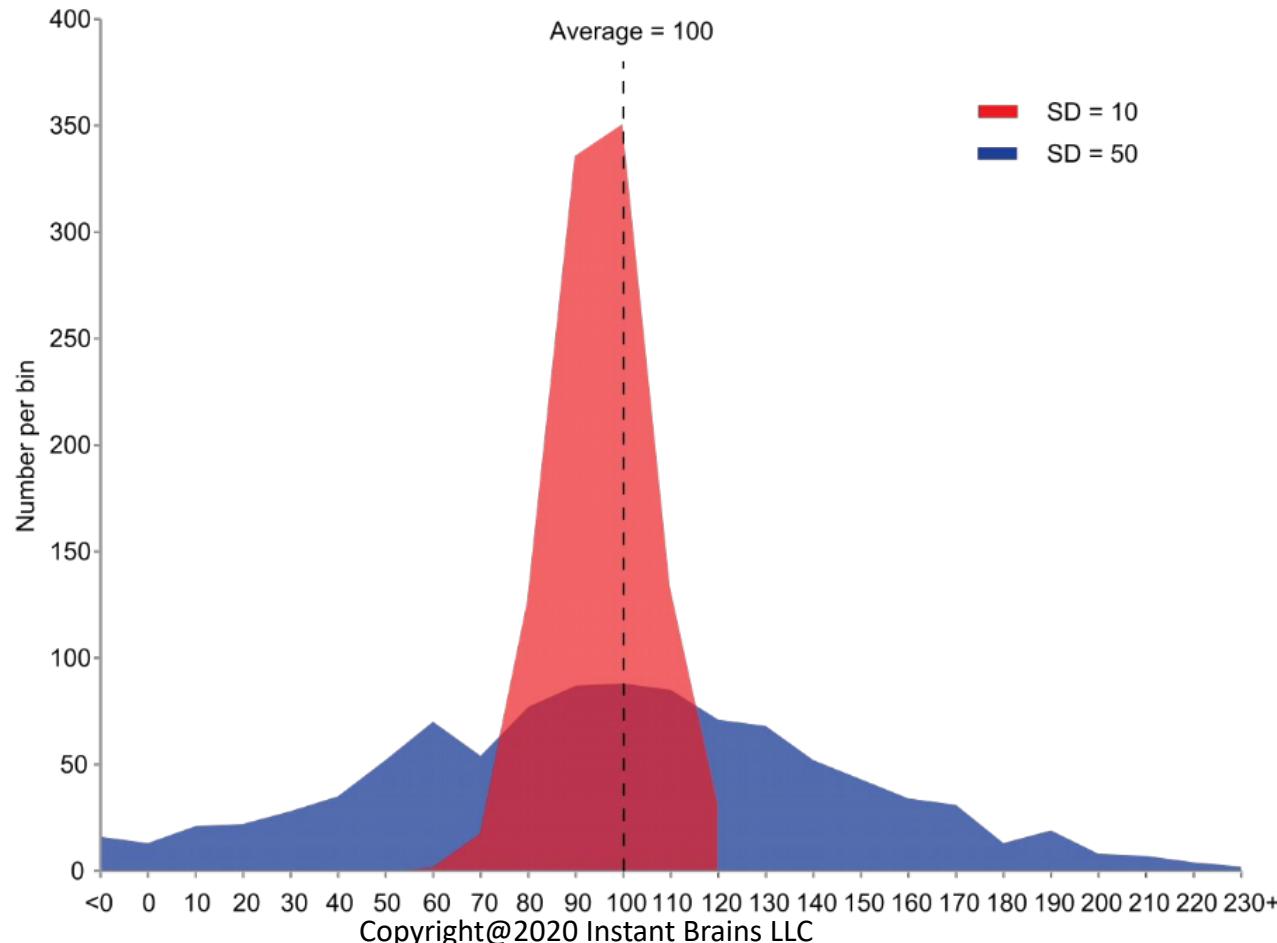
---

# How to compute Standard deviation?

---

# So, how to compute Standard deviation?

The Standard Deviation is a measure of how spread out numbers are.



# So, how to compute Standard deviation?

The Standard Deviation is a measure of how spread out numbers are.

$$\sqrt{\frac{1}{N} \sum_{i=1}^N (x_i - \mu)^2}$$

# So, how to compute Standard deviation?

The Standard Deviation is a measure of how spread out numbers are.

$$\sqrt{\frac{1}{N} \sum_{i=1}^N (x_i - \mu)^2}$$

I. Work out the Mean (the simple average of the numbers) Map, Reduce

# So, how to compute Standard deviation?

The Standard Deviation is a measure of how spread out numbers are.

$$\sqrt{\frac{1}{N} \sum_{i=1}^N (x_i - \mu)^2}$$

1. Work out the Mean (the simple average of the numbers) Map, Reduce
2. Then for each number: subtract the Mean and square the result Map

# So, how to compute Standard deviation?

The Standard Deviation is a measure of how spread out numbers are.

$$\sqrt{\frac{1}{N} \sum_{i=1}^N (x_i - \mu)^2}$$

1. Work out the Mean (the simple average of the numbers) Map, Reduce
2. Then for each number: subtract the Mean and square the result Map
3. Then work out the mean of those squared differences. Map Reduce

# So, how to compute Standard deviation?

The Standard Deviation is a measure of how spread out numbers are.

$$\sqrt{\frac{1}{N} \sum_{i=1}^N (x_i - \mu)^2}$$

1. Work out the Mean (the simple average of the numbers) Map, Reduce
2. Then for each number: subtract the Mean and square the result Map
3. Then work out the mean of those squared differences. Map Reduce
4. Take the square root of that and we are done! Math operation



# “Group” Transformation

# Group Transformation

```
: rdd_mapped = rdd.map(lambda x: (x[0], 1))
print ('Mapped First: ', rdd_mapped.first())
print ('Mapped Count: ', rdd_mapped.count())

# count occurrence of "artist"
rdd_grouped = rdd_mapped.groupBy(lambda w: w[0])
print ('\nGrouped First: ', rdd_grouped.first())
print ('Grouped Count: ', rdd_grouped.count())

print ('\nGrouped first details:')
list(rdd_grouped.first()[1])
```

Mapped First: ('Future', 1)  
Mapped Count: 2017

```
Grouped First: ('Future', <pyspark.resultiterable.ResultIterable object at 0x7f1ca057ced0>)
Grouped Count: 1369
```

Grouped first details:



# Partitions in RDD

## Partitions and re-partitions

```
print ('Current partitions: ', rdd.getNumPartitions())
print ('Record count: ', rdd.count())

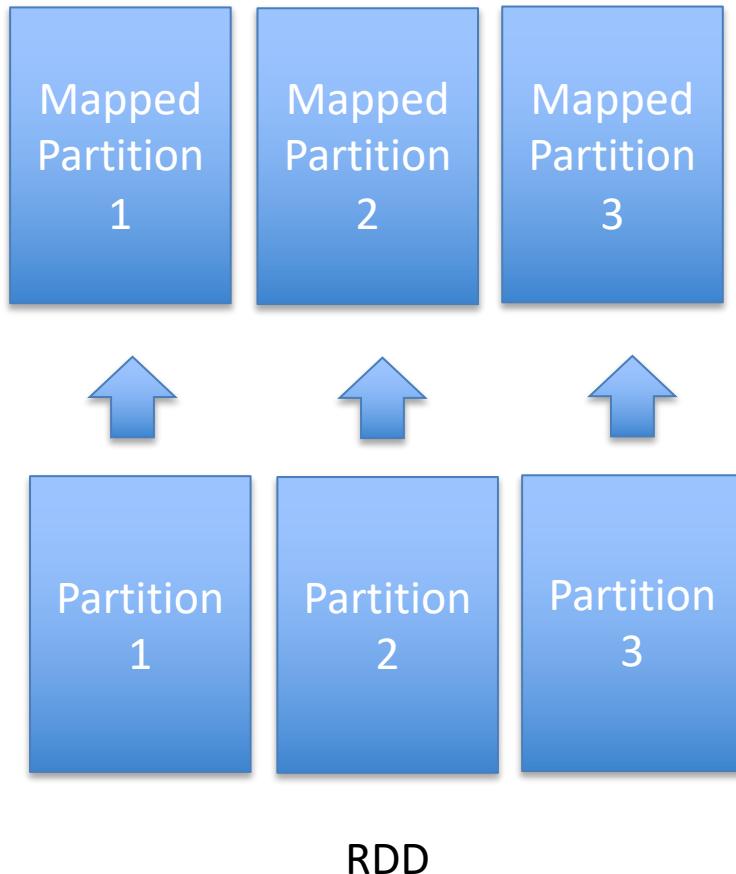
rdd = rdd.repartition(3)
print ('\nNew partitions: ', rdd.getNumPartitions())
print ('Record count: ', rdd.count())
```

```
Current partitions: 1
Record count: 2017
```

```
New partitions: 3
Record count: 2017
```



# mapPartition Transformation



- Map operation over partitions and not over the elements of the partition
- Input is the set of current partitions its output will be another set of partitions.



# mapPartition Transformation

```
rdd_mapped = rdd.map(lambda x: (x[0], 1))
print ('Mapped First: ', rdd_mapped.first())
print ('Mapped Count: ', rdd_mapped.count())

def func(iterator):
    yield list(iterator)[0:5]

rdd_mp = rdd_mapped.mapPartitions(func)
rdd_mp.collect()
```

```
Mapped First:  ('Science Of Sleep', 1)
Mapped Count:  2017
```

```
[[(('Science Of Sleep', 1),
     ('Reaping Asmodeia', 1),
     ('Walking Dead On Broadway', 1),
     ('Darknet', 1),
     ('The Machinist', 1)],
   [('Disclosure', 1),
    ('ILoveMakonnen', 1),
    ('Riff Raff', 1),
    ('Kanye West', 1),
    ('Young M.A.', 1)],
   [('Bro Safari', 1),
    ('MiMOSA', 1),
    ('TR/ST', 1),
    ('Glass Figure', 1),
    ('Dan Croll', 1)]]
```



# “Join” Transformation

## Join Transformation

```
data = [('brocoli', 6), ('melon', 3), ('banana', 1), ('melon', 4), ('brocoli', 13), ('banana', 11)]
label = [('brocoli', 'veggie'), ('melon', 'fruit'), ('banana', 'fruit')]

data_rdd = sc.parallelize(data)
label_rdd = sc.parallelize(label)

print ('Data: ', data_rdd.collect())
print ('\nLabel: ', label_rdd.collect())
```

```
Data:  [('brocoli', 6), ('melon', 3), ('banana', 1), ('melon', 4), ('brocoli', 13), ('banana', 11)]
```

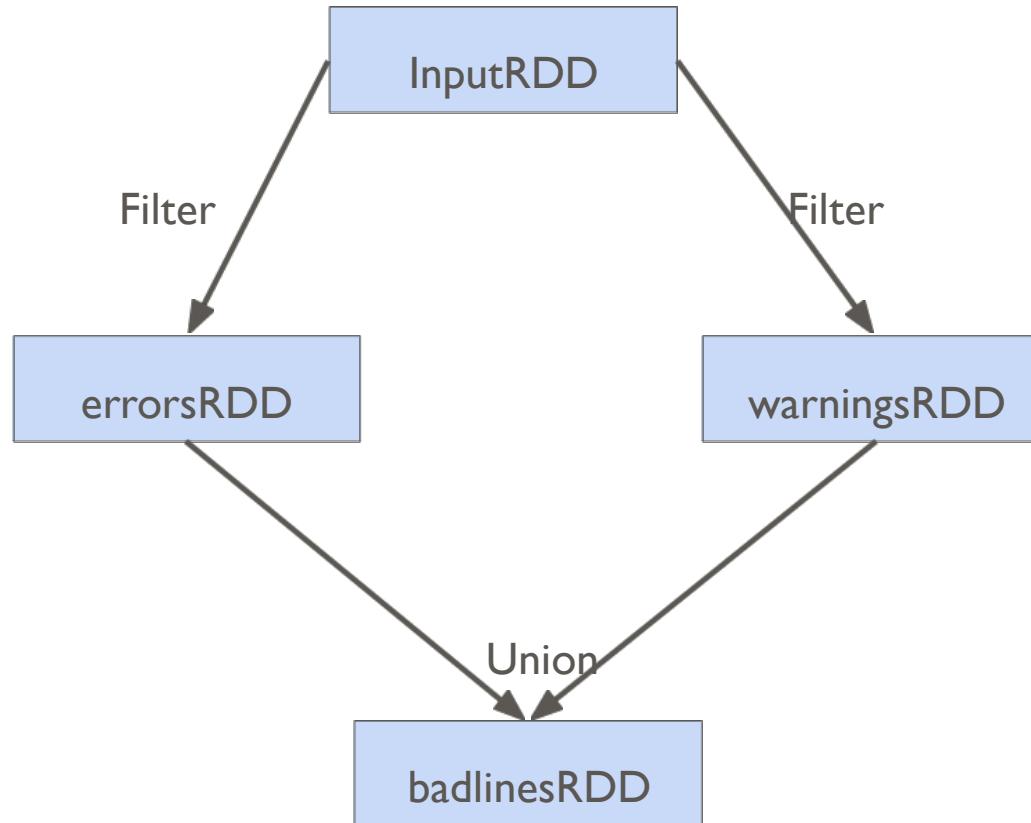
```
Label:  [('brocoli', 'veggie'), ('melon', 'fruit'), ('banana', 'fruit')]
```

```
joined = data_rdd.join(label_rdd).map(lambda x: (x[0], x[1][1], x[1][0]))
joined.collect()
```

```
[('melon', 'fruit', 3),
 ('melon', 'fruit', 4),
 ('melon', 'fruit', 15),
 ('melon', 'fruit', 13),
 ('banana', 'fruit', 1),
 ('banana', 'fruit', 11),
 ('brocoli', 'veggie', 6),
 ('brocoli', 'veggie', 9),
 ('brocoli', 'veggie', 16)]
```



# “Union” Transformation





# “Union” Transformation

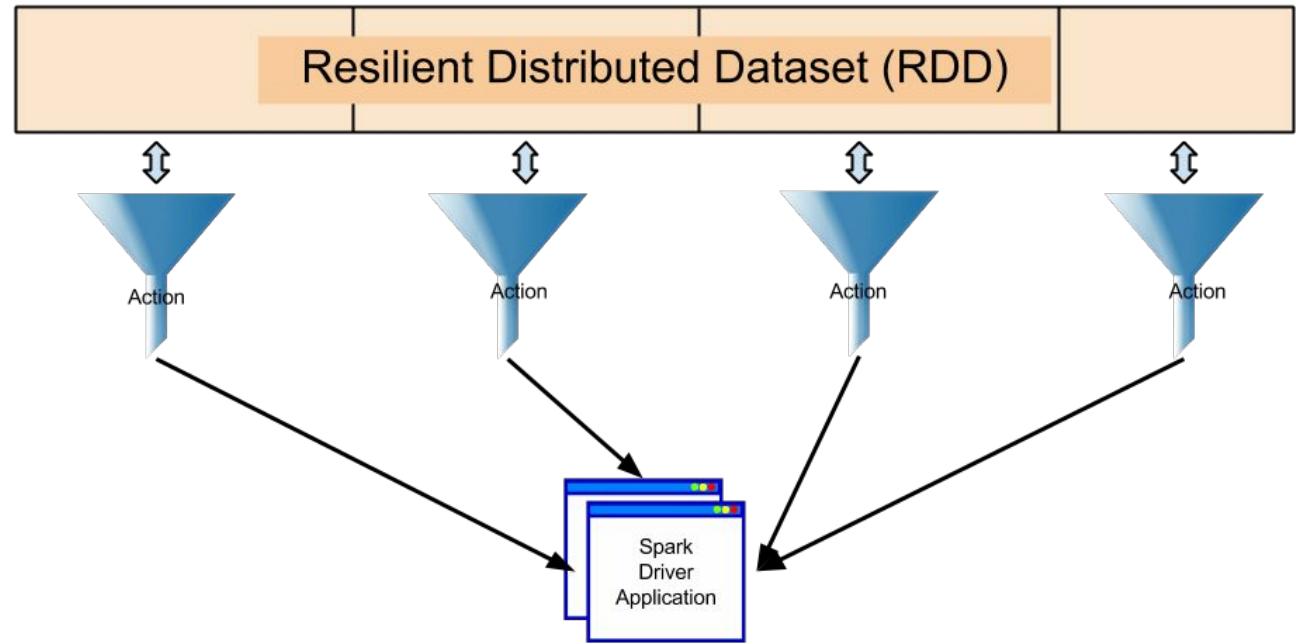
## Union Transformation

```
: print ("UNION")
rdd1 = sc.parallelize([1,2,3])
rdd2 = sc.parallelize(['C', 'D'])
rdd3 = rdd1.union(rdd2)
rdd3.collect()
```

UNION

```
: [1, 2, 3, 'C', 'D']
```

# RDD - Operations :Actions



- Causes the full execution of transformations
- Involves both spark driver as well as the nodes
- Example - Take(): Brings back the data to driver

# RDD - Operations :Actions

Action	Meaning
<code>reduce(func)</code>	Aggregate the elements of the dataset using a function <i>func</i> (which takes two arguments and returns one). The function should be commutative and associative so that it can be computed correctly in parallel.
<code>collect()</code>	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.
<code>count()</code>	Return the number of elements in the dataset.
<code>first()</code>	Return the first element of the dataset (similar to <code>take(1)</code> ).
<code>take(<i>n</i>)</code>	Return an array with the first <i>n</i> elements of the dataset.
<code>takeSample(<i>withReplacement</i>, <i>num</i>, [<i>seed</i>])</code>	Return an array with a random sample of <i>num</i> elements of the dataset, with or without replacement, optionally pre-specifying a random number generator seed.
<code>takeOrdered(<i>n</i>, [<i>ordering</i>])</code>	Return the first <i>n</i> elements of the RDD using either their natural order or a custom comparator.
<code>saveAsTextFile(path)</code>	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 <code>toString</code> on each element to convert it to a line of text in the file.
<code>saveAsSequenceFile(path)</code> (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).

# RDD - Operations :Actions

Action	Meaning
<b>saveAsObjectFile(path)</b> (Java and Scala)	Write the elements of the dataset in a simple format using Java serialization, which can then be loaded using <code>SparkContext.objectFile()</code> .
<b>countByKey()</b>	Only available on RDDs of type (K, V). Returns a hashmap of (K, Int) pairs with the count of each key.
<b>foreach(func)</b>	Run a function <i>func</i> on each element of the dataset. This is usually done for side effects such as updating an <a href="#">Accumulator</a> or interacting with external storage systems. <b>Note:</b> modifying variables other than Accumulators outside of the <code>foreach()</code> may result in undefined behavior. See <a href="#">Understanding closures</a> for more details.

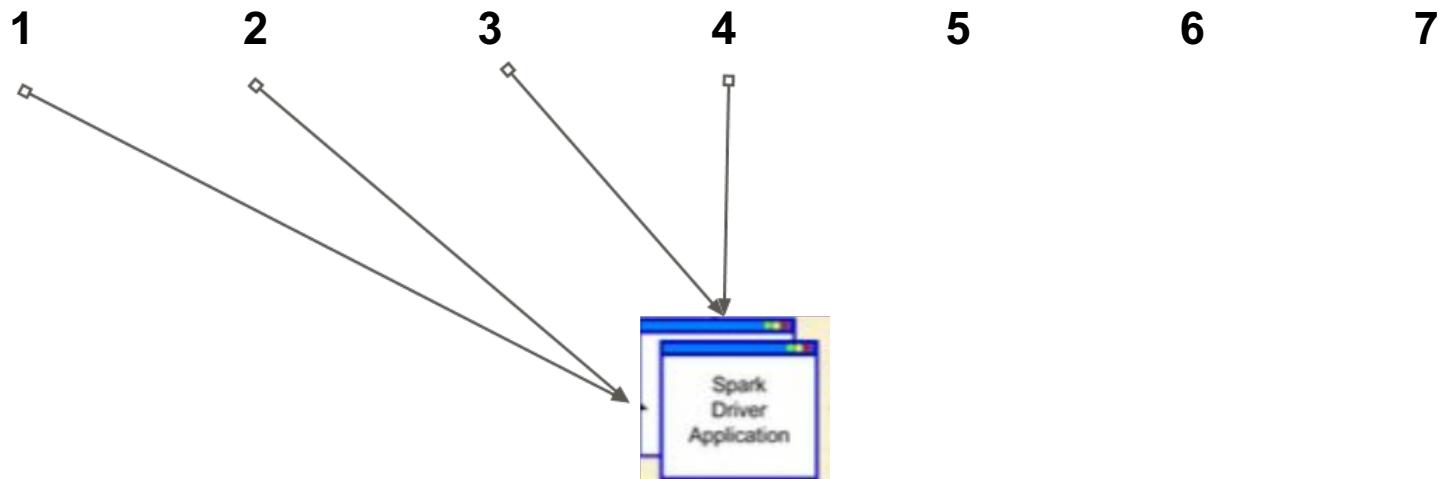


# Actions: take()

Bring only few elements to the driver.

This is more practical than collect()

```
➤ var a = sc.parallelize(Array(1,2,3, 4, 5 , 6, 7));  
➤ var localarray = a.take(4);  
➤ localarray  
[1, 2, 3, 4]
```

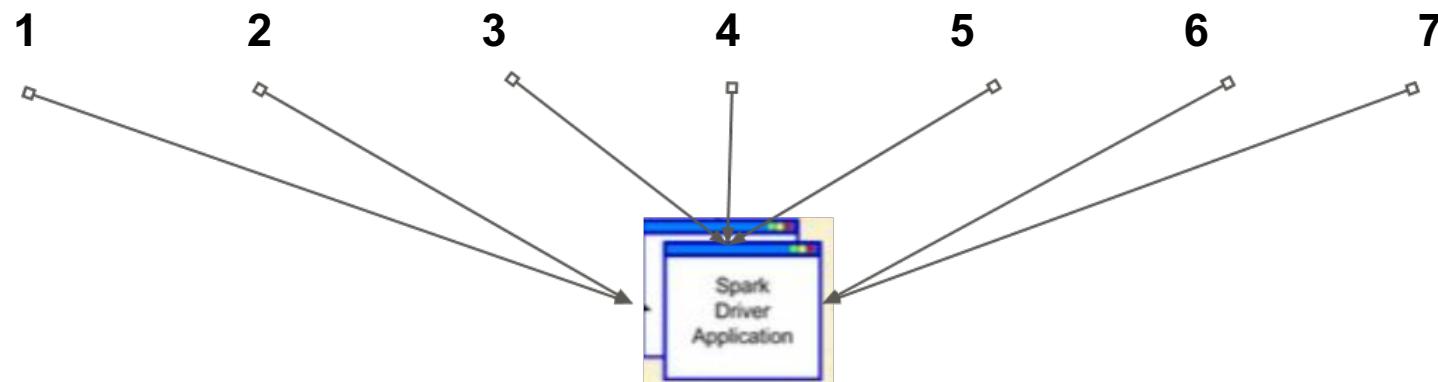




# Actions: collect()

Brings all the elements back to you. Data must fit into memory.  
Mostly it is impractical.

```
> var a = sc.parallelize(Array(1,2,3, 4, 5 , 6, 7));  
> a  
org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[16] at parallelize at <console>:21  
> var localarray = a.collect();  
> localarray  
[1, 2, 3, 4, 5, 6, 7]
```





# Action Example - saveAsTextFile()

To save the results in HDFS or Any other file system  
Call **saveAsTextFile(directoryName)**

It would create directory  
And save the results inside it  
If directory exists, it would throw error.

The screenshot shows the Hue File Browser interface. The top navigation bar includes links for Home, Query Editors, Data Browsers, Workflows, Search, File Browser (selected), Job Browser, and a user profile. The main area is titled "File Browser" and shows a directory structure under "/ user / sandeepgiri9034 / mydirectory". A search bar, actions menu, and move to trash button are at the top of the list view. The list view displays the following files and their details:

Name	Size	User	Group	Permissions	Date
_SUCCESS	0 bytes	sandeepgiri9034	hdfs	-rw-r--r--	March 30, 2017 06:50 AM
part-00000	947 bytes	sandeepgiri9034	hdfs	-rw-r--r--	March 30, 2017 06:50 AM
part-00001	1001 bytes	sandeepgiri9034	hdfs	-rw-r--r--	March 30, 2017 06:50 AM
part-00002	1.2 KB	sandeepgiri9034	hdfs	-rw-r--r--	March 30, 2017 06:50 AM
part-00003	1.2 KB	sandeepgiri9034	hdfs	-rw-r--r--	March 30, 2017 06:50 AM



# Action Example - saveAsTextFile()

```
val arr = 1 to 1000  
val nums = sc.parallelize(arr)  
def multipleByTwo(x:Int):Int = x*2
```

```
var dbls = nums.map(multipleByTwo);  
dbls.saveAsTextFile("mydirectory")
```

Check the HDFS home directory

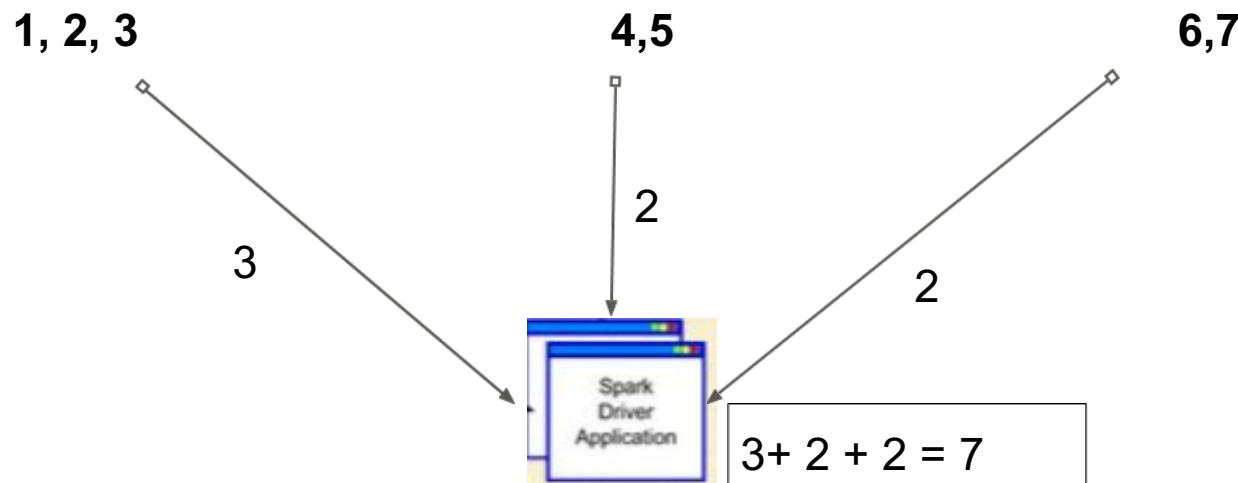
The screenshot shows the Hue File Browser interface. The URL in the address bar is `d.cloudxlab.com/filebrowser/view/user/sandeepgiri9034/mydirectory/part-00000`. The browser title bar includes the Hue logo and navigation links for Query Editors, Data Browsers, Workflows, and Search. The main area is titled "File Browser". On the left, there is a sidebar with "ACTIONS" buttons: View as binary, Edit file, Download, View file location, and Refresh. The main content area shows the path `/ user / sandeepgiri9034 / mydirectory / part-00000`. The file contains the following text:  
2  
4  
6  
8  
10  
12  
14  
16



# Actions: count()

To find out how many elements are there in an RDD.  
Works in distributed fashion.

```
> var a = sc.parallelize(Array(1,2,3, 4, 5 , 6, 7), 3);  
> var mycount = a.count();  
> mycount  
7
```



# Computing random sample from a dataset

The objective of the exercise is to pick a random sample from huge data. Though there is a method provided in RDD but we are creating our own.

# Computing random sample from a dataset

The objective of the exercise is to pick a random sample from huge data. Though there is a method provided in RDD but we are creating our own.

I. Lets try to understand it for say picking 50% records.

# Computing random sample from a dataset

The objective of the exercise is to pick a random sample from huge data. Though there is a method provided in RDD but we are creating our own.

1. Lets try to understand it for say picking 50% records.
2. The approach is very simple. We pick a record from RDD and do a coin toss. If its head, keep the element otherwise discard it. It can be achieved using filter.

# Computing random sample from a dataset

The objective of the exercise is to pick a random sample from huge data. Though there is a method provided in RDD but we are creating our own.

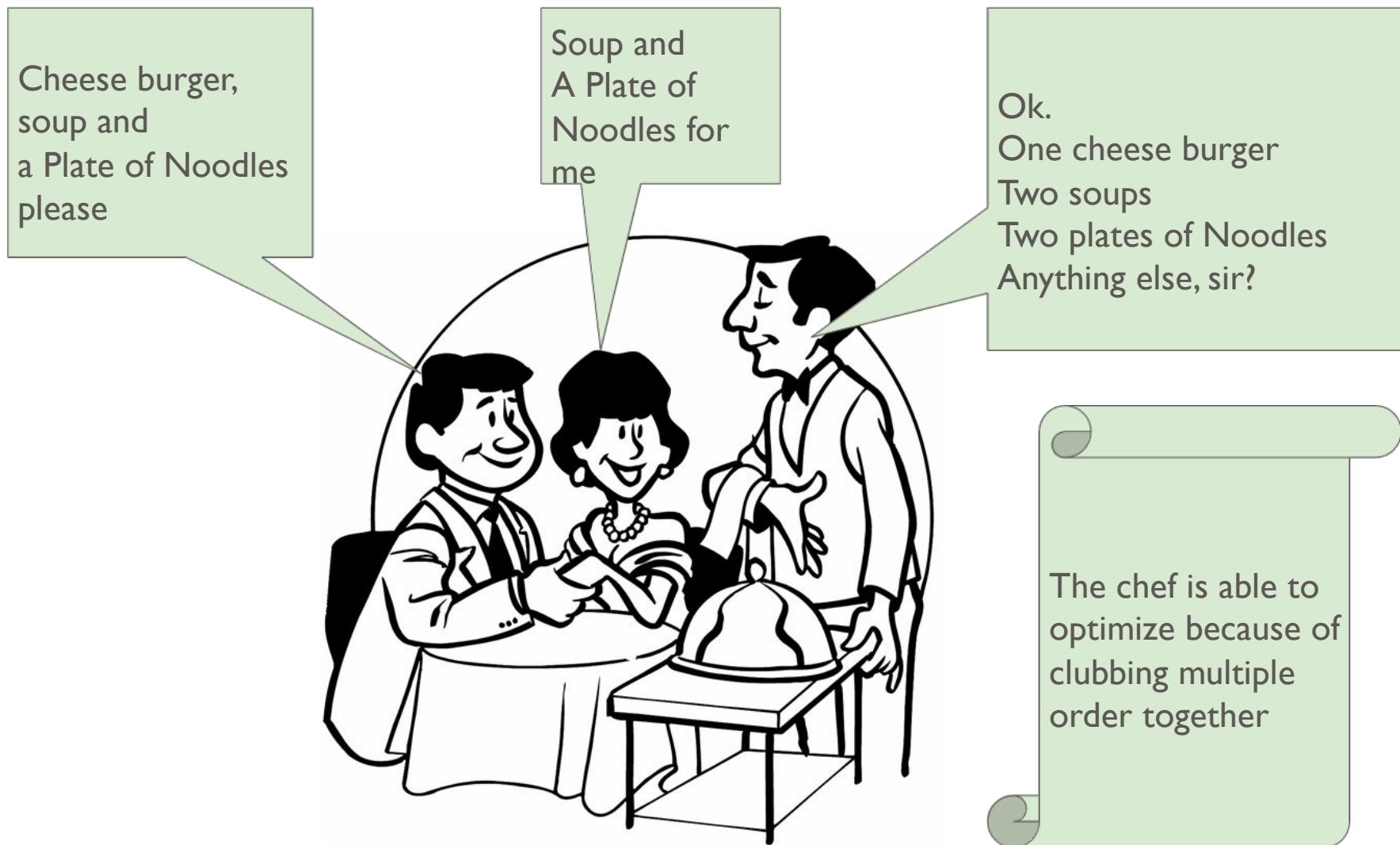
- I. Lets try to understand it for say picking 50% records.
2. The approach is very simple. We pick a record from RDD and do a coin toss. If its head, keep the element otherwise discard it. It can be achieved using filter.
3. For picking any fraction, we might use a coin having 100s of faces or in other words a random number generator.

# Computing random sample from a dataset

The objective of the exercise is to pick a random sample from huge data. Though there is a method provided in RDD but we are creating our own.

1. Lets try to understand it for say picking 50% records.
2. The approach is very simple. We pick a record from RDD and do a coin toss. If its head, keep the element otherwise discard it. It can be achieved using filter.
3. For picking any fraction, we might use a coin having 100s of faces or in other words a random number generator.
4. Please notice that it would not give the sample of exact size

# Lazy Evaluation Example - The waiter takes orders patiently



# Instant Evaluation

Let me get a cheese burger  
for you. I'll be right back!



Cheese Burger...

And Soup?

The soup order will be taken once the waiter is back.

# Instant Evaluation

---

The usual programming languages have instant evaluation.

As you type:

**var x = 2+10.**

It doesn't wait. It immediately evaluates.

# Actions: Lazy Evaluation

---

1. Every time we call an action, entire RDD must be computed from scratch
2. Everytime d gets executed, a,b,c would be run
  - a. `lines = sc.textFile("myfile");`
  - b. `fewlines = lines.filter(...)`
  - c. `uppercaseLines = fewlines.map(...)`
  - d. `uppercaseLines.count()`
3. When we call a transformation, it is not evaluated immediately.
4. It helps Spark optimize the performance
5. Similar to Pig, tensorflow etc.
6. Instead of thinking RDD as dataset, think of it as the instruction on how to compute data



# Actions: Lazy Evaluation - Optimization

```
def Map1(x:String):String =  
x.trim();  
  
def Map2(x:String):String =  
x.toUpperCase();  
  
var lines = sc.textFile(...)  
var lines1 = lines.map(Map1);  
var lines2 = lines1.map(Map2);  
  
lines2.collect()
```



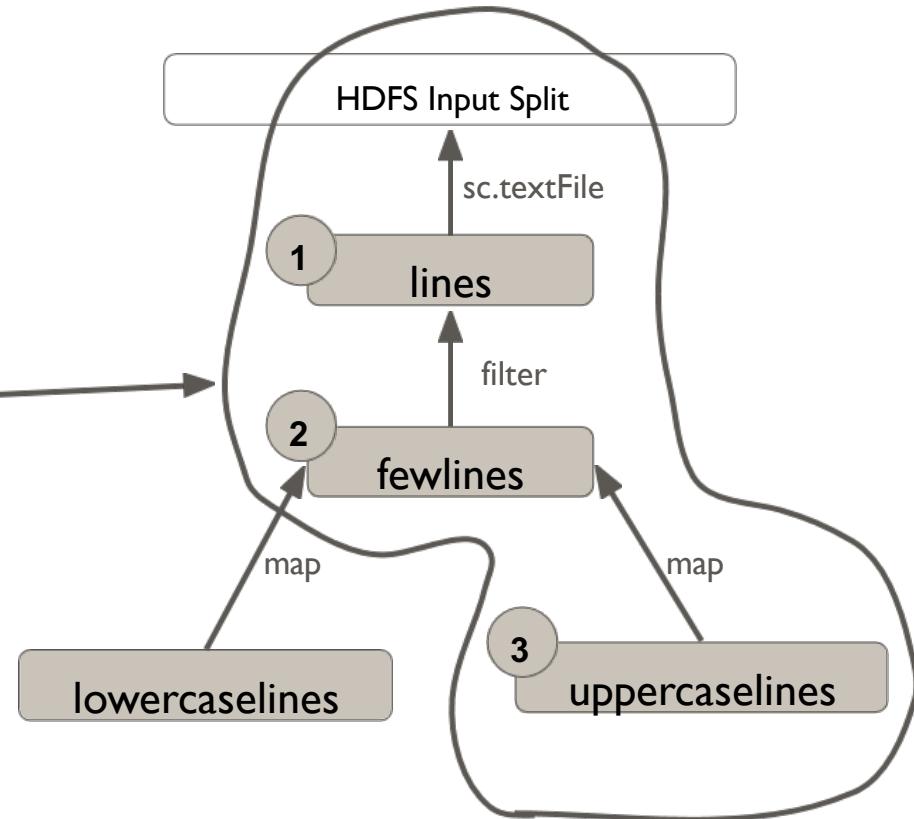
```
def Map3(x:String):String={  
    var y = x.trim();  
    return y.toUpperCase();  
}  
  
lines = sc.textFile(...)  
lines2 = lines.map(Map3);  
  
lines2.collect()
```

# Lineage Graph

Spark Code

```
lines = sc.textFile("myfile");
fewlines = lines.filter(...)
uppercaselines = fewlines.map(...)
lowercaselines = fewlines.map(...)
uppercaselines.count()
```

Lineage Graph



# **4. Spark for Structured Data Processing and Data Sources**

# Spark SQL

## **Integrated**

- Provides **DataFrames**
- Mix SQL queries & Spark programs

# Spark SQL

---

## Uniform Data Access

- Source:
  - HDFS,
  - Hive
  - Relational Databases
- Avro, Parquet, ORC, JSON
- You can even join data across these sources.
- Hive Compatibility
- Standard Connectivity

# DataFrames

## RDD

1 sandeep

2 ted

3 thomas

4 priya

5 kush

## Unstructured

Need code for processing

## Data Frame

### ID Name

1 sandeep

2 ted

3 thomas

4 priya

5 kush

## Structured

Can use SQL or R like syntax:  
`df.sql("select Id where name = 'priya'")`

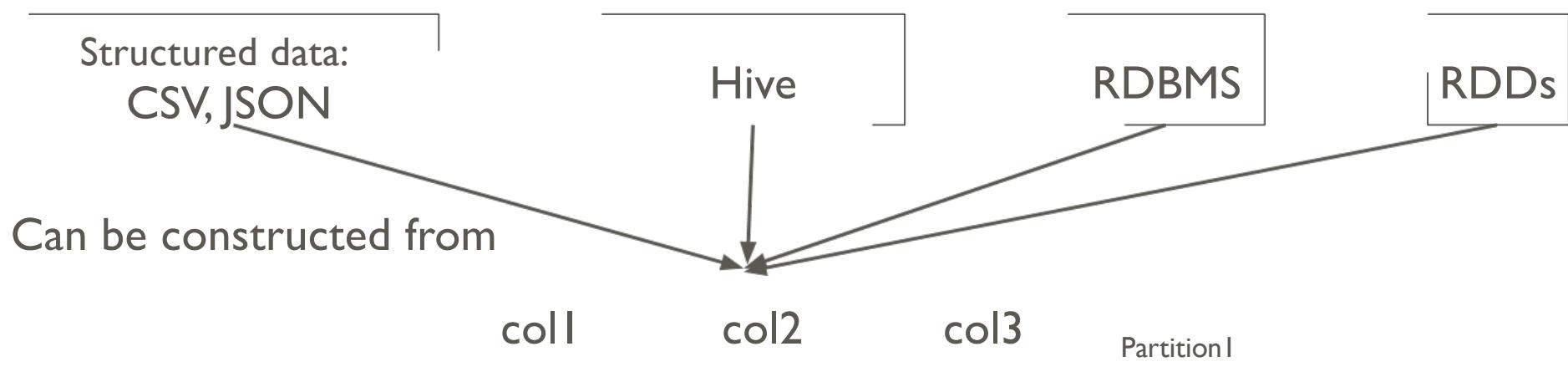
`head(where(df, df$ID > 2))`

# Data Frames



- Collection with named columns
- Distributed
- <> Same as database table
- <> A data frame in Python

# Data Frames



# Data Frames

col1      col2      col3  
Partition 1



05/1/20



Copyright@2020 Instant Brains LLC



Partition2



---

# Starting Point: SparkSession

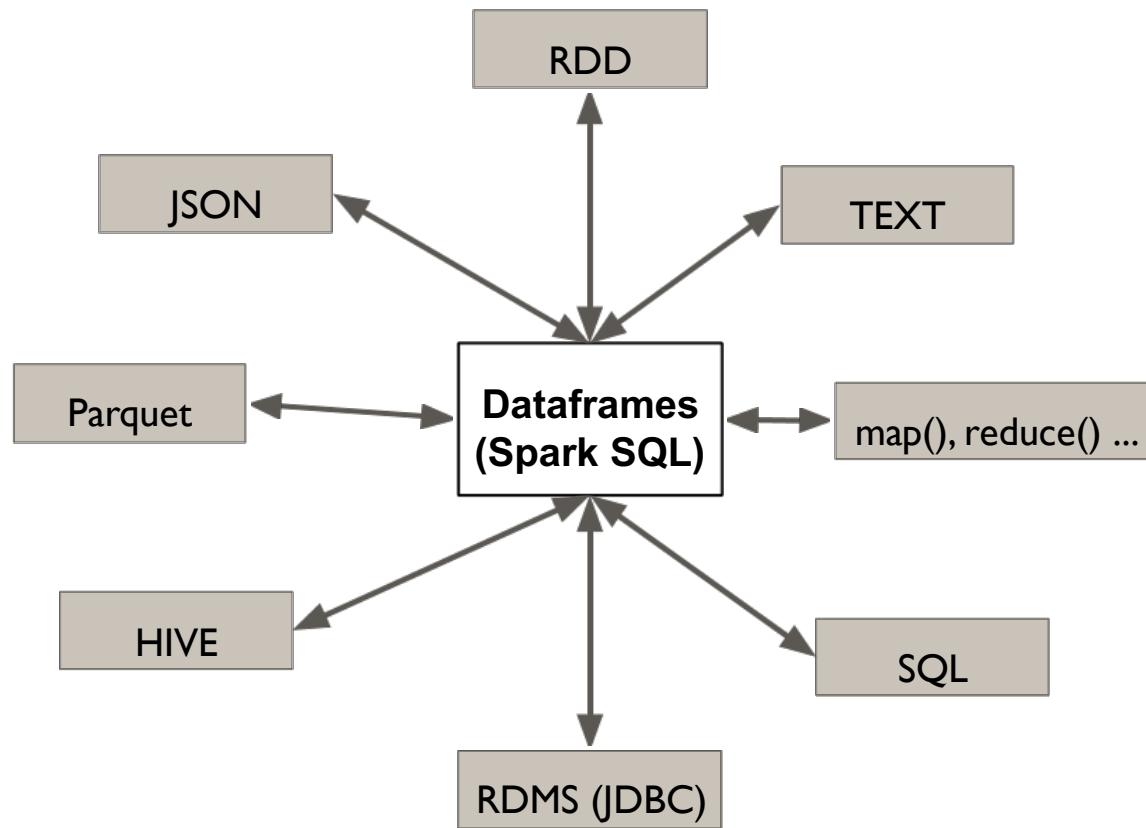
---

```
from pyspark.sql import SparkSession
```

```
spark = SparkSession.builder.appName("Spark SQL basic example").\nenableHiveSupport().getOrCreate()
```

```
sc = spark.sparkContext
```

# Data Frames





# Dataframe Loading: csv

## Load CSV file to DF

```
df = spark.read.option("header", "true").option("mode", "DROPMALFORMED").csv(csv_file)
# df = spark.read.csv(csv_file, header=True, sep=",")
# df = spark.read.format("csv").option("header", "true").load(csv_file)

print ('columns: ', df.columns)
print ('count: ', df.count())
print ('\nschema: ')
df.printSchema()
df.show(5)
```

columns: ['acousticness', 'danceability', 'duration\_ms', 'energy', 'song\_title', 'artist']  
count: 2017

```
schema:  
root  
|-- acousticness: string (nullable = true)  
|-- danceability: string (nullable = true)  
|-- duration_ms: string (nullable = true)  
|-- energy: string (nullable = true)  
|-- song_title: string (nullable = true)  
|-- artist: string (nullable = true)
```

acousticness	danceability	duration_ms	energy	song_title	artist
0.0102	0.833	204600	0.434	Mask Off	Future
0.199	0.743	326933	0.359	Redbone	Childish Gambino
0.0344	0.838	185707	0.412	Xanny Family	Future
0.604	0.494	199413	0.338	Master Of None	Beach House
0.18	0.678	392893	0.561	Parallel Lines	Junior Boys



# Dataframe Loading: json

## Load JSON file to DF

```
df = spark.read.option("header", "true").option("mode", "DROPMALFORMED").json(json_file)

print ('columns: ', df.columns)
print ('count: ', df.count())
print ('\nschema: ')
df.printSchema()
df.show(5)

columns:  ['acousticness', 'artist', 'danceability', 'duration_ms', 'energy', 'song_title']
count:  2017
```

```
schema:
root
|-- acousticness: string (nullable = true)
|-- artist: string (nullable = true)
|-- danceability: string (nullable = true)
|-- duration_ms: string (nullable = true)
|-- energy: string (nullable = true)
|-- song_title: string (nullable = true)
```

acousticness	artist	danceability	duration_ms	energy	song_title
0.0102	Future	0.833	204600	0.434	Mask Off
0.199	Childish Gambino	0.743	326933	0.359	Redbone
0.0344	Future	0.838	185707	0.412	Xanny Family
0.604	Beach House	0.494	199413	0.338	Master Of None
0.18	Junior Boys	0.678	392893	0.561	Parallel Lines



# Dataframe Loading: rdd

## Convert RDD to DF

```
# read csv file as rdd
rdd = sc.textFile(csv_file)
header = rdd.first()
rdd = rdd.filter(lambda x : x != header).map(lambda x : x.split(","))
cols = header.split(",")
df = rdd.toDF(cols)

print ('columns: ', df.columns)
print ('\nschema: ')
df.printSchema()
df.show(5)
```

```
columns:  ['acousticness', 'danceability', 'duration_ms', 'energy', 'song_title', 'artist']
```

```
schema:
```

```
root
  |-- acousticness: string (nullable = true)
  |-- danceability: string (nullable = true)
  |-- duration_ms: string (nullable = true)
  |-- energy: string (nullable = true)
  |-- song_title: string (nullable = true)
  |-- artist: string (nullable = true)
```

acousticness	danceability	duration_ms	energy	song_title	artist
0.0102	0.833	204600	0.434	Mask Off	Future
0.199	0.743	326933	0.359	Redbone	Childish Gambino
0.0344	0.838	185707	0.412	Xanny Family	Future
0.604	0.494	199413	0.338	Master Of None	Beach House
0.18	0.678	392893	0.561	Parallel Lines	Junior Boys



# Dataframe Loading: views

```
: spark.sql("show databases").show()  
spark.sql("show tables in default").show()
```

```
+-----+  
|databaseName|  
+-----+  
|      default|  
+-----+
```

```
+-----+-----+-----+  
|database|tableName|isTemporary|  
+-----+-----+-----+  
+-----+-----+-----+
```



# Dataframe Loading: views

## Load DF from Temporary View

```
: df.createOrReplaceTempView("spotify")

# SQL statements can be run by using the sql methods provided by spark
df = spark.sql("SELECT * FROM spotify")
print ('columns: ', df.columns)
print ('\nschema: ')
df.printSchema()
df.show(5)

columns:  ['acousticness', 'danceability', 'duration_ms', 'energy', 'song_title', 'artist']

schema:
root
|--- acousticness: string (nullable = true)
|--- danceability: string (nullable = true)
|--- duration_ms: string (nullable = true)
|--- energy: string (nullable = true)
|--- song_title: string (nullable = true)
|--- artist: string (nullable = true)

+-----+-----+-----+-----+-----+
| acousticness | danceability | duration_ms | energy | song_title | artist |
+-----+-----+-----+-----+-----+
| 0.0102 | 0.833 | 204600 | 0.434 | Mask Off | Future |
| 0.199 | 0.743 | 326933 | 0.359 | Redbone | Childish Gambino |
| 0.0344 | 0.838 | 185707 | 0.412 | Xanny Family | Future |
| 0.604 | 0.494 | 199413 | 0.338 | Master Of None | Beach House |
| 0.18 | 0.678 | 392893 | 0.561 | Parallel Lines | Junior Boys |
+-----+-----+-----+-----+-----+
only showing top 5 rows
```



# Dataframe Loading: Delete Views

```
print ("Before Delete")
spark.sql("show databases").show()
spark.sql("show tables in default").show()

spark.sql("DROP TABLE spotify")
```

```
print ("After Delete")
spark.sql("show databases").show()
spark.sql("show tables in default").show()
```

```
Before Delete
```

```
+-----+
|databaseName|
+-----+
|      default|
+-----+
```

```
+-----+-----+-----+
|database|tableName|isTemporary|
+-----+-----+-----+
|          | spotify|     true|
+-----+-----+-----+
```

```
After Delete
```

```
+-----+
|databaseName|
+-----+
|      default|
+-----+
```

```
+-----+-----+-----+
|database|tableName|isTemporary|
+-----+-----+-----+
|          |          |
```

# Dataframe Loading: Databases

---

- Spark SQL also includes a data source that can read data from DBs using JDBC.
- Results are returned as a DataFrame
- Easily be processed in Spark SQL or joined with other data sources



# Dataframe Loading: Databases

## Load DF using JDBC

```
#### Include necessary jars
./bin/pyspark
--driver-class-path mysql-connector-java-5.1.36-bin.jar
--jars mysql-connector-java-5.1.36-bin.jar

df = spark.read\
    .format("jdbc")\
    .option("url", "jdbc:mysql://localhost/database_name")\
    .option("driver", "com.mysql.jdbc.Driver")\
    .option("dbtable", "employees").option("user", "root")\
    .option("password", "12345678").load()
```

## Load DF from MongoDB

```
#### Include necessary jars
./bin/pyspark
--driver-class-path mongo-spark-connector_2.11.jar
--jars mongo-spark-connector_2.11.jar

df = spark.read.format("mongo").option("uri", "mongodb://127.0.0.1/people.contacts").load()
```

# AVRO

---

## Avro is:

1. A Remote Procedure call
2. Data Serialization Framework
3. Uses JSON for defining data types and protocols
4. Serializes data in a compact binary format
5. Similar to Thrift and Protocol Buffers
6. Doesn't require running a code-generation program

Its primary use is in Apache Hadoop, where it can provide both a serialization format for **persistent** data, and a **wire format** for communication between Hadoop nodes, and from client programs to the Hadoop services.

Apache Spark SQL can access Avro as a data source.[1]

# Loading AVRO

---

<https://github.com/databricks/spark-avro>

**Start Spark-Shell:**

*/usr/spark2.0.1/bin/spark-shell --packages com.databricks:spark-avro\_2.11:3.2.0*

# Loading AVRO

<https://github.com/databricks/spark-avro>

## Start Spark-Shell:

```
/usr/spark2.0.1/bin/spark-shell --packages com.databricks:spark-avro_2.11:3.2.0
```

## Load the Data:

```
val df = spark.read.format("com.databricks.spark.avro")
    .load("/data/spark/episodes.avro")
```

## Display Data:

```
df.show()
```

title	air_date	doctor
The Eleventh Hour	3 April 2010	11
The Doctor's Wife	14 May 2011	11



# Data Sources

- Columnar storage format
- Any project in the Hadoop ecosystem
- Regardless of
  - Data processing framework
  - Data model
  - Programming language.

<https://parquet.apache.org/>



# Data Sources

## **Method1 - Automatically (parquet unless otherwise configured)**

```
var df = spark.read.load("/data/spark/users.parquet")
```



# Data Sources

## Method2 - Manually Specifying Options

```
df = spark.read.format("json").load("/data/spark/people.json")
df = df.select("name", "age")
df.write.format("parquet").save("namesAndAges.parquet")
```

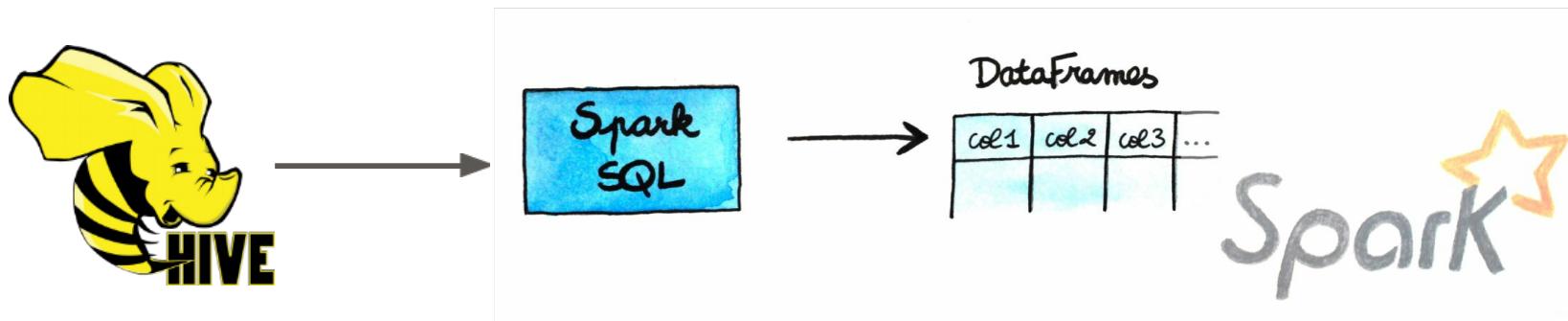


# Data Sources

## Method3 - Directly running sql on file

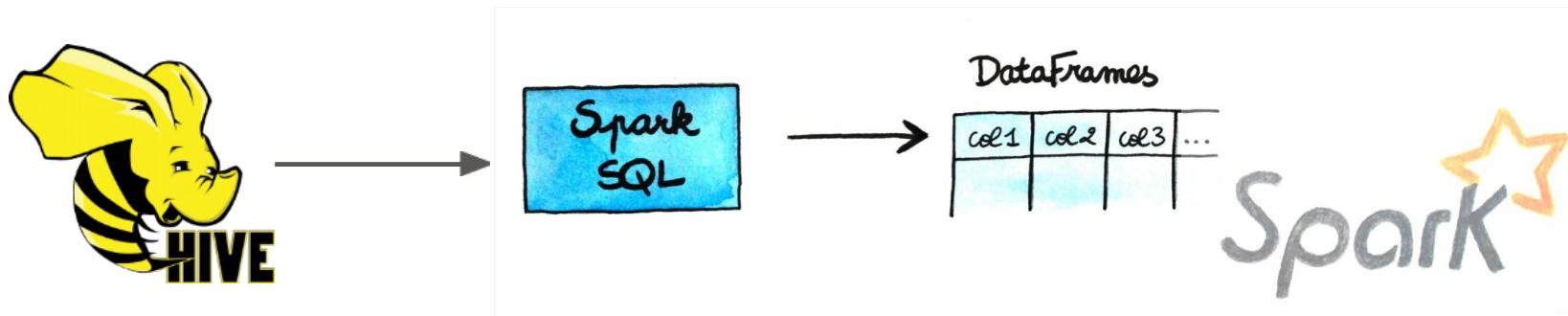
```
val sqlDF = spark.sql("SELECT * FROM parquet.`/data/spark/users.parquet`")
val sqlDF = spark.sql("SELECT * FROM json.`/data/spark/people.json`")
```

# Hive Tables



- Spark SQL also supports reading and writing data stored in Apache Hive.
- Since Hive has a large number of dependencies, it is not included in the default Spark assembly.

# Hive Tables



- Place your `hive-site.xml`, `core-site.xml` and `hdfs-site.xml` file in `conf/`

# Hive Tables - Example

```
import java.io.File

val spark = SparkSession
  .builder()
  .appName("Spark Hive Example")
.enableHiveSupport()
  .getOrCreate()
```

# Hive Tables - Example

```
/usr/spark2.0.1/bin/spark-shell
```

```
scala> import spark.implicits._  
import spark.implicits._
```

```
scala> var df = spark.sql("select * from a_student")  
scala> df.show()
```

	name	grade	marks	stream
1	Student1	A	1	CSE
2	Student2	B	2	IT
3	Student3	A	3	ECE
4	Student4	B	4	EEE
5	Student5	A	5	MECH
6	Student6	B	6	CHEM

# SequenceFiles

---

- Popular Hadoop format
  - For handling small files
  - Create InputSplits without too much transport
- Composed of flat files with key/value pairs.
- Has Sync markers
  - Allow to seek to a point
  - Then resynchronize with the record boundaries
  - Allows Spark to efficiently read in parallel from multiple nodes

# Loading SequenceFiles - Example

```
import org.apache.hadoop.io.DoubleWritable  
import org.apache.hadoop.io.Text  
  
rdd = sc.sequenceFile("pysequencefile1", classOf[Text],  
classOf[DoubleWritable])  
  
result = myrdd.map{case (x, y) => (x.toString, y.get())}  
result.collect()  
  
//Array((key1,1.0), (key2,2.0), (key3,3.0))
```

# Saving SequenceFiles - Example

```
var rdd = sc.parallelize(Array(("key1", 1.0), ("key2", 2.0), ("key3", 3.0)))
rdd.saveAsSequenceFile("pysequencefile1")
```

```
[sandeep@ip-172-31-60-179 ~]$ hadoop fs -ls pysequencefile1/
Found 5 items
-rw-r--r--  3 sandeep hdfs          0 2017-06-21 21:52 pysequencefile1/_SUCCESS
-rw-r--r--  3 sandeep hdfs         88 2017-06-21 21:52 pysequencefile1/part-00000
-rw-r--r--  3 sandeep hdfs        109 2017-06-21 21:52 pysequencefile1/part-00001
-rw-r--r--  3 sandeep hdfs        109 2017-06-21 21:52 pysequencefile1/part-00002
-rw-r--r--  3 sandeep hdfs        109 2017-06-21 21:52 pysequencefile1/part-00003
```

# Pickle File

- Python way of handling object files
- Uses Python's pickle serialization library
- Saving - saveAsPickleFile() on an RDD
- Loading - pickleFile() on SparkContext
- Can also be quite slow as Object FIELDS

# Protocol buffers

- Developed at Google for internal RPCs
- Open sourced
- Structured data - fields & types of fields defined
- Fast for encoding and decoding (20-100x than XML)
- Take up the minimum space (3-10x than xml)
- Defined using a domain-specific language
- Compiler generates accessor methods in variety of languages
- Consist of fields: optional, required, or repeated
- While parsing
  - A missing optional field => success
  - A missing required field => failure
- So, make new fields as optional (remember object file failures?)

# Protocol buffers - Example

```
package tutorial;
message Person {
    required string name = 1;
    required int32 id = 2;
    optional string email = 3;
    enum PhoneType {
        MOBILE = 0;
        HOME = 1;
        WORK = 2;
    }
    message PhoneNumber {
        required string number = 1;
        optional PhoneType type = 2 [default = HOME];
    }
    repeated PhoneNumber phone = 4;
}

message AddressBook {
    repeated Person person = 1;
}
```

# Protocol buffers - Steps

---

1. [Download](#) and install protocol buffer compiler
2. pip install protobuf
3. protoc -I=\$SRC\_DIR --python\_out=\$DST\_DIR  
\$SRC\_DIR/addressbook.proto
4. Create objects
5. Convert those into protocol buffers

---

# Loading + Saving Data: File Systems

## Local/“Regular” FS

1. `rdd = sc.textFile("file:///home/holden/happypandas.gz")`
2. The path has to be available on all nodes.  
Otherwise, load it locally and distribute using `sc.parallelize`

# Amazon S3

1. Popular option
2. Good if nodes are inside EC2
3. Use path in all input methods (textFile, hadoopFile etc)  
s3n://bucket/path-within-bucket
4. Set Env.Vars: AWS\_ACCESS\_KEY\_ID

AWS\_SECRET\_ACCESS\_KEY

More details

<https://sparkbyexamples.com/spark/spark-read-text-file-from-s3/>

# HDFS

1. The Hadoop Distributed File System
2. Spark and HDFS can be collocated on the same machines
3. Spark can take advantage of this data locality to avoid network overhead
4. In all i/o methods, use path: hdfs://master:port/path
5. Use only the version of spark w.r.t HDFS version

# File Compression

---

1. To Save Storage & Network Overhead
2. With most hadoop output formats we can specify compression codecs
3. Compression should not require the whole file at once
4. Each worker can find start of record => splitable

# File Compression Options

---

**Characteristics of compression:**

- Splittable
- Speed
- Effectiveness on Text
- Code

# File Compression Options

<b>Format</b>	<b>Splittable</b>	<b>Speed</b>	<b>Effectiveness on text</b>	<b>Hadoop compression codec</b>	<b>comments</b>
gzip	N	Fast	High	org.apache.hadoop.io.com.GzipCodec	
<i>lzo</i>	Y	V. Fast	Medium	<i>com.hadoop.compression.lzo.LzoCodec</i>	<i>LZO requires installation on every worker node</i>
<b>bzip2</b>	Y	Slow	V. High	org.apache.hadoop.io.com.BZip2Codec	<i>Uses pure Java for splittable version</i>
zlib	N	Slow	Medium	org.apache.hadoop.io.com.DefaultCodec	Default compression codec for Hadoop
Snappy	N	V. Fast	Low	org.apache.hadoop.io.com.SnappyCodec	There is a pure Java port of Snappy but it is not yet available in Spark/ Hadoop

# **5. DataFrames Operations**

# DataFrame Operations

## Operations

- Read to dataframe, write from dataframe
- Manipulating records and columns
- Columns – dropping, renaming, filtering
- Records – dropping, filtering
- Joins
- Statistics
- EDA – Exploratory Data Analysis
- Plotting

[https://github.com/shekhar2010us/spark-examples/blob/master/notebooks/tutorial/2\\_sql\\_tutorial.ipynb](https://github.com/shekhar2010us/spark-examples/blob/master/notebooks/tutorial/2_sql_tutorial.ipynb)

# **6. Apache Spark Accumulators and Broadcasting**

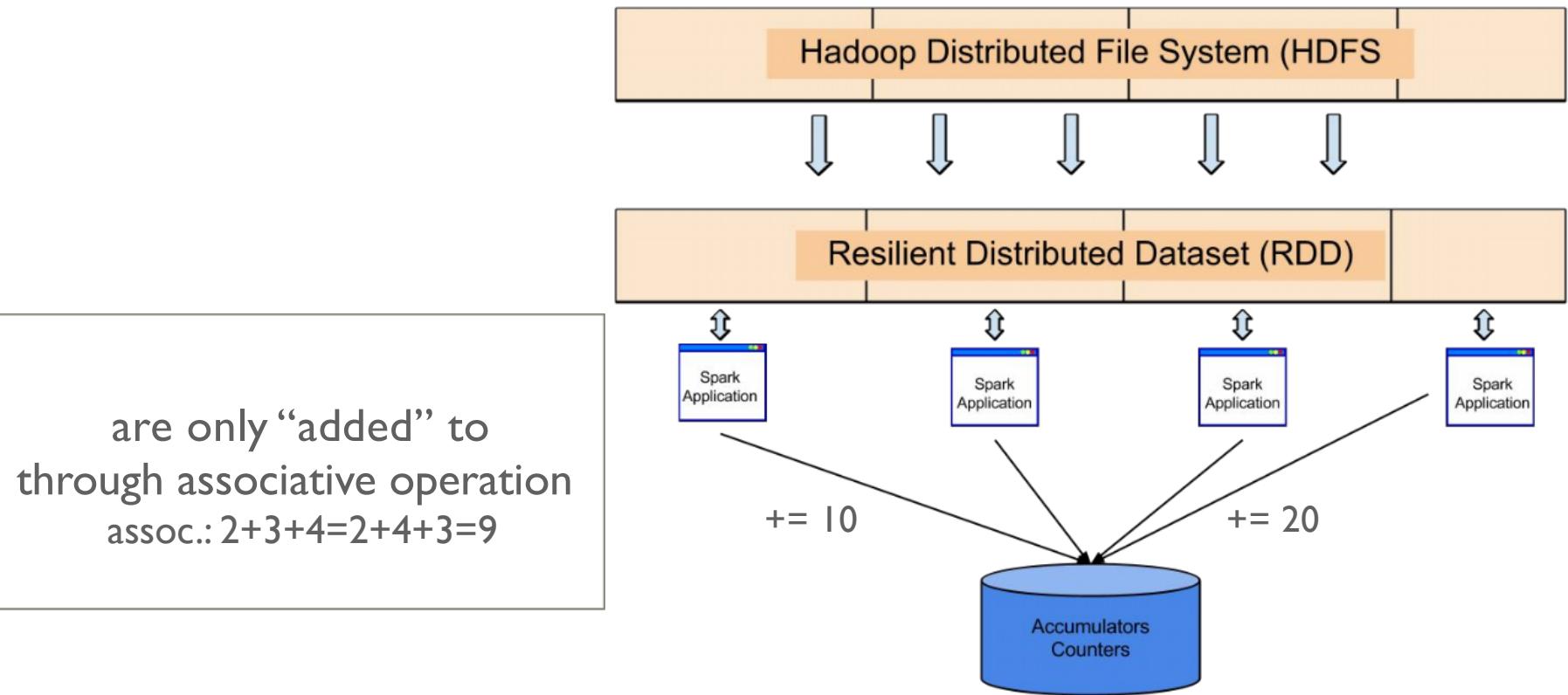
# Shared variables:

- When we pass functions such map()
- Every node gets a copy of the variable
- The change to these variables is not communicated back
- After starting of the map(), changes to the variable on driver doesn't impact the worker.

## Two Kinds:

1. **Accumulators** to aggregate information
2. **Broadcast variables** to efficiently distribute large values

# SHARED MEMORY - Accumulators



# Accumulators

- Accumulators are variables that are only “added” to through an associative operation
- Can therefore be efficiently supported in parallel.
- They can be used to implement counters (as in MapReduce) or sums.

# Accumulator : Empty line count

```
rdd = sc.textFile('/home/ec2-user/data/empty_lines.txt')
print ('line count:', rdd.count())
rdd.take(2)

print ('empty lines: ', rdd.filter(lambda x : len(x) == 0).count())
```

```
line count: 31
empty lines: 16
```

# Accumulator : Empty line count

## Accumulators

```
: def line_len(line:str):
    if len(line.strip()) == 0:
        global num_empty_lines
        num_empty_lines += 1
    return len(line)
```

## Accumulators, take 5

```
: num_empty_lines = sc.accumulator(0)
words = rdd.map(line_len)
print (words.take(5))
print (num_empty_lines.value)
```

```
[58, 0, 56, 0, 0]
3
```

# Accumulator : Empty line count

## Accumulators, take 20

```
num_empty_lines = sc.accumulator(0)
words = rdd.map(line_len)
print (words.take(20))
print (num_empty_lines.value)
```

```
[58, 0, 56, 0, 0, 15, 0, 106, 0, 77, 0, 58, 0, 167, 0, 0, 387, 0, 151, 0]
11
```

## Accumulators, collect

```
num_empty_lines = sc.accumulator(0)
words = rdd.map(line_len)
_coll = (words.collect())
print (num_empty_lines.value)
```

# Accumulators and Fault Tolerance

- Spark Re-executes failed or slow tasks.
- Preemptively launches “speculative” copy of slow worker task

**The net result is: The same function may run multiple times on the same data.**

**Does it mean accumulators will give wrong result?  
YES, for accumulators in Transformation.  
No, for accumulators in Action**

# Accumulators and Fault Tolerance

---

- For accumulators in actions, Each task's accumulator update applied once.
- For reliable absolute value counter, put it inside an action
- In transformations, this guarantee doesn't exist.
- In transformations, use accumulators for debug only.

# Custom Accumulators

- Out of the box, Spark supports accumulators of type Double, Long, and Float.
- Spark also includes an API to define custom accumulator types and custom aggregation operations
  - (e.g., finding the maximum of the accumulated values instead of adding them).

# Broadcast Variables : Introduction

```
commonWords = List("a", "an", "the", "of", "at",
"is", "am", "are", "this", "that", "", "at")
```

If we need to remove the common words from our wordcount, what do we need to do?

- > We can create a local variable and use it
- > Is it inefficient?

# Broadcast Variables : Introduction

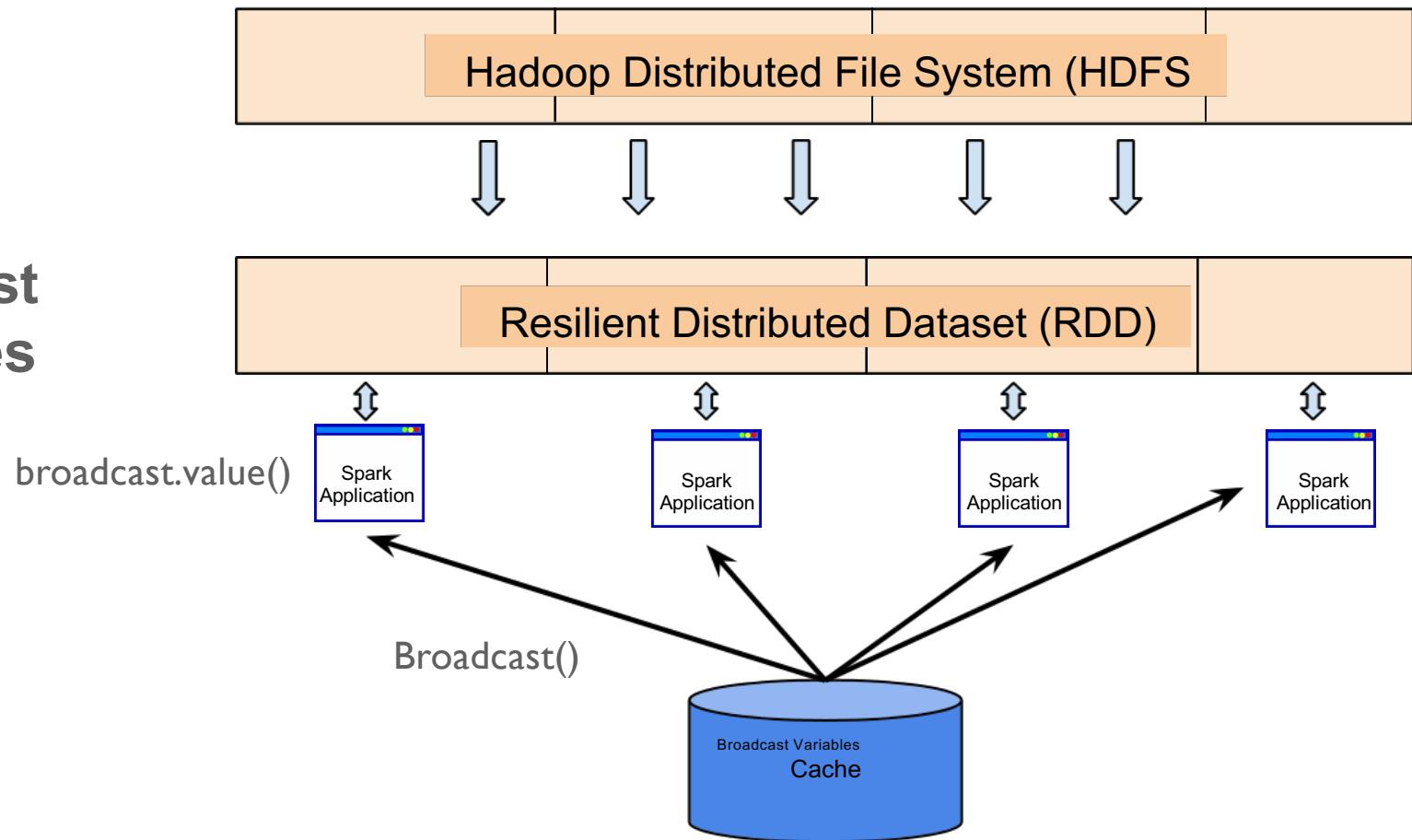
**Yes, because**

- 1. Spark sends referenced variables to all workers.
- 1. The default task launching mechanism is optimised for small task sizes.
- 2. If using multiple times, spark will be sending it again to all nodes

So, we use broadcast variable instead.

# SHARED MEMORY

## Broadcast Variables



# Broadcast Variables

- Efficiently send a large, read-only value to workers
- For example:
  - Send a large, read-only lookup table to all the nodes
  - Large feature vector in a machine learning algorithm
- It is like a **distributed cache** of Hadoop
- Spark distributes broadcast variables efficiently to reduce communication cost.
- Useful when
  - Tasks across multiple stages need the same data
  - Caching the data in deserialized form is important.

# Broadcast Variables : Example

## Broadcast

### Remove stop words

```
stopwords = ["a", "an", "the", "of", "at", "is", "am", "are", "this", "that", '', 'at', 'for', 'with']
print (stopwords)

['a', 'an', 'the', 'of', 'at', 'is', 'am', 'are', 'this', 'that', '', 'at', 'for', 'with']

words = sc.broadcast(stopwords)
print ('broadcast: ', words.value)

broadcast:  ['a', 'an', 'the', 'of', 'at', 'is', 'am', 'are', 'this', 'that', '', 'at', 'for', 'with']
```

# Broadcast Variables : Example

```
rdd = sc.textFile('/home/ec2-user/data/empty_lines.txt')
print ('line count:', rdd.count())
rdd.take(2)

def toWords(line:str):
    words = line.split(" ")
    words = [i for i in words if i not in swords.value]
    return words

not_stopwords = rdd.flatMap(toWords)
not_stopwords.take(10)
```

line count: 31

```
[ 'Apache',
  'Spark',
  'open-source',
  'distributed',
  'general-purpose',
  'cluster-computing',
  'framework.',
```

# Key Performance Considerations

1. Level of Parallelism
2. Serialization Format
3. Memory Management
4. Hardware Provisioning

# Level of Parallelism

## By Default

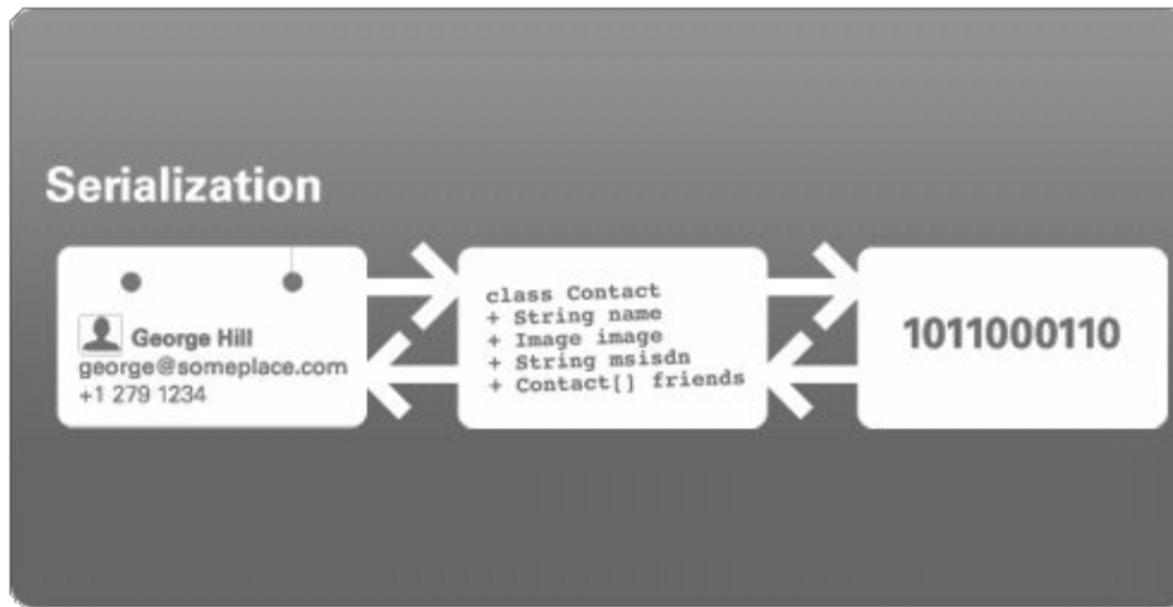
- A single task per one partition,
- A single core in the cluster to execute.
- Default partitions are based on underlying storage or CPU
- HDFS RDDs - One partition per block

# Key Performance Considerations

1. Level of Parallelism - How many default partitions?

# Serialization Format

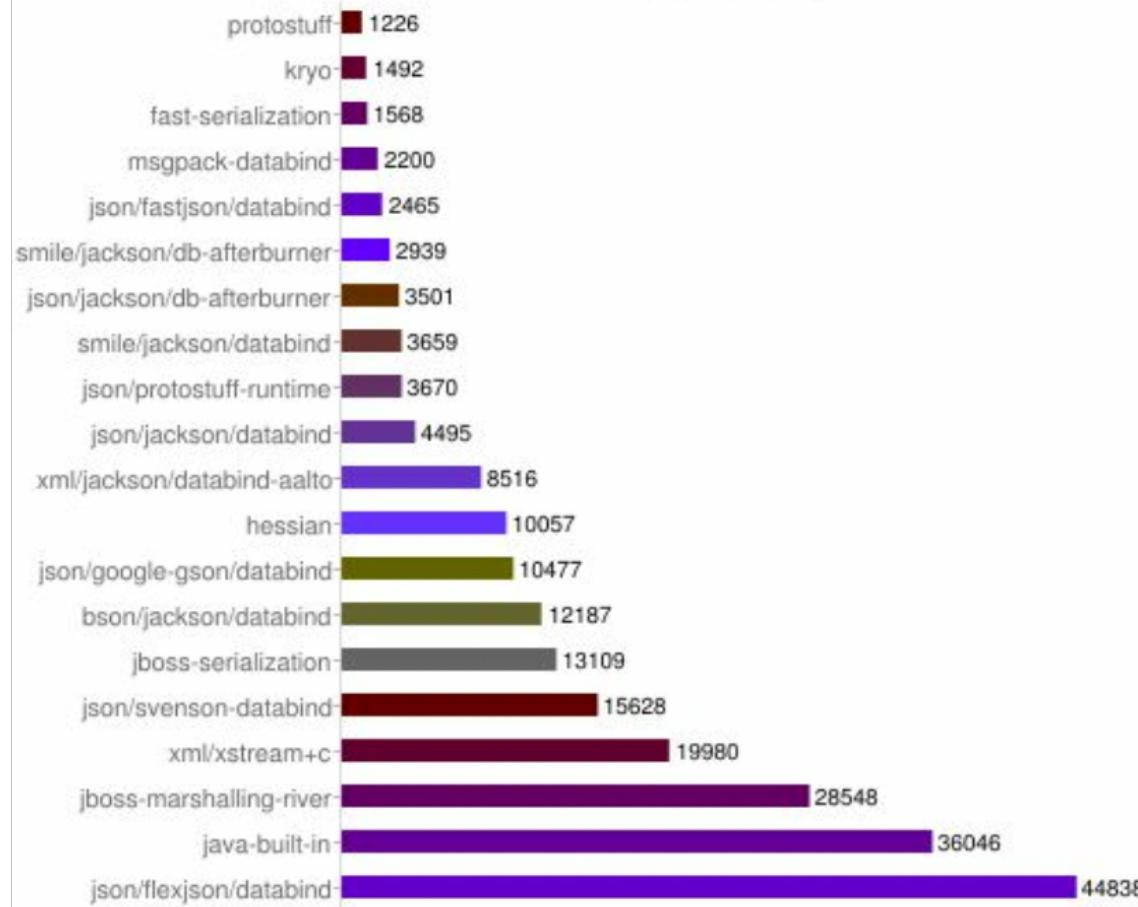
- While transferring or saving objects need serialization
- Comes into play during large transfers
- By default Spark will use Java's built-in serializer.



# Serialization Format

## Benchmarks

total (nanos)



# Serialization Format

---

## Kryo

- Spark also supports the use of Kryo
- Faster and more compact
- But cannot serialize all types of objects “out of the box.”
- Almost all applications will benefit from shifting to Kryo
- To use,
  - `sc.getConf.set("spark.serializer", "org.apache.spark.serializer.KryoSerializer")`
- For best performance, register classes with Kryo
  - `sc.getConf.registerKryoClasses(Array(classOf[MyClass1], classOf[MyClass2]))`
  - Class needs to implement Java’s Serializable interface

# Memory Management

## RDD storage

- persist()'ed memory
- spark.storage.memoryFraction - Default: 60%
- If exceeded, older will be dropped
  - will be computed on demand
- For huge data, use persist() with MEMORY\_AND\_DISK

# Memory Management

## RDD storage

- persist()'ed memory
- spark.storage.memoryFraction - Default: 60%
- If exceeded, older will be dropped
  - will be computed on demand
- For huge data, use persist() with MEMORY\_AND\_DISK

## Shuffle and aggregation buffers

- For storing shuffle output data
- spark.shuffle.memoryFraction - Default: 20%

# Memory Management

## RDD storage

- persist()'ed memory
- spark.storage.memoryFraction - Default: 60%
- If exceeded, older will be dropped
  - will be computed on demand
- For huge data, use persist() with MEMORY\_AND\_DISK

## Shuffle and aggregation buffers

- For storing shuffle output data
- spark.shuffle.memoryFraction - Default: 20%

## User Code

Remaining

Default: 20% of memory

# Hardware Provisioning

- Main Parameters
  - Executor's Memory (`spark.executor.memory`)
  - Number of cores per Executor,
  - Total number of executors
  - No. of disks

# **7. Spark Streaming**

# Spark Streaming - Introduction

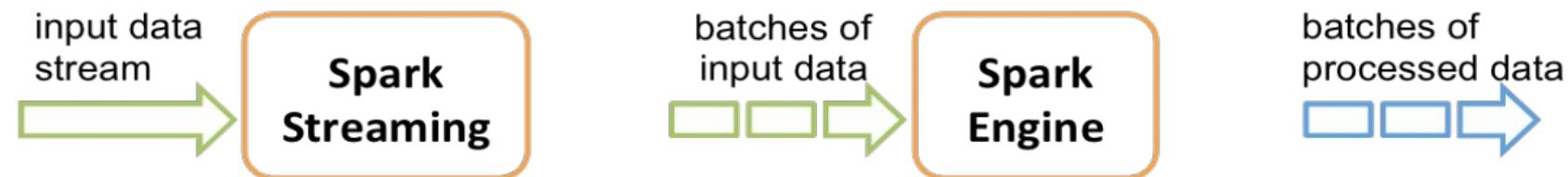
---

- Extension of the core Spark API
- Enable Stream Processing
  - Scalable
  - High-throughput
  - Fault-tolerant

# Spark Streaming - Introduction



# Spark Streaming - Workflow



Spark Streaming  
receives  
input data streams

Divides data into  
batches

These batches are  
processed by Spark  
engine

# Spark Streaming - Use Case



## Real-time Analytics

# Spark Streaming - Use Case

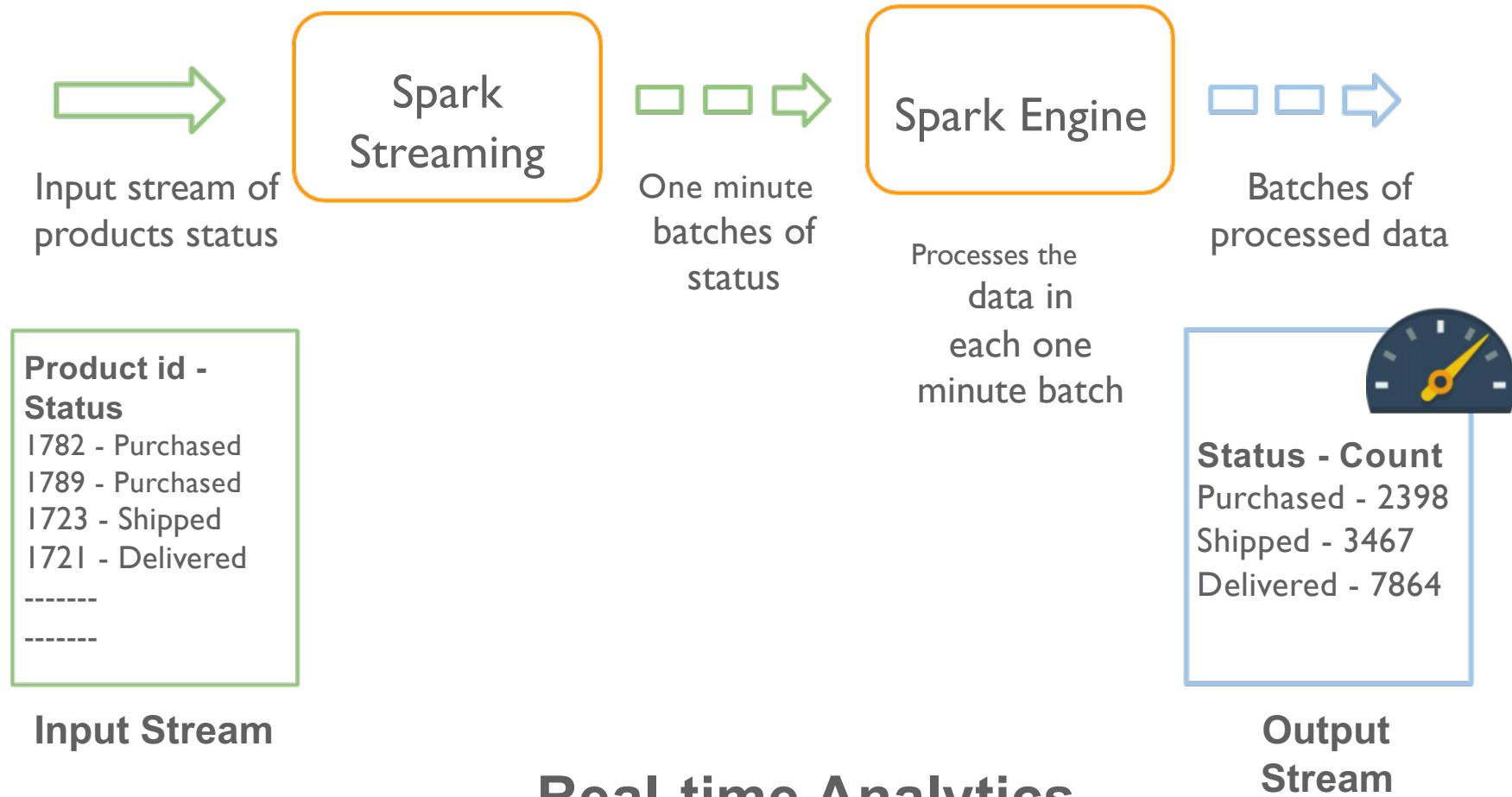
---

## Real-time Analytics

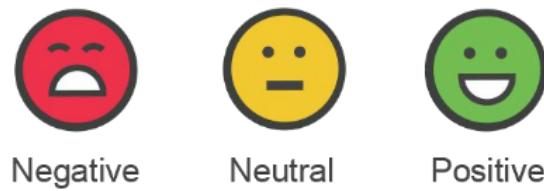
**Problem: Build real-time analytics dashboard with information on how many products are getting**

- Purchased
- Shipped
- Delivered every minute

# Spark Streaming - Use Case - Ecommerce



# Spark Streaming - Use Case



## Real-time Sentiment Analysis

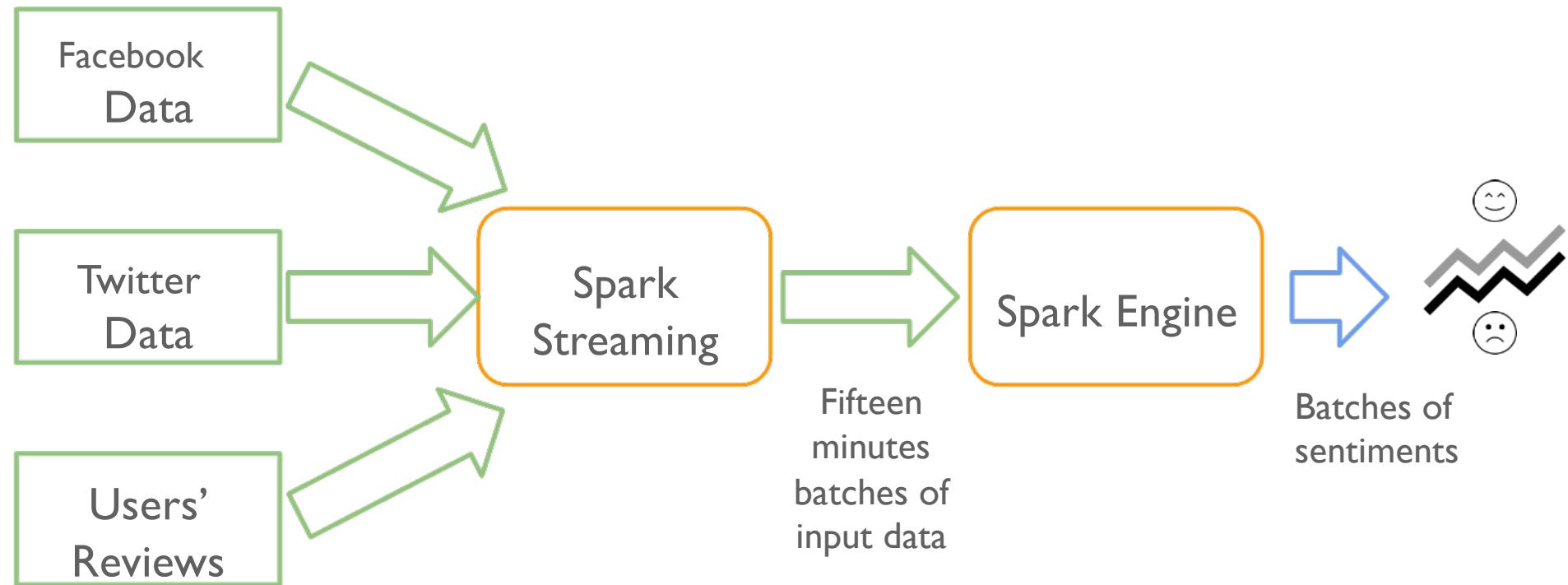
# Spark Streaming - Use Case

---

## Real-time Sentiment Analysis

**Problem:** Build Real-time sentiment analytics system to find out sentiment of users every fifteen minute by analyzing data from various sources such as Facebook, Twitter, users' feedback, comments and reviews

# Spark Streaming - Use Case



Real-time Sentiment  
Analysis

# Spark Streaming - Use Case

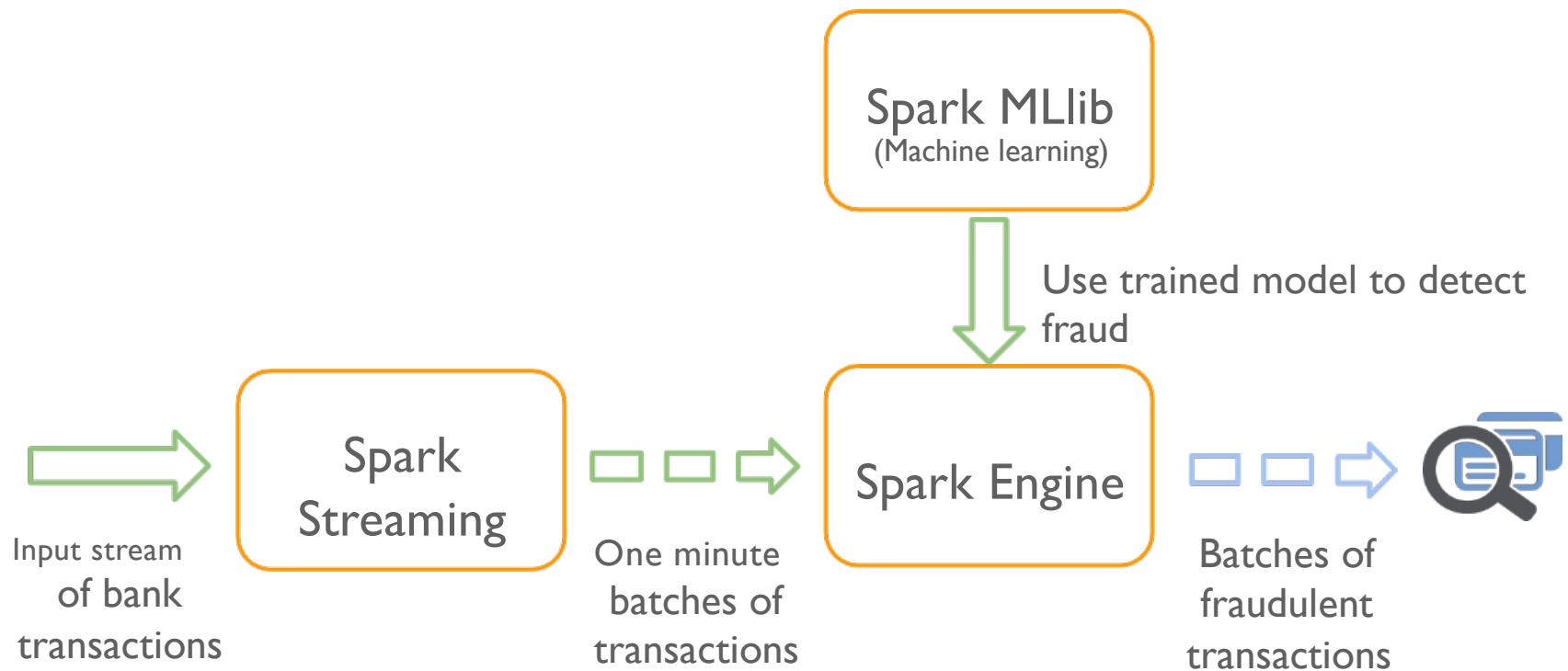
## Real-time Fraud Detection

**Problem: Build a real-time fraud detection system for a bank to find out the fraudulent transactions**

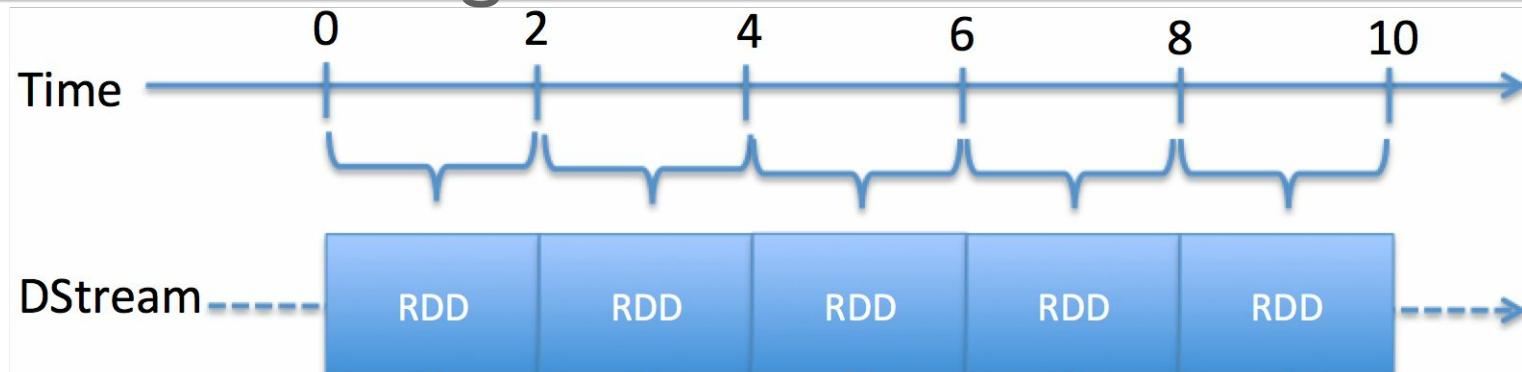


# Spark Streaming - Use Case

## Real-time Fraud Detection



# Spark Streaming - DStream



Discretized stream or DStream:

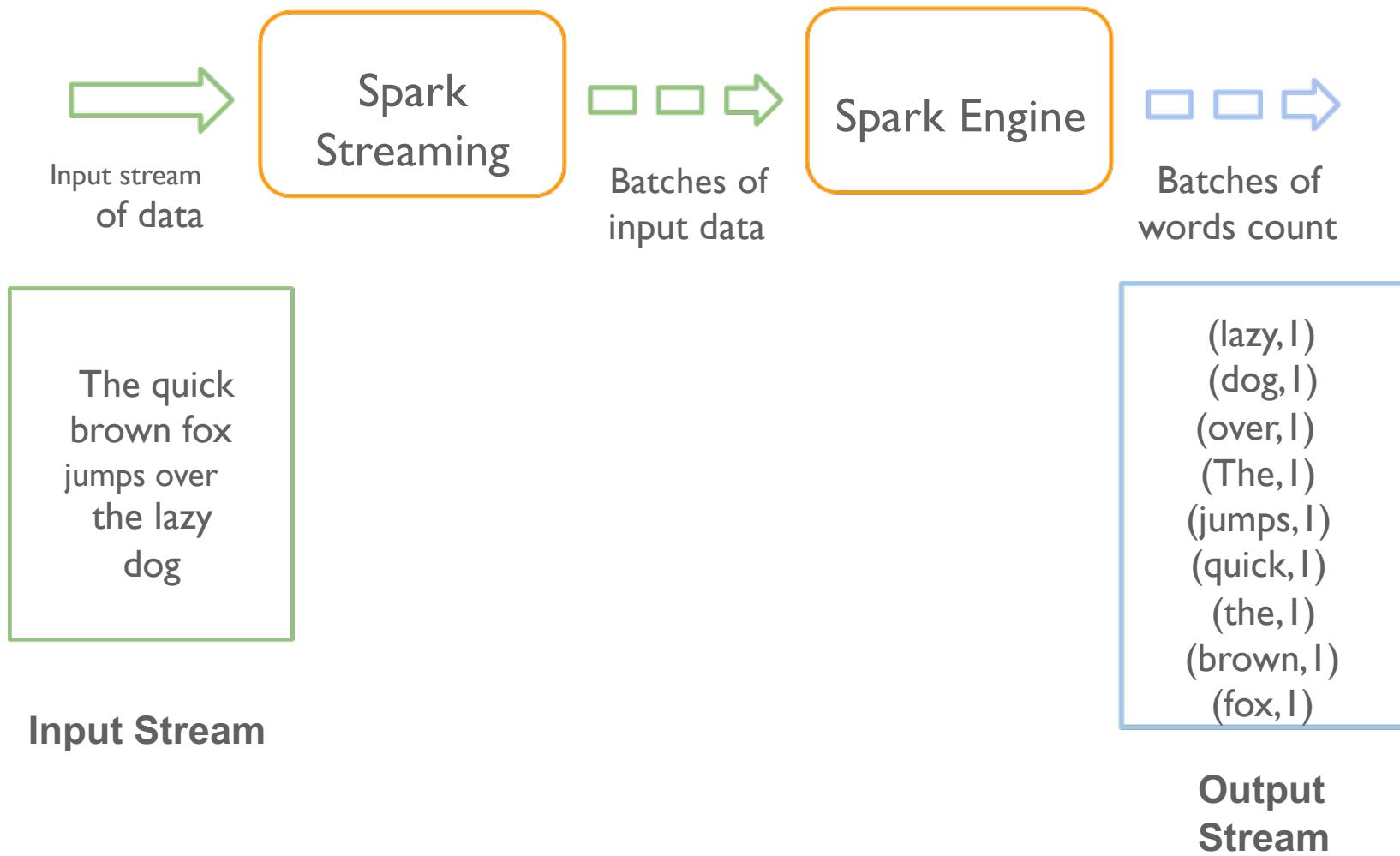
- Represents continuous stream of data
- Can be created
  - From input data streams
  - Or from other DStreams
- Is represented as a sequence of RDDs internally
- Is used to write Spark Streaming programs

# Spark Streaming - Hands-on - Word Count

---

**Problem: Count the number of words in the text**  
data received from a server listening on a host and a  
port

# Spark Streaming - Hands-on - Word Count



# Spark Streaming - Word Count - Producer

---

## Create the data producer

- Open a new web console
- Run the following command to start listening to 9999 port  
`nc -lk 9999`
- Whatever you type here would be passed to a process connecting at 9999 port

---

# Spark Streaming - Adding Dependencies

---

Source	Artifact
Kafka	spark-streaming-kafka_2.10
Flume	spark-streaming-flume_2.10
Kinesis	spark-streaming-kinesis-asl_2.10 [Amazon Software License]
Twitter	spark-streaming-twitter_2.10
ZeroMQ	spark-streaming-zeromq_2.10
MQTT	spark-streaming-mqtt_2.10

---

# Spark Streaming - A Quick Recap

---

1. First initialize the StreamingContext. It is initialized in **ssc** variable in our code
2. Define the input sources by creating input DStreams. It is defined in **lines** variable in our code
3. Define the streaming computations by applying transformations to DStreams. It is defined in **words**, **pairs** and **wordCounts** variables in our code

# Spark Streaming - A Quick Recap

---

4. Start receiving data and processing it using `streamingContext.start()`.
5. Wait for the processing to be stopped (manually or due to any error) using `streamingContext.awaitTermination()`.
6. The processing can be manually stopped using `streamingContext.stop()`.

# Spark Streaming - Running Locally

---

**For running locally,**

- Do not use “local” or “local[1]” as the master URL.
  - As it uses only one thread for receiving the data
  - Leaves no thread for processing the received data
- So, Always use “local[n]” as the master URL , where n > no. of receivers

# Apache Kafka - Introduction

---



Kafka is used for building real-time data pipelines and streaming applications.

# Apache Kafka - Introduction

---

- Distributed publish-subscribe messaging system, similar to a message queue or enterprise messaging system
- Originally developed at LinkedIn and later on became part of Apache project.
- It is fast, scalable, durable and distributed by design

# Apache Kafka - Key Concepts

---

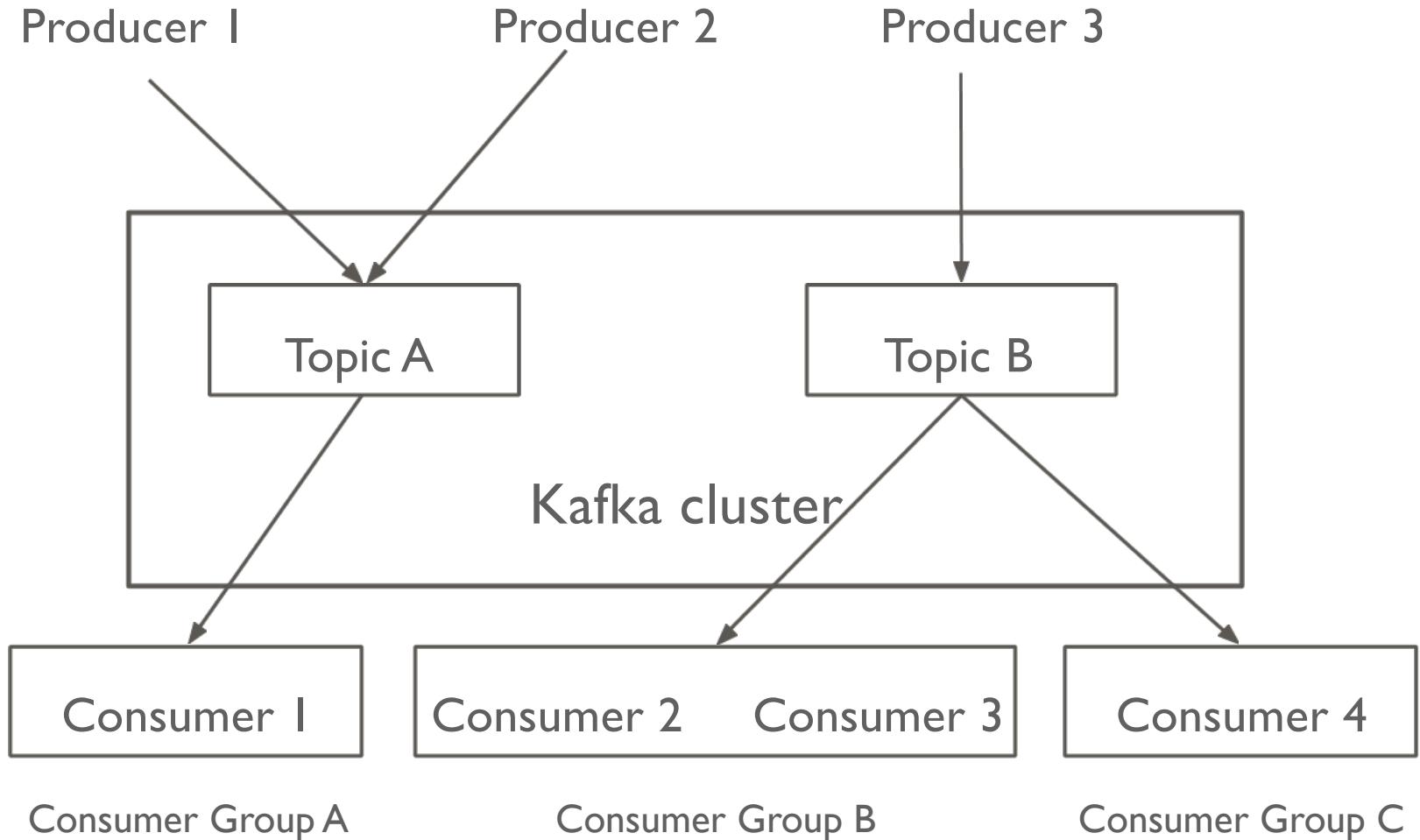
- Runs on a cluster of one or more servers. Each node in the cluster is called **broker**
- Stores records in the categories called **topics**
- Kafka topics are divided into a number of partitions
- Partitions split the data in a particular topic across multiple brokers

# Apache Kafka - Key Concepts

---

- Each topic partition in kafka is replicated “n” times where “n” is the replication factor of topic
  - Automatic failover to replicas when a server in the cluster fails
- There can be multiple topics in the Kafka cluster
  - One topic for website activity tracking
  - Another topic for storing application performance metrics

# Apache Kafka - Producers and Consumers



# Spark Streaming + Kafka Integration

**Problem - Count the words from the messages stored in Kafka every 10 seconds**

## Steps

- Publish stream of “y” using yes command to Kafka topic
- Spark streaming code consumes the stream of “y” from the Kafka topic in the batch interval of 2 seconds
- Print number of “y” consumed or processed

[https://github.com/cloudxlab/bigdata/tree/master/spark/examples/streaming/word\\_count\\_kafka](https://github.com/cloudxlab/bigdata/tree/master/spark/examples/streaming/word_count_kafka)

# Spark Streaming + Kafka Integration

## Transformation

## Meaning

**map(*func*)**

Return a new DStream by passing each element of the source DStream through a function *func*.

**flatMap(*func*)**

Similar to map, but each input item can be mapped to 0 or more output items.

**filter(*func*)**

Return a new DStream by selecting only the records of the source DStream on which *func* returns true.

**repartition(*numPartitions*)**

Changes the level of parallelism in this DStream by creating more or fewer partitions.

**union(*otherStream*)**

Return a new DStream that contains the union of the elements in the source DStream and *otherDStream*.

**count()**

Return a new DStream of single-element RDDs by counting the number of elements in each RDD of the source DStream.

# Spark Streaming + Kafka Integration

## Transformation

## Meaning

**reduce(func)**

Return a new DStream of single-element RDDs by aggregating the elements in each RDD of the source DStream using a function *func* (which takes two arguments and returns one). The function should be associative so that it can be computed in parallel.

**countByValue()**

When called on a DStream of elements of type K, return a new DStream of (K, Long) pairs where the value of each key is its frequency in each RDD of the source DStream.

**reduceByKey(func,**

**[numTasks])**

When called on a DStream of (K, V) pairs, return a new DStream of (K, V) pairs where the values for each key are aggregated using the given reduce function. **Note:** By default, this uses Spark's default number of parallel tasks (2 for local mode, and in cluster mode the number is determined by the config property `spark.default.parallelism`) to do the grouping. You can pass an optional `numTasks` argument to set a different number of tasks.

**join(otherStream,**

**[numTasks])**

When called on two DStreams of (K, V) and (K, W) pairs, return a new DStream of (K, (V, W)) pairs with all pairs of elements for each key.

# Spark Streaming + Kafka Integration

Transformation	Meaning
<b>cogroup(<i>otherStream</i>, [<i>numTasks</i>])</b>	When called on a DStream of (K, V) and (K, W) pairs, return a new DStream of (K, Seq[V], Seq[W]) tuples.
<b>transform(<i>func</i>)</b>	Return a new DStream by applying a RDD-to-RDD function to every RDD of the source DStream. This can be used to do arbitrary RDD operations on the DStream.
<b>updateStateByKey(<i>func</i>)</b>	Return a new "state" DStream where the state for each key is updated by applying the given function on the previous state of the key and the new values for the key. This can be used to maintain arbitrary state data for each key.

# Spark Streaming - updateStateByKey

---

- In the last hands-on we just printed the word count in interval of 10 seconds
- What if we also want to count the each word seen in the input data stream in last 24 hours
- How do we maintain the running count of each word in last 24 hours?

# Spark Streaming - updateStateByKey

---

- To keep track of statistics, a state must be maintained while processing RDDs in the DStream
- If we maintain state for key-value pairs, the data may become too big to fit in memory on one machine
- We can use updateStateByKey function of Spark Streaming library

# Spark Streaming - updateStateByKey

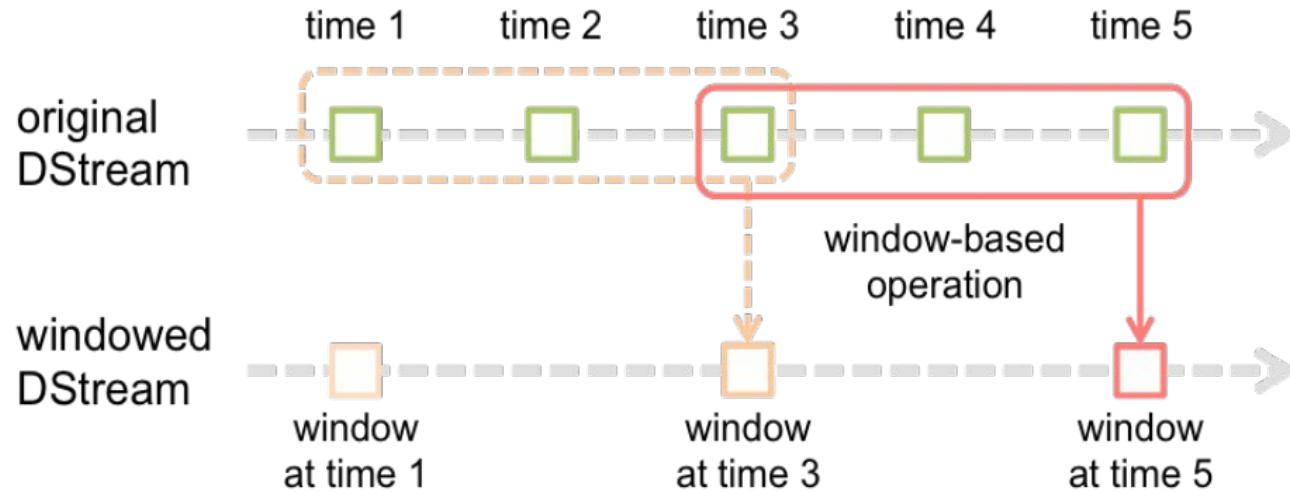
- The `updateStateByKey` operation allows us to maintain arbitrary state while continuously updating it with new information
- To use this, we will have to do two steps
  - Define the state - The state can be an arbitrary data type
  - Define the state update function - Specify with a function how to update the state using the previous state and the new values from an input stream

# Window Operations

---

- Apply transformations over a sliding window of data
- Use case in monitoring web server logs
  - Find out what happened in the last one hour and refresh that statistics every one minute
- **Window length - 1 hour**
- **Slide interval - 1 minute**

# Window Operations



# Window Operations - Use case

---

*# Reduce last 30 seconds of data, every 10 seconds*

```
windowedWordCounts =
```

```
    pairs.reduceByKeyAndWindow(lambda x, y: x + y, lambda x, y: x - y, 30, 10)
```

# Output Operations on DStreams

- Output operations allow DStream's data to be pushed out to external systems like
  - a database
  - or a file system
- Since the output operations actually allow the transformed data to be consumed by external systems, they trigger the actual execution of all the DStream transformations (similar to actions for RDDs)

# Output Operations on DStreams

## `print()`

- Prints the first ten elements of every batch of data in a DStream on the driver node running the streaming application.
- This is useful during development and debugging

## `saveAsTextFiles(prefix, [suffix])`

- Saves DStream's contents as text files.
- The file name at each batch interval is generated based on prefix and suffix: "prefix-TIME\_IN\_MS[.suffix]"

# Output Operations on DStreams

## **saveAsObjectFiles(prefix, [suffix])**

- Save this DStream's contents as SequenceFiles of serialized Java objects.
- The file name at each batch interval is generated based on prefix and suffix: "prefix-TIME\_IN\_MS[.suffix]".

## **saveAsHadoopFiles(prefix, [suffix])**

- Save this DStream's contents as Hadoop files.
- The file name at each batch interval is generated based on prefix and suffix: "prefix-TIME\_IN\_MS[.suffix]".

# Output Operations on DStreams

## **foreachRDD(func)**

- The most generic output operator that applies a function, func, to each RDD in the stream.
- This function should push the data in each RDD to an external system, such as
  - Saving the RDD to files
  - Or writing it over the network to a database
- Note that the function func is executed in the driver node running the streaming application

# Output Operations on DStreams

## **foreachRDD(func)** - Design 1

```
dstream.foreachRDD { rdd =>
    val connection = createNewConnection() // executed at the driver
    rdd.foreach { record =>
        connection.send(record) // executed at the worker
    }
}
```

# Output Operations on DStreams

## **foreachRDD(func) - Design 2**

```
dstream.foreachRDD { rdd =>
    rdd.foreach { record =>
        val connection = createNewConnection()
        connection.send(record)
        connection.close()
    }
}
```

# Output Operations on DStreams

## **foreachRDD(func) - Design 3**

```
dstream.foreachRDD { rdd =>
    rdd.foreachPartition { partitionOfRecords =>
        val connection = createNewConnection()
        partitionOfRecords.foreach(record => connection.send(record))
        connection.close()
    }
}
```

# **8. Spark Data-Frames and ML Pipelines**

# Spark for Data Science

## DataFrames

- Structured data
- Familiar API based on R & Python Pandas
- Distributed, optimized implementation

## Machine Learning Pipelines

*Simple construction and tuning of ML workflows*

# About Spark MLlib

## Classification

- Logistic regression
- Naive Bayes
- Streaming logistic regression
- Linear SVMs
- Decision trees
- Random forests
- Gradient-boosted trees
- 

## Regression

- Ordinary least squares
- Ridge regression
- Lasso
- Isotonic regression
- Decision trees
- Random forests
- Gradient-boosted trees
- Streaming linear methods
- 

## Clustering

- Gaussian mixture models
- K-Means
- Streaming K-Means
- Latent Dirichlet Allocation
- Power Iteration Clustering

## Recommendation

- Alternating Least Squares

## Feature extraction & selection

- Word2Vec
- Chi-Squared selection
- Hashing term frequency
- Inverse document frequency
- Normalizer
- 
- Standard scaler
- Tokenizer

## Statistics

- Pearson correlation
- Spearman correlation
- Online summarization
- Chi-squared test
- Kernel density estimation

## Linear algebra

- Local dense & sparse vectors & matrices
  - Distributed matrices
    - Block-partitioned matrix
    - Row matrix
    - Indexed row matrix
    - Coordinate matrix
  - Matrix decompositions
- Frequent itemsets
- FP-growth
- Model import/export

# ML Workflows are complex

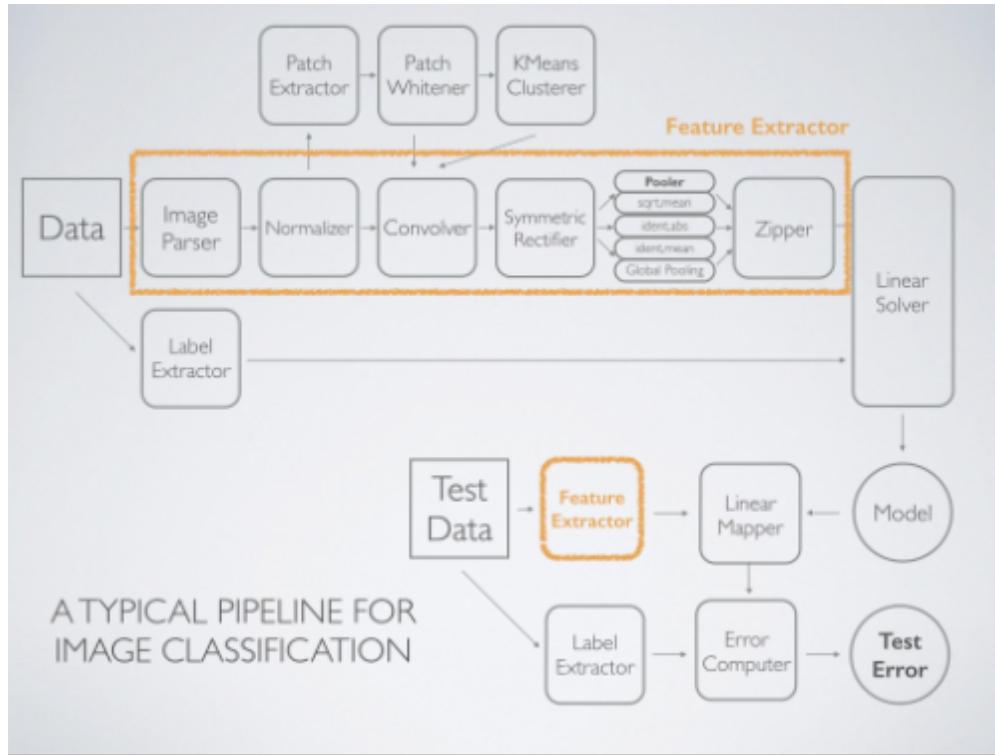


Image classification pipeline\*

- \* Specify pipeline
- \* Inspect & debug
- \* Re-run on new data
- \* Tune parameters

# Example: Text Classification

Goal: Given a text document, predict its topic.

## Features

Subject: Re: Lexan Polish?  
Suggest McQuires #1 plastic  
polish. It will help somewhat  
but nothing will remove deep  
scratches without making it  
worse than it already is.  
McQuires will do something...

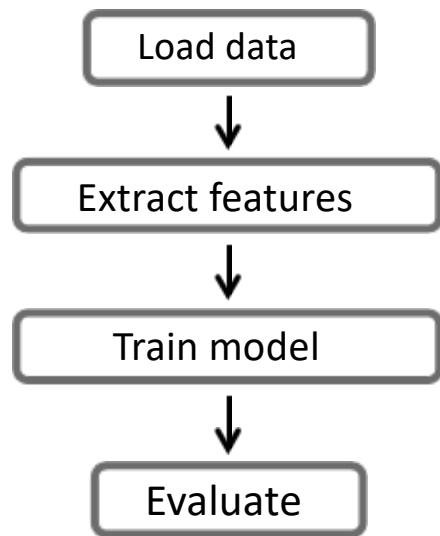


## Label

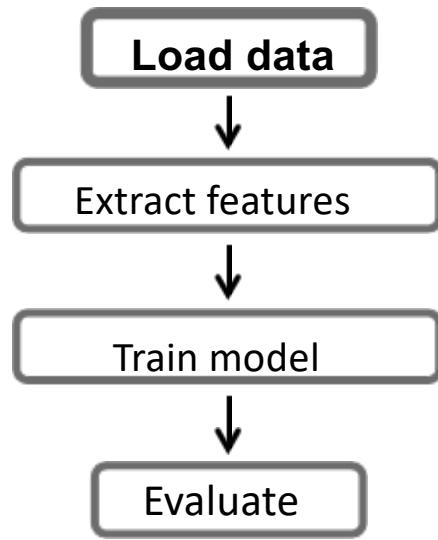
1: about science  
0: not about science

Dataset: “20 Newsgroups”  
*From UCI KDD Archive*

# ML Workflow



# Load Data



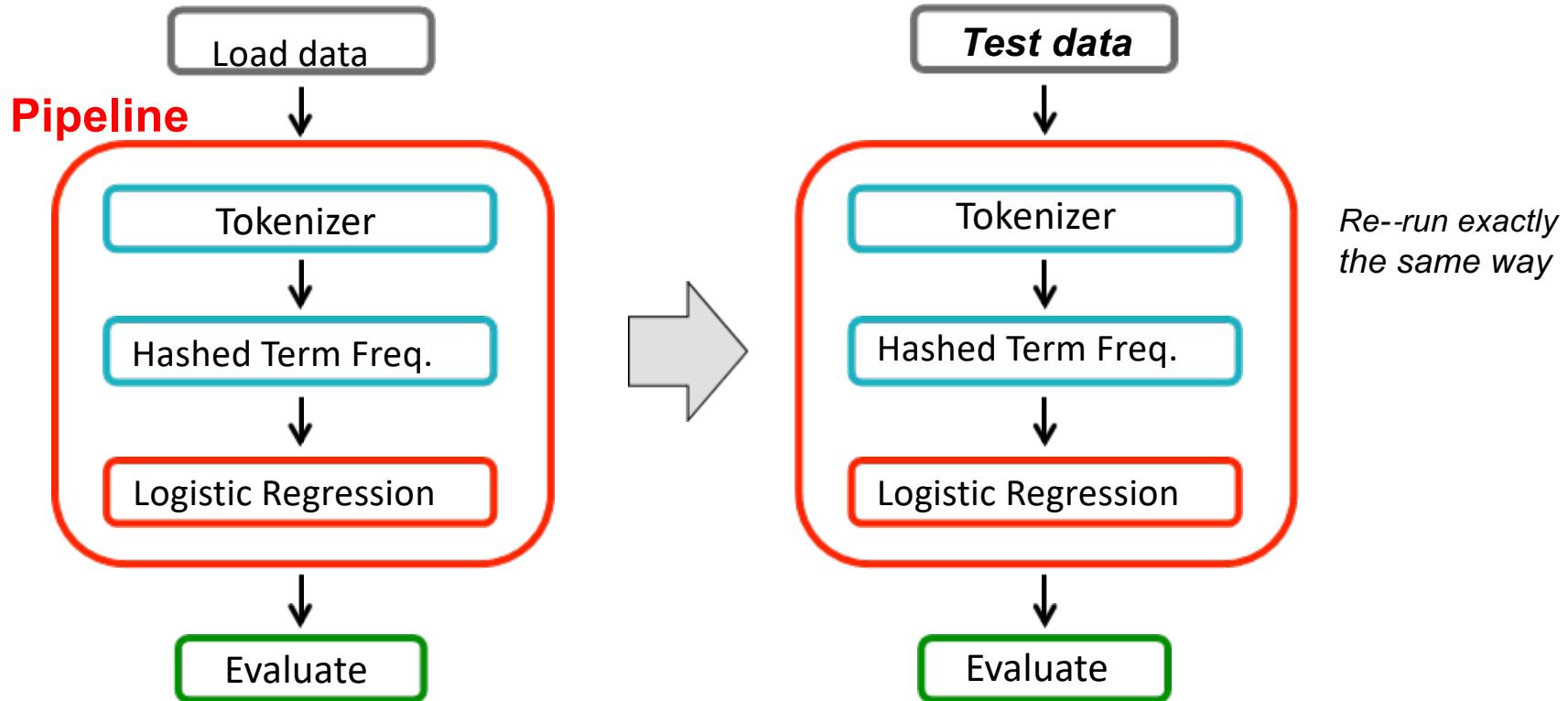
## Data sources for DataFrames

built-in

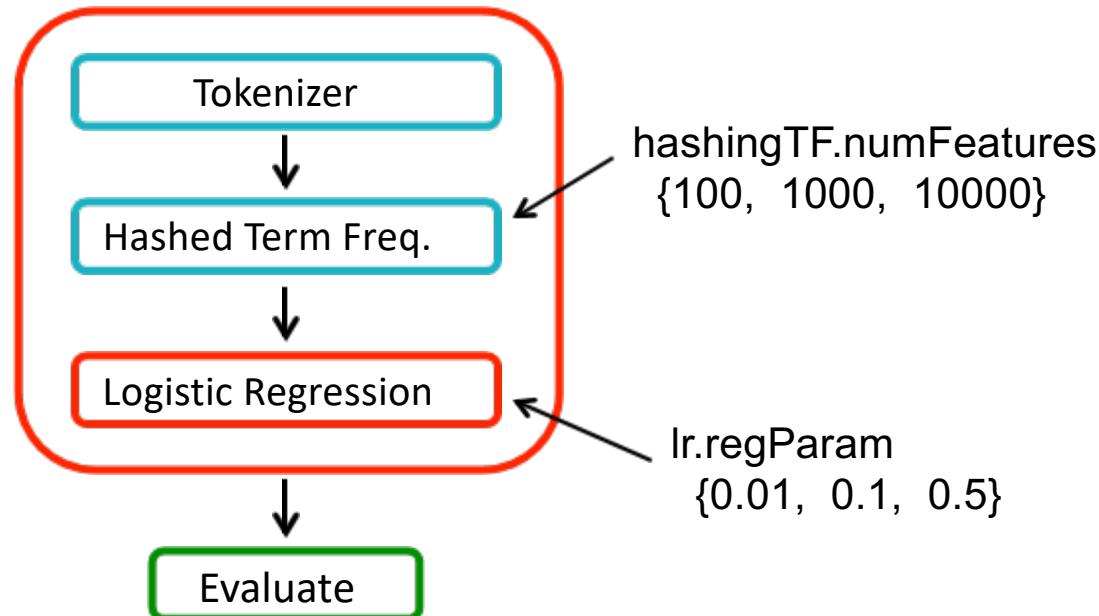


external

# ML Pipelines



# Parameter Tuning



## CrossValidator

Given:

- Estimator
- Parameter grid
- Evaluator

Find best parameters

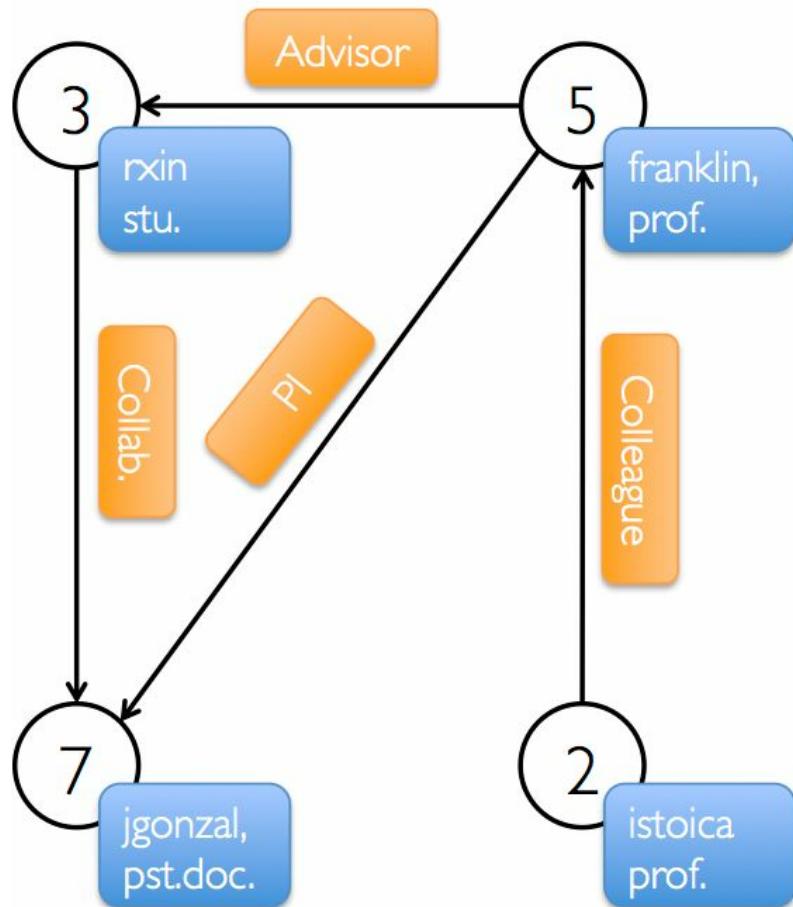
## Demo: ML Pipelines

[https://github.com/shekhar2010us/spark-examples/blob/master/notebooks/tutorial/5\\_spark\\_ml.ipynb](https://github.com/shekhar2010us/spark-examples/blob/master/notebooks/tutorial/5_spark_ml.ipynb)

# **9. Spark GraphX**

# What is a graph

## Property Graph



## Vertex Table

Id	Property (V)
3	(rxin, student)
7	(jgonzal, postdoc)
5	(franklin, professor)
2	(istoica, professor)

## Edge Table

SrcId	DstId	Property (E)
3	7	Collaborator
5	3	Advisor
2	5	Colleague
5	7	PI

---

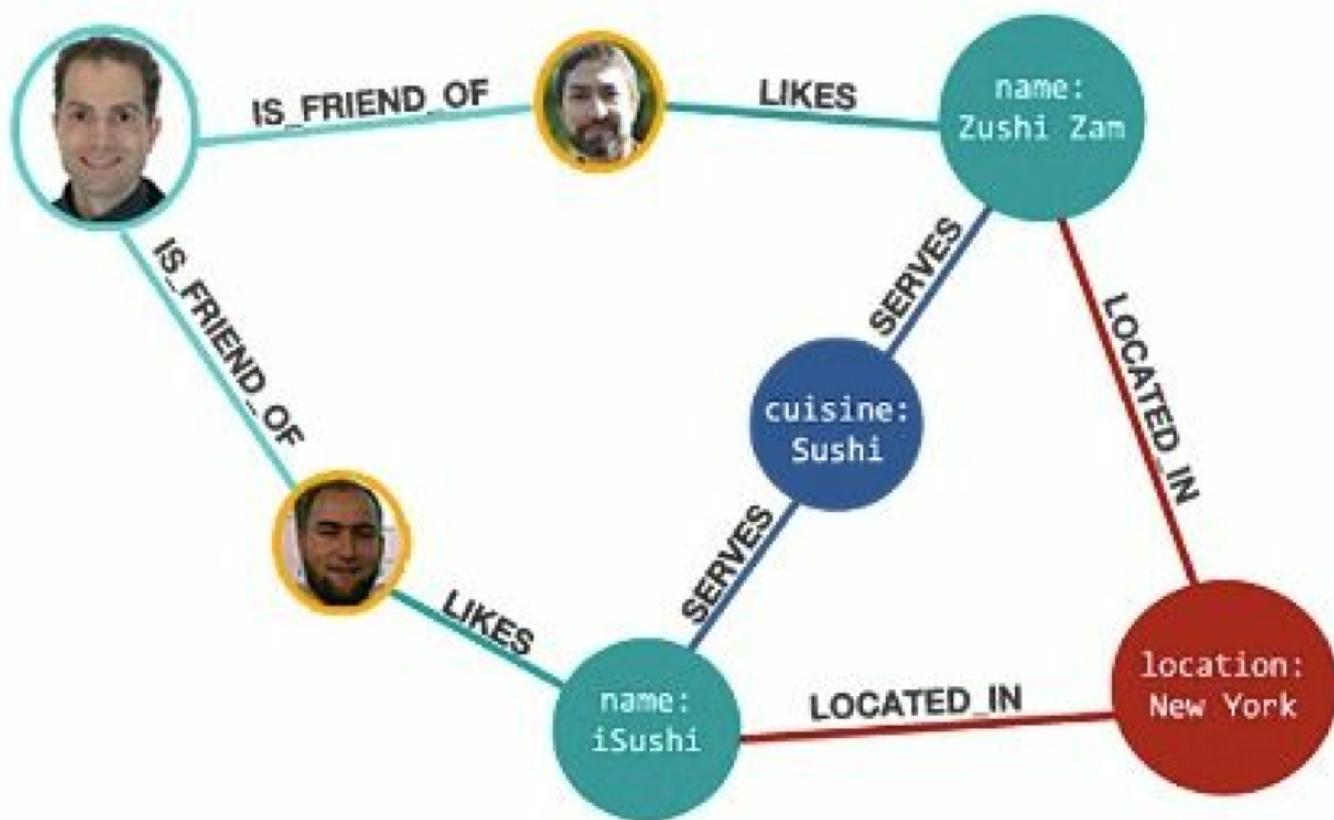
# Examples of graph computations

---

- Finding common friends
- Finding the page rank
- And Many more...

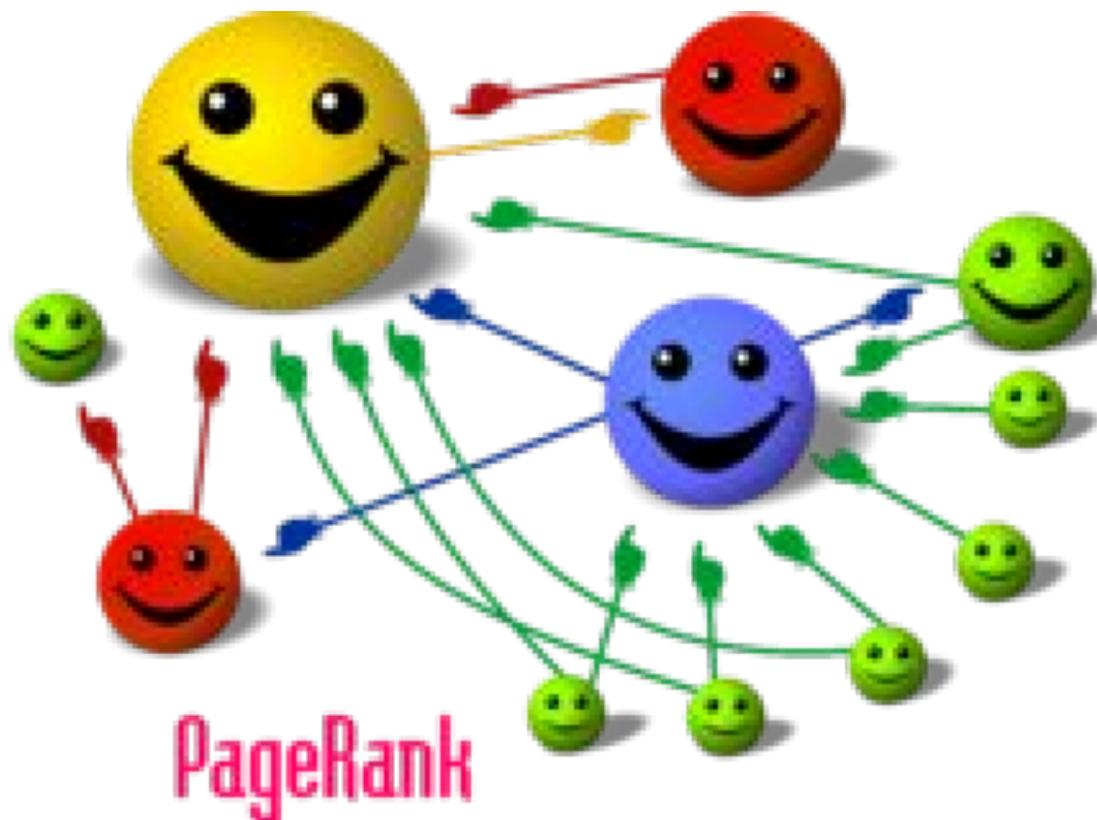
# Examples of graph computations

## Finding common friends



# Examples of graph computations

## Finding Page Rank



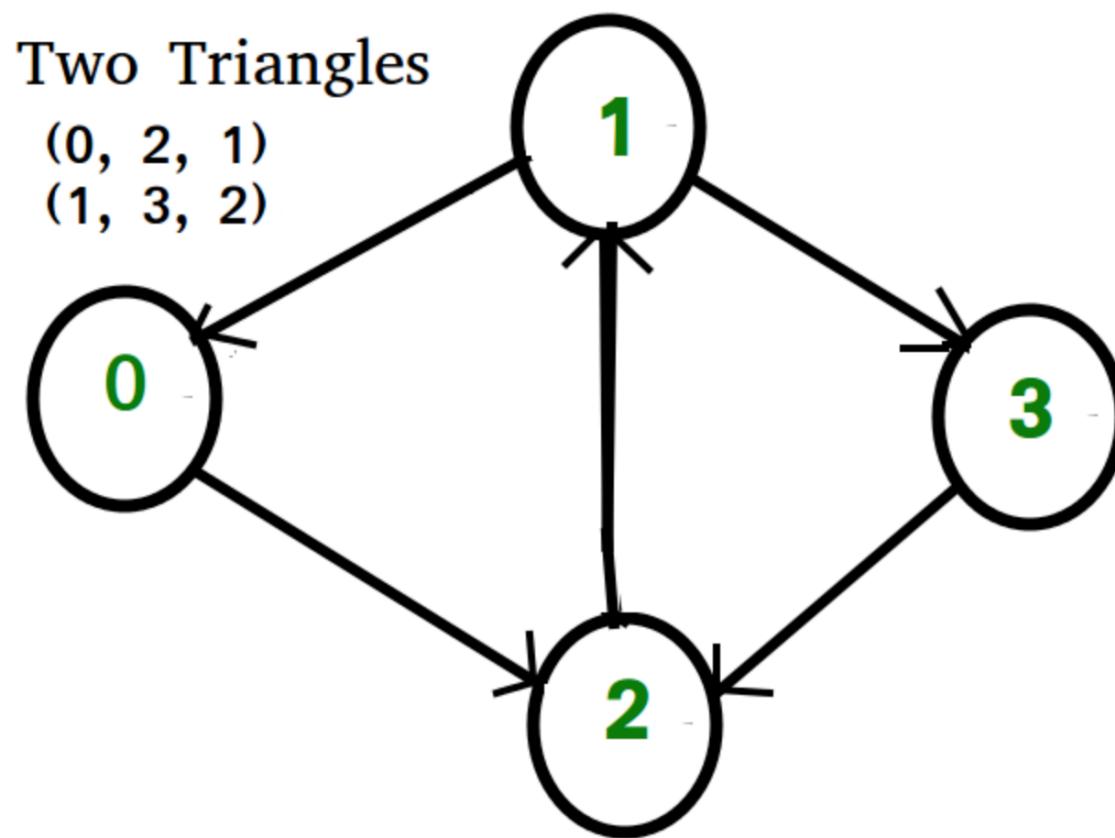
# GraphX

---

Has library of algorithms

- **PageRank**
  - If important pages link you, you are more important
- **Connected Components**
  - Clusters amongst your facebook friends
- **Triangle Counting**
  - Triangles passing through each vertex => measure of clustering.
- **Strongly connected components**

# Example Graph

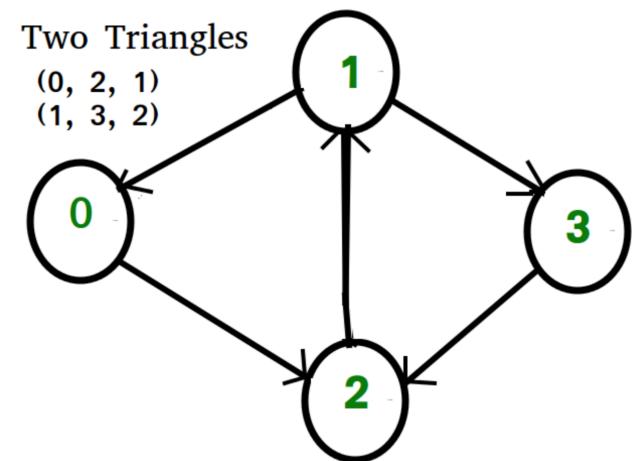


# GraphX – Triangle Count

**TriangleCount** counts the number of triangles passing through each vertex in a graph.

A triangle can be defined as a group of three vertices that is interrelated, i.e.  $a$  has an edge to  $b$ ,  $b$  has an edge to  $c$ , and  $c$  has an edge to  $a$ .

The example here shows a graph with two triangles.



# GraphX - Pagerank

---

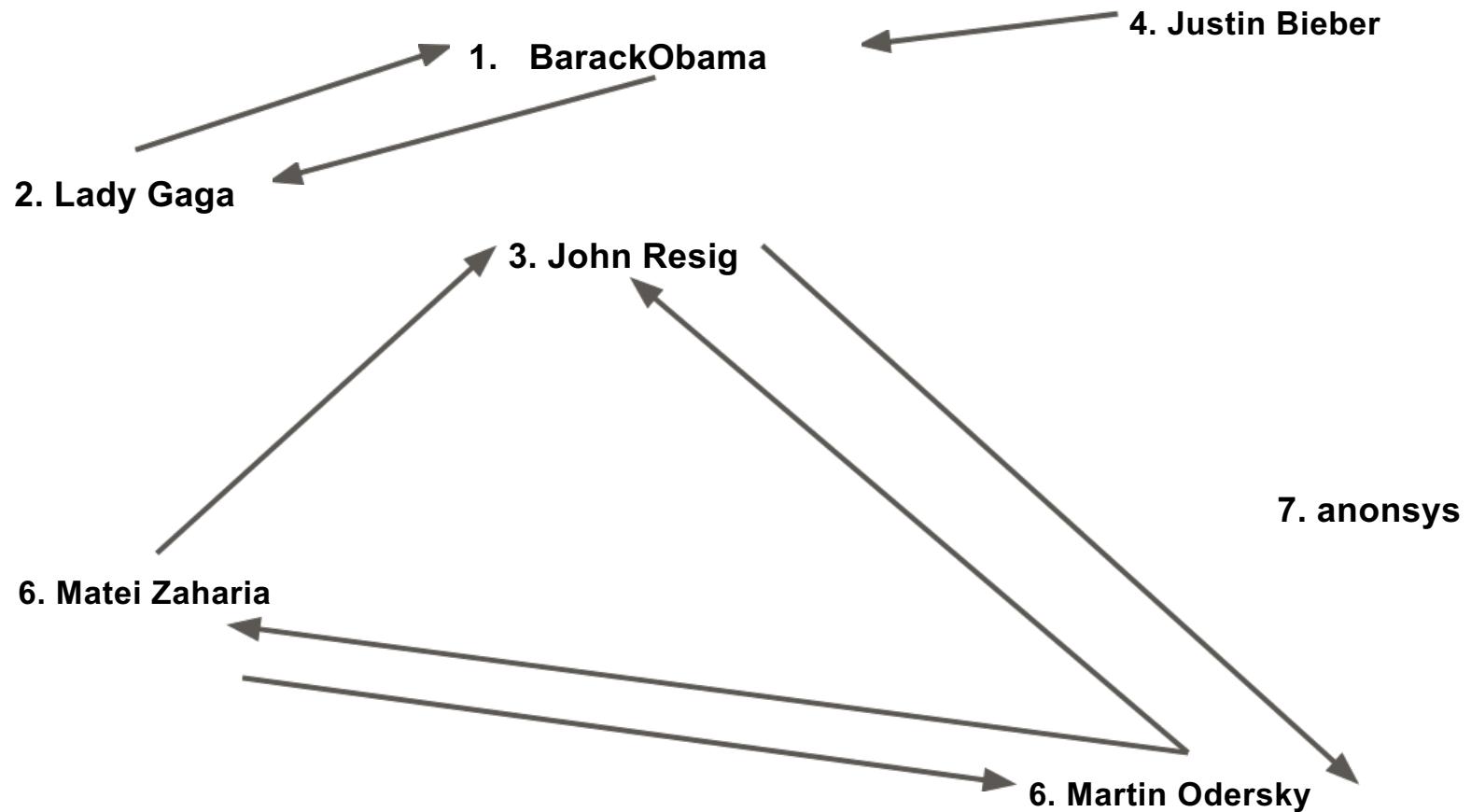
**PageRank** measures the importance of each vertex in a graph, assuming an edge from u to v represents an endorsement of v's importance by u.

For example, if a Twitter user is followed by many others, the user will be ranked highly.

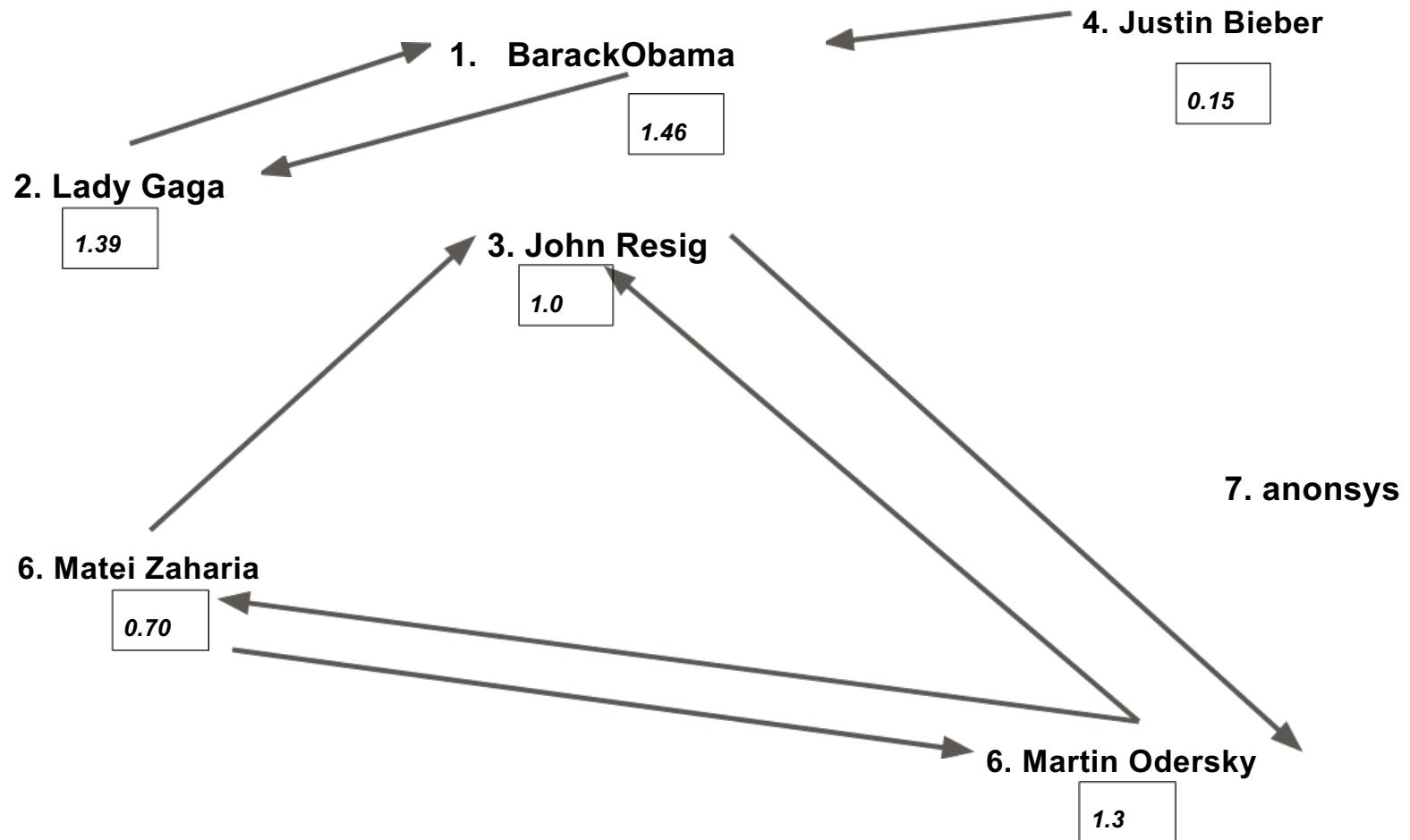
**PageRank** works by counting the number and quality of links to a page to determine a rough estimate of how important the website is.

The underlying assumption is that more important websites are likely to receive more links from other websites.

# GraphX - Pagerank



# GraphX - Pagerank



[https://github.com/shekhar2010us/spark-examples/blob/master/notebooks/tutorial/6\\_spark\\_graphx.ipynb](https://github.com/shekhar2010us/spark-examples/blob/master/notebooks/tutorial/6_spark_graphx.ipynb)

**THANK YOU !!**