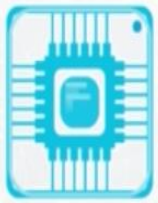
Prepared By : Ravi

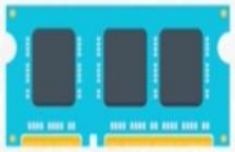
Source: spark.apache.org & www.databricks.com

Fastest



CPU

200x faster than memory



MEMORY

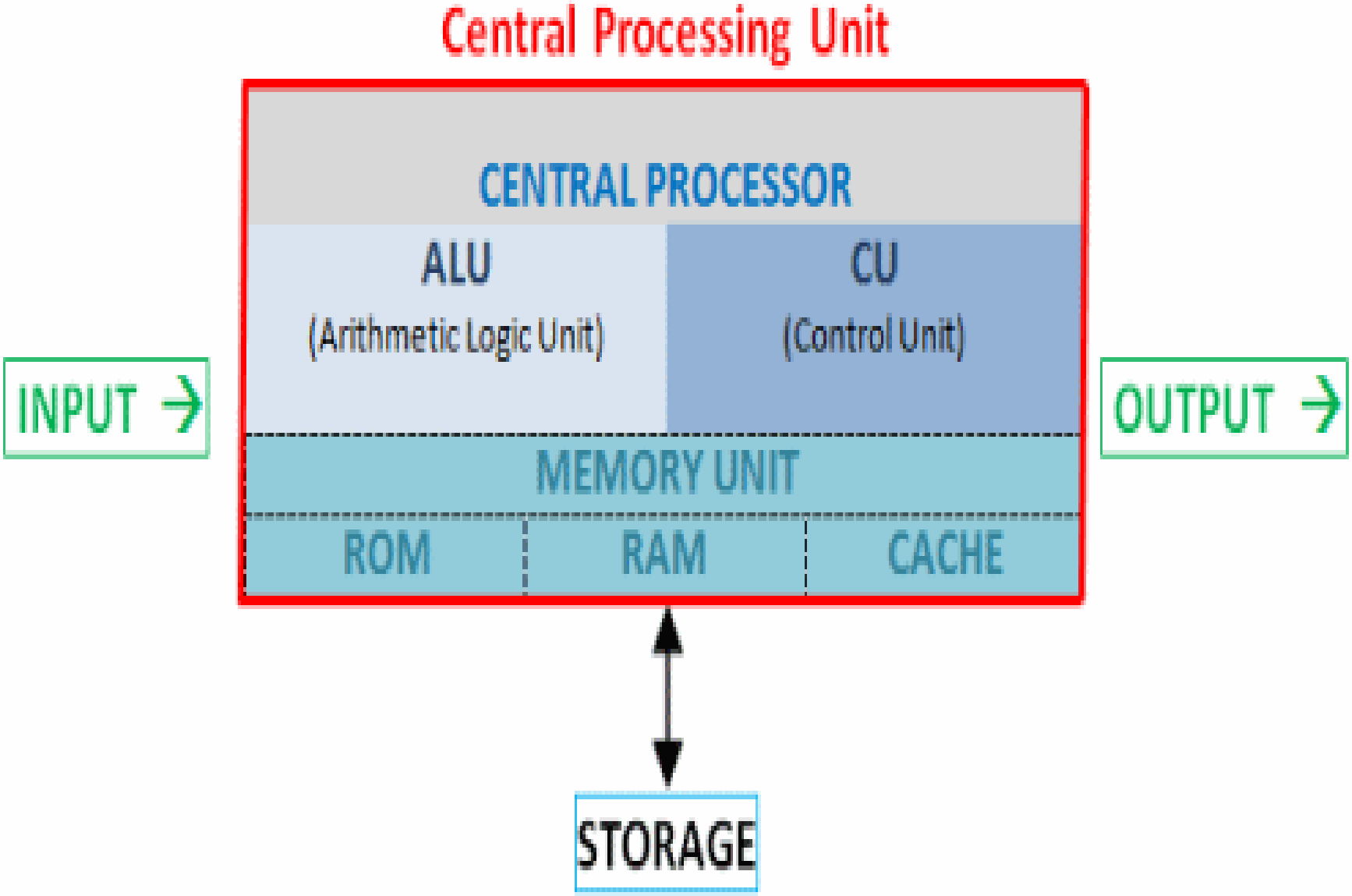
15x faster than SSD

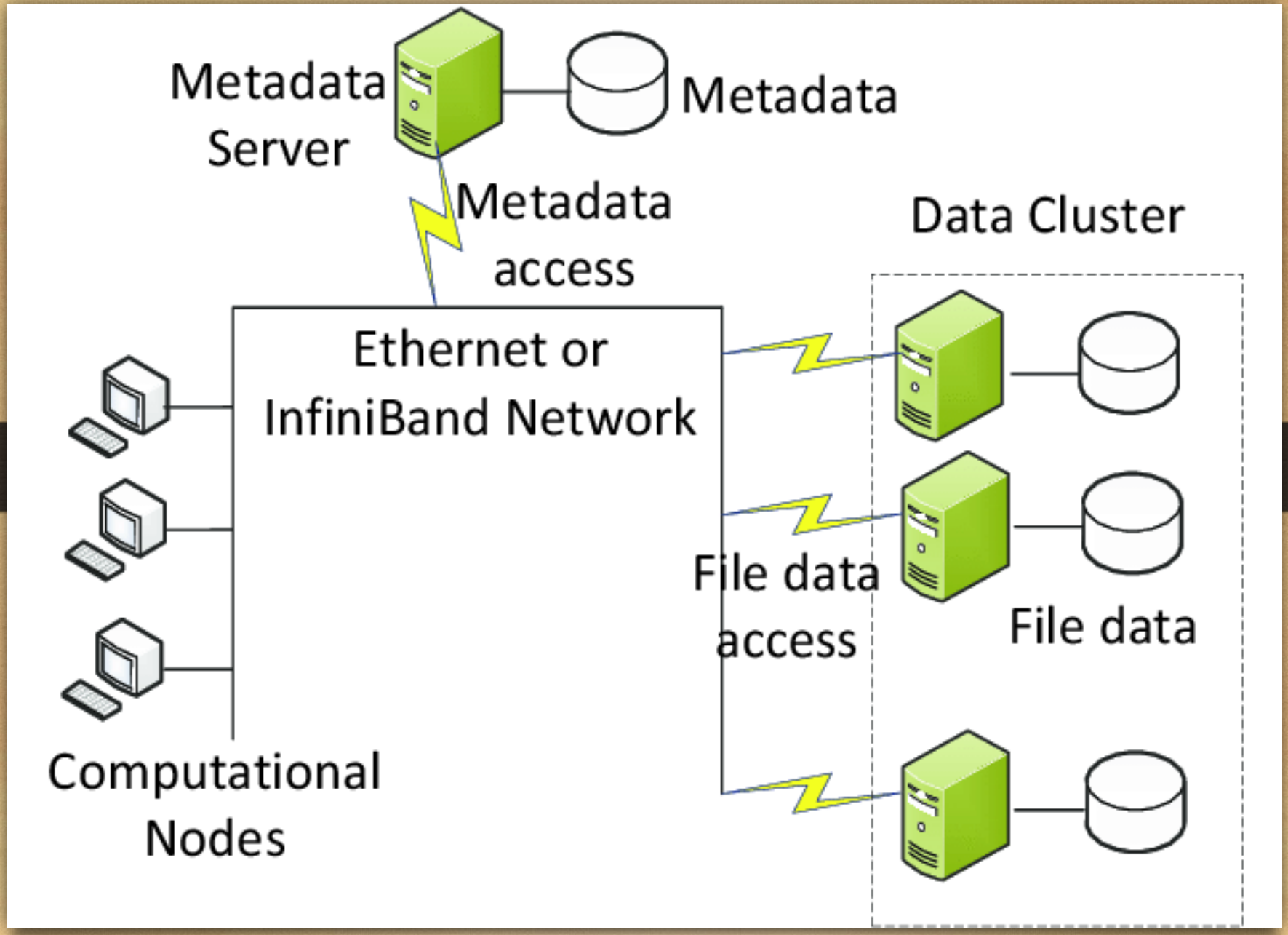
SSD

20x faster than network

NETWORK

Slowest


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

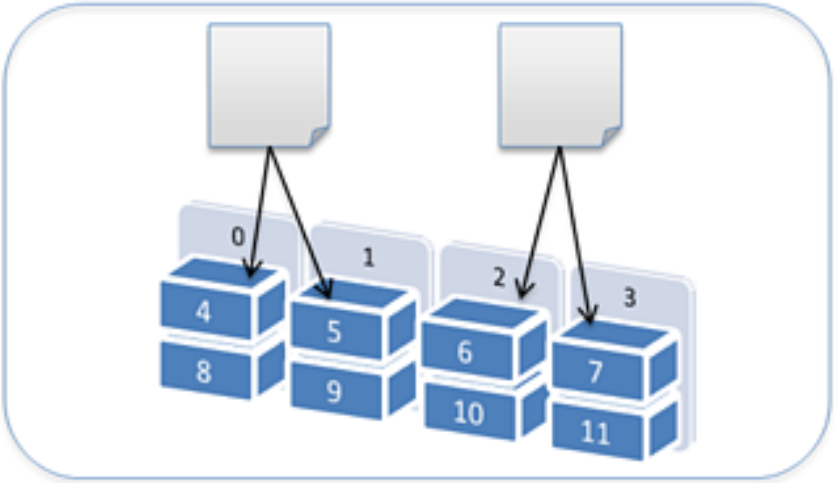

Metadata Server
Metadata
Metadata access
Data Cluster
Ethernet or InfiniBand Network
File data access
File data
Computational Nodes

HTTP(S) Interface

Object Storage

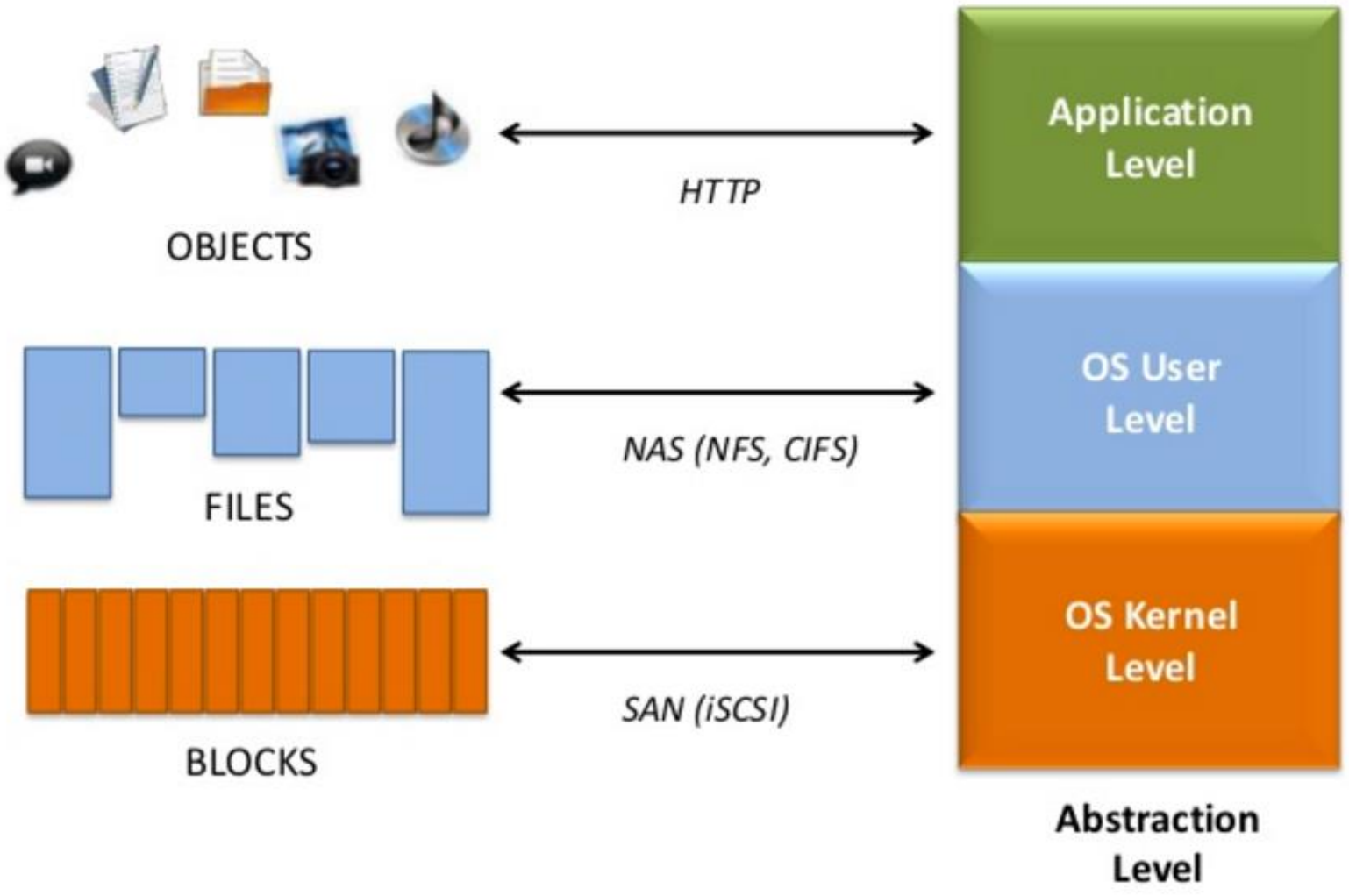
- Store virtually unlimited files.
- Maintain file revisions.
- HTTP(S) based interface.
- Files are distributed in different physical nodes.

Block Storage

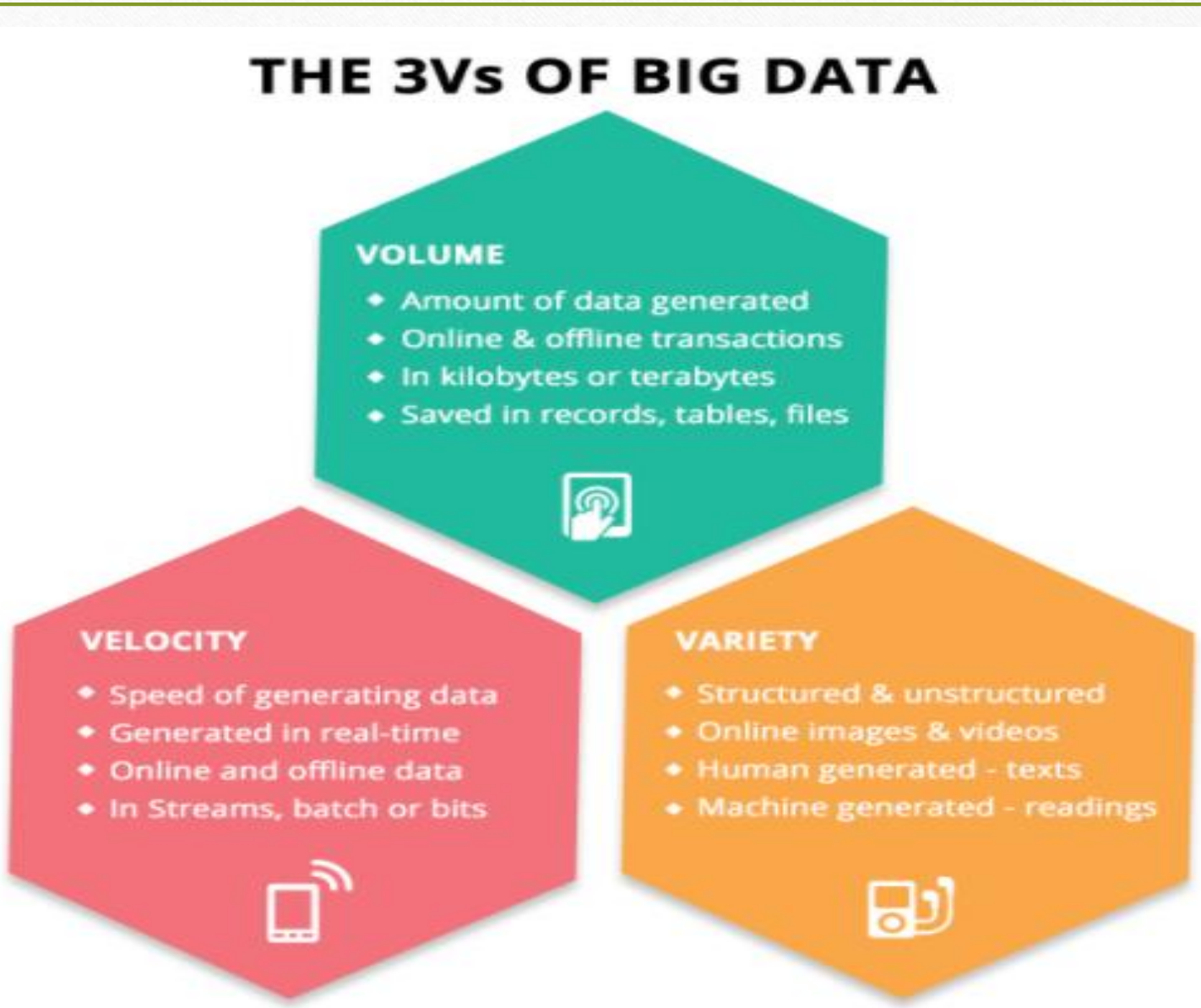


- File is split and stored in fixed sized blocks.
- Capacity can be increased by adding more nodes.
- Suitable for applications which require high IOPS, database, transactional data.

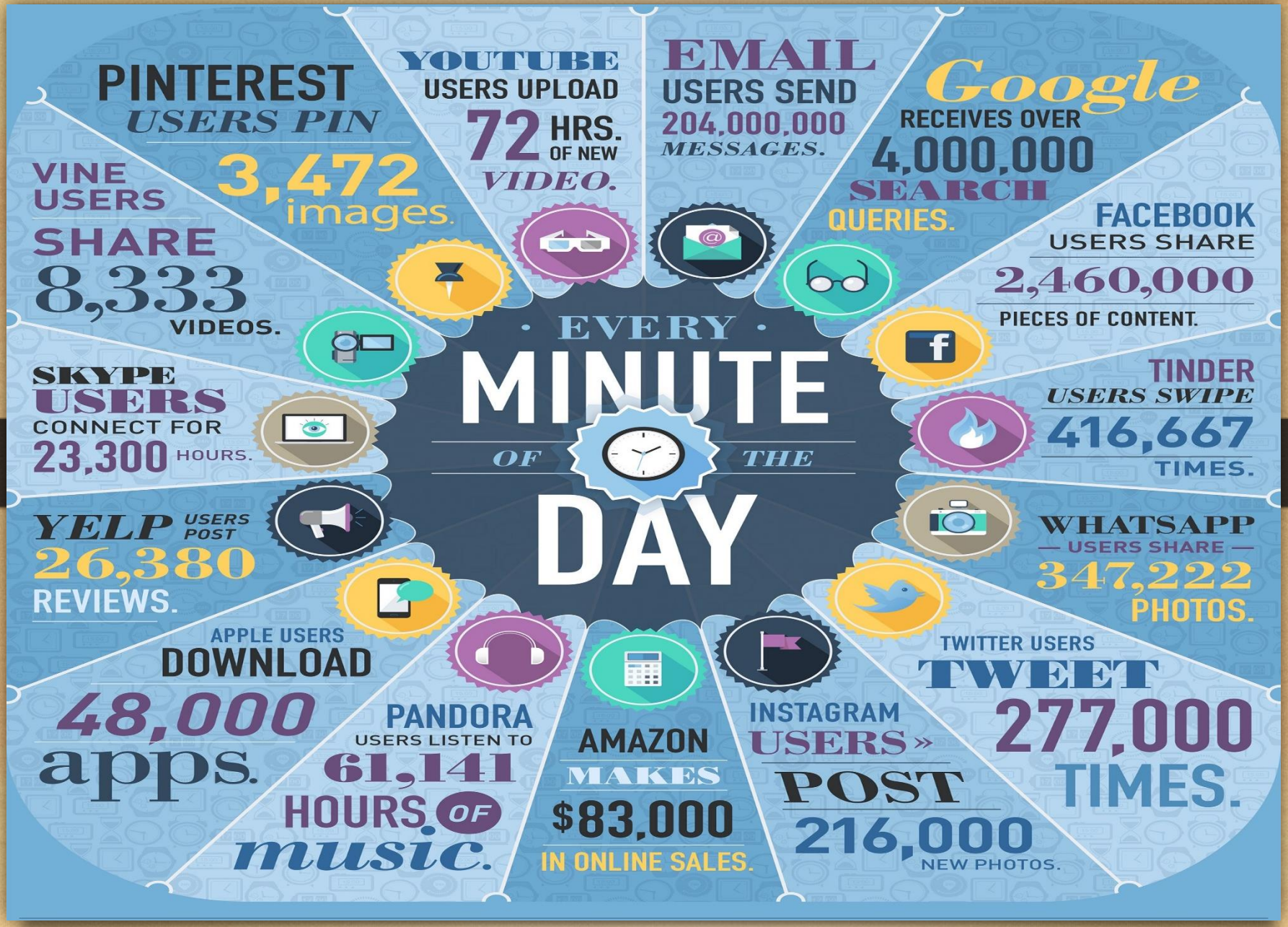
Object vs. File vs. Block Storage



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

500m

tweets are sent every day

4PB

of data created by Facebook, including

350m photos

100m hours of video watch time

65bn

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

95m

photos and videos are shared on Instagram

320bn

emails to be sent each day by 2021

306bn

emails to be sent each day by 2020

294bn

billion emails are sent

3.9bn

people use emails

4TB

of data produced by a connected car

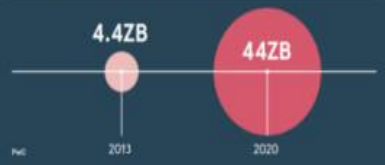
Searches made a day — 5bn

Searches made a day from Google — 3.5bn

28PB

to be generated from wearable devices by 2020

ACCUMULATED DIGITAL UNIVERSE OF DATA



RACONTEUR

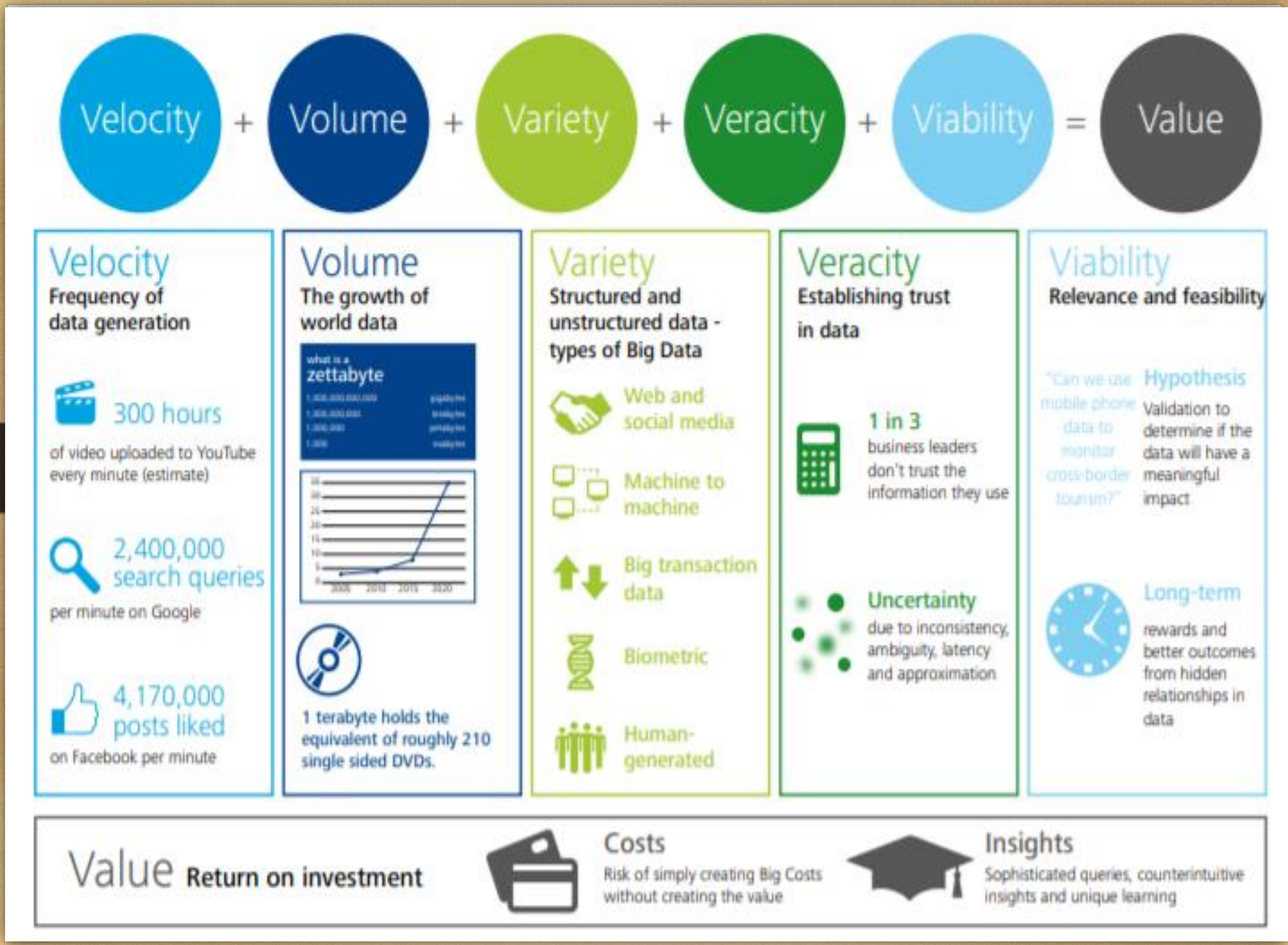
The six Vs of big data

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
what is a zettabyte
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
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

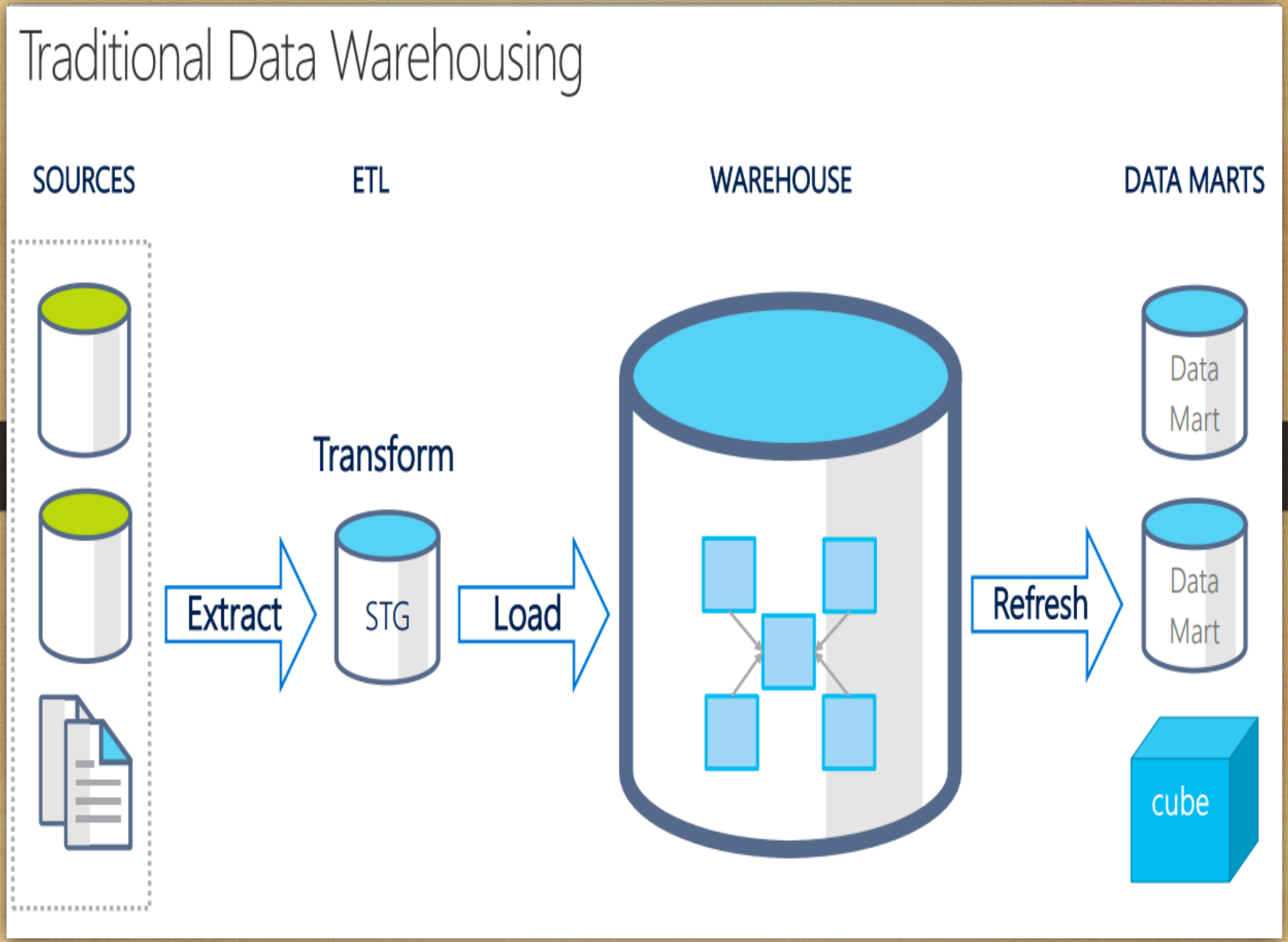
Exploit data to:

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

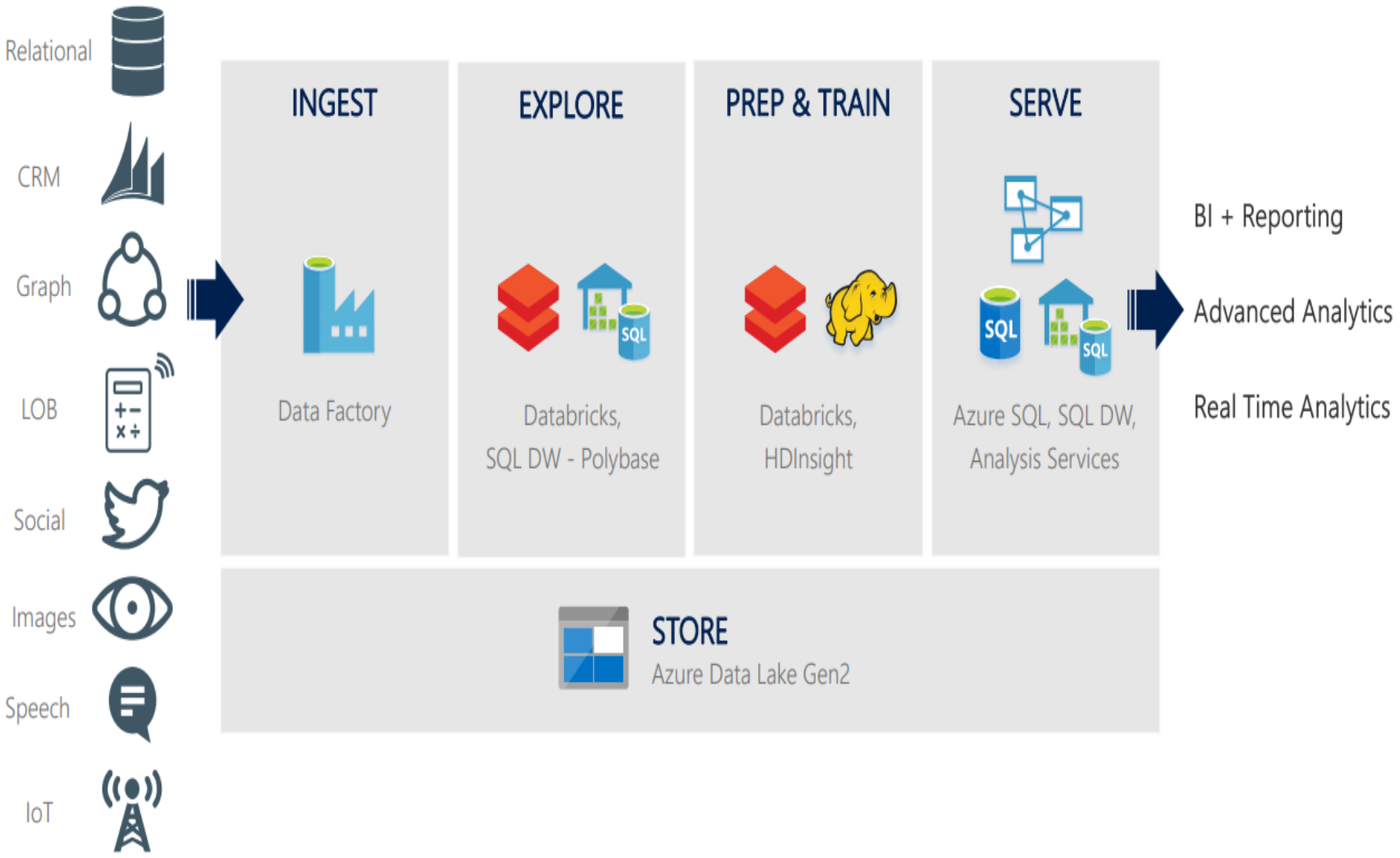
New Business Models

Capitalize on data by:

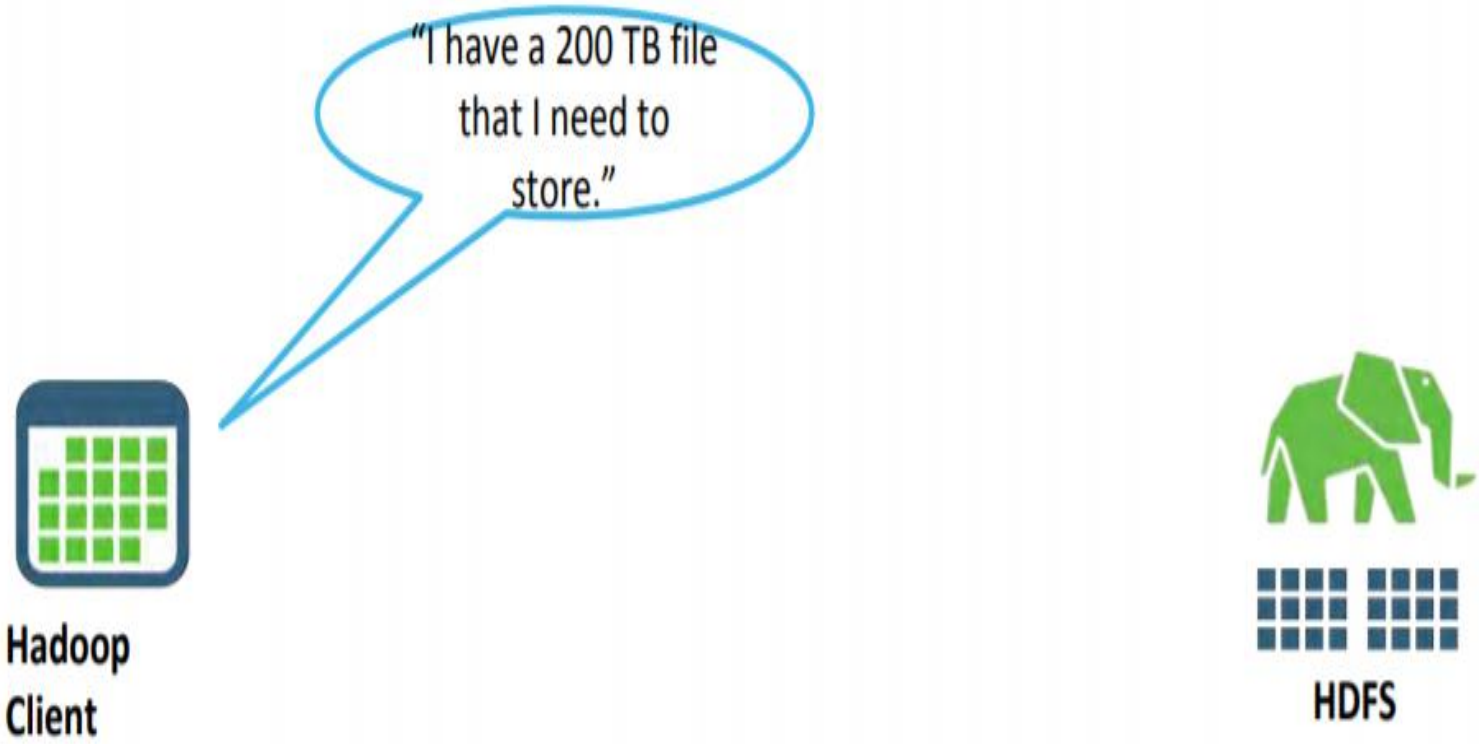
- Expanding revenue streams from existing products
- Creating new revenue streams from entirely new (data) products



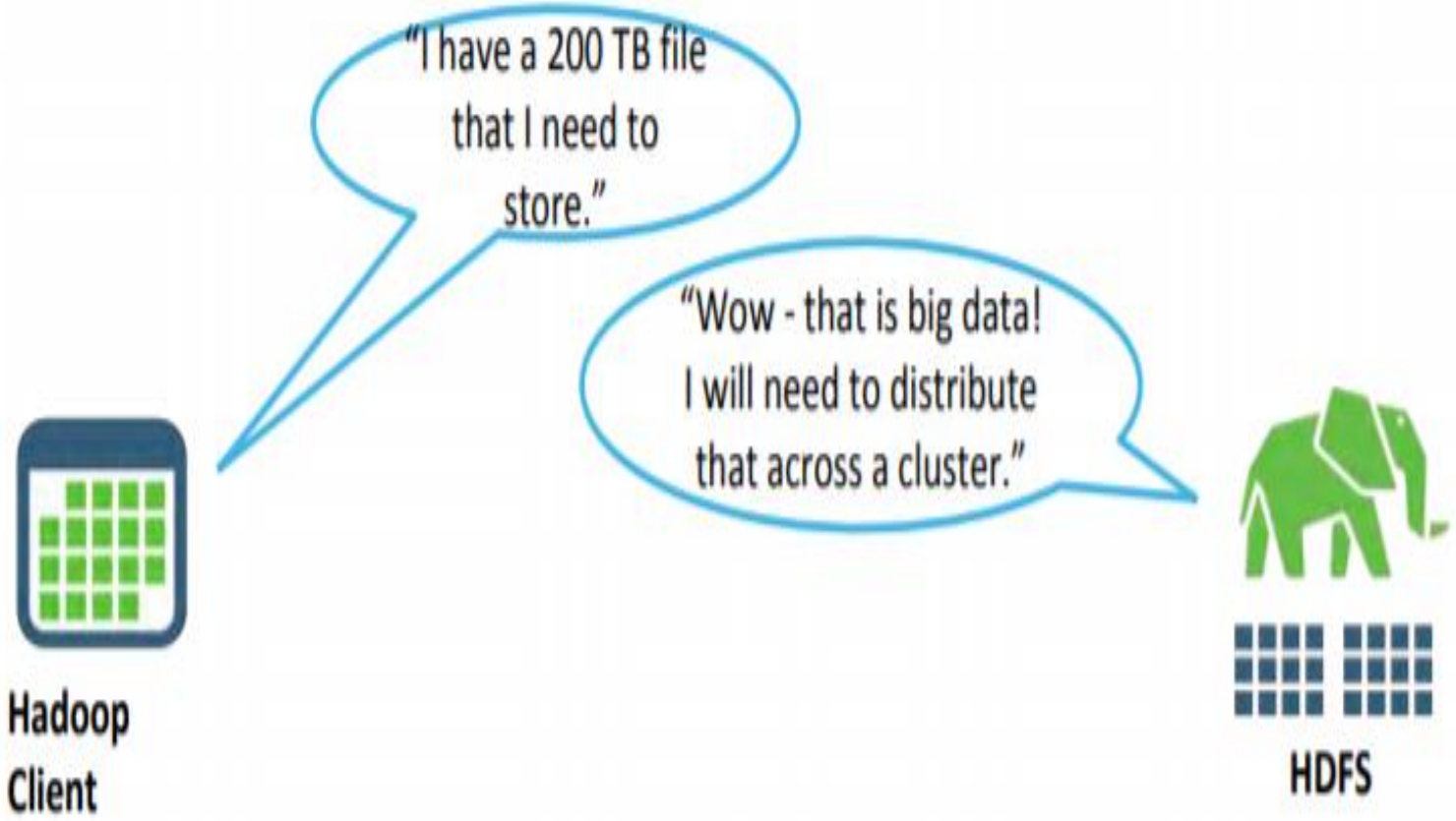
Modern Data Warehouse on Azure



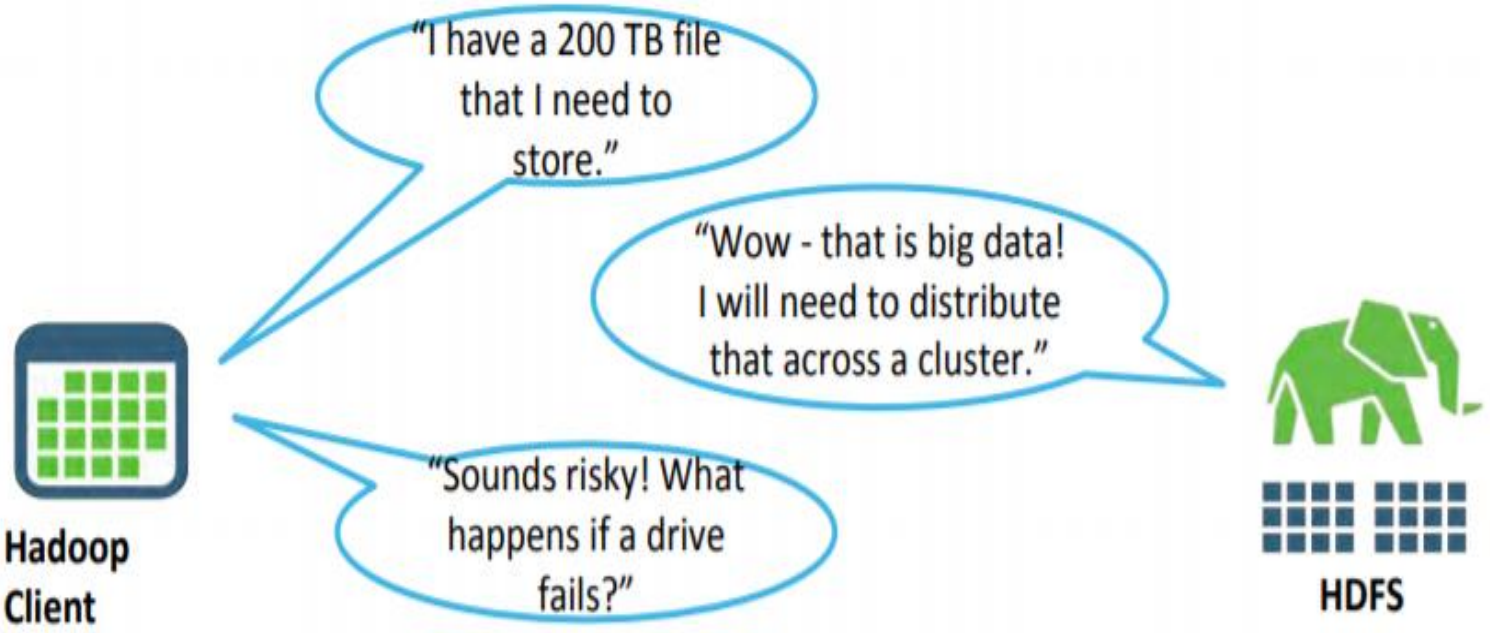
What is HDFS?



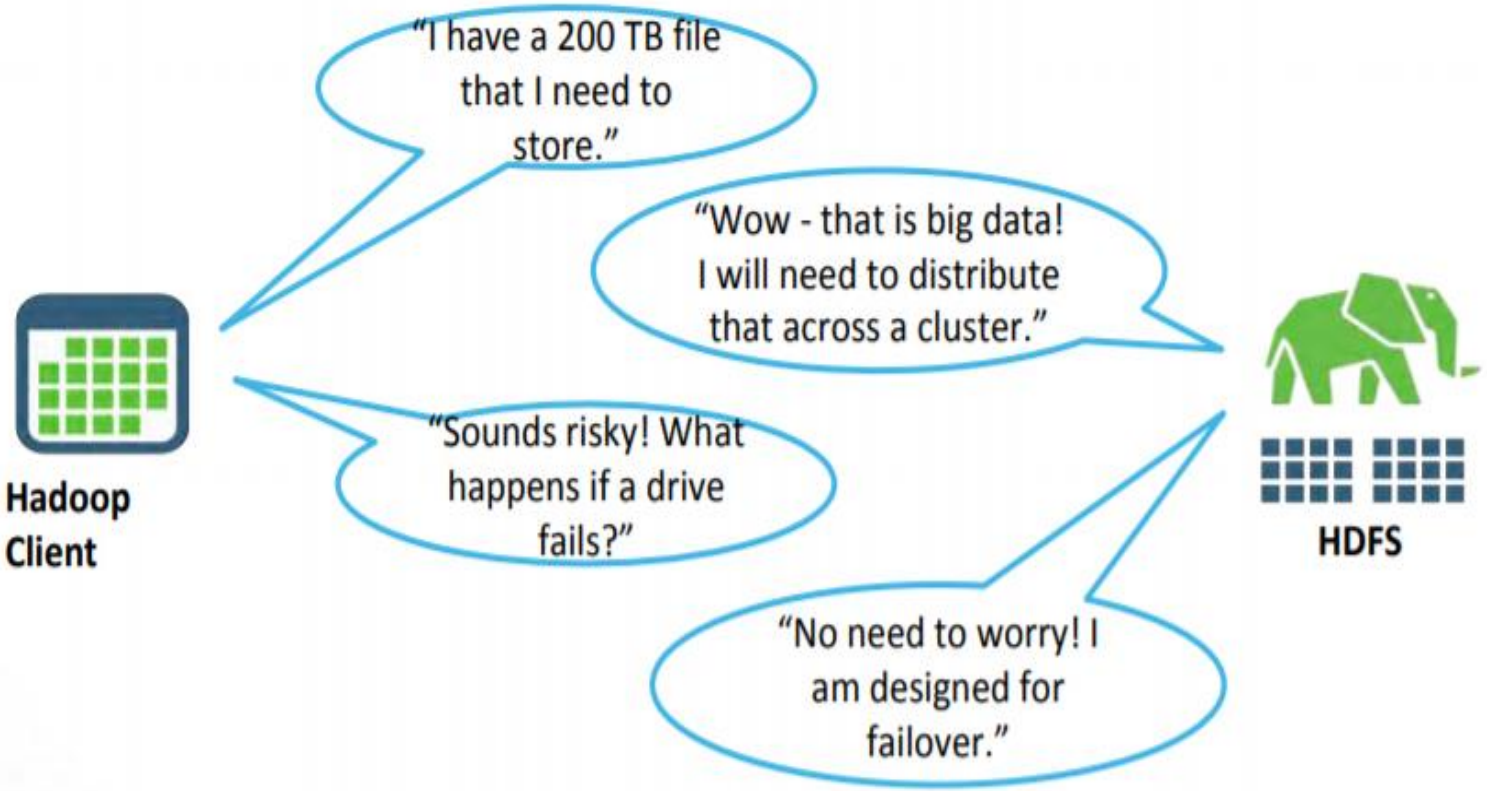
What is HDFS?


"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

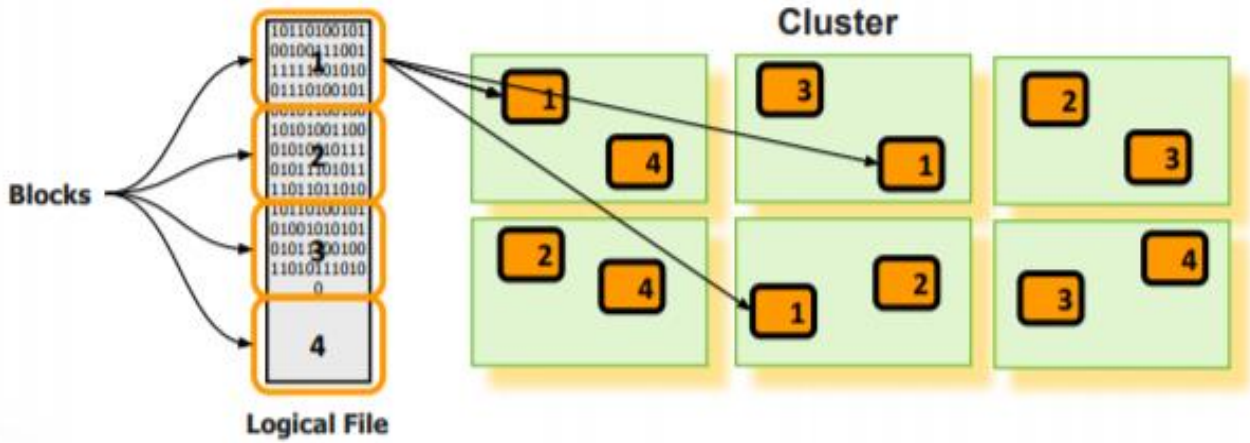
What is HDFS?


"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?"

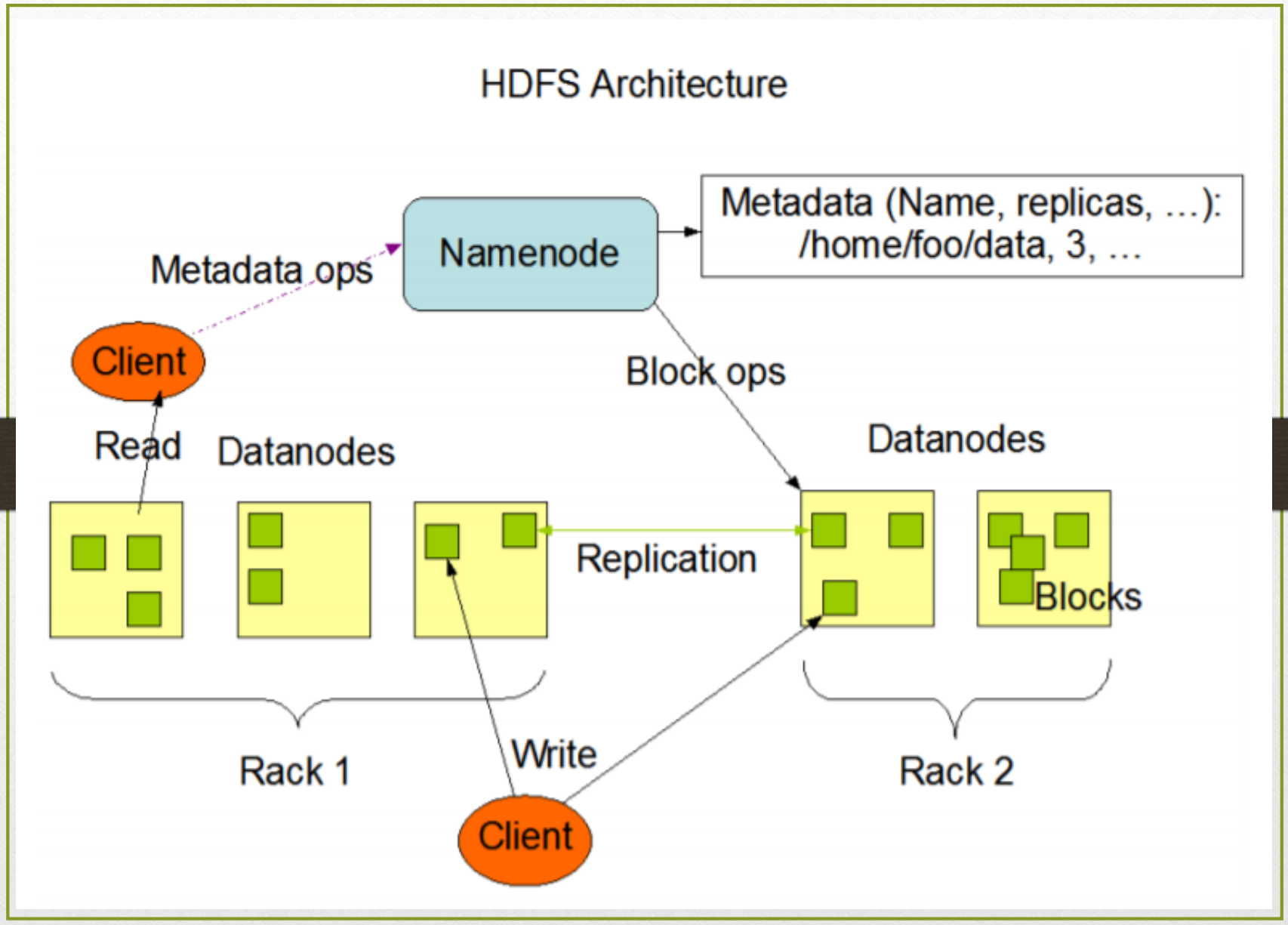


History of Hadoop

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.

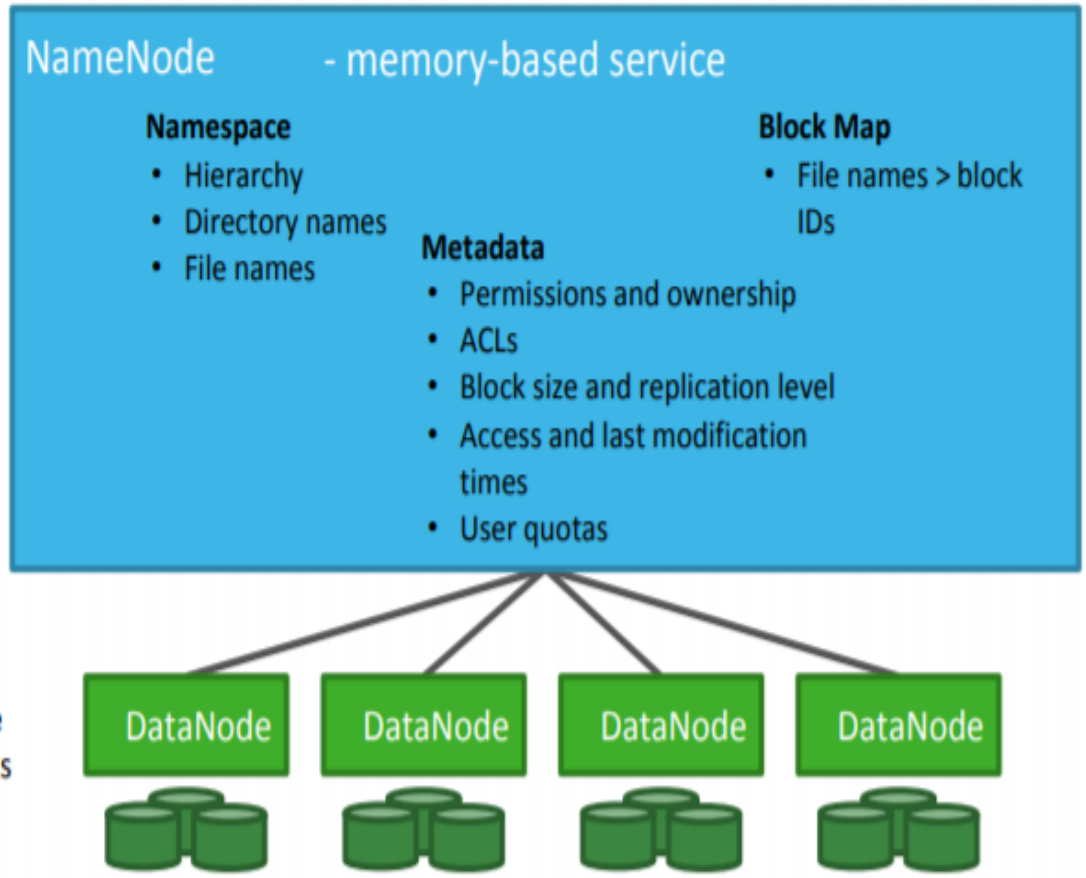
Map/Reduce History

- **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.



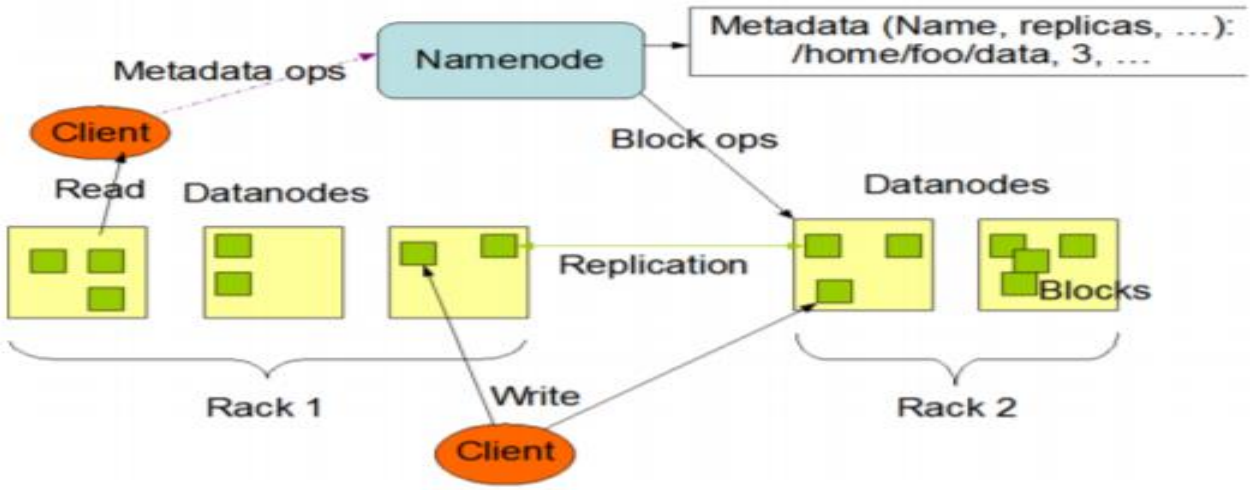
Working of HDFS

- 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

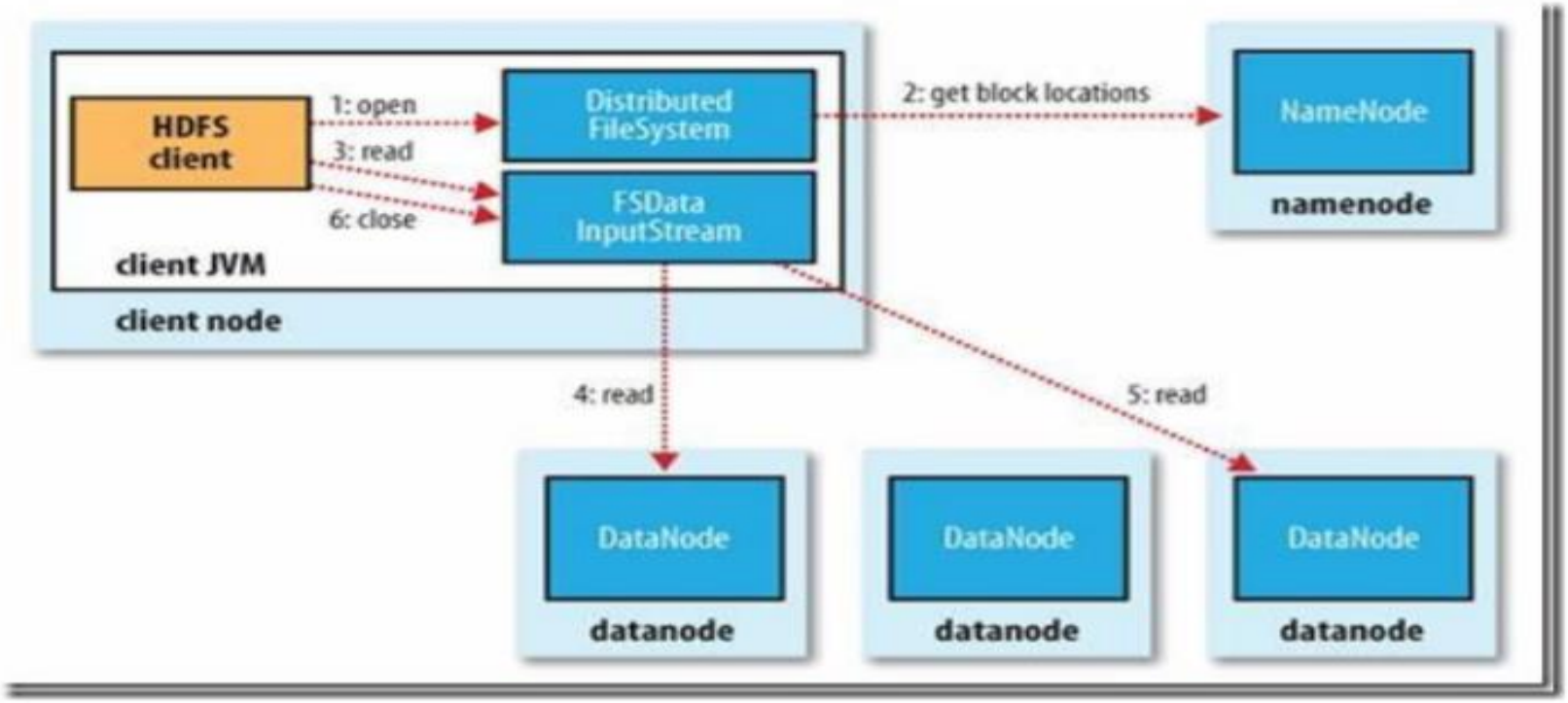
- NameNode responds to the client and requests to store 150MB data in the nodes which has space.
- Client store all the txt files in different DataNodes. However, all the files need not be in sequence order.

- 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.

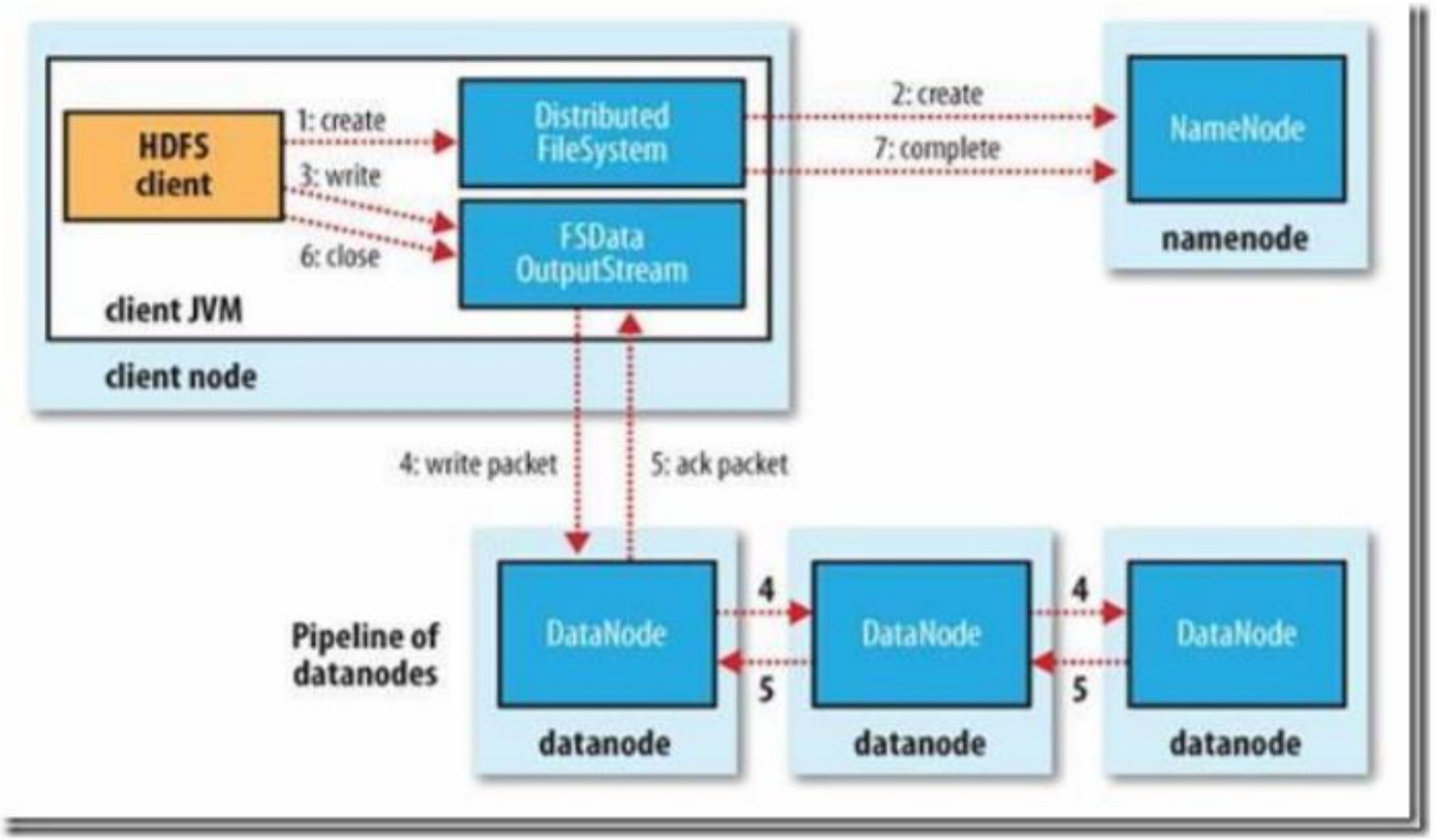


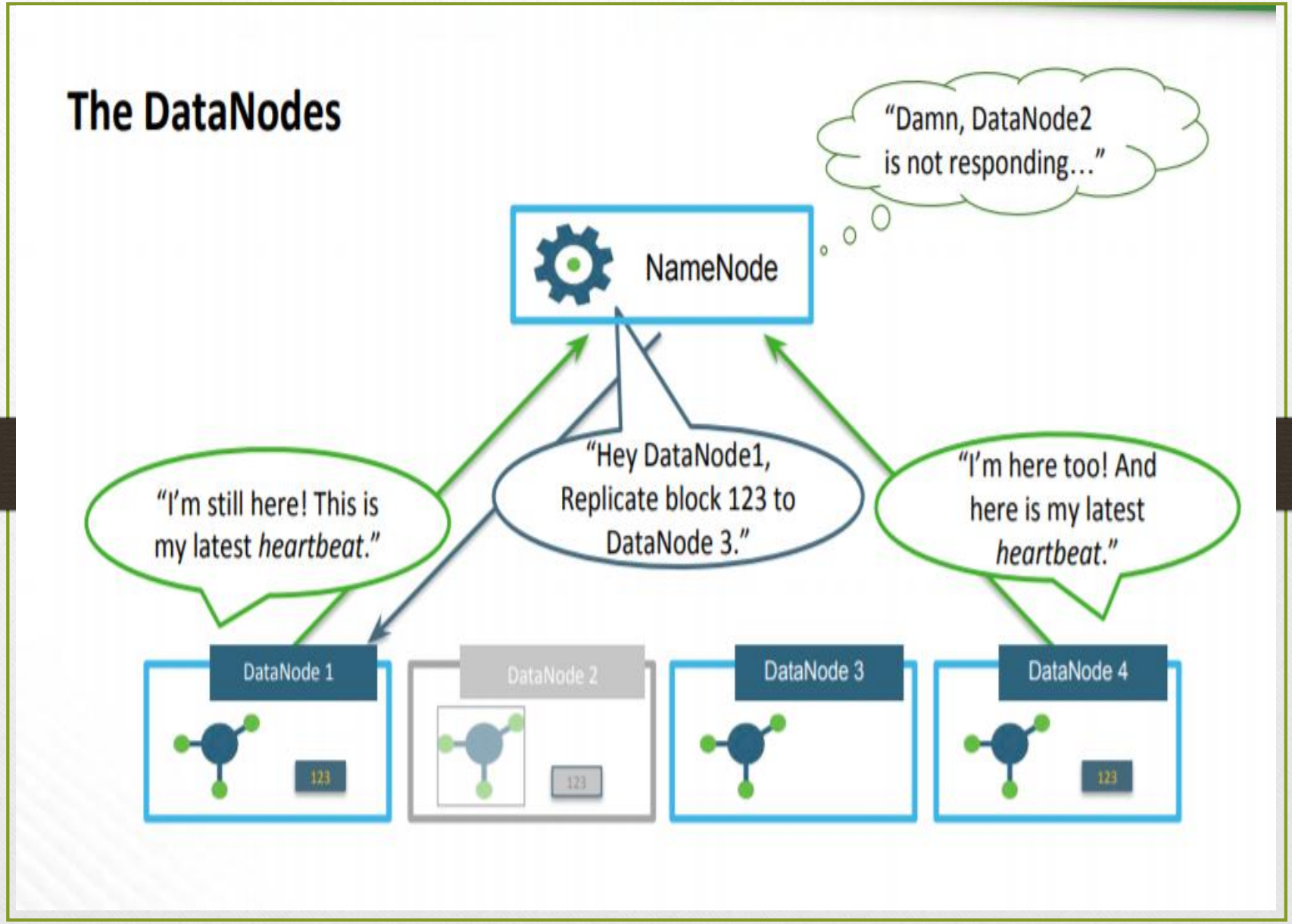
Data Reading Process in HDFS

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.

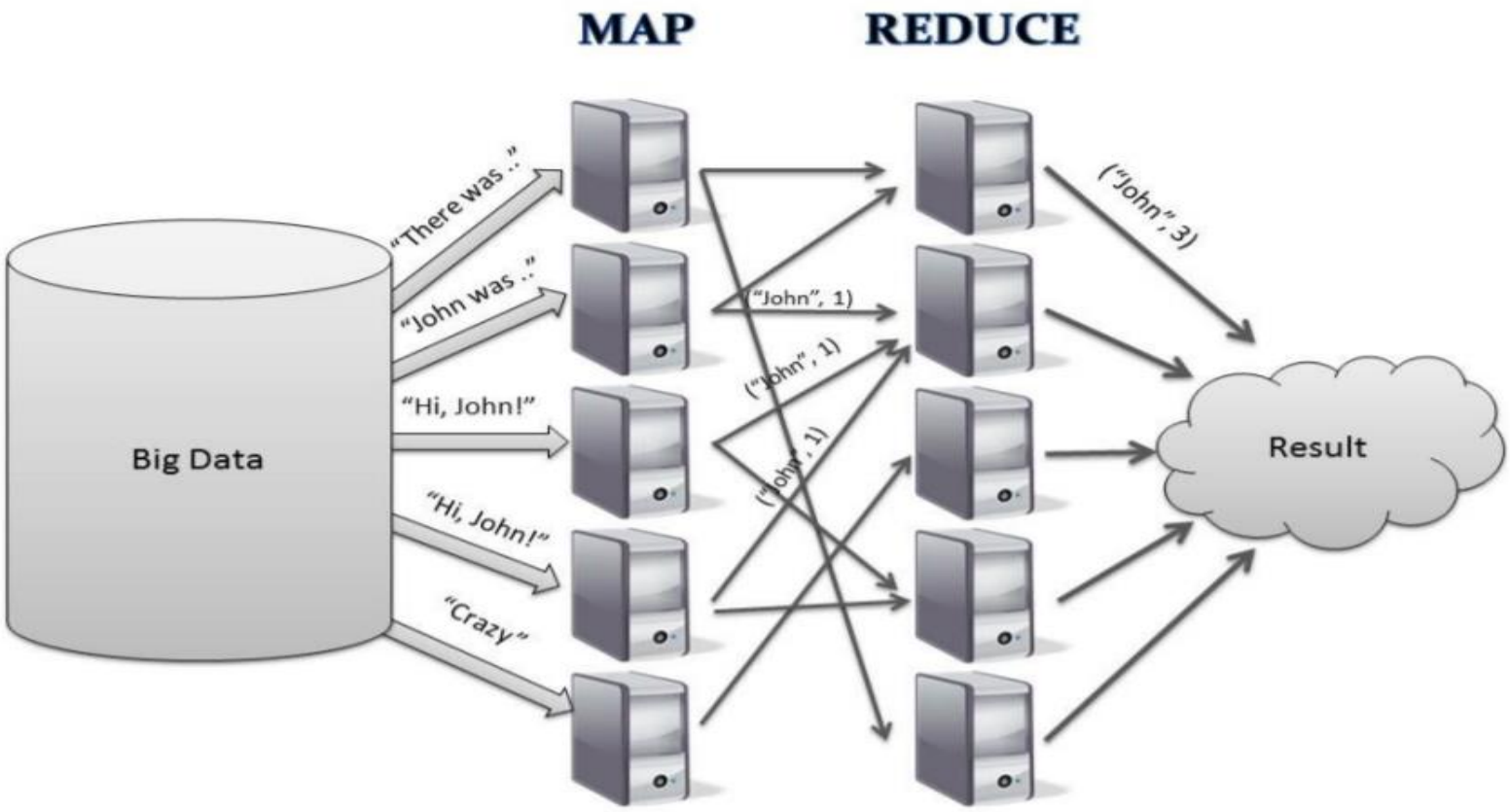


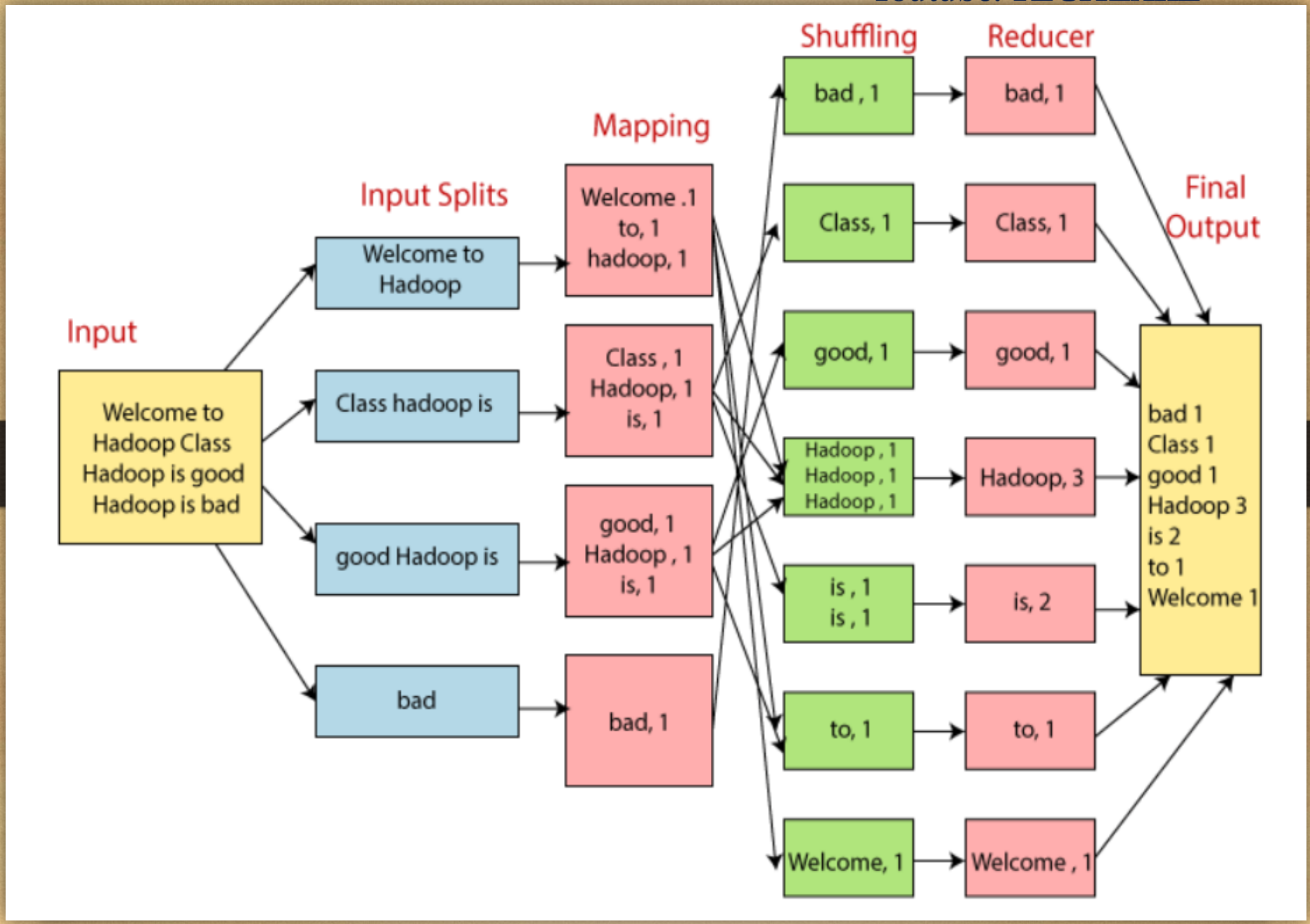
The structure of HDFS reading process is similar to the writing process. There are the following seven steps:




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

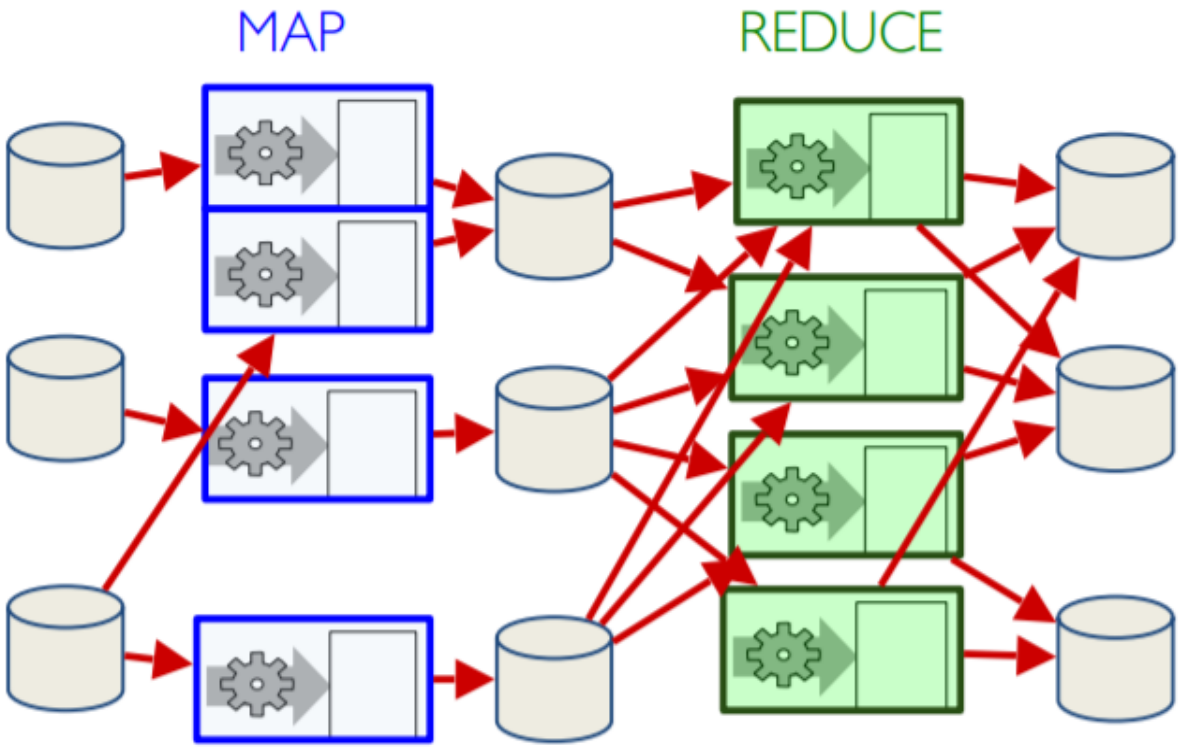

Input
Welcome to Hadoop Class Hadoop is good Hadoop is bad
Input Splits
Welcome to Hadoop
Class hadoop is
good Hadoop is
bad
Mapping
Welcome .1 to, 1 hadoop, 1
Class , 1 Hadoop, 1 is, 1
good, 1 Hadoop , 1 is, 1
bad, 1
Shuffling
bad , 1
Class, 1
good, 1
Hadoop , 1 Hadoop , 1 Hadoop , 1
is , 1 is , 1
to, 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

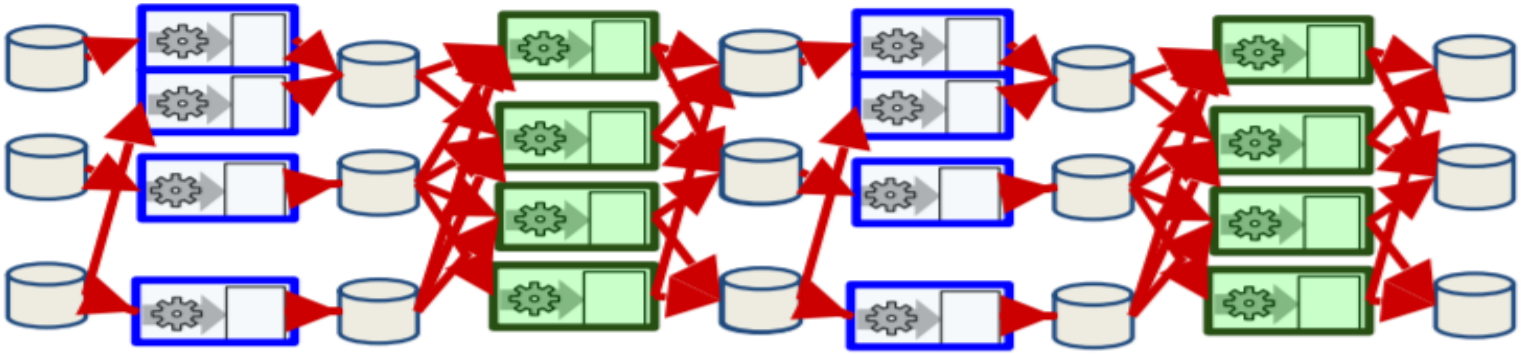


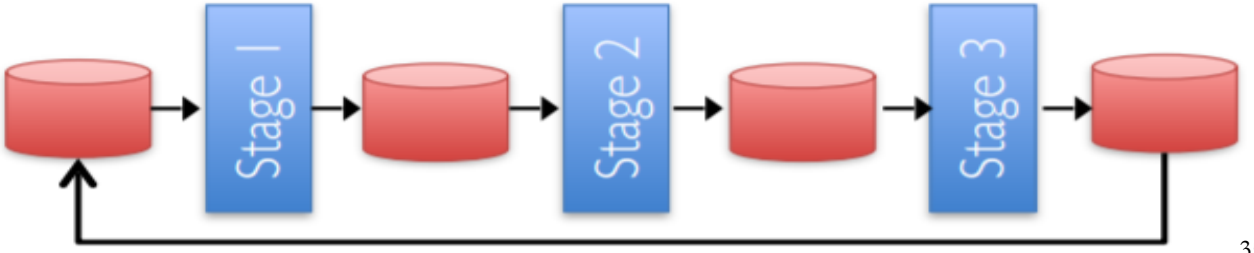
Why Spark?

Iterative Jobs

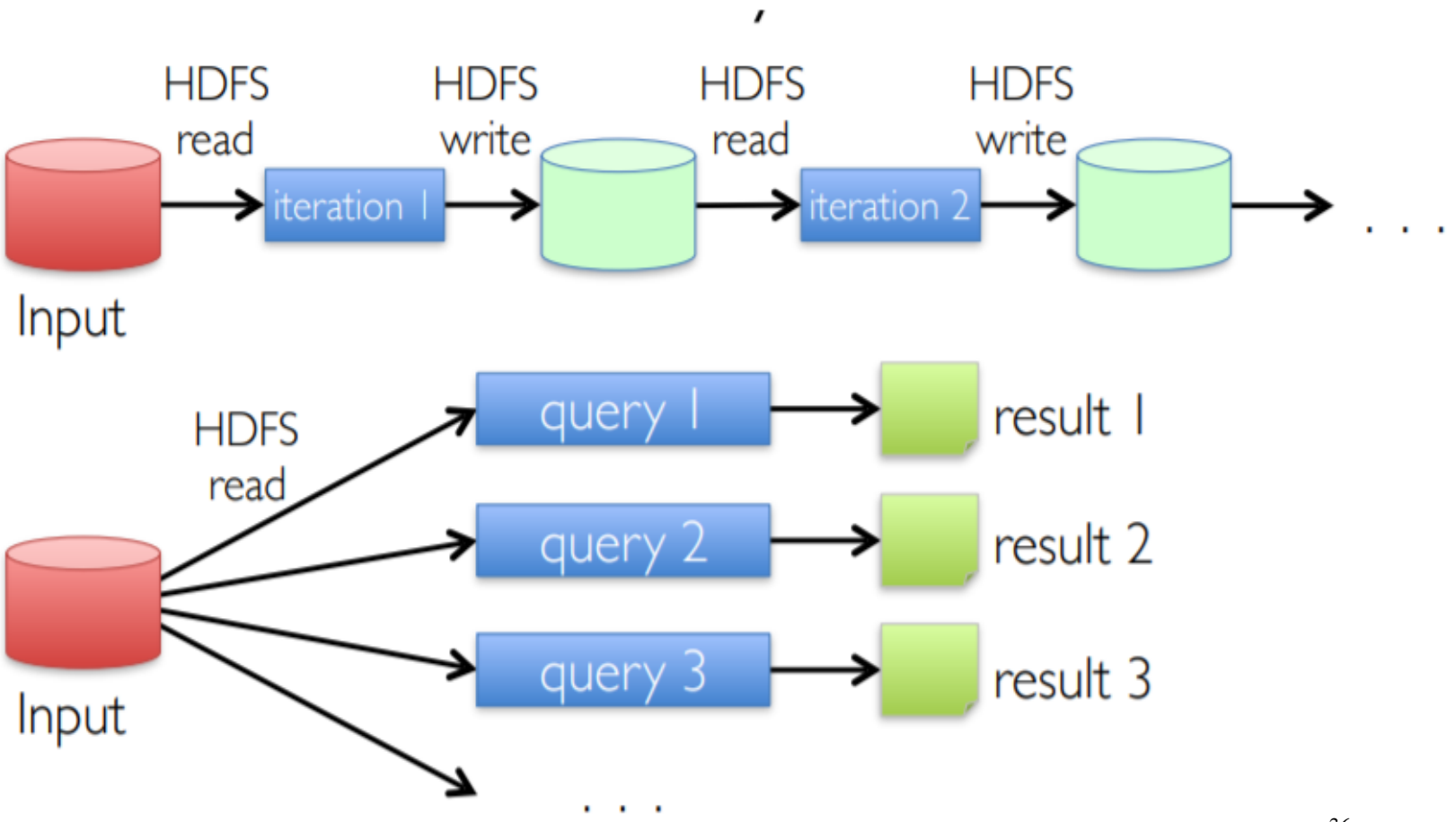
- Disk I/O for each repetition

  → Slow when executing many small iterations



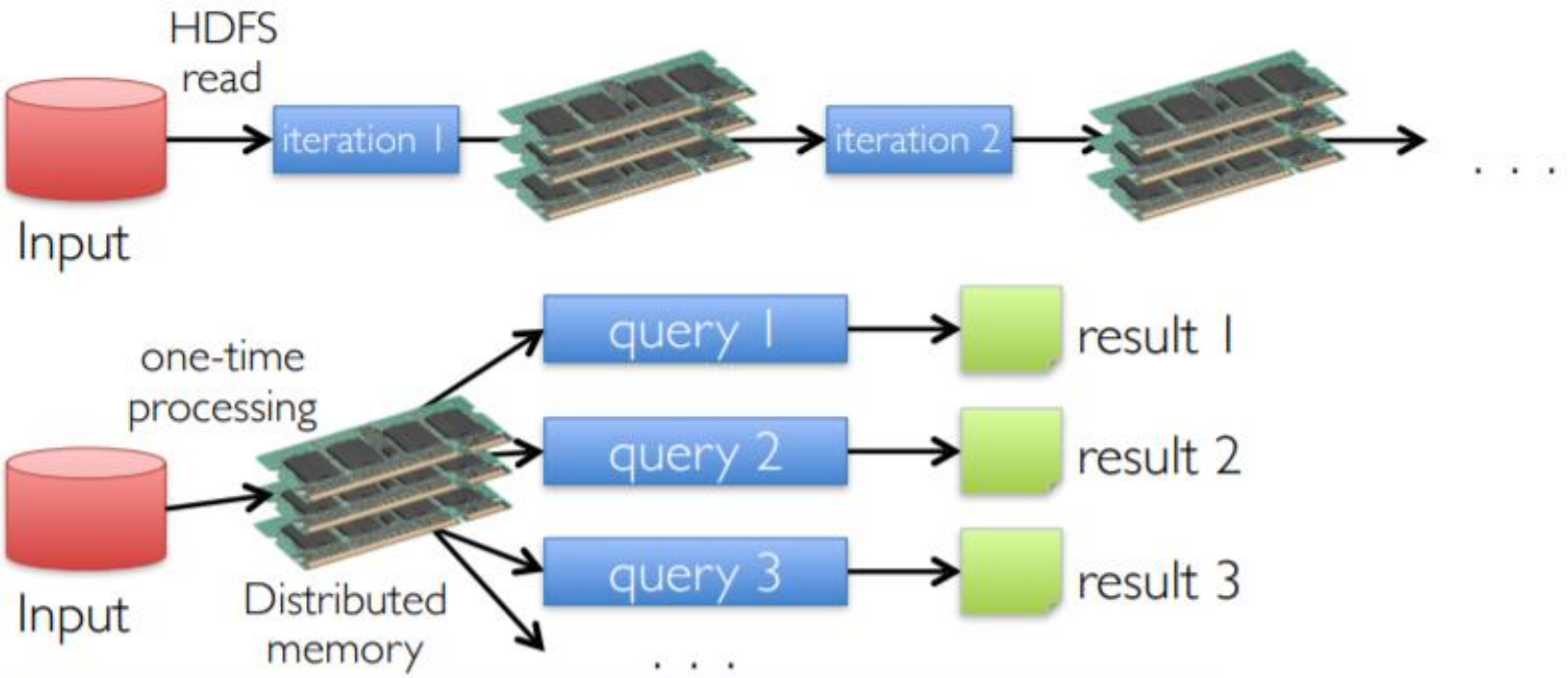


Replace Disk with Memory

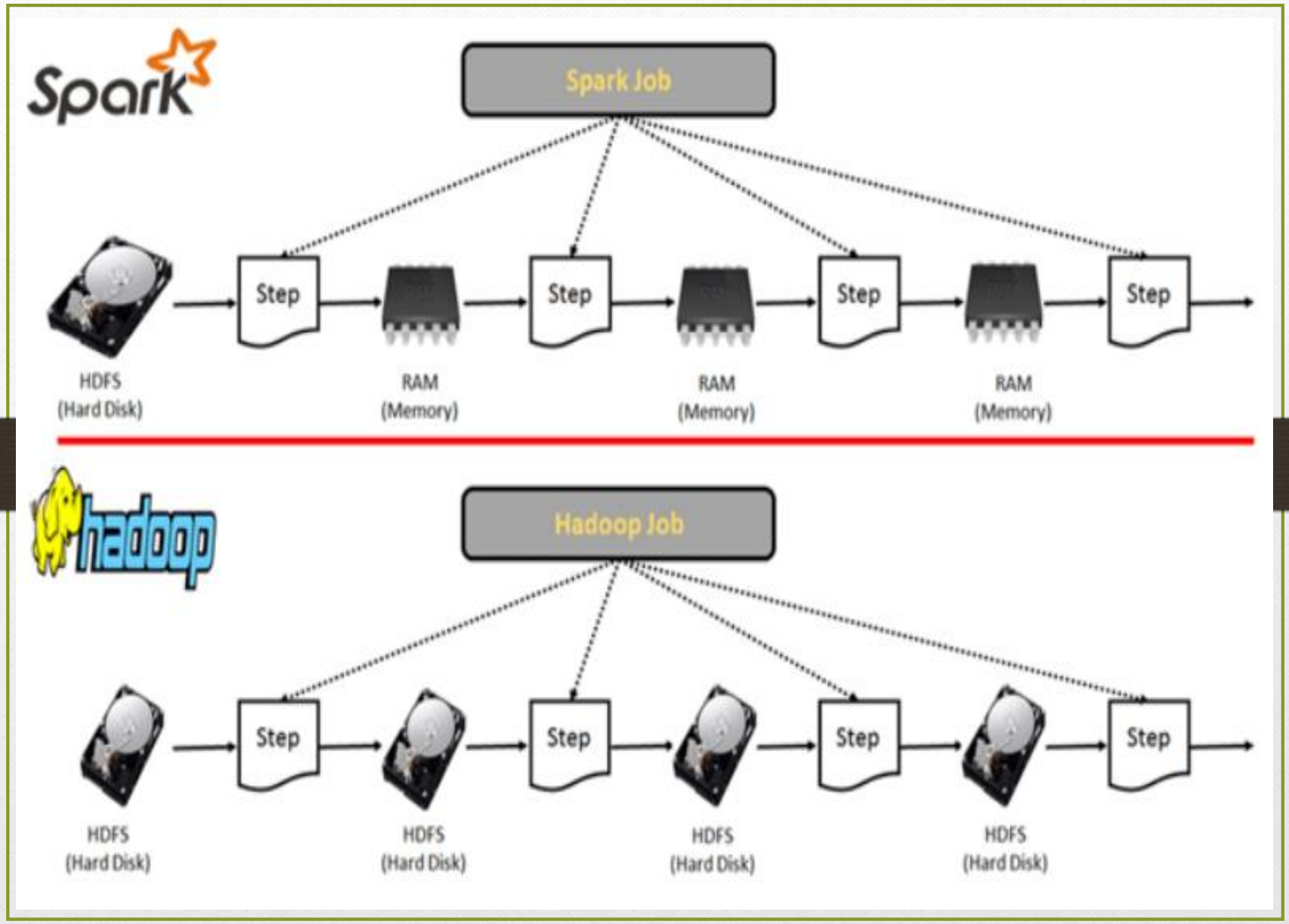


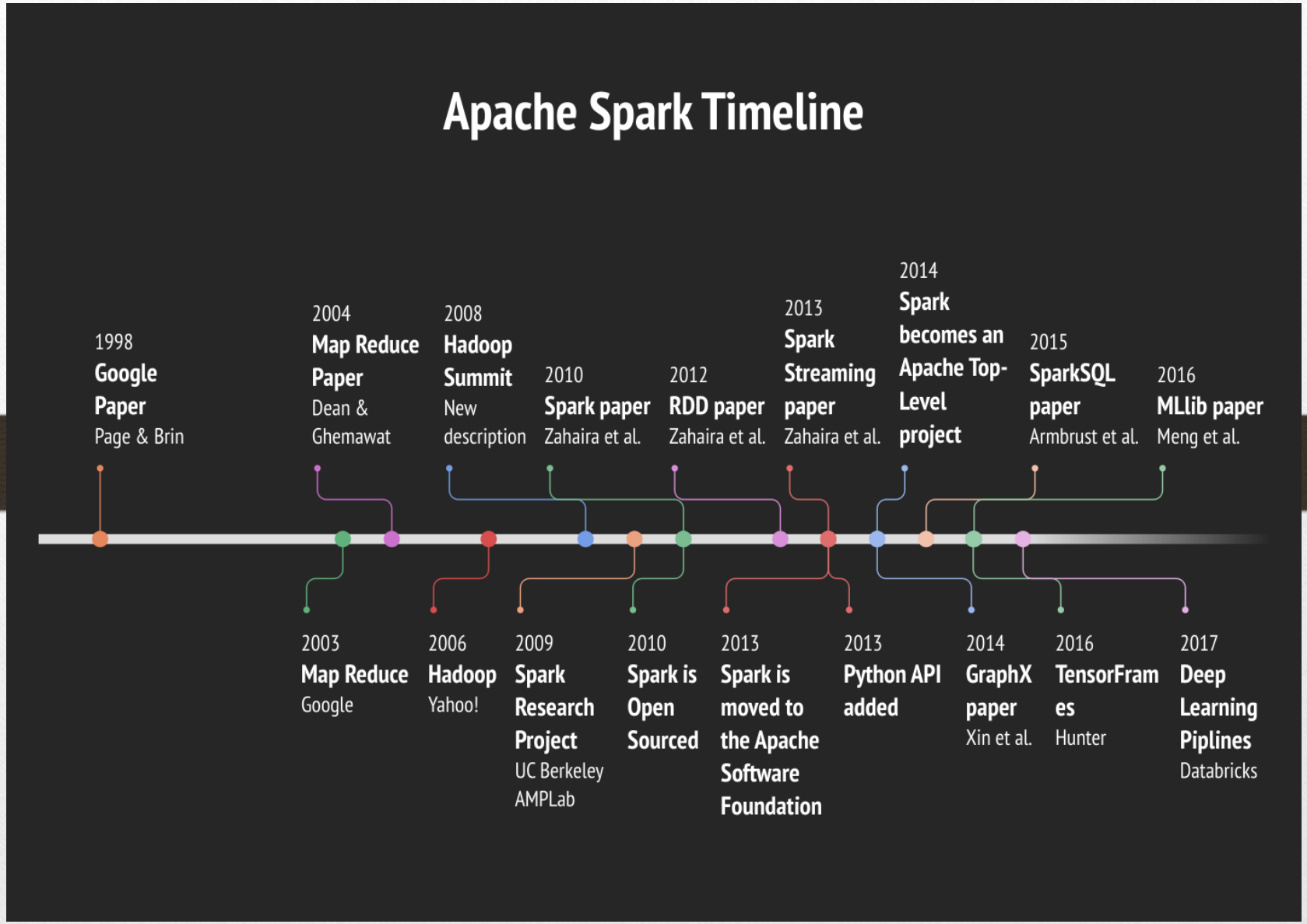
Replace Disk with Memory

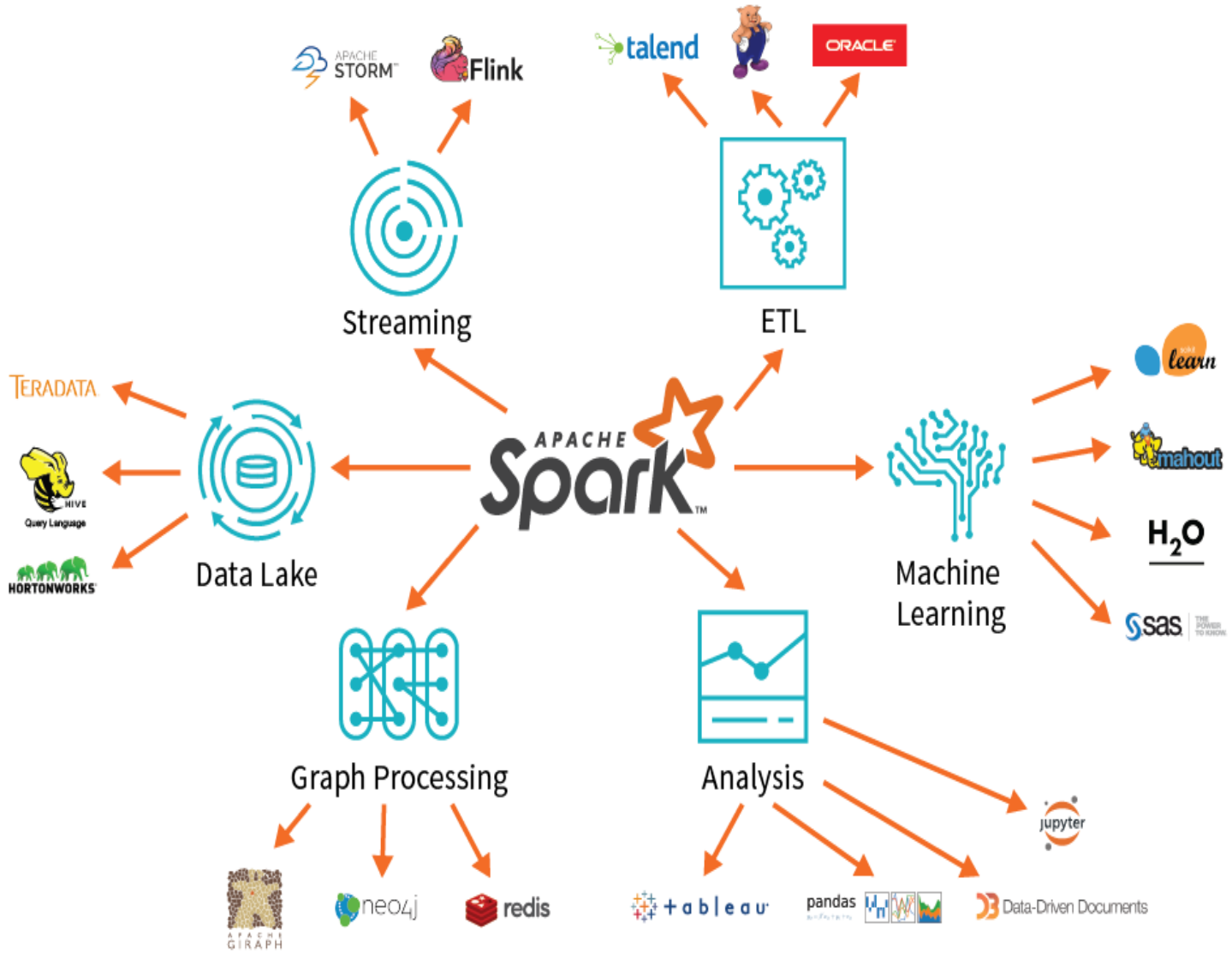
In-Memory Data Sharing



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)

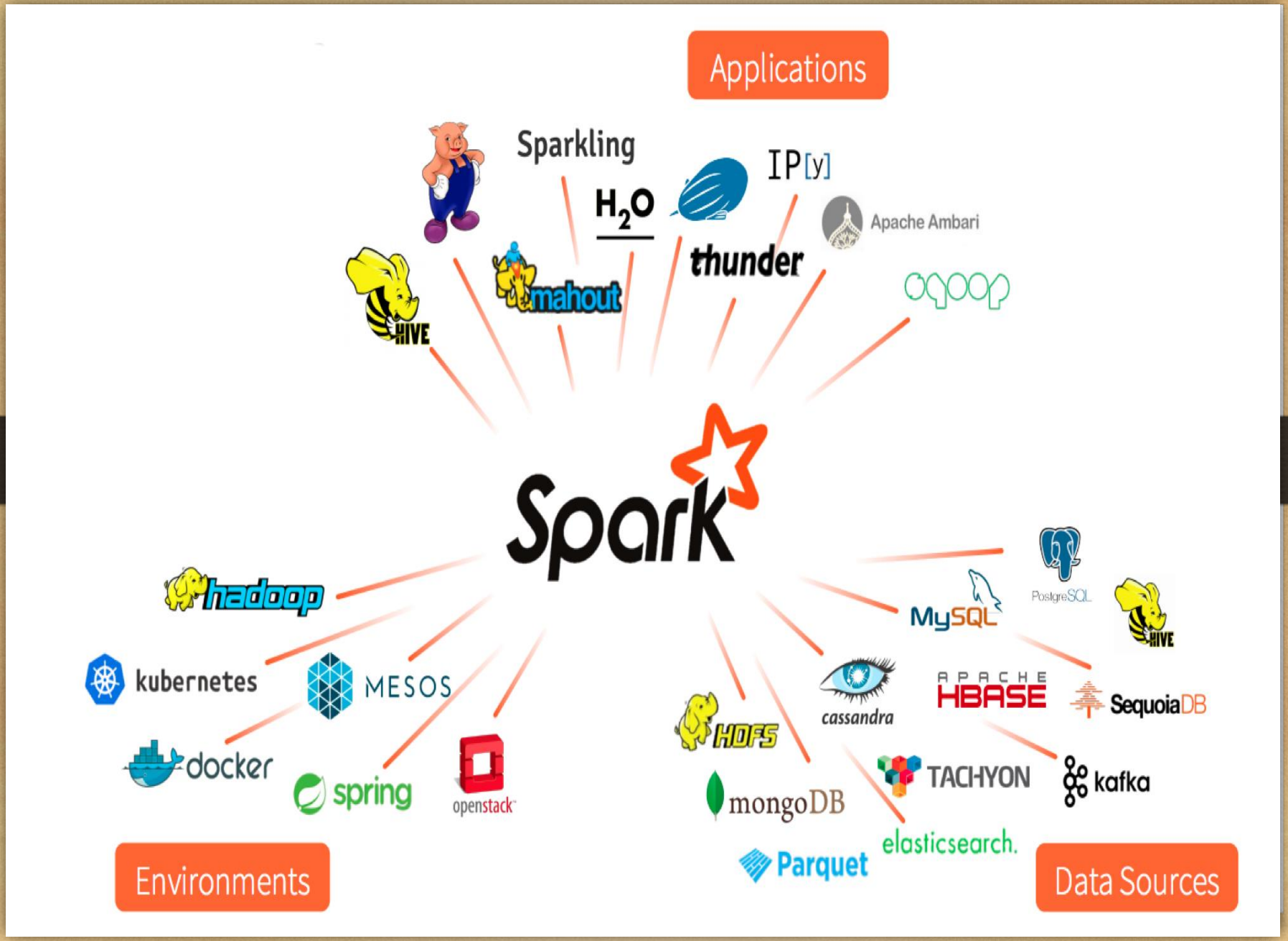

Apache Spark Timeline
1998
Google Paper
Page & Brin
2003
Map Reduce
Google
2004
Map Reduce Paper
Dean & Ghemawat
2006
Hadoop
Yahoo!
2008
Hadoop Summit
New description
2009
Spark Research Project
UC Berkeley AMPLab
2010
Spark paper
Zahaira et al.
2010
Spark is Open Sourced
2012
RDD paper
Zahaira et al.
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


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

In-Memory Processing

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

→ Spark distributed execution engine


Applications
Sparkling
H2O
IP[y]
Apache Ambari
thunder
mahout
HIVE
sqoop
Spark
hadoop
kubernetes
MESOS
docker
spring
openstack
MySQL
PostgreSQL
HIVE
HDFS
cassandra
APACHE HBASE
SequoiaDB
TACHYON
kafka
mongoDB
elasticsearch.
Parquet
Environments
Data Sources

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*

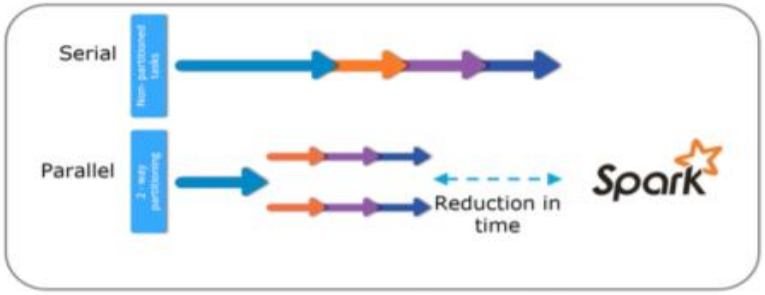
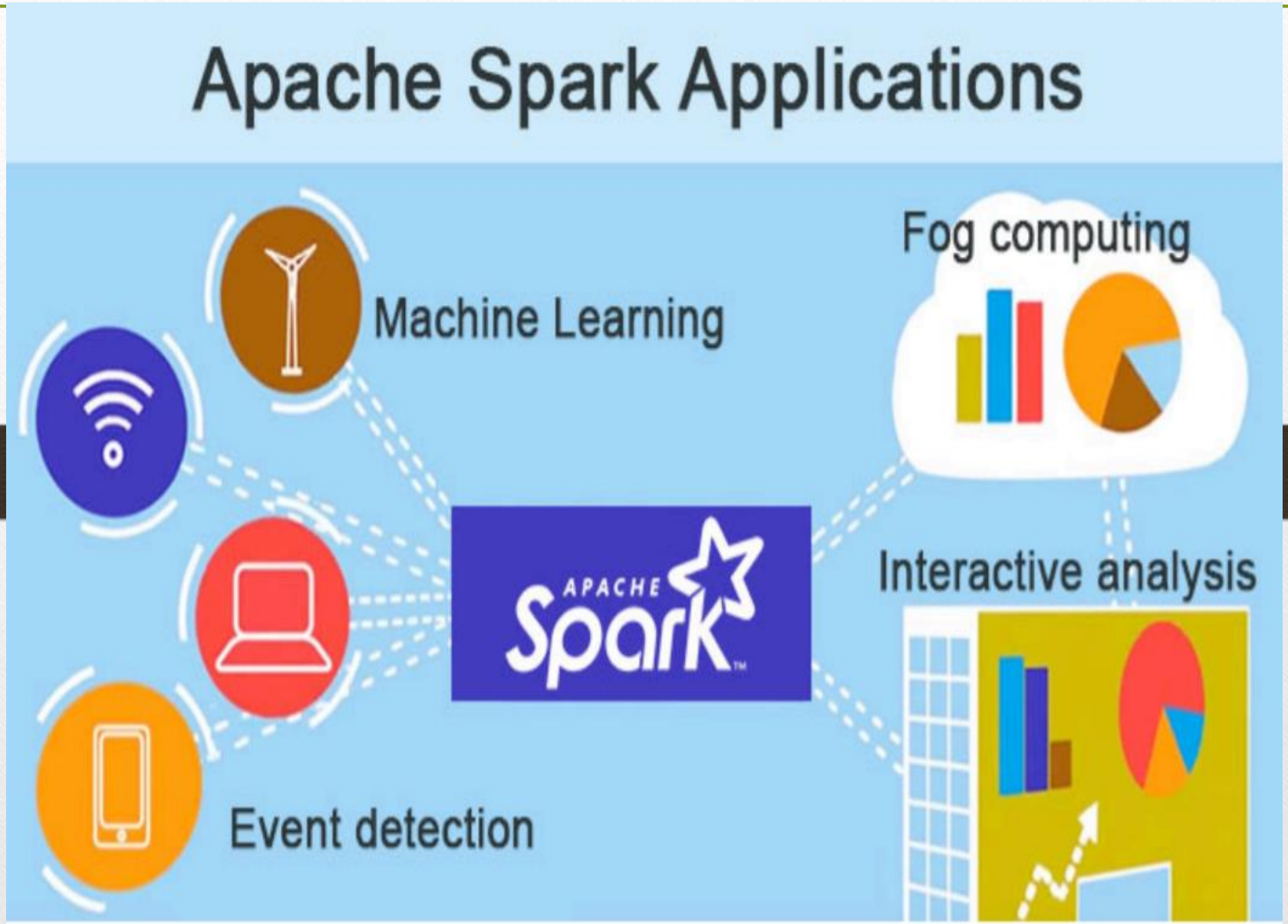


Figure: *Data Parallelism In Spark*


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

Uses Cases

NETFLIX

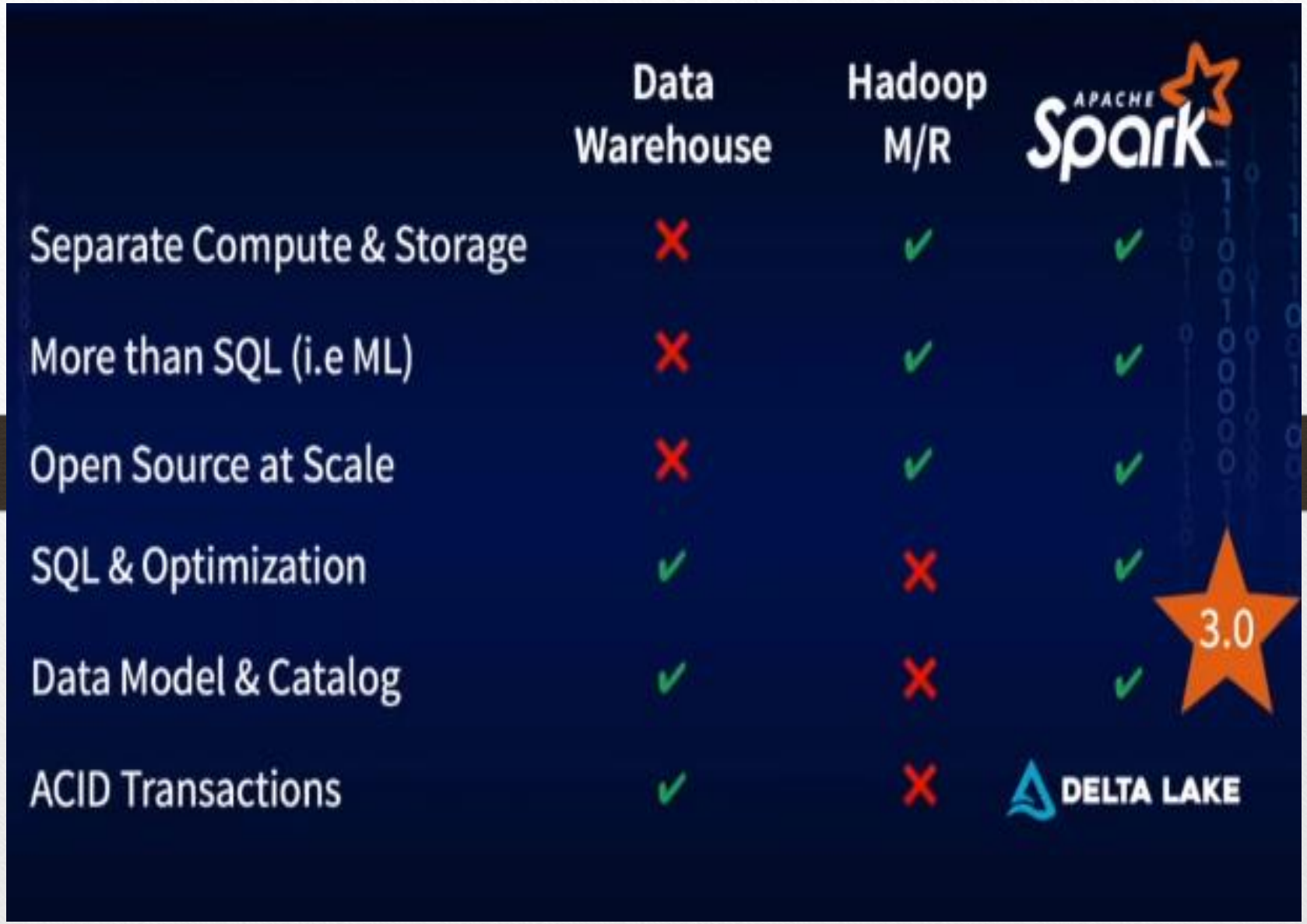
Uses Spark Streaming to provide the best-in-class movie streaming and recommendation tool to its users.

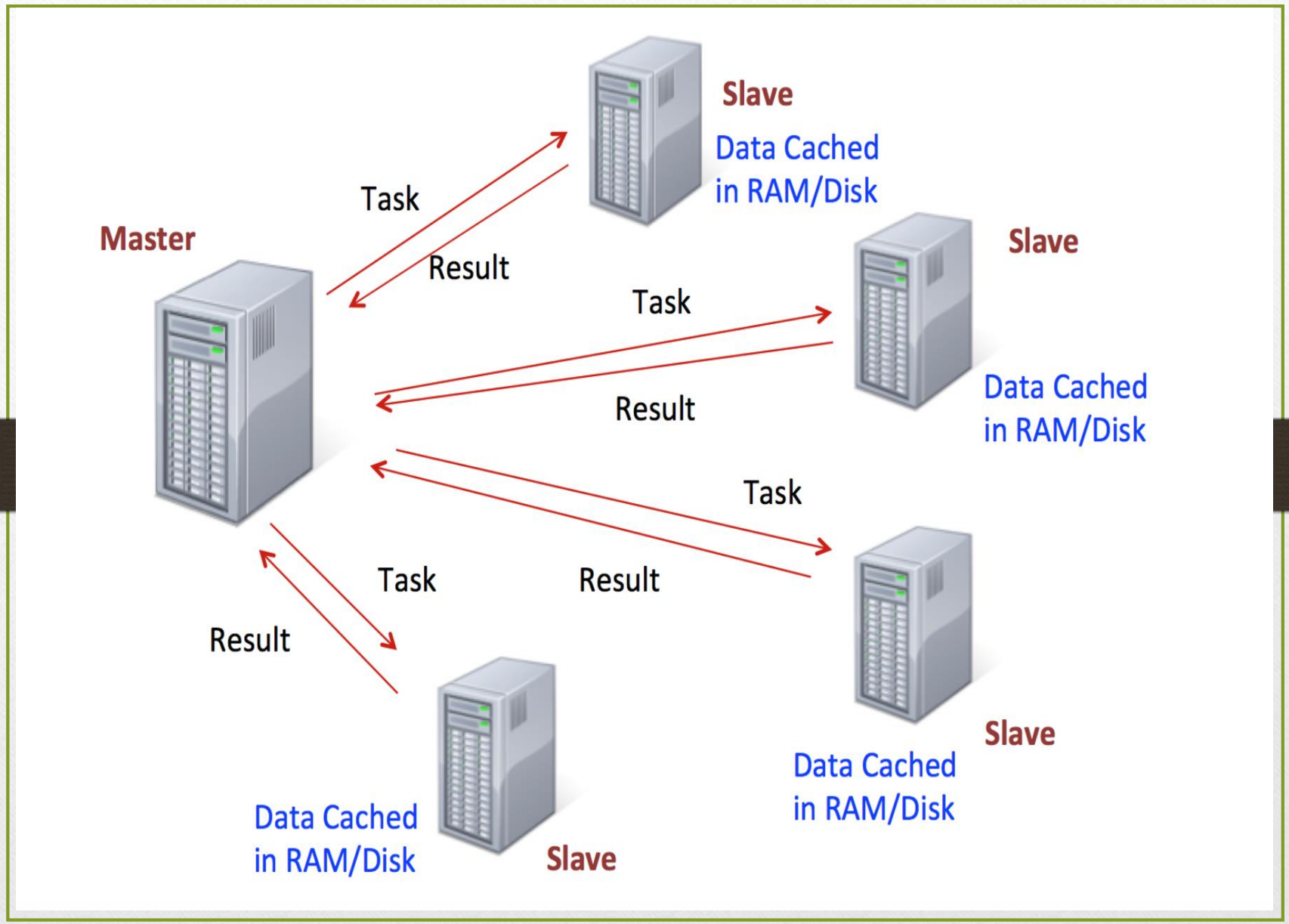
UBER

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


Slave
Data Cached
in RAM/Disk
Task
Result
Master
Slave
Task
Result
Data Cached
in RAM/Disk
Task
Result
Task
Result
Slave
Data Cached
in RAM/Disk
Data Cached
in RAM/Disk
Slave

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.

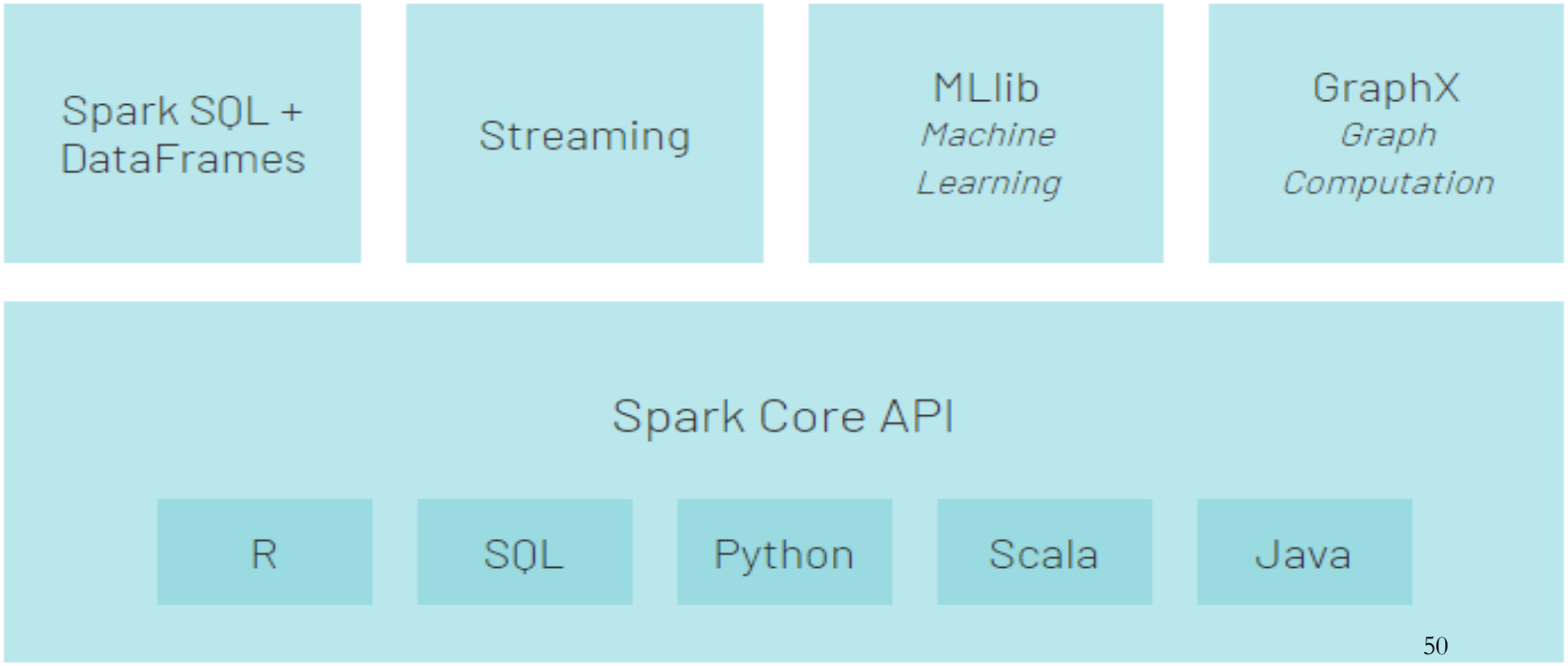
What is Apache Spark used for?

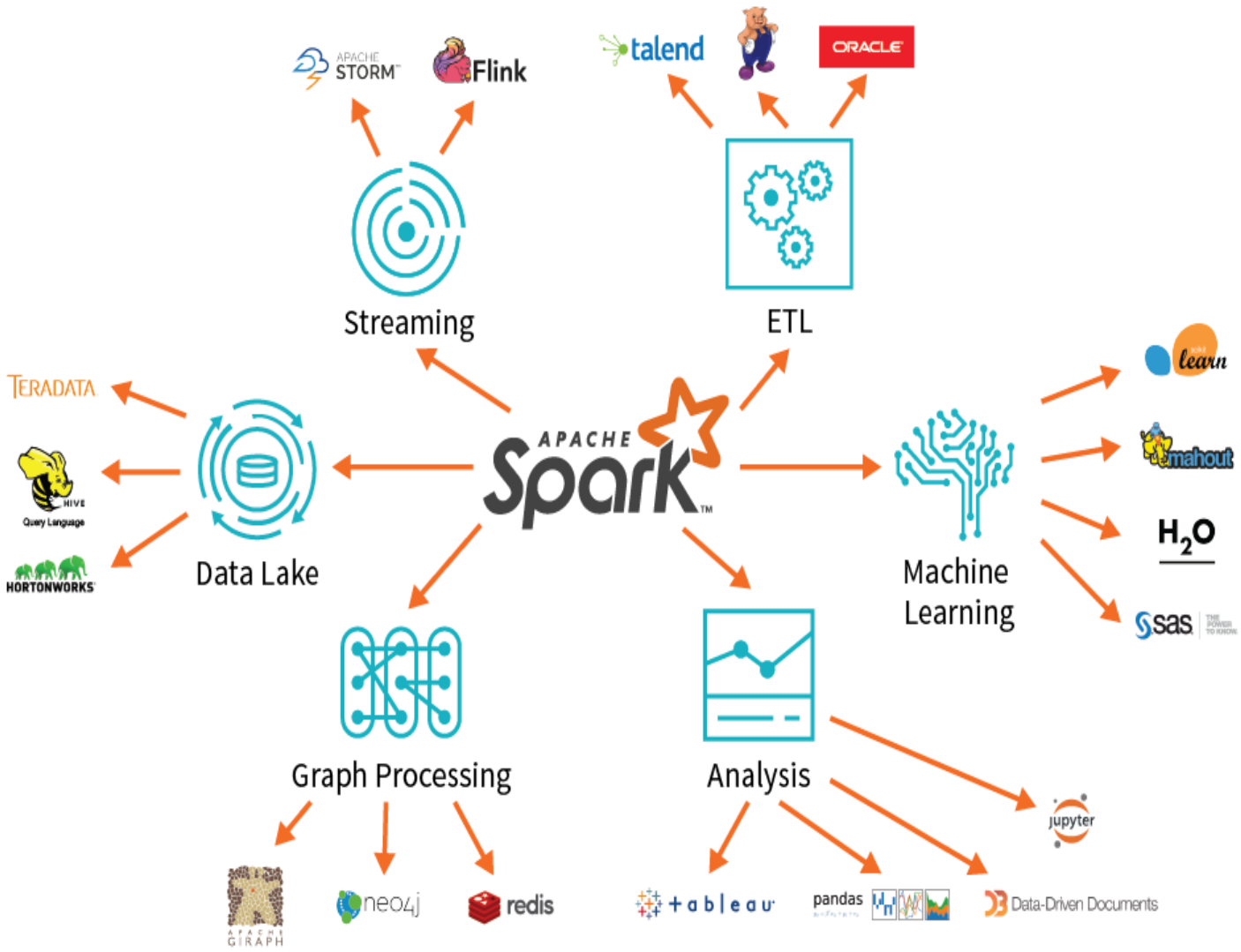
Spark is used for many types of data processing. It supports ETL, interactive queries (SQL), advanced analytics (e.g. machine learning) and structured streaming over large datasets.

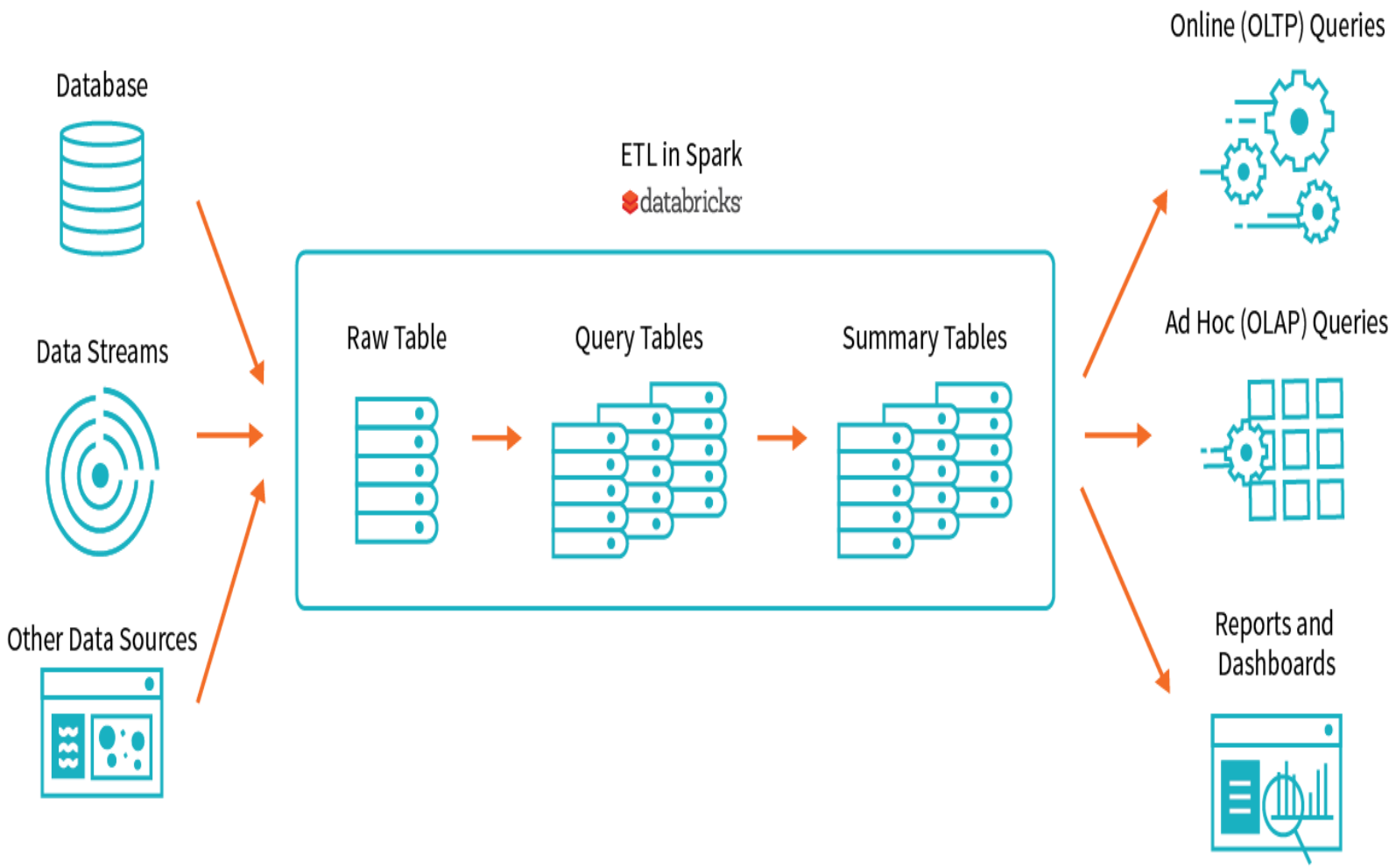
Spark integrates with many storage systems (e.g. HDFS, Cassandra, MySQL, HBase, MongoDB, S3). Spark is also pluggable, with dozens of applications, data sources, and environments

Spark Architecture & Components

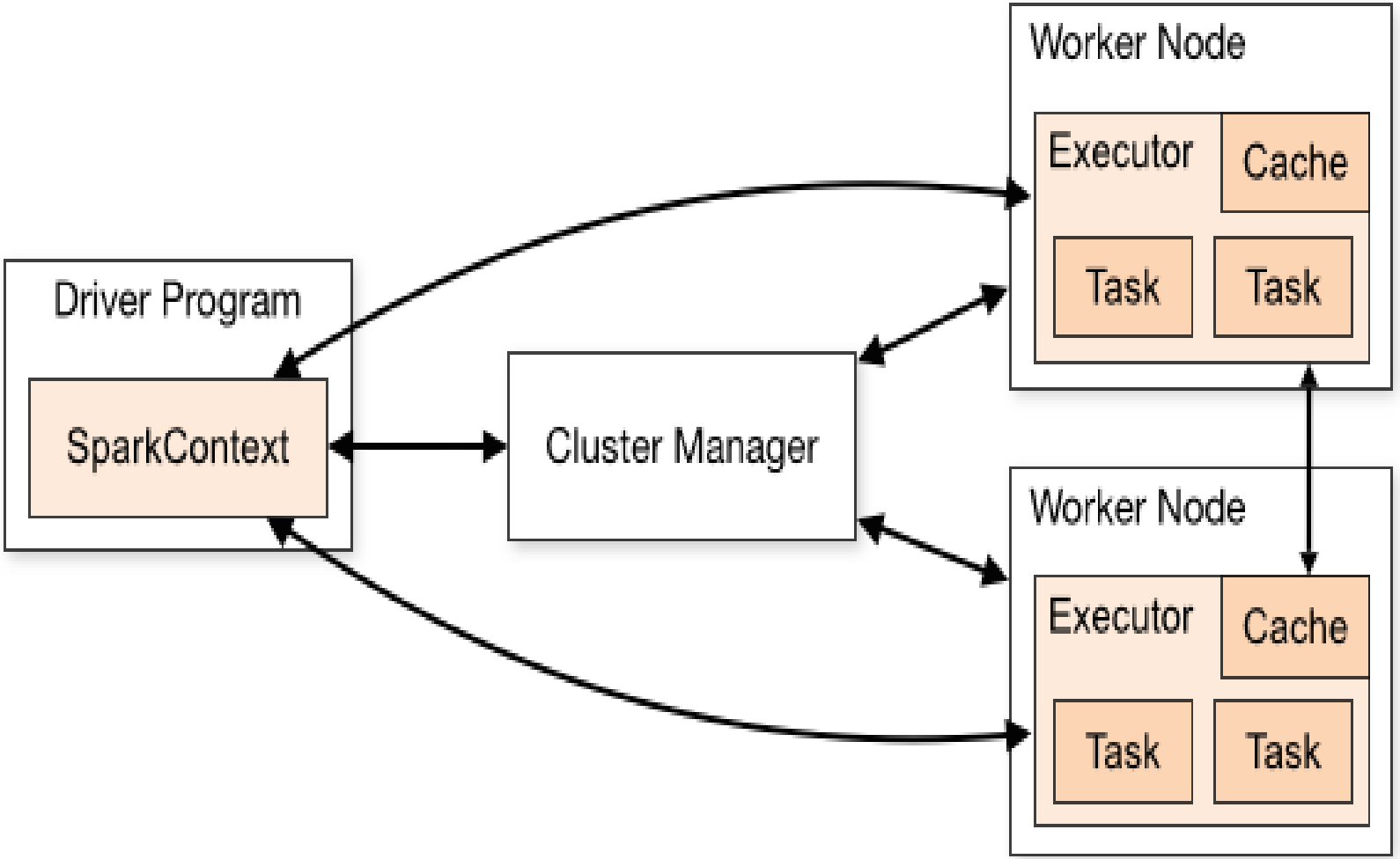
Apache Spark is a powerful open-source processing engine built around speed, ease of use, and sophisticated analytics.



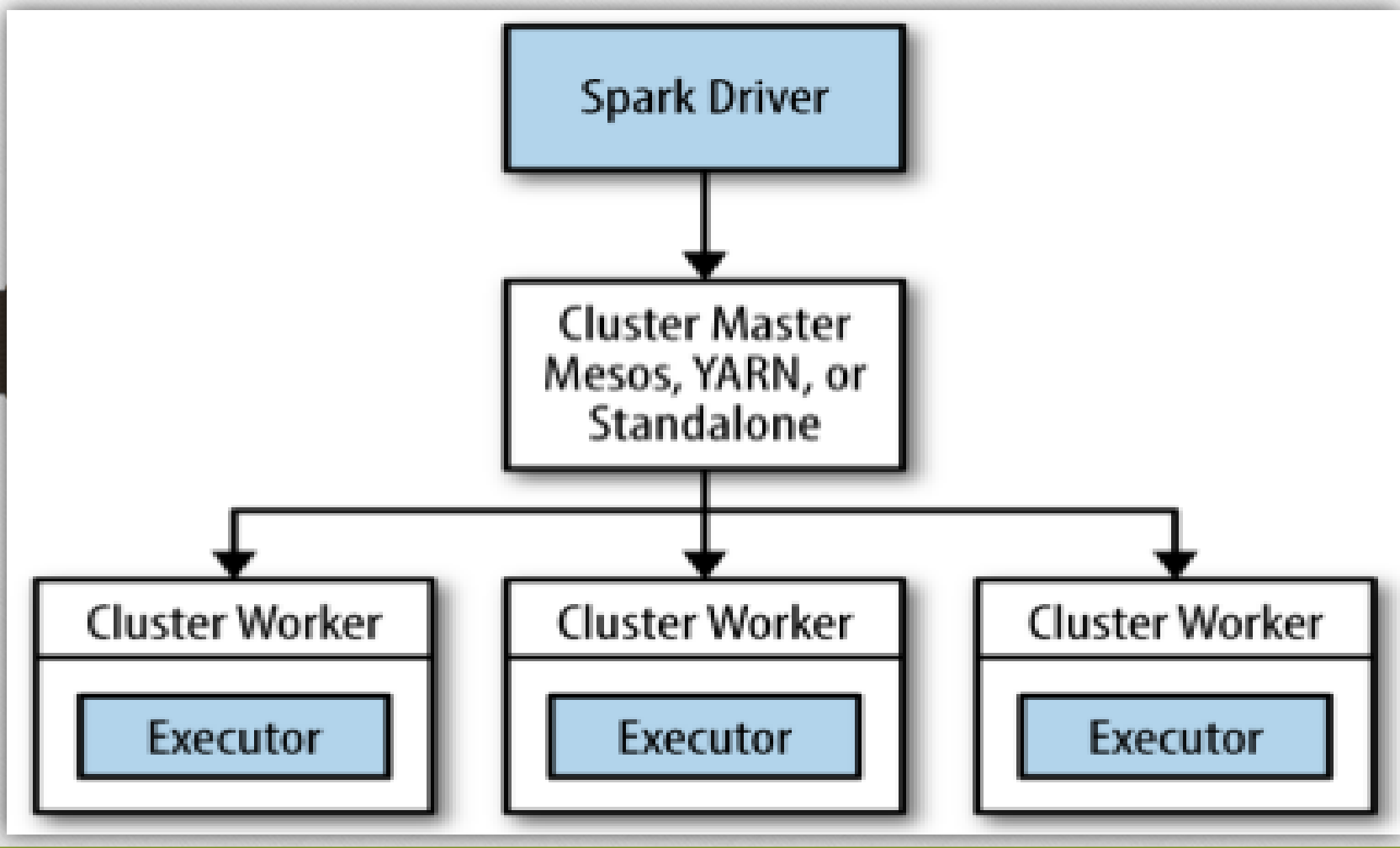

APACHE STORM
Flink
talend
ORACLE
Streaming
ETL
TERADATA
HIVE
Query Language
HORTONWORKS
Data Lake
APACHE
Spark
learn
mahout
H2O
sas
Machine Learning
Graph Processing
Analysis
jupyter
APACHE GIRAPH
neo4j
redis
+ableau
pandas
Data-Driven Documents

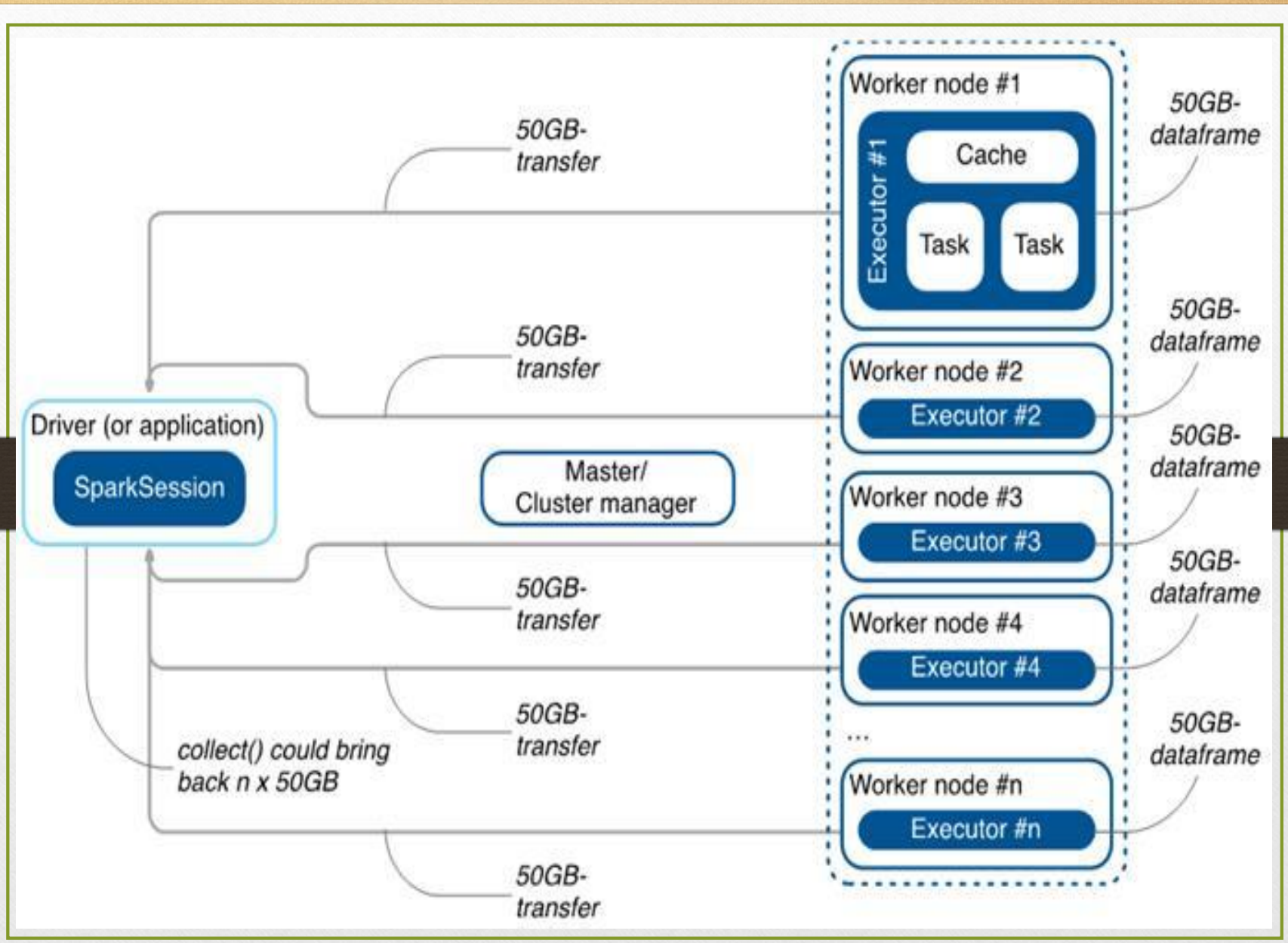

Database
Data Streams
Other Data Sources
ETL in Spark
databricks
Raw Table
Query Tables
Summary Tables
Online (OLTP) Queries
Ad Hoc (OLAP) Queries
Reports and Dashboards

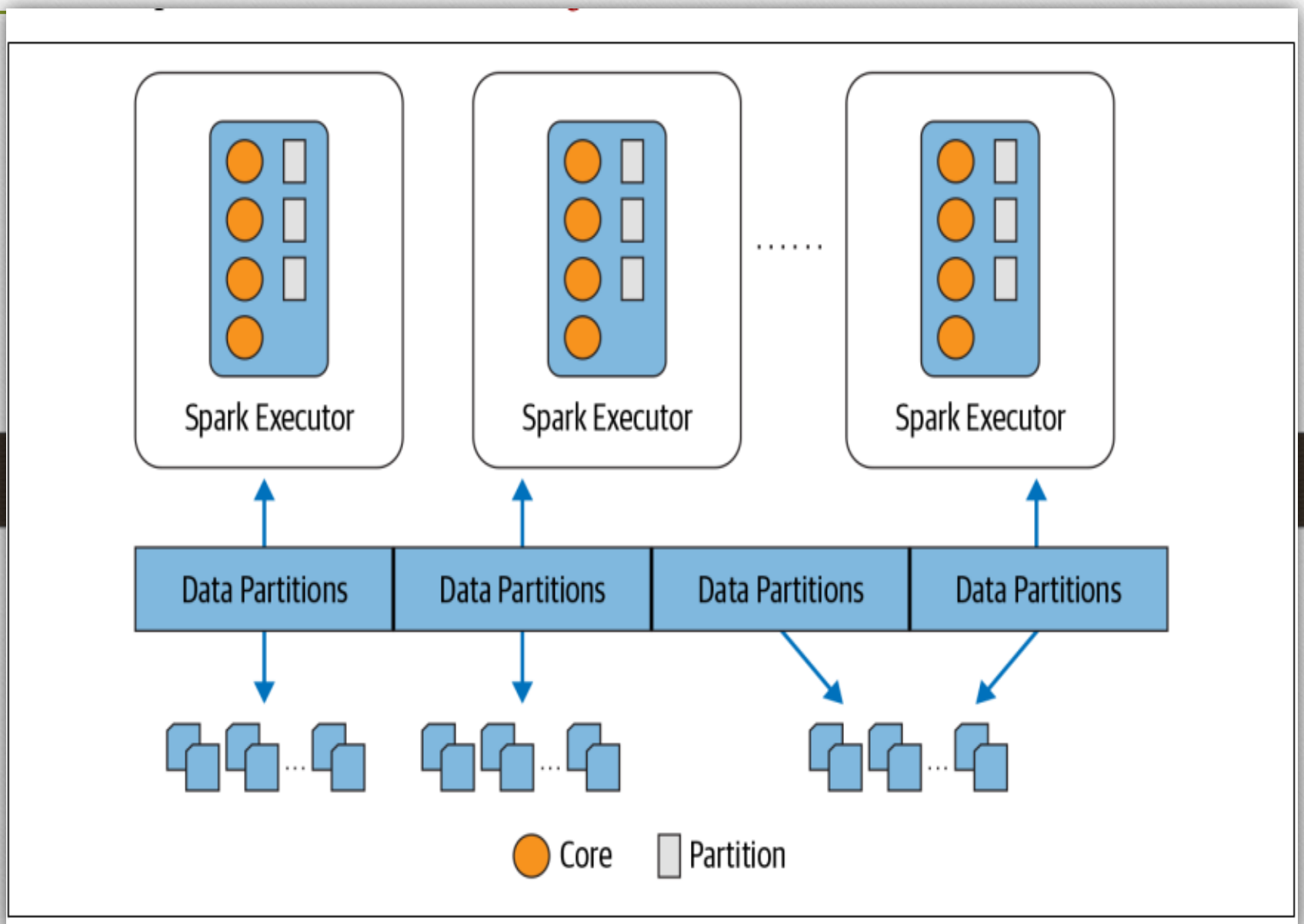
Spark Components Architecture



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.



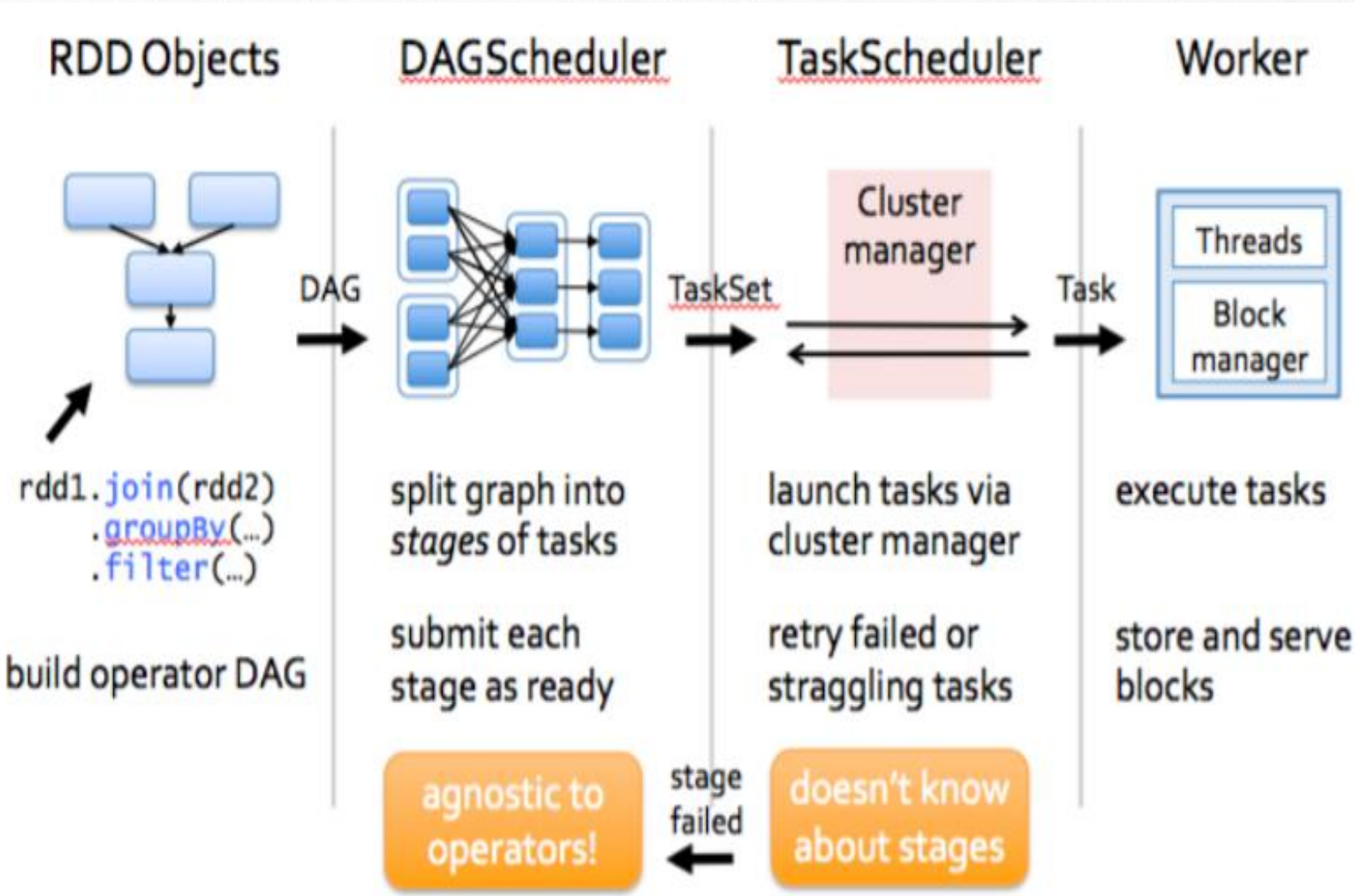

Worker node #1
Executor #1
Cache
Task
Task
50GB-dataframe
50GB-transfer
Worker node #2
Executor #2
50GB-dataframe
50GB-transfer
Driver (or application)
SparkSession
Master/
Cluster manager
Worker node #3
Executor #3
50GB-dataframe
50GB-transfer
Worker node #4
Executor #4
50GB-dataframe
50GB-transfer
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

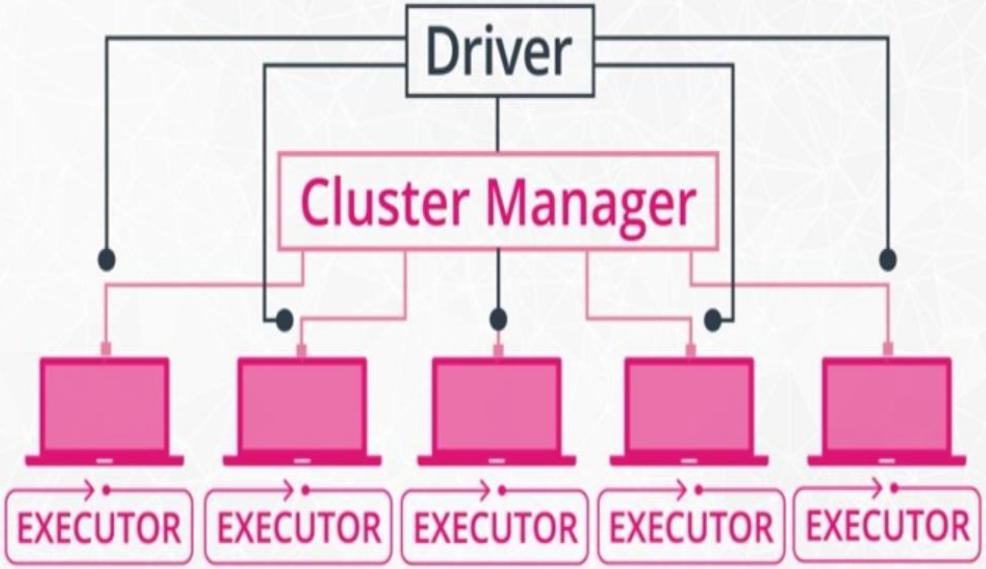


SPARK EXECUTION PROCESS STEPS

- 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



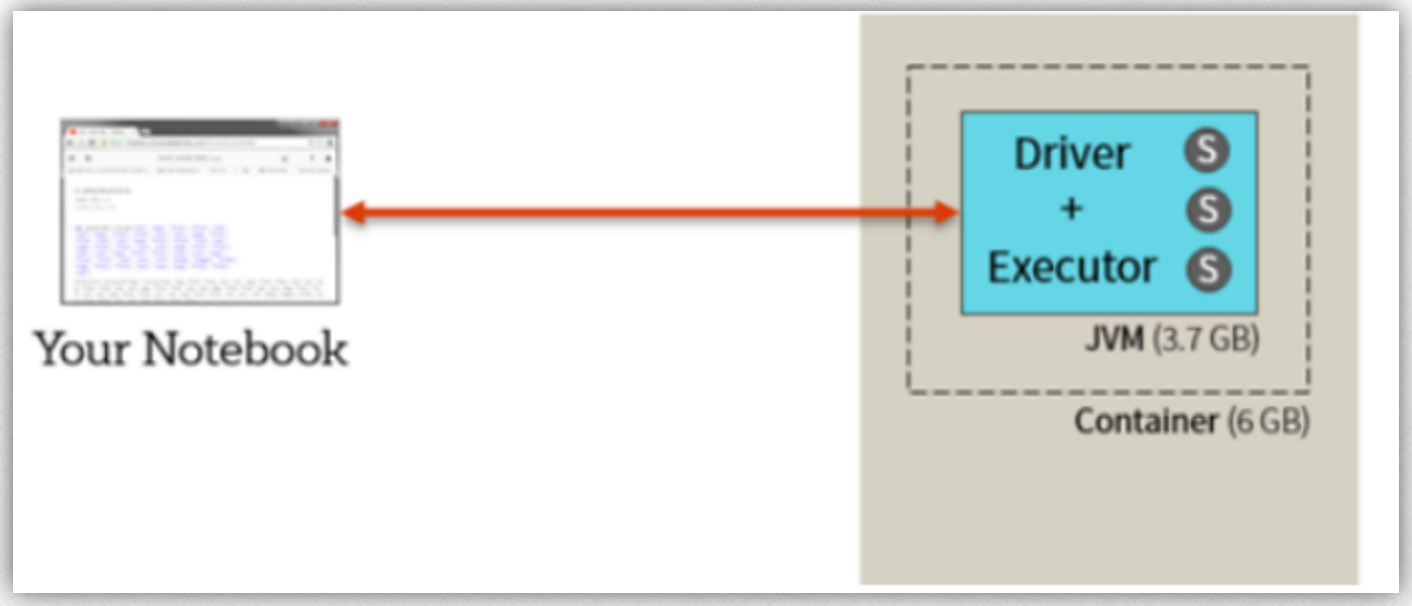
CLUSTER MODES

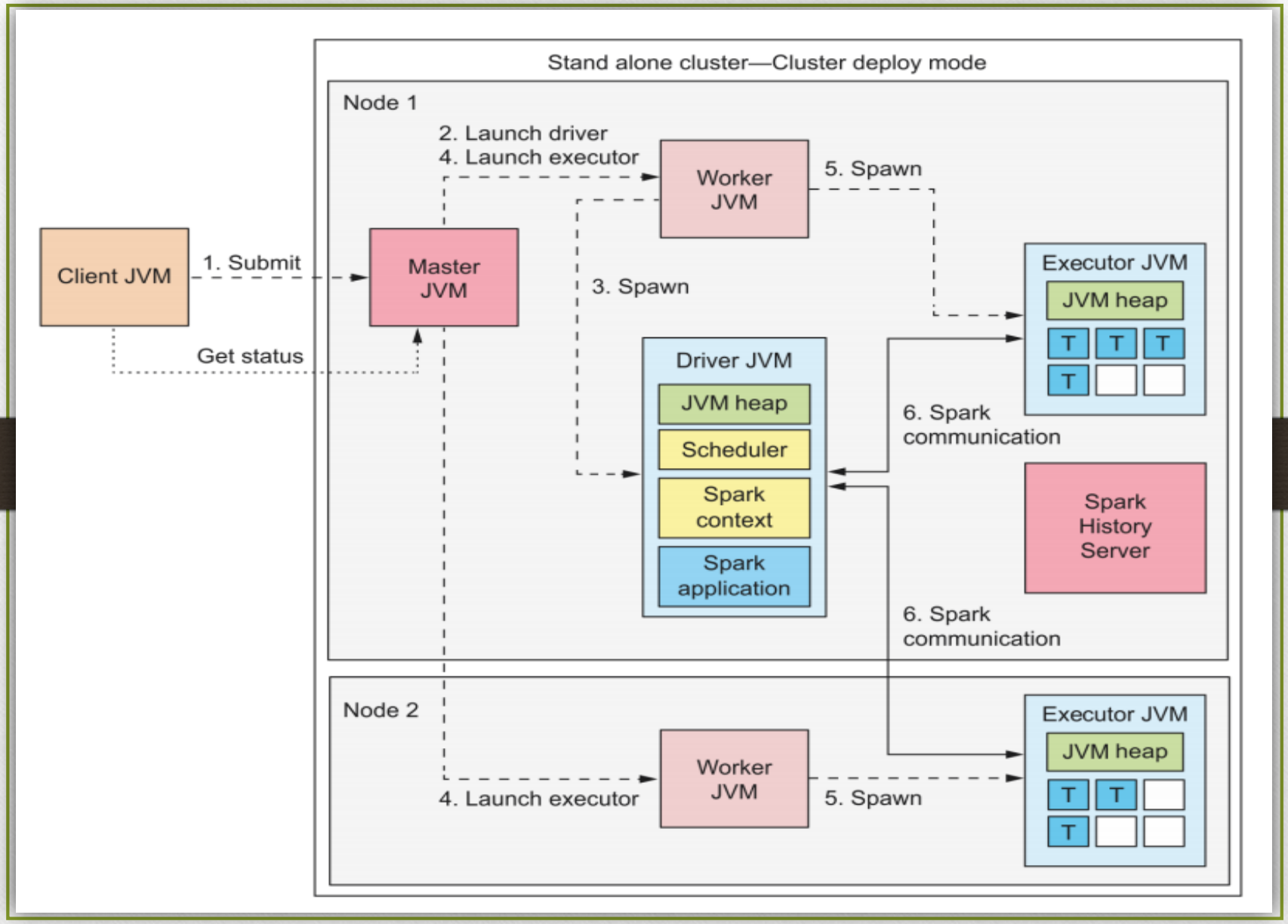
- Standalone
- YARN
- Mesos

SPARK Cluster Execution Types

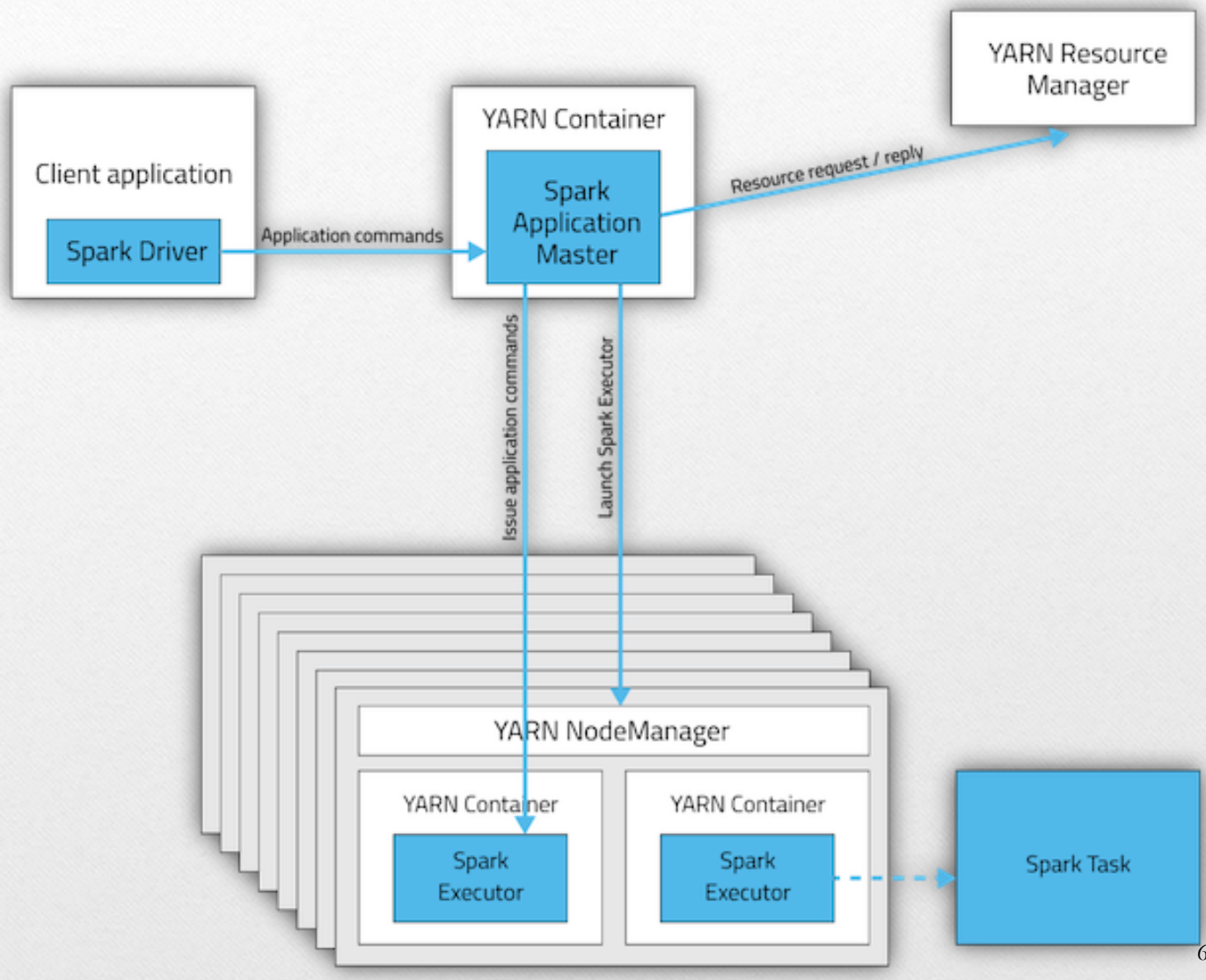
Mode	Spark driver	Spark executor	Cluster manager
Local	Runs on a single JVM, like a laptop or single node	Runs on the same JVM as the driver	Runs on the same host
Standalone	Can run on any node in the cluster	Each node in the cluster will launch its own executor JVM	Can be allocated arbitrarily to any 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

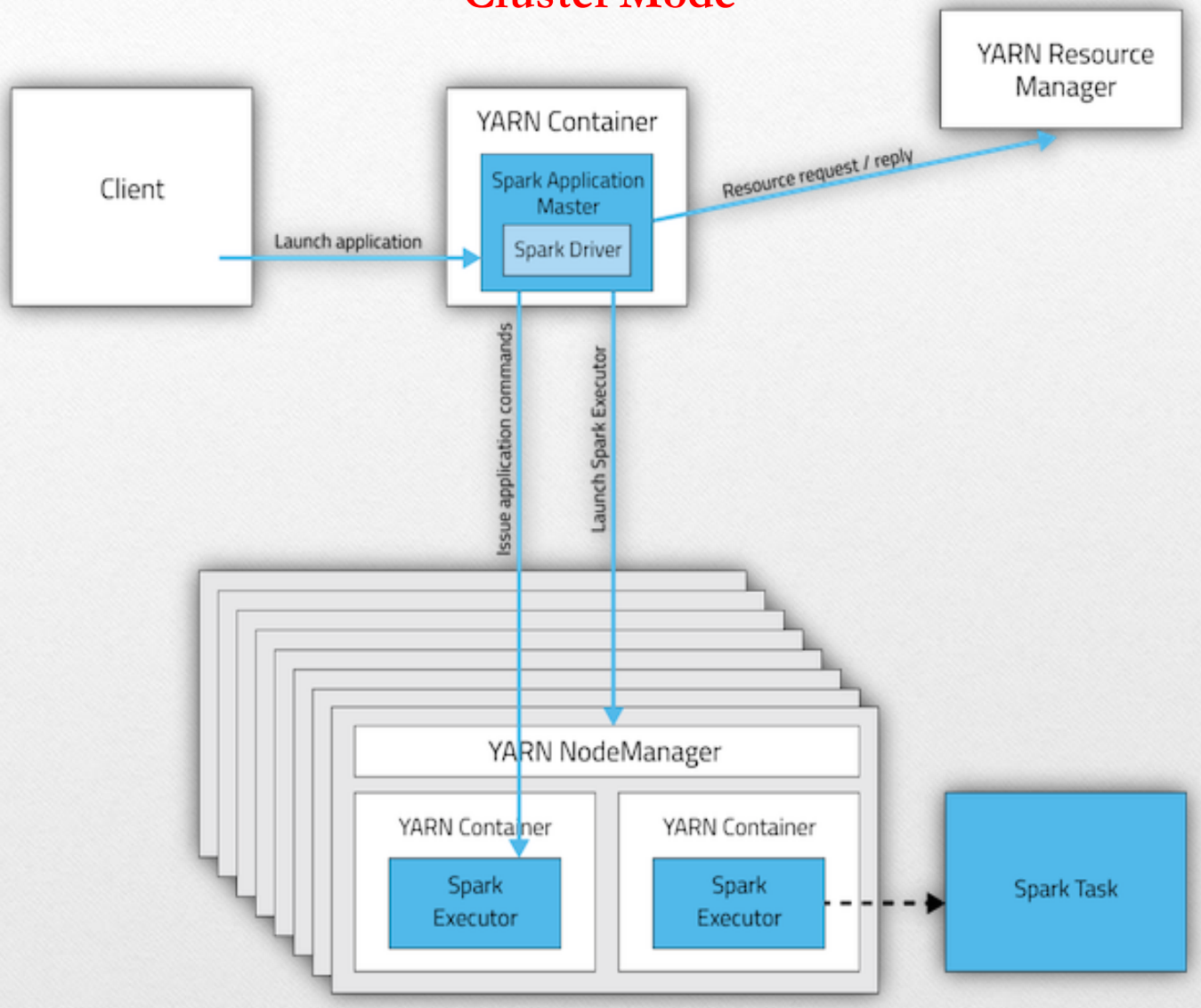



Stand alone cluster—Cluster deploy mode
Node 1
2. Launch driver
4. Launch executor
Worker JVM
5. Spawn
Client JVM
1. Submit
Master JVM
3. Spawn
Executor JVM
JVM heap
T T T
T
Get status
Driver JVM
JVM heap
Scheduler
Spark context
Spark application
6. Spark communication
Spark History Server
6. Spark communication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



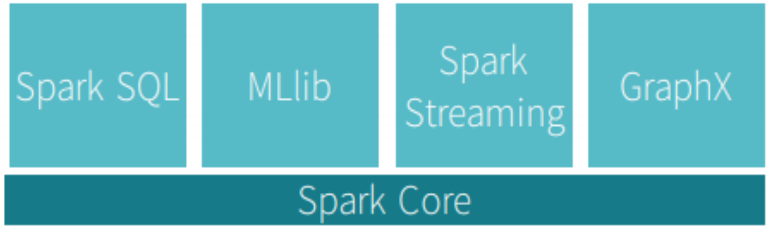
	YARN Cluster	YARN Client	Spark Standalone
Driver runs in:	Application Master	Client	Client
Who requests resources?	Application Master	Application Master	Client
Who starts executor processes?	YARN NodeManager	YARN NodeManager	Spark Slave
Persistent services	YARN ResourceManager and NodeManagers	YARN ResourceManager 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

Creating RDD And Working on RDD

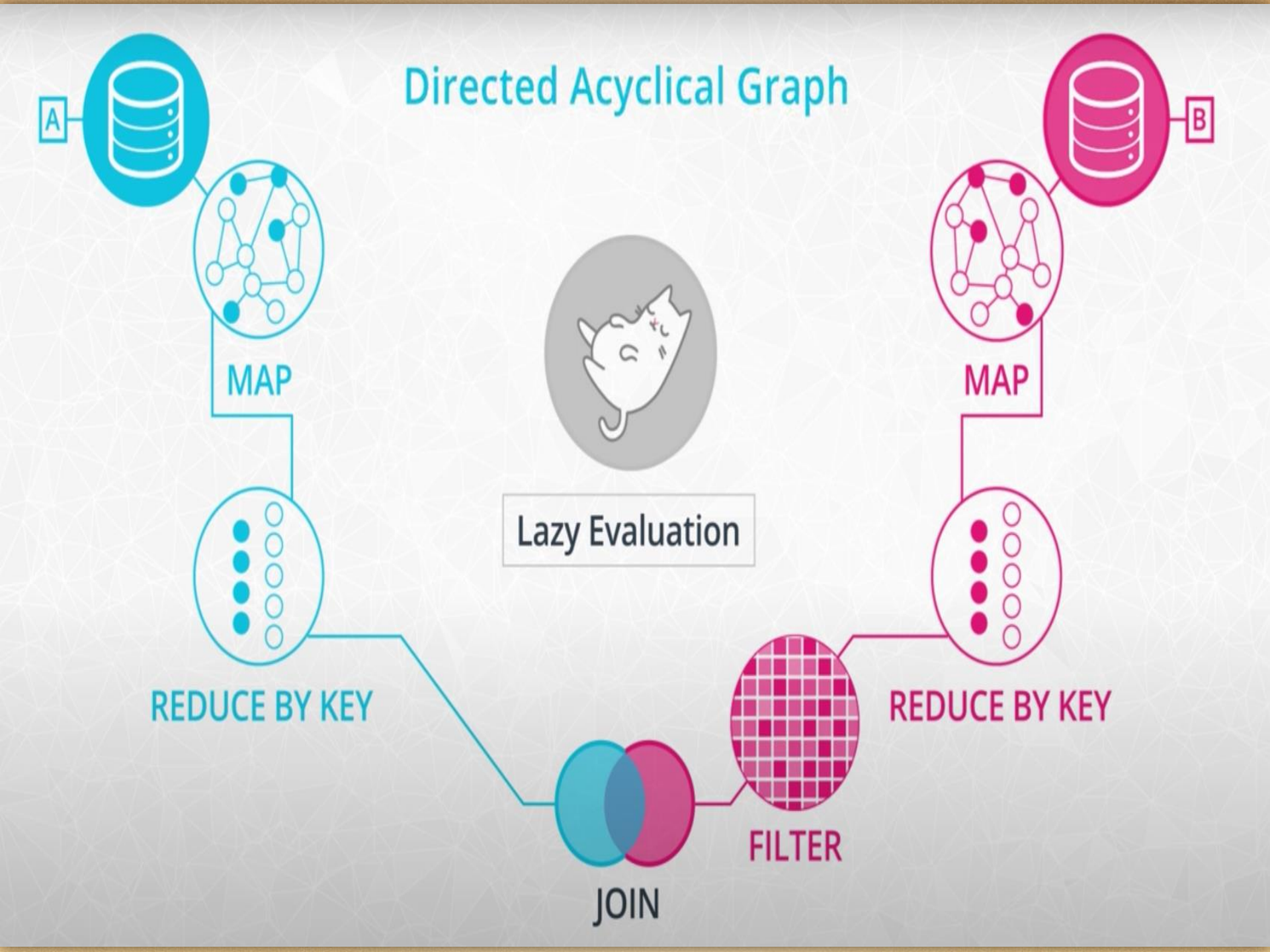
TRANSFORMATIONS

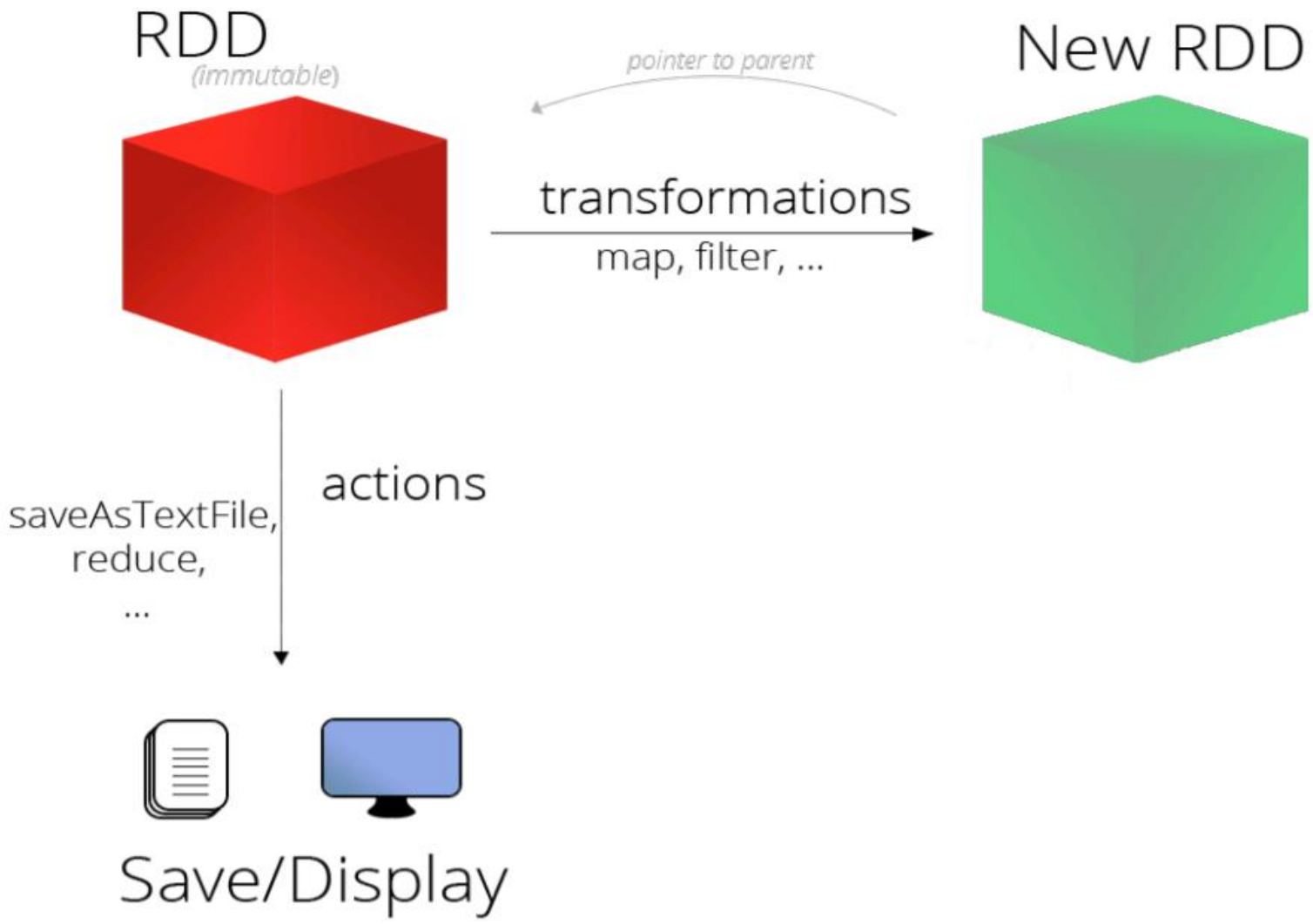
Spark Operations = +

ACTIONS

Operations on RDDs

- Transformations: lazy execution
  - Map, filter, intersection, groupByKey, zipWithIndex …
- Actions: trigger execution of transformations
  - Collect, count, reduce, saveAsTextFile …
- Caching
  - Avoid multiple executions of transformations when re-using RDD


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

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

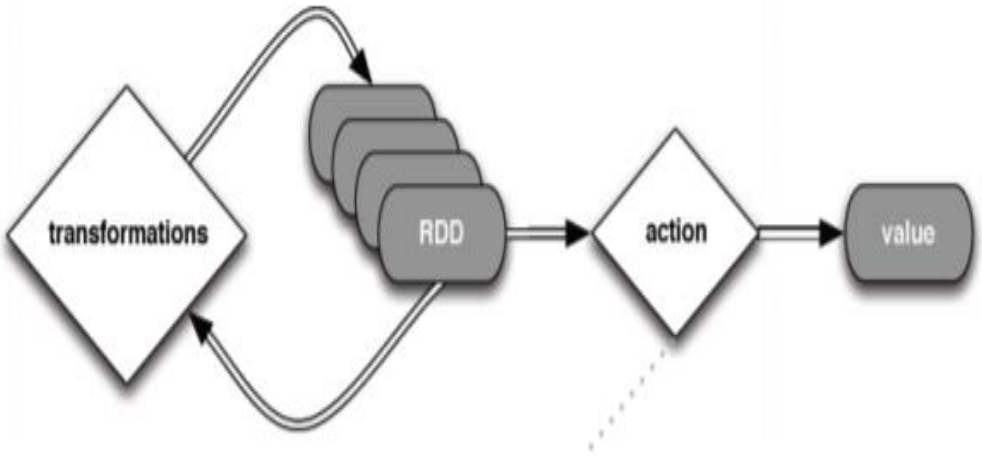
```
# Read a local txt file in Python
linesRDD = sc.textFile("/path/to/README.md")
```

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

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.



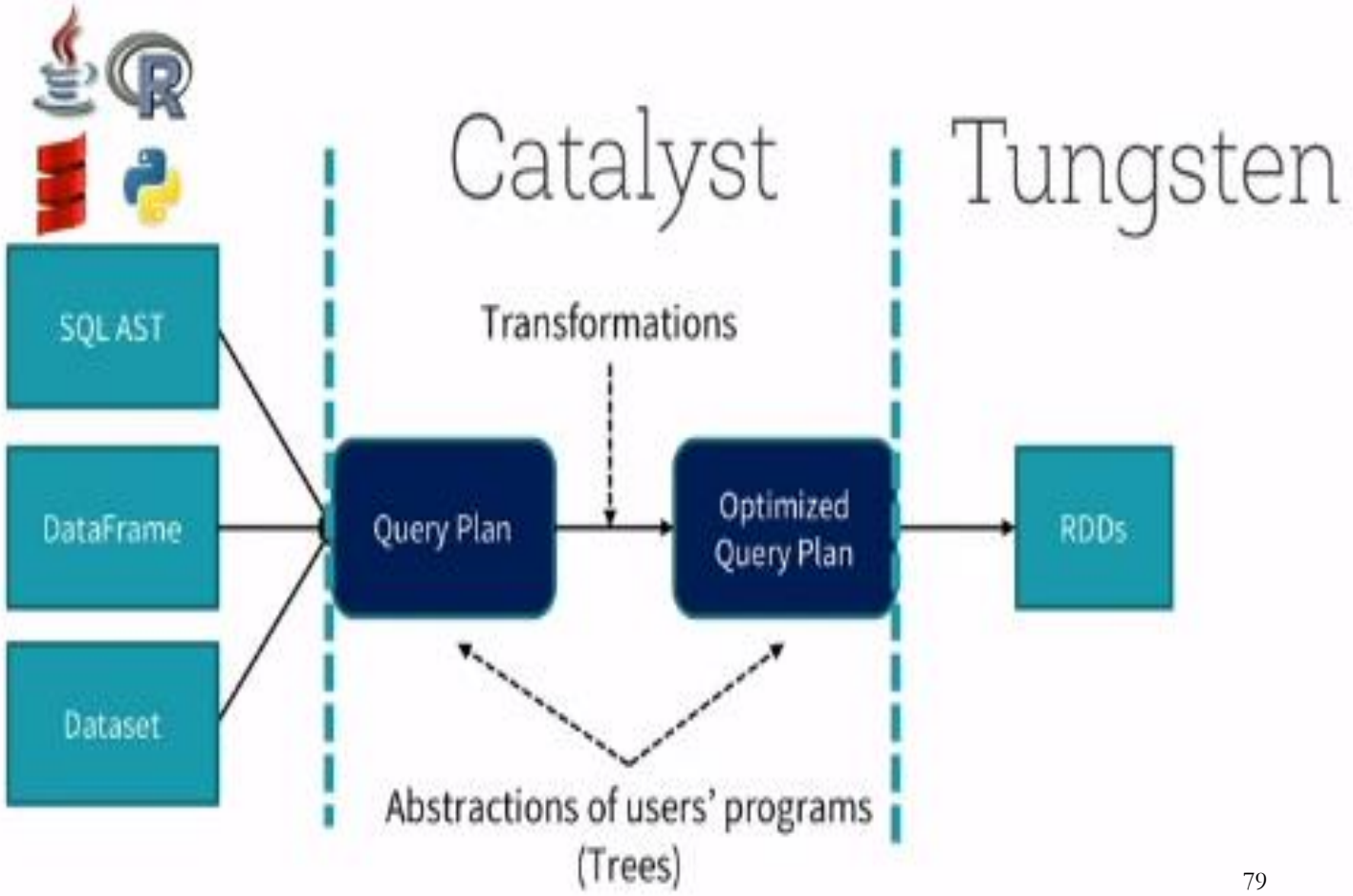
Transformations

- 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.

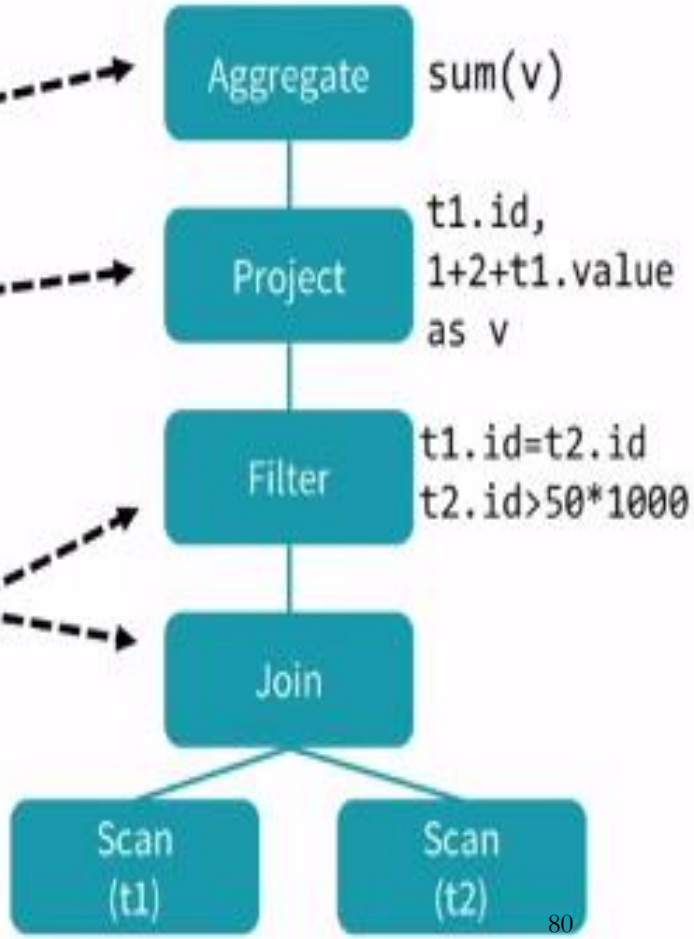
Spark SQL Overview



Trees: Abstractions of Users' Programs

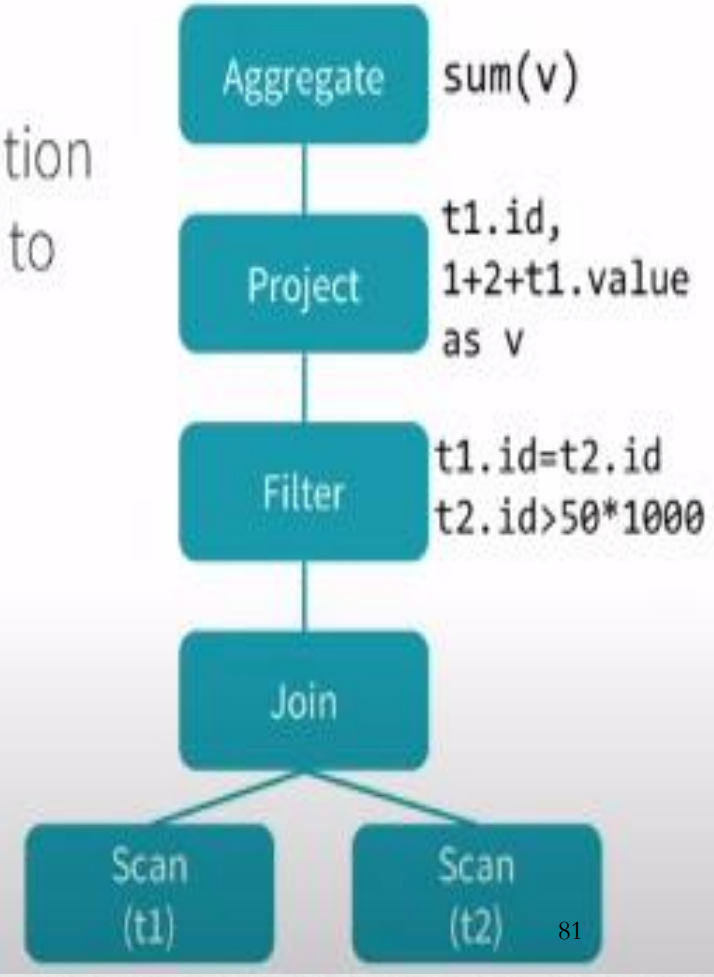
Query Plan

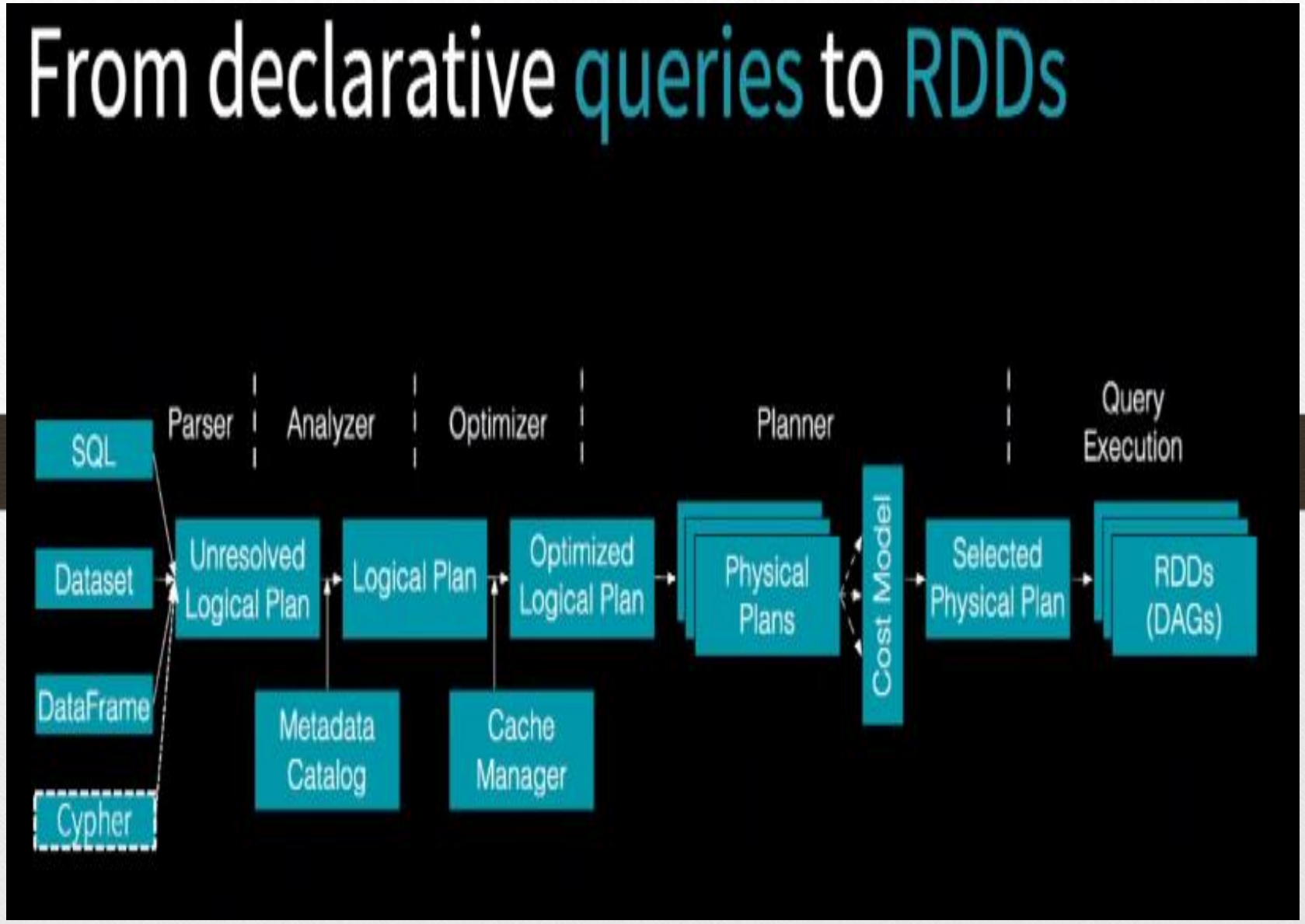
```
SELECT sum(v)
FROM (
  SELECT
    t1.id,
    1 + 2 + t1.value AS v
  FROM t1 JOIN t2
  WHERE
    t1.id = t2.id AND
    t2.id > 50 * 1000) tmp
```



Logical Plan

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

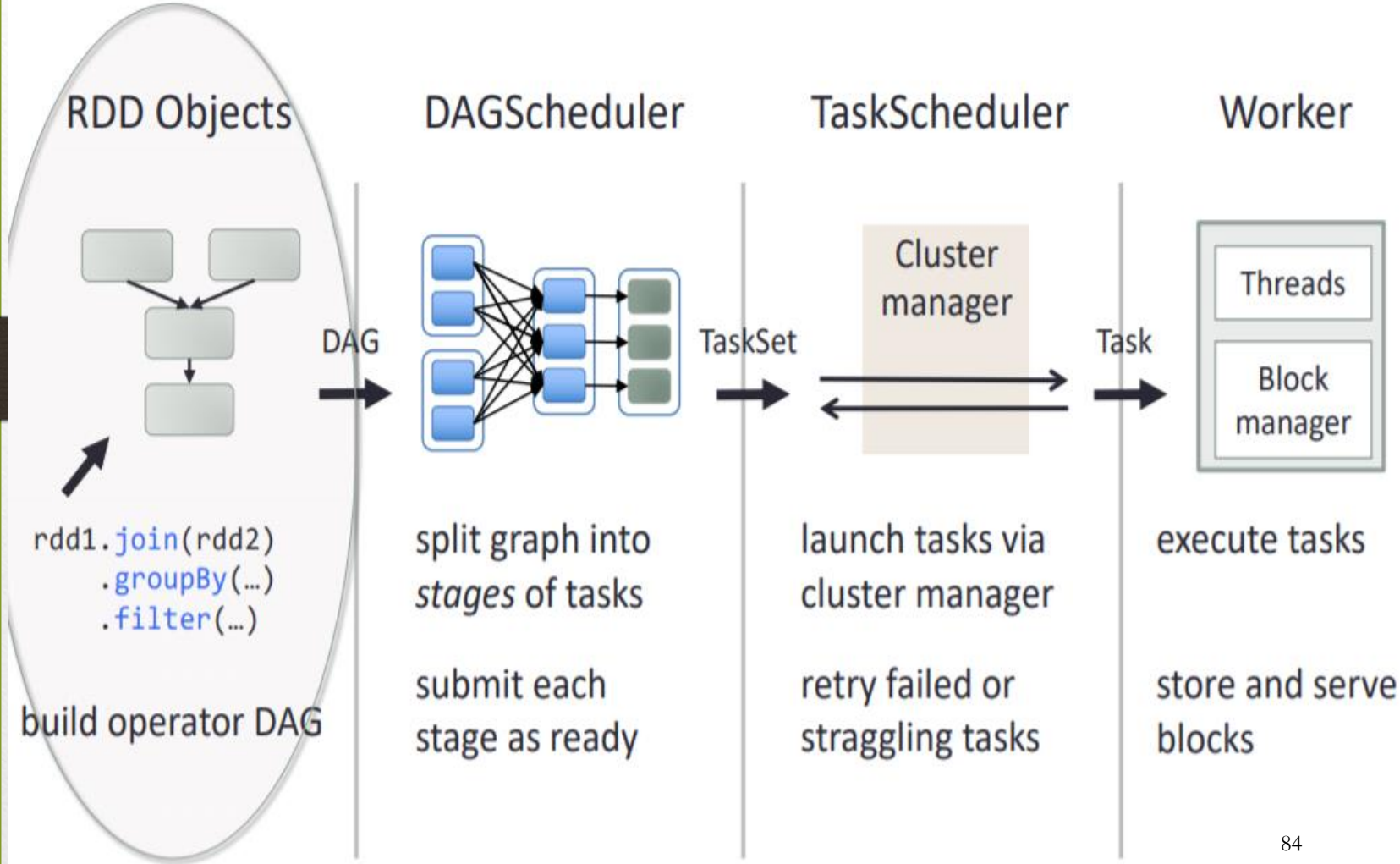



From declarative queries to RDDs
SQL
Dataset
DataFrame
Cypher
Parser
Analyzer
Optimizer
Planner
Query Execution
Unresolved Logical Plan
Logical Plan
Optimized Logical Plan
Physical Plans
Cost Model
Selected Physical Plan
RDDs (DAGs)
Metadata Catalog
Cache Manager

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.

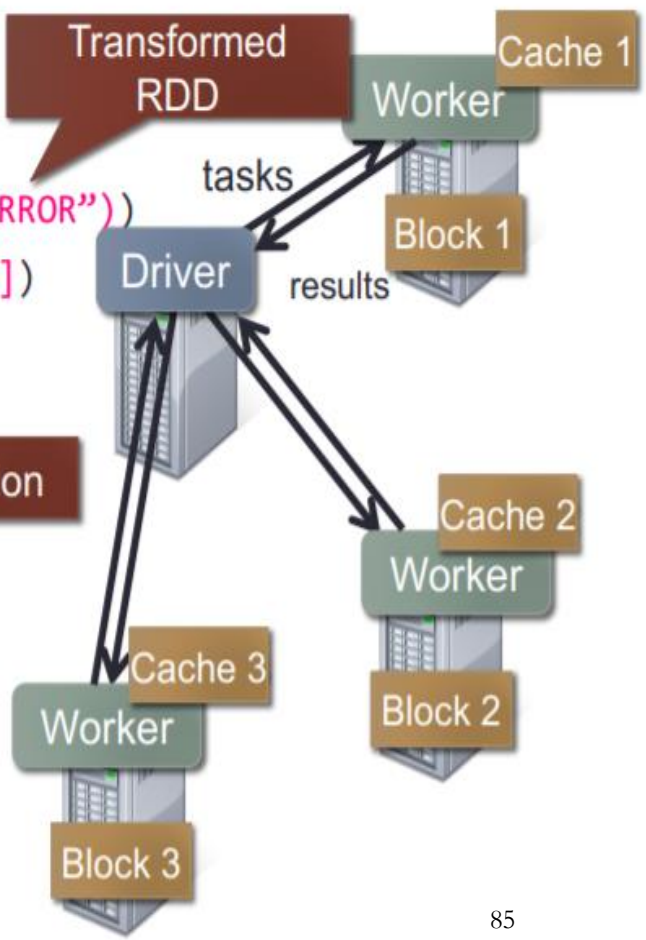
Job scheduling



Example: Mining Console Logs

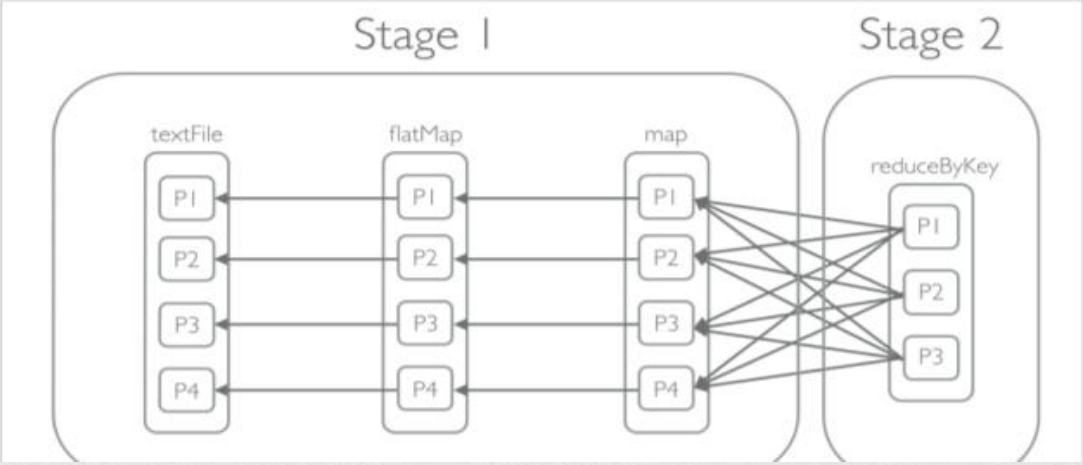
Load error messages from a log into memory, then interactively search for patterns

```
lines = spark.textFile(“hdfs://...”)
errors = lines.filter(lambda s: s.startswith(“ERROR”))
messages = errors.map(lambda s: s.split(‘\t’)[2])
messages.cache()


messages.filter(lambda s: “foo” in s).count()
messages.filter(lambda s: “bar” in s).count()
. . .
```



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



RDD Fault Tolerance

- Hadoop conservative/pessimistic approach

  →Go to disk / stable storage (HDFS)

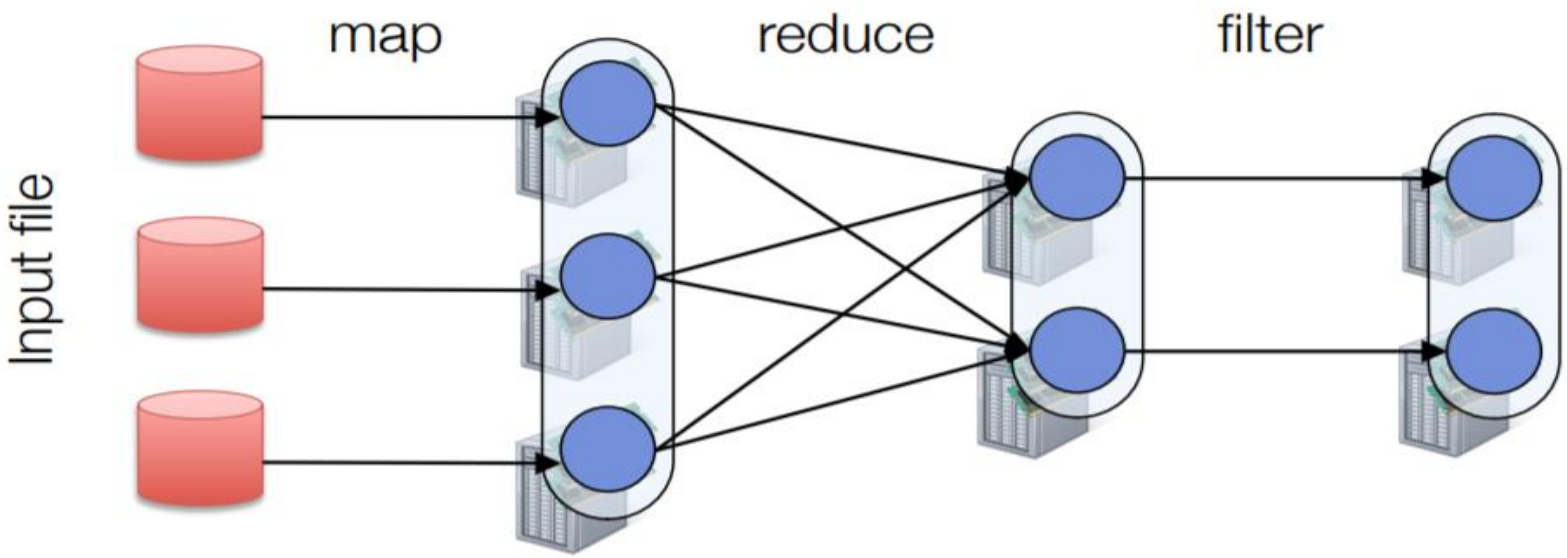
- Spark optimistic

  →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

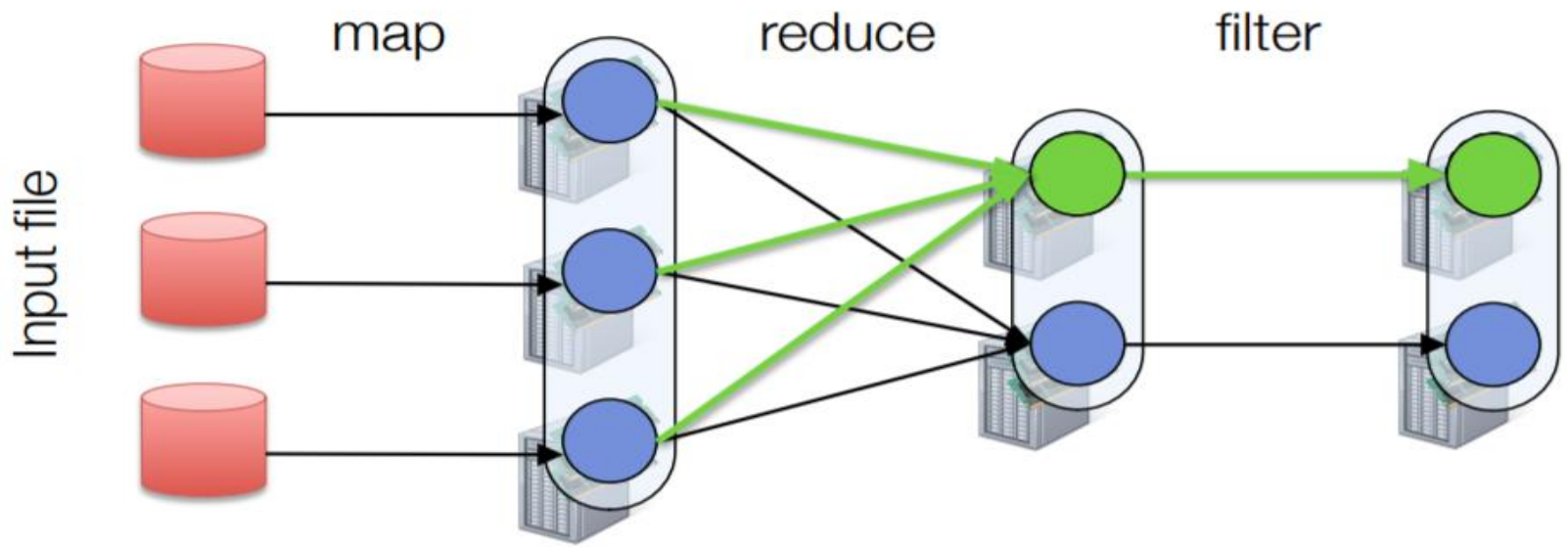
```
file.map(lambda rec: (rec.type, 1))
    .reduceByKey(lambda x, y: x + y)
    .filter(lambda (type, count): count > 10)
```



Fault Tolerance

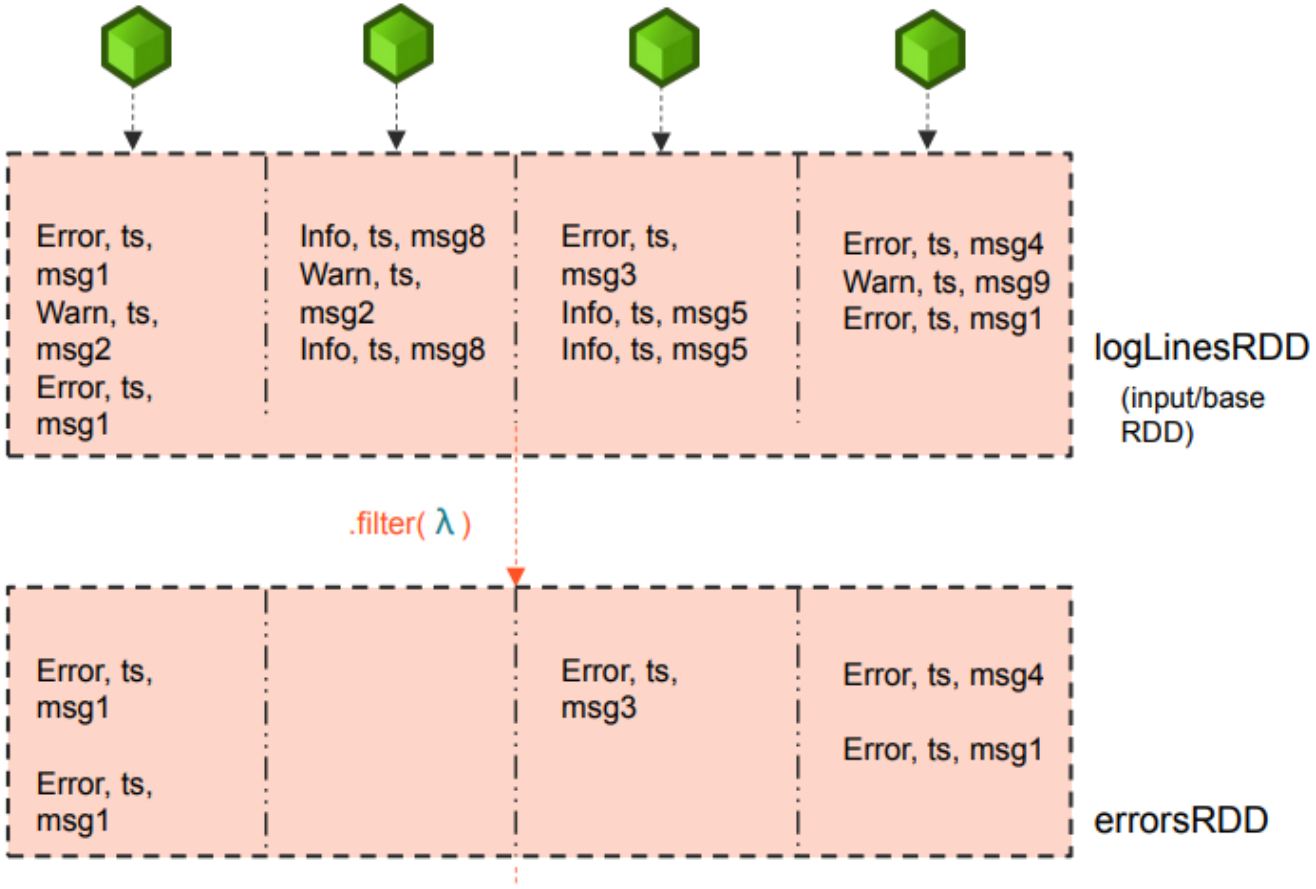
RDDs track *lineage* info to rebuild lost data

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

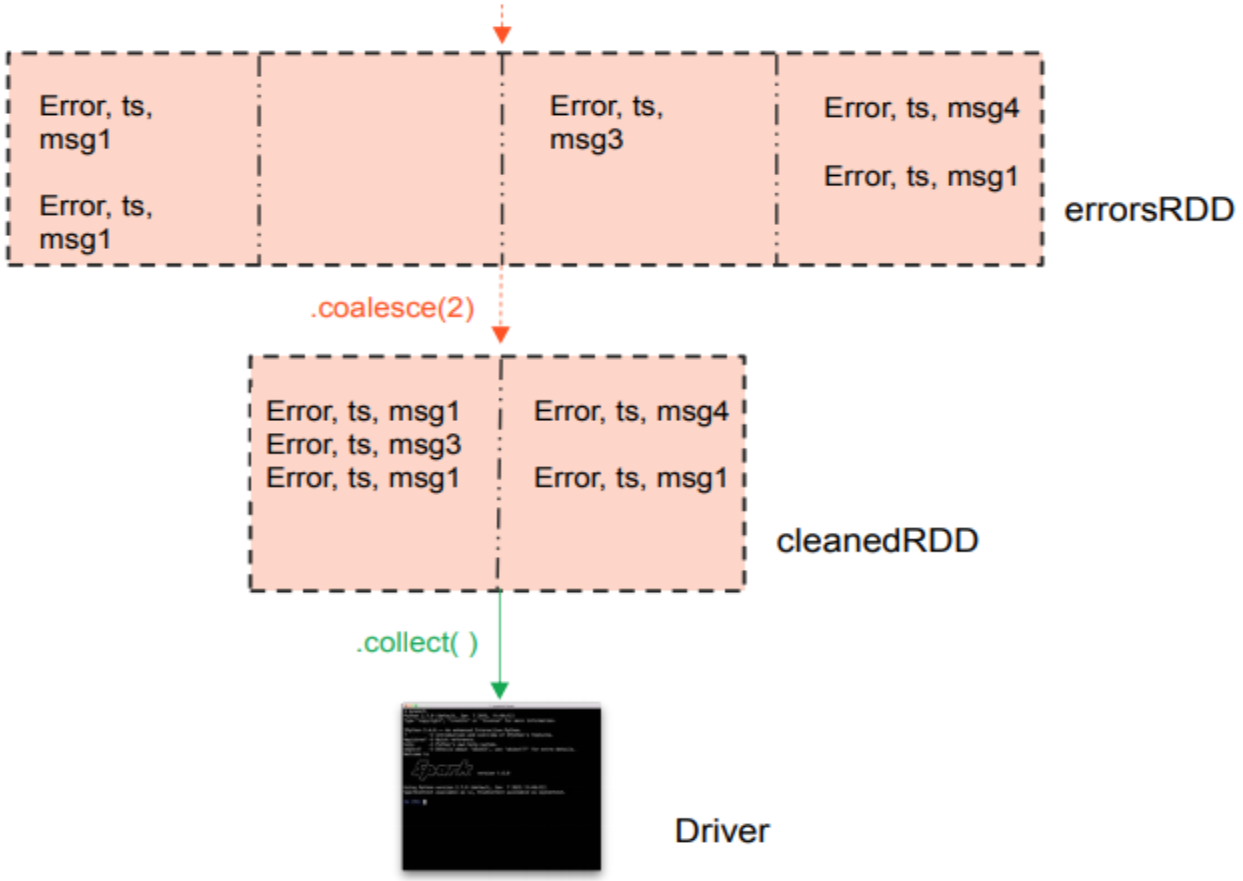


RDD Example

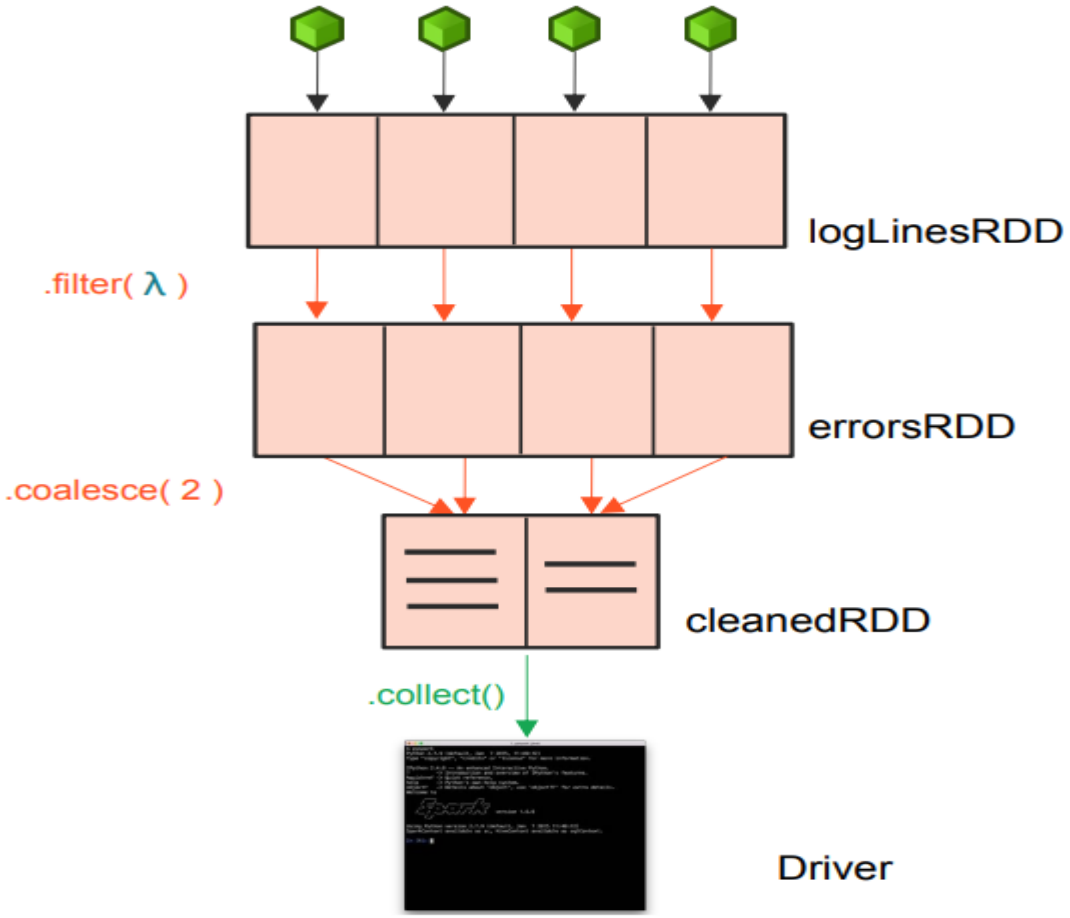

hadoop
HDFS


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

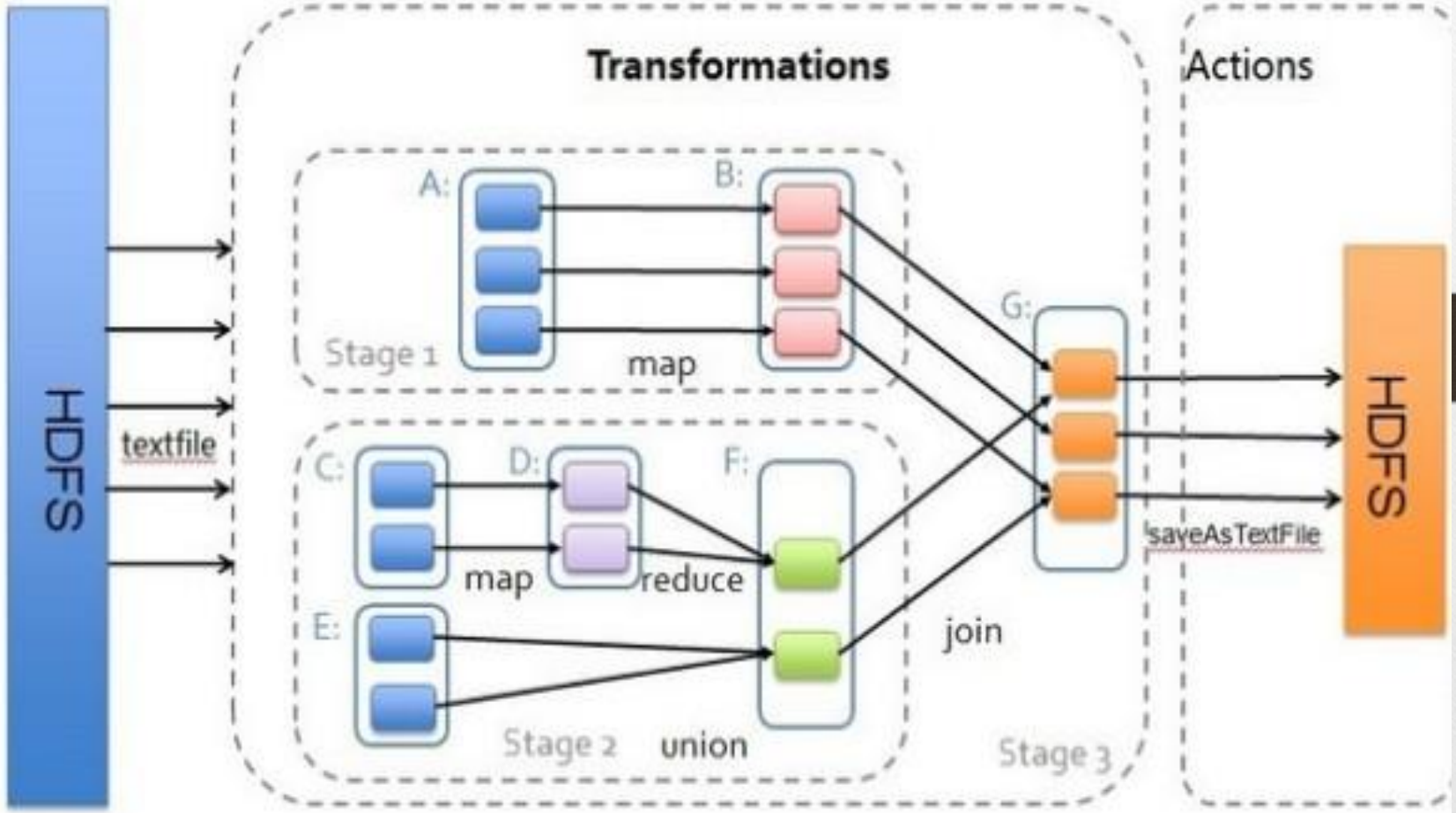
RDD Example


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

= easy = medium

Essential Core & Intermediate Spark Operations

TRANSFORMATIONS

General

- map
- filter
- flatMap
- mapPartitions
- mapPartitionsWithIndex
- groupBy
- sortBy

Math / Statistical

- sample
- randomSplit

Set Theory / Relational

- union
- intersection
- subtract
- distinct
- cartesian
- zip

Data Structure / I/O

- keyBy
- zipWithIndex
- zipWithUniqueID
- zipPartitions
- coalesce
- repartition
- repartitionAndSortWithinPartitions
- pipe

ACTIONS

- reduce
- collect
- aggregate
- fold
- first
- take
- forEach
- top
- treeAggregate
- treeReduce
- forEachPartition
- collectAsMap

- count
- takeSample
- max
- min
- sum
- histogram
- mean
- variance
- stdev
- sampleVariance
- countApprox
- countApproxDistinct

- takeOrdered

- saveAsTextFile
- saveAsSequenceFile
- saveAsObjectFile
- saveAsHadoopDataset
- saveAsHadoopFile
- saveAsNewAPIHadoopDataset
- saveAsNewAPIHadoopFile

 = easy  = medium

Essential Core & Intermediate PairRDD Operations

General	Math / Statistical	Set Theory / Relational	Data Structure
• flatMapValues • groupByKey • reduceByKey • reduceByKeyLocally • foldByKey • aggregateByKey • sortByKey • combineByKey	• sampleByKey	• cogroup (=groupWith) • join • subtractByKey • fullOuterJoin • leftOuterJoin • rightOuterJoin	• partitionBy

• keys • values	• countByKey • countByValue • countByValueApprox • countApproxDistinctByKey • countApproxDistinctByKey • countByKeyApprox • sampleByKeyExact

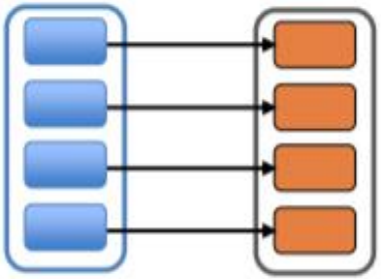
vs

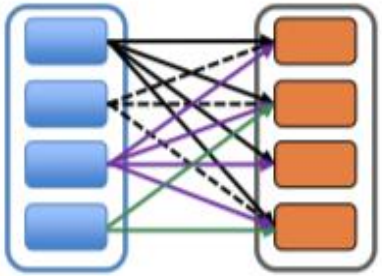
narrow

wide

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

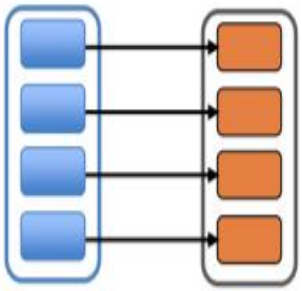
multiple child RDD partitions may depend on a single parent RDD partition



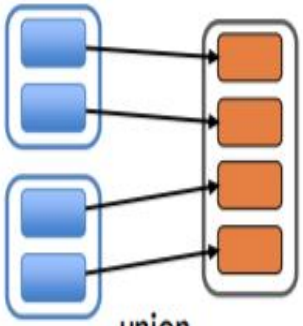


narrow

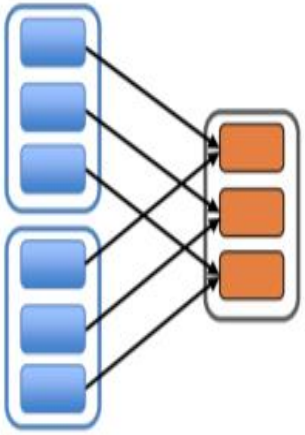
each partition of the parent RDD is used by at most one partition of the child RDD



map, filter



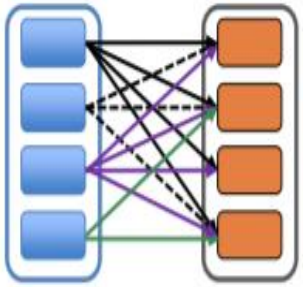
union



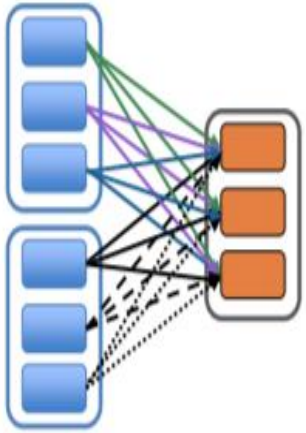
join w/ inputs co-partitioned

wide

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



groupByKey



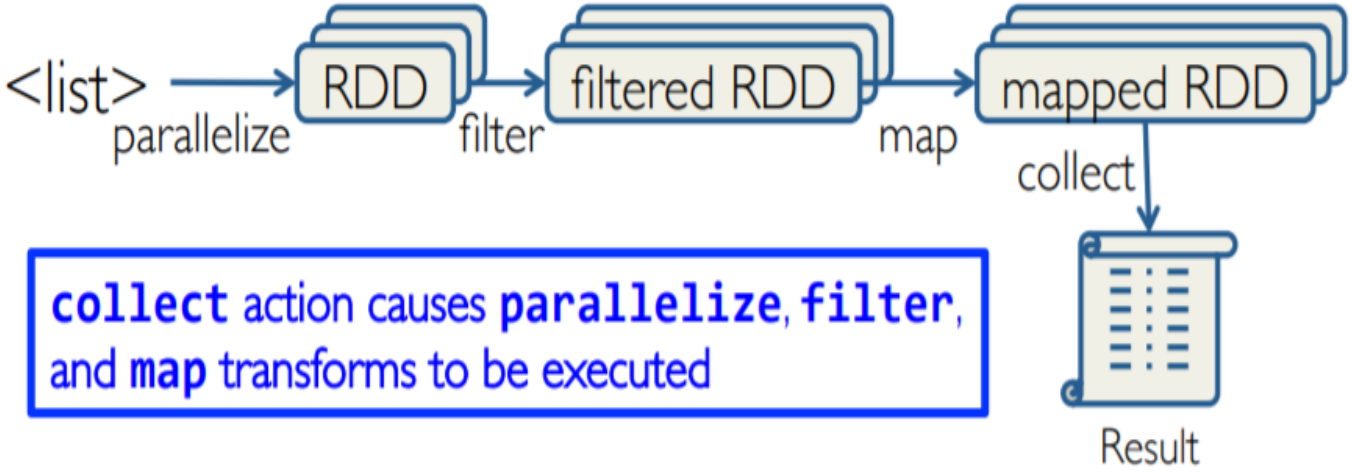
join w/ inputs not co-partitioned

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.

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



collect action causes **parallelize**, **filter**, and **map** transforms to be executed

Creating an RDD

- Create RDDs from Python collections (lists)

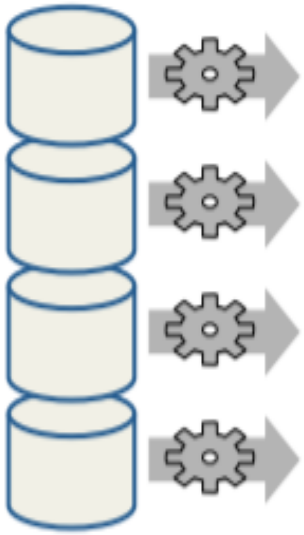
```
>>> data = [1, 2, 3, 4, 5]

>>> 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

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 Hive queries
5. 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.

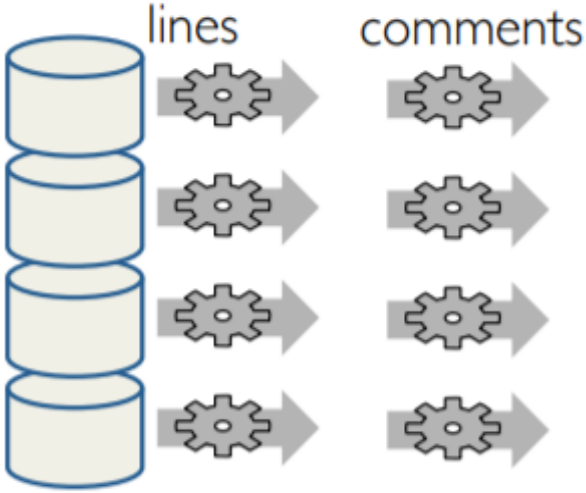
Dir() function: The dir() function is used to display the defined methods.

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

RDD - TRANSFORMATIONS

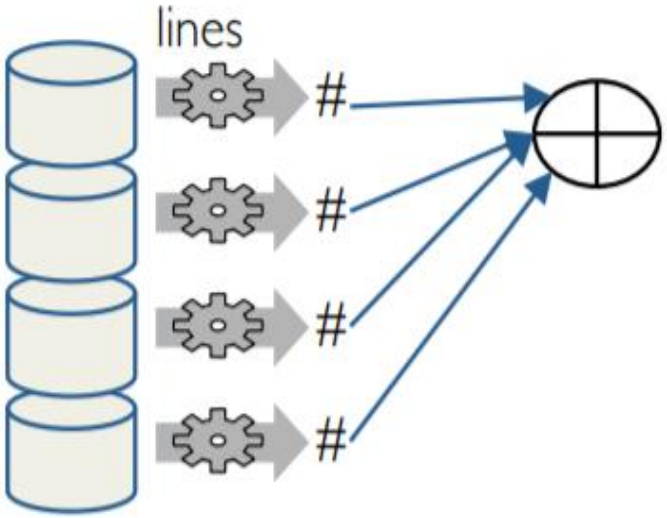
No.	RDD Action	Expecting Result
1	map()	Return a new distributed dataset formed by passing each element of the source through a function *func*.
2	filter()	Return a new dataset formed by selecting those elements of the source on which *func* returns true.
3	flatMap()	Similar to map, but each input item 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.

Spark Programming Model

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

print lines.count()
```

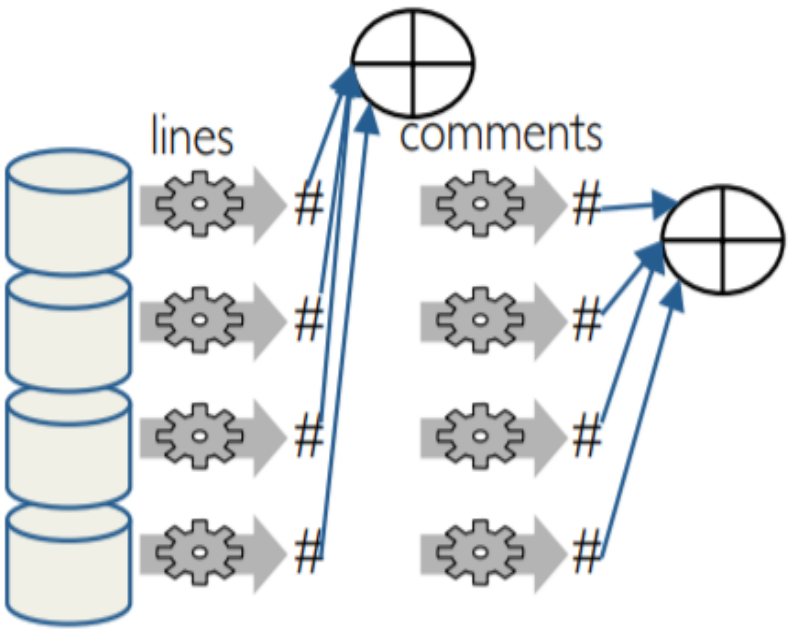


`count()` causes Spark to:

- 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

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

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

Review: Python `lambda` Functions

- Small anonymous functions (not bound to a name)

```
lambda a, b: a + b
```

  » returns the sum of its two arguments

- Can use lambda functions wherever function objects are required

- Restricted to a single expression

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

RDD Persist different storage level

MEMORY_ONLY Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, some partitions will not be cached and will be recomputed on the fly each time they're needed. This is the default level.

MEMORY_AND_DISK Store RDD as deserialized Java objects in the JVM. If the RDD does not fit
in memory, store the partitions that don't fit on disk, and read them from there when they're needed.

MEMORY_ONLY_SER (Java and Scala) Store RDD as serialized Java objects (one byte array per partition). This is generally more space-efficient than deserialized objects, especially when using a fast serializer, but more CPU-intensive to read.

MEMORY_AND_DISK_SER (Java and Scala) Similar to MEMORY_ONLY_SER, but spill partitions that don't fit in memory to disk instead of recomputing them on the fly each time they're needed.

DISK_ONLY Store the RDD partitions only on disk.

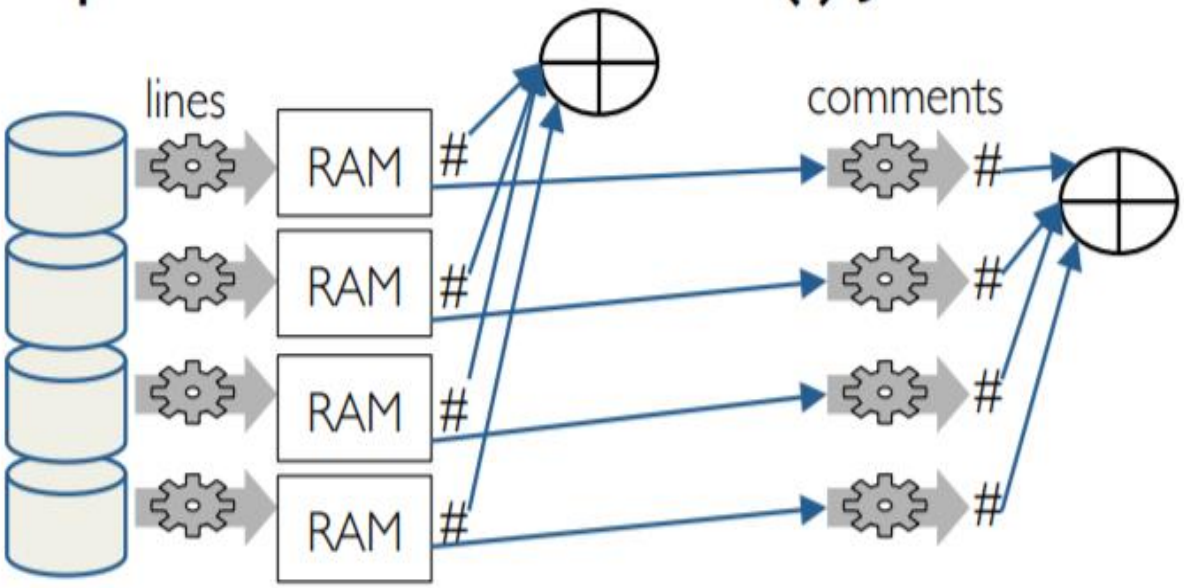
MEMORY_ONLY_2, MEMORY_AND_DISK_2, etc. Same as the levels above, but replicate each partition on two cluster nodes.

OFF_HEAP (experimental) Similar to MEMORY_ONLY_SER, but store the data in off-heap memory.

This requires off-heap memory to be enabled.

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.

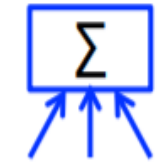
pySpark Shared Variables

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:

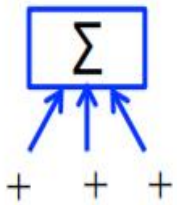
```
#Create Broadcast Variable using SparkContext.broadcast (LIST)
broadcast_v = sc.broadcast([1, 2, 3])
# Print the broadcast variable value using .value
broadcast_v.value
```

- **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)

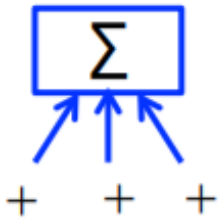
Delete cached copies of this broadcast on the executors. If the broadcast is used after this is called, it will need to be re-sent to each executor.



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

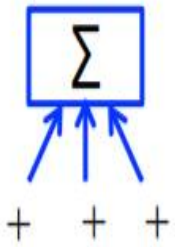
```
>>> accum = sc.accumulator(0)
>>> rdd = sc.parallelize([1, 2, 3, 4])
>>> def f(x):
>>>    global accum
>>>    accum += x

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



Accumulators Example

- Counting empty lines

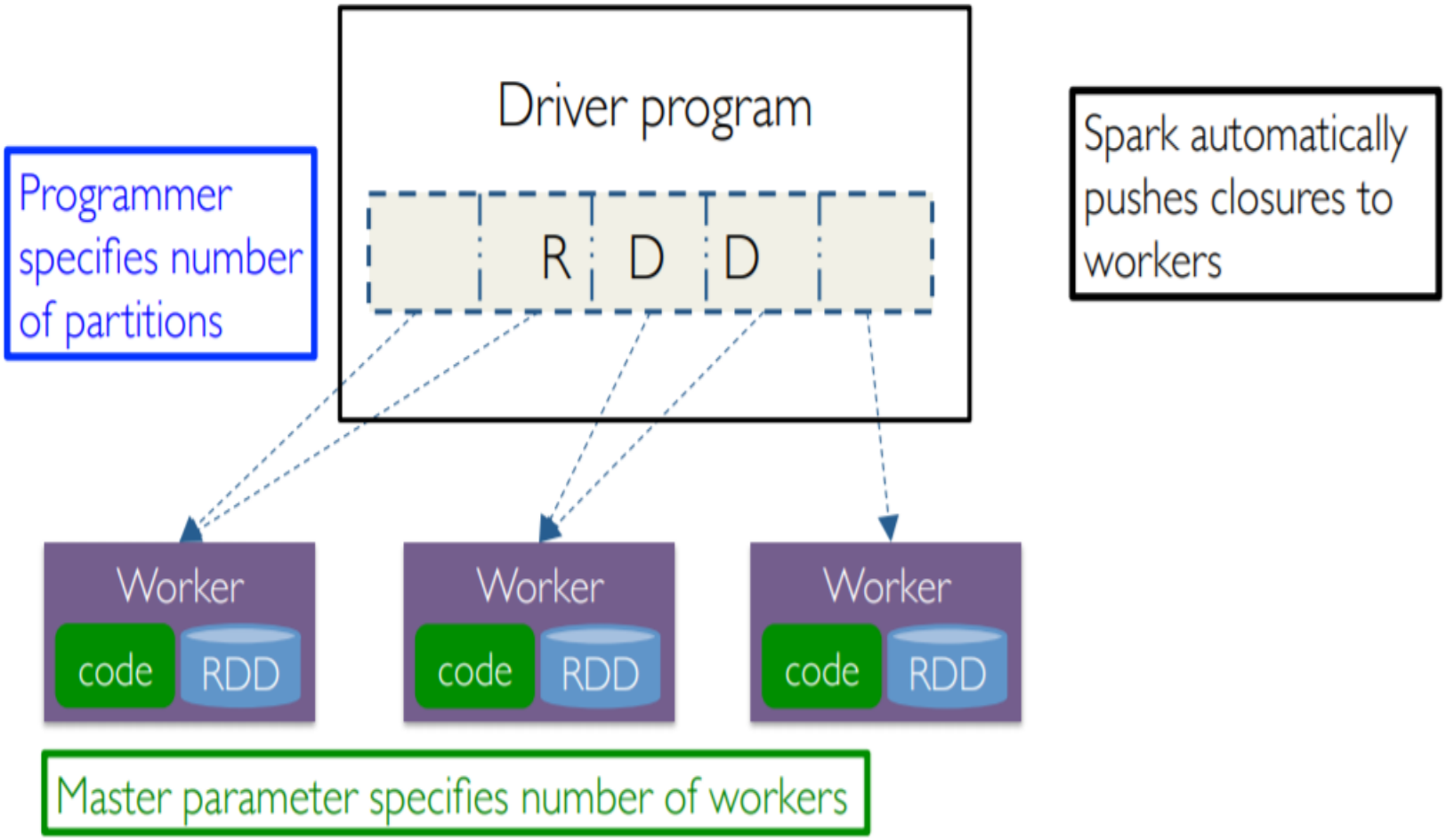
```
file = sc.textFile(inputFile)
# Create Accumulator[Int] initialized to 0
blankLines = sc.accumulator(0)

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

THANK YOU