

# HDFS

## The Hadoop Distributed File System

Bernard Lee Kok Bang

# Installing mrjob on HDP 2.6.5

- ***Install PIP***
  - yum-config-manager --save --setopt=HDP-SOLR-2.6-100.skip\_if\_unavailable=true
  - yum install https://repo.ius.io/ius-release-el7.rpm https://dl.fedoraproject.org/pub/epel/epel-release-latest-7.noarch.rpm
  - yum install python-pip
- ***Install MRJob***
  - pip install pathlib
  - pip install mrjob==0.7.4
  - pip install PyYAML==5.4.1
- ***Download u.data***
  - wget <http://media.sundog-soft.com/hadoop/ml-100k/u.data>

# Handles big files

- optimized for handling very large files
- distributed and broken up across an entire cluster
- e.g., big logs of information from sensors, or web servers, etc.

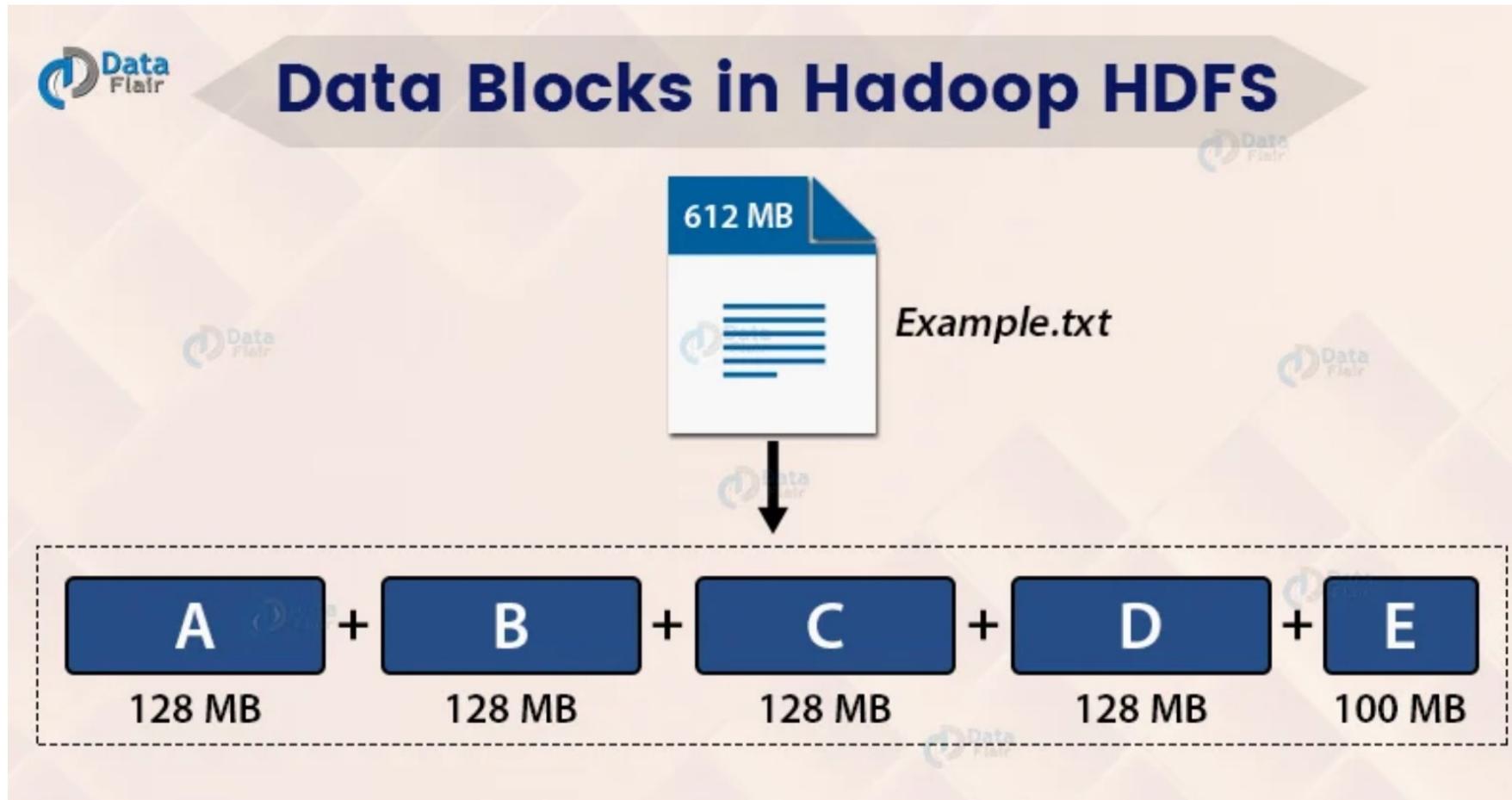
**e.g. health sensor, Netflix**

# By breaking into blocks

**partitioned**

- input data files are divided into **blocks of a particular size (128 mb by default)**
- these blocks of data are stored on different **DataNodes**
- Each block's information (its **address** on which data node is stored) is placed in **NameNode**
- if the block size is **too small**, then there will be **many blocks** to be stored on DataNodes
  - also a **large amount of metadata** information needs to be stored on NameNode
- if the block size **too large, parallelism** can't be achieved

# Data Blocks in Hadoop HDFS

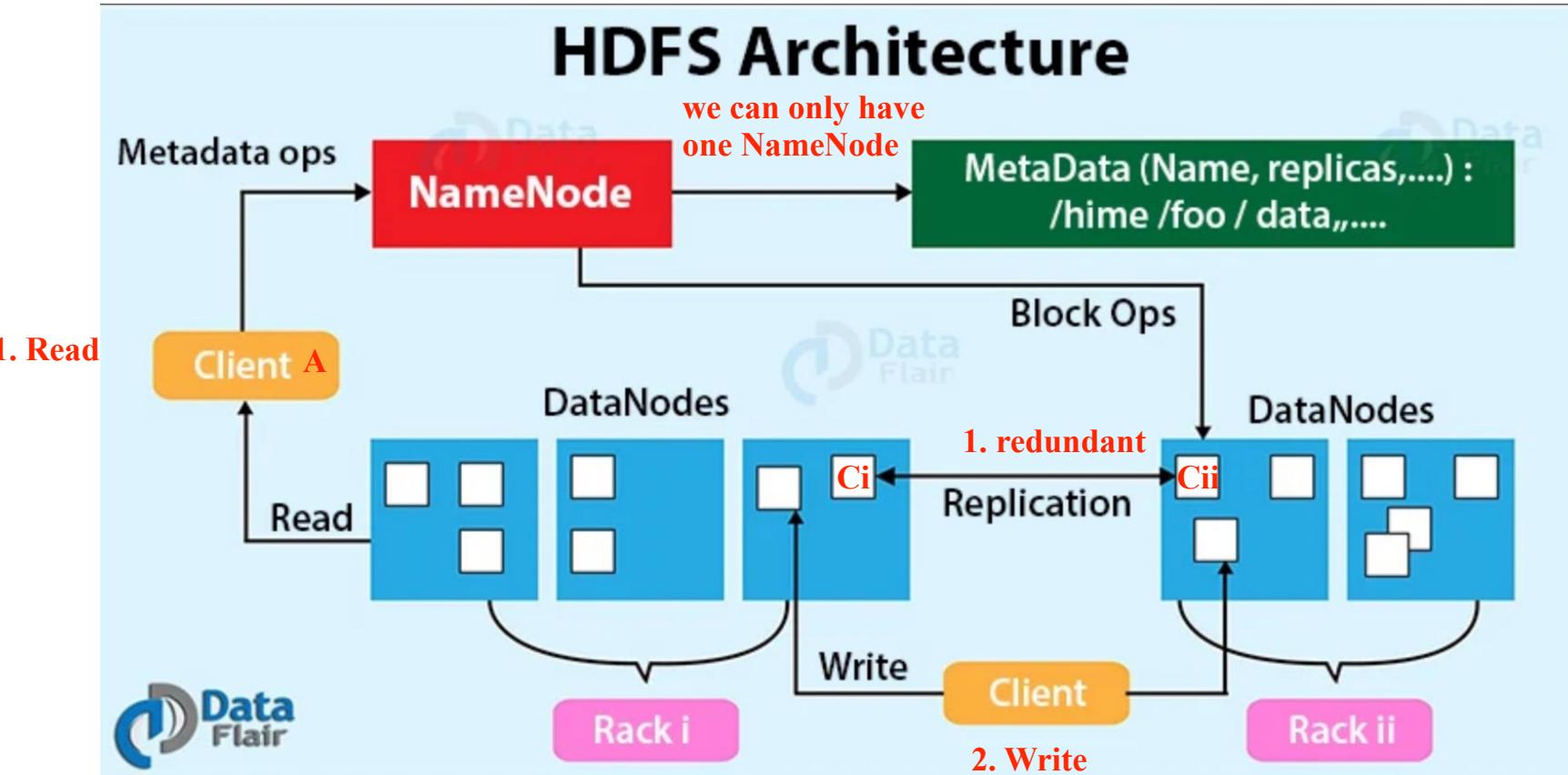


# Stored Across Several Commodity Computers

- Blocks can be **stores across several commodity computers**
- Store **more than one copy** of each block to handle failure
- HDFS **retrieve information from back up copy** of that block in the event failure happened
- **not a big deal** if one single computer goes down

**horizontal scaling vs vertical scaling**

# HDFS Architecture



- master node
- hold metadata

## Master-Slave architecture

- single master node **NameNode**
- multiple slave nodes **DataNode**

## Master node [**NameNode**]

- stores and manages the file system **namespace** **metadata**
- information about blocks of files like block locations, permissions, edit logs

## Slave nodes [**DataNodes**]

- store data blocks of files

<https://bit.ly/2z3g50p>

# What happens if NameNode goes down?

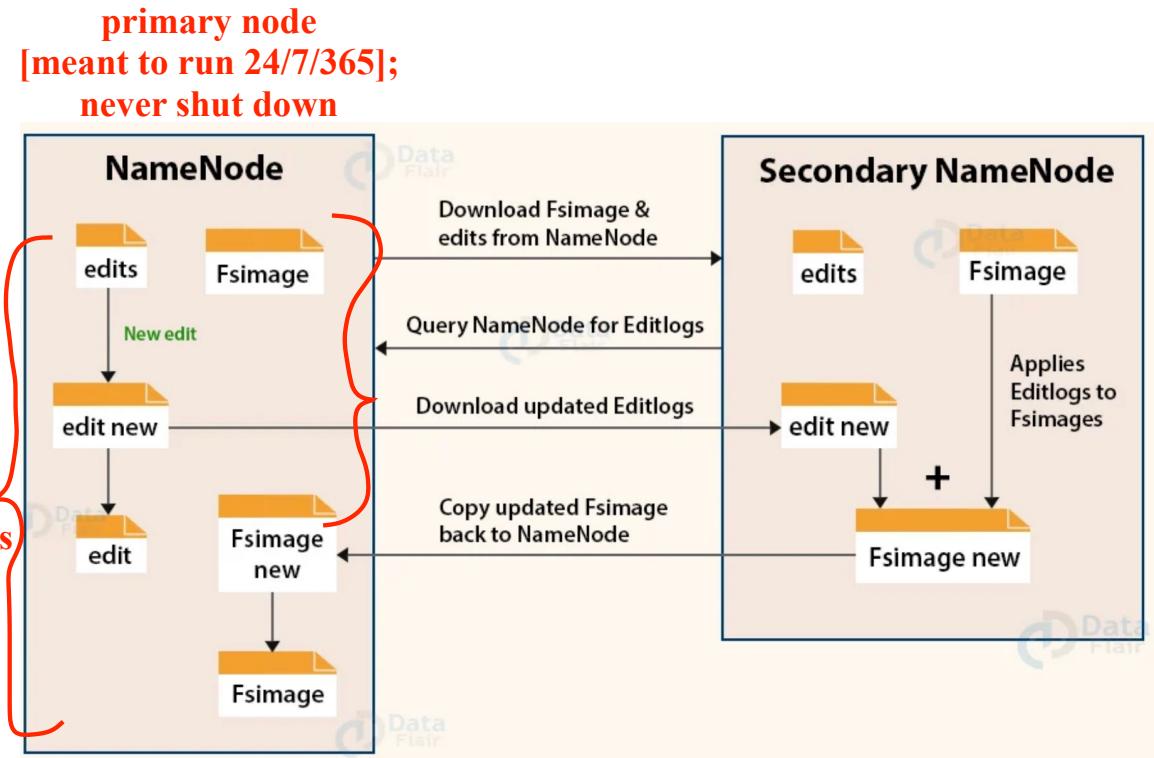
- very important to have **one NameNode active** at a given time
- NameNode is very resilience:
- 1. **Back up Metadata**
  - writes the edit logs to **local disk** and **Network File System (NFS)** [in different rack or data centre]
    - NFS → a network file sharing protocol that defines the way files are stored and retrieved from storage devices across networks
  - **might have some lag** when writing out the back up data [still have something to start from] **in the event of failure happened**

# What happens if NameNode goes down?

[cont...]

- 2. Secondary Namenode

- works as a **helper node** to primary NameNode but doesn't replace primary NameNode *-> in sleeping mode file system image*
- downloads the **Fsimage** file and edit logs file from NameNode
- **periodically applies edit logs to Fsimage and refreshes the edit logs**
- The updated Fsimage is then sent to the NameNode so that NameNode doesn't have to re-apply the edit log records during its restart
- keeps the **edit log size small and reduces the NameNode restart time**



1. edit logs would be super long

- say NameNode crashed on 364th day
- don't need refresh the edit logs from day 0

<https://bit.ly/2z3g50p>

# What happens if NameNode goes down?

[cont...]

- each NameNode **manages a specific namespace volume**
  - especially when NameNode reach their breaking points of handling too much small files
  - separate NameNode responsible for **different namespace volume**
  - only lose a portion of the data if one of those NameNodes went down
  - namespace → a file or directory which is handled by NameNode [information about files, metadata, directories, etc...]

# What happens if NameNode goes down?

[cont...]

- 4. High availability **not an inherent part of the HDFS architecture**
- Hot standby NameNode using shared edit log [**not part of HDFS**]
- Zookeeper tracks active NameNode
- Use **extreme measures** to ensure **only one NameNode is used** at a time [such as physically cut off the power to a NameNode if Zookeeper thinks an active NameNode is down]

# Fsimage in HDFS

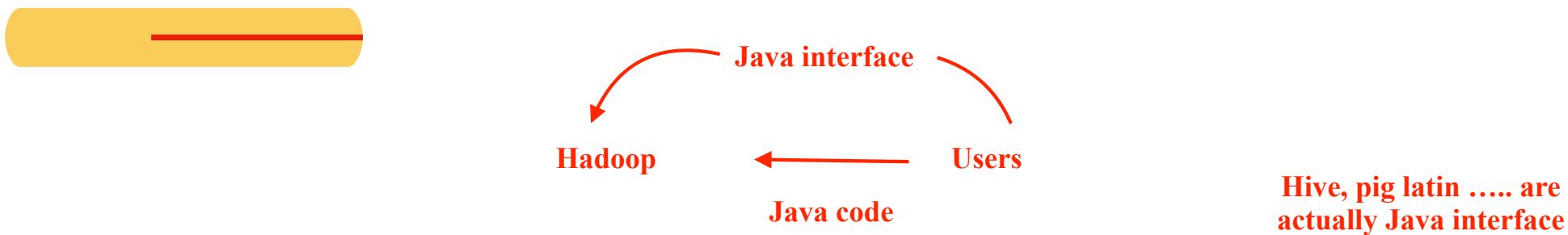
- Fsimage (File System Image) in Hadoop Distributed File System (HDFS) represents a *snapshot of the file system's metadata*
- Contains **metadata** such as *file names, directory structure, permissions, modification times, and file sizes*
- Periodically created by the NameNode as *part of the checkpointing process*
- *Merges latest edits from edit logs* to create an updated snapshot of metadata
- NameNode *loads fsimage into memory and applies edit logs* since last checkpoint instead of replaying all edit logs
- Stored on NameNode's disk along with edit logs to *enable recovery of NameNode's state in case of failure.*

# Edit Logs in HDFS

- Edit logs in Hadoop Distributed File System (HDFS) are *sequential records of metadata changes* in *chronological order* since the last checkpoint
- Record operations like *file creations, deletions, modifications, and directory changes*
- Each entry in edit log corresponds to a *single transaction or operation*
- Used with fsimage to restore metadata after NameNode failure or restart
- Edit log replayed to *reconstruct file system state* since last checkpoint
- Kept until merged into new fsimage during checkpoint
- After merge, can be truncated to reclaim storage space

# Using HDFS

- UI (Ambari) [copy files, view files...]
- Command-Line interface
- HTTP / HDFS Proxies
- Java interface [Hadoop is written in Java under the hood; doesn't mean you need to write Java code 😊; you can still use R, Python...]

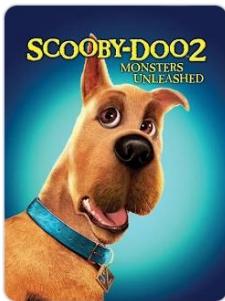


# MapReduce Fundamental Concept

- Why MapReduce?
  - Distributes the processing of data on the cluster
  - Divides our data into partitions that are **MAPPED (transformed)** and **REDUCED (aggregated)** by mapper and reducer functions
  - **Resilient to failure** – an application master monitors mappers and reducers on each partition
- **Mapper** transforms the data as it comes in **one line at a time**
  - Extracting and organizing the data as **key/value pair**
- **Reducer** aggregates data together

**MapReduce -> Fundamental building blocks of Hadoop**

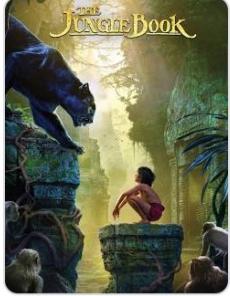
# Illustrate with an example



Scooby Doo 2: ...



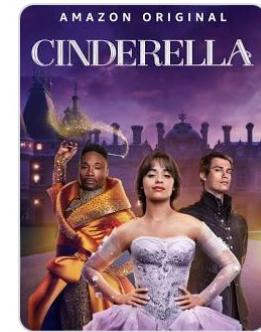
Kung Fu Panda



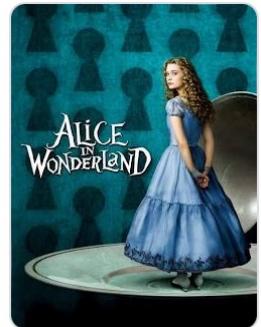
The Jungle Book



Hotel Transylvan...



Cinderella

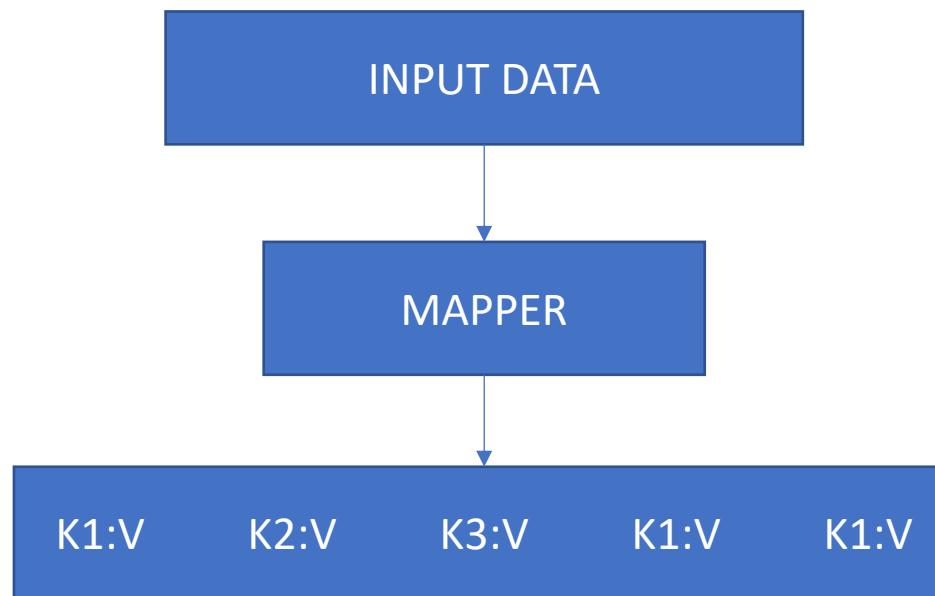


Alice in Wonderl...

How many movies did each user rate in the  
MovieLens data set?

# How MapReduce Works: Mapper

- The MAPPER converts raw source data into key/value pair

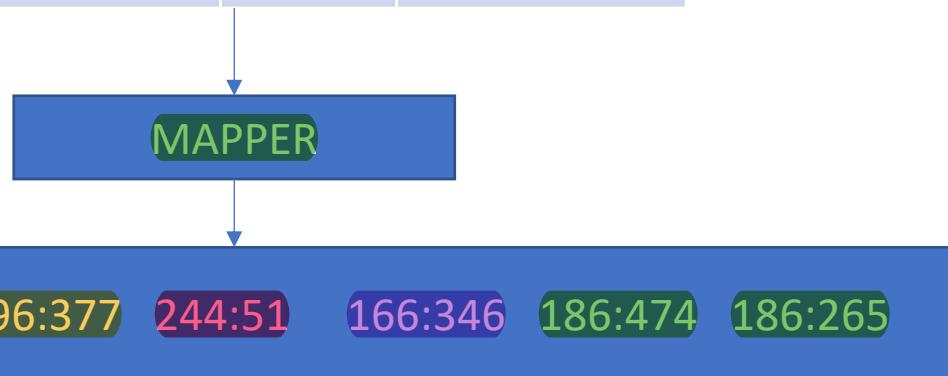


Key [K] = user ID  
Value [V] = movie name

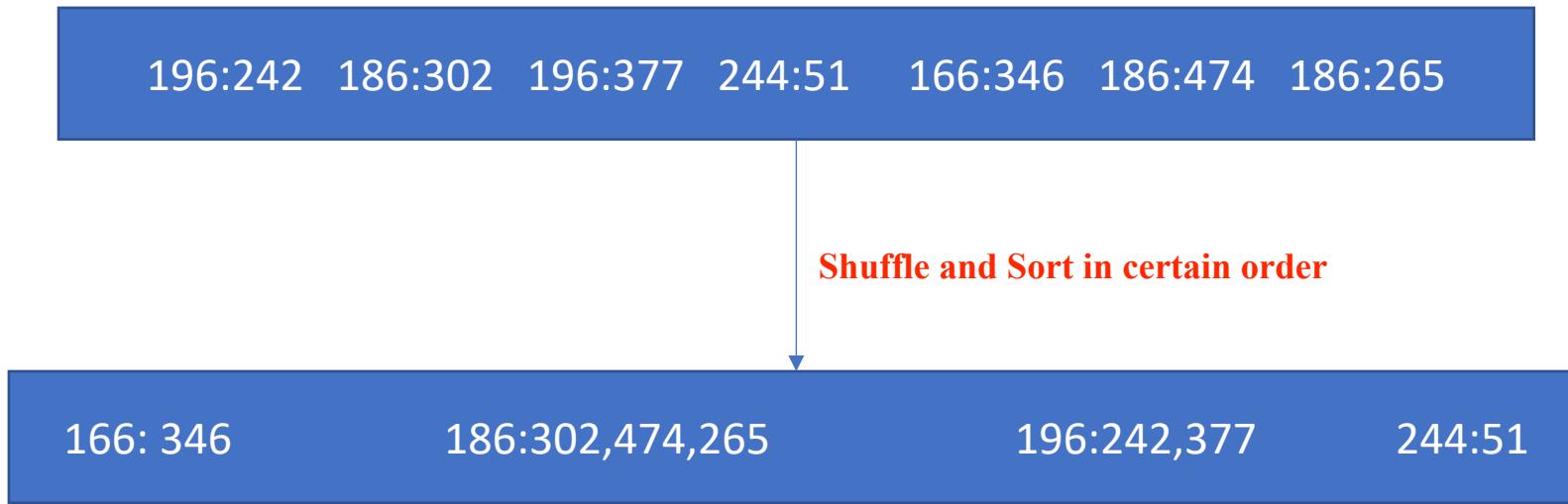
Row by Row

# Example: MovieLens data (u.data)

USER ID	MOVIEID	Rating	TIMESTAMP
196	242	3	881250949
186	302	3	891717742
196	377	1	878887116
244	51	2	880606923
166	346	1	886397596
186	474	4	884182806
186	265	2	881171488

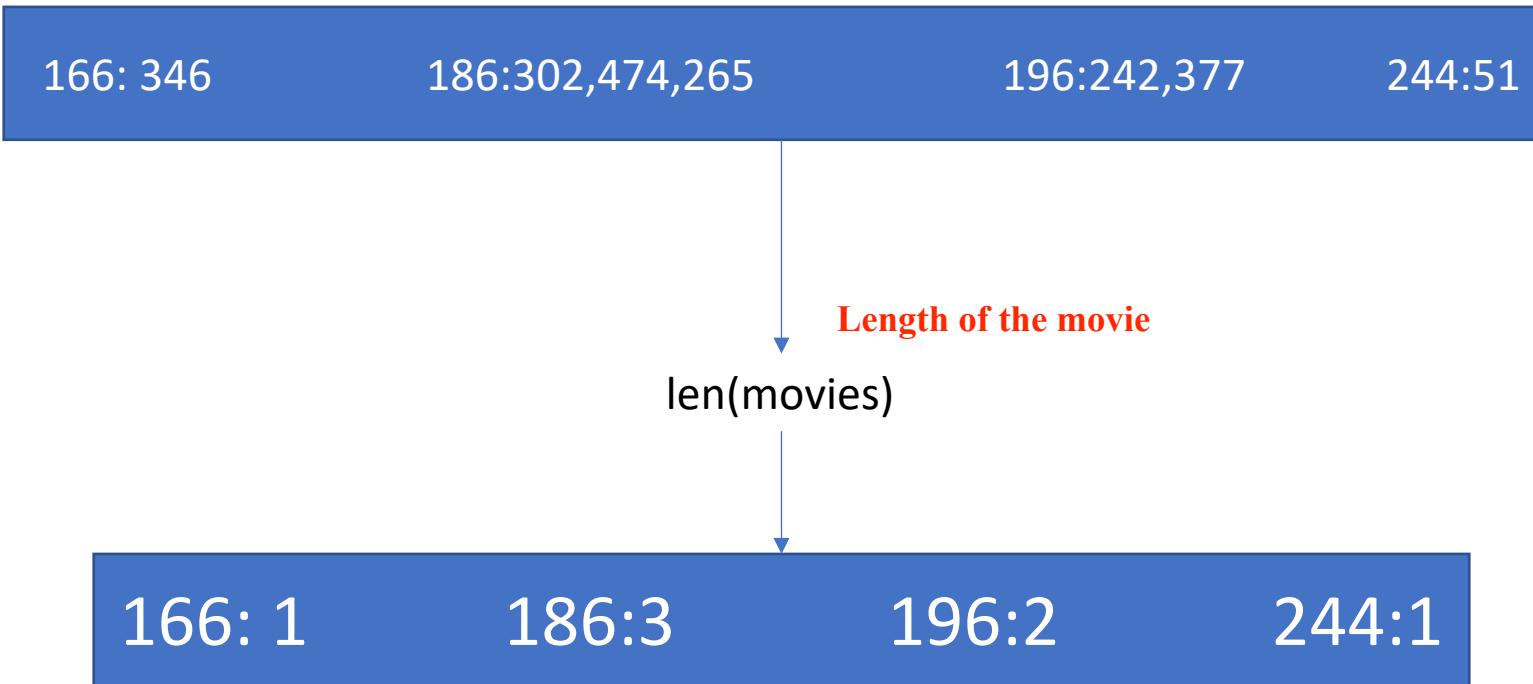


# MapReduce Sorts and Groups the Mapped Data (“Shuffle and Sort”)



- 1. Aggregate all given values for each unique key
- 2. Sort all given keys

# The REDUCER Processes Each Key's Value



**How many movies did each user rate?**

# Putting it All Together

## HDFS

- data would be distributed / broken up / partitioned
- data be reduced
- data would be replicated / redundant
- from back up copy

MapReduce, NameNode, DataNode....  
They always have back up copy

USER ID	MOVIEID	Rating	TIMESTAMP
196	242	3	881250949
186	302	3	891717742
196	377	1	878887116
244	51	2	880606923
166	346	1	886397596
186	474	4	884182806
186	265	2	881171488

raw files

## MAPPER

reads the data one line at one time to become key/value pair

196:242 186:302 196:377 244:51 166:346 186:474 186:265

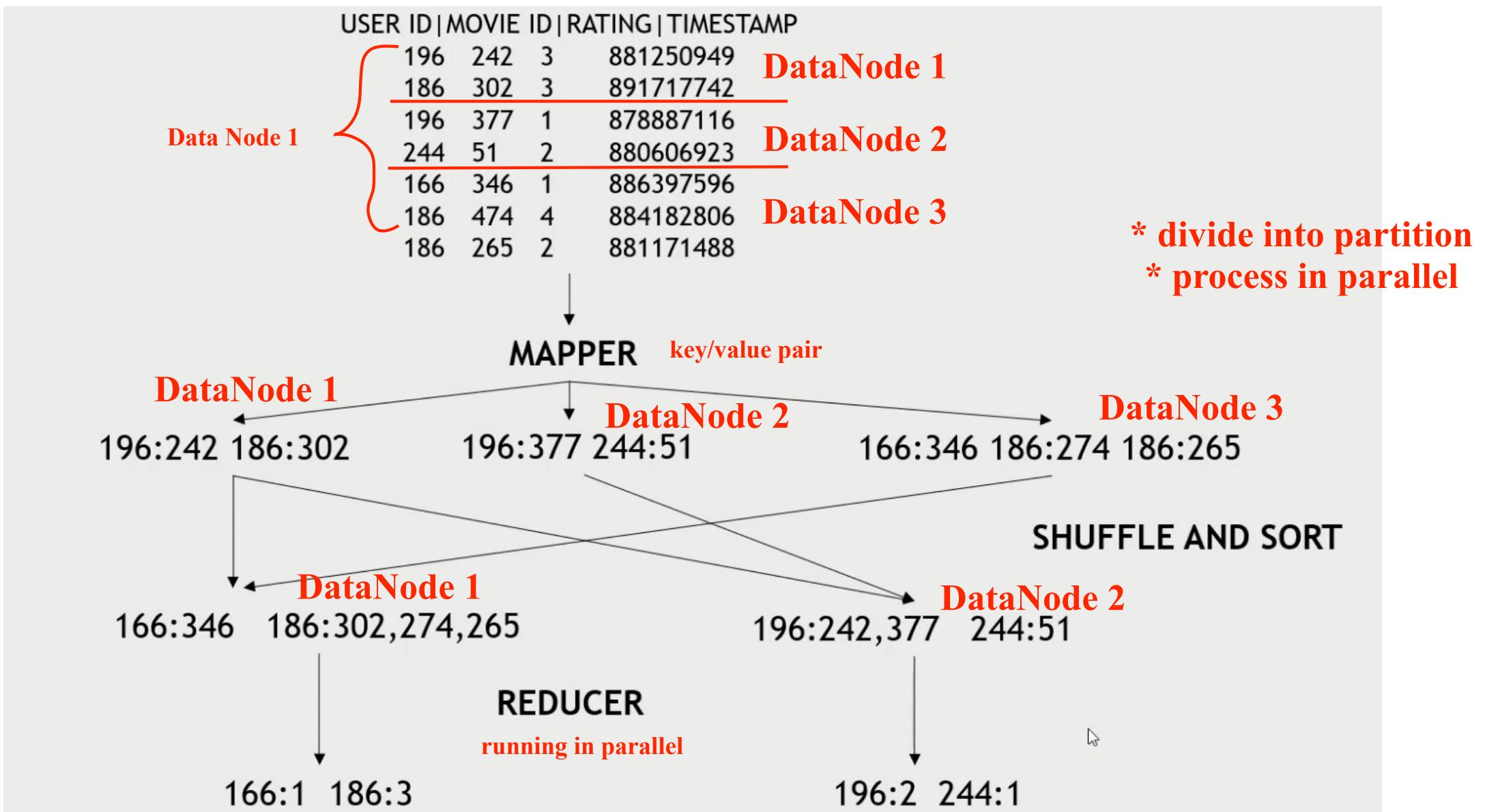
## Shuffle & Sort

166: 346 186:302,474,265 196:242,377 244:51

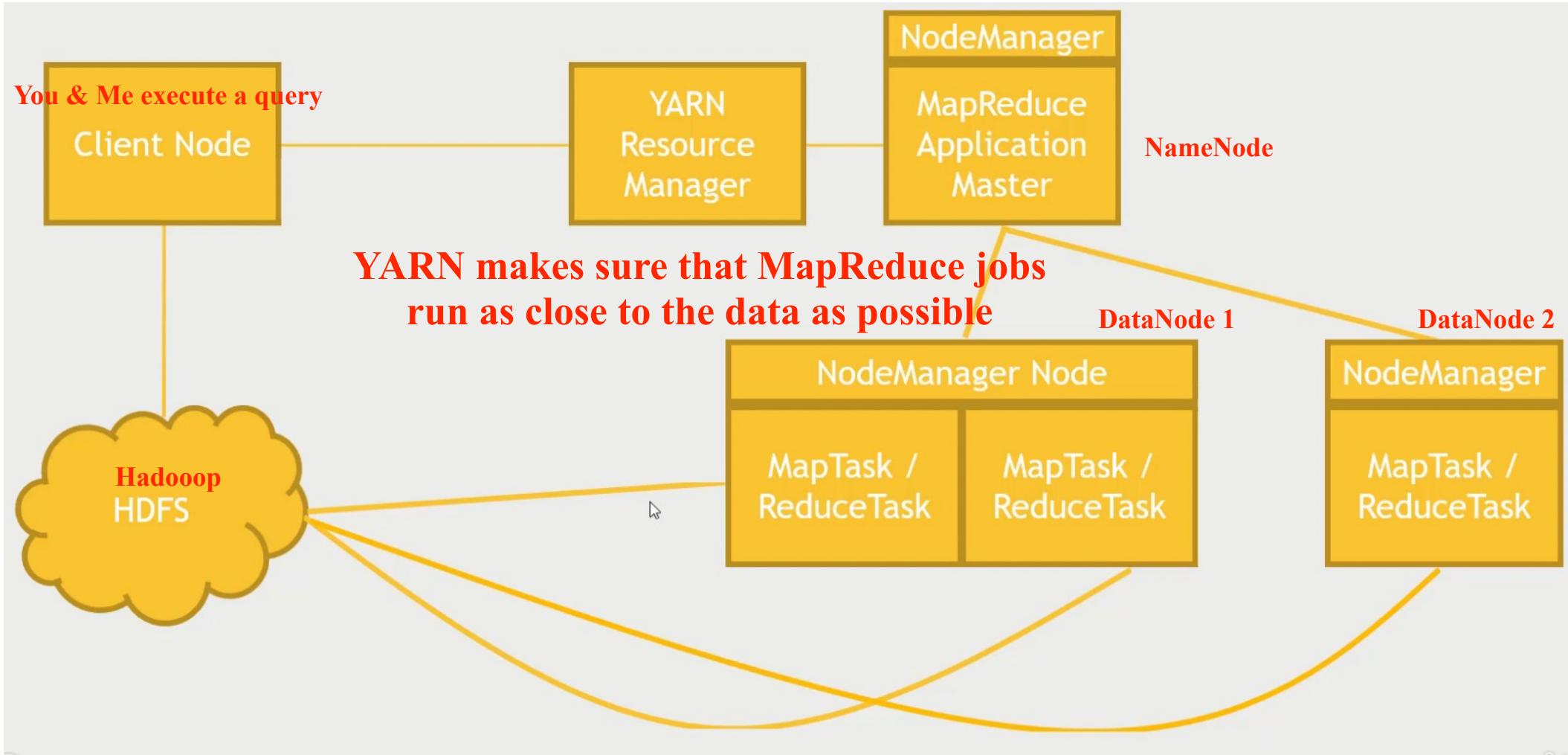
## Reducer

166: 1 186:3 196:2 244:1

## MapReduce - if you understand this messy figure



# What's happening



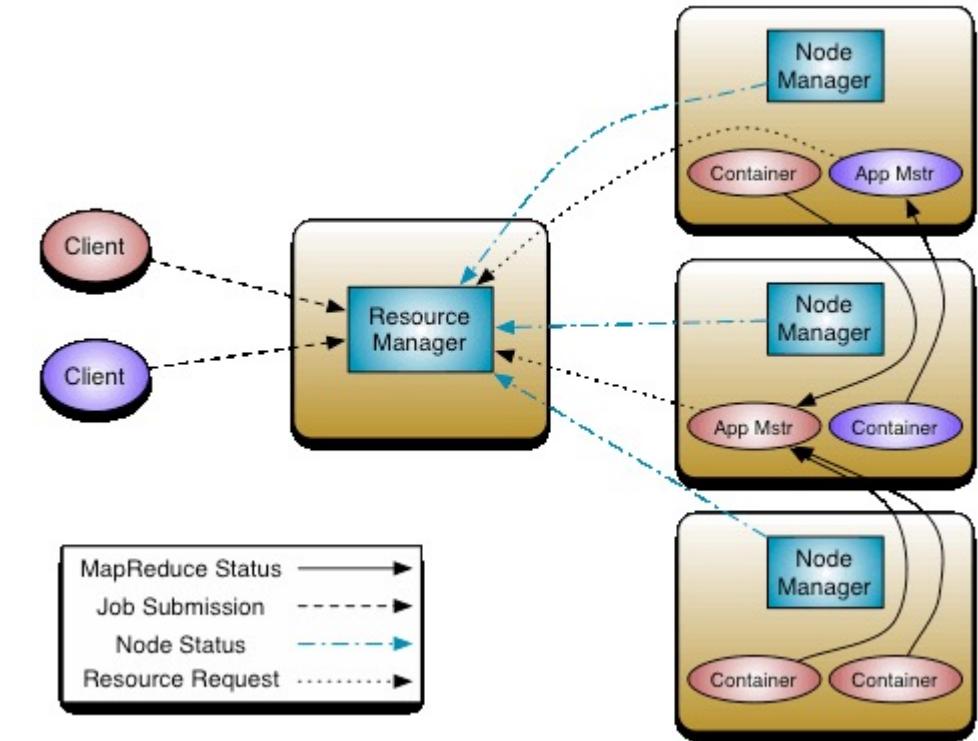
# Handling Failure

- Application master monitors worker tasks for errors or hanging
  - *Restart as needed*
  - *Preferably on a different node*
- What if the application master goes down?
  - *YARN can try to restart it*
- What if entire Node goes down?
  - *This could be the application master*
  - *The resource manager will try to restart it*
- What if the resource manager goes down?
  - *Can set up “high availability” (HA) using Zookeeper to have a hot standby*

**Overwatch of the watch**

# YARN – Yet Another Resource Negotiator

- employs a master-slave model and includes several components
- The global **Resource Manager** is the ultimate authority that arbitrates resources among all applications in the system.
- The per-application **Application Master** negotiates resources from the Resource Manager and works with the Node Managers to execute and monitor the component tasks.
- The per-node slave **Node Manager** is responsible for launching the applications' containers, monitoring their resource usage and reporting to the Resource Manager.



<https://bit.ly/3vqVPmX>

# MapReduce: An example

- How many of each rating type exist [from MovieLens dataset]?



How many people give ratings of 5

...

...

...

...

...

...

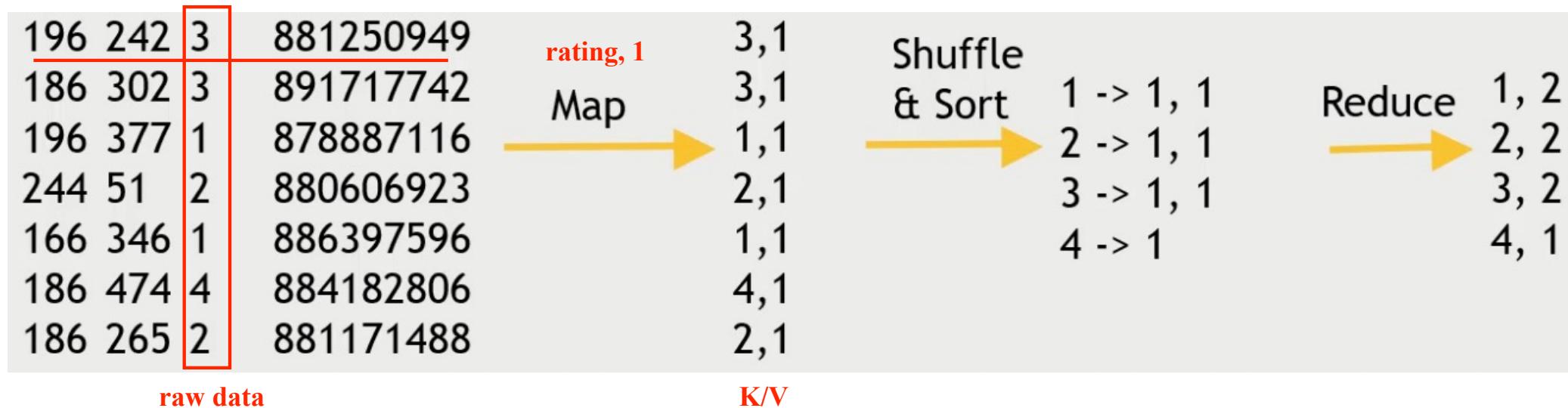
...

...

How many people give ratings of 1

# Turning it into a MapReduce Problem

- **MAP** each input line to (*rating*, 1)
- **REDUCE** each rating with the sum of all the 1's



# Writing the Mapper



Python function

3 parameters     "\_" -> used to hold redundant data / data that we don't need

```
def mapper_get_ratings(self, _, line):  
    (userID, movieID, rating, timestamp) = line.split('\t')  
    yield rating, 1  
    output
```

# Writing the Reducer



Python function

3 parameters

```
▶ def reducer_count_ratings(self, key, values):  
    yield key, sum(values)
```

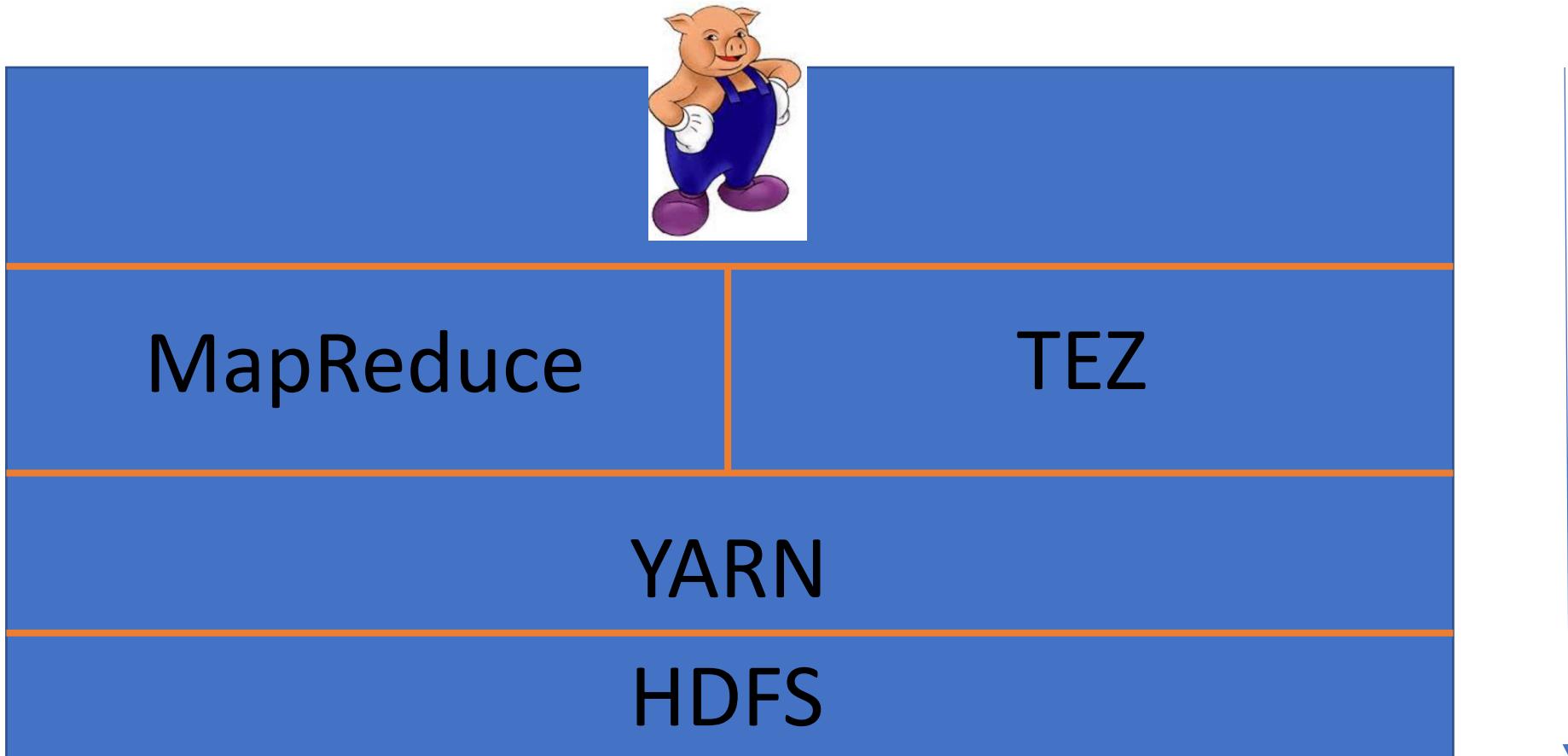
# Putting all together

Save as:  
MapReducer.py

```
▶ from mrjob.job import MRJob  
from mrjob.step import MRStep  
  
class RatingsBreakdown(MRJob):  
    def steps(self):  
        return [  
            MRStep(mapper=self.mapper_get_ratings,  
                   reducer=self.reducer_count_ratings)  
        ]  
  
    def mapper_get_ratings(self, _, line):  
        (userID, movieID, rating, timestamp) = line.split('\\t')  
        yield rating, 1  Mapper function  
  
    def reducer_count_ratings(self, key, values):  
        yield key, sum(values) Reducer function  
  
if __name__ == '__main__': → runs from command-line  
    RatingsBreakdown.run()
```

*to run:*  
***python MapReduce.py***

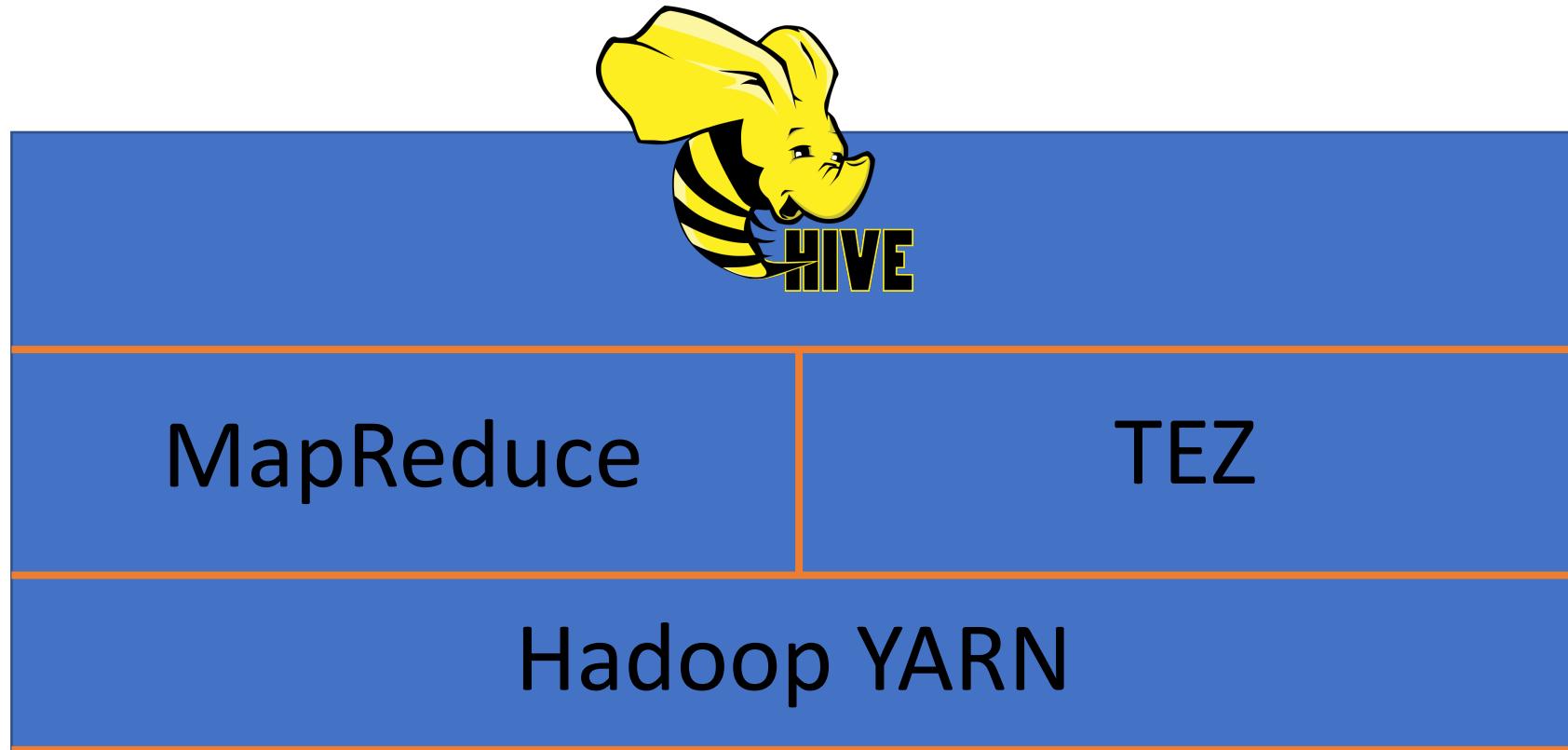
# Pig Architecture



# Why Pig?

- Writing mappers and reducers by hand takes a long time
- Pig introduces Pig Latin, a scripting language that lets us use SQL-like syntax to define the map and reduce steps

# Hive Architecture



- Translates SQL queries to MapReduce or TEZ jobs on Hadoop clusters
- Hive will break down the SQL queries into Mappers and Reducers

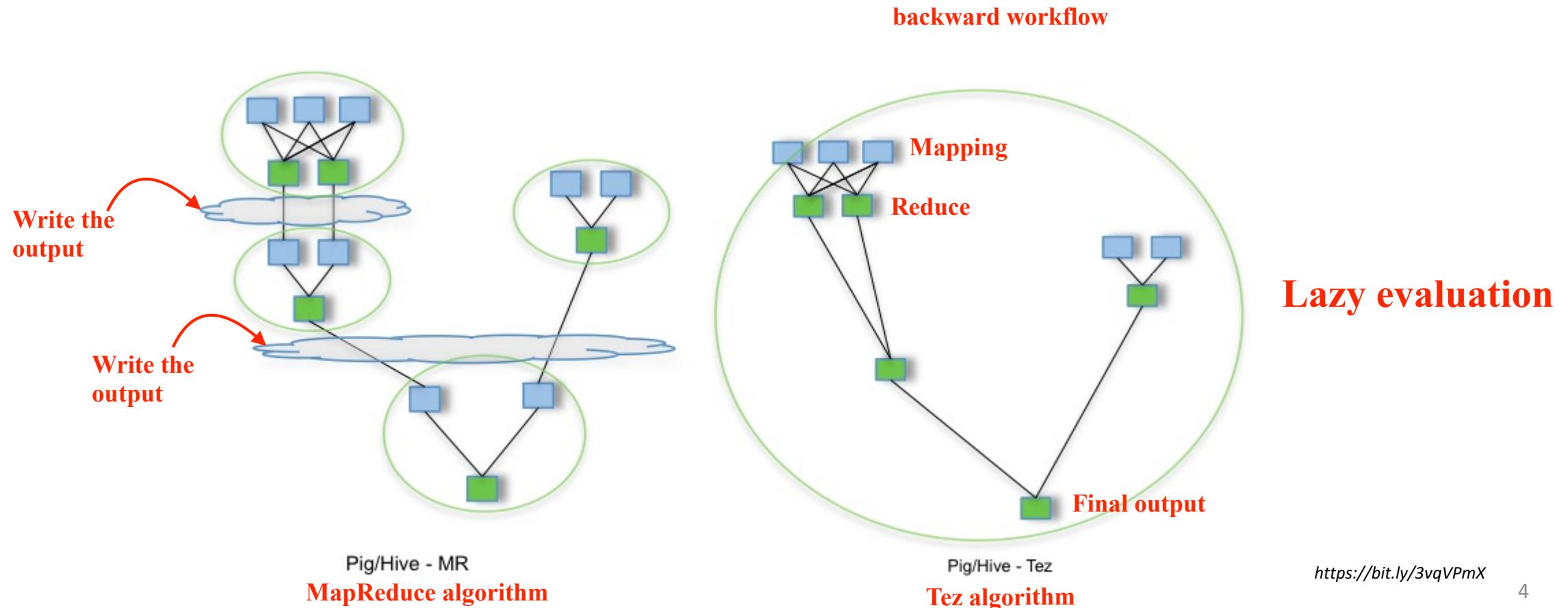
# Why Hive

- Use familiar SQL syntax (**HiveQL**)
- Interactive
- Scalable – works with “big data” on a cluster
  - Appropriate for data warehouse application
- Easy Online Analytics processing (OLAP) – much easier than writing MapReduce in Java

# TEZ

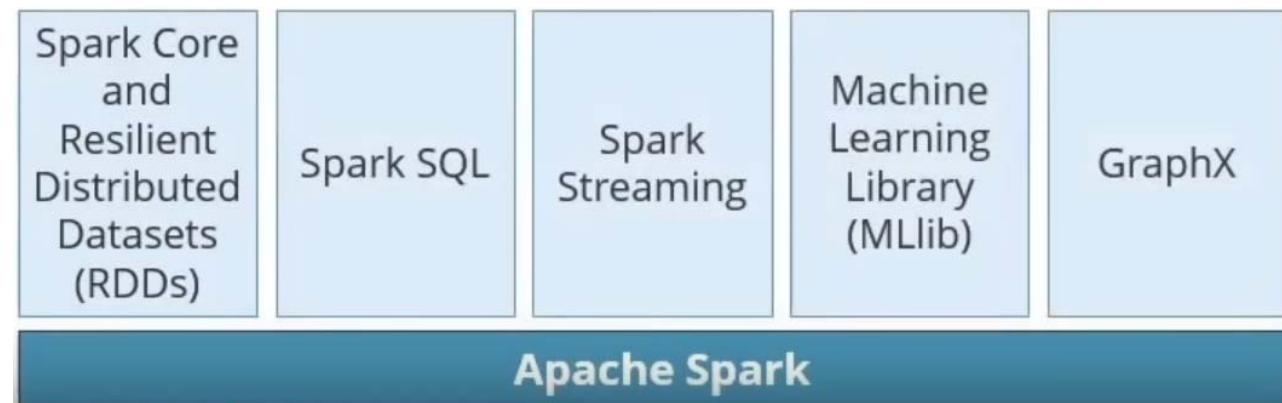
Once client execute a function, Tez will evaluate what's the final output; then work backward to get the most optimal pathway

- an application framework which allows for a complex **directed-acyclic-graph (DAG)** of tasks for processing data.



# Introduction to Spark

- Spark is an open-source cluster computing Framework
  - It is suitable for **real-time** processing, trivial operations, and processing large data on network.
  - Provides up to 100 times faster performance for a few applications with **in-memory** primitives, as compared to the two-stage disk-based MapReduce paradigm of Hadoop.
  - Is suitable for machine learning algorithms, as it allows programs to load and query data repeatedly



# Application of In-Memory Processing

With column-centric database coming, the similar information can be stored together. The working of in-memory processing can be explained as below:

- The entire information is **loaded into memory**, eliminating the need for indexes, aggregates, optimized databases, star schemas, and cubes.
- **Compression algorithms** are used by most of the in-memory tools, thereby reducing the in-memory size.
  - reduce the number of bytes required to represent data and the amount of memory required to store images
- With in-memory tools, the analysis of data can be flexible in size and can be accessed within seconds by concurrent users with an excellent analytics potential.

# Row-based vs column-centric database

Name	City	Age
Matt	Los Angeles	27
Dave	San Francisco	30
Tim	Oakland	33

**Normal Table**

Facebook\_Friends

Name	City	Age
Matt	Los Angeles	27
Dave	San Francisco	30
Tim	Oakland	33

Row-based

Column-based

Hadoop is about partitioned data; distributed processing

Matt	Los Angeles	27	Dave	San Francisco	30	Tim	Oakland	33
------	-------------	----	------	---------------	----	-----	---------	----

- \* Same type of Data
- \* Mapped new data easier

Disk 1		
Name	City	Age
Matt	Los Angeles	27

Disk 2		
Name	City	Age
Dave	San Francisco	30

Disk 3		
Name	City	Age
Tim	Oakland	33

Names  
that is  
age  $\geq 30$   
years

Disk 1		
Name	City	Age
Matt	Dave	Tim

Disk 2		
City	Age	Age
Los Angeles	San Francisco	Oakland

Disk 3		
Age	Age	Age
27	30	33

Names  
that is  
age  $\geq 30$   
years

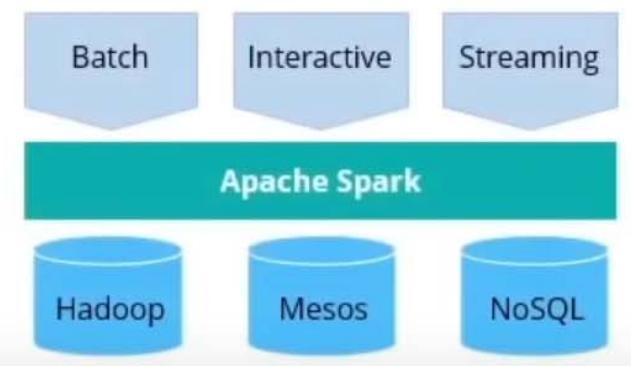
# Spark Advantages

- **Speed:** Extending the MapReduce model to support computations like stream processing and interactive queries.
- **Combination:** Covering various workloads that used to require different distributed systems, which makes combining different processing types and allows easy tools management
- **Hadoop Support:** Allowing to create distributed datasets from any file stored in the Hadoop Distributed File System (HDFS) or any other supported storage systems.



## Why does unification matter?

- Developers need to learn only one Platform
- Users can take their apps to everywhere



# A good programmer is a super lazy person :)

## Lazy Evaluation Example - The waiter takes orders patiently

