Prepared By : Ravi

Source: spark.apache.org & www.databricks.com

Unstructured data

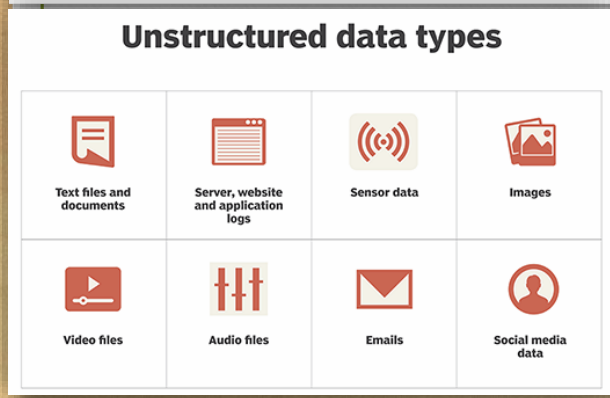
The university has 5600 students.
John's ID is number 1, he is 18 years old and already holds a B.Sc. degree.
David's ID is number 2, he is 31 years old and holds a Ph.D. degree. Robert's ID is number 3, he is 51 years old and also holds the same degree as David, a Ph.D. degree.

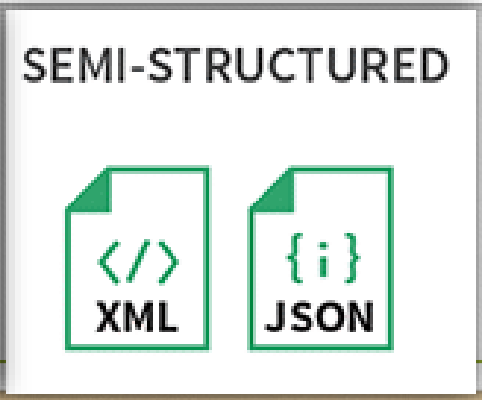
Semi-structured data

```
<University>
 <Student ID="1">
  <Name>John</Name>
  <Age>18</Age>
  <Degree>B.Sc.</Degree>
 </Student>
 <Student ID="2">
  <Name>David</Name>
  <Age>31</Age>
  <Degree>Ph.D. </Degree>
 </Student>
....
</University>
```

Structured data

ID	Name	Age	Degree
1	John	18	B.Sc.
2	David	31	Ph.D.
3	Robert	51	Ph.D.
4	Rick	26	M.Sc.
5	Michael	19	B.Sc.





What is Data & Metadata?

Unstructured data

The university has 5600 students.
John's ID is number 1, he is 18 years old and already holds a B.Sc. degree.
David's ID is number 2, he is 31 years old and holds a Ph.D. degree. Robert's ID is number 3, he is 51 years old and also holds the same degree as David, a Ph.D. degree.

Semi-structured data

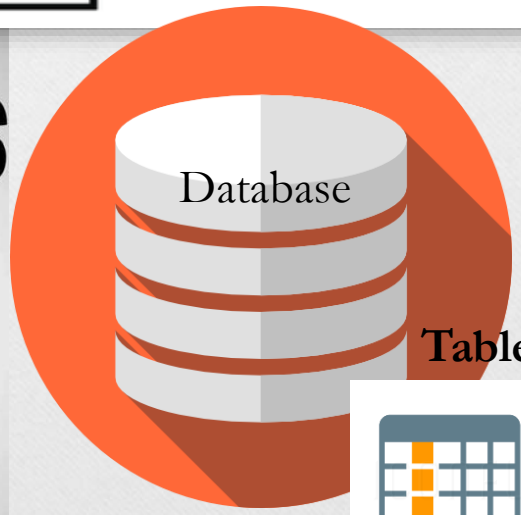
```
<University>
  <Student ID="1">
    <Name>John</Name>
    <Age>18</Age>
    <Degree>B.Sc.</Degree>
  </Student>
  <Student ID="2">
    <Name>David</Name>
    <Age>31</Age>
    <Degree>Ph.D. </Degree>
  </Student>
 ....
</University>
```

Structured data

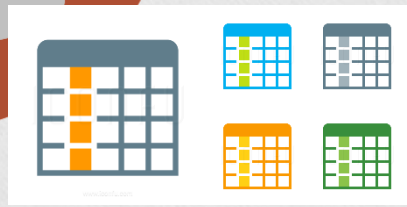
ID	Name	Age	Degree
1	John	18	B.Sc.
2	David	31	Ph.D.
3	Robert	51	Ph.D.
4	Rick	26	M.Sc.
5	Michael	19	B.Sc.

Files

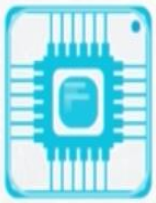
Folders



Tables and cols

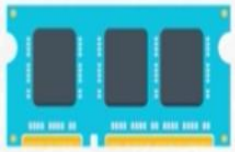


Fastest



CPU

200x faster than memory



MEMORY

15x faster than SSD

SSD

20x faster than network

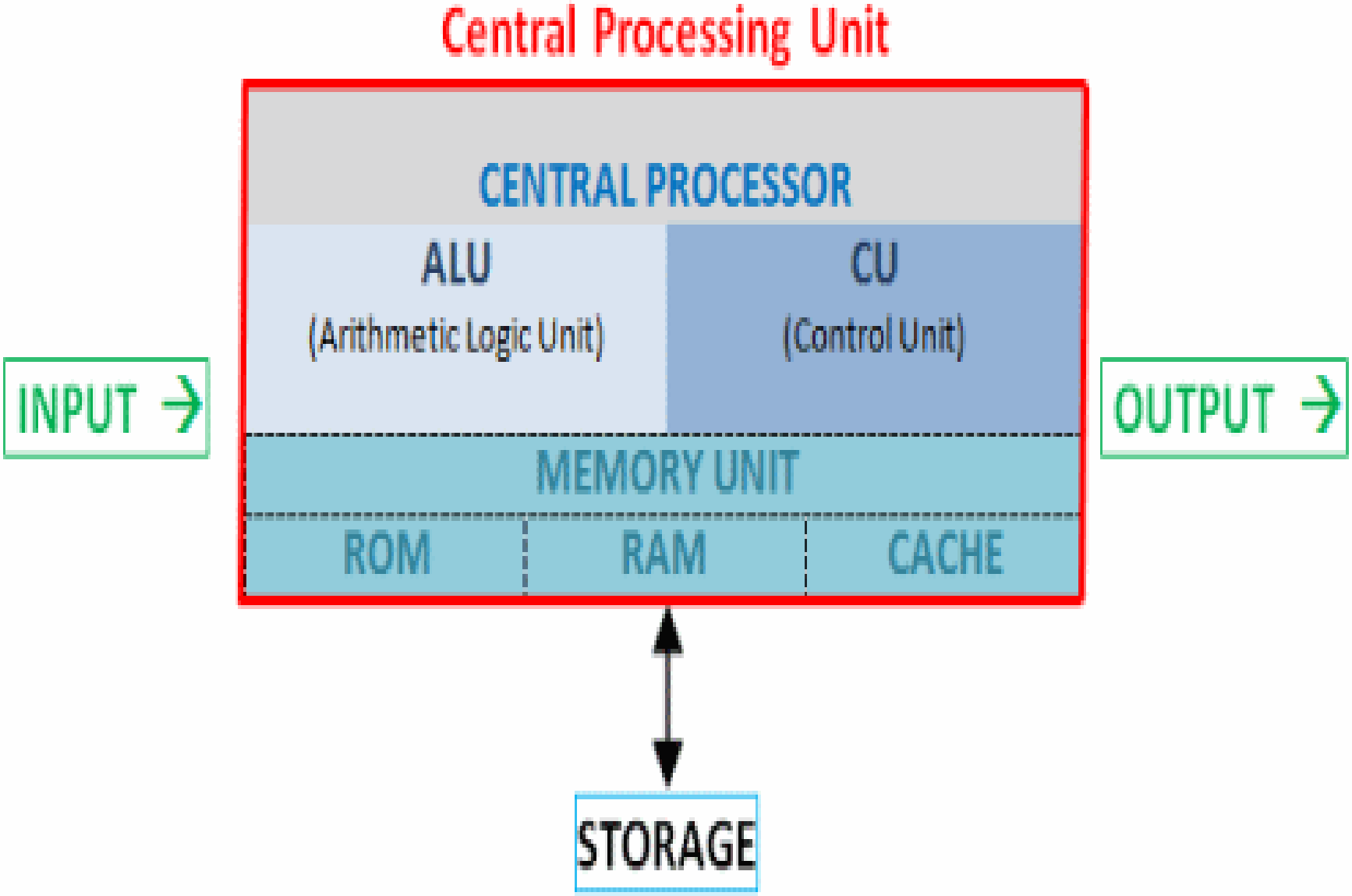
NETWORK

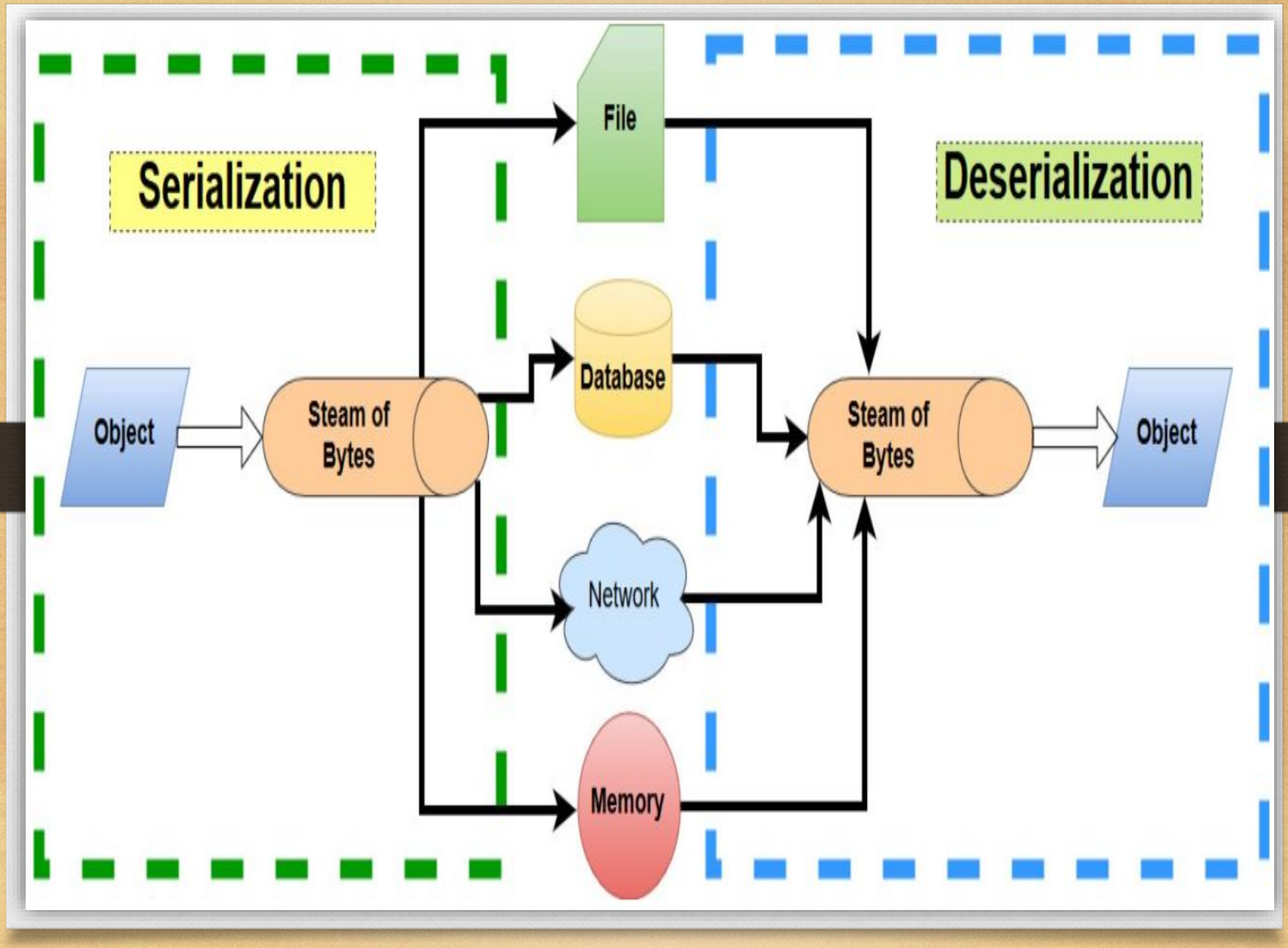
Slowest

Data Transfer stats

Data size \ bandwidth	50 Mbps	100 Mbps	200 Mbps	500 Mbps	1 Gbps	10 Gbps
1GB	2.7 min	1.4 min	0.7 min	0.3 min	0.1 min	0.0 min
10GB	27.3 min	13.7 min	6.8 min	2.7 min	1.3 min	0.1 min
100GB	4.6 hrs	2.3 hrs	1.1 hrs	0.5 hrs	0.2 hrs	0.0 hrs
1TB	46.6 hrs	23.3 hrs	11.7 hrs	4.7 hrs	2.3 hrs	0.2 hrs
10TB	19.4 days	9.7 days	4.9 days	1.9 days	0.9 days	0.1 days
100TB	194.2 days	97.1 days	48.5 days	19.4 days	9.5 days	0.9 days
1PB	64.7 mo	32.4 mo	16.2 mo	6.5 mo	3.2 mo	0.3 mo
10PB	647.3 mo	323.6 mo	161.8 mo	64.7 mo	31.6 mo	3.2 mo

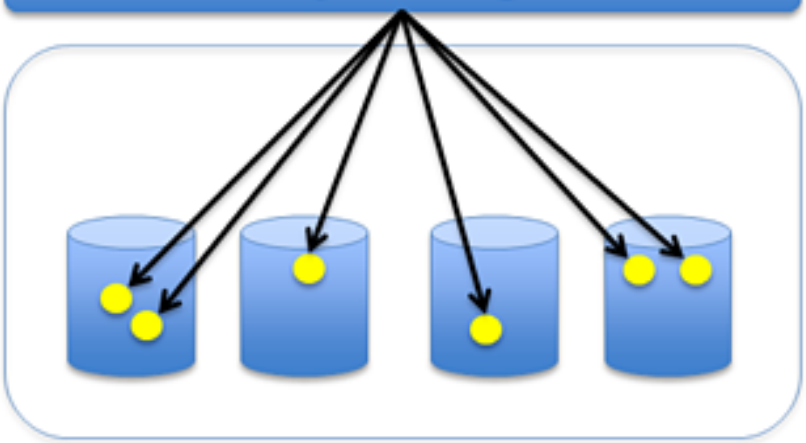
Online	Offline


Central Processing Unit
CENTRAL PROCESSOR
ALU
(Arithmetic Logic Unit)
CU
(Control Unit)
INPUT
OUTPUT
MEMORY UNIT
ROM
RAM
CACHE
STORAGE


Serialization
Deserialization
File
Database
Network
Memory
Object
Steam of Bytes
Steam of Bytes
Object

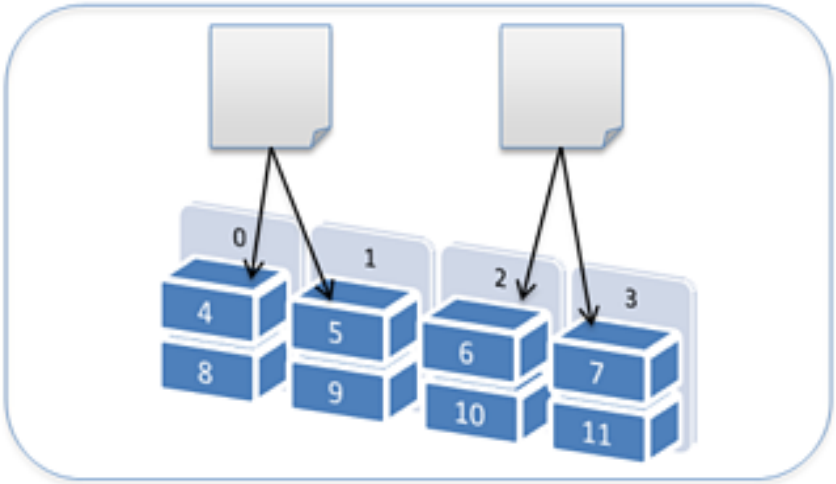
HTTP(S) Interface

Object Storage



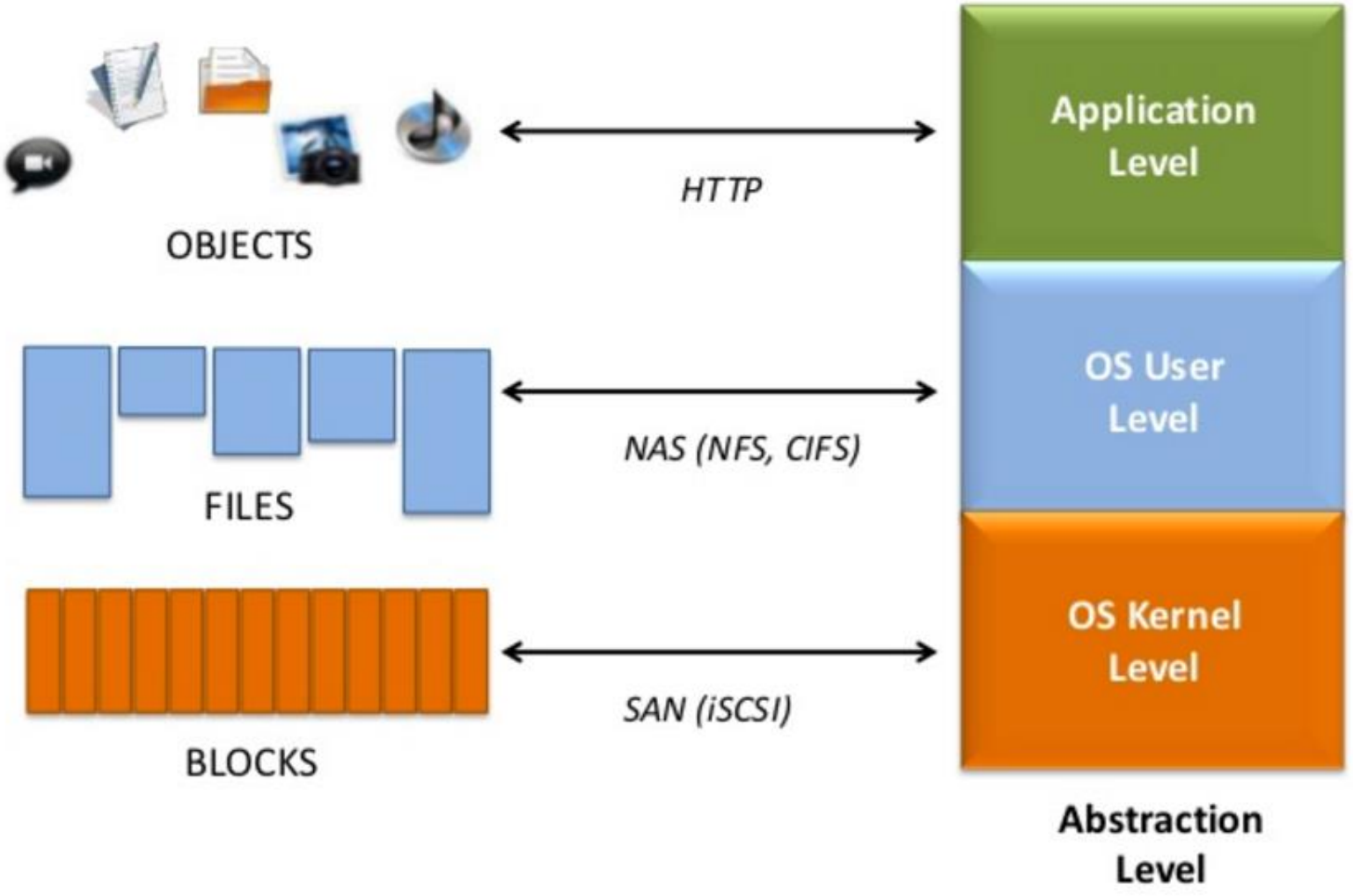
- Store virtually unlimited files.
- Maintain file revisions.
- HTTP(S) based interface.
- Files are distributed in different physical nodes.

Block Storage

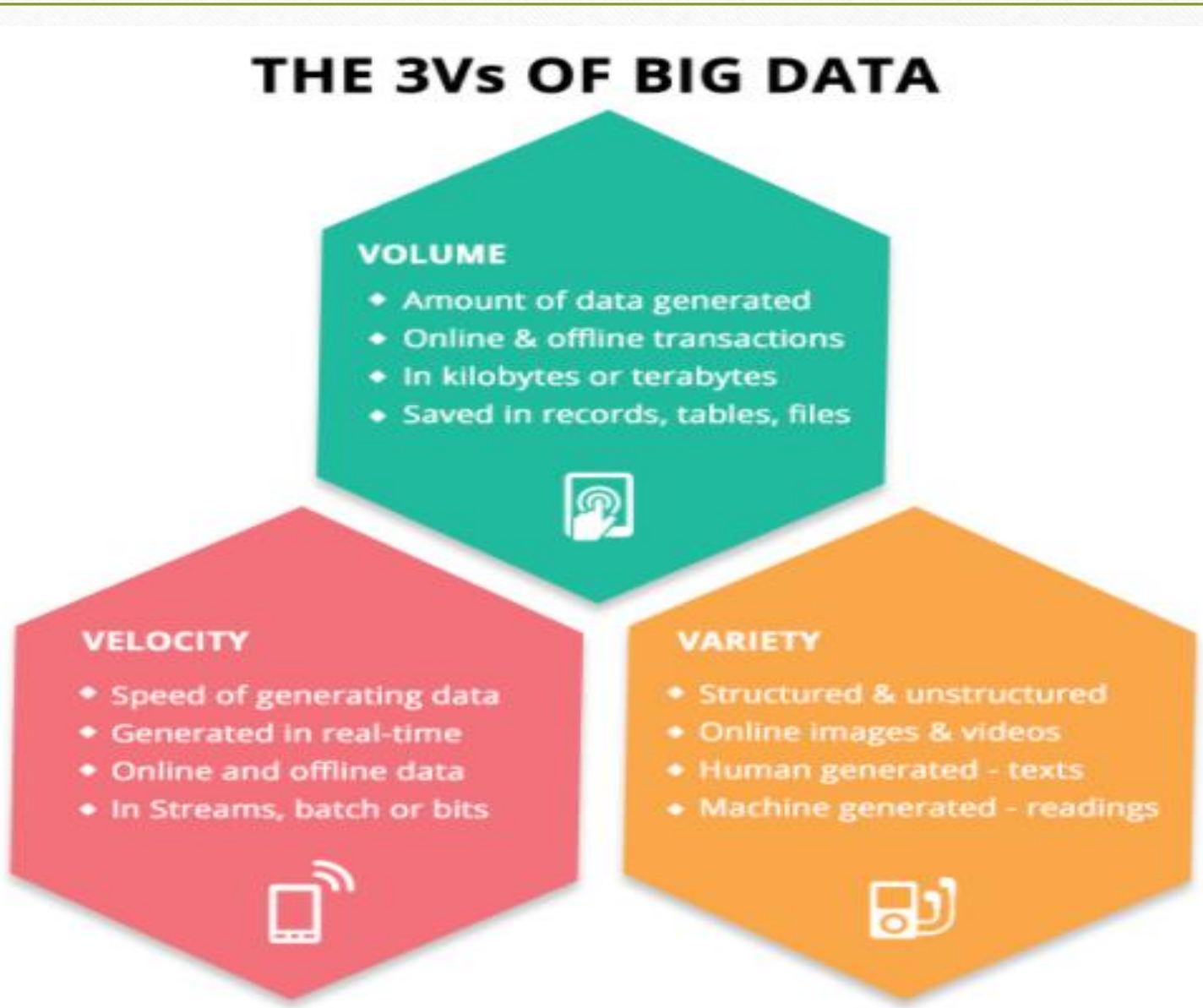


- File is split and stored in fixed sized blocks.
- Capacity can be increased by adding more nodes.
- Suitable for applications which require high IOPS, database, transactional data.

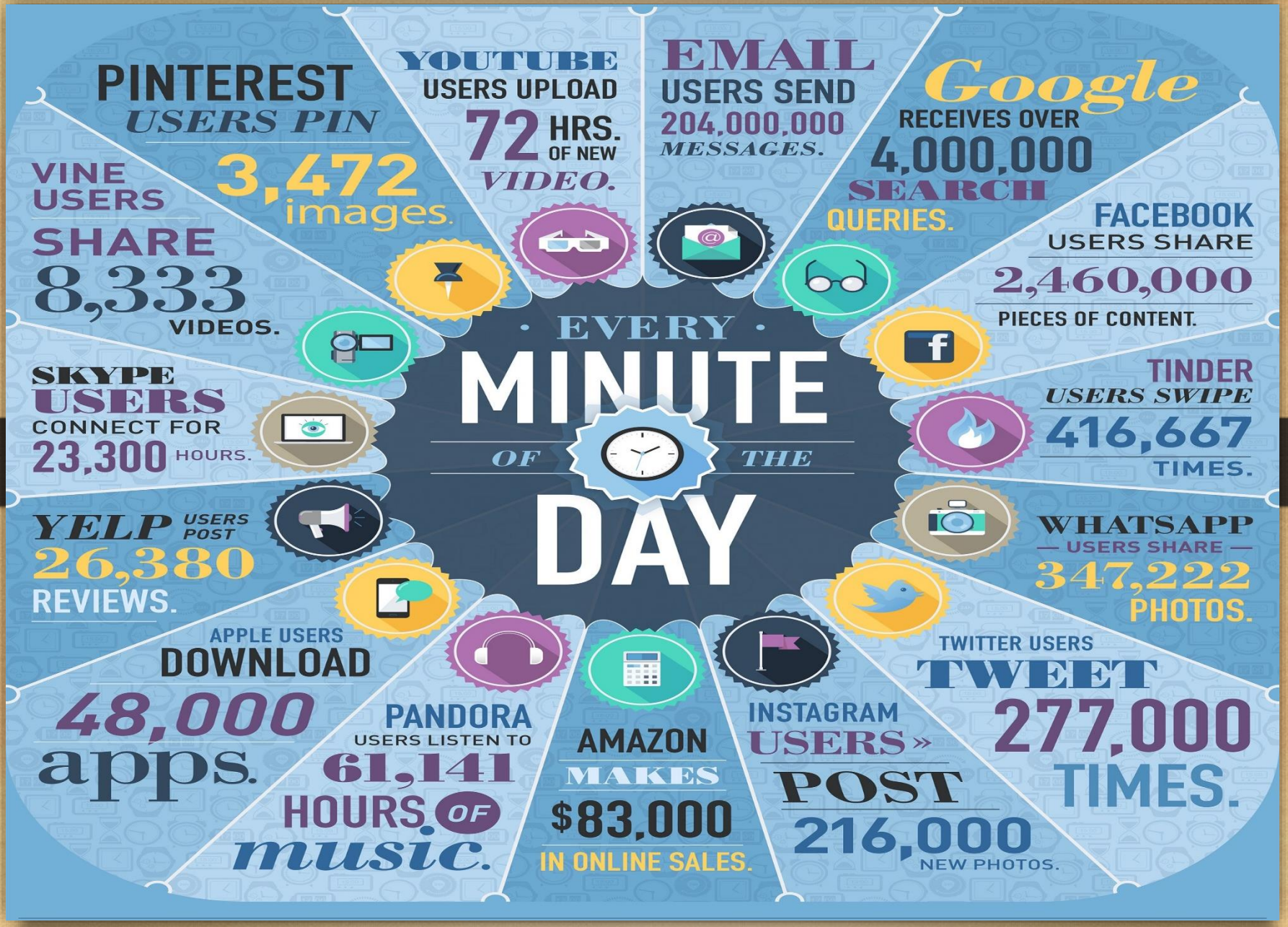
Object vs. File vs. Block Storage



What is Big Data?


THE 3Vs OF BIG DATA
VOLUME
Amount of data generated
Online & offline transactions
In kilobytes or terabytes
Saved in records, tables, files
VELOCITY
Speed of generating data
Generated in real-time
Online and offline data
In Streams, batch or bits
VARIETY
Structured & unstructured
Online images & videos
Human generated - texts
Machine generated - readings

Youtube: TECHLAKE



A DAY IN DATA

The exponential growth of data is undisputed, but the numbers behind this explosion - fuelled by internet of things and the use of connected devcies - are hard to comprehend, particularly when looked at in the context of one day

DEMYSTIFIYING DATA UNITS

From the more familiar 'bit' or 'megabyte', larger units of measurement are more frequently being used to explain the masses of data

Unit		Value	Size
b	bit	0 or 1	1/8 of a byte
B	byte	8 bits	1 byte
KB	kilobyte	1,000 bytes	1,000 bytes
MB	megabyte	1,000² bytes	1,000,000 bytes
GB	gigabyte	1,000³ bytes	1,000,000,000 bytes
TB	terabyte	1,000⁴ bytes	1,000,000,000,000 bytes
PB	petabyte	1,000⁵ bytes	1,000,000,000,000,000 bytes
EB	exabyte	1,000⁶ bytes	1,000,000,000,000,000,000 bytes
ZB	zettabyte	1,000⁷ bytes	1,000,000,000,000,000,000,000 bytes
YB	yottabyte	1,000⁸ bytes	1,000,000,000,000,000,000,000,000 bytes

*A lowercase "b" is used as an abbreviation for bits, while an uppercase "B" represents bytes.

463EB

of data will be created every day by 2025

IDC

500m

tweets are sent every day

Twitter

4PB

of data created by Facebook, including

350m photos

100m hours of video watch time

Facebook Research

95m

photos and videos are shared on Instagram

Instagram Business

65bn

messages sent over WhatsApp and two billion minutes of voice and video calls made

Facebook

320bn

emails to be sent each day by 2021

294bn

billion emails are sent

Radicati Group

306bn

emails to be sent each day by 2020

3.9bn

people use emails

4TB

of data produced by a connected car

Intel

Searches made a day — **5bn**

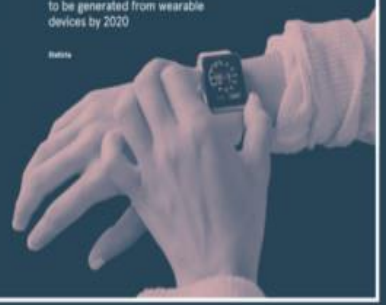
Searches made a day from Google — **3.5bn**

Smart Insights

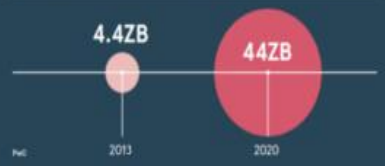
28PB

to be generated from wearable devices by 2020

Statista



ACCUMULATED DIGITAL UNIVERSE OF DATA



IDC

RACONTEUR

The six Vs of big data

Big data is a collection of data from various sources, often characterized by what's become known as the 3Vs: *volume*, *variety* and *velocity*. Over time, other Vs have been added to descriptions of big data:

VOLUME	VARIETY	VELOCITY	VERACITY	VALUE	VARIABILITY
The amount of data from myriad sources.	The types of data: structured, semi-structured, unstructured.	The speed at which big data is generated.	The degree to which big data can be trusted.	The business value of the data collected.	The ways in which the big data can be used and formatted.






Velocity + Volume + Variety + Veracity + Viability = Value

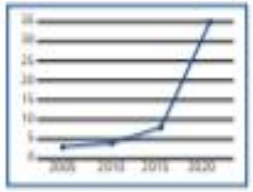
Velocity
Frequency of data generation

300 hours
of video uploaded to YouTube every minute (estimate)

2,400,000 search queries
per minute on Google

4,170,000 posts liked
on Facebook per minute

Volume
The growth of world data



1 terabyte holds the equivalent of roughly 210 single sided DVDs.

Variety
Structured and unstructured data - types of Big Data

Web and social media

Machine to machine

Big transaction data

Biometric

Human-generated

Veracity
Establishing trust in data

1 in 3
business leaders don't trust the information they use

Uncertainty
due to inconsistency, ambiguity, latency and approximation

Viability
Relevance and feasibility

"Can we use mobile phone data to monitor cross-border tourism?"

Hypothesis
Validation to determine if the data will have a meaningful impact

Long-term
rewards and better outcomes from hidden relationships in data

Value
Return on investment

Costs
Risk of simply creating Big Costs without creating the value

Insights
Sophisticated queries, counterintuitive insights and unique learning

Operational Efficiency

Use data to:

- Increase level of transparency
- Optimize resource consumption
- Improve process quality and performance

Customer Experience

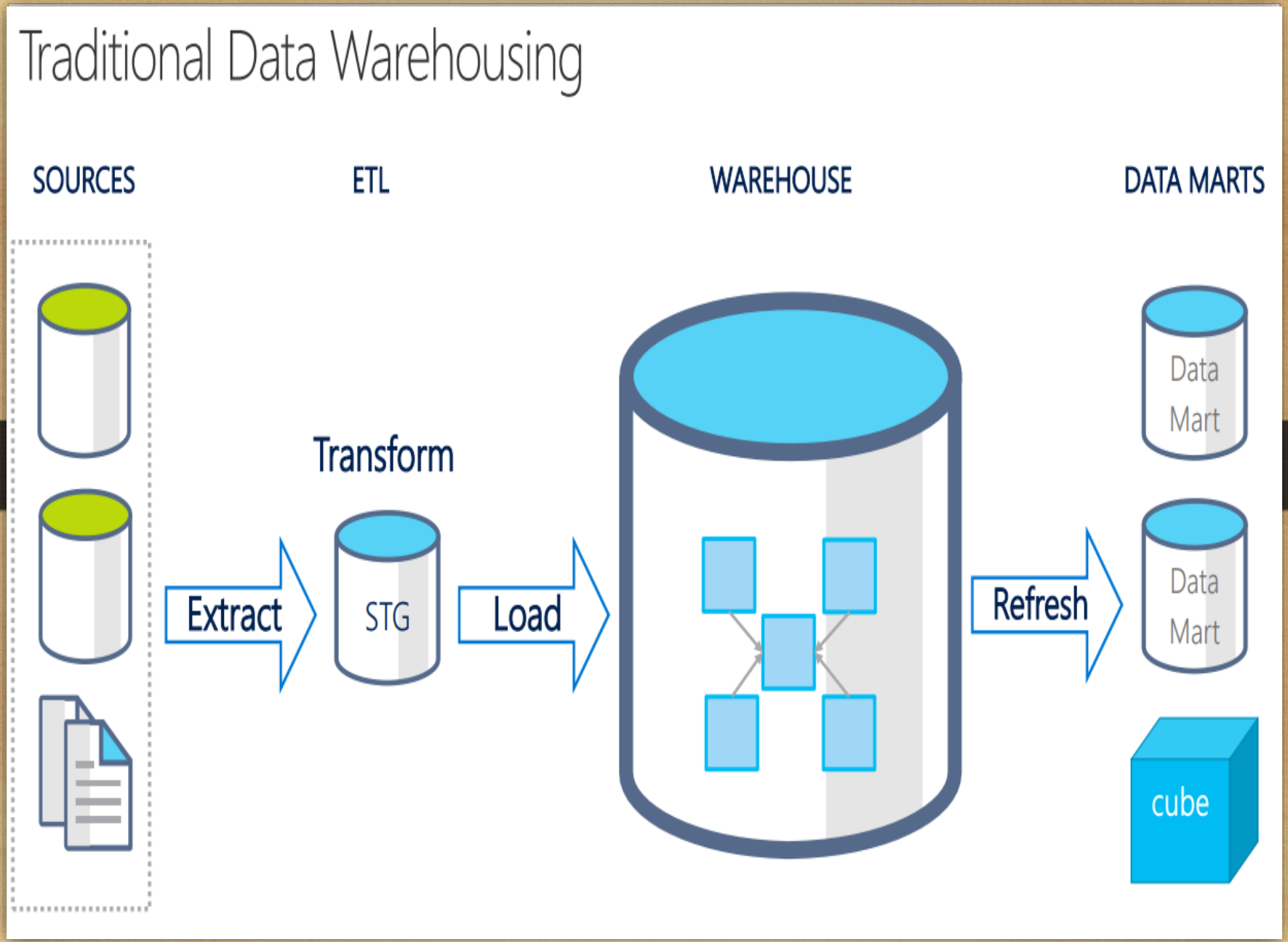
Exploit data to:

- Increase customer loyalty and retention
- Perform precise customer segmentation and targeting
- Optimize customer interaction and service

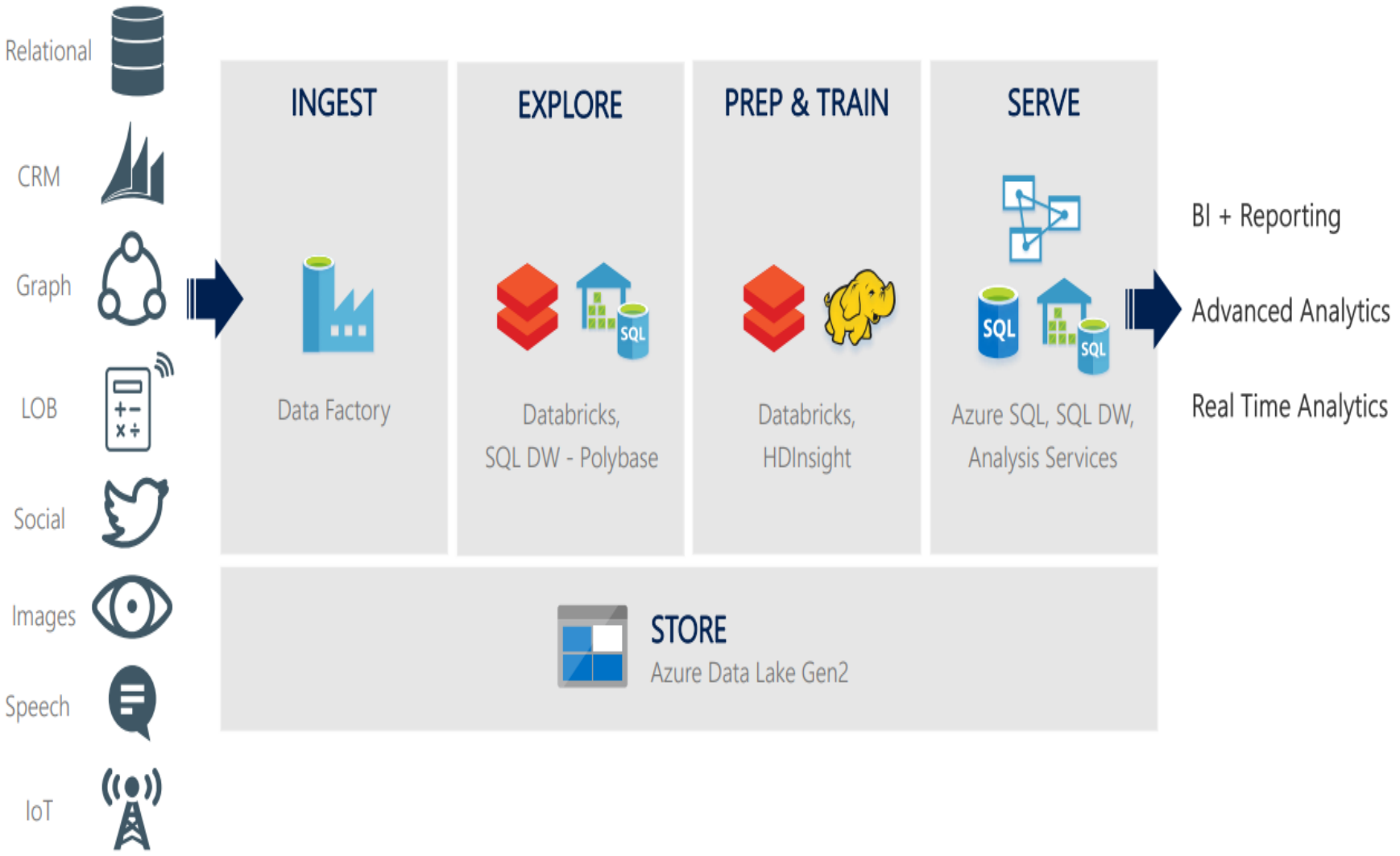
New Business Models

Capitalize on data by:

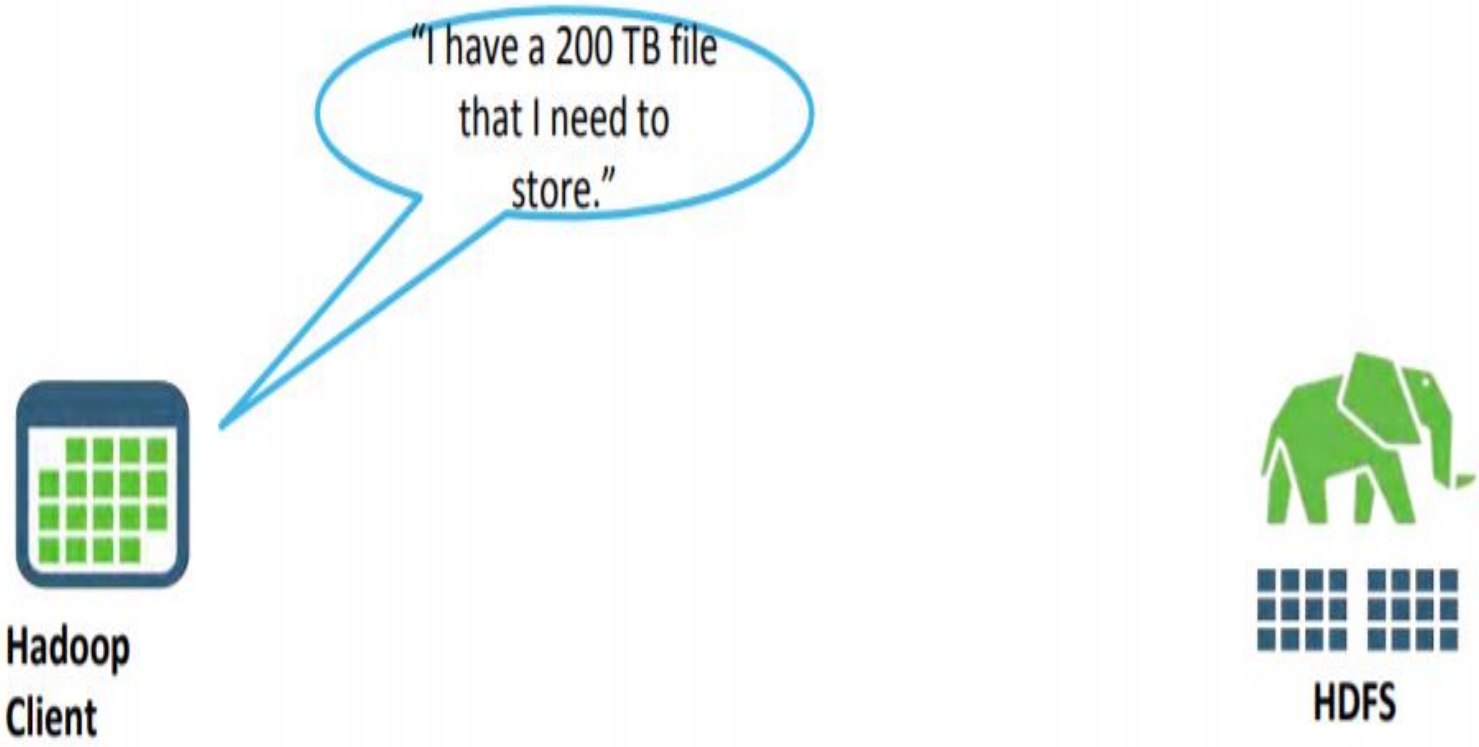
- Expanding revenue streams from existing products
- Creating new revenue streams from entirely new (data) products


Traditional Data Warehousing
SOURCES
ETL
WAREHOUSE
DATA MARTS
Transform
Extract
STG
Load
Refresh
Data Mart
Data Mart
cube

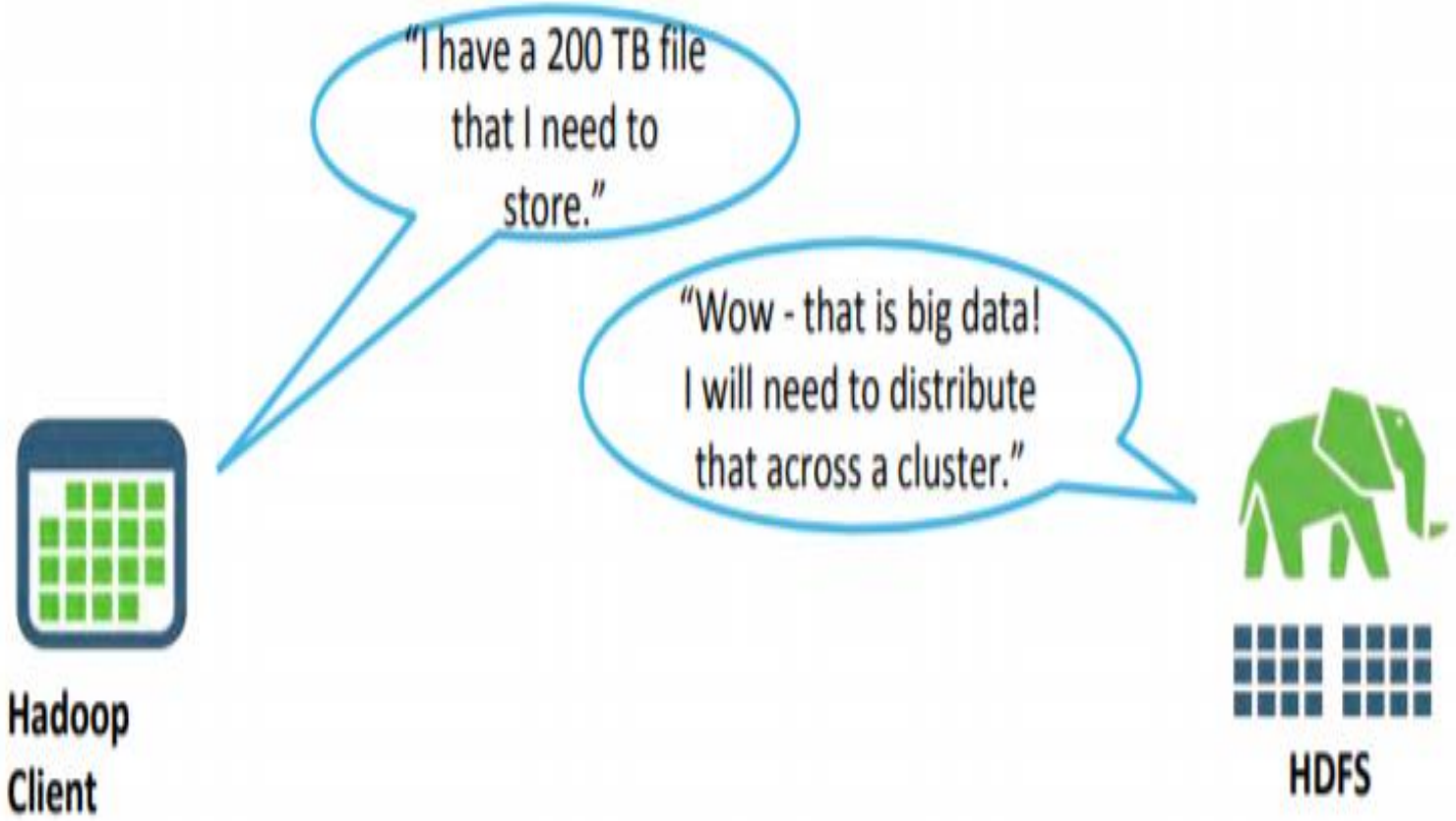
Modern Data Warehouse on Azure



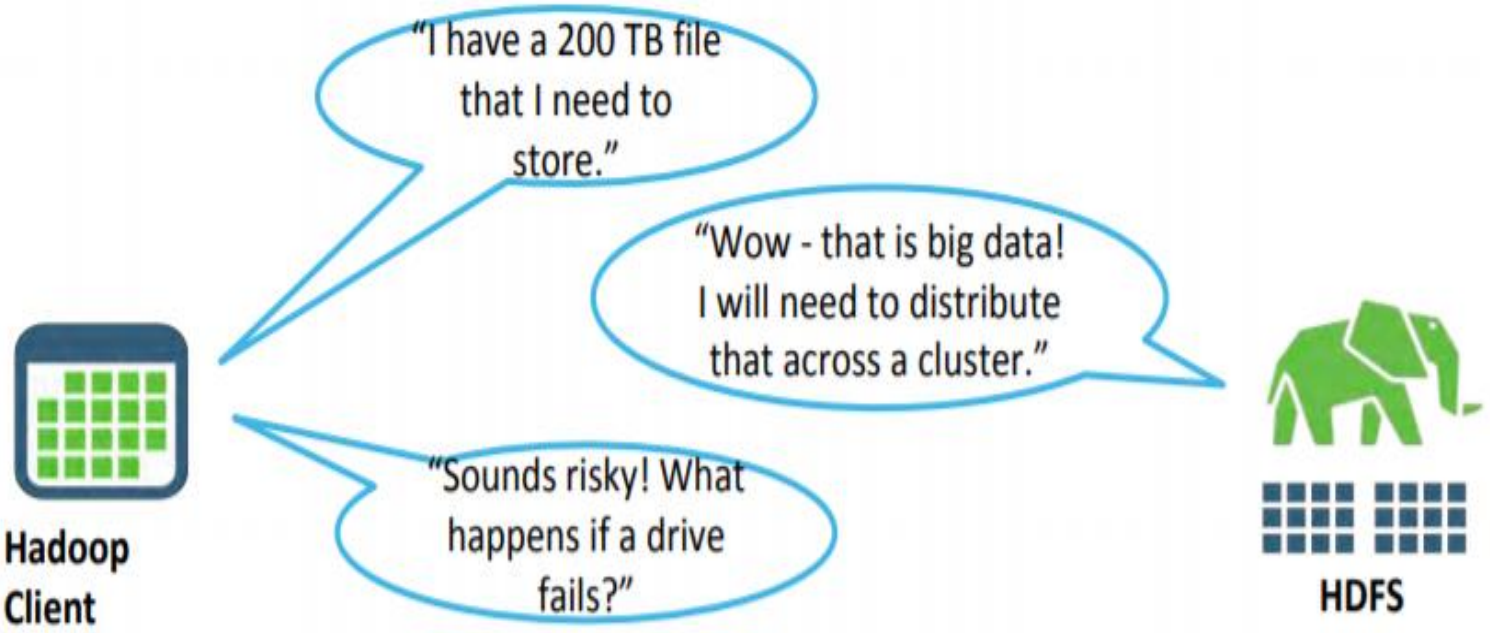
What is HDFS?



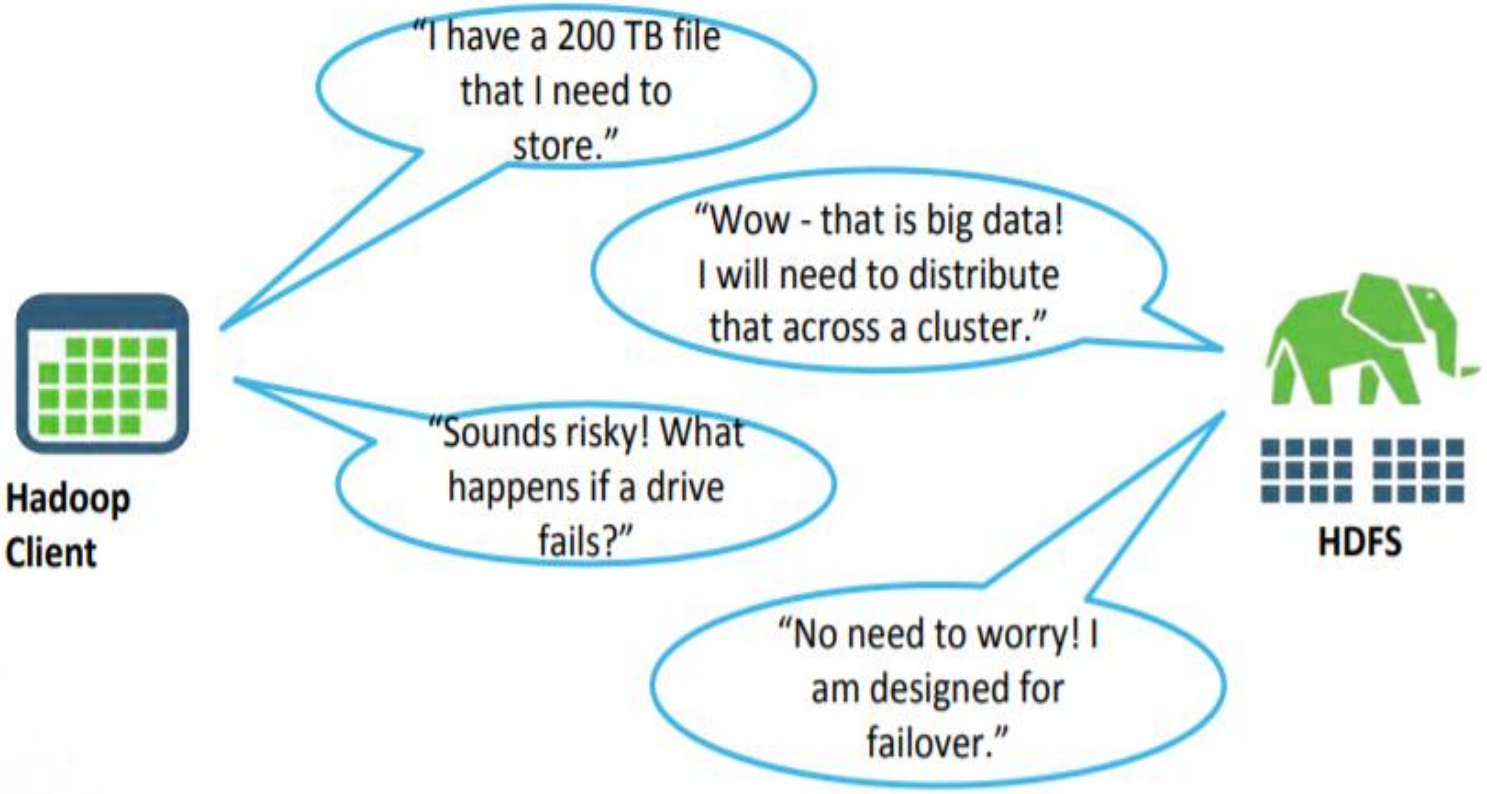
What is HDFS?


"I have a 200 TB file that I need to store."
"Wow - that is big data! I will need to distribute that across a cluster."
Hadoop Client
HDFS

What is HDFS?


"I have a 200 TB file that I need to store."
"Wow - that is big data! I will need to distribute that across a cluster."
"Sounds risky! What happens if a drive fails?"
Hadoop Client
HDFS

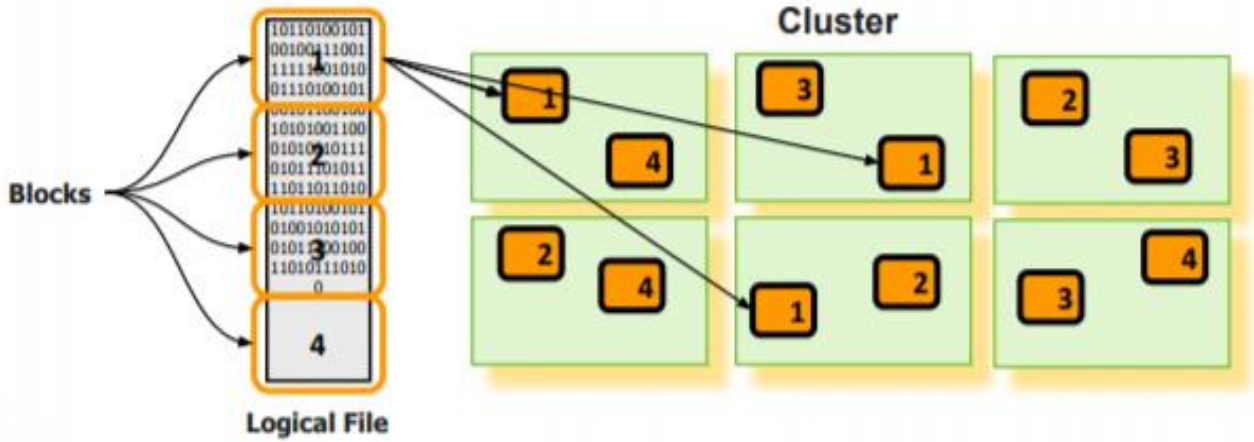
What is HDFS?


"I have a 200 TB file that I need to store."
"Wow - that is big data! I will need to distribute that across a cluster."
"Sounds risky! What happens if a drive fails?"
"No need to worry! I am designed for failover."
Hadoop Client
HDFS

HDFS

Key Ideas

- Write Once, Read Many times (WORM)
- Divide files into big blocks and distribute across the cluster
- Store multiple replicas of each block for reliability
- Programs can ask "where do the pieces of my file live?"

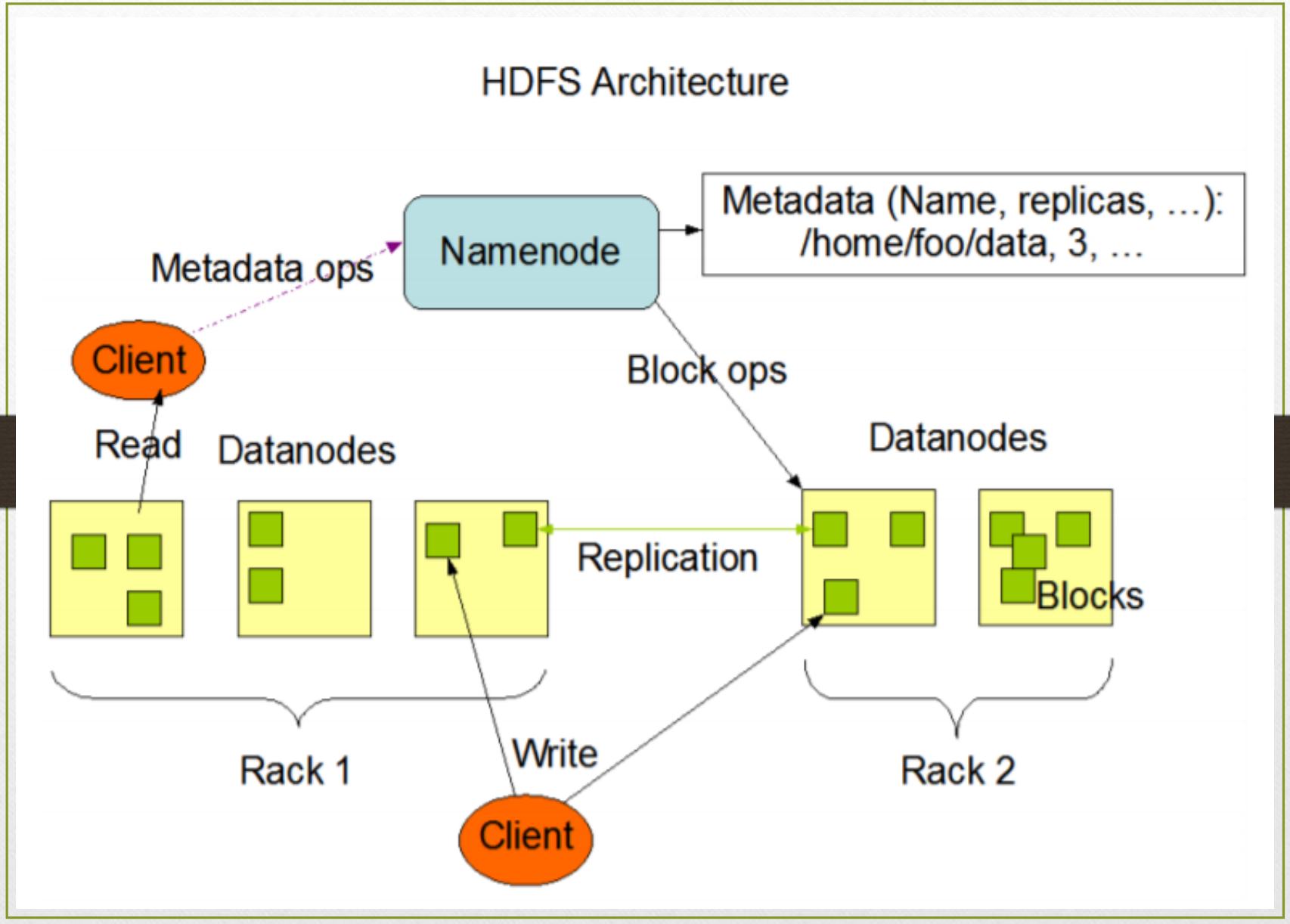


History of Hadoop

In 1990's Google had to come up with more data and to get the proper solution it has taken 13 years. In 2003, they had introduced GFS (Google File System) which is a technique to store huge data. In 2004, they have introduced MapReduce which is the best processing technique. They have published a "white paper" which has a description of GFS and MapReduce. Later, Yahoo which is the next best search engine after Google introduced HDFS in the year 2006-2007 and MapReduce was introduced in 2007-2008. They have taken the white paper which was given by Google and started implementing and came up with HDFS (Hadoop Distributed File System) and MapReduce. These are the two core components of Hadoop. Hadoop was then introduced by Doug Cutting in 2005.

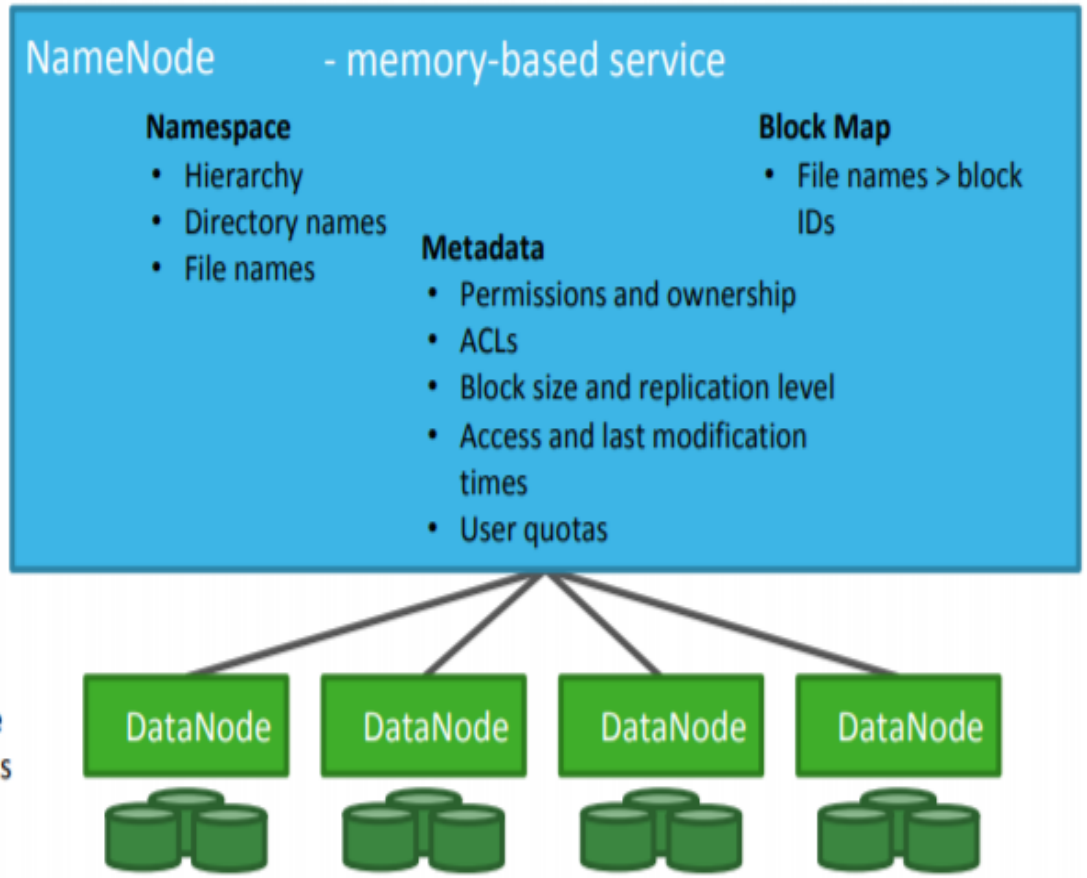
Map/Reduce History

- **Invented by Google:**
  - Inspired by functional programming languages map and reduce functions
  - Seminal paper: Dean, Jeffrey & Ghemawat, Sanjay (OSDI 2004), "MapReduce: Simplified Data Processing on Large Clusters"
  - Used at Google to completely regenerate Google's index of the World Wide Web.
  - It replaced the old ad-hoc programs that updated the index and ran the various analysis
- **Uses:**
  - distributed pattern-based searching, distributed sorting, web link-graph reversal, term-vector per host, web access log stats, inverted index construction, document clustering, machine learning, statistical machine translation
- **Apache Hadoop:** Open source implementation matches Google's specifications
- **Amazon Elastic MapReduce** running on Amazon EC2


HDFS Architecture
Namenode
Metadata (Name, replicas, …):
/home/foo/data, 3, …
Metadata ops
Client
Block ops
Read
Datanodes
Datanodes
Replication
Blocks
Rack 1
Write
Rack 2
Client

HDFS Architecture

- The NameNode (master node) and DataNodes (worker nodes) are daemons running in a Java virtual machine.



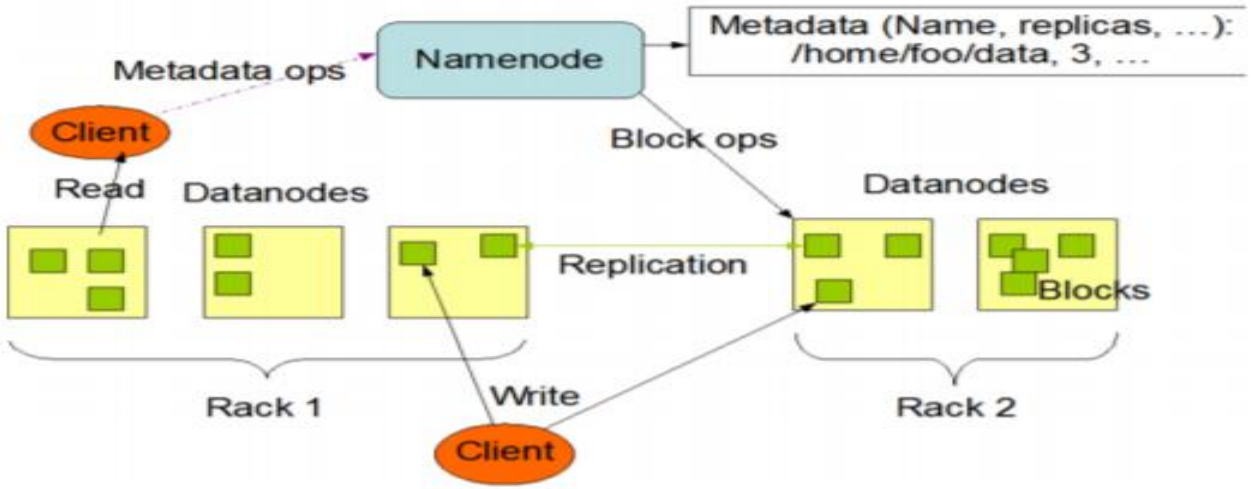
Working of HDFS

- Suppose a client is willing to put 150MB of data in a cluster and sends a request to the NameNode cluster as metadata. Metadata stores the data about the data given by the Client.
- 150MB of data is stored in a file with the file name as file.txt as shown in Figure2.
- The file is divided into 3 input splits a.txt, b.txt, c.txt of each 64MB block size (150MB / 64MB).

Figure 2. File.txt input splits

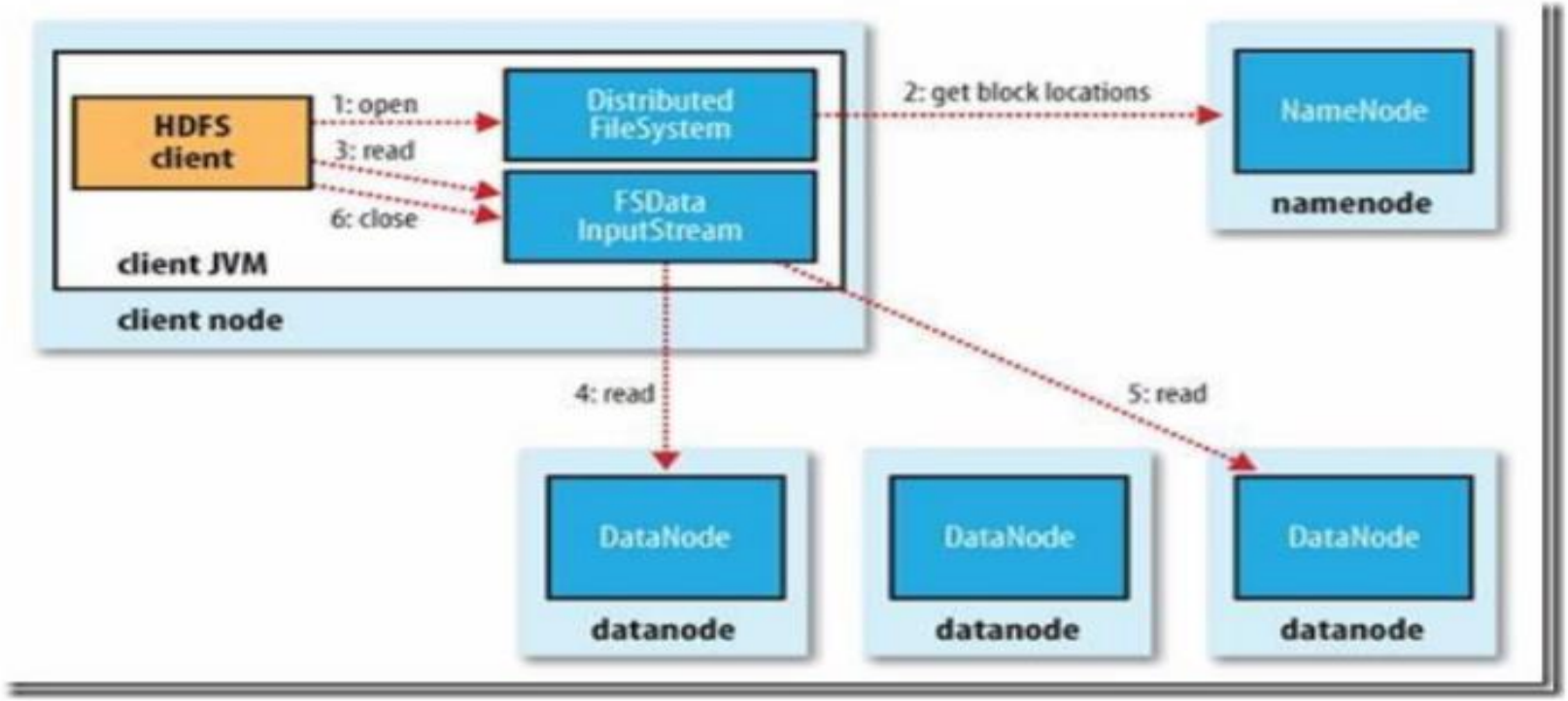
- NameNode responds to the client and requests to store 150MB data in the nodes which has space.
- Client store all the txt files in different DataNodes. However, all the files need not be in sequence order.

- DataNodes are commodity hardware which means if the system goes down the data doesn't lose since HDFS has been given 3 replications by default. Hence it has 2 more backup files for each text files stored in different DataNodes. Hence, the a.txt file occupies 450 MB (150 MB * 3) of files in the whole cluster because of the replication. The same way other text files are also allocated to DataNodes with their corresponding replications. All the DataNodes which are SlaveNodes for that NameNode give proper block report and heartbeat to the NameNode. This acknowledgment gives the information of the condition of the DataNodes. Block report shows the DataNodes are still allocated with some size of block and heartbeat gives the status of the nodes. This is how the data is stored in HDFS.

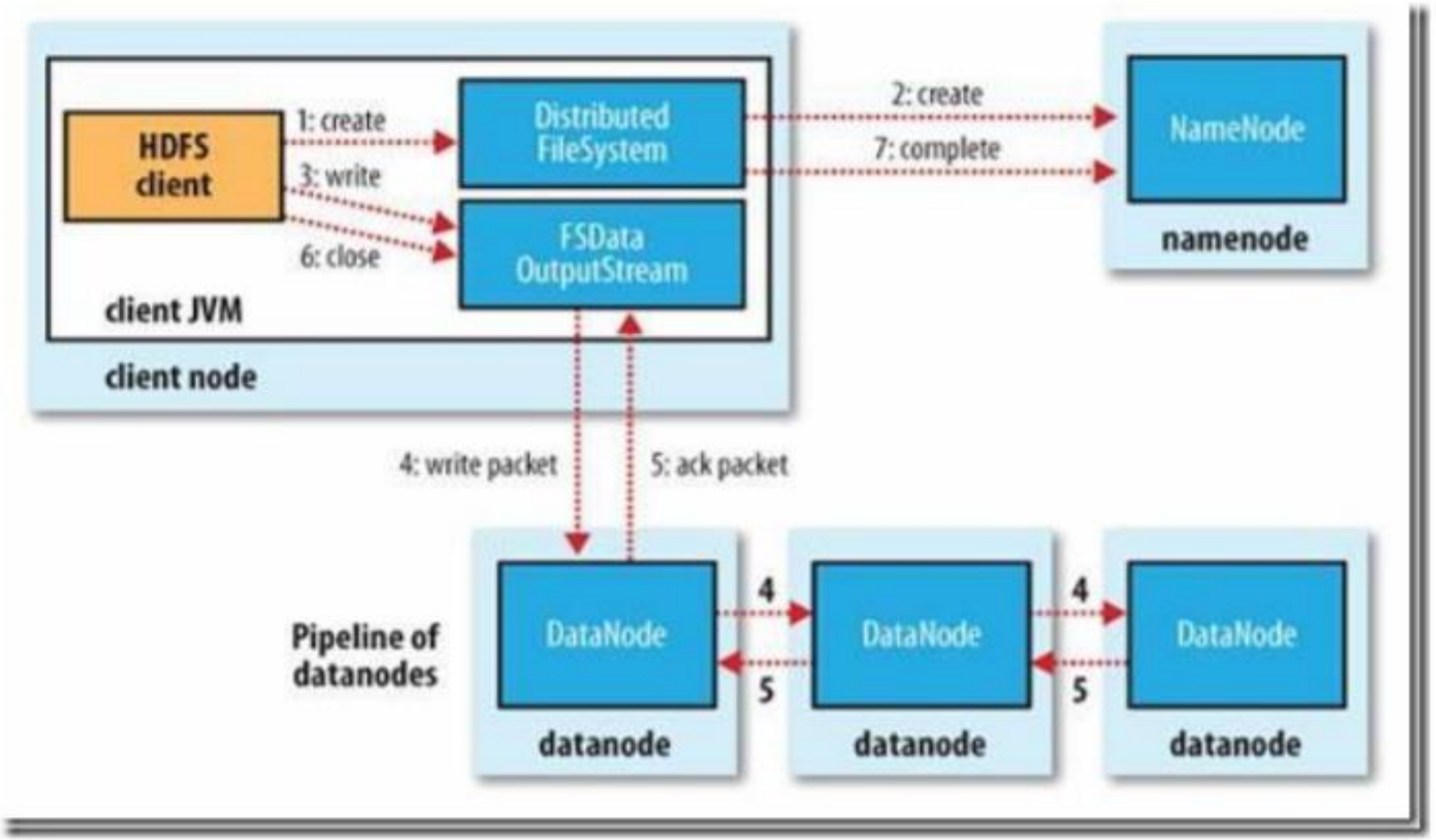


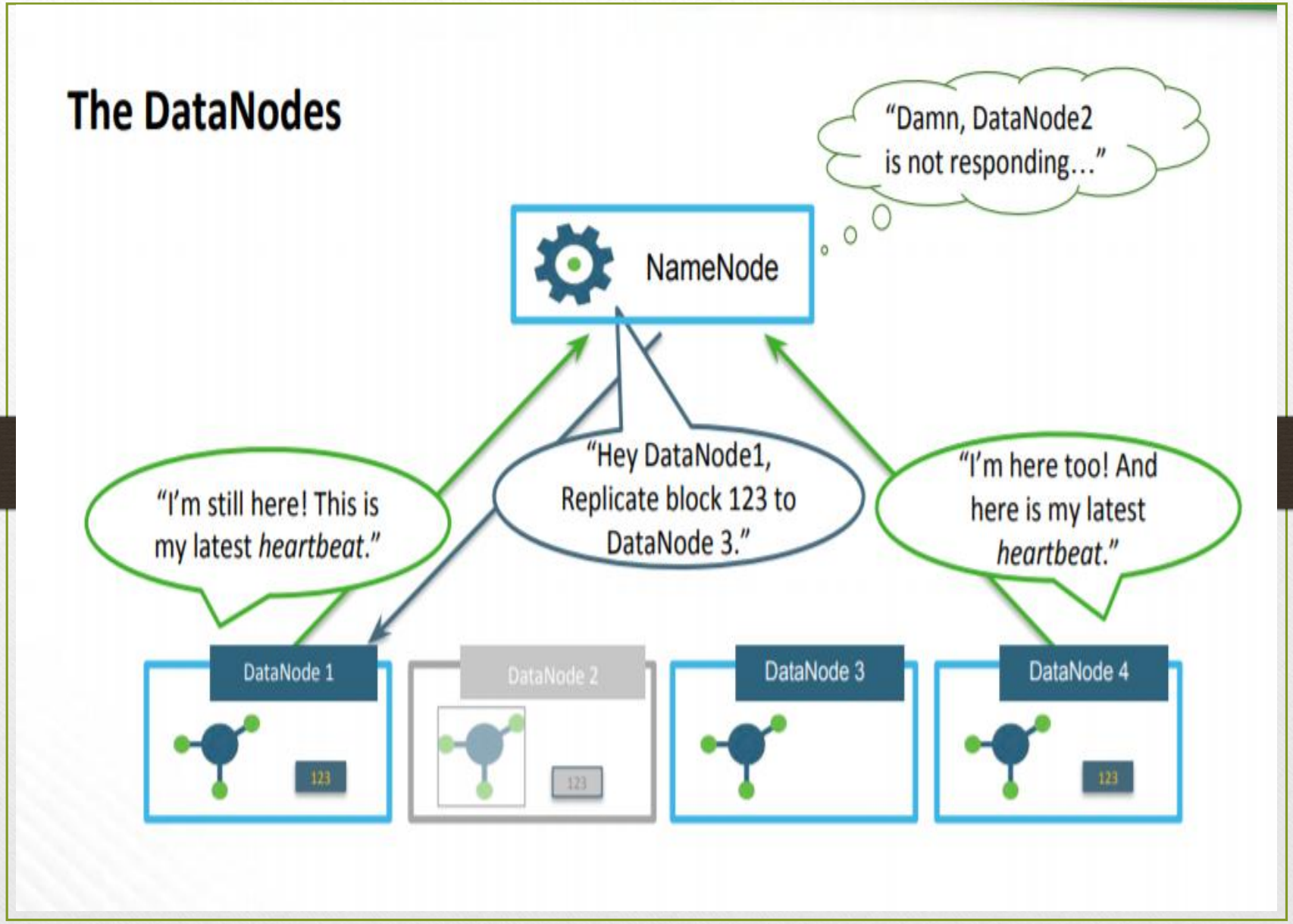
Data Reading Process in HDFS

The data reading process in HDFS is not difficult. It is similar to the programming logic which has created the object, i.e., calling the method and pergorming the execution. The following section will introduce the reading processing of the HDFS.

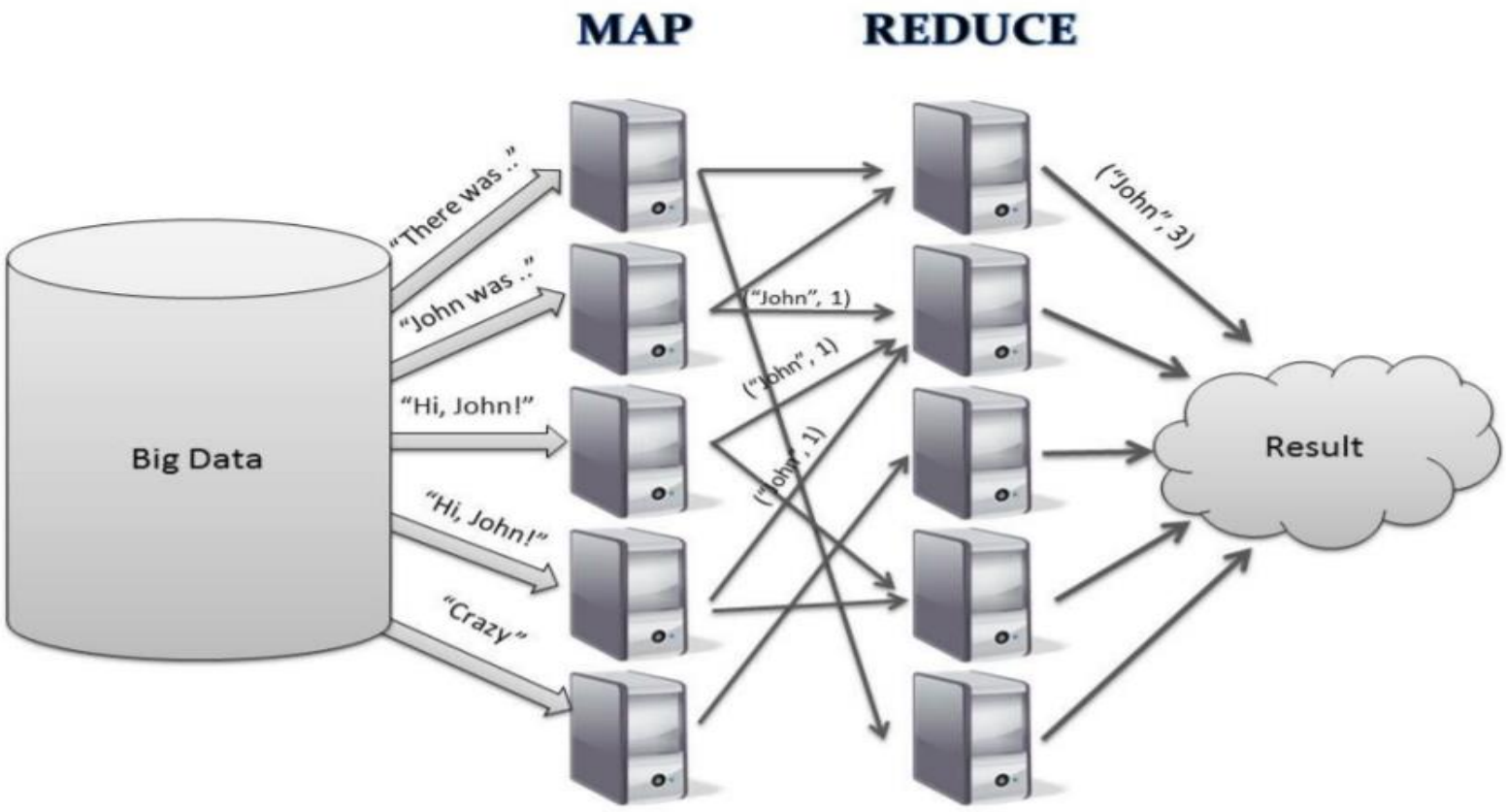


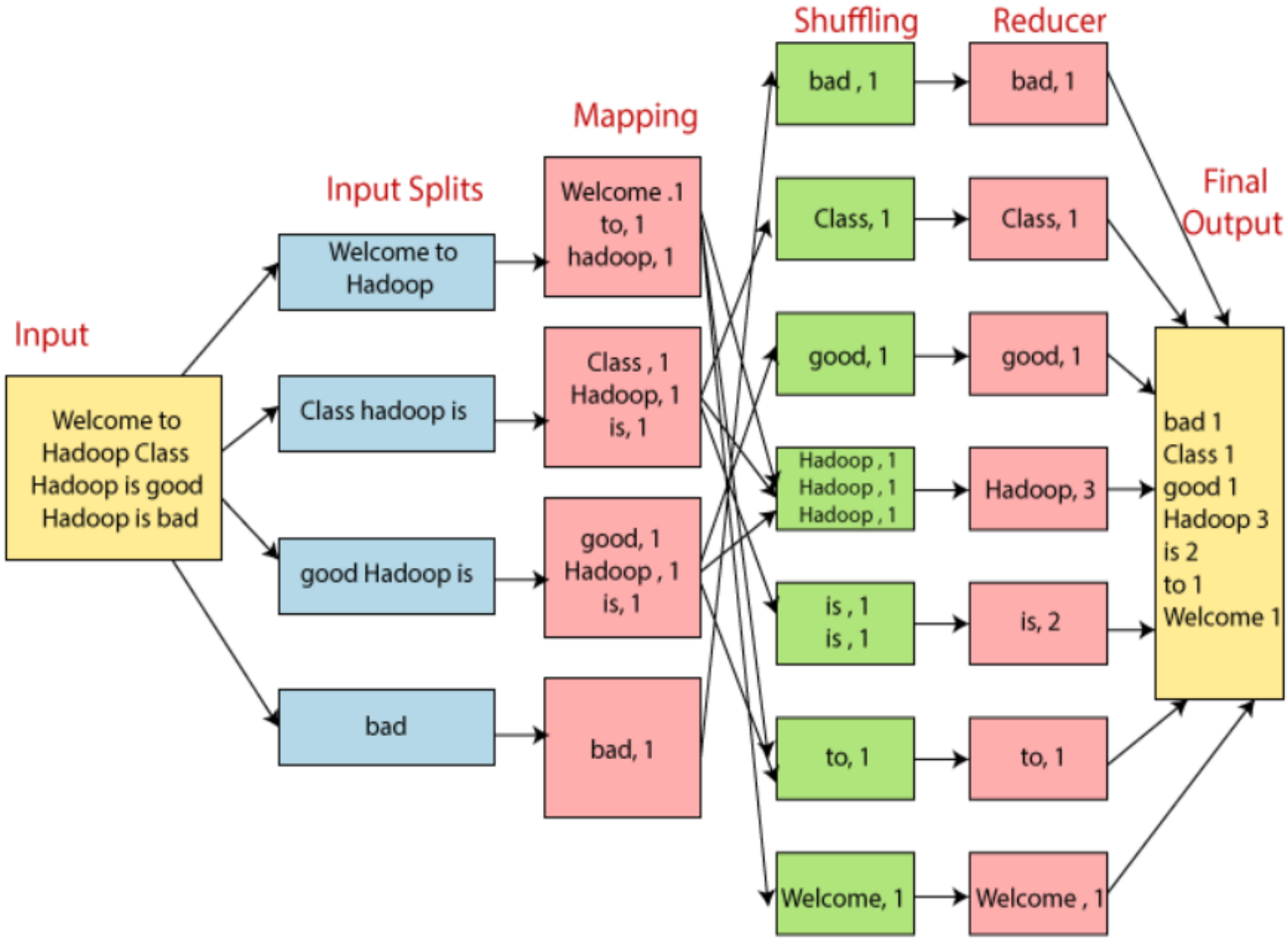
The structure of HDFS reading process is similar to the writing process. There are the following seven steps:




The DataNodes
"Damn, DataNode2 is not responding…"
NameNode
"I'm still here! This is my latest heartbeat."
"Hey DataNode1, Replicate block 123 to DataNode 3."
"I'm here too! And here is my latest heartbeat."
DataNode 1
DataNode 2
DataNode 3
DataNode 4
123
123
123

Map/Reduce


MAP
REDUCE
Big Data
"There was .."
"John was .."
"Hi, John!"
"Hi, John!"
"Crazy"
("John", 1)
("John", 1)
("John", 1)
("John", 3)
Result

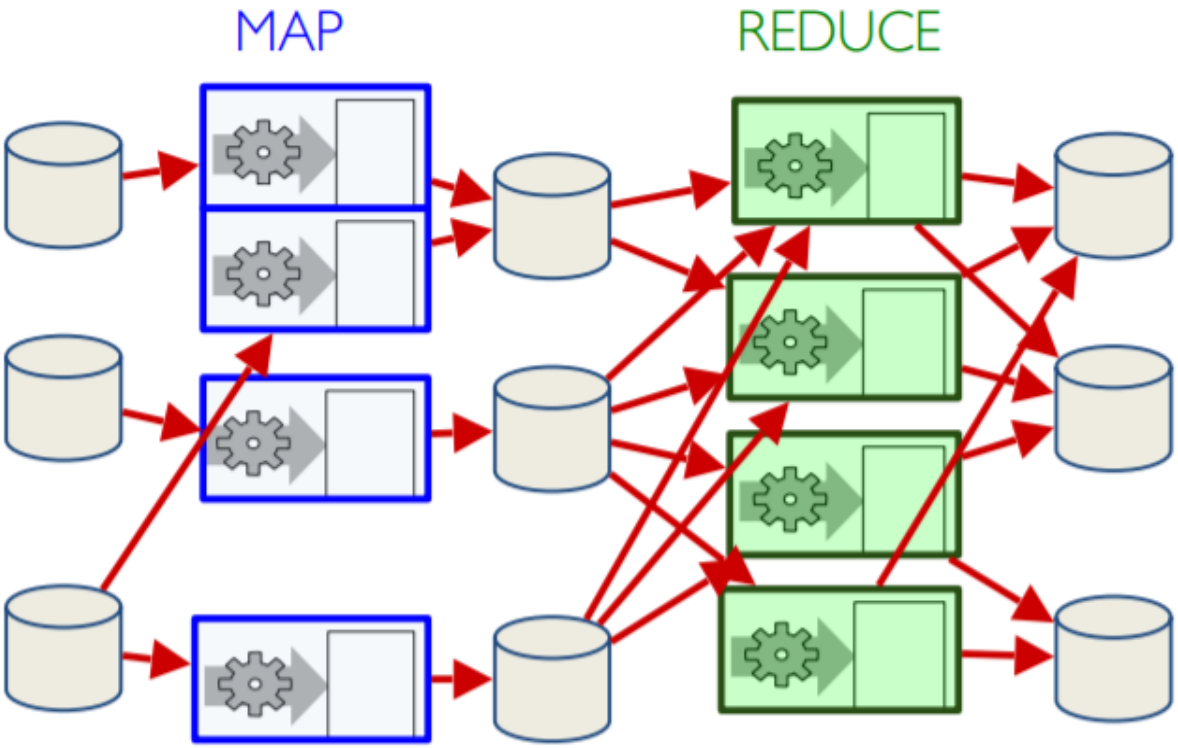

Input
Welcome to Hadoop Class Hadoop is good Hadoop is bad
Input Splits
Welcome to Hadoop
Class hadoop is
good Hadoop is
bad
Mapping
Welcome .1 to, 1 hadoop, 1
Class , 1 Hadoop, 1 is, 1
good, 1 Hadoop , 1 is, 1
bad, 1
Shuffling
bad , 1
Class, 1
good, 1
Hadoop , 1 Hadoop , 1 Hadoop , 1
is , 1 is , 1
to, 1
Welcome, 1
Reducer
bad, 1
Class, 1
good, 1
Hadoop, 3
is, 2
to, 1
Welcome , 1
Final Output
bad 1
Class 1
good 1
Hadoop 3
is 2
to 1
Welcome 1

Map/Reduce

- MapReduce is a framework for processing **parallelizable** problems across huge datasets using a large number of computers (nodes), collectively referred to as a **cluster**.
- **Map step:** Each worker node applies the "map()" function to the local data, and writes the output to a temporary storage. A master node orchestrates that for redundant copies of input data, only one is processed.
- **Shuffle step:** Worker nodes redistribute data based on the output keys (produced by the "map()" function), such that all data belonging to one key is located on the same worker node.
- **Reduce step:** Worker nodes now process each group of output data, per key, in parallel.

Why Spark?

MapReduce Execution



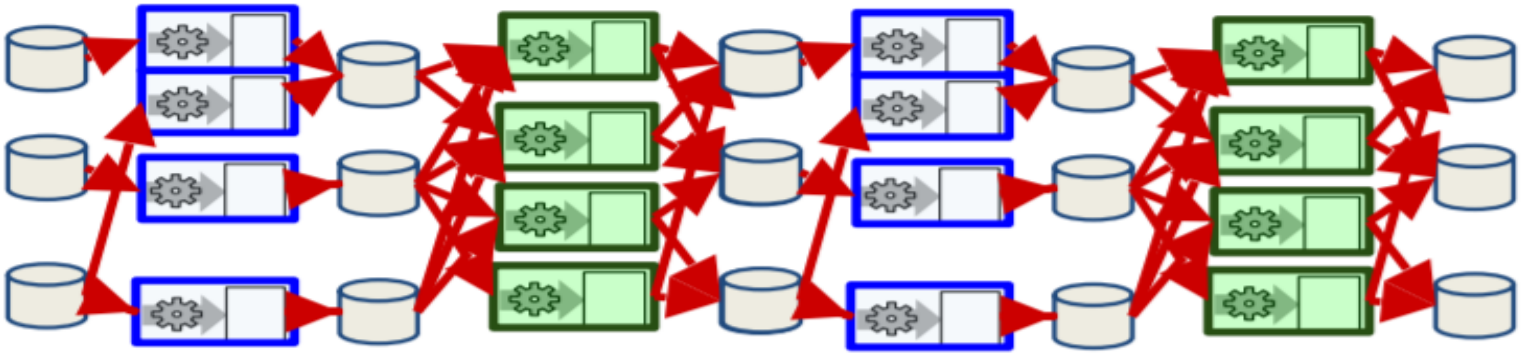
Each stage passes through the hard drives

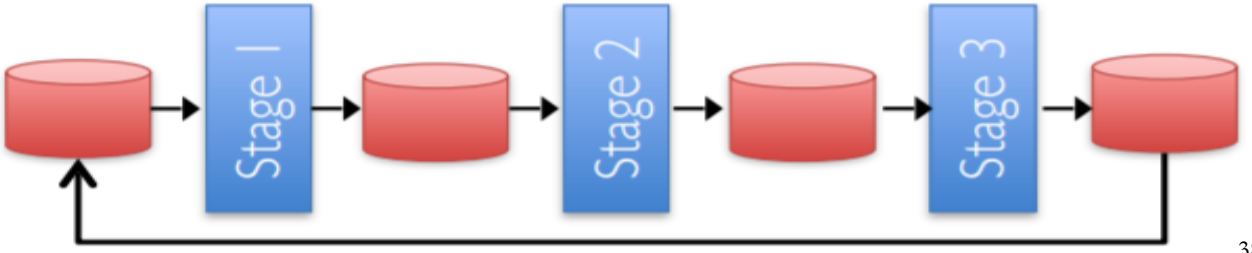
Why Spark?

Iterative Jobs

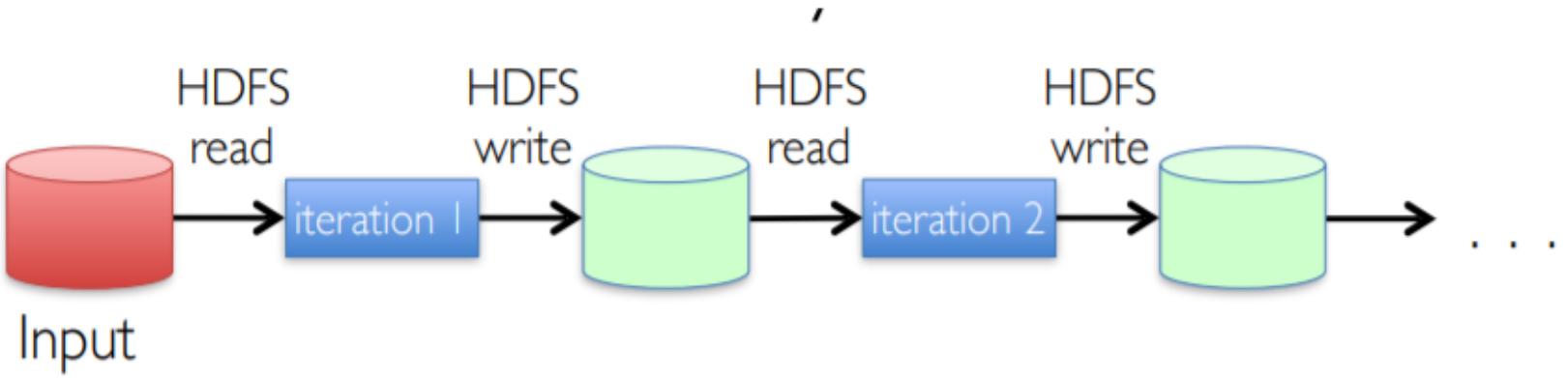
- Disk I/O for each repetition

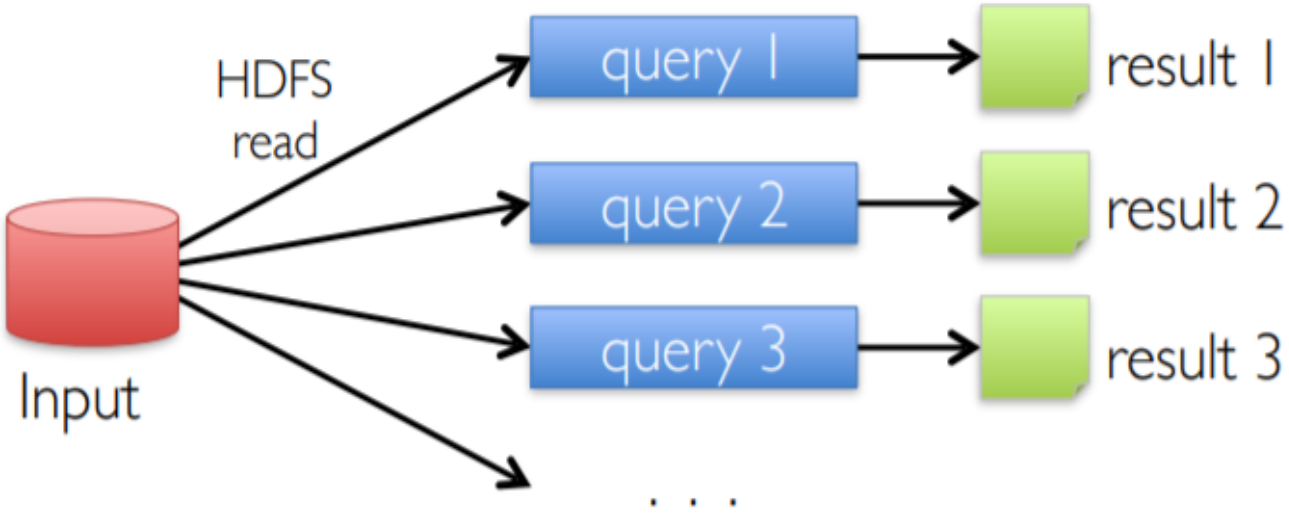
  → Slow when executing many small iterations





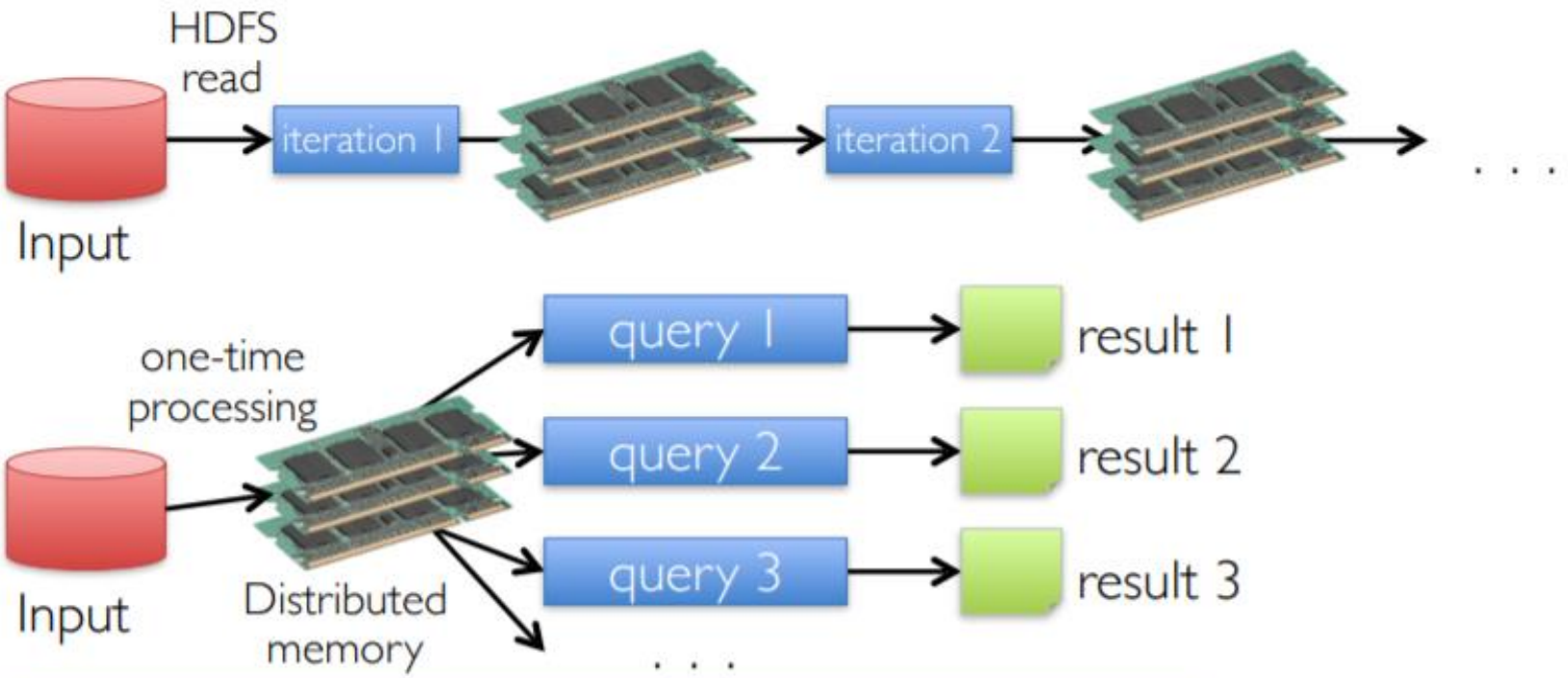
Replace Disk with Memory



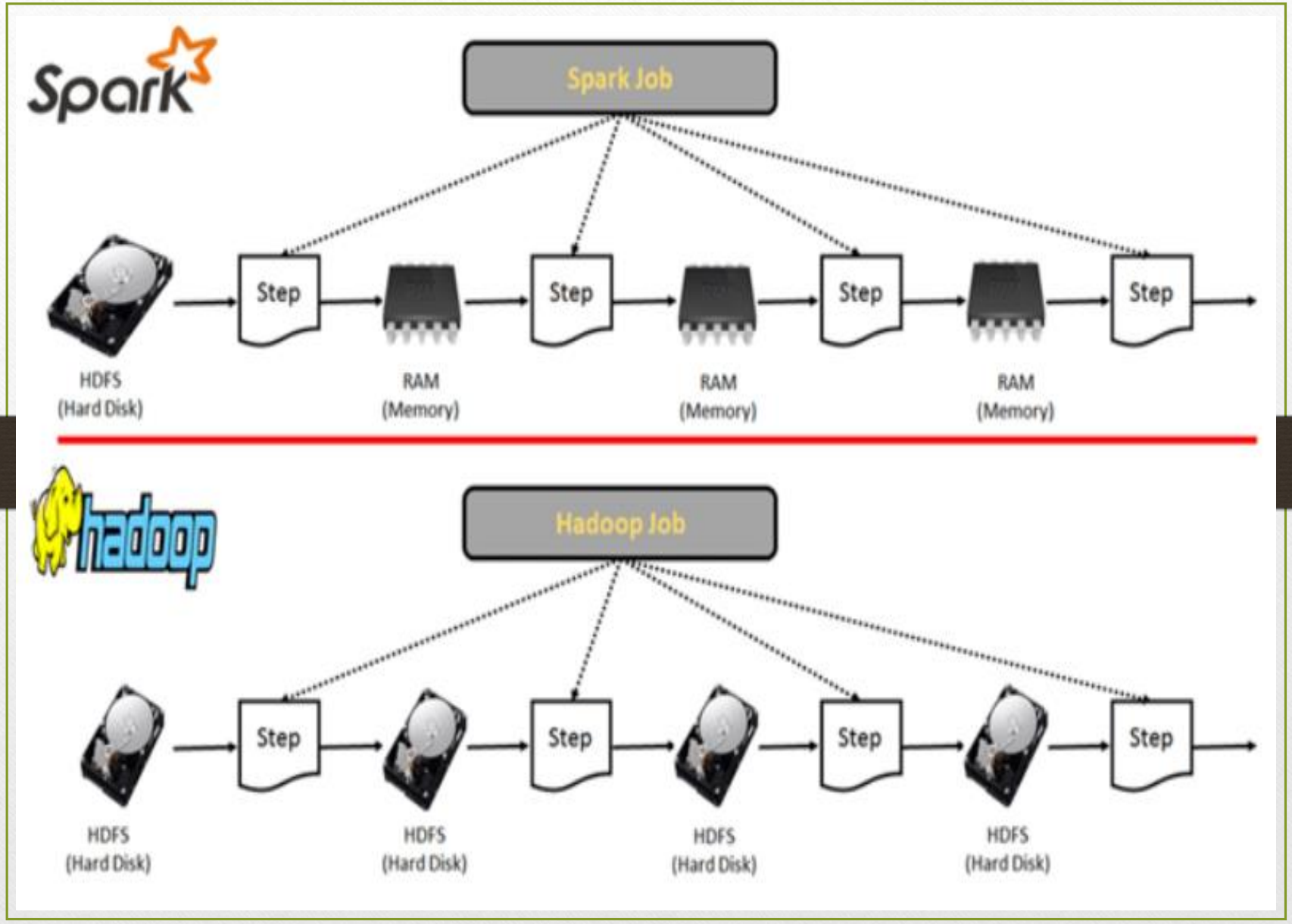


Replace Disk with Memory

In-Memory Data Sharing

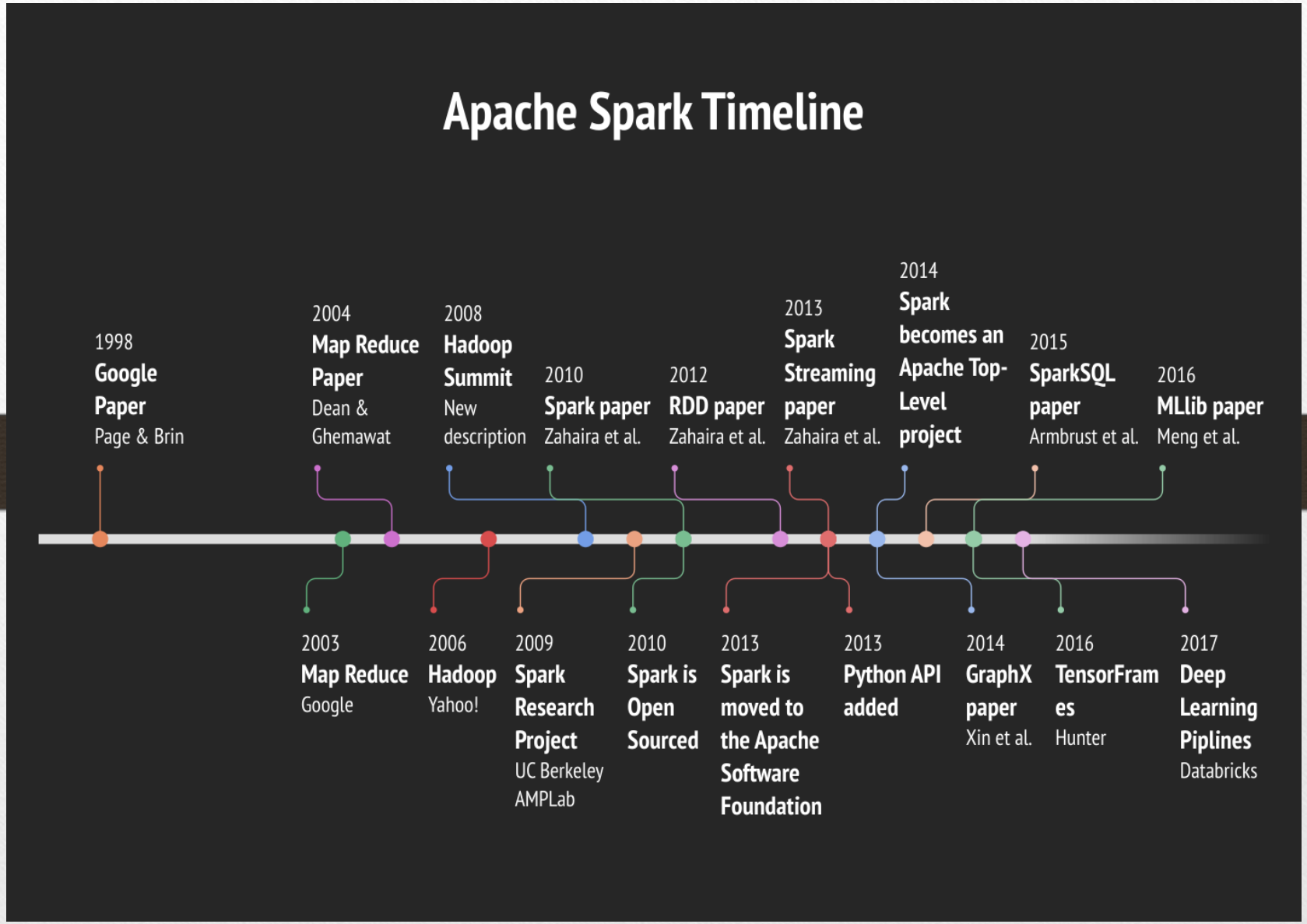


10-100x faster than network and disk


Spark
Spark Job
Step
Step
Step
Step
HDFS
(Hard Disk)
RAM
(Memory)
RAM
(Memory)
RAM
(Memory)
hadoop
Hadoop Job
Step
Step
Step
Step
HDFS
(Hard Disk)
HDFS
(Hard Disk)
HDFS
(Hard Disk)
HDFS
(Hard Disk)

World Record Data Process in 2014

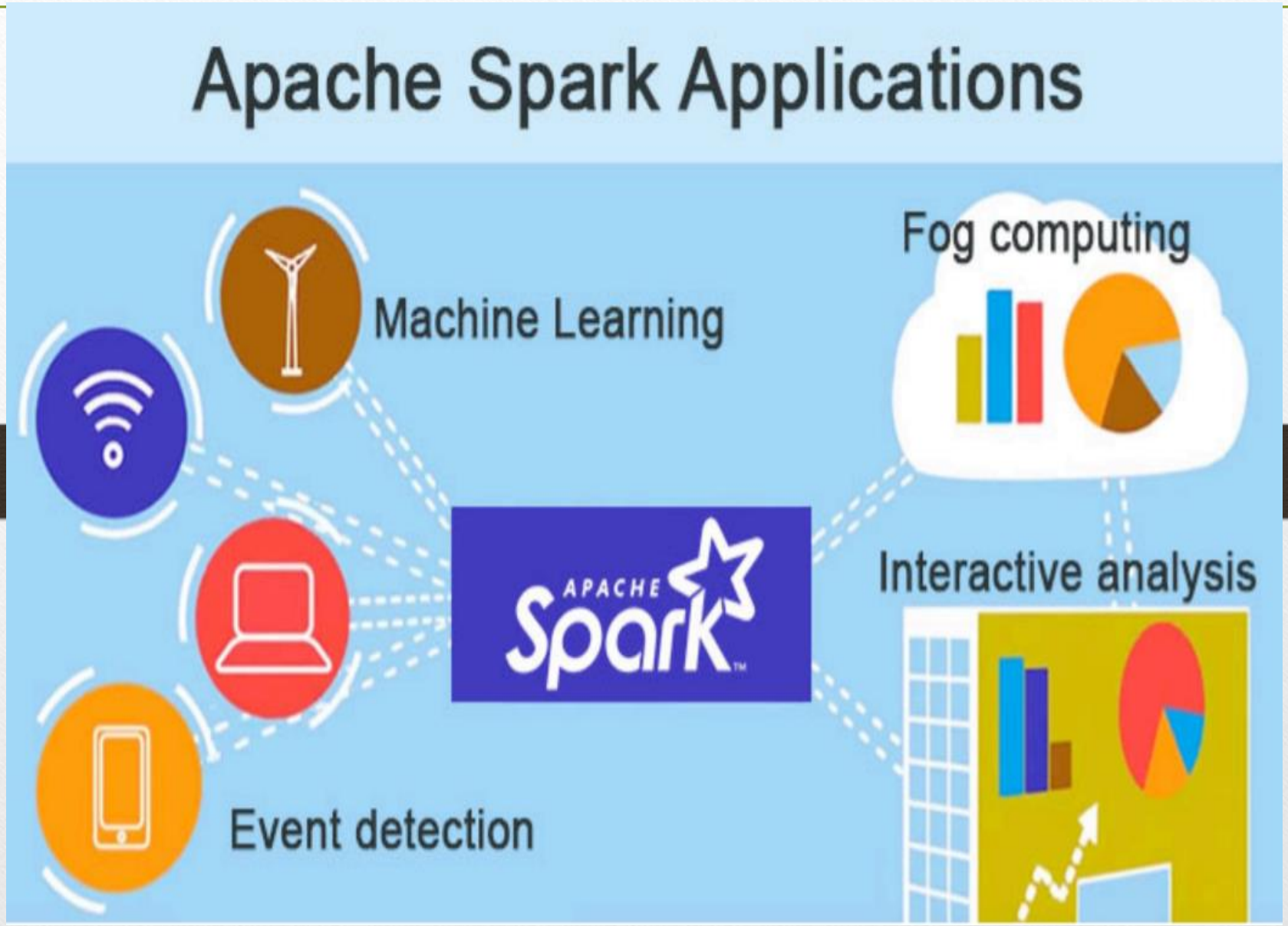
	Hadoop MR Record	Spark Record	Spark 1 PB
Data Size	102.5 TB	100 TB	1000 TB
Elapsed Time	72 mins	23 mins	234 mins
# Nodes	2100	206	190
# Cores	50400 physical	6592 virtualized	6080 virtualized
Cluster disk throughput	3150 GB/s (est.)	618 GB/s	570 GB/s
Sort Benchmark Daytona Rules	Yes	Yes	No
Network	dedicated data center, 10Gbps	virtualized (EC2) 10Gbps network	virtualized (EC2) 10Gbps network
Sort rate	**1.42 TB/min**	**4.27 TB/min**	**4.27 TB/min**
Sort rate/node	**0.67 GB/min**	**20.7 GB/min**	**22.5 GB/min**


Apache Spark Timeline
1998
Google Paper
Page & Brin
2003
Map Reduce
Google
2004
Map Reduce Paper
Dean & Ghemawat
2006
Hadoop
Yahoo!
2008
Hadoop Summit
New description
2009
Spark Research Project
UC Berkeley AMPLab
2010
Spark paper
Zahaira et al.
2010
Spark is Open Sourced
2012
RDD paper
Zahaira et al.
2013
Spark is moved to the Apache Software Foundation
2013
Spark Streaming paper
Zahaira et al.
2013
Python API added
2014
Spark becomes an Apache Top-Level project
2014
GraphX paper
Xin et al.
2015
SparkSQL paper
Armbrust et al.
2016
TensorFrames
Hunter
2016
MLlib paper
Meng et al.
2017
Deep Learning Piplines
Databricks

In-Memory Processing

- Many datasets fit in memory (of a cluster)
- Memory is fast and avoid disk I/O

→ Spark distributed execution engine


Apache Spark Applications
Machine Learning
Fog computing
APACHE
Spark
Interactive analysis
Event detection

Uses Cases

NETFLIX

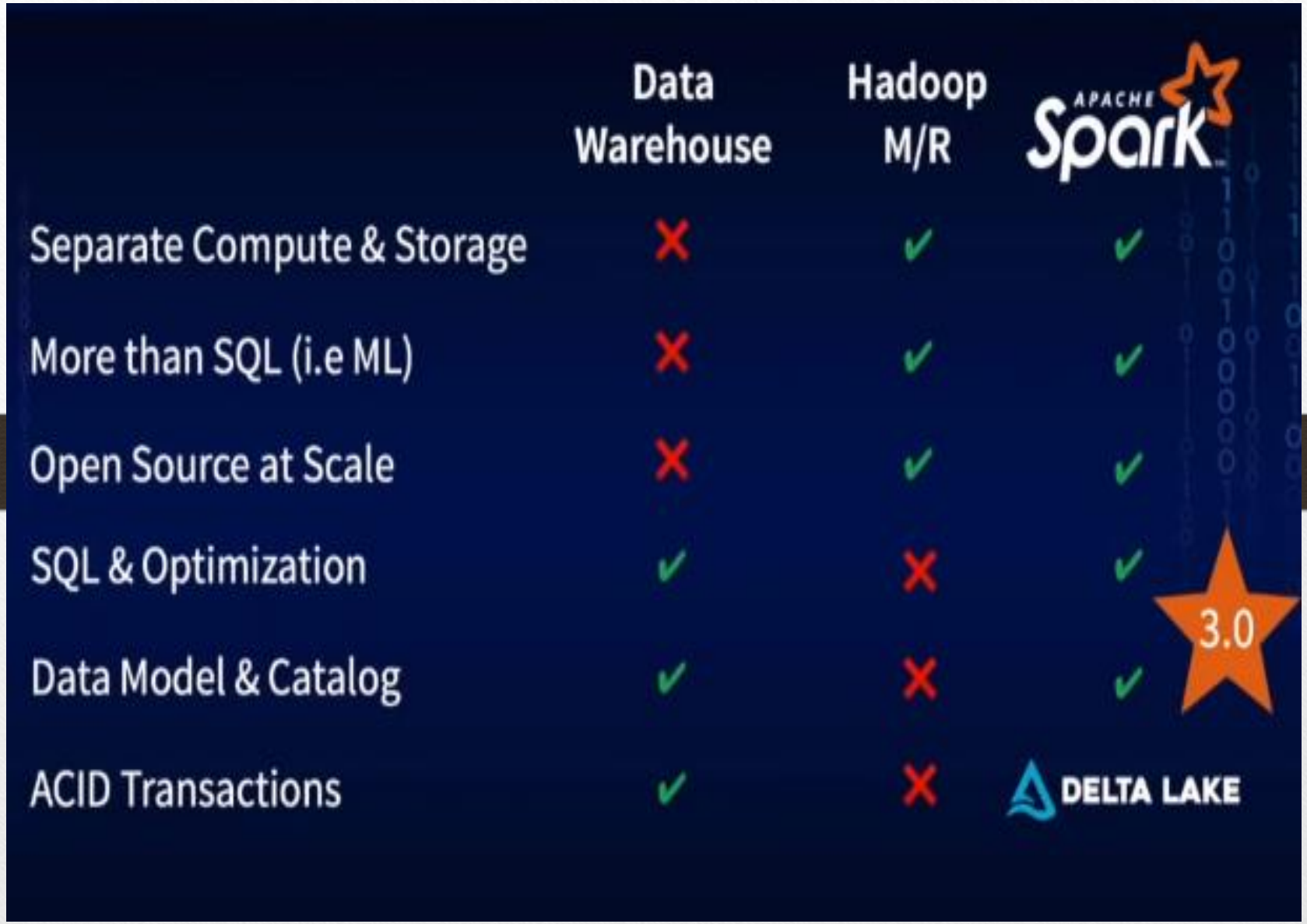
Uses Spark Streaming to provide the best-in-class movie streaming and recommendation tool to its users.

UBER

Uses Spark to collect TBs of raw and unstructured data every day from its users to convert it into structured data. This makes it ready for further complex analytics.

Pinterest

Feeds real-time data into Spark via Spark Streaming to get instant insights on how users are engaging with Pins globally. This makes Pinterest's recommendations (i.e. to show Pins) to be accurate.


Data Warehouse
Hadoop M/R
APACHE Spark
Separate Compute & Storage
More than SQL (i.e ML)
Open Source at Scale
SQL & Optimization
Data Model & Catalog
ACID Transactions
3.0
DELTA LAKE

HADOOP MAPREDUCE	SPARK
The data is stored in the disc.	The data is stored in-memory.
Computing is based on the disc.	Computing relies on RAM.
Fault tolerance is done through replication.	Fault tolerance is done through RDD.
Hard to work with real-time data.	Easy to work with real-time data.
Less costly in comparison to spark.	More costly.
For batch processing only.	Supports interactive query.

What is Apache Spark?

Apache Spark is a parallel processing framework and unified analytics engine that supports in-memory processing to boost the performance of big-data analytic and machine learning.

What is Apache Spark used for?

Spark is used for many types of data processing. It supports ETL, interactive queries (SQL), advanced analytics (e.g. machine learning) and structured streaming over large datasets.

Spark integrates with many storage systems (e.g. HDFS, Cassandra, MySQL, HBase, MongoDB, S3). Spark is also pluggable, with dozens of applications, data sources, and environments

Introduction to Apache Spark

- Fast, expressive cluster computing system compatible with Apache Hadoop
- It is much faster and much easier than Hadoop MapReduce to use due its rich APIs
- Large community
- Goes far beyond batch applications to support a variety of workloads:
  - including interactive queries, streaming, machine learning, and graph processing

Figure: *Real Time Processing In Spark*

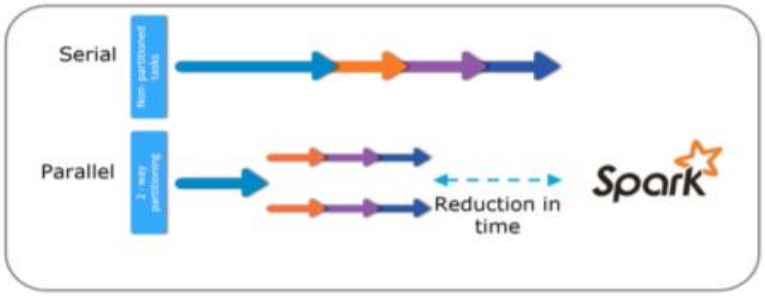
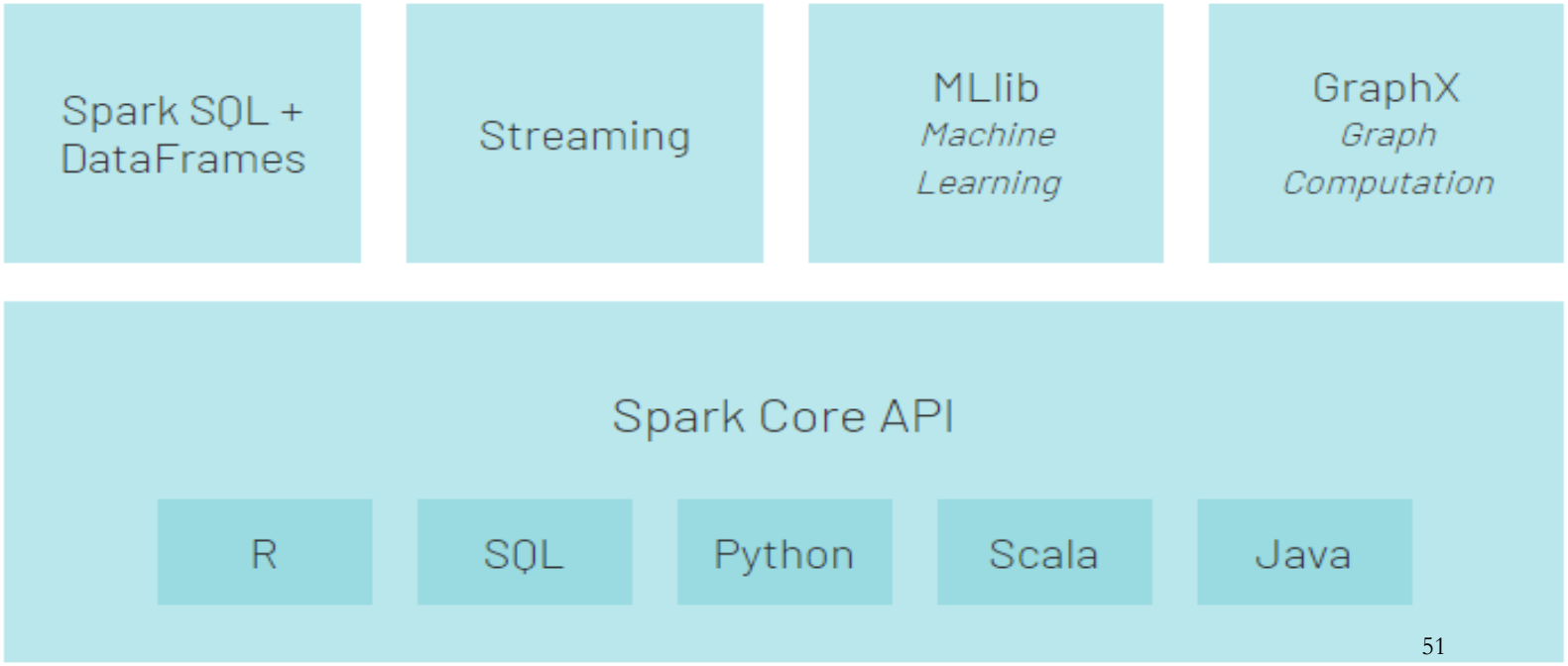
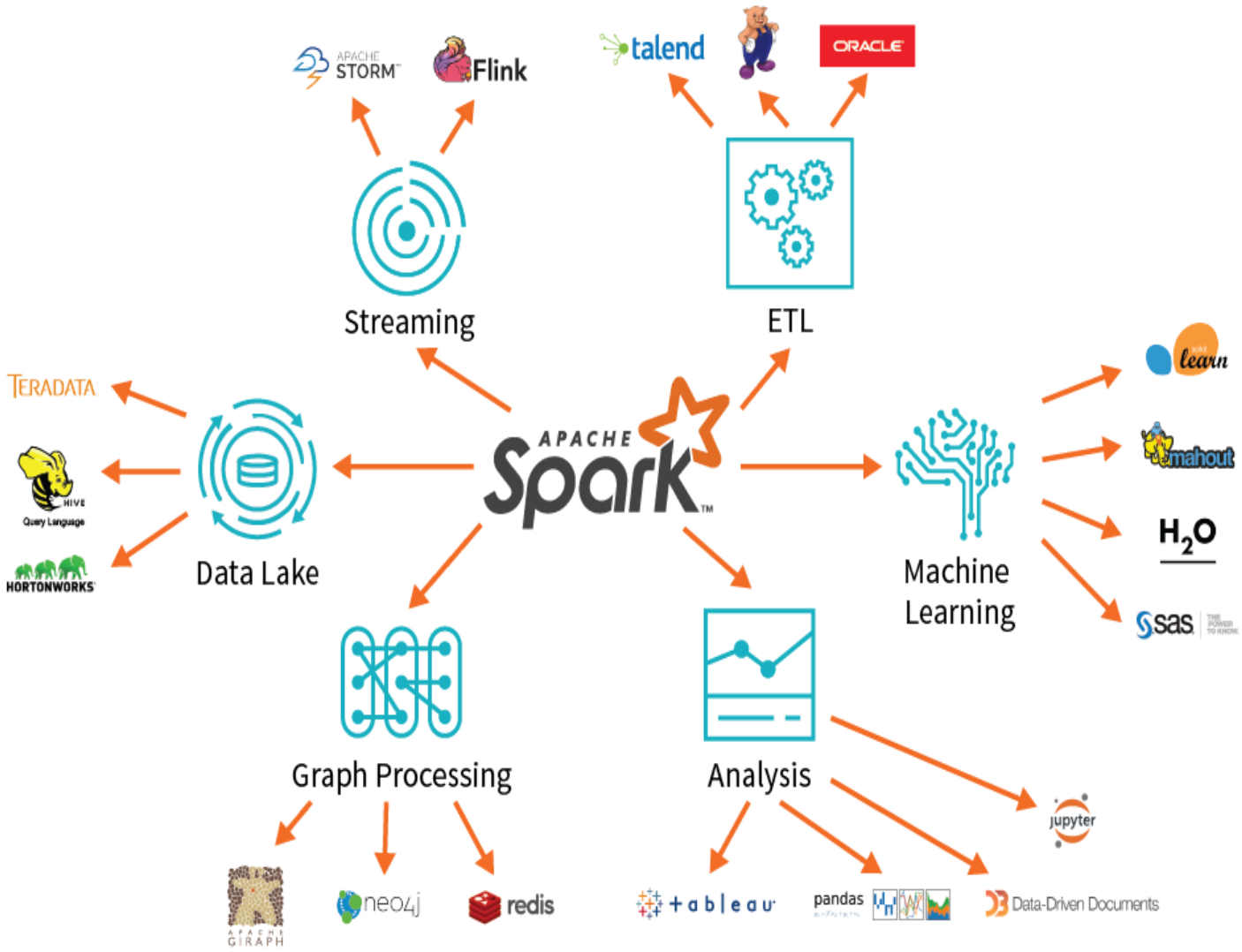


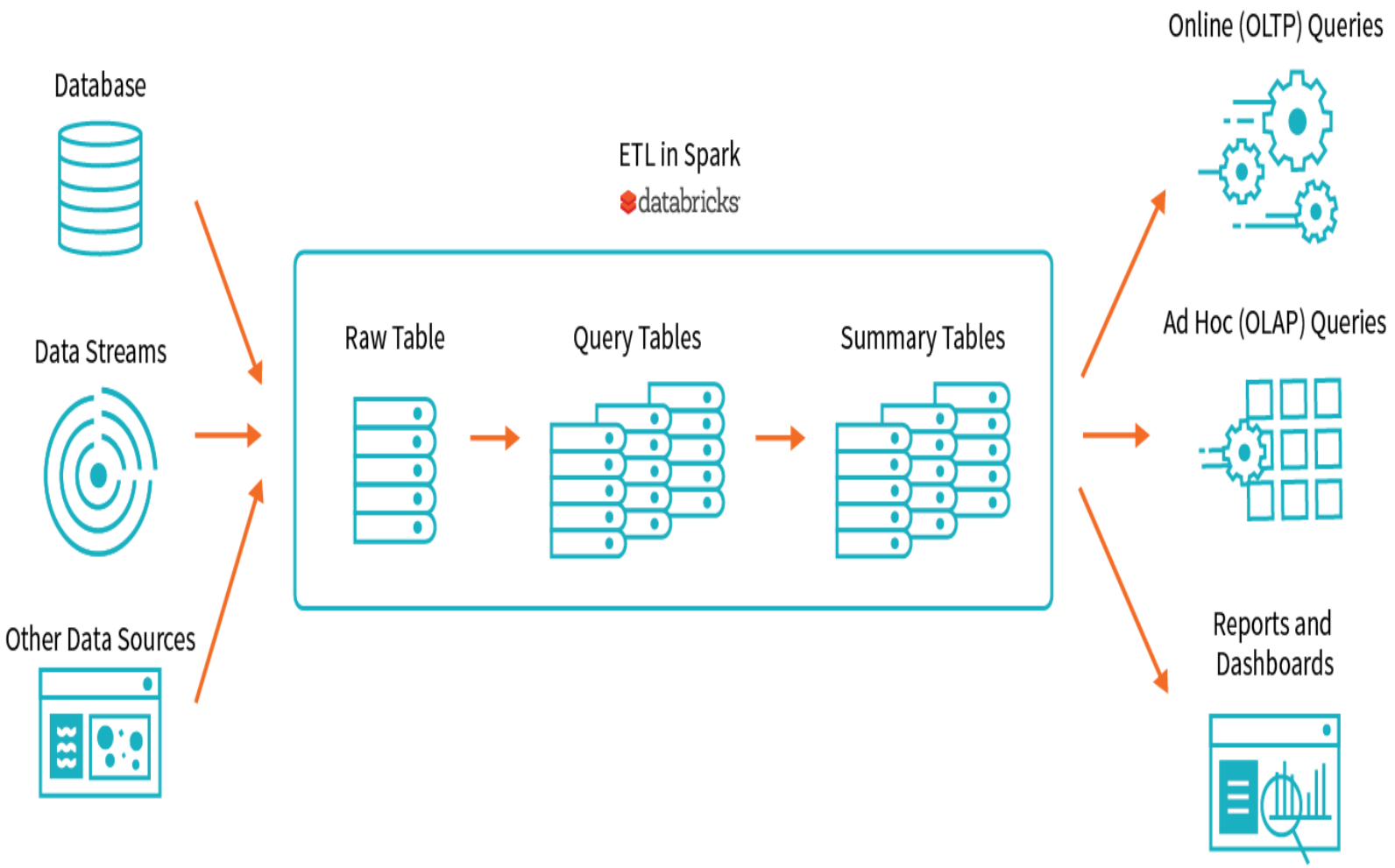
Figure: *Data Parallelism In Spark*

Spark Architecture & Components

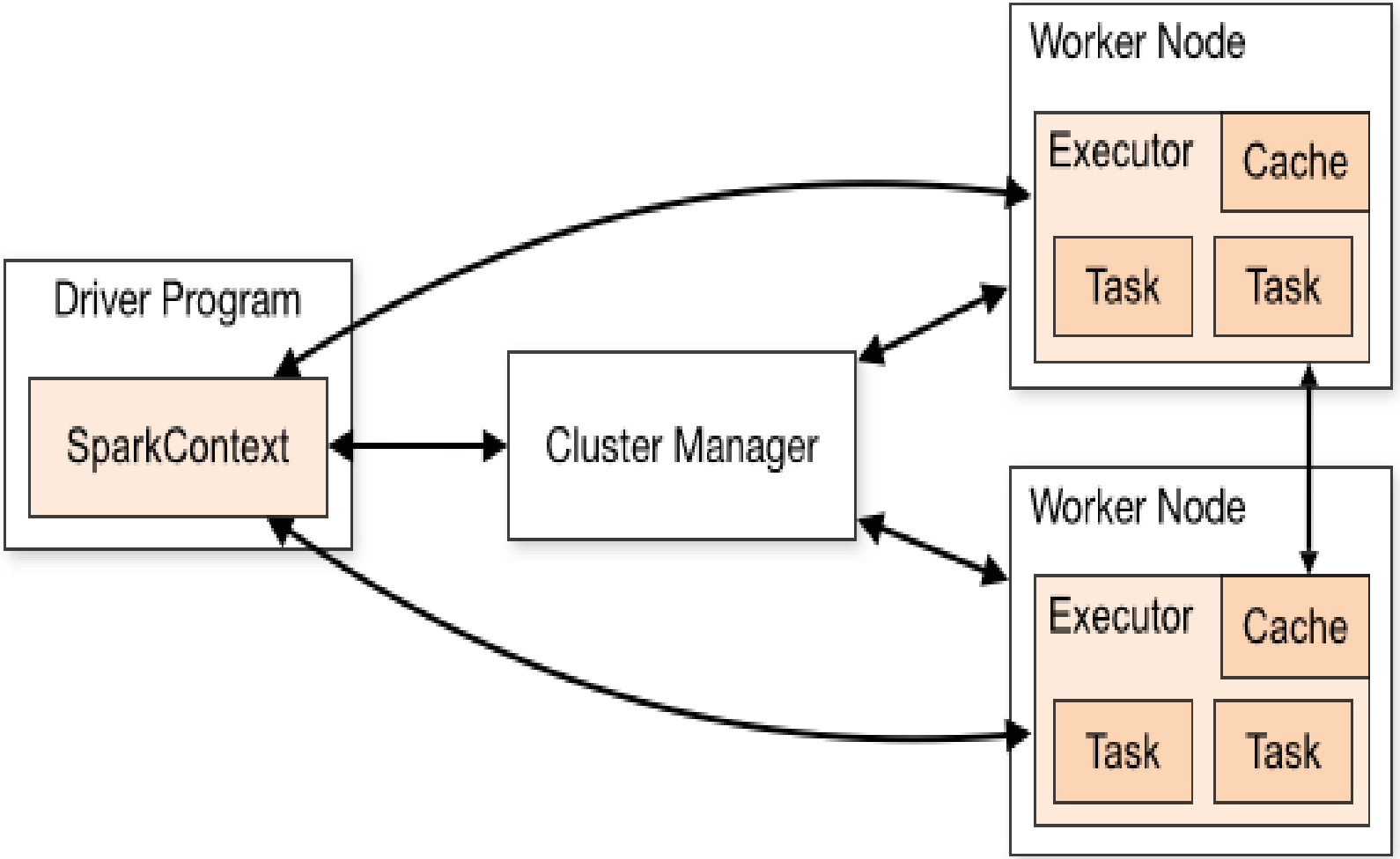
Apache Spark is a powerful open-source processing engine built around speed, ease of use, and sophisticated analytics.



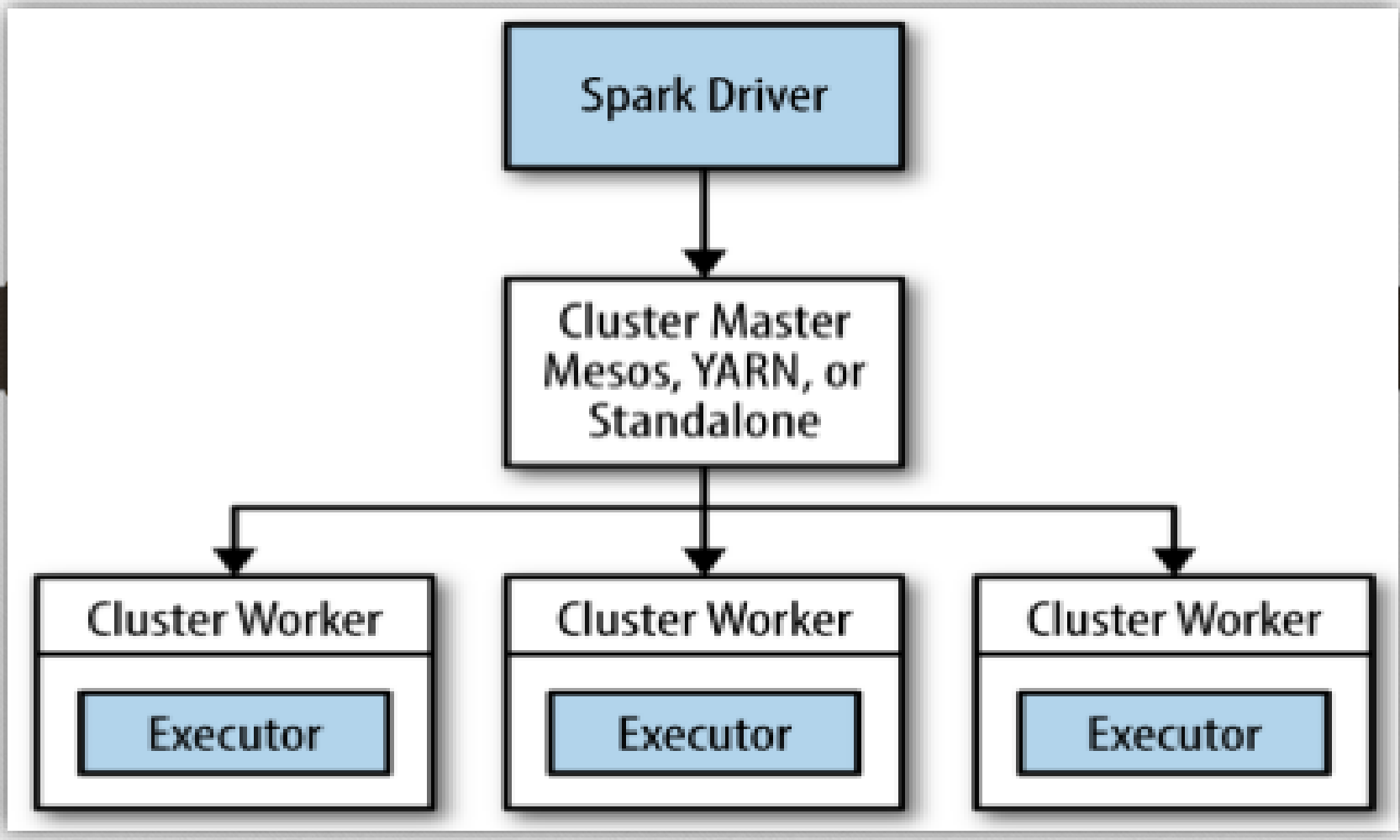

APACHE STORM
Flink
talend
ORACLE
Streaming
ETL
TERADATA
HIVE
Query Language
HORTONWORKS
Data Lake
APACHE
Spark
learn
mahout
H2O
sas
Machine Learning
Graph Processing
Analysis
jupyter
APACHE GIRAPH
neo4j
redis
+ableau
pandas
Data-Driven Documents

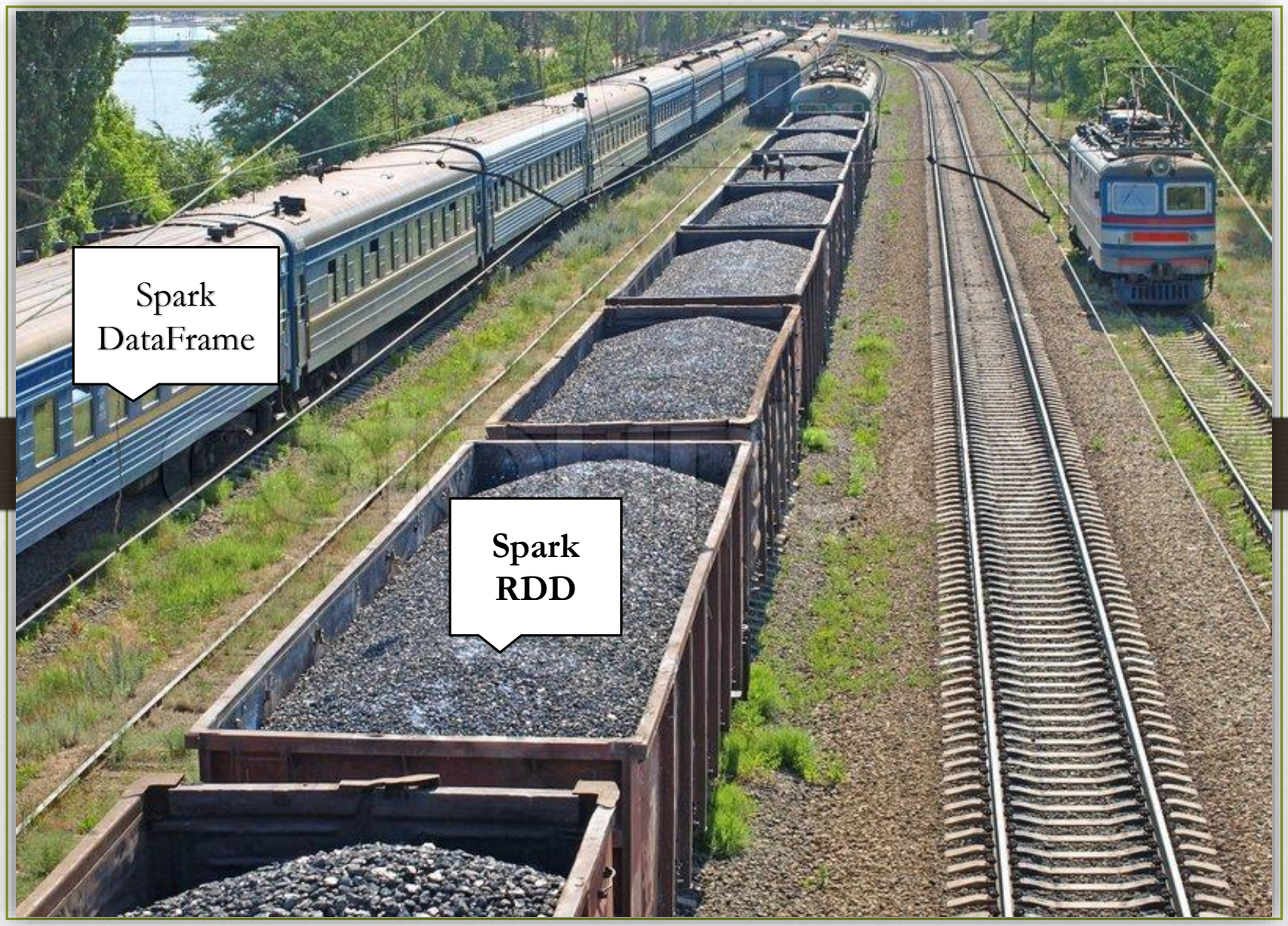

Database
Data Streams
Other Data Sources
ETL in Spark
databricks
Raw Table
Query Tables
Summary Tables
Online (OLTP) Queries
Ad Hoc (OLAP) Queries
Reports and Dashboards

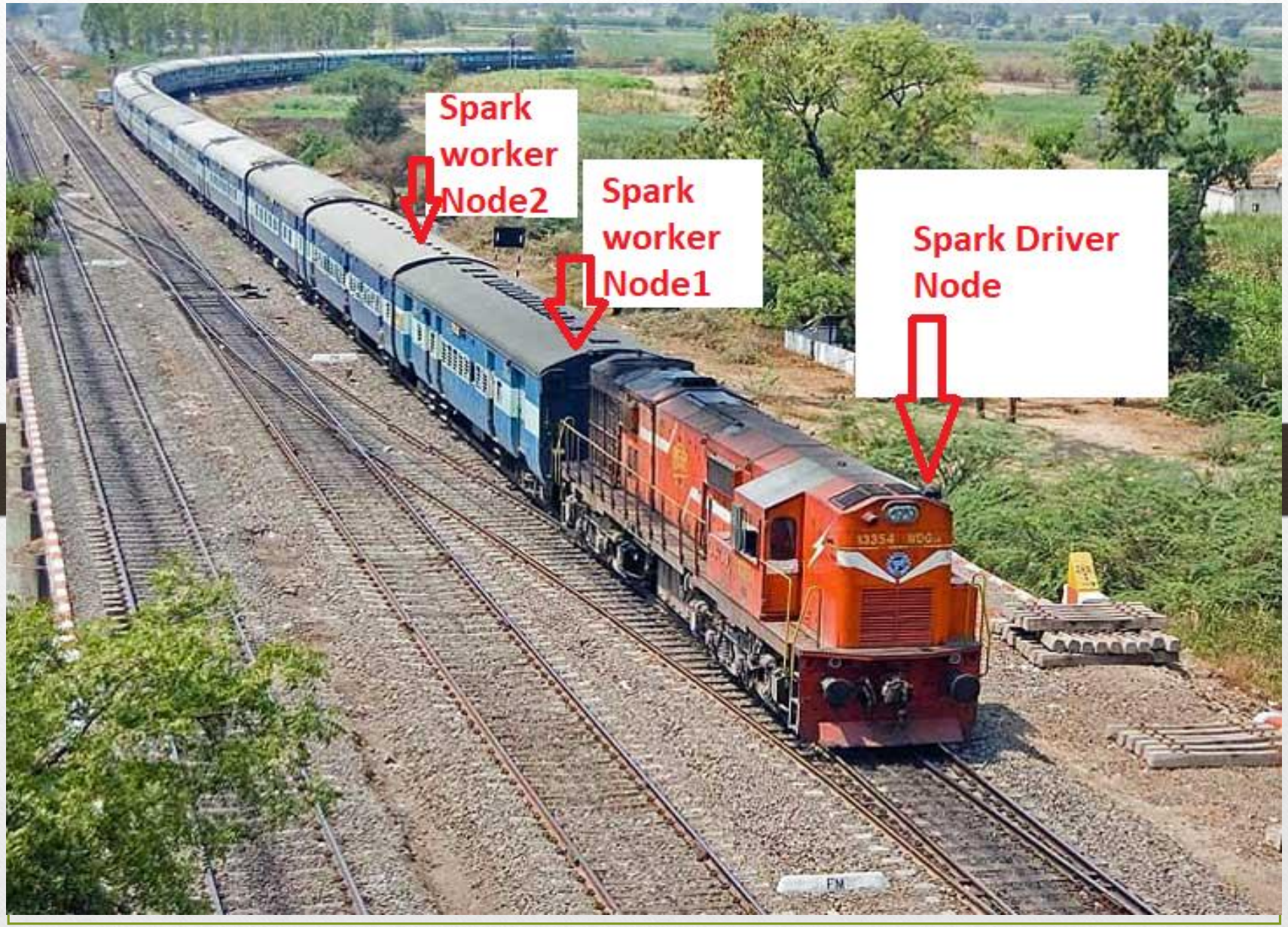
Spark Components Architecture

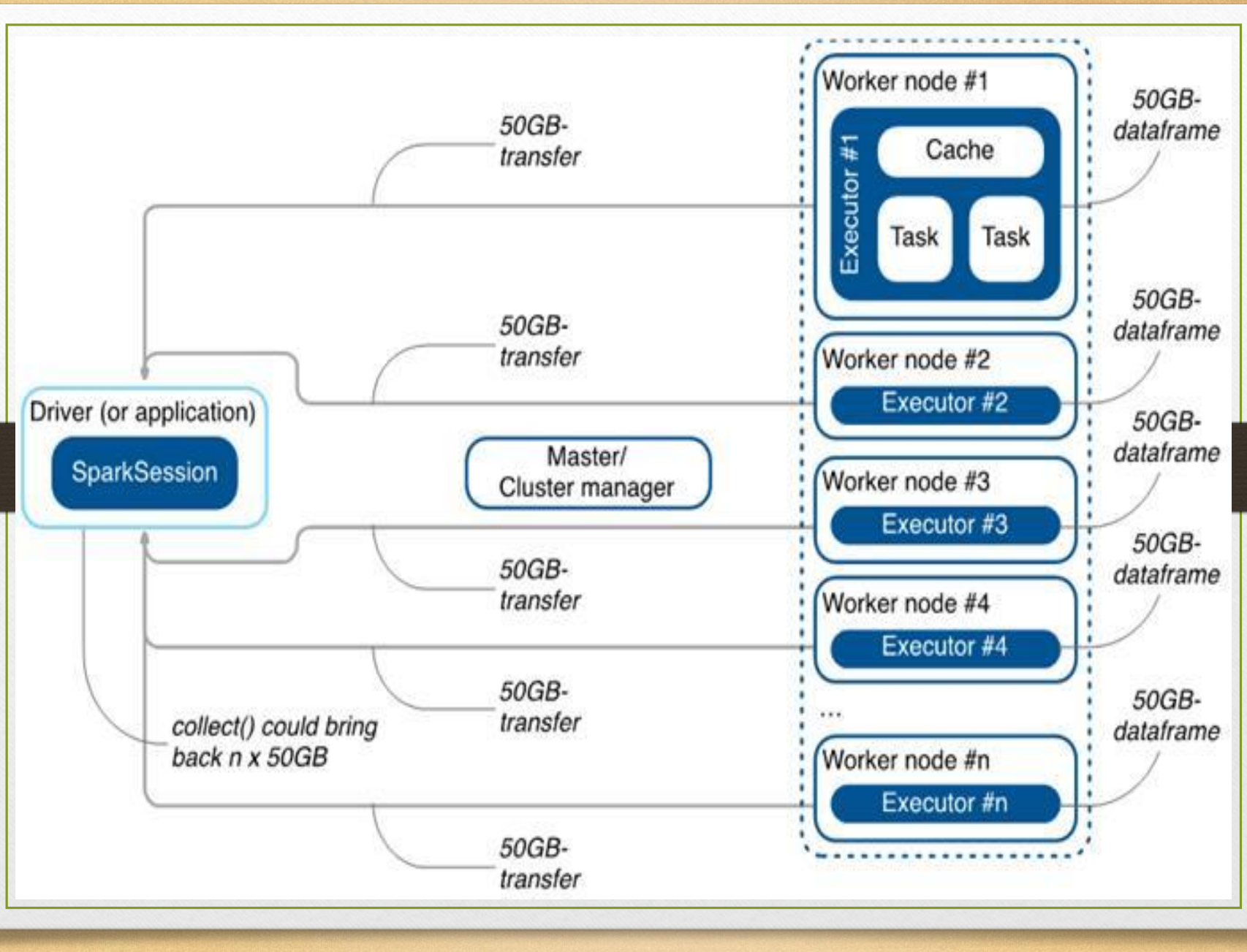


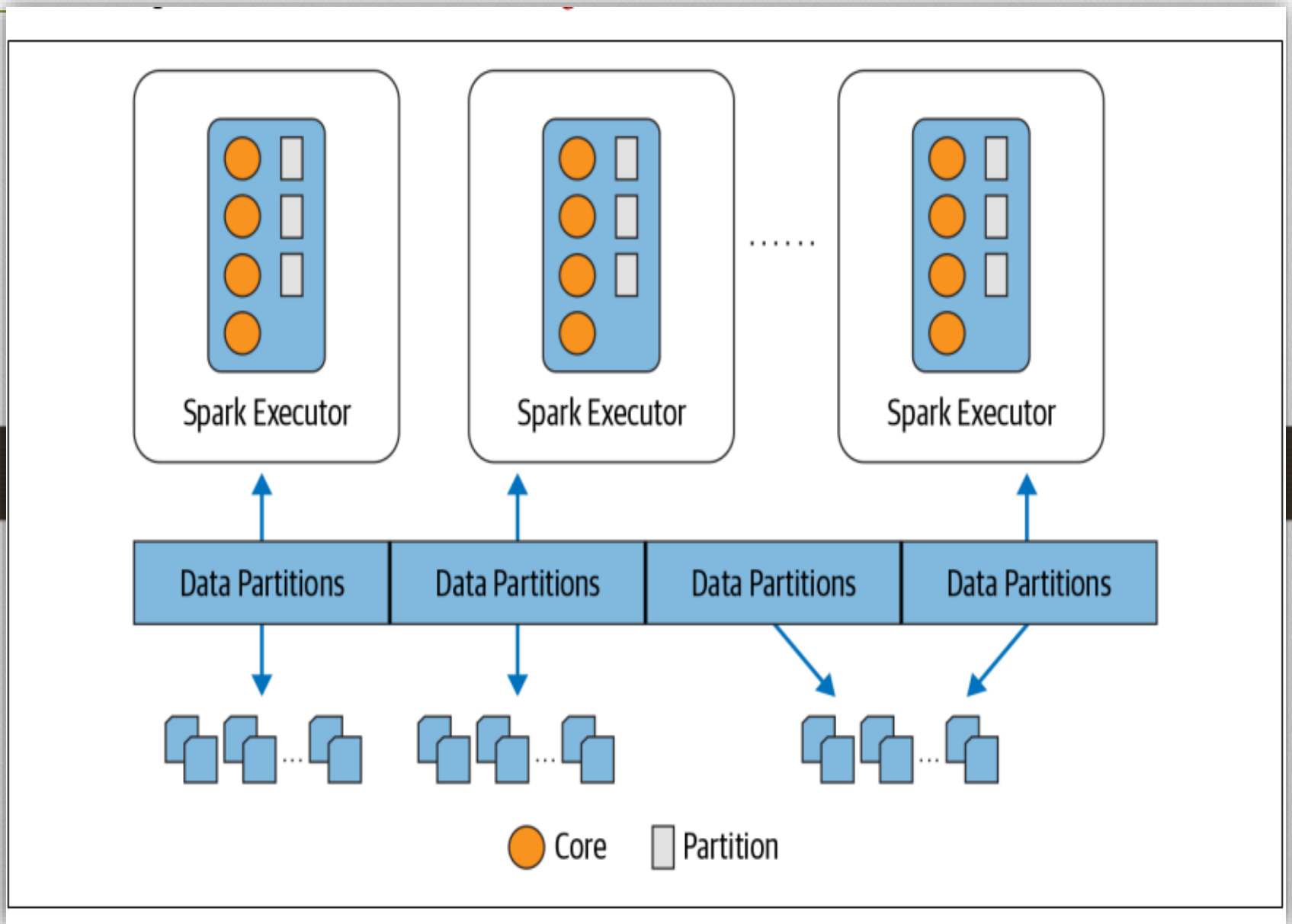
In the cluster mode, Spark uses the master/slave architecture. The central coordinator is called the driver, and the driver communicates with the scattered workers. The scattered workers are also called executors. These drivers and executors are collectively called Spark Application.




Spark
DataFrame
Spark
RDD


Spark worker Node2
Spark worker Node1
Spark Driver Node

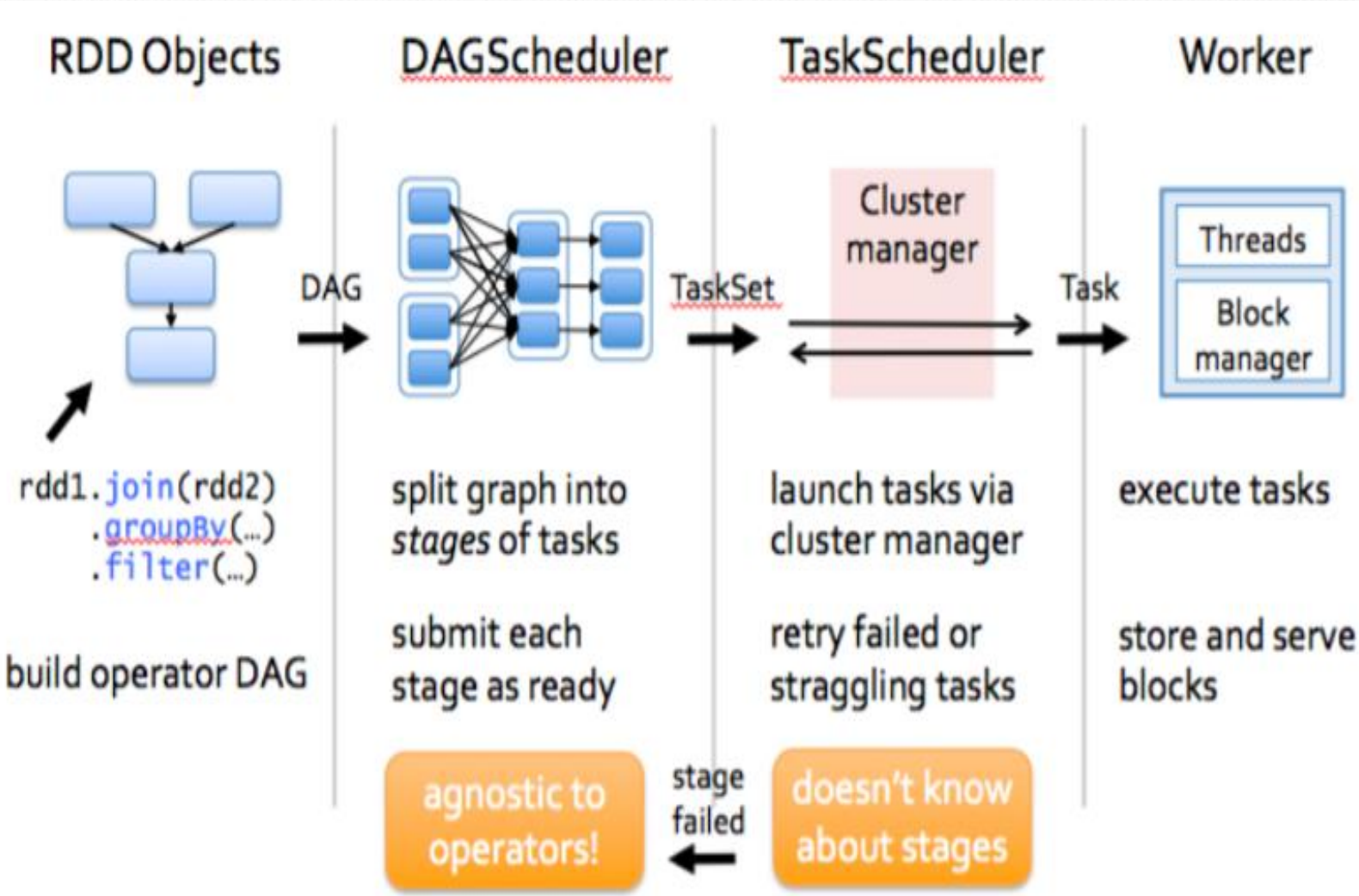

Worker node #1
Executor #1
Cache
Task
Task
50GB-dataframe
50GB-transfer
Worker node #2
Executor #2
50GB-dataframe
50GB-transfer
Driver (or application)
SparkSession
Master/
Cluster manager
Worker node #3
Executor #3
50GB-dataframe
50GB-transfer
Worker node #4
Executor #4
50GB-dataframe
50GB-transfer
collect() could bring
back n x 50GB
...
Worker node #n
Executor #n
50GB-dataframe
50GB-transfer


Spark Executor
Spark Executor
Spark Executor
Data Partitions
Data Partitions
Data Partitions
Data Partitions
Core
Partition

Spark Core

- Contains the basic functionality for
  - task scheduling,
  - memory management,
  - fault recovery,
  - interacting with storage systems,
  - and more.
- Defines the Resilient Distributed Data sets (RDDs)
  - main Spark programming abstraction.

SPARK EXECUTION

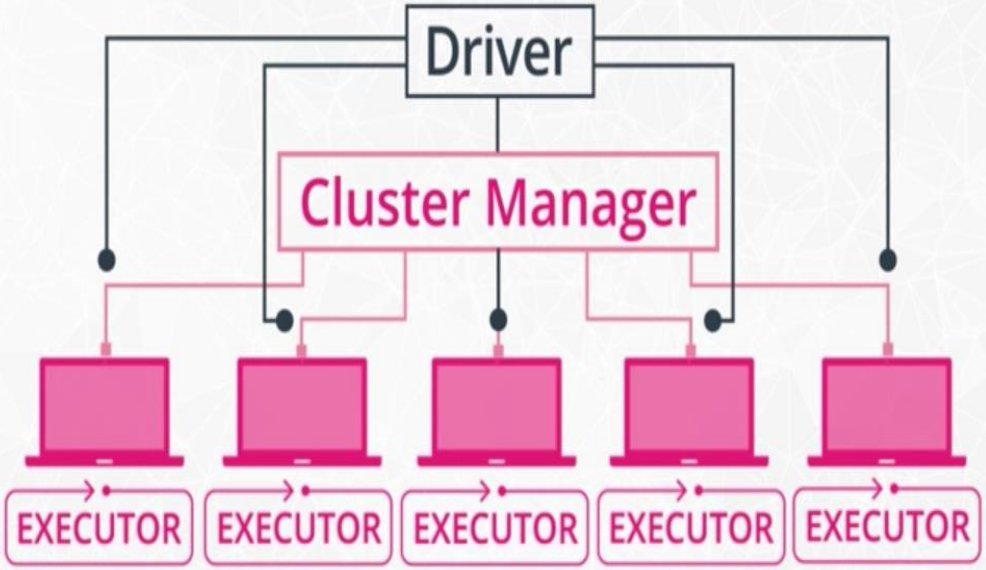


SPARK EXECUTION PROCESS STEPS

- Job: A piece of code which reads some input from HDFS or local, performs some computation on the data and writes some output data.
- Stages: Jobs are divided into stages. Stages are classified as a Map or reduce stages (Its easier to understand if you have worked on Hadoop and want to correlate). Stages are divided based on computational boundaries, all computations (operators) cannot be Updated in a single Stage. It happens over many stages.
- Tasks: Each stage has some tasks, one task per partition. One task is executed on one partition of data on one executor (machine).
- DAG: DAG stands for Directed Acyclic Graph, in the present context its a DAG of operators.
- Executor: The process responsible for executing a task.
- Master: The machine on which the Driver program runs
- Slave: The machine on which the Executor program runs

SPARK MODES

LOCAL MODE



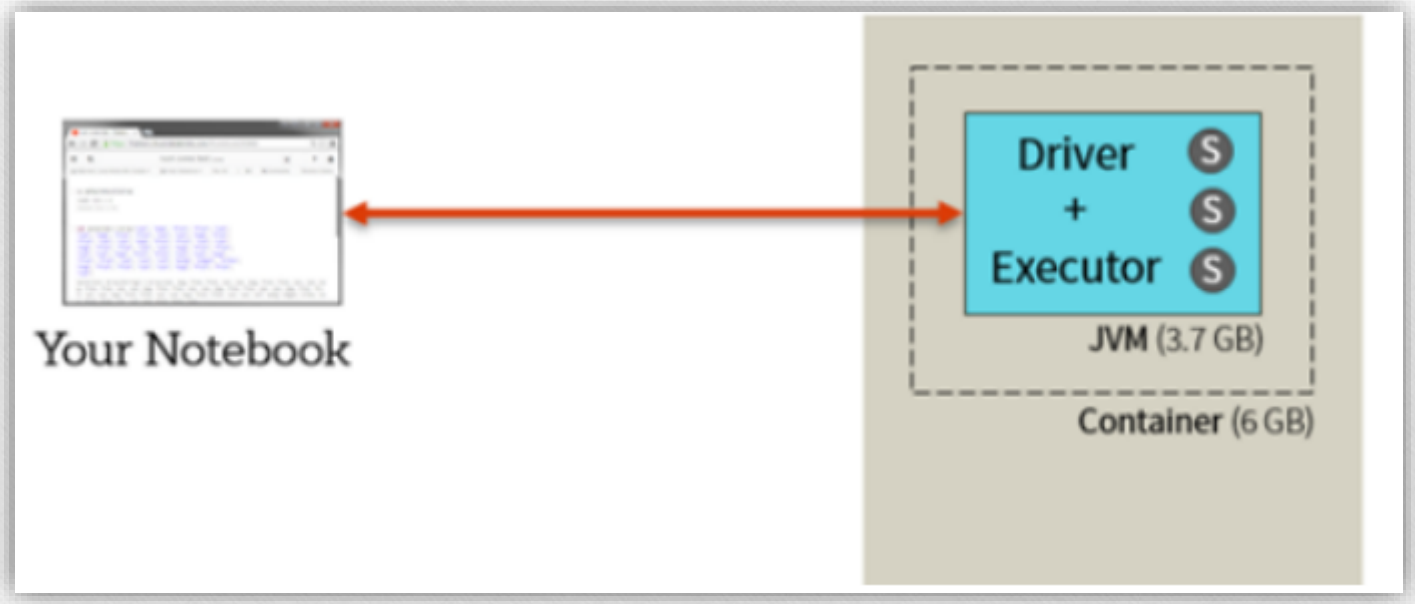
CLUSTER MODES

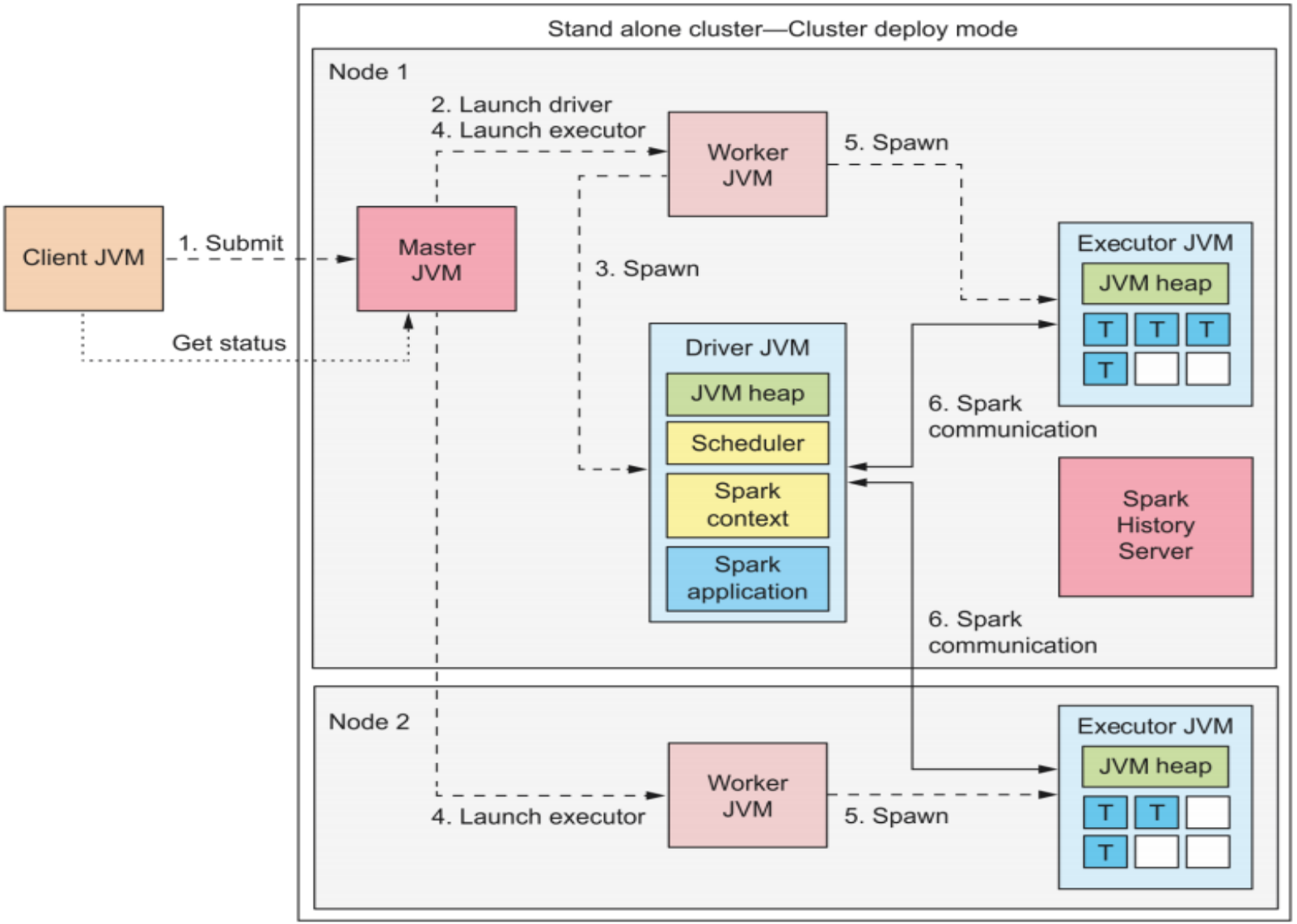
- Standalone
- YARN
- Mesos

SPARK Cluster Execution Types

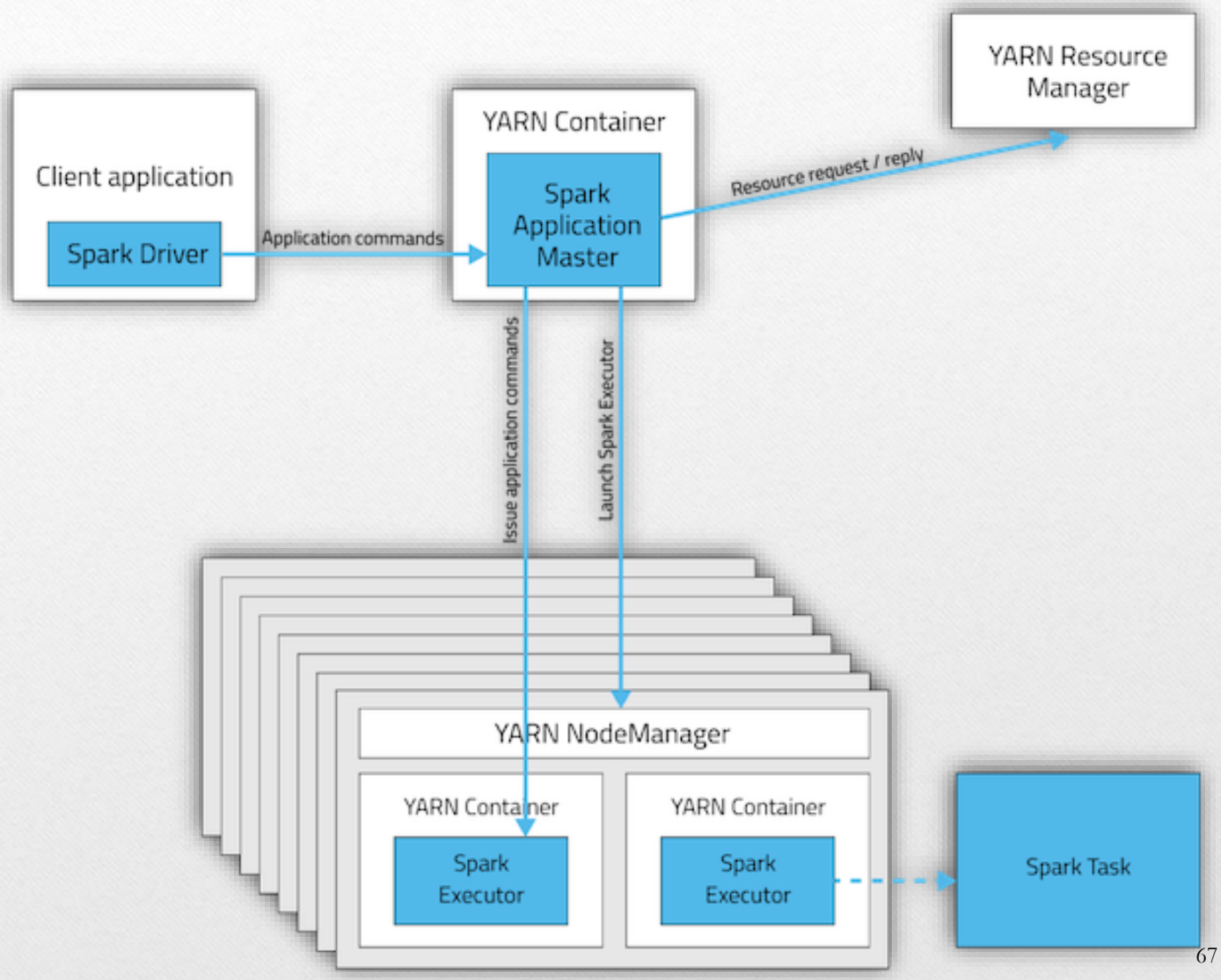
Mode	Spark driver	Spark executor	Cluster manager
Local	Runs on a single JVM, like a laptop or single node	Runs on the same JVM as the driver	Runs on the same host
Standalone	Can run on any node in the cluster	Each node in the cluster will launch its own executor JVM	Can be allocated arbitrarily to any host in the cluster
YARN (client)	Runs on a client, not part of the cluster	YARN's NodeManager's container	YARN's Resource Manager works with YARN's Application Master to allocate the containers on NodeManagers for executors
YARN (cluster)	Runs with the YARN Application Master	Same as YARN client mode	Same as YARN client mode
Kubernetes	Runs in a Kubernetes pod	Each worker runs within its own pod	Kubernetes Master

Local Mode

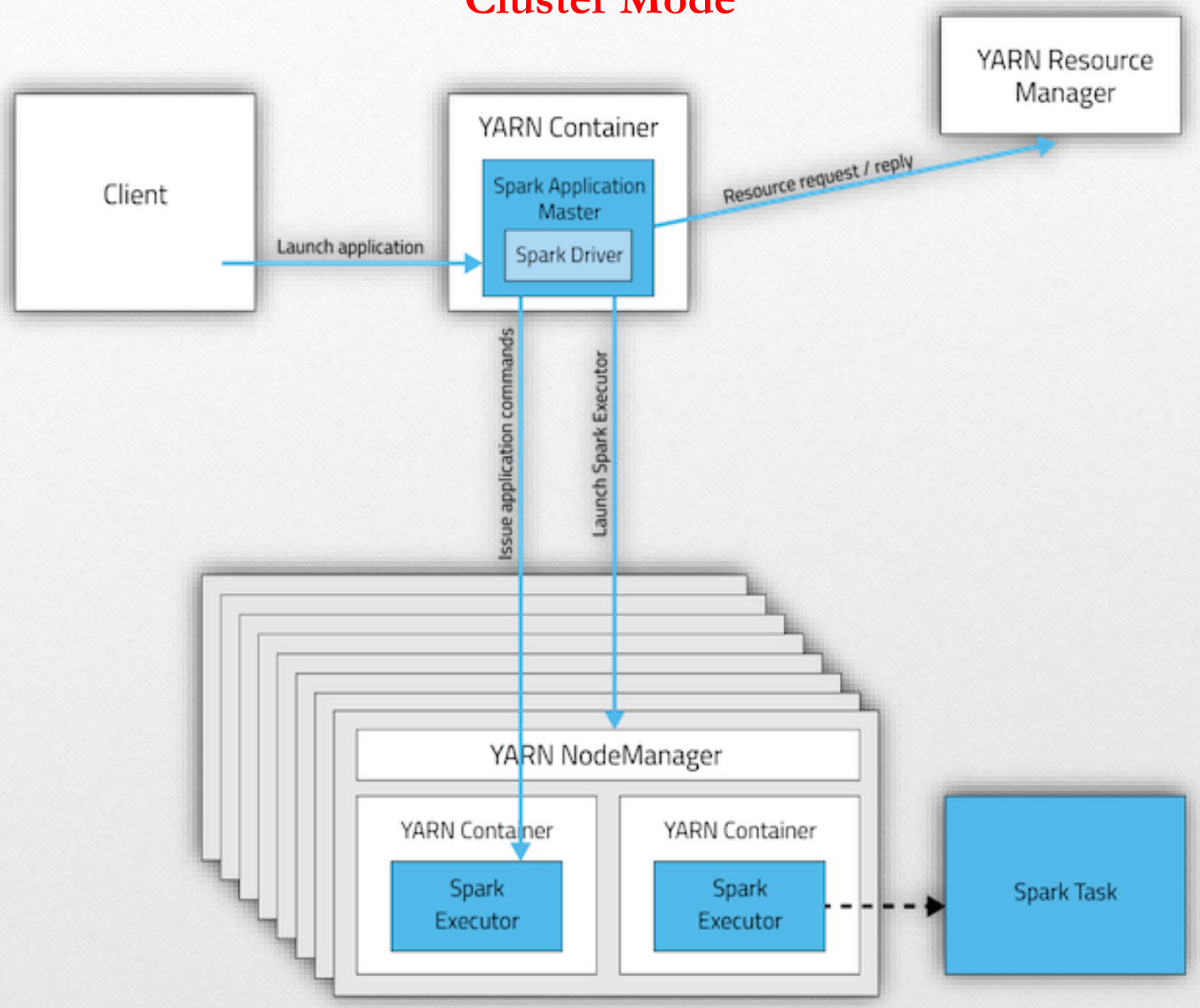



Stand alone cluster—Cluster deploy mode
Node 1
2. Launch driver
4. Launch executor
Worker JVM
5. Spawn
Client JVM
1. Submit
Master JVM
3. Spawn
Executor JVM
JVM heap
T T T
T
Get status
Driver JVM
JVM heap
Scheduler
Spark context
Spark application
6. Spark communication
Spark History Server
6. Spark communication
Node 2
Executor JVM
JVM heap
T T
T
Worker JVM
4. Launch executor
5. Spawn

Client Mode


YARN Resource Manager
Client application
Spark Driver
Application commands
YARN Container
Spark Application Master
Resource request / reply
Issue application commands
Launch Spark Executor
YARN NodeManager
YARN Container
Spark Executor
YARN Container
Spark Executor
Spark Task

Cluster Mode



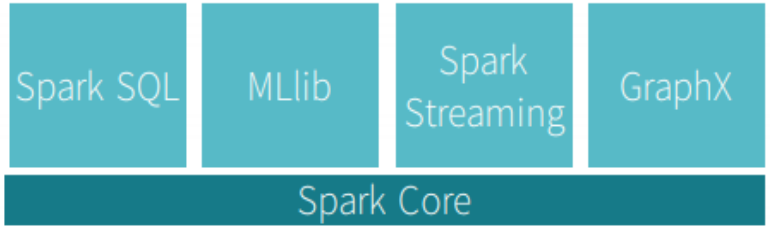
	YARN Cluster	YARN Client	Spark Standalone
Driver runs in:	Application Master	Client	Client
Who requests resources?	Application Master	Application Master	Client
Who starts executor processes?	YARN NodeManager	YARN NodeManager	Spark Slave
Persistent services	YARN ResourceManager and NodeManagers	YARN ResourceManager and NodeManagers	Spark Master and Workers
Supports Spark Shell?	No	Yes	Yes

In **yarn-cluster mode**, the driver runs in the Application Master. This means that the same process is responsible for both driving the application and requesting resources from YARN, and this process runs inside a YARN container. The client that starts the app doesn't need to stick around for its entire lifetime.

The yarn-cluster mode, is not well suited to using Spark interactively. Spark applications that require user input, like spark-shell and PySpark, need the Spark driver to run inside the client process that initiates the Spark application. In yarn-client mode, the Application Master is merely present to request executor containers from YARN. The client communicates with those containers to schedule work after they start:

Spark Libraries on top of RDDs

- SQL (Spark SQL)
  - Full Hive SQL support with UDF, UDAFs, etc
  - how: Internally keep RDDs of row objects (or RDD of column segments)
- Machine Learning (MLlib)
  - Library of machine learning algorithms
  - how: Cache an RDD, repeatedly iterate it



- Streaming (Spark Streaming)
  - Streaming of real-time data
  - how: Series of RDDs, each containing seconds of real-time data
- Graph Processing (GraphX)
  - Iterative computation on graphs (e.g. social network)
  - how: RDD of Tuple<Vertex, Edge, Vertex> and perform self joins

History of Spark APIs

Distribute collection of JVM objects

Functional Operators (map, filter, etc.)

Distribute collection of Row objects

Expression-based operations and UDFs

Logical plans and optimizer

Fast/efficient internal representations

Internally rows, externally JVM objects

Almost the "Best of both worlds": **type safe + fast**

But slower than DF
Not as good for interactive analysis, especially Python

databricks

Spark API's

1) Resilient Distributed Dataset (RDD) => (Spark1.0)

An RDD stands for Resilient Distributed Datasets. It is Read-only partition collection of records. RDD is the fundamental data structure of Spark. It allows a programmer to perform in-memory computations on large clusters in a fault-tolerant manner. Thus, speed up the task

RDD Limitations: Handling structured data ->Unlike Dataframe and datasets, RDDs don't infer the schema of the ingested data and requires the user to specify it.

2) DataFrame => (Spark1.3)

DataFrame data organized into named columns. For example a table in a relational database. It is an immutable distributed collection of data. DataFrame in Spark allows developers to impose a structure onto a distributed collection of data, allowing higher-level abstraction.

Dataframe Limitations: Compile-time type safety

3) Dataset => (Spark1.6)

It is an extension to Dataframe API, the latest abstraction which tries to provide best of both RDD and Dataframe. Datasets API provides compile time safety which was not available in Data frames.

DataSet Provides best of both RDD and Dataframe

RDD (functional programming, type safe),

DataFrame (relational model, Query optimazation , Tungsten execution, sorting and shuffling)

The Dataset API is available in Scala and Java

Differences Between RDD & DataFrame & DataSet

RDD Features:-

1) Distributed collection
2) Immutable
3) Fault tolerant
4) Lazy evaluations
5) Functional transformations => Transformations and Actions
6) Data processing formats => structured as well as unstructured data
7) Programming Languages supported => Java, Scala, Python and R.

DataFrame Features:

1) Distributed collection of Row Object
2) Data Processing
3) Optimization using catalyst optimizer
4) Hive Compatibility
5) Tungsten
6) Programming Languages supported => Java, Scala, Python and R.

DataSet Features:

1) Provides best of both RDD and Dataframe
2) Encoders
3) Programming Languages supported => Java, Scala
4) Type Safety

Disadvantages Of RDD & DATAFRAME & DATASET

Disadvantages of RDDs

If you choose to work with RDD you will have to optimize each and every RDD. In addition, unlike Datasets and DataFrames, RDDs don't infer the schema of the data ingested therefore you will have to specify it.

Disadvantages of DataFrames

The main drawback of DataFrame API is that it does not support compile time safely, as a result, the user is limited in case the structure of the data is not known.

Disadvantages of DataSets

The main disadvantage of datasets is that they require typecasting into strings.

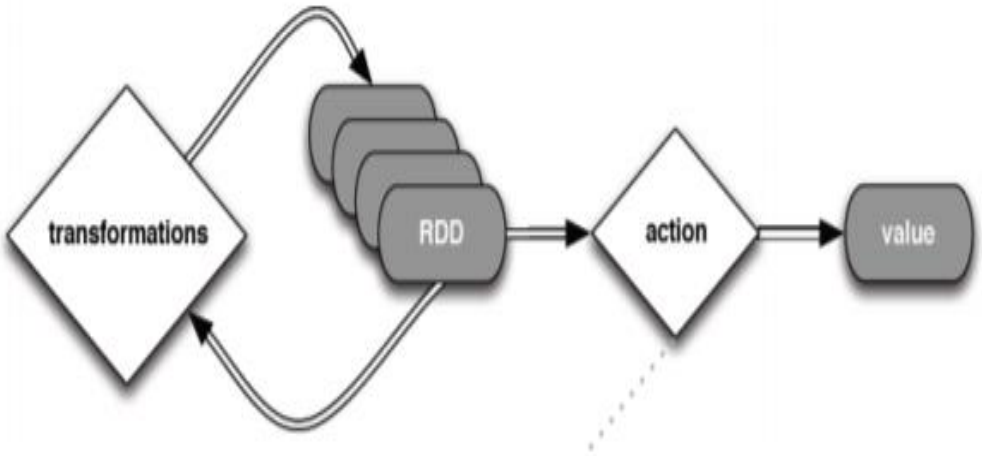
Spark Context

- A Spark program first creates a **SparkContext** object
  - » Tells Spark how and where to access a cluster
  - » pySpark shell and Databricks Cloud automatically create the **sc** variable
  - » iPython and programs must use a constructor to create a new **SparkContext**
- Use **SparkContext** to create RDDs

Spark Core

RDD – Resilient Distributed Dataset

- A primary abstraction in Spark – a fault-tolerant collection of elements that can be operated on in parallel.
- Two Types
  - Parallelized Scala collections
  - Hadoop datasets
- Transformations and Actions can be performed on RDDs.



Transformations

- Operate on an RDD and return a new RDD.
- Are Lazily Evaluated

Actions

- Return a value after running a computation on a RDD.
- The DAG is evaluated only when an action takes place.

Initializing Pyspark

```
pip install pyspark
```

```
Requirement already satisfied: pyspark in c:\users\raveendra\anacon
Requirement already satisfied: py4j==0.10.9 in c:\users\raveendra\a
Note: you may need to restart the kernel to use updated packages.
```

```
pip install findspark
```

```
Requirement already satisfied: findspark in c:\users\raveendra\anac
Note: you may need to restart the kernel to use updated packages.
```

```
#### Importing findspark and intializing findspark...
```

```
import findspark
findspark.init()
#findspark.init(SPARK_HOME)
```

Creating Pyspark Session in local system

```
#### Creating Spark Session
```

```
from pyspark.sql import SparkSession
#spark = SparkSession.builder\
#.master("local".format(2))\
#.appName("Localspark")\
#.getOrCreate()

n_cpu = 2
spark = SparkSession.builder.appName('localSpark')\
.master('local[{}]'.format(n_cpu))\
.config("spark.driver.memory", "1g")\
.config('spark.executor.memory', '2g')\
.config('spark.executor.cores', '3')\
.config('spark.cores.max', '3')\
.getOrCreate()
```

```
spark
```

```
]: SparkSession - in-memory
   SparkContext
```

Creating SparkContext in local System

```
from pyspark import SparkContext

sc = SparkContext.getOrCreate()

sc
```

]:

SparkContext

Spark UI

Version
`v3.0.1`

Master
`local[2]`

AppName
`localSpark`

Stopping SparkSession & SparkContext using STOP() method

```
#stop spark session
#spark.stop()
# stop SparkContext
#sc.stop()
```

Get All Spark Config Parameter Values

```
#Get all configuration
sc.getConf().getAll()
```

```
5]: [('spark.driver.memory', '4g'),
     ('spark.executor.memory', '4g'),
     ('spark.executor.id', 'driver'),
     ('spark.executor.cores', '4'),
     ('spark.cores.max', '4'),
     ('spark.driver.host', 'VICKY'),
     ('spark.app.id', 'local-1603445567945'),
     ('spark.app.name', 'Spark Updated Conf'),
     ('spark.rdd.compress', 'True'),
     ('spark.serializer.objectStreamReset', '100'),
     ('spark.submit.pyFiles', ''),
     ('spark.submit.deployMode', 'client'),
     ('spark.ui.showConsoleProgress', 'true'),
     ('spark.master', 'local[2]'),
     ('spark.driver.port', '50729')]
```

Setting SparkContext Configuration Parameters Values

SET Configuration parameters in spark

Cmd 11

```
conf = spark.sparkContext._conf.setAll([('spark.executor.memory', '2g'),
                                        ('spark.app.name', 'Spark Updated Conf'),
                                        ('spark.executor.cores', '4'),
                                        ('spark.cores.max', '4'),
                                        ('spark.driver.memory','4g')])
```

Spark Program Lifecycle

1. Create RDDs from external data or parallelize a collection in your driver program
2. Lazily transform them into new RDDs
3. **cache()** some RDDs for reuse
4. Perform actions to execute parallel computation and produce results

Creating RDDs

There are many ways to create RDD objects:

1. From list or arrays defined within the program
2. By reading from normal files
3. Reading from Hadoop HDFS
4. From the output of normal databases queries

Help() function: The help() function is used to display the documentation string and also facilitates you to see the help related to modules, keywords, attributes, etc.

Dir() function: The dir() function is used to display the defined methods.

An RDD can be created 2 ways

- Parallelize a collection

```
# Parallelize in Python
wordsRDD = sc.parallelize([“fish", “cats“, “dogs”])
```

- Take an existing in-memory collection and pass it to SparkContext's parallelize method
- Not generally used outside of prototyping and testing since it requires entire dataset in memor on one machine

- Read from File

```
# Read a local txt file in Python
linesRDD = sc.textFile("/path/to/README.md")
```

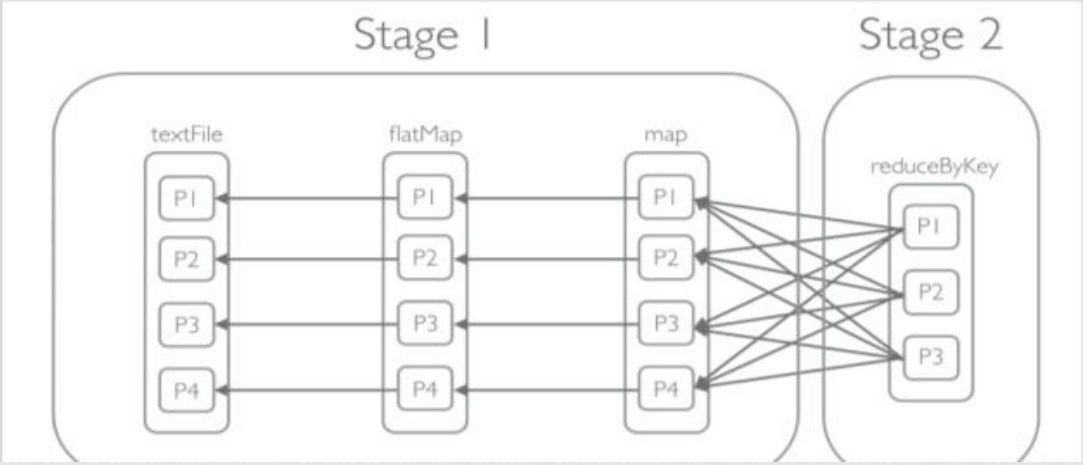
- There are other methods to read data from HDFS, C*, S3, HBase, etc.

Programming with RDDs

- All work is expressed as either:
  - creating new RDDs
  - transforming existing RDDs
  - calling operations on RDDs to compute a result.
- Distributes the data contained in RDDs across the nodes (executors) in the cluster and parallelizes the operations.
- Each RDD is split into multiple **partitions**, which can be computed on different nodes of the cluster.

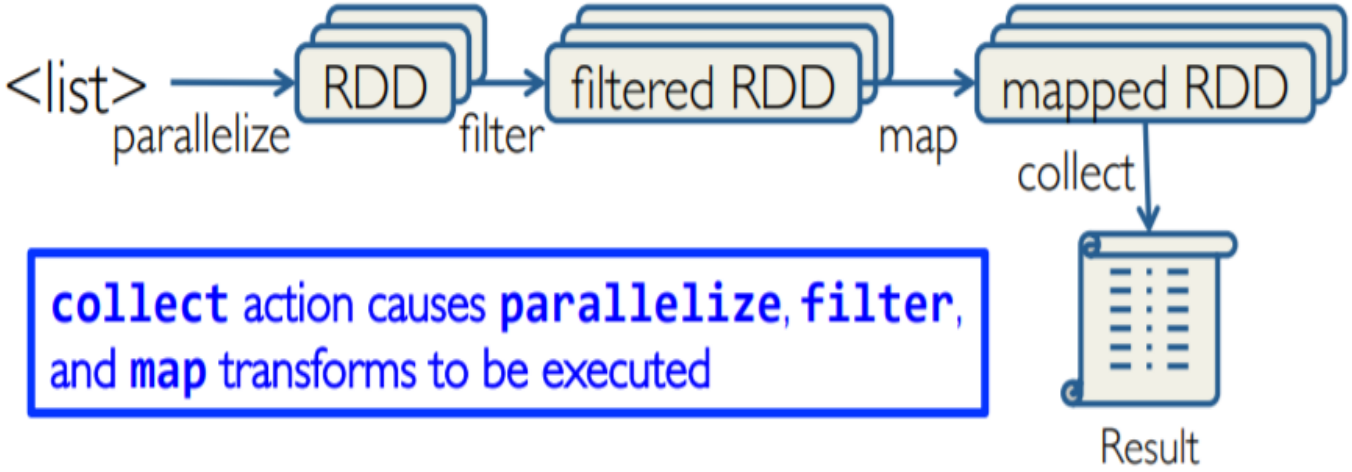
Partitions

- Each RDD is split up into a number of partitions
- Parallelism is determined by the number of partitions
  - `rdd.getNumberOfPartitions()`
- Why is important ?
  - Operations can be performed in parallel in each partition:
    - Textfile + flatMap + map operations can be performed in parallel in P1, P2, P3, P4
    - Operations that can run on the same partition are executed in stages



Working with RDDs

- Create an RDD from a data source:  <list>
- Apply transformations to an RDD: map filter
- Apply actions to an RDD: collect count



Creating an RDD

- Create RDDs from Python collections (lists)

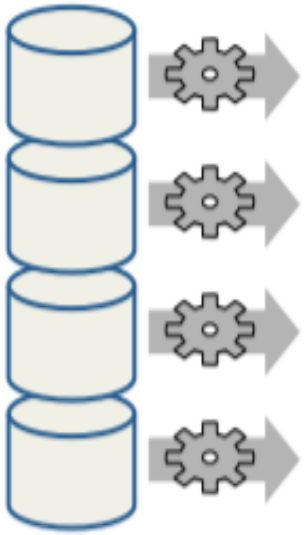
```
>>> data = [1, 2, 3, 4, 5]

>>> data

[1, 2, 3, 4, 5]


>>> rDD = sc.parallelize(data, 4)

>>> rDD

ParallelCollectionRDD[0] at parallelize at PythonRDD.scala:229
```

No computation occurs with **sc.parallelize()**

- Spark only records how to create the RDD with four partitions

Creating an RDD from a File

```
distFile = sc.textFile("...", 4)
```



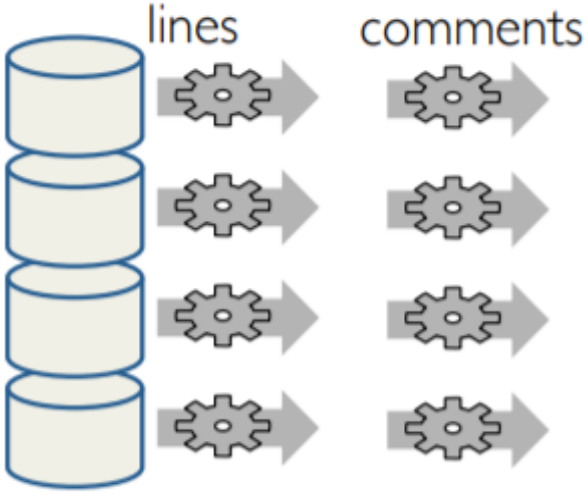
- RDD distributed in 4 partitions
- Elements are lines of input
- *Lazy evaluation* means no execution happens now

Spark Transformations

- Create new datasets from an existing one
- Use *lazy evaluation*: results not computed right away – instead Spark remembers set of transformations applied to base dataset
  - » Spark optimizes the required calculations
  - » Spark recovers from failures and slow workers
- Think of this as a recipe for creating result

Transforming an RDD

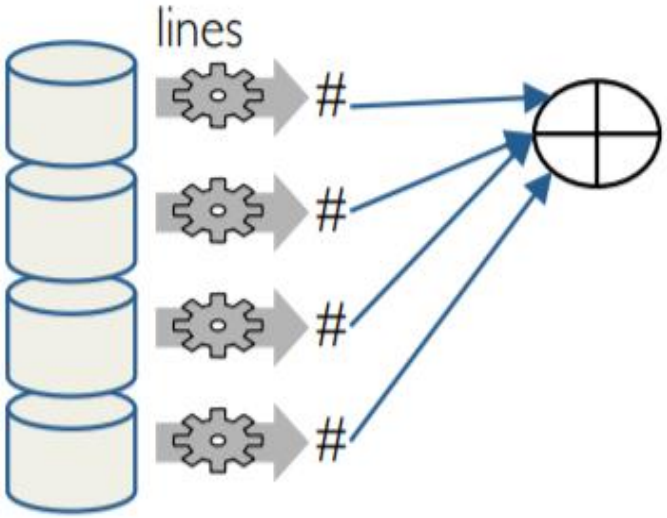
```
lines = sc.textFile("...", 4)

comments = lines.filter(isComment)
```



Lazy evaluation means nothing executes – Spark saves recipe for transforming source

Spark Programming Model

```
lines = sc.textFile("...", 4)

print lines.count()
```

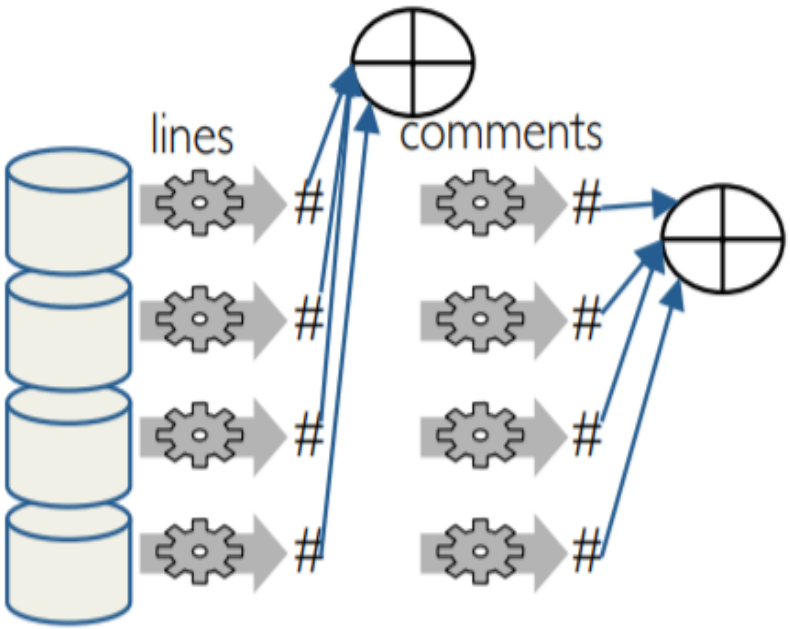


`count()` causes Spark to:

- read data
- sum within partitions
- combine sums in driver

Spark Programming Model

```
lines = sc.textFile("...", 4)
comments = lines.filter(isComment)
print lines.count(), comments.count()
```



Spark recomputes **lines**:

- read data (again)
- sum within partitions
- combine sums in driver

Spark Operations

TRANSFORMATIONS

Spark Operations = +

ACTIONS

RDD ACTIONS LIST

No.	RDD Action	Expecting Result
1	collect()	Convert RDD to in-memory list
2	take(3)	First 3 elements of RDD
3	top(3)	Top 3 elements of RDD
4	count()	Find total no of values in RDD.
5	min()	Find minimum value from the RDD list
6	max()	Find maximum value from the RDD List
7	sum()	Find element sum (assumes numeric elements)
8	mean()	Find element mean (assumes numeric elements)
9	stdev()	Find element deviation (assumes numeric elements)
10	takeSample(withReplacement=True, 3)	Create sample of 3 elements with replacement

RDD ACTIONS - LIST 2

No.	RDD Action	Expecting Result
11	reduce()	Reduce is a spark action that aggregates a data set (RDD) element using a function.
12	countByKey()	Count the number of elements for each key, and return the result to the master as a dictionary.
13	CountByValue()	Return the count of each unique value in this RDD as a dictionary of (value, count) pairs.
14	fold()	Aggregate the elements of each partition
15	range()	Create a new RDD of int containing elements from *start* to *end* (exclusive)
16	variance()	Compute the variance of this RDD's elements.
17	sampleVariance()	Compute the sample variance of this RDD's elements (which corrects for bias in estimating the variance by dividing by N-1 instead of N).
18	saveAsTextFile()	Save this RDD as a text file, using string representations of elements.
19	saveAsPickleFile()	Save this RDD as a SequenceFile of serialized objects
20	Stats()	Stats will give complete information count, min, max, stdev and mean

RDD - TRANSFORMATIONS

No.	RDD Action	Expecting Result
1	**map()**	Return a new distributed dataset formed by passing each element of the source through a function *func*.
2	**filter()**	Return a new dataset formed by selecting those elements of the source on which *func* returns true.
3	**flatMap()**	Similar to map, but each input item can be mapped to 0 or more output items (so *func* should return a Seq rather than a single item).
4	**mapPartitions()**	Similar to map, but runs separately on each partition (block) of the RDD
5	**mapPartitionsWithIndex()**	Similar to mapPartitions, but also provides func with an integer value representing the index of the partition
6	**sample()**	Sample a fraction *fraction* of the data, with or without replacement, using a given random number generator seed.
7	**union()**	Return a new dataset that contains the union of the elements in the source dataset and the argument.
8	**intersection()**	Return a new RDD that contains the intersection of elements in the source dataset and the argument.
9	**distinct()**	Return a new dataset that contains the distinct elements of the source dataset.
10	**groupByKey()**	When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable<V>) pairs

RDD – TRANSFORMATIONS - JOINS

No	TRANSFORMATION	Expecting Result
1	join()	This takes in 2 Pair RDDs, and returns only matched records from both RDD's
2	leftOuterJoin()	This takes in 2 Pair RDDs, and its returns matched records from both RDD's and unmatched record from left RDD
3	rightOuterJoin()	This takes in 2 Pair RDDs, and its returns matched records from both RDD's and unmatched record from Right RDD
4	fullOuterJoin()	This takes in 2 Pair RDDs, and its returns matched records from both RDD's and unmatched records from both left and right RDD's
5	cartesian()	This takes in 2 pair RDD's data and it will apply cross product (multiplication) of each key (record) to another RDD.

RDD Dependencies

- Narrow dependencies
  - allow for pipelined execution on one cluster node
  - easy fault recovery
- Wide dependencies
  - require data from all parent partitions to be available and to be shuffled across the nodes
  - a single failed node might cause a complete re-execution.

RDD (Resilient Distributed Dataset)

Terminologies

- RDD stands for Resilient Distributed Dataset, these are the elements that run and operate on multiple nodes to do parallel processing on a cluster.

RDDs are...

- immutable
- fault tolerant / automatic recovery
- can apply multiple ops on RDDs

RDD operation are...

- Transformation
- Action

Basic Operations (Ops)

- count(): Number of elements in the RDD is returned.
- collect(): All the elements in the RDD are returned.
- foreach(f): input callable, and returns only those elements which meet the condition of the function inside foreach.
- filter(f): input callable, and returns new RDDs containing the elements which satisfy the given callable
- map(f, preservesPartitioning = False): A new RDD is returned by applying a function to each element in the RDD
- reduce(f): After performing the specified commutative and associative binary operation, the element in the RDD is returned.
- join(other, numPartitions = None): It returns RDD with a pair of elements with the matching keys and all the values for that particular key.
- cache(): Persist this RDD with the default storage level (MEMORY_ONLY). You can also check if the RDD is cached or not

ACTIONS

Cmd 70

reduce(func) Action

- Aggregate the elements of the dataset using a function func (which takes two arguments and returns one).
- The function should be commutative and associative so that it can be computed correctly in parallel.

Cmd 71

```
1  # reduce numbers 1 to 10 by adding them up
2  x = sc.parallelize([1,2,3,4,5,6,7,8,9,10])
3  y = x.reduce(lambda a,b: a+b)
4  print(x.collect())
5  print(y)
6
```

▸ (2) Spark Jobs

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
55
```

Command took 0.57 seconds -- by pysparktelugu@gmail.com at 10/27/2020, 7:03:25 PM on datacluster

Stats()

- Return a StatCounter object that captures the mean, variance and count of the RDD's elements in one operation.

Cmd 101

```
1 list_rdd = sc.parallelize([1,2,3,4,5,6,7,8,9,10,11,12,13,14,15])
2 list_rdd.stats()
```

▸ (1) Spark Jobs

```
Out[90]: (count: 15, mean: 8.0, stdev: 4.320493798938574, max: 15.0, min: 1.0)
```

Command took 0.51 seconds -- by pysparktelugu@gmail.com at 10/27/2020, 7:36:50 PM on datacluster

count()

- Return the number of elements in the dataset.

Cmd 73

```
1  x = sc.parallelize([1,2,3,4])
2  x.count()
```

▸ (1) Spark Jobs

Out[72]: 4

Command took 0.18 seconds -- by pysparktelugu@gmail.com at 10/27/202

Cmd 74

first() Action

- Return the first element of the dataset (similar to take(1)).

Cmd 75

```
1  x = sc.parallelize([1,2,3,4])
2  x.first()
```

▸ (2) Spark Jobs

Out[73]: 1

takeSample(withReplacement, num, [seed])

- Return an array with a random sample of num elements of the dataset, with or without replacement,
- optionally pre-specifying a random number generator seed.
- Return a fixed-size sampled subset of this RDD
- withReplacement whether sampling is done with replacement
- num size of the returned sample
- seed seed for the random number generator
- returns sample of specified size in an array

Cmd 79

```
1  rdd = sc.parallelize(range(0, 10))
2  print(rdd.takeSample(True, 20, 1))
3  print(rdd.takeSample(False, 20, 1))
4
5  print(rdd.takeSample(False, 5, 2))
6
7  print(rdd.takeSample(False, 8, 5))
8
```

▸ (8) Spark Jobs

```
[3, 9, 3, 1, 0, 0, 6, 8, 3, 6, 9, 9, 2, 1, 0, 1, 2, 2, 4, 9]
[6, 8, 9, 7, 5, 3, 0, 4, 1, 2]
[5, 9, 3, 4, 6]
[2, 3, 1, 0, 8, 7, 6, 5]
```

take(n) Action

- Return an array with the first n elements of the dataset.

Cmd 79

```
1  x = sc.parallelize([1,2,3,4])
2  x.take(2)
```

▸ (2) Spark Jobs

```
Out[82]: [1, 2]
```

Command took 0.35 seconds -- by pysparktelugu@gmail.com at 10/27/2020, 7:06:58 PM on datacluster

Cmd 80

countByValue(self)

- Return the count of each unique value in this RDD as a dictionary of (value, count) pairs.

Cmd 81

```
1  x = sc.parallelize([1, 2, 1, 2, 2])
2  y = x.countByValue().items()
3  print(y)
```

▸ (1) Spark Jobs

```
dict_items([(1, 2), (2, 3)])
```

isEmpty()

- Returns true if and only if the RDD contains no elements at all.
- note:: an RDD may be empty even when it has at least 1 partition

Cmd 83

```
1  x =  sc.parallelize(range(10))
2  print(x.isEmpty())
3  y =  sc.parallelize(range(0))
4  print(y.isEmpty())
```

▸ (4) Spark Jobs

```
False
True
```

Command took 0.41 seconds -- by pysparktelugu@gmail.com at 10/27/2020, 6:01:30 PM on data

Cmd 84

keys()

- Return an RDD with the keys of each tuple.

Cmd 85

```
1  x = sc.parallelize([(1, 2), (3, 4)])
2  y = x.keys()
3  print(y.collect())
```

▸ (1) Spark Jobs

```
[1, 3]
```

saveAsTextFile(path, compressionCodecClass=None)

- Save the RDD to the filesystem indicated in the path

Cmd 87

```
1 dbutils.fs.rm("dbfs:/tmp/test_data/",True)
2 x = sc.parallelize([2,4,1])
3 x.saveAsTextFile("dbfs:/tmp/test_data/saveAs")
4 y = sc.textFile("dbfs:/tmp/test_data/saveAs")
5 print(y.collect())
```

▸ (2) Spark Jobs

```
['2', '4', '1']
```

saveAsPickleFile(self, path, batchSize=10)

- Save this RDD as a SequenceFile of serialized objects. The serializer
- used is :class: `pyspark.serializers.PickleSerializer` , default batch size is 10.

Cmd 89

```
1  dbutils.fs.rm("dbfs:/tmp/test_data/picklefile/",True)
2  x = sc.parallelize([2,4,1])
3  x.saveAsPickleFile("dbfs:/tmp/test_data/picklefile")
4  y = sc.pickleFile("dbfs:/tmp/test_data/picklefile")
5  print(y.collect())
6
7
```

▸ (2) Spark Jobs

```
[2, 4, 1]
```

Command took 2.36 seconds -- by pysparktelugu@gmail.com at 10/27/2020, 6:01:30 PM on datacluster

STDEV()

- Return the standard deviation of the items in the RDD

Cmd 92

```
1  x = sc.parallelize([2,4,1])
2  y = x.stdev()
3  print(x.collect())
4  print(y)
```

▸ (2) Spark Jobs

```
[2, 4, 1]
1.247219128924647
```

MIN()

- Return the MIN value of the items in the RDD

Cmd 94

```
1  x = sc.parallelize([2,4,1,3,5,6,7,-5,-8])
2  y = x.min()
3  print(x.collect())
4  print(y)
```

▸ (2) Spark Jobs

```
[2, 4, 1, 3, 5, 6, 7, -5, -8]
-8
```

MAX()

- REturn the MAX Value of the items in the RDD

Cmd 96

```
1  x = sc.parallelize([2,4,1])
2  y = x.max()
3  print(x.collect())
4  print(y)
```

▸ (2) Spark Jobs

```
[2, 4, 1]
4
```

MEAN()

- Return the mean of the items in the RDD

Cmd 98

```
1  x = sc.parallelize([2,4,1])
2  y = x.mean()
3  print(x.collect())
4  print(y)
```

▸ (2) Spark Jobs

```
[2, 4, 1]
2.3333333333333335
```

Command took 0.20 seconds -- by pysparktelugu@gmail.com at 10/27/2020,

SUM()

- Return the Sum of the items in the RDD

Cmd 100

```
1 x = sc.parallelize([2,4,1])
2 y = x.sum()
3 print(x.collect())
4 print(y)
```

▸ (2) Spark Jobs

```
[2, 4, 1]
7
```

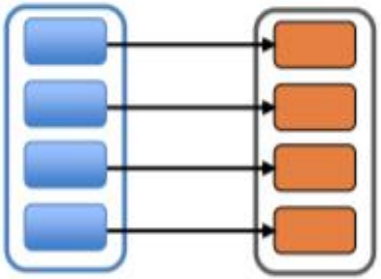
vs

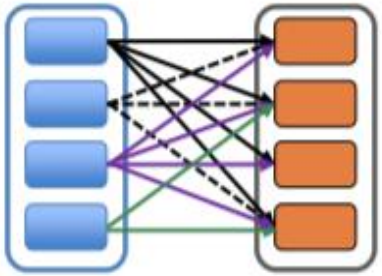
narrow

wide

each partition of the parent RDD is used by at most one partition of the child RDD

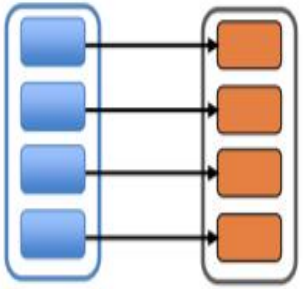
multiple child RDD partitions may depend on a single parent RDD partition



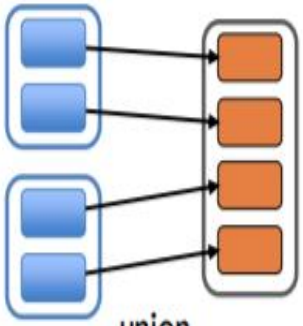


narrow

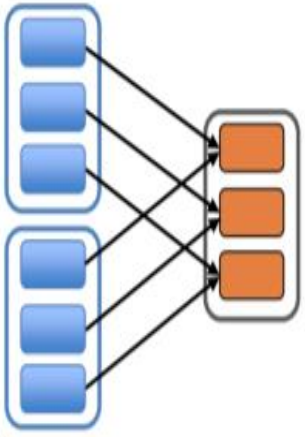
each partition of the parent RDD is used by at most one partition of the child RDD



map, filter



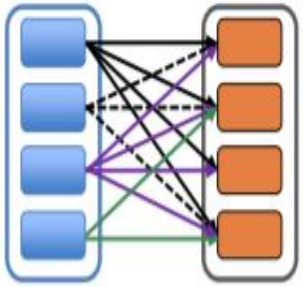
union



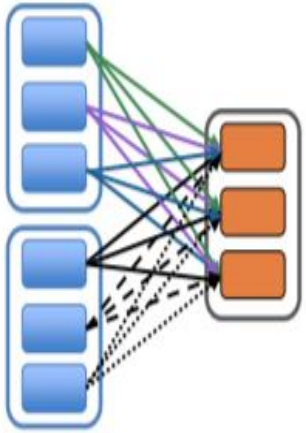
join w/ inputs co-partitioned

wide

multiple child RDD partitions may depend on a single parent RDD partition



groupByKey



join w/ inputs not co-partitioned

Lambda Function

- A lambda function is a small anonymous function.
- A lambda function can take any number of arguments, but can only have one expression.

Cmd 2

```
1 x = lambda a : a + 10
2 print(x(5))
3 print(x(10))
```

Cmd 3

```
1 x = lambda a, b : a * b
2 print(x(5, 6))
```

Creating RDD using SparkContext

- Applying MAP Transformation in RDD.
- MAP(func)
- Return a new distributed dataset formed by passing each element of the source through a function func.

Cmd 37

```
1 x=sc.parallelize([1,2,3,4,5,6,7])
2 y=x.map(lambda a: a+10 )
3 print(x.collect())
4 print(y.collect())
```

▸ (2) Spark Jobs

```
[1, 2, 3, 4, 5, 6, 7]
[11, 12, 13, 14, 15, 16, 17]
```

Command took 4.06 seconds -- by pysparktelugu@gmail.com at 12/15/2020, 6:57:25 AM on datacluster

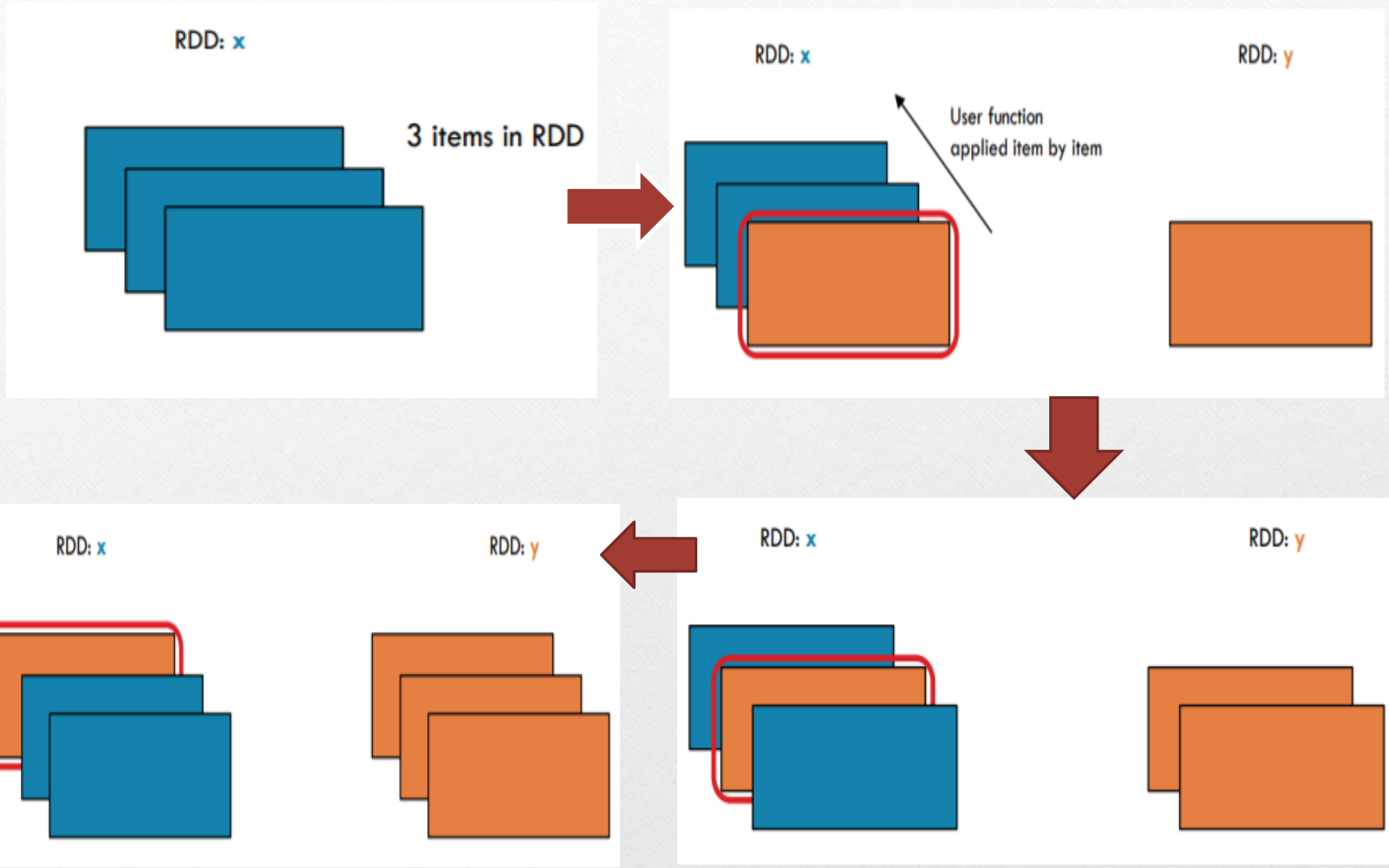
Cmd 38

```
1 print("X RDD VAlues :",x.collect())
2 print("Y RDD Values after applying map and lambda transformation",y.collect())
```

▸ (2) Spark Jobs

```
X RDD VAlues : [1, 2, 3, 4, 5, 6, 7]
Y RDD Values after applying map and lambda transformation [11, 12, 13, 14, 15, 16, 17]
```

MAP Transformation



Filter Transformation in RDD

filter(func) Transformation

- Return a new dataset formed by selecting those elements of the source on which func returns true.

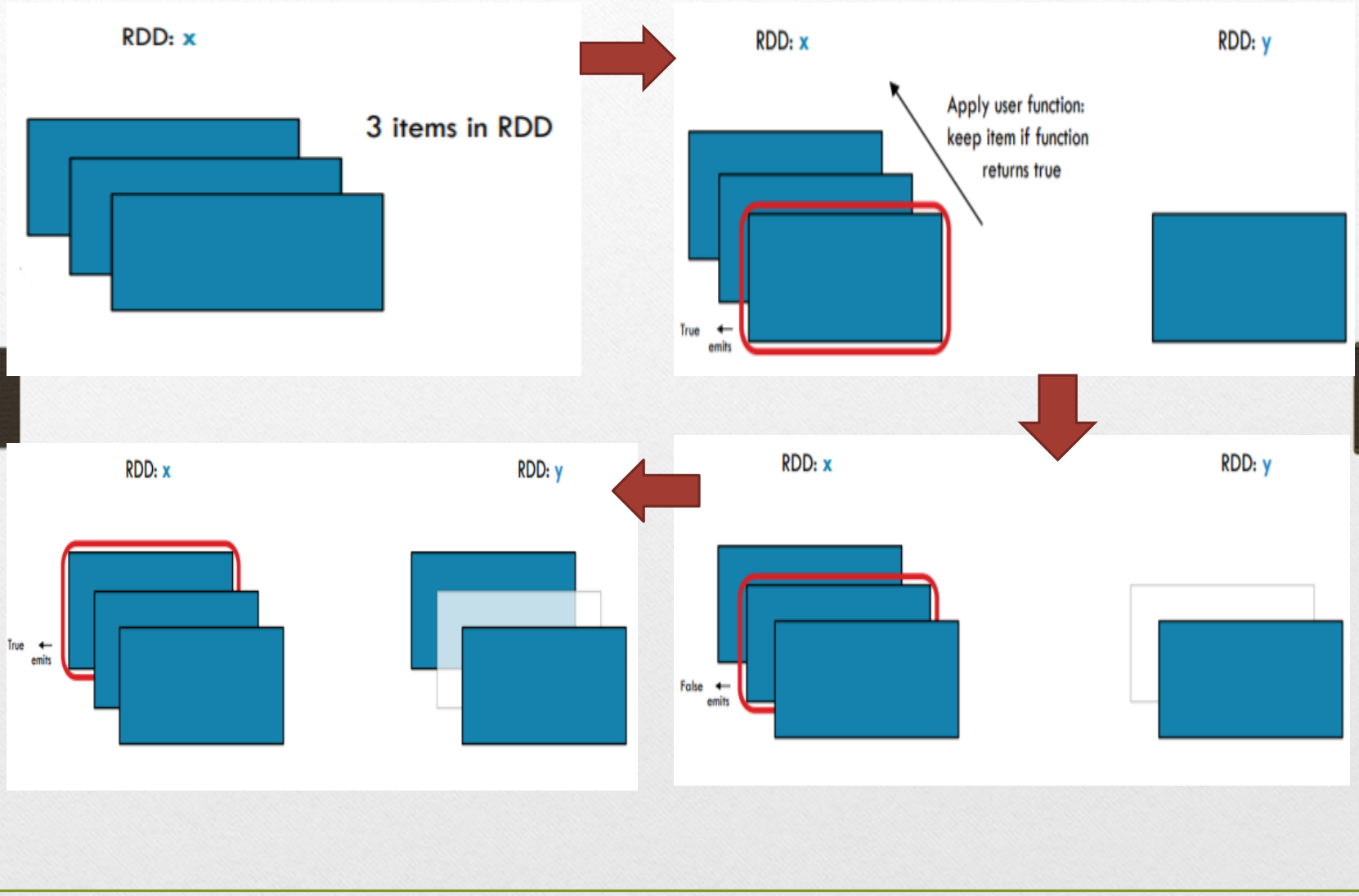
Cmd 5

```
1 x = sc.parallelize([1,2,3])
2 y = x.filter(lambda x: x%2 == 1) #keep odd values
3 print('RDD X Values Before Filter : ',x.collect())
4 print('RDD Y Values After apllying Filter in X RDD : ',y.collect())
```

▸ (2) Spark Jobs

```
RDD X Values Before Filter :  [1, 2, 3]
RDD Y Values After apllying Filter in X RDD :  [1, 3]
```

Command took 0.34 seconds -- by pysparktelugu@gmail.com at 10/23/2020, 5:32:43 PM on datacluster


RDD: x
3 items in RDD
RDD: x
RDD: y
Apply user function:
keep item if function
returns true
True
emits
RDD: x
RDD: y
False
emits
RDD: x
RDD: y
True
emits

flatMap Transformation in RDD

flatMap(func) Transformation

- Similar to map, but each input item can be mapped to 0 or more output items
- (so func should return a Seq rather than a single item).

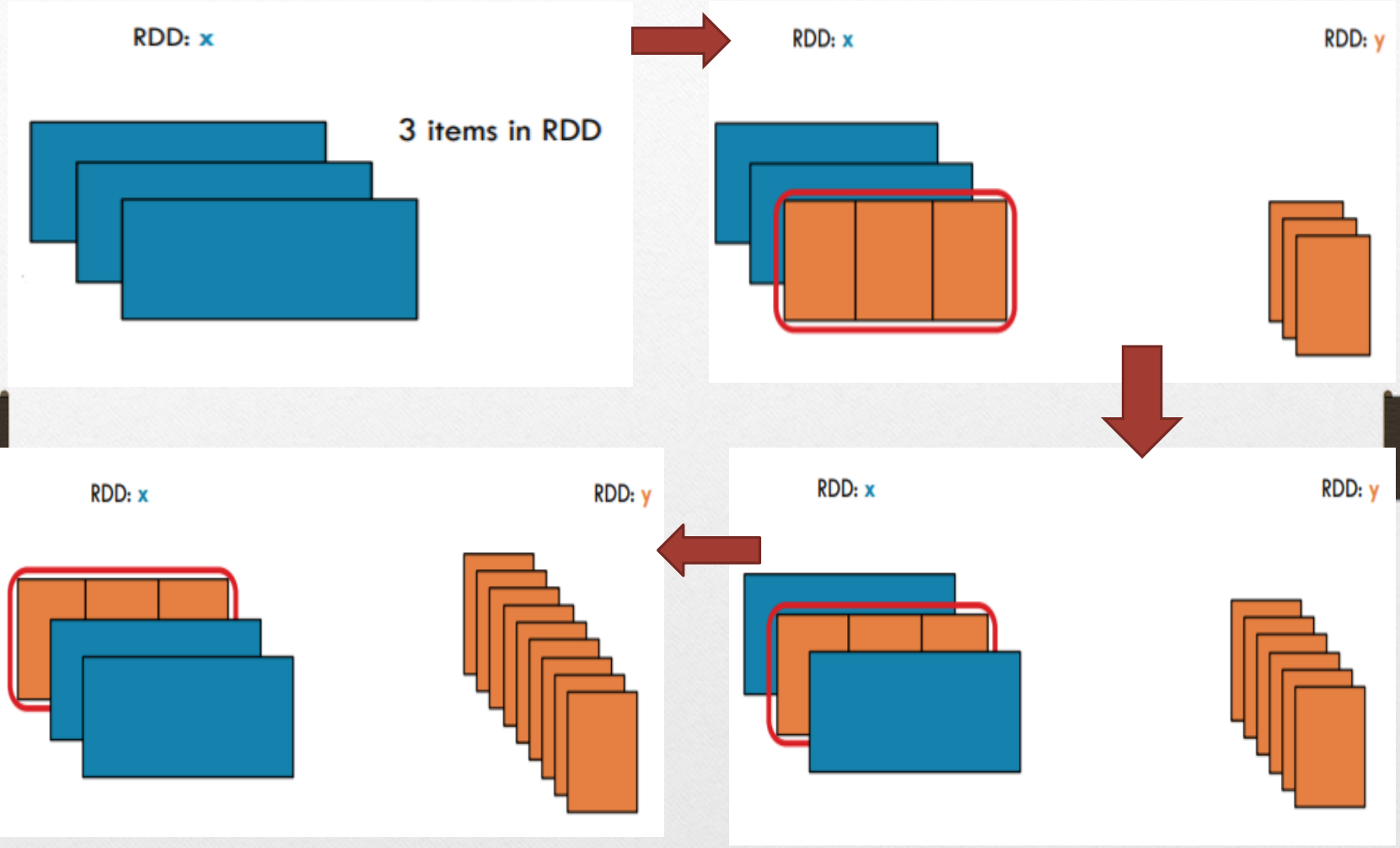
Cmd 7

```
1 x = sc.parallelize([1,2,3])
2 y = x.flatMap(lambda x: (x, x*100, 66))
3 print(x.collect())
4 print(y.collect())
```

▸ (2) Spark Jobs

```
[1, 2, 3]
[1, 100, 66, 2, 200, 66, 3, 300, 66]
```

Command took 0.30 seconds -- by pysparktelugu@gmail.com at 10/23/2020, 5:42:40 PM on da


RDD: x
3 items in RDD
RDD: x
RDD: y
RDD: x
RDD: y
RDD: x
RDD: y

groupBy() Transformation in RDD

groupBy(func) Transformation in RDD

- Group the data in the original RDD. Create pairs where the key is the output of
- a user function, and the value is all items for which the function yields this key

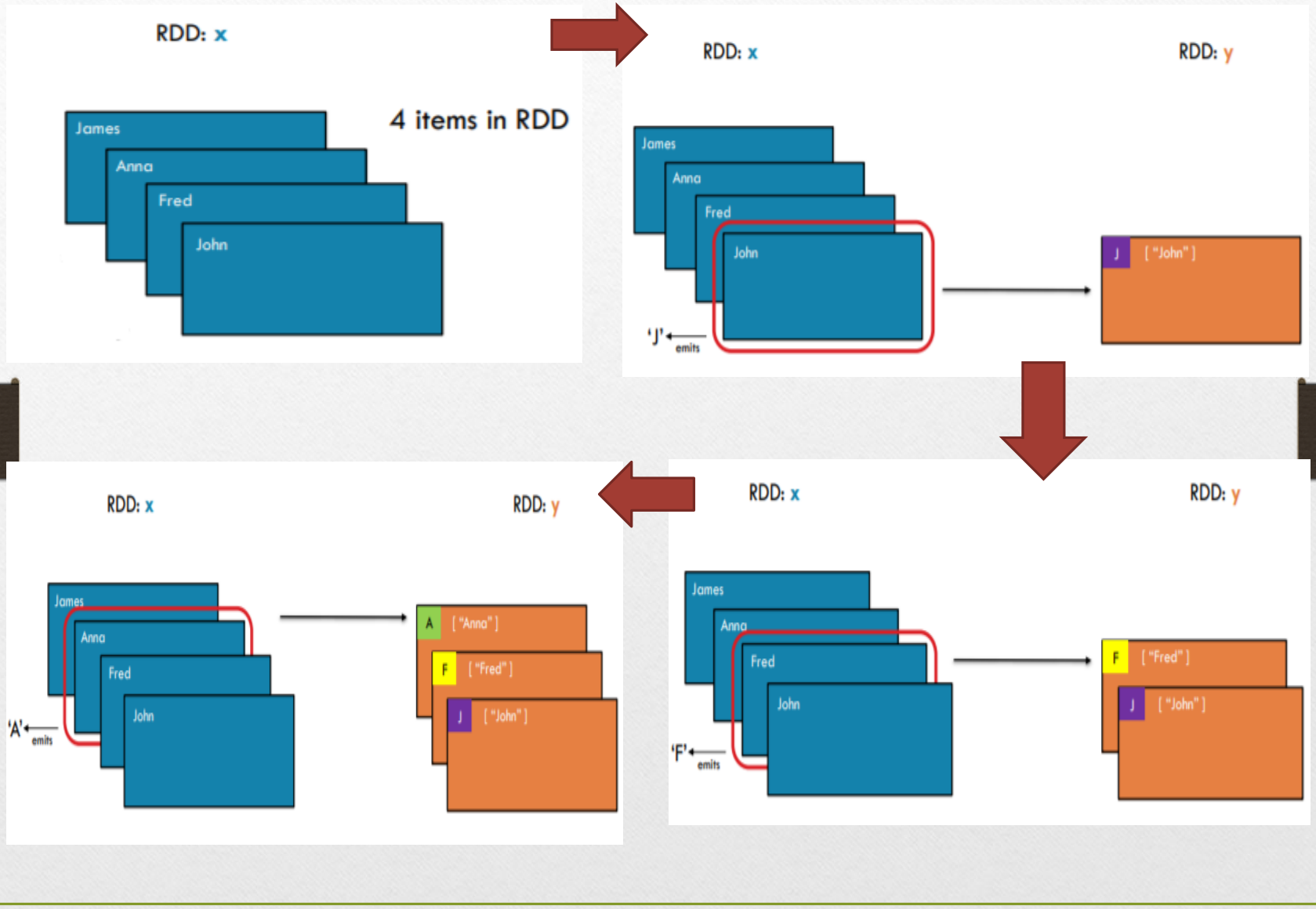
Cmd 9

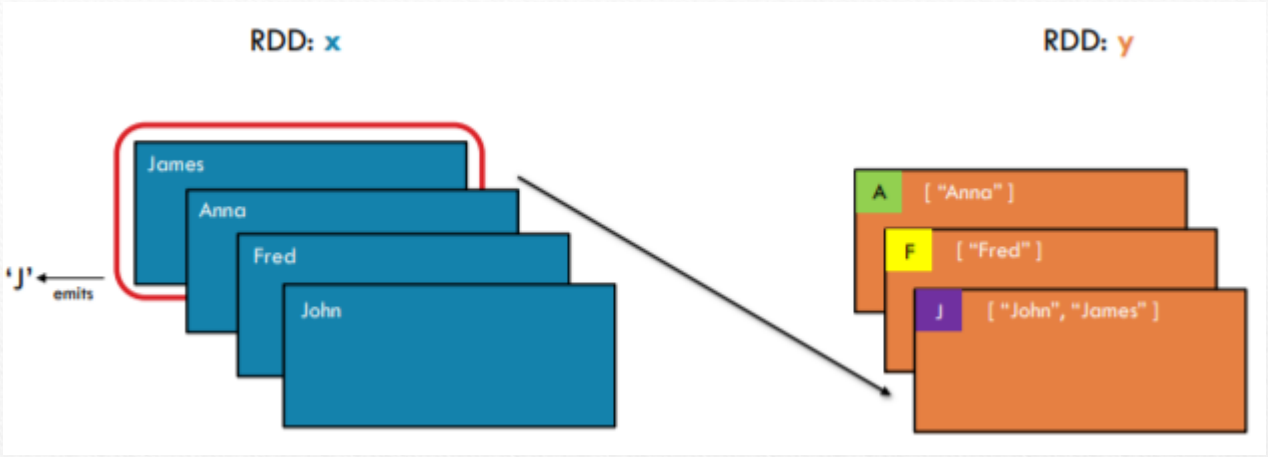
```
1  x = sc.parallelize(['John', 'Fred', 'Anna', 'James'])
2  y = x.groupBy(lambda w: w[0])
3  print(x.collect())
4  print([(k, list(v)) for (k, v) in y.collect()])
```

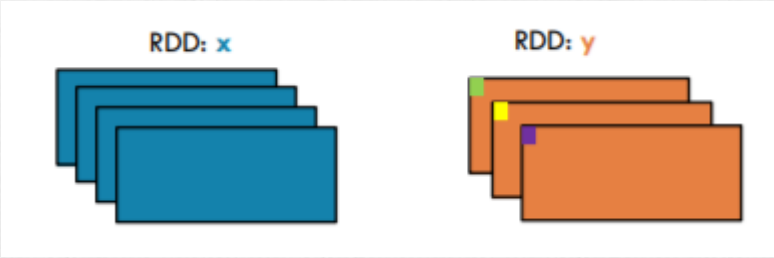
▸ (2) Spark Jobs

```
['John', 'Fred', 'Anna', 'James']
[('J', ['John', 'James']), ('F', ['Fred']), ('A', ['Anna'])]
```

Command took 0.78 seconds -- by pysparktelugu@gmail.com at 10/23/2020, 6:15:30 PM on datac


RDD: x
4 items in RDD
James
Anna
Fred
John
RDD: x
RDD: y
James
Anna
Fred
John
'J' emits
J ["John"]
RDD: x
RDD: y
James
Anna
Fred
John
'F' emits
F ["Fred"]
J ["John"]
RDD: x
RDD: y
James
Anna
Fred
John
'A' emits
A ["Anna"]
F ["Fred"]
J ["John"]


RDD: x
RDD: y
James
Anna
Fred
John
'J'
emits
A
["Anna"]
F
["Fred"]
J
["John", "James"]


RDD: x
RDD: y

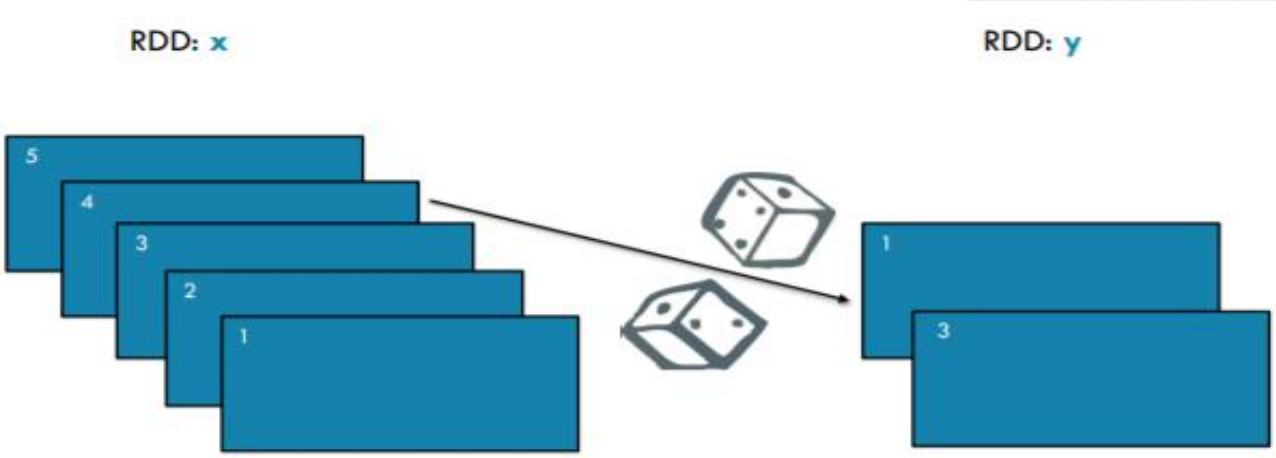
Sample Transformation

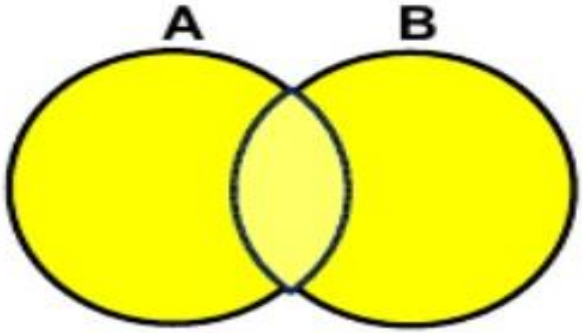
* Return a sampled subset of this RDD.

```
1  x = sc.parallelize([1,2, 3, 4, 5])
2  y = x.sample(False, 0.4, 15)
3  print(x.collect())
4  print(y.collect())
```

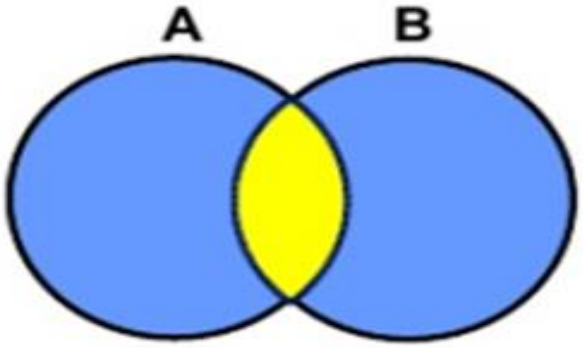
▸ (2) Spark Jobs

```
[1, 2, 3, 4, 5]
[1, 3]
```

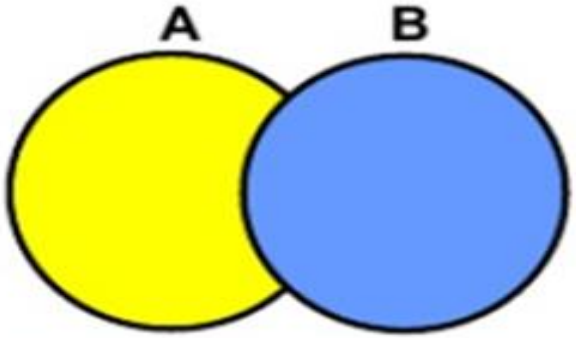




UNION



INTERSECTION



SUBTRACT

Union(DataSet) Transformation

- Return a new dataset that contains the union of the elements in the source dataset and the argument.
- glom() Return an RDD created by coalescing all elements within each partition into an array

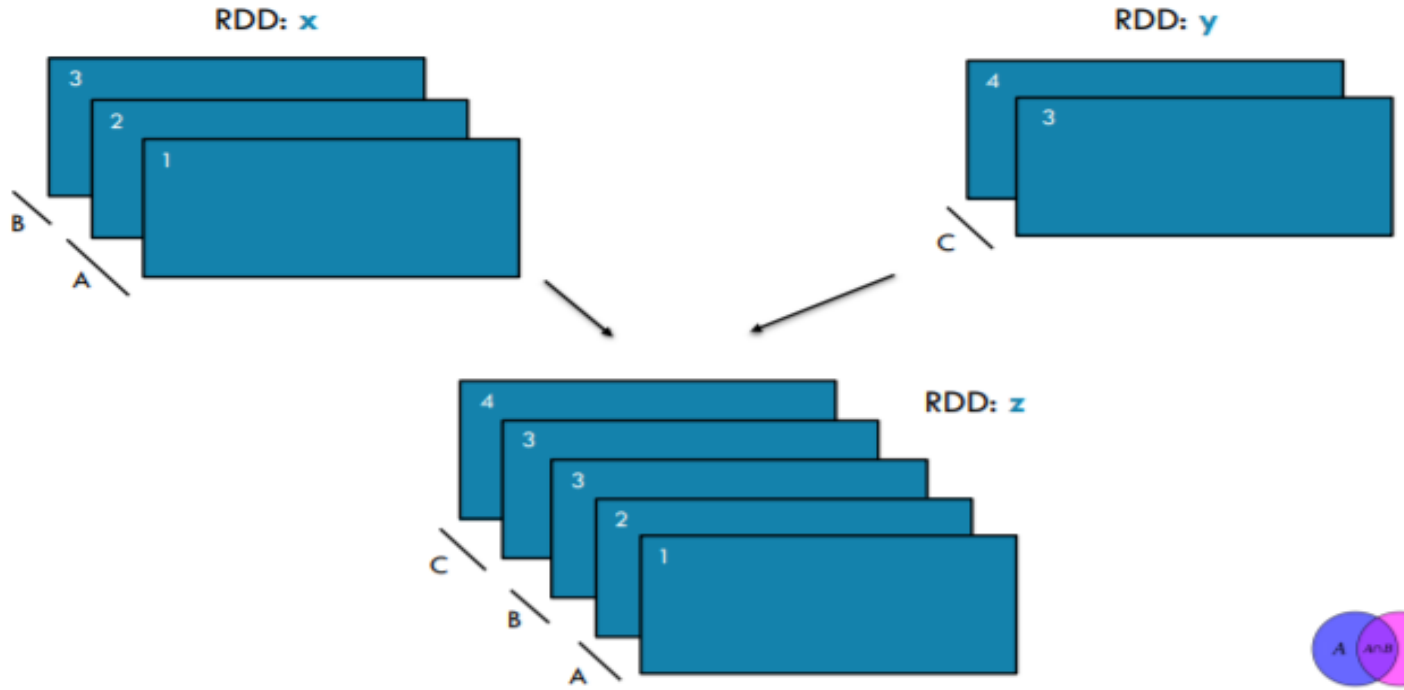
Cmd 41

```
1 x = sc.parallelize([1,2,3], 2)
2 y = sc.parallelize([3,4], 1)
3 z = x.union(y)
4 print(z.collect())
5 print(z.glom().collect())
```

▸ (2) Spark Jobs

```
[1, 2, 3, 3, 4]
[[1], [2, 3], [3, 4]]
```

UNION




```
1  x = sc.parallelize([1,2,3], 2)
2  y = sc.parallelize([3,4], 1)
3  z = x.union(y)
4  print(z.glom().collect())
```

▸ (1) Spark Jobs

```
[[1], [2, 3], [3, 4]]
```

intersection(otherDataset)

- Return a new RDD that contains the intersection of elements in the source dataset and the argument.

Cmd 43

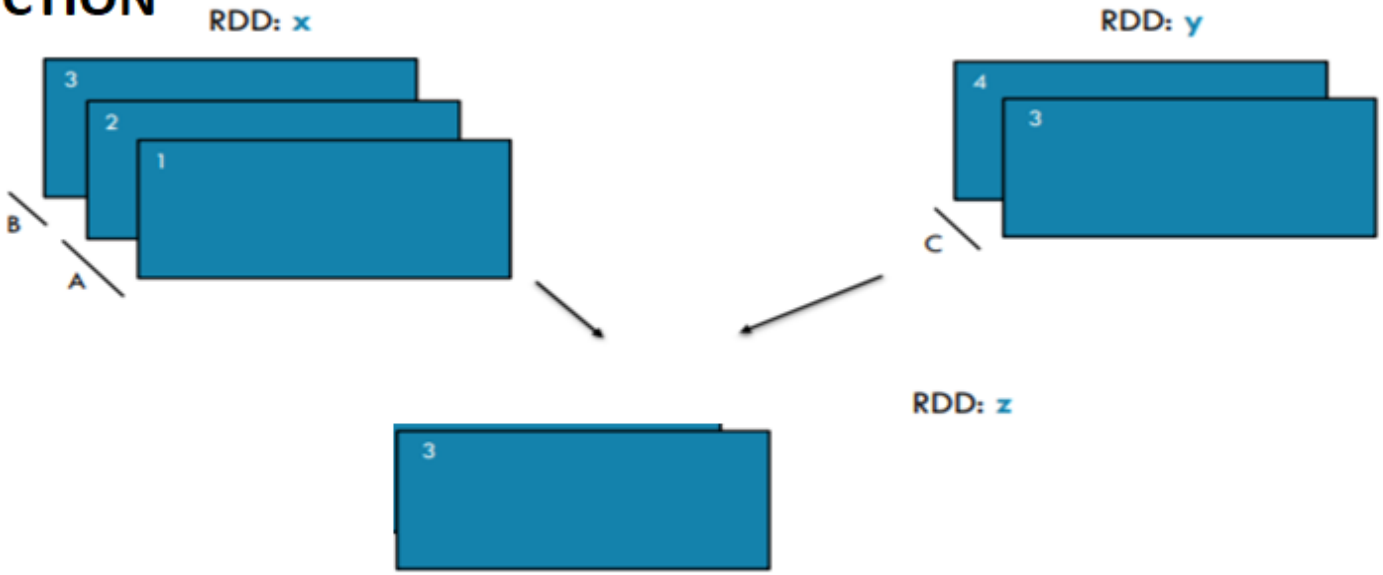
```
1 x = sc.parallelize([1,2,3,4,5], 2)
2 y = sc.parallelize([3,4,5,6,7], 1)
3 z = x.intersection(y)
4 print(z.collect())
5 print(z.glom().collect())
```

▸ (2) Spark Jobs

```
[3, 4, 5]
[[3], [4], [5]]
```

Command took 1.18 seconds -- by pysparktelugu@gmail.com at 10/25/2020, 2:13:55 PM on datacluster

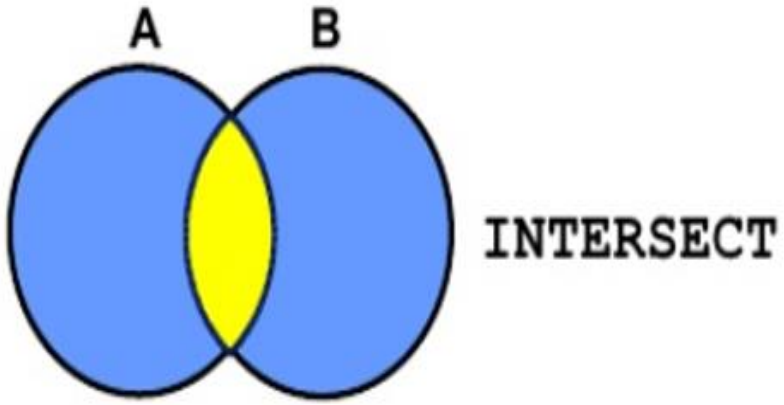
INTERSECTION




```
1  x = sc.parallelize([1,2,3], 2)
2  y = sc.parallelize([3,4], 1)
3  z = x.intersection(y)
4  print(z.collect())
5  print(z.glom().collect())
```

▸ (2) Spark Jobs

```
[3]
[[3], [], []]
```



substract(otherdataset) Transformation

- It returns an RDD that has only value present in the first RDD and not in second RDD.
- its returns if first RDD is having any duplicates. its wont remove any duplicate

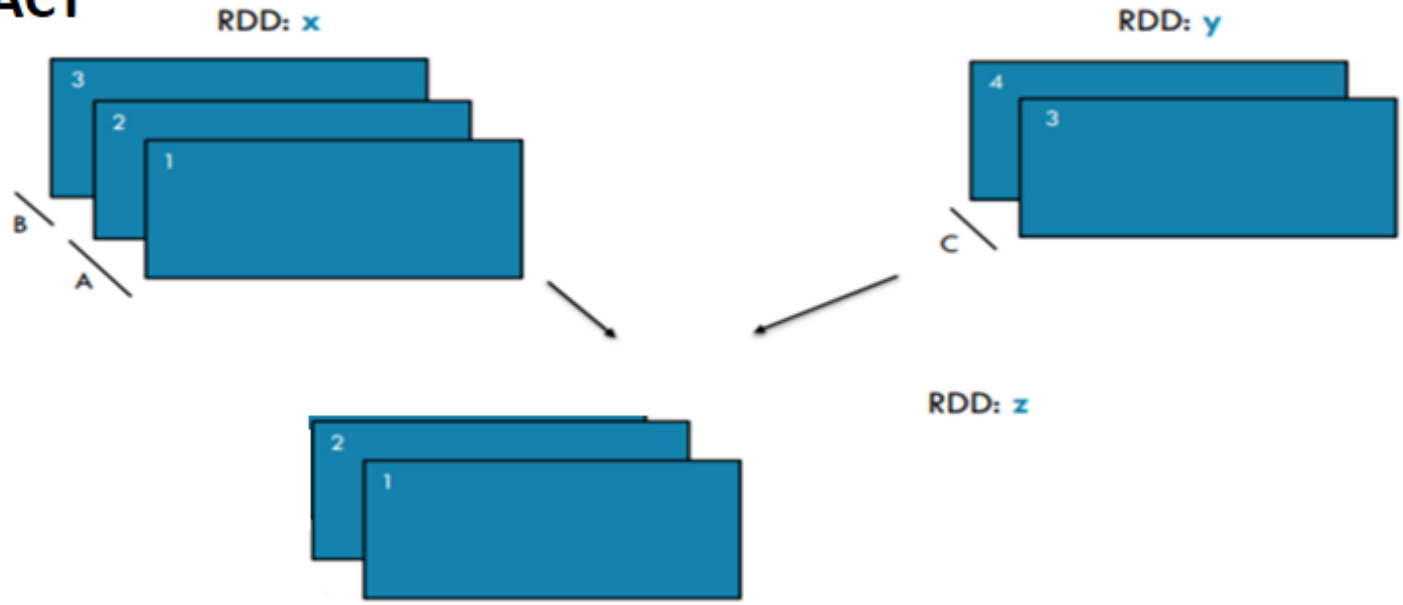
Cmd 46

```
1 x = sc.parallelize([1,2,2,3,3,4,5,7,8], 2)
2 y = sc.parallelize([3,4,3,4,5,6], 1)
3 z = x.subtract(y)
4 print(z.collect())
5 print(z.glom().collect())
```

▸ (2) Spark Jobs

```
[1, 7, 2, 2, 8]
[[], [1, 7], [2, 2, 8]]
```

SUBTRACT

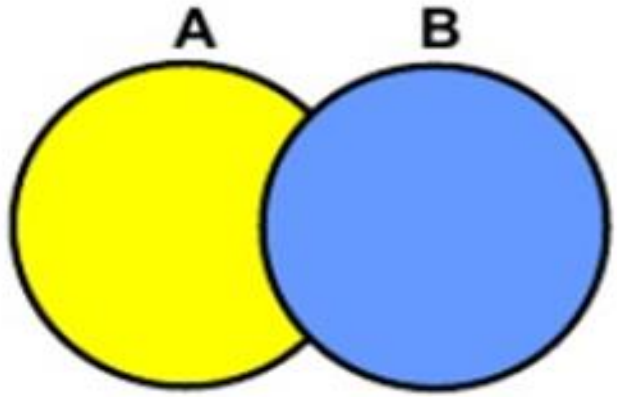



```
1  x = sc.parallelize([1,2,3], 2)
2  y = sc.parallelize([3,4], 1)
3  z = x.subtract(y)
4  print(z.collect())
5  print(z.glom().collect())
```

▸ (2) Spark Jobs

```
[1, 2]
[[], [1], [2]]
```

A - B

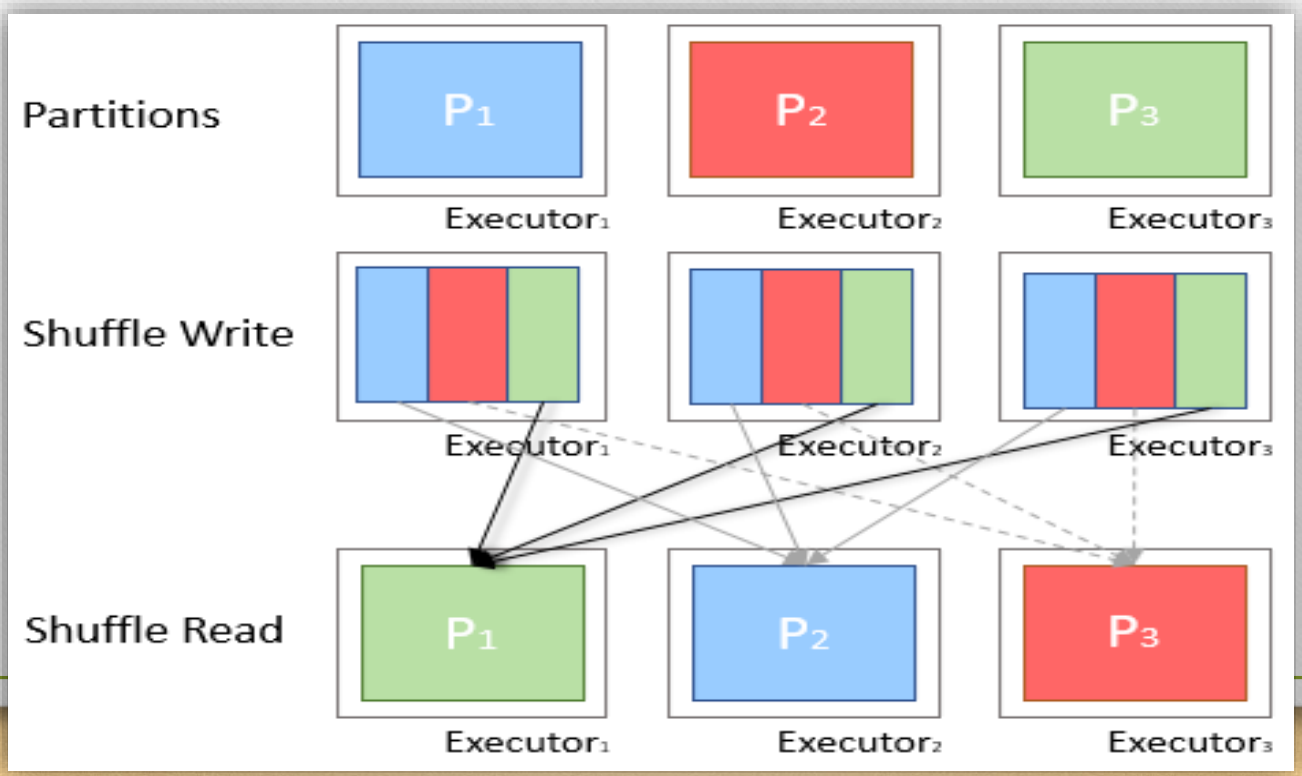


What is Shuffling?

A shuffle occurs when data is rearranged between partitions. This is required when a transformation requires information from other partitions, such as summing all the values in a column. Spark will gather the required data from each partition and combine it into a new partition, likely on a different executor. During a shuffle, data is written to disk and transferred across the network, halting Spark's ability to do processing in-memory and causing a performance bottleneck. Consequently we want to try to reduce the number of shuffles being done or reduce the amount of data being shuffled.

Shuffle read: Total shuffle bytes and records read, includes both data read locally and data read from remote executors

Shuffle write: Bytes and records written to memory(disk) in order to be read by a shuffle in a future stage



Distinct Transformation

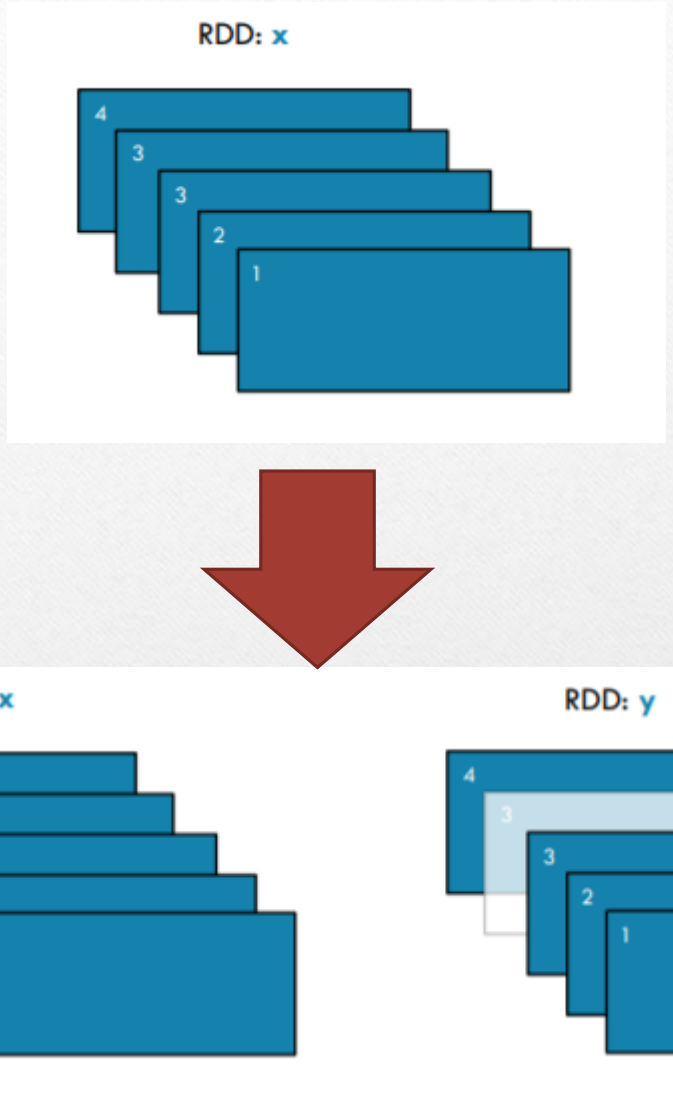
- Return a new RDD containing distinct items from the original RDD (omitting all duplicates)

Cmd 48

```
1 x = sc.parallelize([1,2,3,3,4])
2 y = x.distinct()
3 print(y.collect())
4
```

▸ (1) Spark Jobs

```
[1, 2, 3, 4]
```


RDD: x
4
3
3
2
1
RDD: x
4
3
3
2
1
RDD: y
4
3
3
2
1

Coalesce Transformation

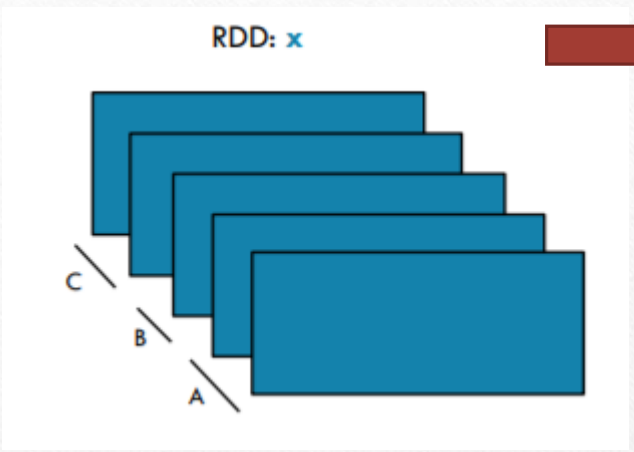
- Return a new RDD which is reduced to a smaller number of partitions

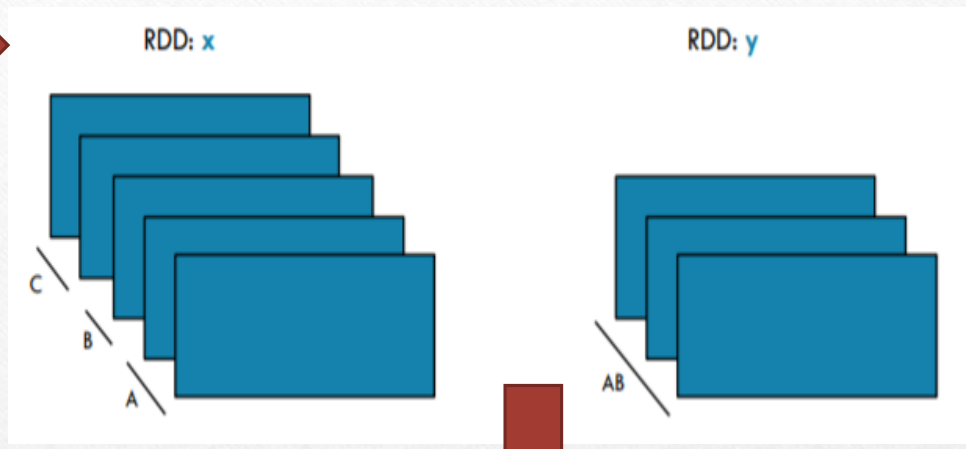
Cmd 52

```
1 x = sc.parallelize([1, 2, 3, 4, 5], 3)
2 y = x.coalesce(2)
3 print(x.glom().collect())
4 print(y.glom().collect())
5
```

▸ (2) Spark Jobs

```
[[1], [2, 3], [4, 5]]
[[1], [2, 3, 4, 5]]
```

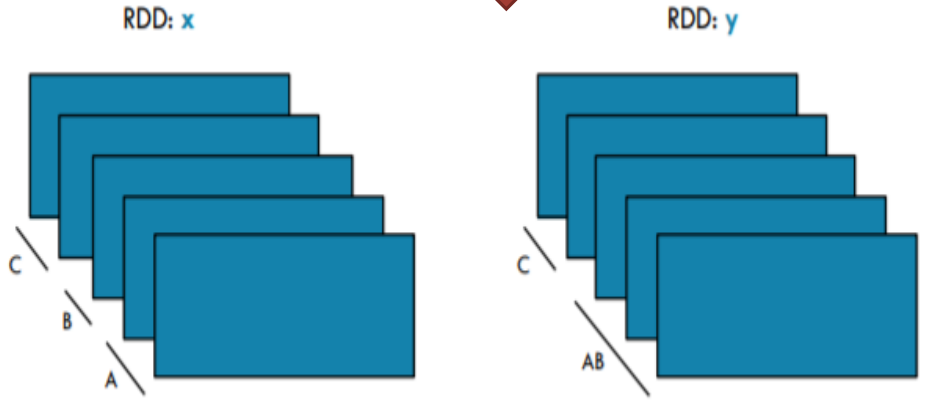





```
1 x = sc.parallelize([1, 2, 3, 4, 5], 3)
2 y = x.coalesce(2)
3 print(x.glom().collect())
4 print(y.glom().collect())
```

▸ (2) Spark Jobs

```
[[1], [2, 3], [4, 5]]
[[1], [2, 3, 4, 5]]
```



repartition(numPartitions) Transformation

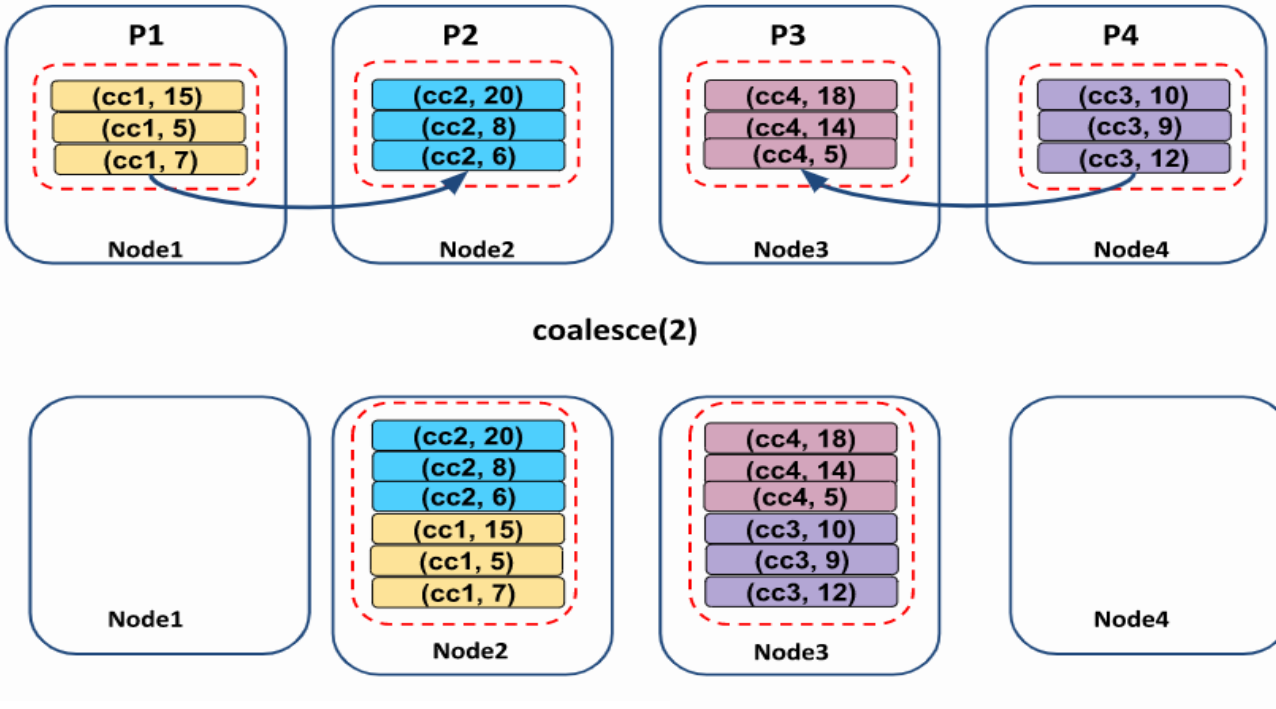
- Reshuffle the data in the RDD randomly to create either more or fewer partitions and balance it across them.
- This always shuffles all data over the network. Repartition by column We can also repartition by columns.
- syntax: `repartition(numPartitions, *cols)`

Cmd 54

```
1 x = sc.parallelize([1, 2, 3, 4, 5,6,7,8,9,10,11,12,13,14,15], 3)
2 y = x.repartition(6)
3 print(x.glom().collect())
4 print(y.glom().collect())
5 print(y.getNumPartitions())
```

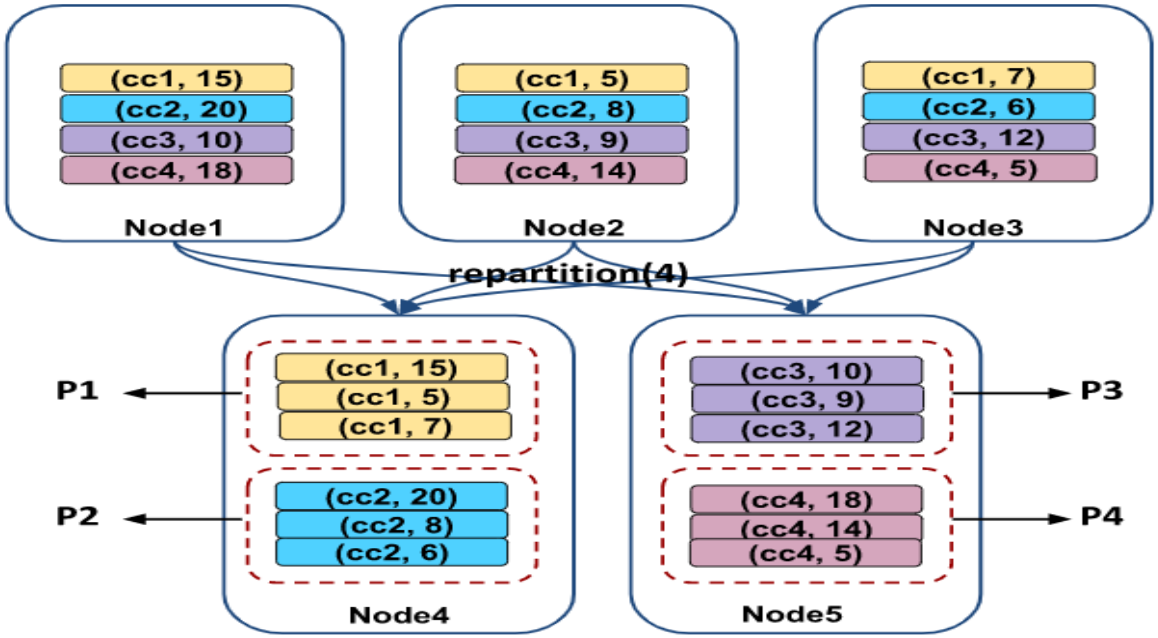
▸ (2) Spark Jobs

```
[[1, 2, 3, 4, 5], [6, 7, 8, 9, 10], [11, 12, 13, 14, 15]]
[[], [1, 2, 3, 4, 5], [], [], [11, 12, 13, 14, 15], [6, 7, 8, 9, 10]]
6
```



COALESCE() Just adjust the data in existing partitions

REPARTITION() will delete exsiting partitions And it will recreate all partitions



PartitionBy Transformation

- Return a new RDD with the specified number of partitions,
- placing original items into the partition returned by a user supplied function

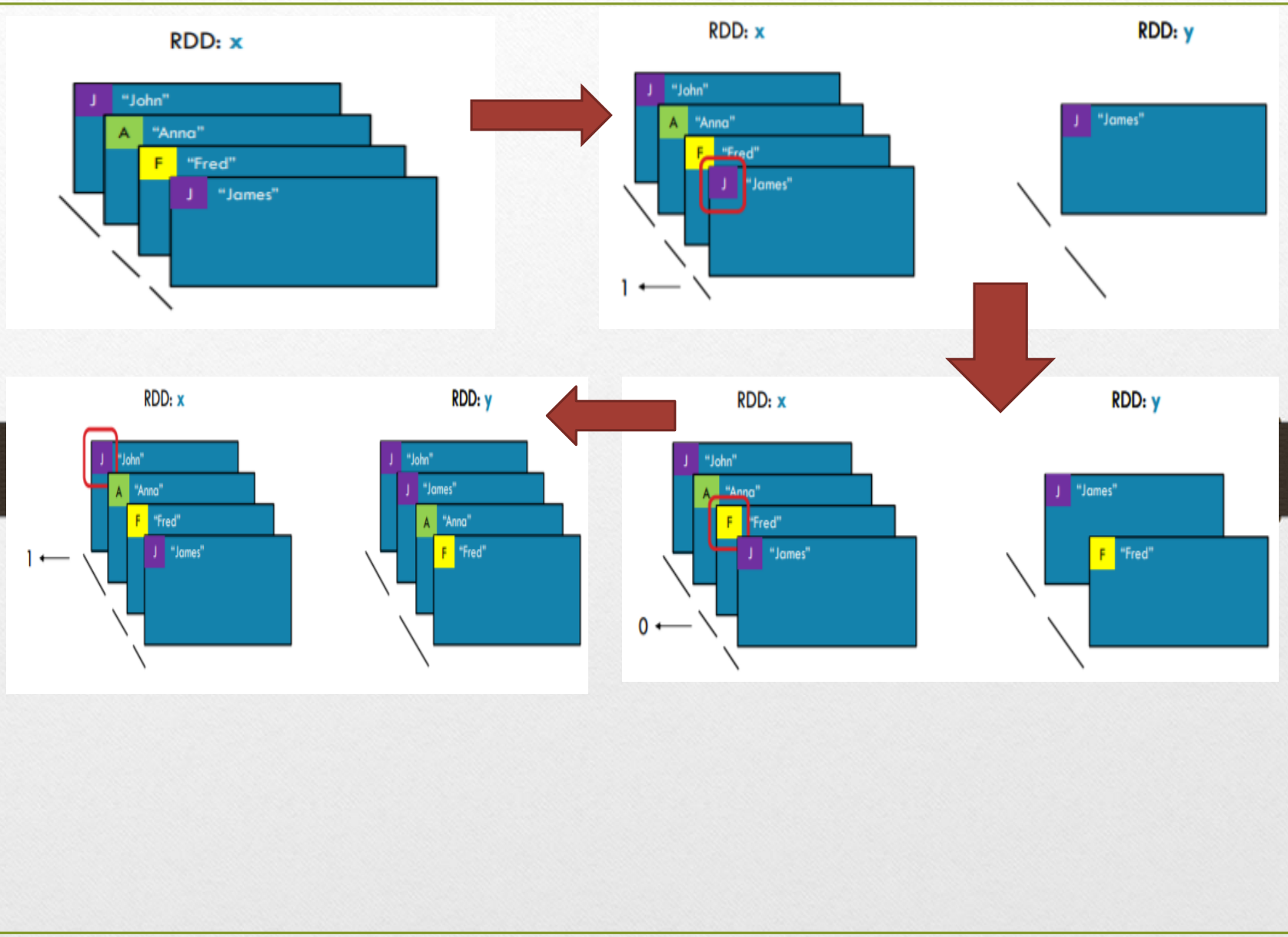
Cmd 56

```
1 x = sc.parallelize([('J','James'),('F','Fred'),('A','Anna'),('J','John')], 3)
2 y = x.partitionBy(2, lambda w: 0 if w[0] < 'H' else 1)
3 print (x.glom().collect())
4 print (y.glom().collect())
5 print('X RDD No.OF Partitiones : ',x.getNumPartitions())
6 print('Y RDD No.OF Partitiones : ',y.getNumPartitions())
```

▸ (2) Spark Jobs

```
[[('J', 'James')], [('F', 'Fred')], [('A', 'Anna'), ('J', 'John')]]
[[('F', 'Fred'), ('A', 'Anna')], [('J', 'James'), ('J', 'John')]]
X RDD No.OF Partitiones :  3
Y RDD No.OF Partitiones :  2
```

Command took 0.41 seconds -- by pysparktelugu@gmail.com at 10/25/2020, 2:50:31 PM on datacluster


RDD: x
J "John"
A "Anna"
F "Fred"
J "James"
RDD: x
J "John"
A "Anna"
F "Fred"
J "James"
1
RDD: y
J "James"
RDD: x
J "John"
A "Anna"
F "Fred"
J "James"
0
RDD: y
J "James"
F "Fred"
RDD: x
J "John"
A "Anna"
F "Fred"
J "James"
1
RDD: y
J "John"
J "James"
A "Anna"
F "Fred"

ZIP Transformation

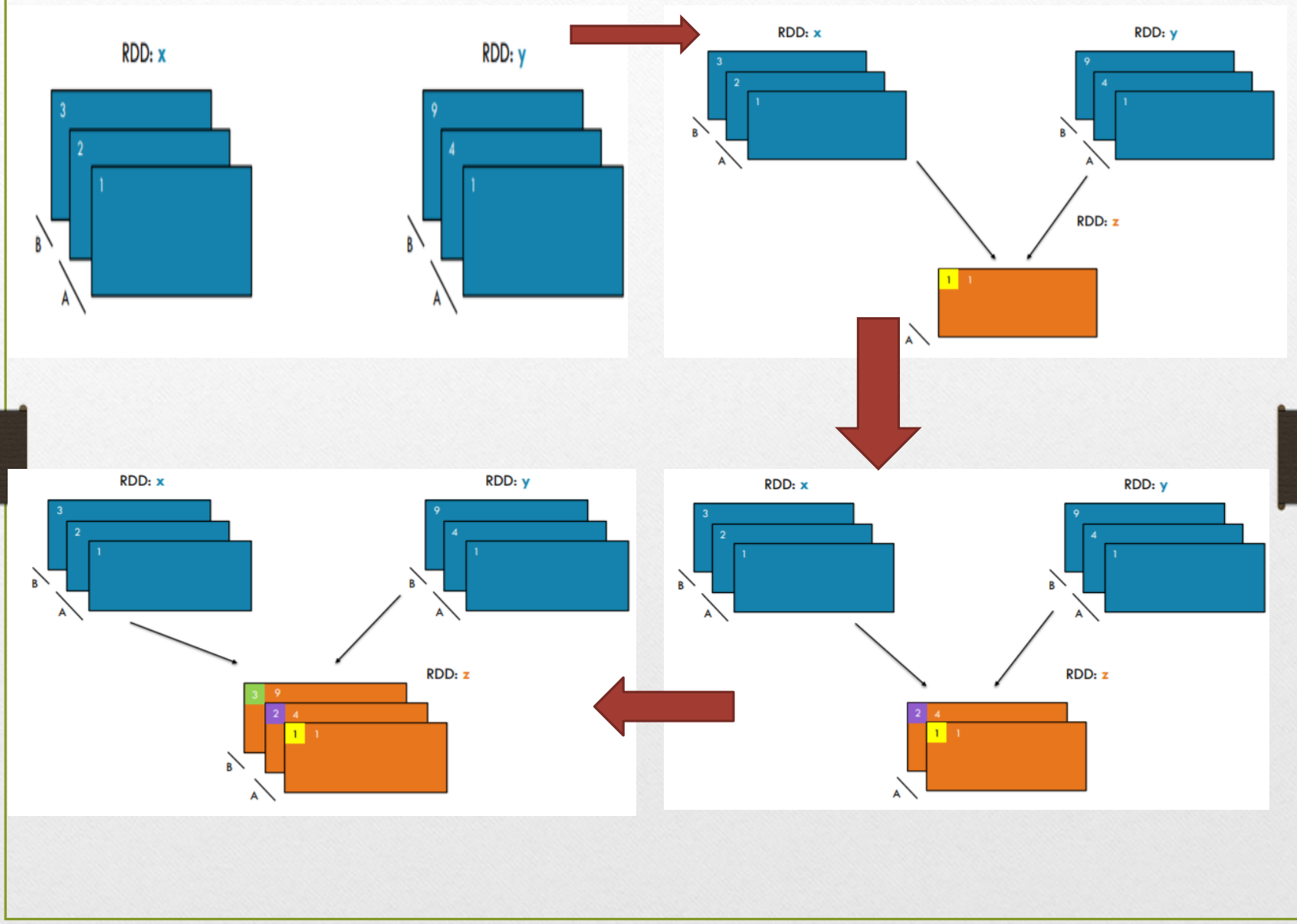
- Return a new RDD containing pairs whose key is the item in the original RDD, and whose
- value is that item's corresponding element (same partition, same index) in a second RDD

Cmd 58

```
1 x = sc.parallelize([1, 2, 3])
2 y = x.map(lambda n:n*n)
3 z = x.zip(y)
4 print(z.collect())
```

▸ (1) Spark Jobs

```
[(1, 1), (2, 4), (3, 9)]
```


RDD: x
RDD: y
RDD: z
B
A

groupByKey Transformation in RDD

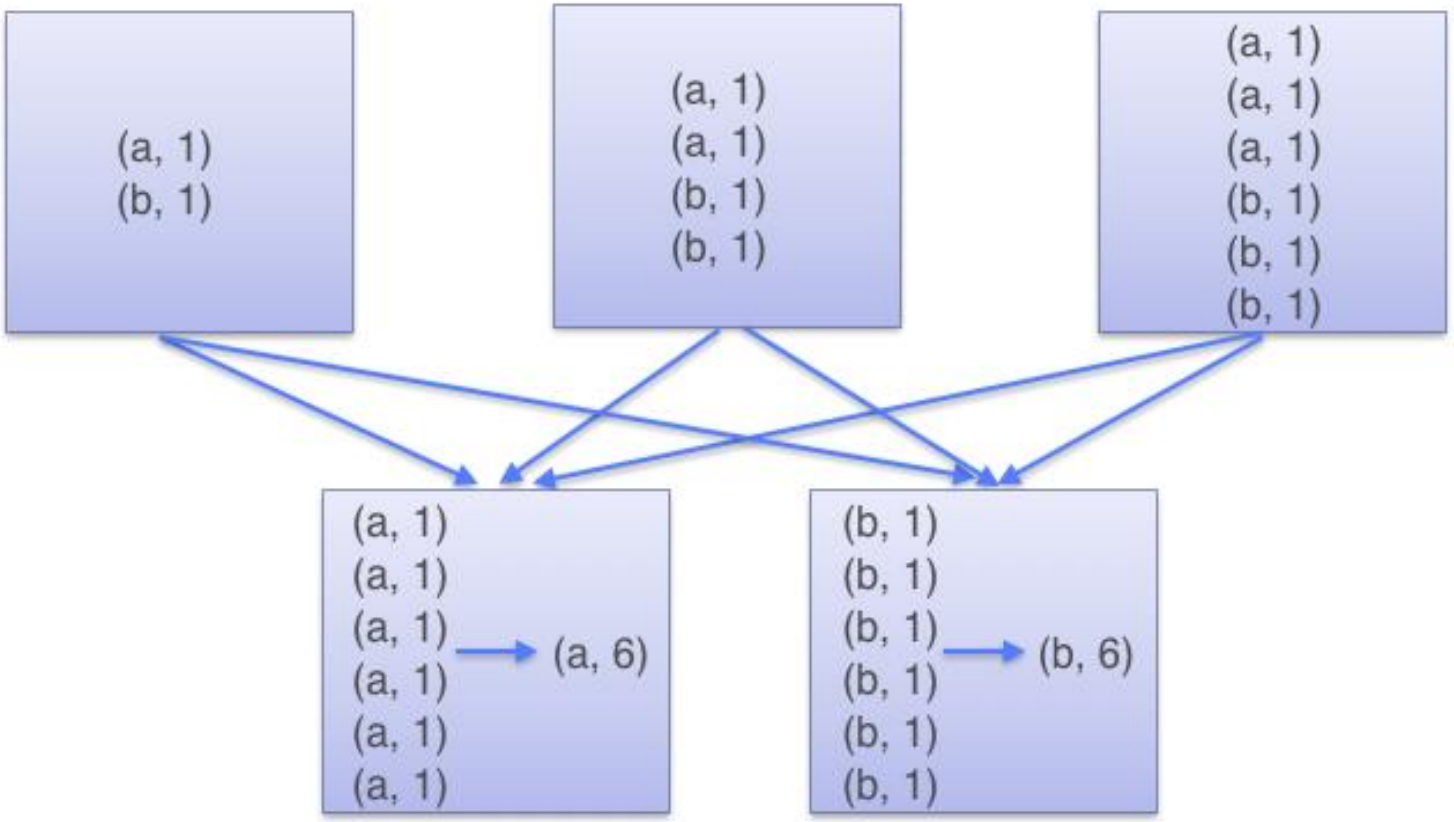
* When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable<V>) pairs.
* Note: If you are grouping in order to perform an aggregation (such as a sum or average) over each key, using reduceByKey or aggregateByKey will yield much better performance.
* Note: By default, the level of parallelism in the output depends on the number of partitions of the parent RDD. You can pass an optional numPartitions argument to set a different number of tasks.

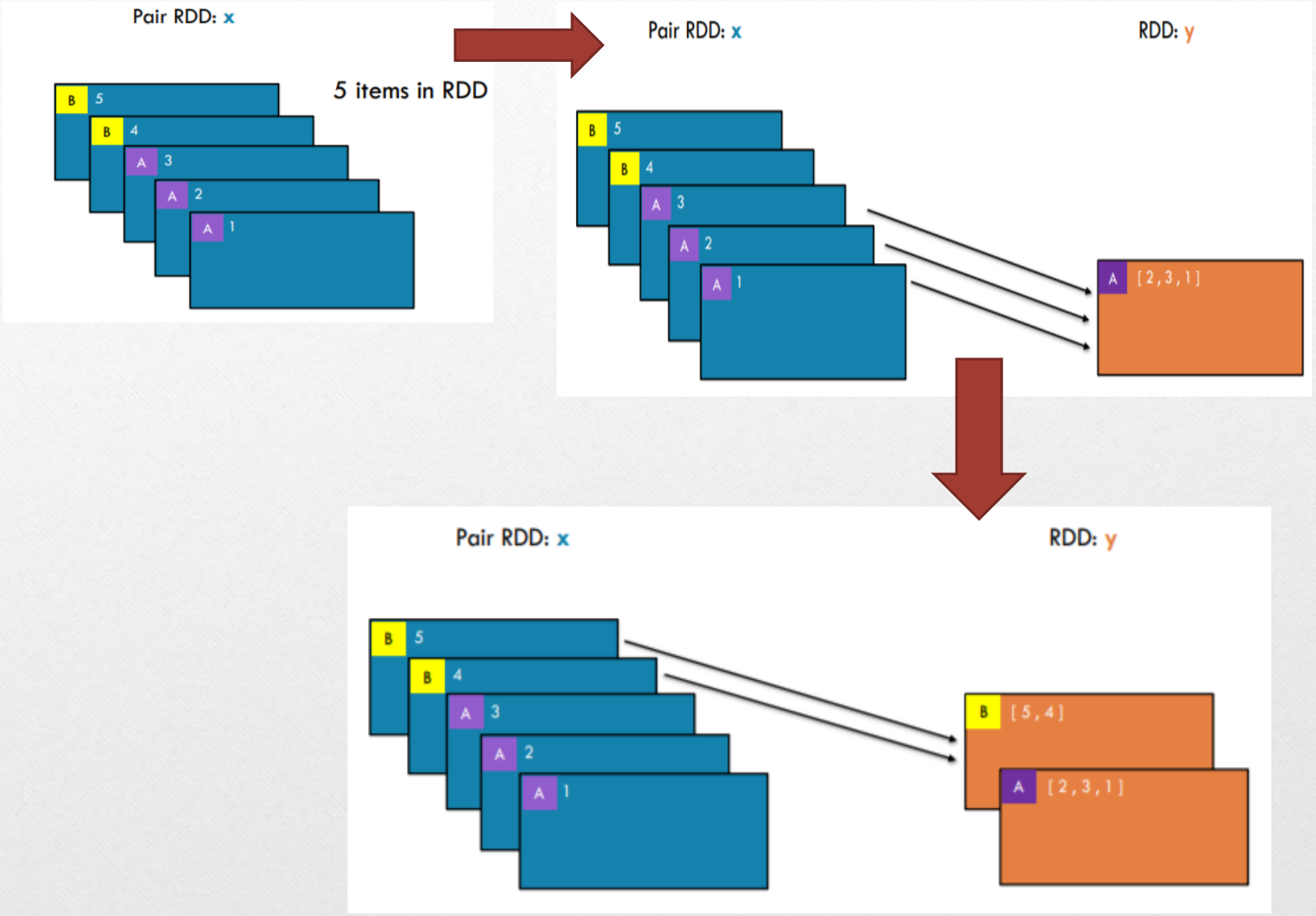
```
1  x = sc.parallelize([('B',5),('B',4),('A',3),('A',2),('A',1)])
2  y = x.groupByKey()
3  print(x.collect())
4  print(list((j[0], list(j[1])) for j in y.collect()))
```

▸ (2) Spark Jobs

```
[('B', 5), ('B', 4), ('A', 3), ('A', 2), ('A', 1)]
[('B', [5, 4]), ('A', [3, 2, 1])]
```

GroupByKey


(a, 1)
(b, 1)
(a, 1)
(a, 1)
(b, 1)
(b, 1)
(a, 1)
(a, 1)
(a, 1)
(b, 1)
(b, 1)
(b, 1)
(a, 1)
(a, 1)
(a, 1)
(a, 1)
(a, 1)
(a, 1)
(a, 6)
(b, 1)
(b, 1)
(b, 1)
(b, 1)
(b, 1)
(b, 1)
(b, 6)


Pair RDD: x
5 items in RDD
B 5
B 4
A 3
A 2
A 1
Pair RDD: x
RDD: y
A [2,3,1]
Pair RDD: x
RDD: y
B [5,4]
A [2,3,1]

reduceByKey(func, [numPartitions])

- 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 func, which must be of type (V,V) => V. Like in groupByKey, the number of reduce tasks is configurable through an optional second argument.
- NOTE: Note If you are grouping using (groupbykey) in order to perform an aggregation (such as a sum or average) over each key, using `reduceByKey` or `aggregateByKey` will provide much better performance.

Cmd 2

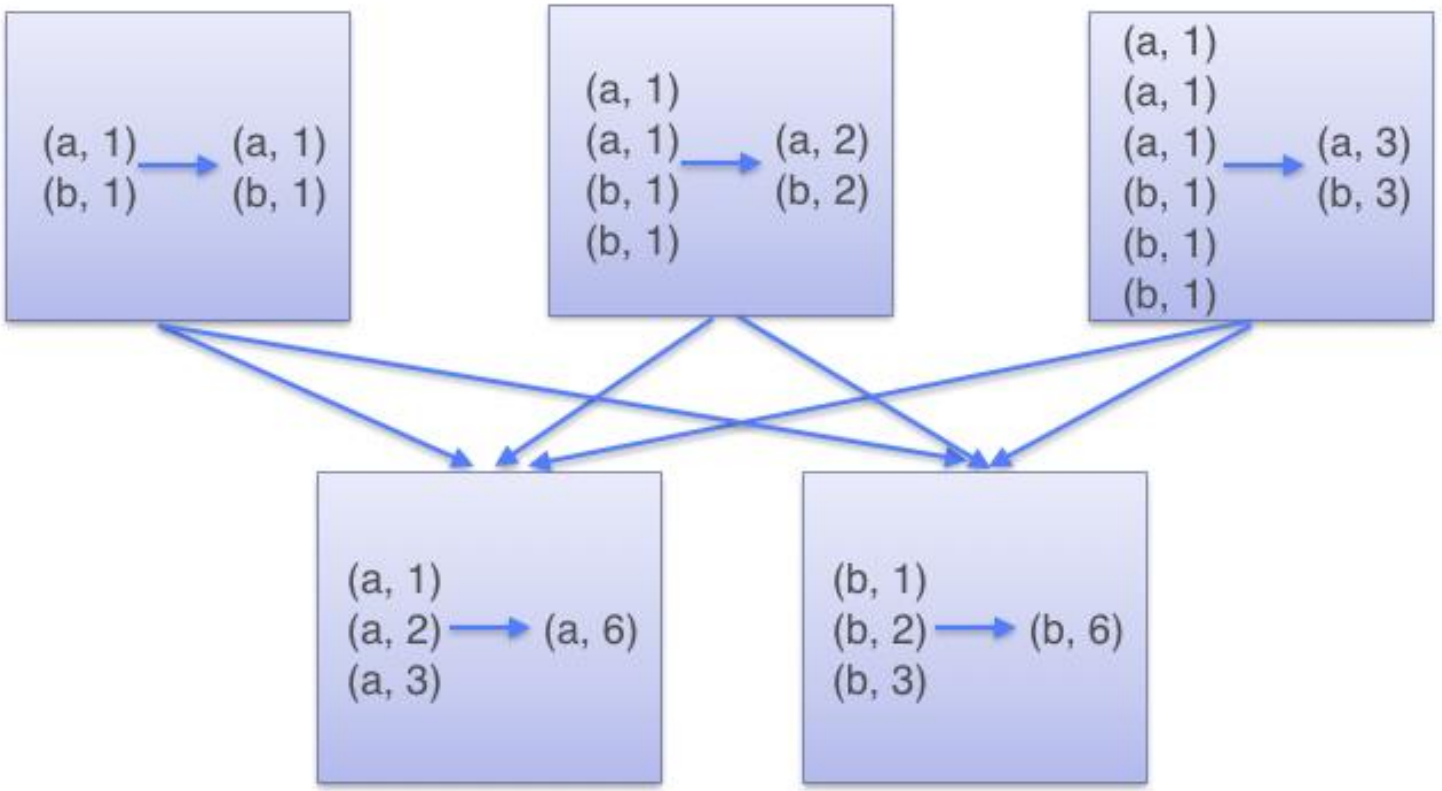
```
1 wordsList = ['cat', 'elephant', 'rat', 'rat', 'cat']
2 wordsRDD = sc.parallelize(wordsList, 4)
3 wordCountsCollected = (wordsRDD
4                        .map(lambda w: (w, 1))
5                        .reduceByKey(lambda x,y: x+y)
6                        .collect())
7 print(wordCountsCollected)
```

▸ (1) Spark Jobs

```
[('cat', 2), ('elephant', 1), ('rat', 2)]
```

Command took 0.77 seconds -- by pysparktelugu@gmail.com at 10/26/2020, 10:39:25 PM on datacluster

ReduceByKey



Spark Caching & Persistence

Spark caching can be used to pull data sets into a cluster-wide in-memory cache. This is very useful for accessing repeated data, such as querying a small "hot" dataset or when running an iterative algorithm

There are two ways to persist RDDs in Spark:

1. cache( )
2. persist( )

There are some advantages of RDD caching and persistence mechanism in spark.

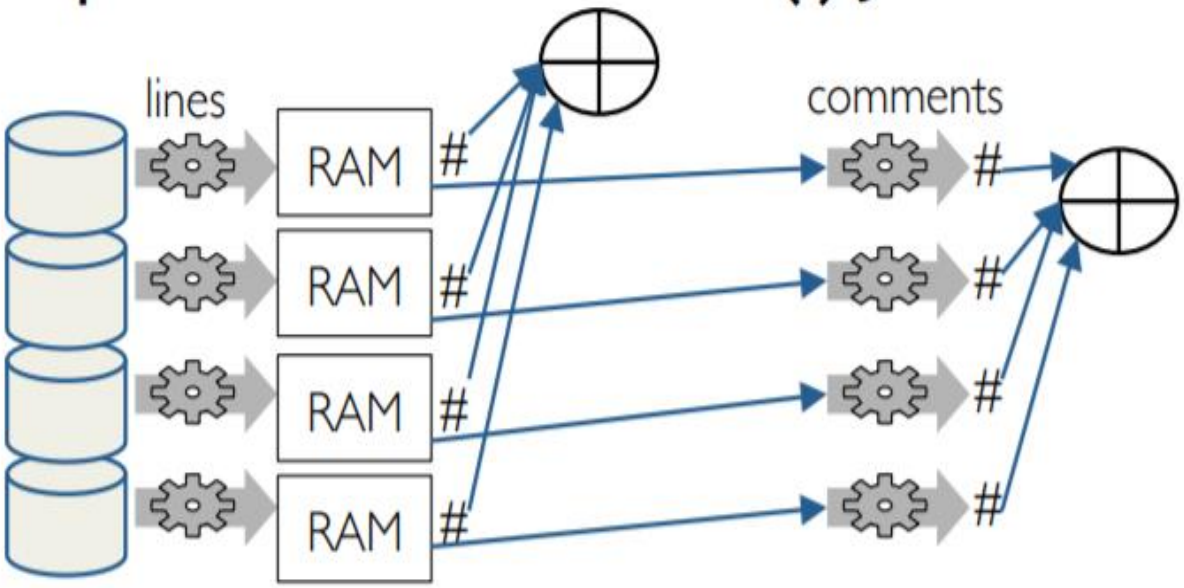
- Time efficient
- Cost efficient
- Lesser the execution time.

STORAGE TYPES:

1) MEMORY_ONLY
2) MEMORY_AND_DISK
3) DISK_ONLY
4) MEMORY_ONLY_SER
5) MEMORY_AND_DISK_SER

Caching RDDs

```
lines = sc.textFile("...", 4)
lines.cache() # save, don't recompute!
comments = lines.filter(isComment)
print lines.count(),comments.count()
```



Clear Spark Cache

RDD Unpersist

Spark automatically monitors cache usage on each node and drops out old data partitions in a least-recently-used (**LRU**) fashion.

If you would like to manually remove an RDD instead of waiting for it to fall out of the cache,
use the **RDD.unpersist()** method.

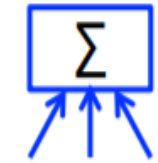
pySpark Shared Variables

Broadcast Variables

» Efficiently send large, ***read-only*** value to all workers

» Saved at workers for use in one or more Spark operations

» Like sending a large, read-only lookup table to all the nodes



Accumulators

» Aggregate values from workers back to driver

» Only driver can access value of accumulator

» For tasks, accumulators are write-only

» Use to count errors seen in RDD across workers

Broadcast Variables

Broadcast variables allow the programmer to keep a read-only variable cached on each machine rather than shipping a copy of it with tasks.

They can be used, for example, to give every node a copy of a large input dataset in an efficient manner. Spark also attempts to distribute broadcast variables using efficient broadcast algorithms to reduce communication cost.

Syntax:

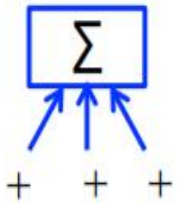
```
#Create Broadcast Variable using SparkContext.broadcast (LIST)
broadcast_v = sc.broadcast([1, 2, 3])
# Print the broadcast variable value using  .value
broadcast_v.value
```

- **Broadcast variables** are created from a variable v by calling **SparkContext.broadcast(v)**.
- The broadcast variable is a wrapper around v, and its value can be accessed by calling the **.value** method.

```
broadcast_v.unpersist()
```

unpersist(self, blocking=False)

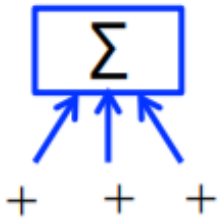
Delete cached copies of this broadcast on the executors. If the broadcast is used after this is called, it will need to be re-sent to each executor.



Accumulators

- Variables that can only be "added" to by associative op
- Used to efficiently implement parallel counters and sums
- Only driver can read an accumulator's value, not tasks

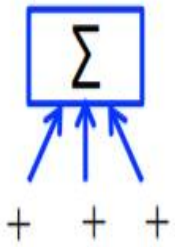
```
>>> accum = sc.accumulator(0)
>>> rdd = sc.parallelize([1, 2, 3, 4])
>>> def f(x):
>>>    global accum
>>>    accum += x

>>> rdd.foreach(f)
>>> accum.value
Value: 10
```



Accumulators Example

- Counting empty lines

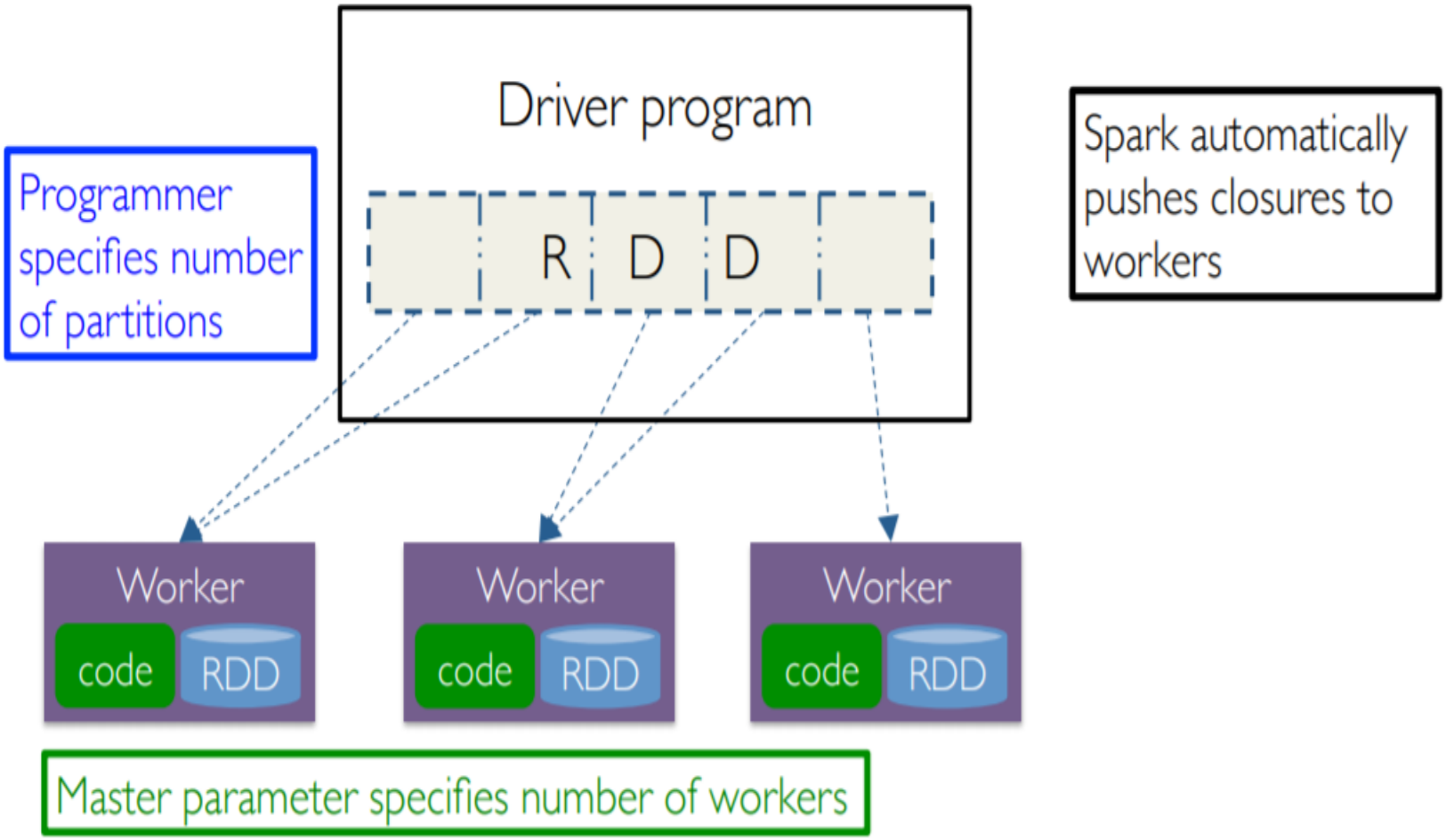
```
file = sc.textFile(inputFile)
# Create Accumulator[Int] initialized to 0
blankLines = sc.accumulator(0)

def extractCallSigns(line):
    global blankLines # Make the global variable accessible
    if (line == ""):
        blankLines += 1
    return line.split(" ")

callSigns = file.flatMap(extractCallSigns)
print "Blank lines: %d" % blankLines.value
```



Accumulators

- Tasks at workers cannot access accumulator's values
- Tasks see accumulators as write-only variables
- Accumulators can be used in actions or transformations:
  - » Actions: each task's update to accumulator is *applied only once*
  - » Transformations: *no guarantees* (use only for debugging)
- Types: integers, double, long, float
  - » See lab for example of custom type

Summary


Driver program
R D D
Programmer specifies number of partitions
Spark automatically pushes closures to workers
Worker
code
RDD
Worker
code
RDD
Worker
code
RDD
Master parameter specifies number of workers

RDD Fault Tolerance

- Hadoop conservative/pessimistic approach

  →Go to disk / stable storage (HDFS)

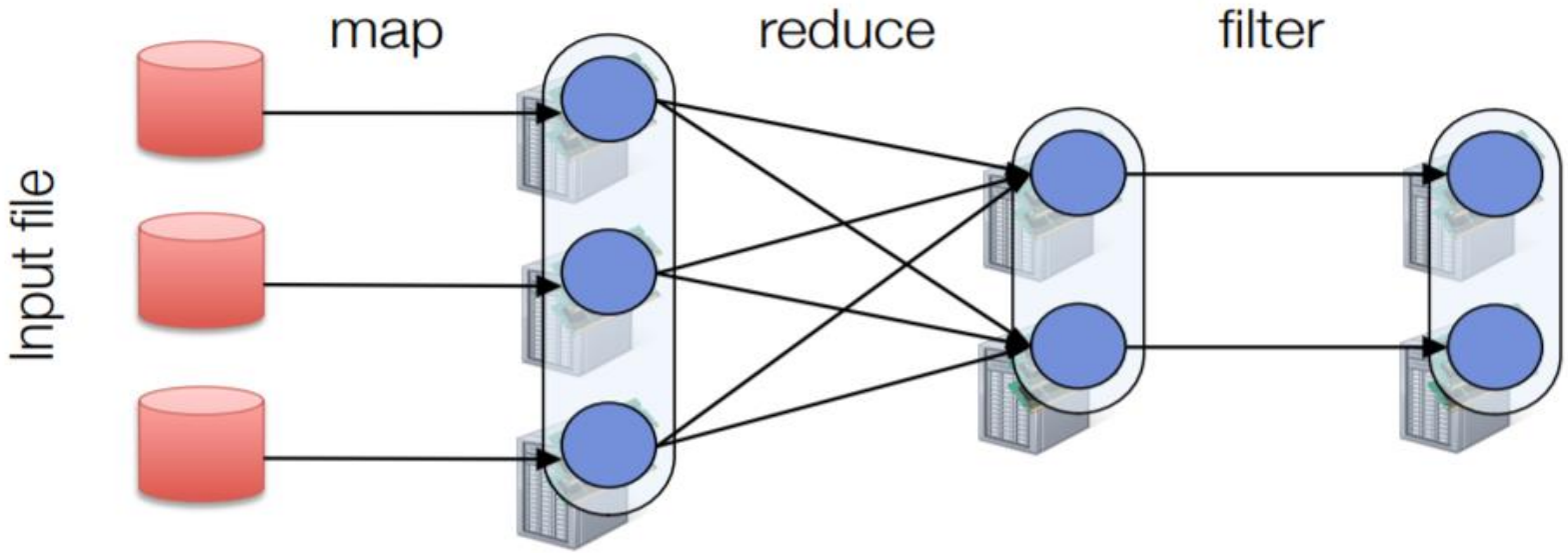
- Spark optimistic

  →Don't "waste" time writing to disk, re-compute in case of crash using lineage

Fault Tolerance

RDDs track *lineage* info to rebuild lost data

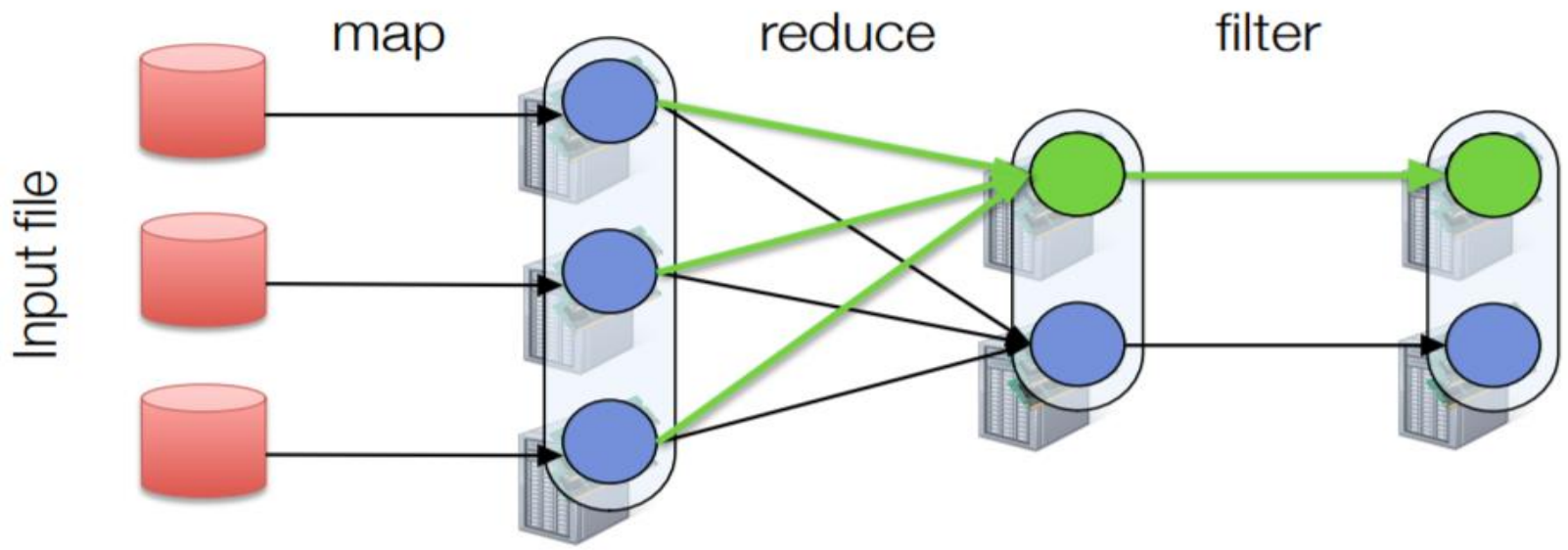
```
file.map(lambda rec: (rec.type, 1))
    .reduceByKey(lambda x, y: x + y)
    .filter(lambda (type, count): count > 10)
```



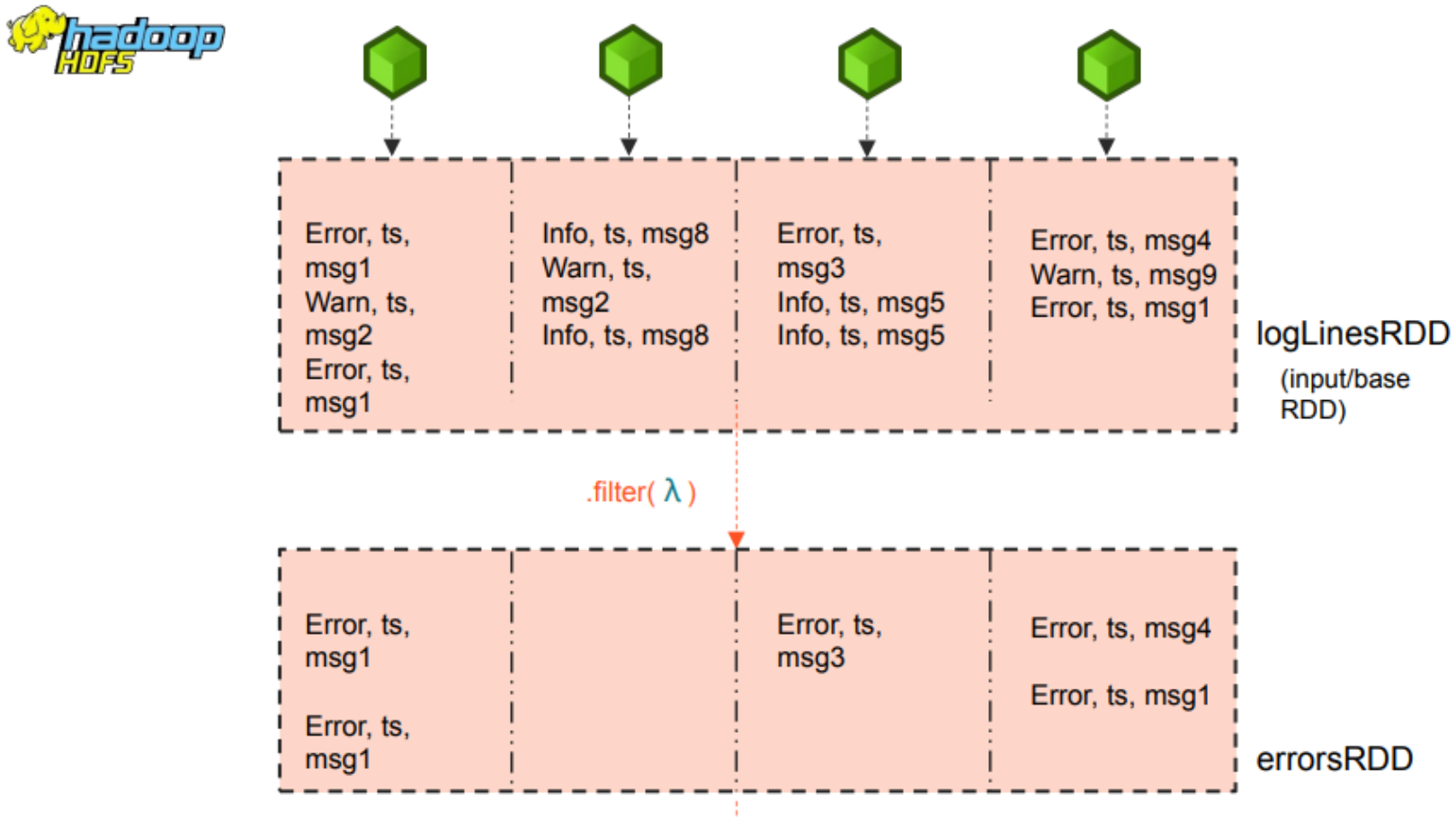
Fault Tolerance

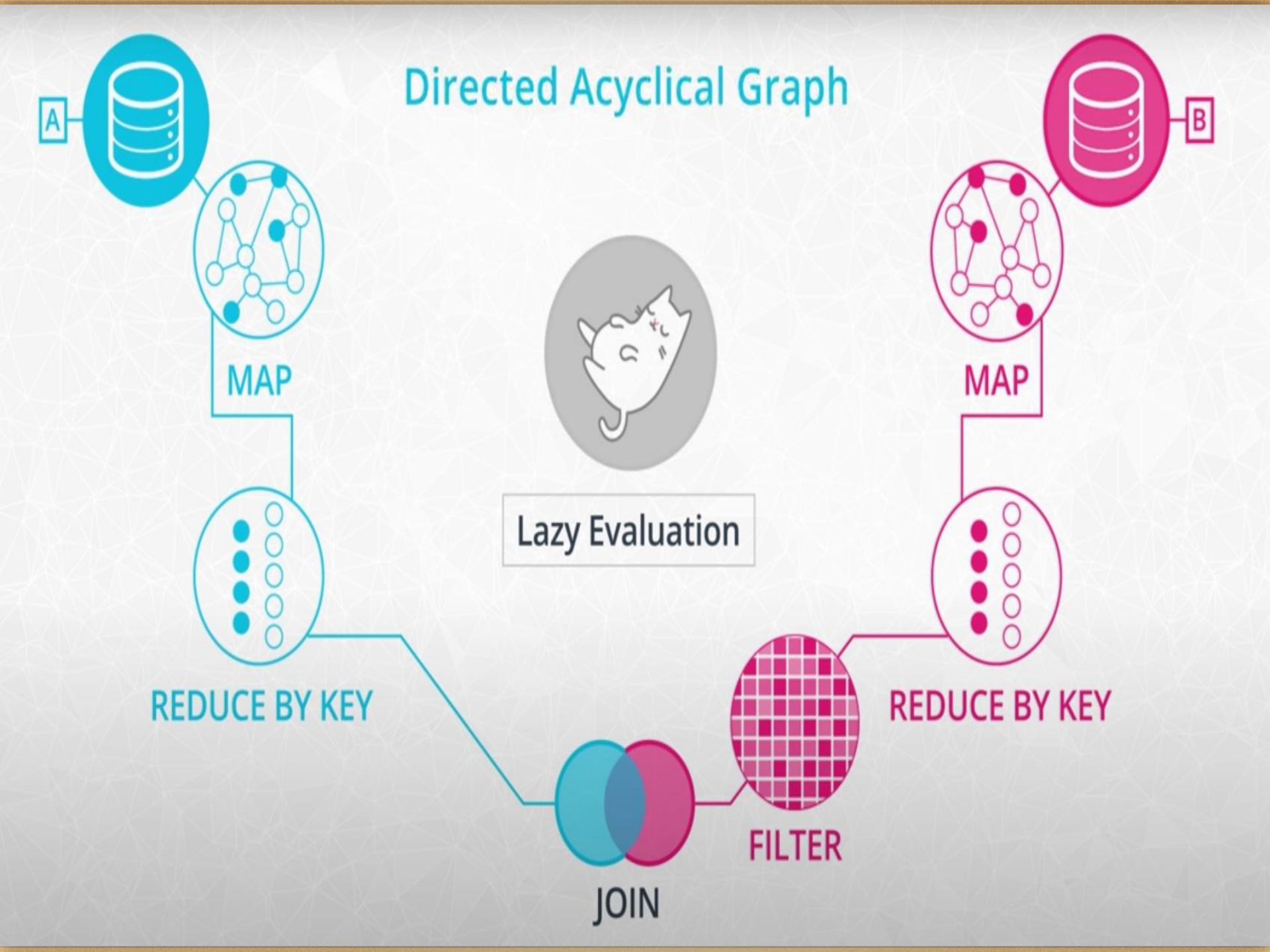
RDDs track *lineage* info to rebuild lost data

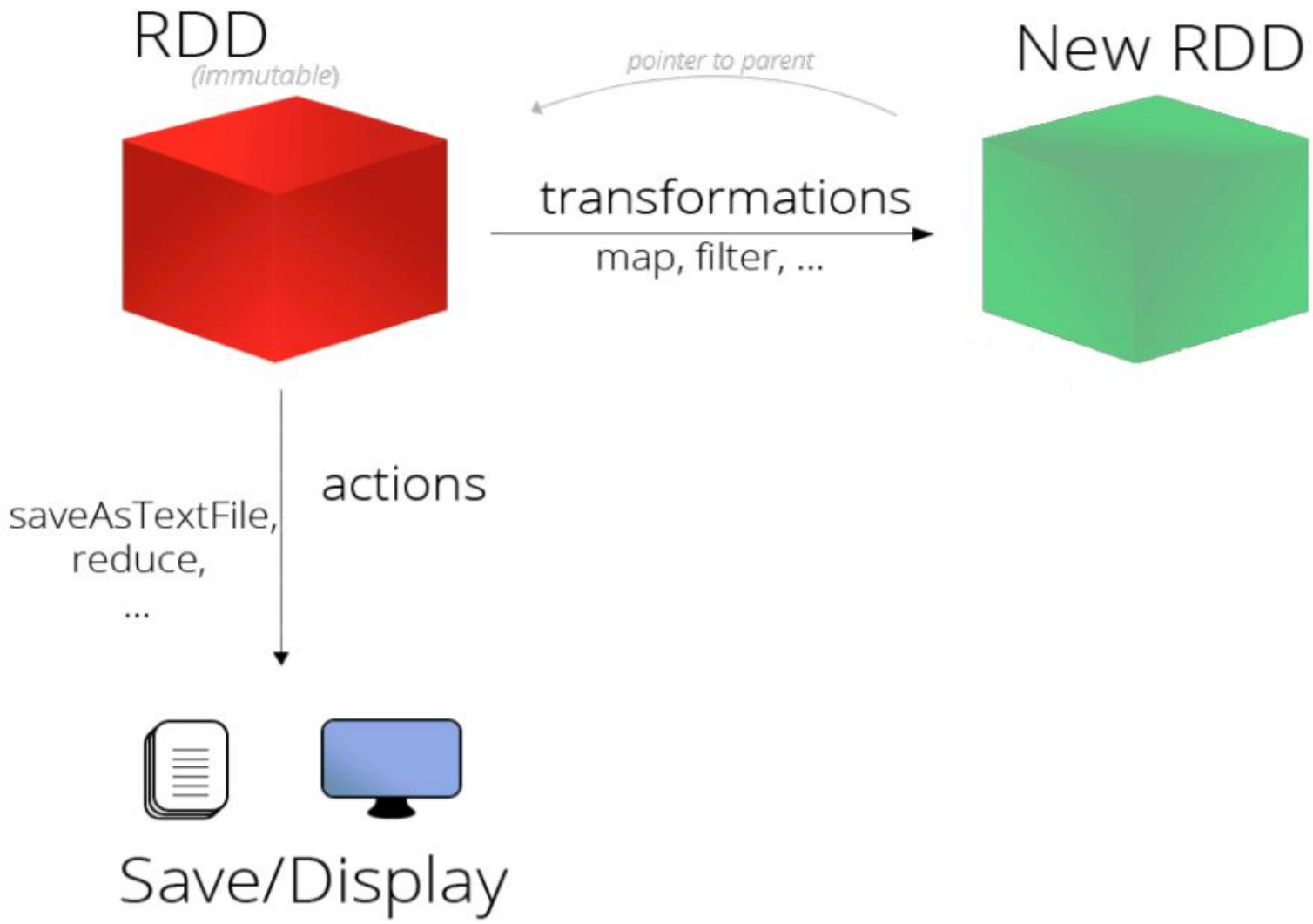
```
file.map(lambda rec: (rec.type, 1))
    .reduceByKey(lambda x, y: x + y)
    .filter(lambda (type, count): count > 10)
```



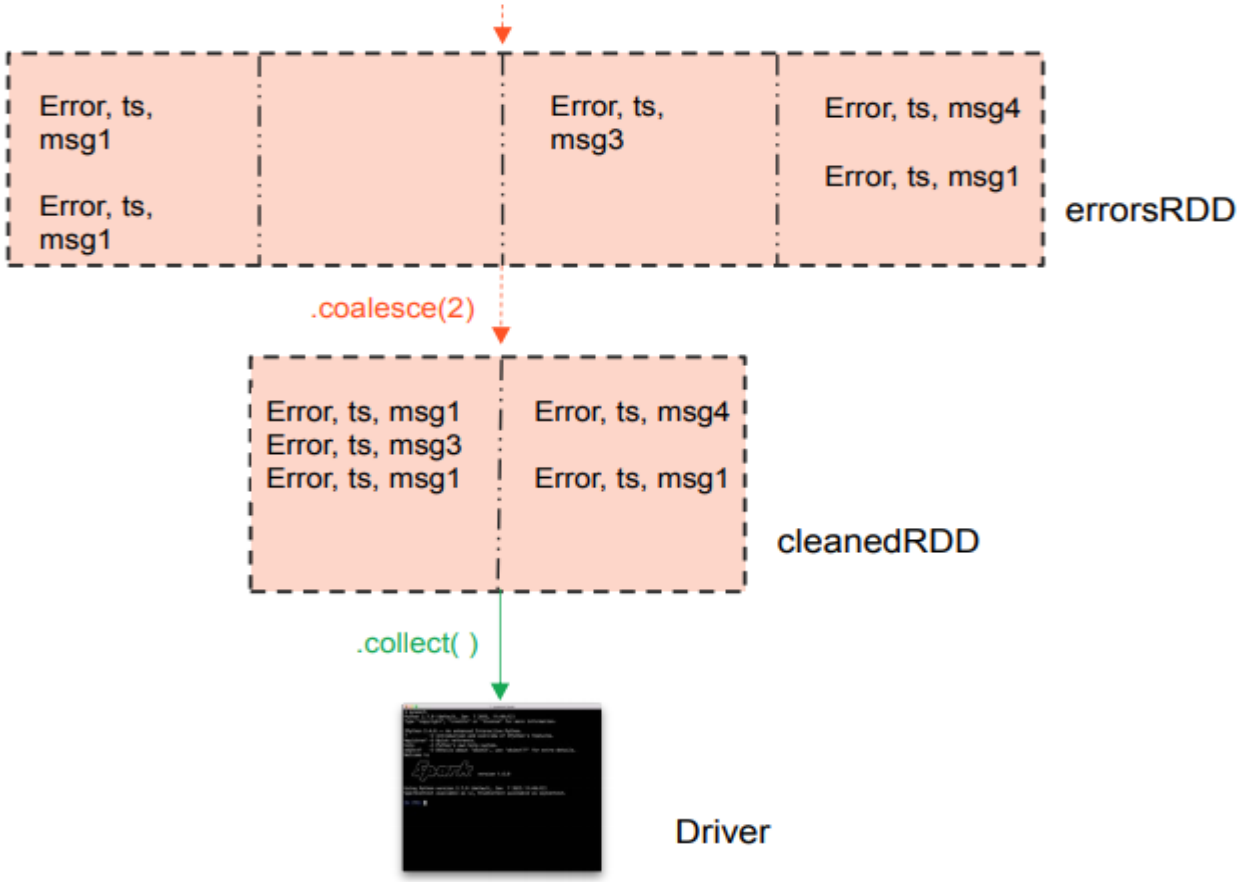
RDD Example


hadoop
HDFS
Error, ts, msg1
Warn, ts, msg2
Error, ts, msg1
Info, ts, msg8
Warn, ts, msg2
Info, ts, msg8
Error, ts, msg3
Info, ts, msg5
Info, ts, msg5
Error, ts, msg4
Warn, ts, msg9
Error, ts, msg1
logLinesRDD
(input/base RDD)
.filter(λ)
Error, ts, msg1
Error, ts, msg1
Error, ts, msg3
Error, ts, msg4
Error, ts, msg1
errorsRDD

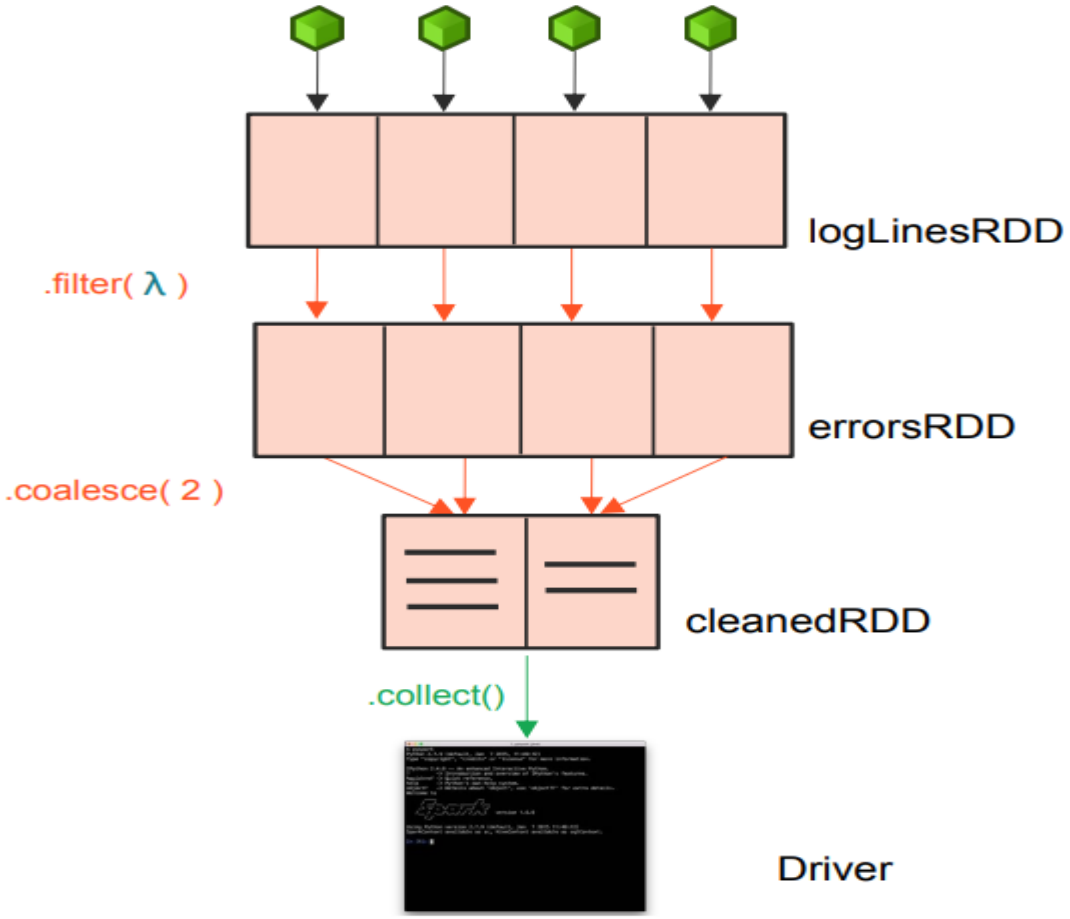

Directed Acyclical Graph
A
B
MAP
MAP
Lazy Evaluation
REDUCE BY KEY
REDUCE BY KEY
FILTER
JOIN


RDD
(immutable)
pointer to parent
New RDD
transformations
map, filter, ...
actions
saveAsTextFile,
reduce,
...
Save/Display

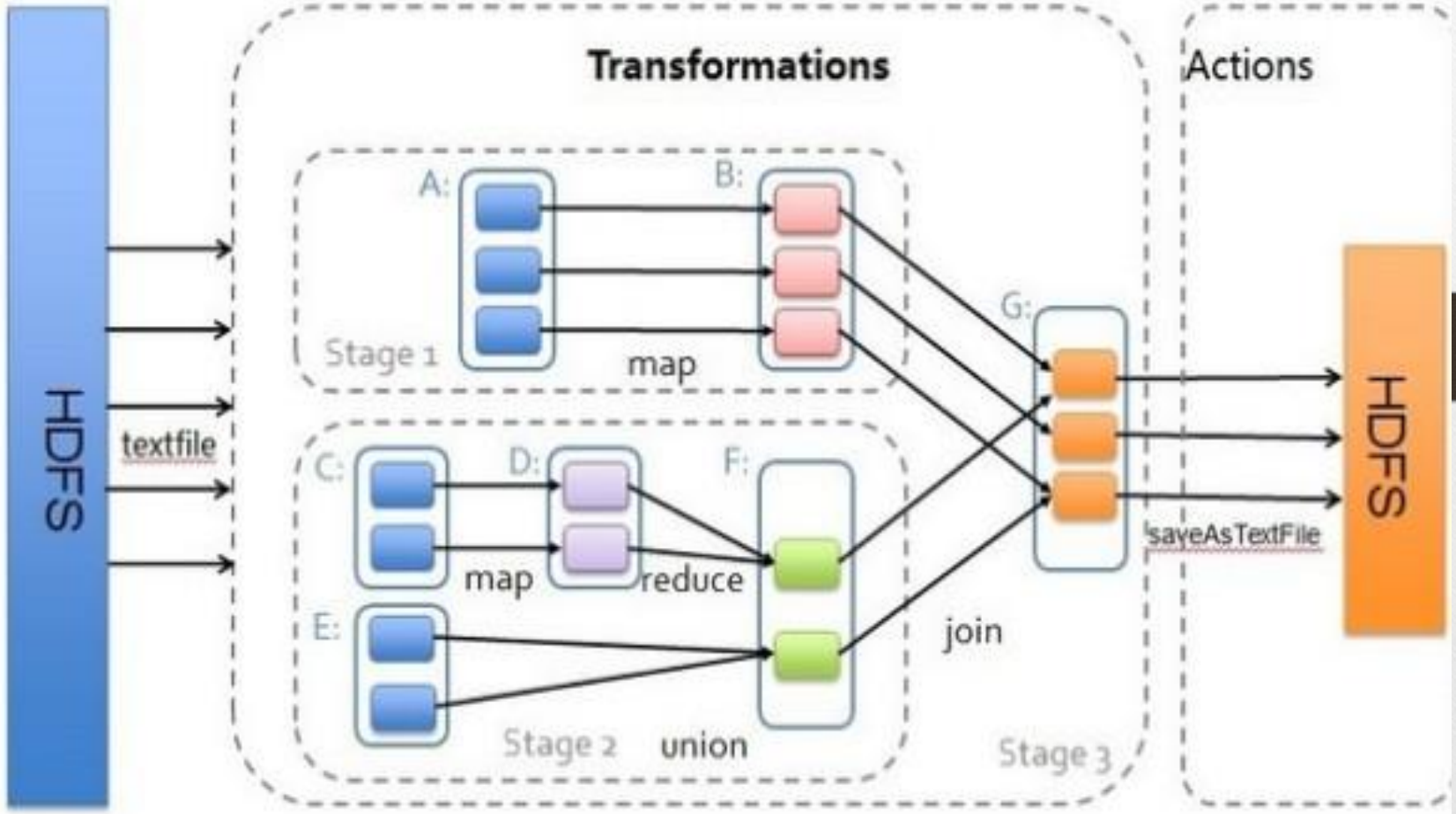
RDD Example


Error, ts, msg1
Error, ts, msg1
Error, ts, msg3
Error, ts, msg4
Error, ts, msg1
errorsRDD
.coalesce(2)
Error, ts, msg1
Error, ts, msg3
Error, ts, msg1
Error, ts, msg4
Error, ts, msg1
cleanedRDD
.collect( )
Driver

RDD Lineage


logLinesRDD
.filter(λ)
errorsRDD
.coalesce(2)
cleanedRDD
.collect()
Driver

Spark:Transformations & Actions



Spark End to End operations

Pyspark DataFrame

Unstructured data

The university has 5600 students.
John's ID is number 1, he is 18 years old and already holds a B.Sc. degree.
David's ID is number 2, he is 31 years old and holds a Ph.D. degree. Robert's ID is number 3, he is 51 years old and also holds the same degree as David, a Ph.D. degree.

Semi-structured data

```
<University>
 <Student ID="1">
  <Name>John</Name>
  <Age>18</Age>
  <Degree>B.Sc.</Degree>
 </Student>
 <Student ID="2">
  <Name>David</Name>
  <Age>31</Age>
  <Degree>Ph.D. </Degree>
 </Student>
....
</University>
```

Structured data

ID	Name	Age	Degree
1	John	18	B.Sc.
2	David	31	Ph.D.
3	Robert	51	Ph.D.
4	Rick	26	M.Sc.
5	Michael	19	B.Sc.

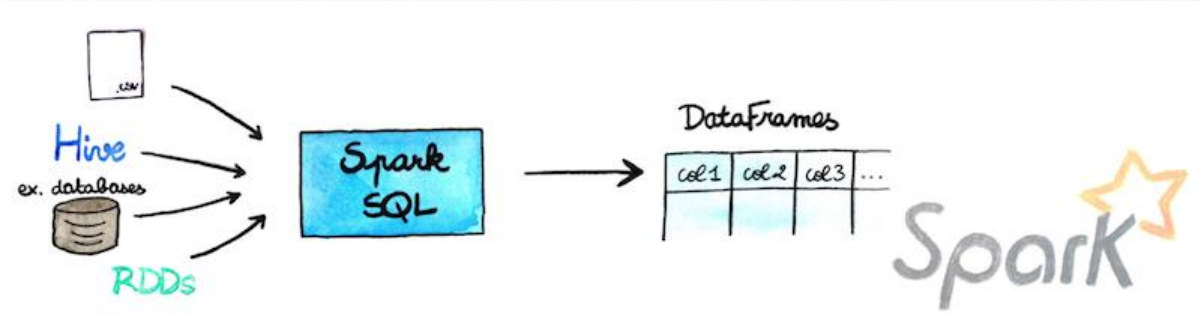
What is Spark Data Frame

Data frames usually refer to a data structure, which is tabular in nature. It represents Rows, each consisting of a number of observations. Rows can have many different data formats (Heterogeneous), while a column can have data of the same data type (Uniform). Data frames usually contain some metadata in addition to data; for example, column and row names.

DATAFRAME CHARACTERISTICS

1. The data frame is distributed in nature, which makes it error-tolerant and highly available data structures.
2. The lazy evaluation is an evaluation strategy that keeps evaluating an expression until its value is needed. It avoids repeated reviews. The lazy rating in Spark means execution won't start until an action is triggered. In Spark, a picture of laziness appears when Spark transforms occur.
3. Dataframes are by nature Immutable. Immutable, I mean it's an object whose state cannot be modified after it has been created. But we can transform its values by applying a certain transformation, as in RDD.

DATAFRAME CREATION



DATAFRAME

- DISPLAY() And SHOW() we will use for DataFrame and it will display in table formation. if we use Collect() it will display row format
- We will use Spark Session or SQLContext for DataFrame Creation
- Dataframe will contain two Dimension Structured Data(Like Columns & Rows)

RDD

- COLLECT() we will in RRD to get the result set. DISPLAY() and SHOW() actions are not available in RDD.
- We will use SparkContext for RDD Creation
- RDD can store RAW Format Data like individual items...

Data Frame Can Be Created following Ways:

```
DF = spark.createDataFrame(data,schema)
DF = Spark.sql("table_name")
DF = spark.read.csv("csvFile")
DF = spark.read.json("jsonFile")
DF = spark.read.orc("orcFile")
DF = spark.read.parquet("parquetFile")
DF = spark.read.format("com.databricks.spark.xml").load("xmlfile")
DF = spark.read.format("com.crealytics.spark.excel").load("ExcelFile")
DF = spark.read.format("jdbc").option().load()
DF = spark.read.format("avro").load("avroFile")
```

Creating DataFrame Using SQLContext

```
sql_df = sqlContext.createDataFrame([('Reshwanth',9,'Bangalore'),('Vikranth',5,'Bangalore')],['name','age','Locaiton'])
```

```
sql_df: pyspark.sql.dataframe.DataFrame
    name: string
    age: long
    Locaiton: string
```

Displaying Data using **df.show()** method

```
sql_df.show()
```

(3) Spark Jobs

```
+---------+---+---------+
|     name|age| Locaiton|
+---------+---+---------+
|Reshwanth|  9|Bangalore|
| Vikranth|  5|Bangalore|
+---------+---+---------+
```

Displaying Data using **display(df)** method

```
display(sql_df)
```

(3) Spark Jobs

	name	age	Locaiton
1	Reshwanth	9	Bangalore
2	Vikranth	5	Bangalore

Showing all 2 rows.

Reading Metadata And Data Functions

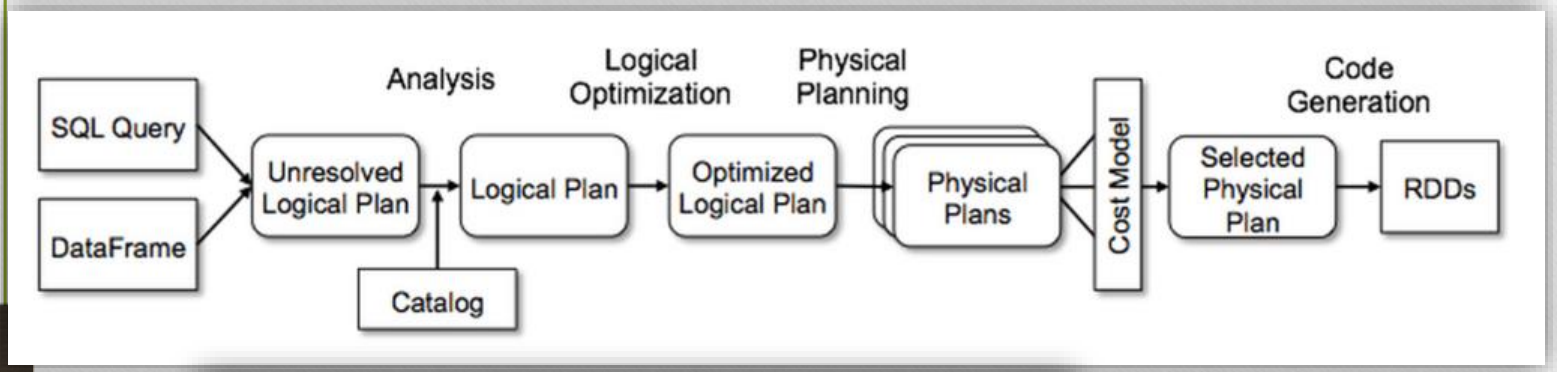
READ_METADATA	READ_DATA
df.columns	df.show()
df.printschema()	display(df)
df.dtypes	df.collect()
df.schema	df.describe("*").show()

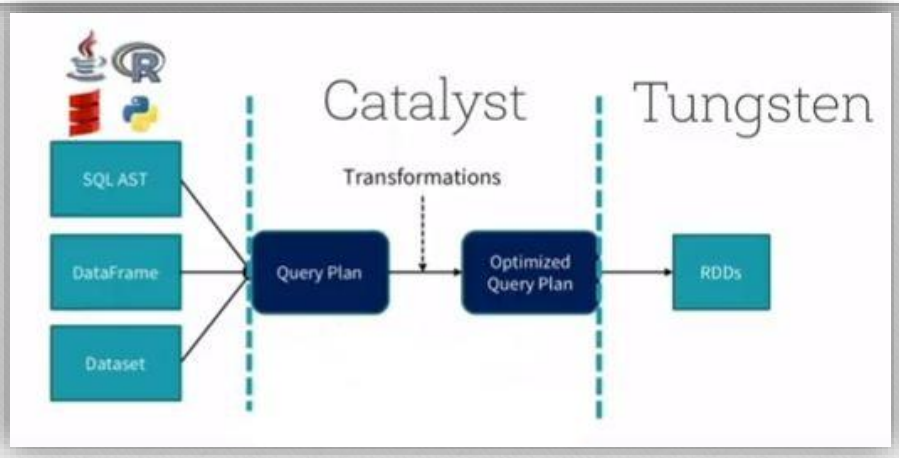
Modifying Metadata And MetaData.

MODIFY_METADATA	MODIFY_DATA
df.WithColumn()	df.fillna("val")
df.WithColumnRenamed()	df.dropna()
df.drop("column")	df.dropDuplicates()
df.toDF()	df.filter("col==condition")
	df.select("*")
	df.distinct()

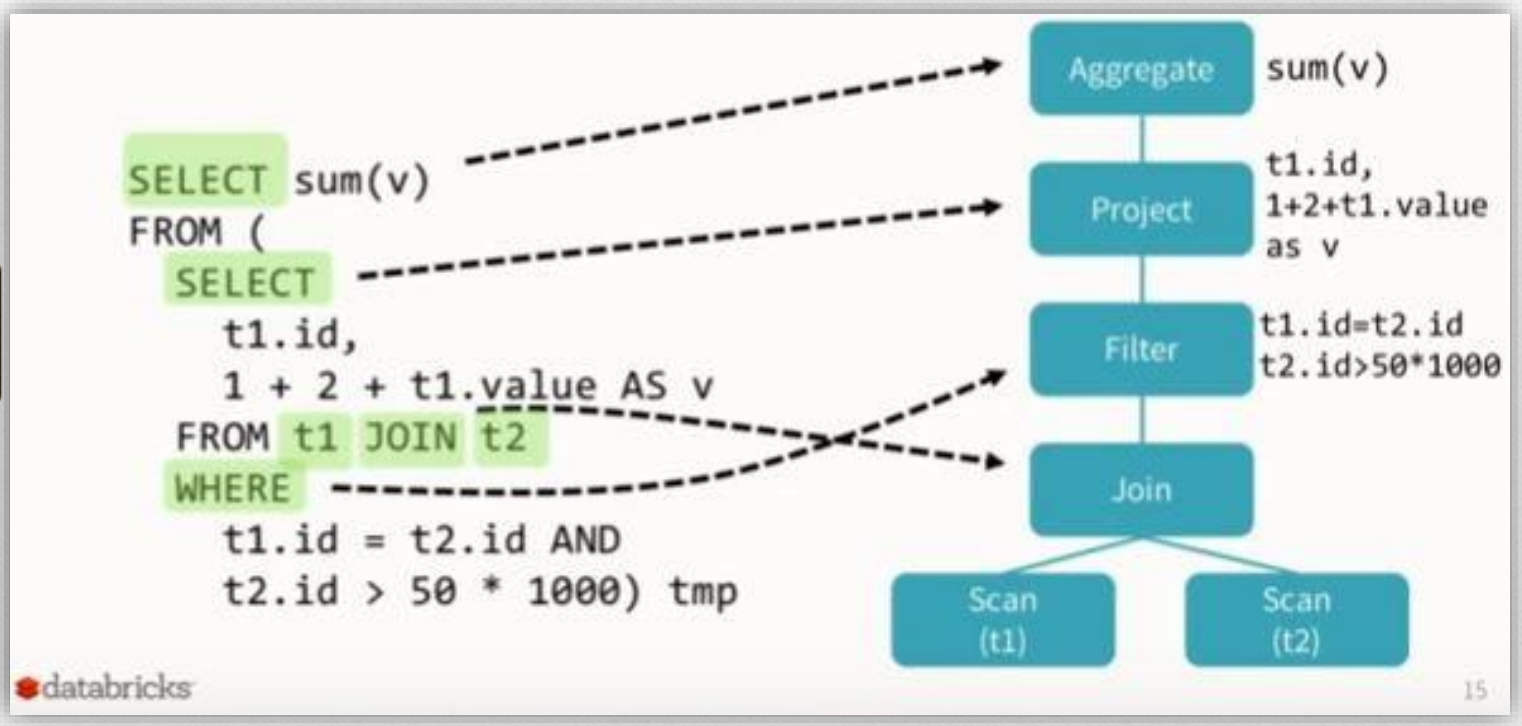
Any Data Engineering Project Common Operations Are **Metadata Validations** and **Data Validations**.
Like validating **Columns**, **Data Types**,
Validating **Data** like **Duplicate Data**, **Null Data** **using above functions.**

Spark SQL's Catalyst

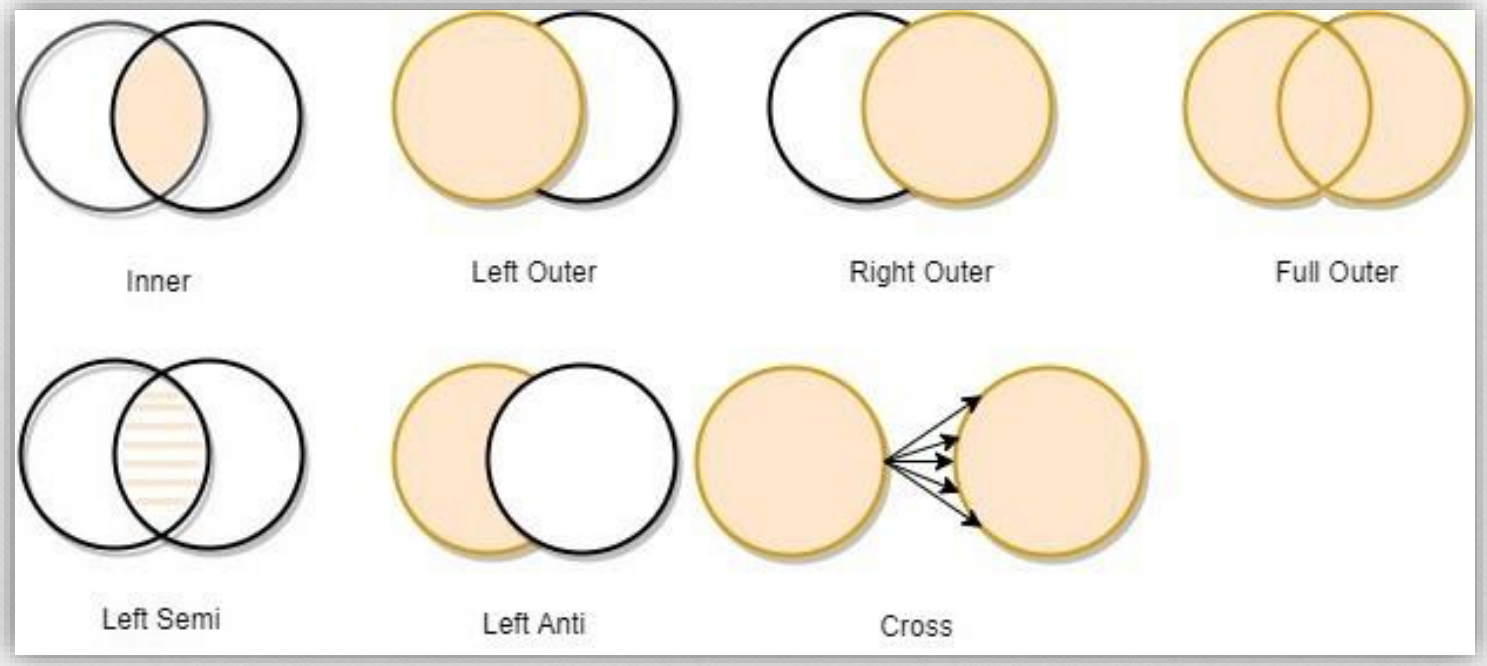




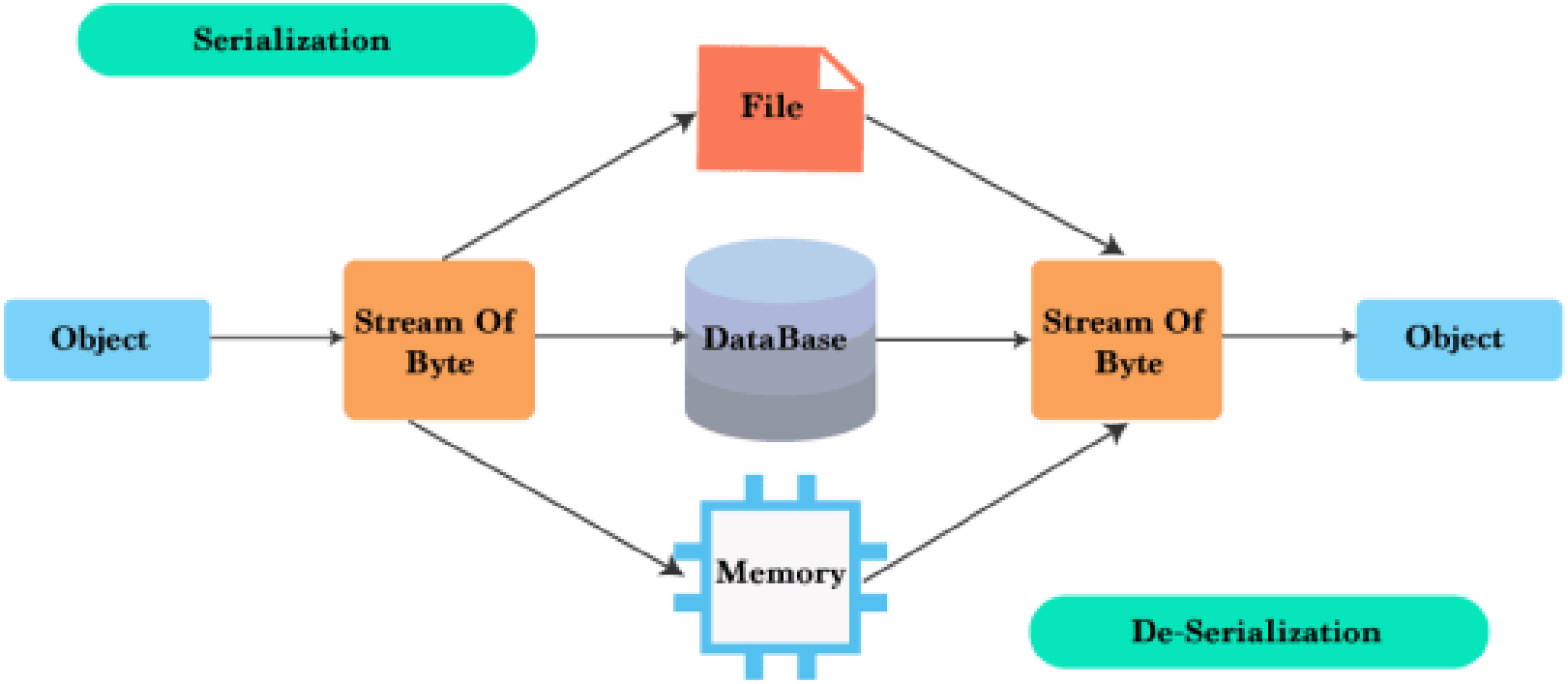
How Catalyst Works?

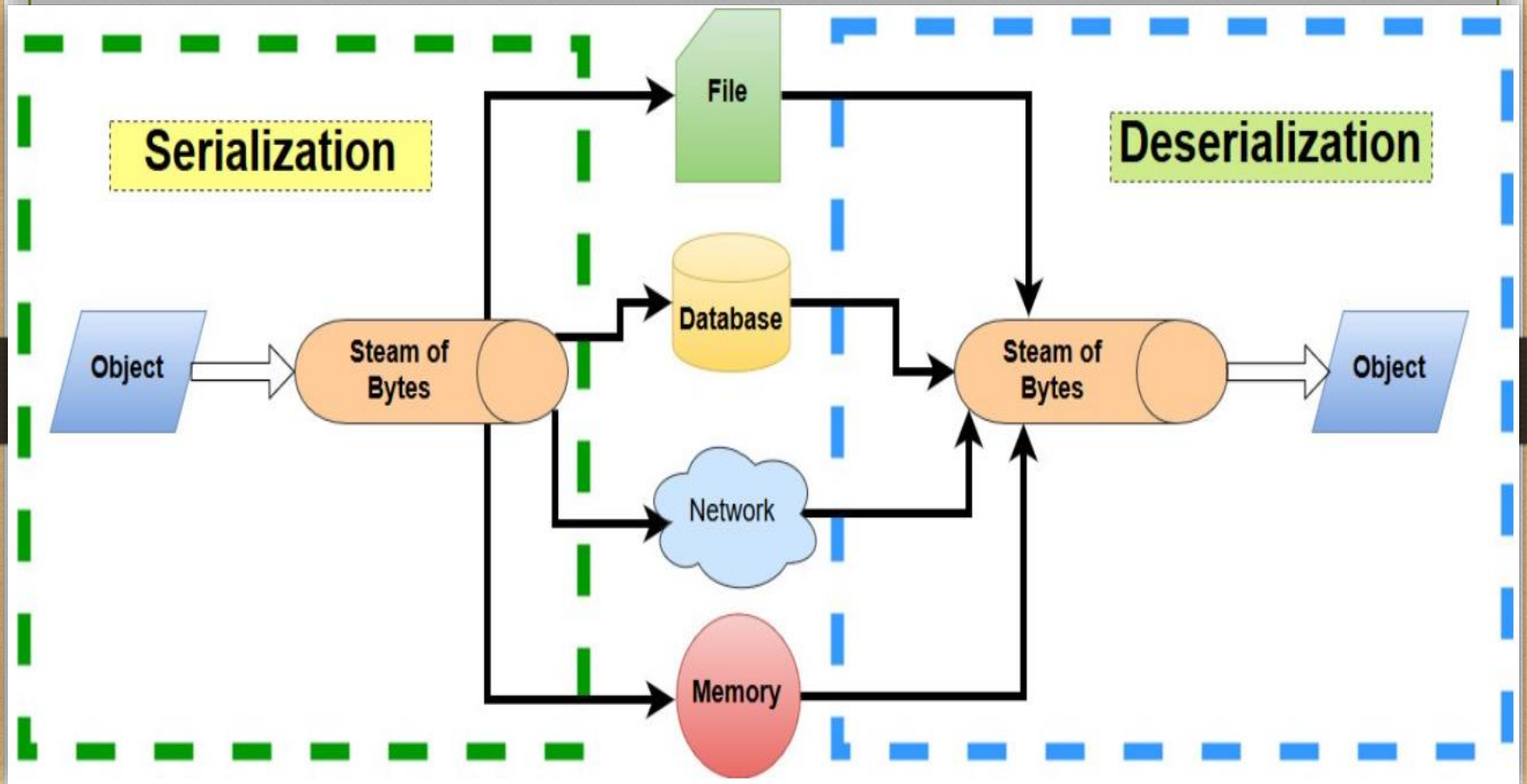


Types of Joins.

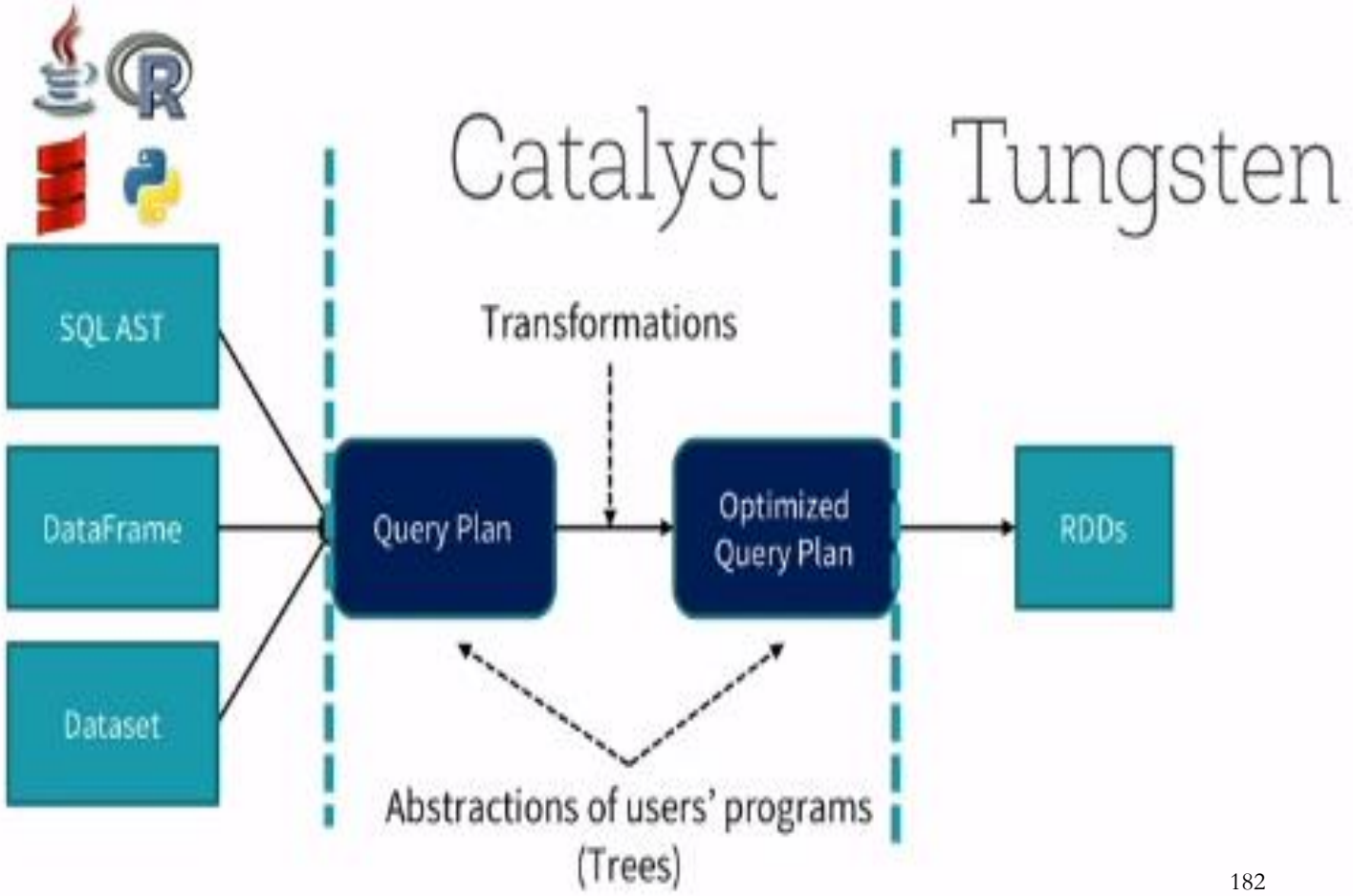


How to Serialize(Save) and De-serialize(Re-store) objects?




Serialization
Deserialization
File
Database
Network
Memory
Object
Steam of Bytes
Steam of Bytes
Object

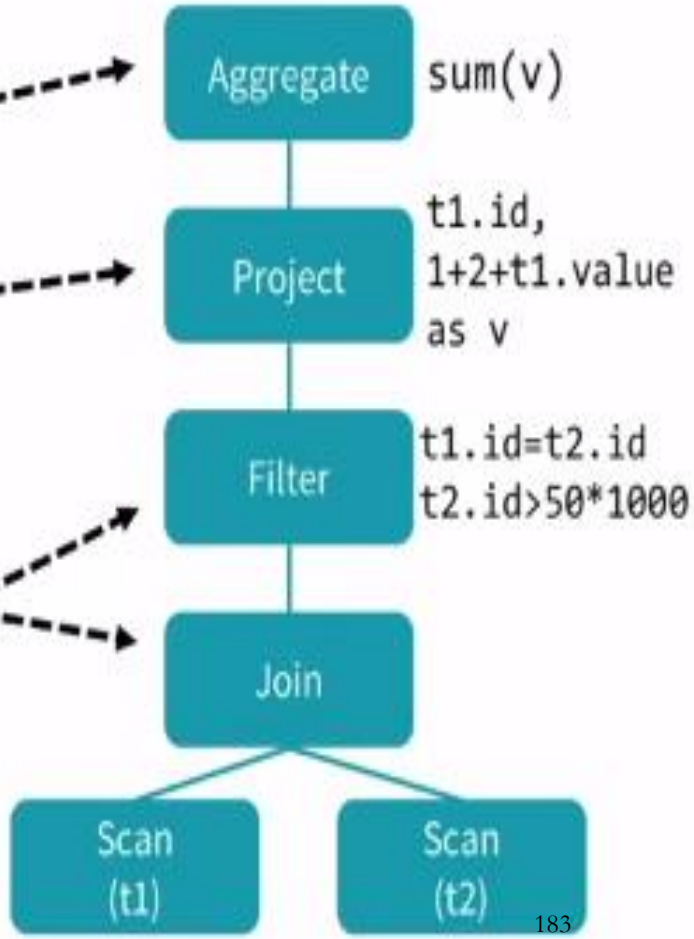
Spark SQL Overview



Trees: Abstractions of Users' Programs

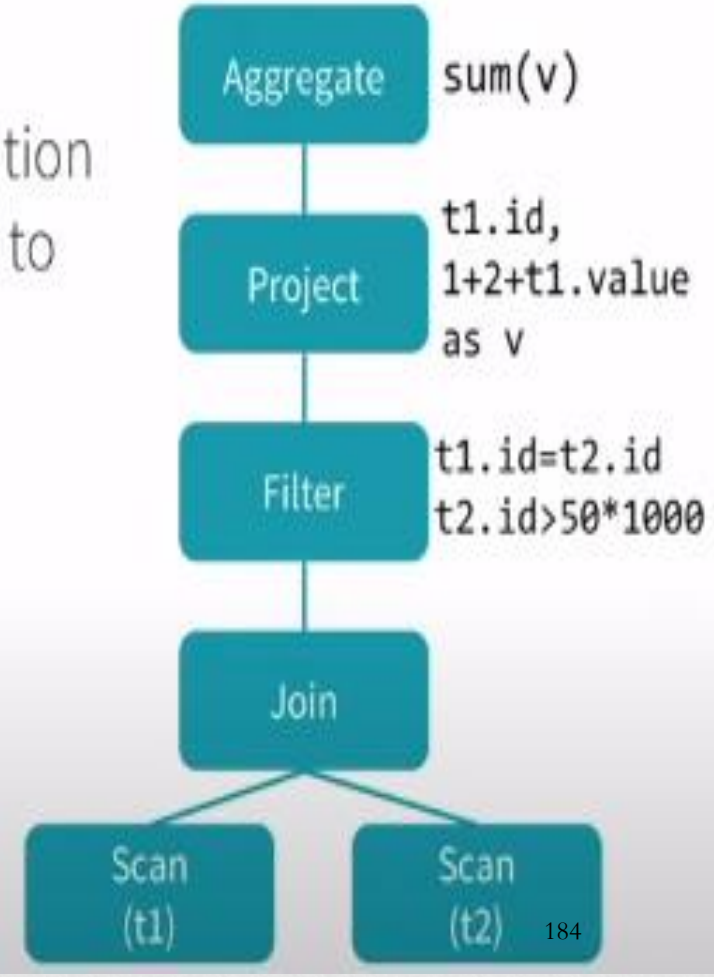
Query Plan

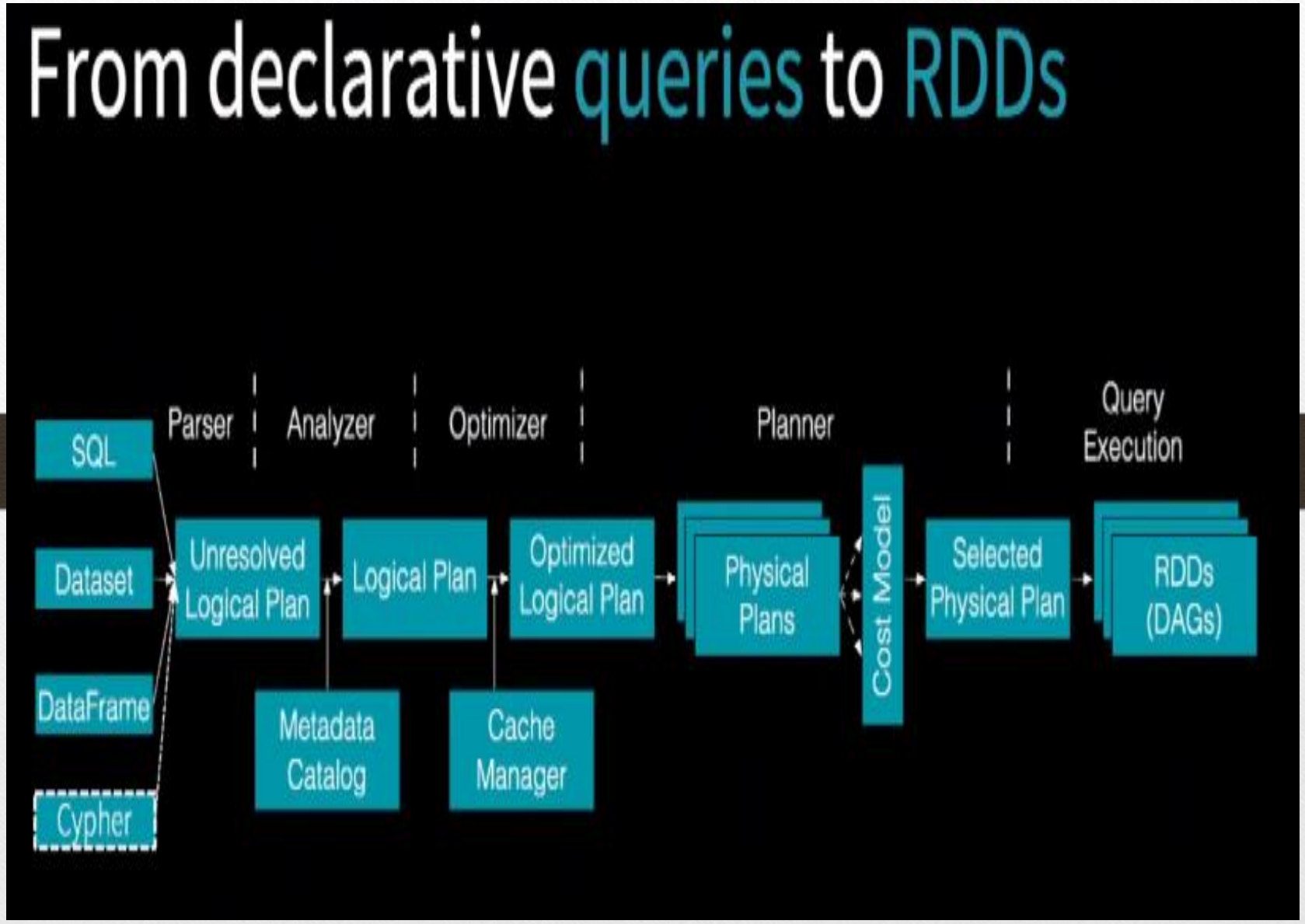
```
SELECT sum(v)
FROM (
  SELECT
    t1.id,
    1 + 2 + t1.value AS v
  FROM t1 JOIN t2
  WHERE
    t1.id = t2.id AND
    t2.id > 50 * 1000) tmp
```



Logical Plan

- A Logical Plan describes computation on datasets **without** defining how to conduct the computation




From declarative queries to RDDs
SQL
Dataset
DataFrame
Cypher
Parser
Analyzer
Optimizer
Planner
Query Execution
Unresolved Logical Plan
Logical Plan
Optimized Logical Plan
Physical Plans
Cost Model
Selected Physical Plan
RDDs (DAGs)
Metadata Catalog
Cache Manager

THANK YOU