APACHE SPARK

Dr. Muhammad Safyan

Map-Reduce Operations

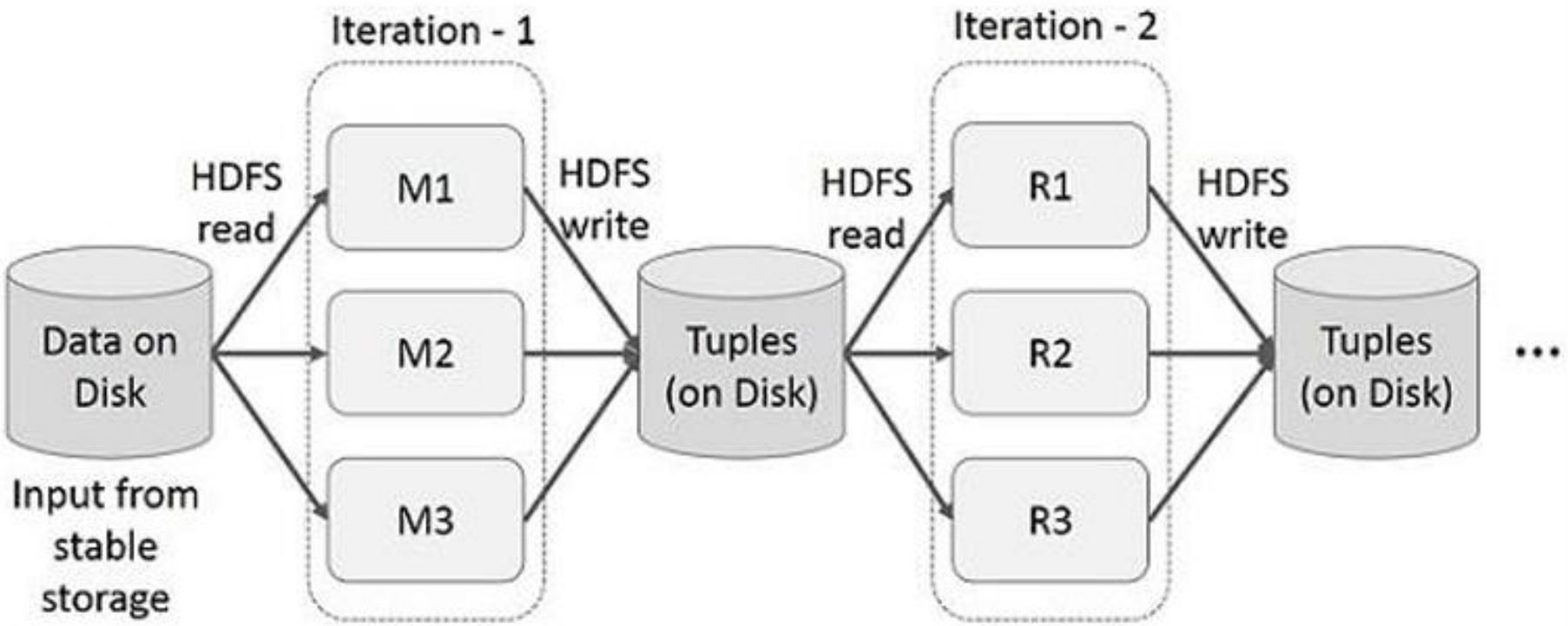
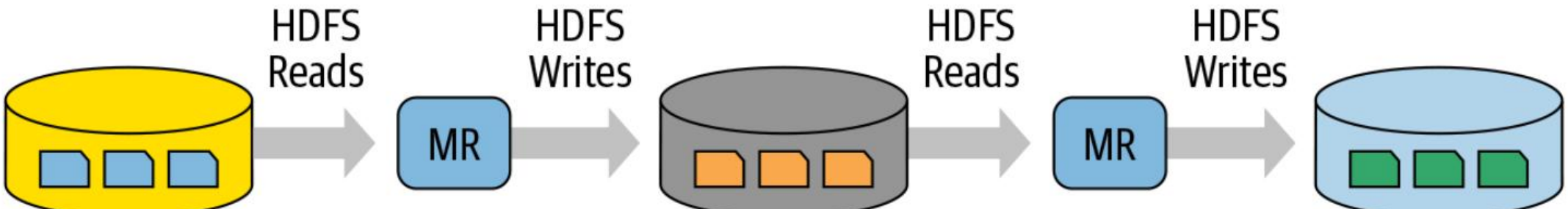
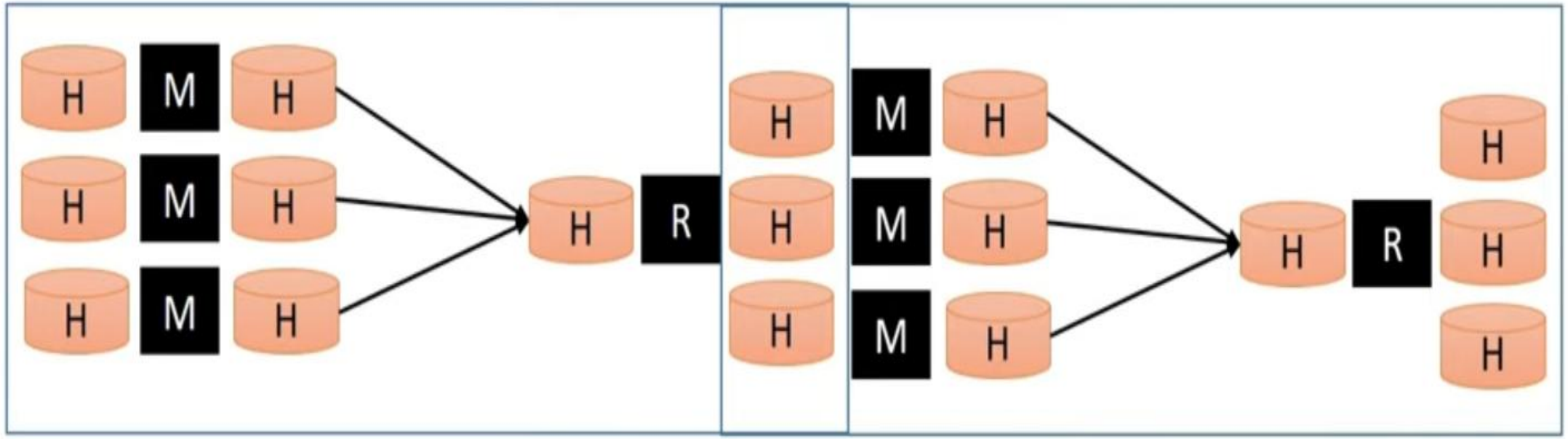


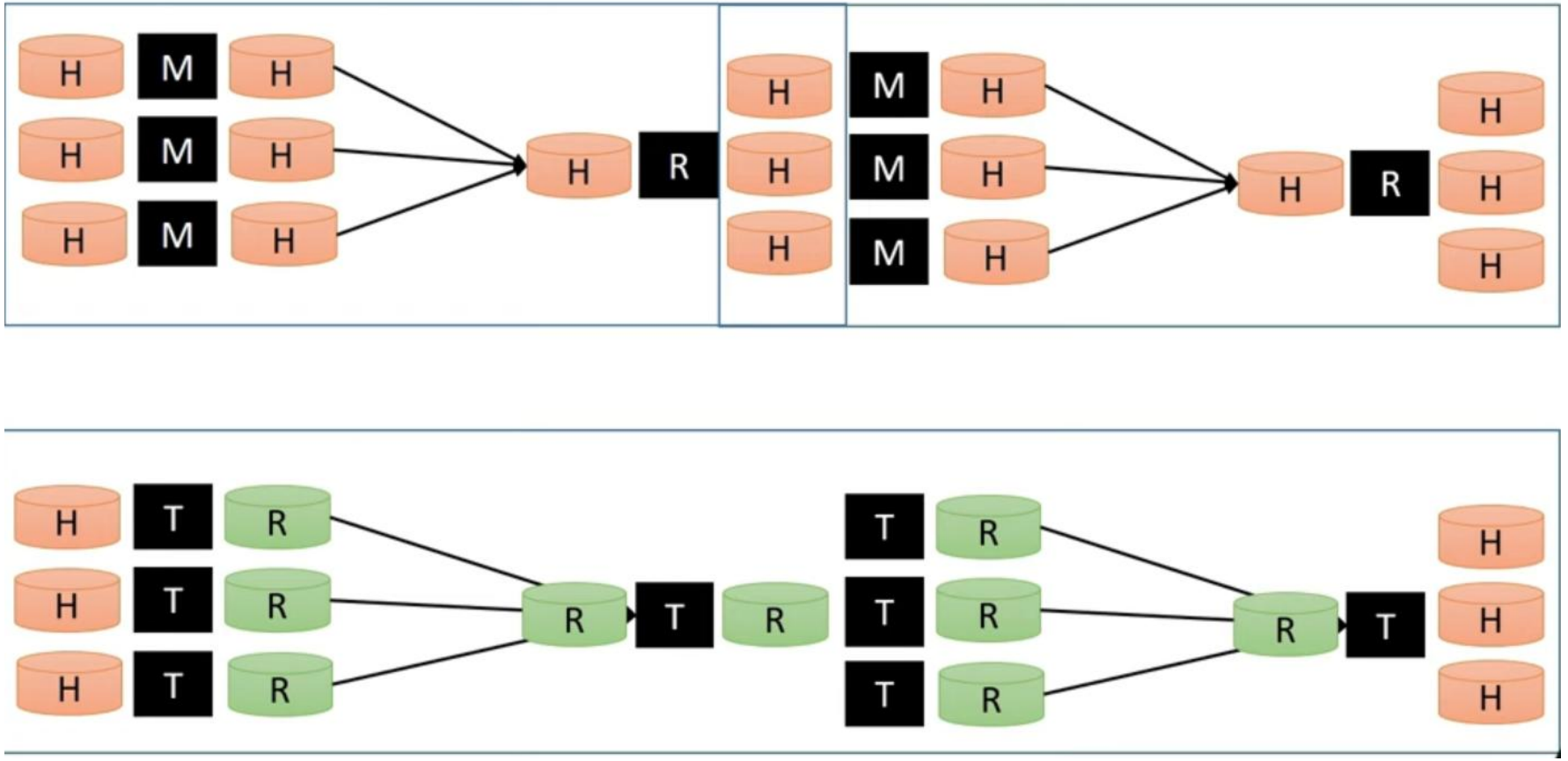
Figure: Iterative operations on MapReduce



Why spark is faster than MR



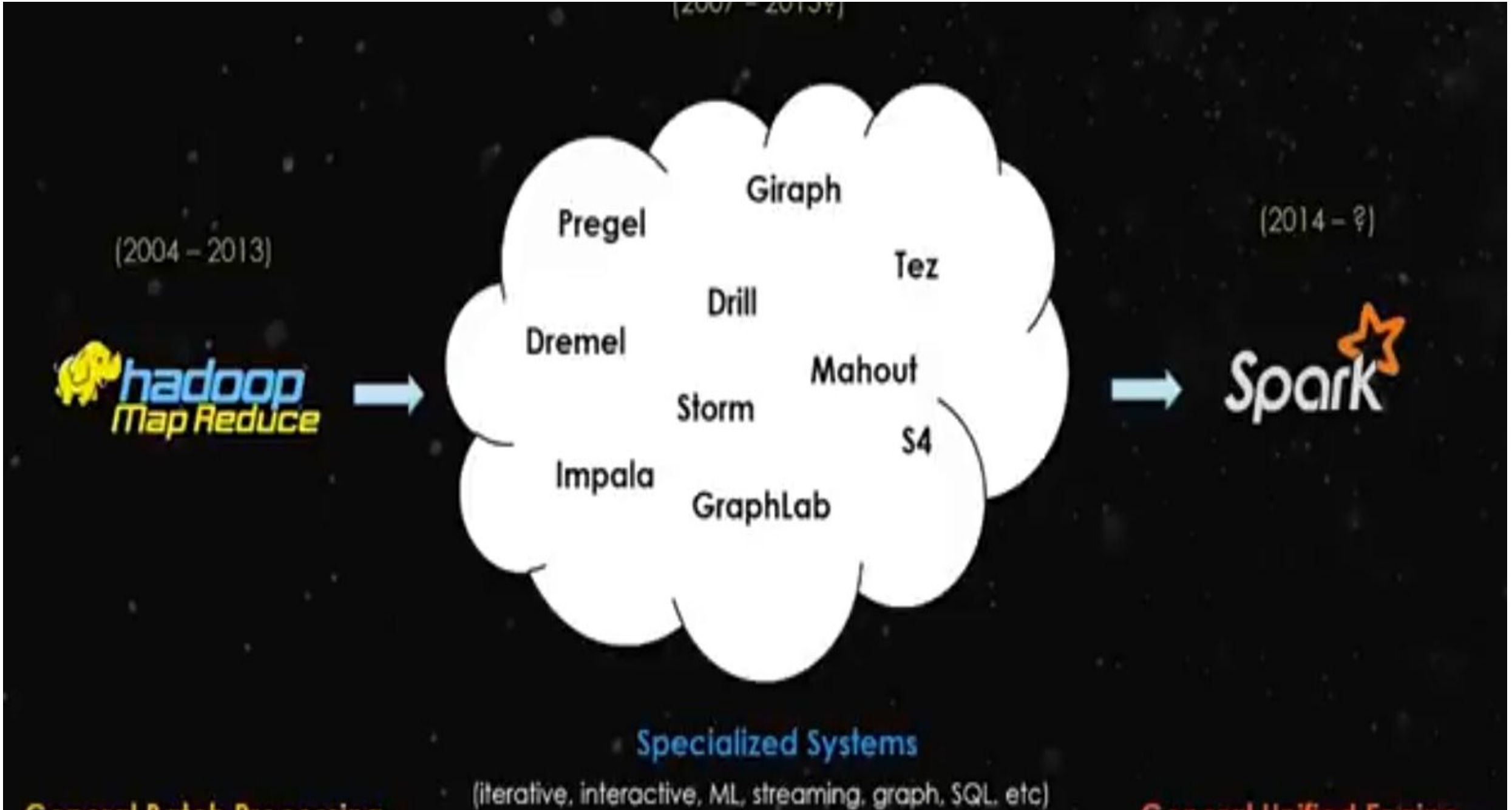
Why spark is faster than MR



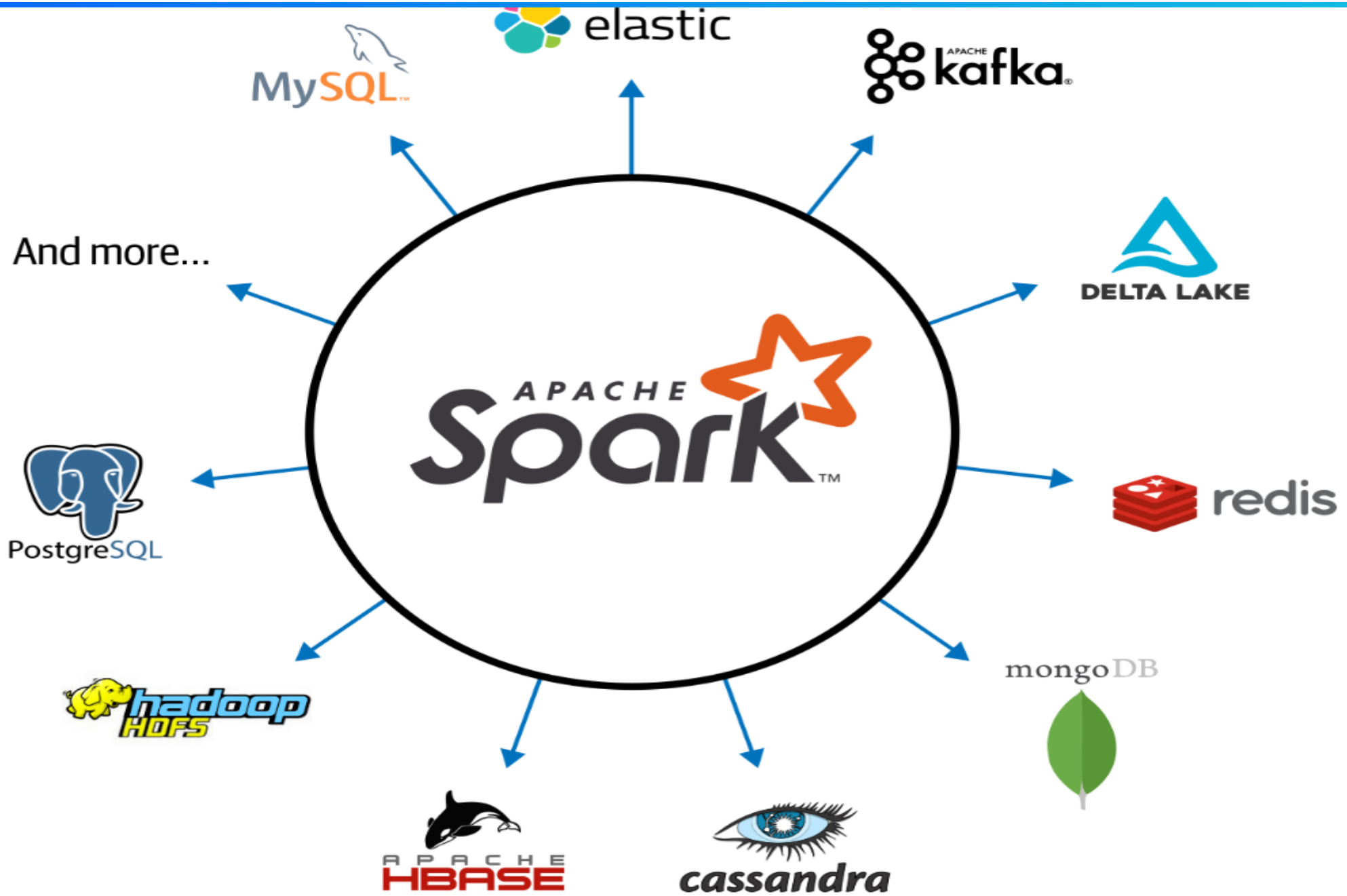
HDD : 100MBPS
RAM : 10GBPS

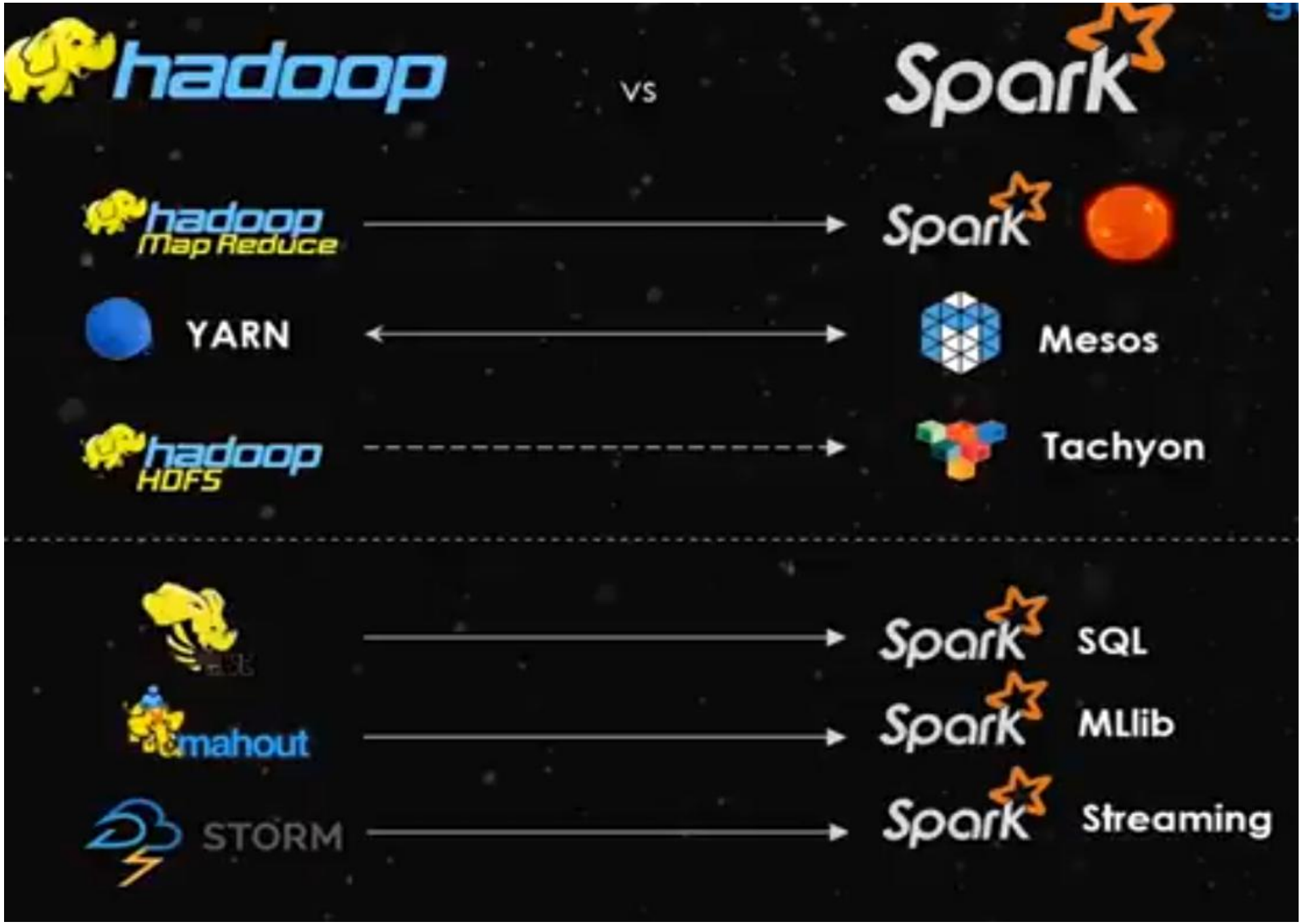
Limitations of MR

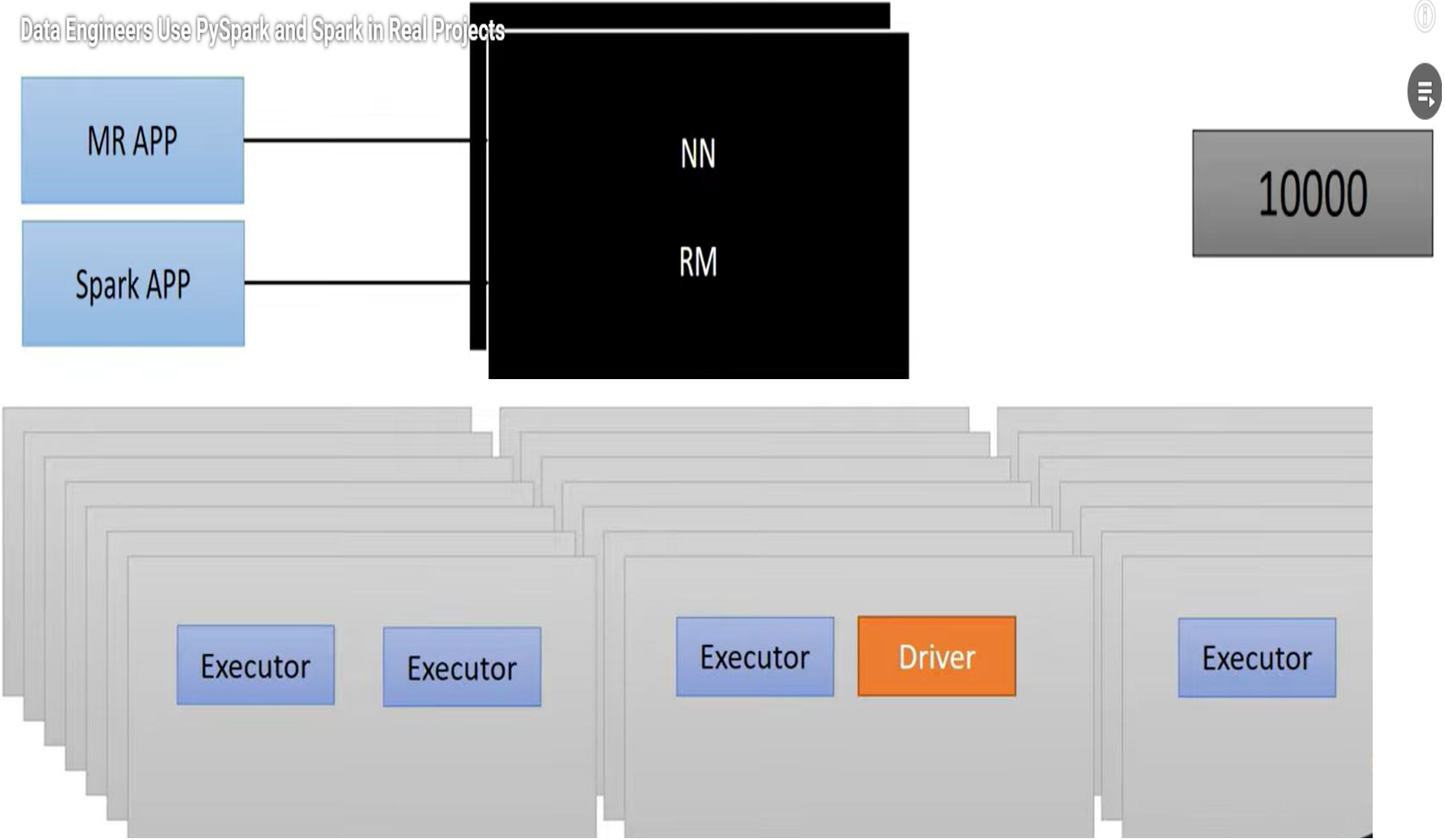
- - Interactive
- - Iterative | ML Algo | inception, interstellar, tensor
- - Code complexity
- - Consumes more time in development
- - Maintaining / Modifications MR code is difficult


(2004 – 2013)
hadoop
Map Reduce
Giraph
Pregel
Tez
Drill
Dremel
Mahout
Storm
S4
Impala
GraphLab
(2014 – ?)
Spark
Specialized Systems
(iterative, interactive, ML, streaming, graph, SQL, etc)


hadoop
HDFS
mongoDB
RDBMS
Neo4j
APACHE
HBASE
FILE://
SQL
Streaming
GraphX
MLlib

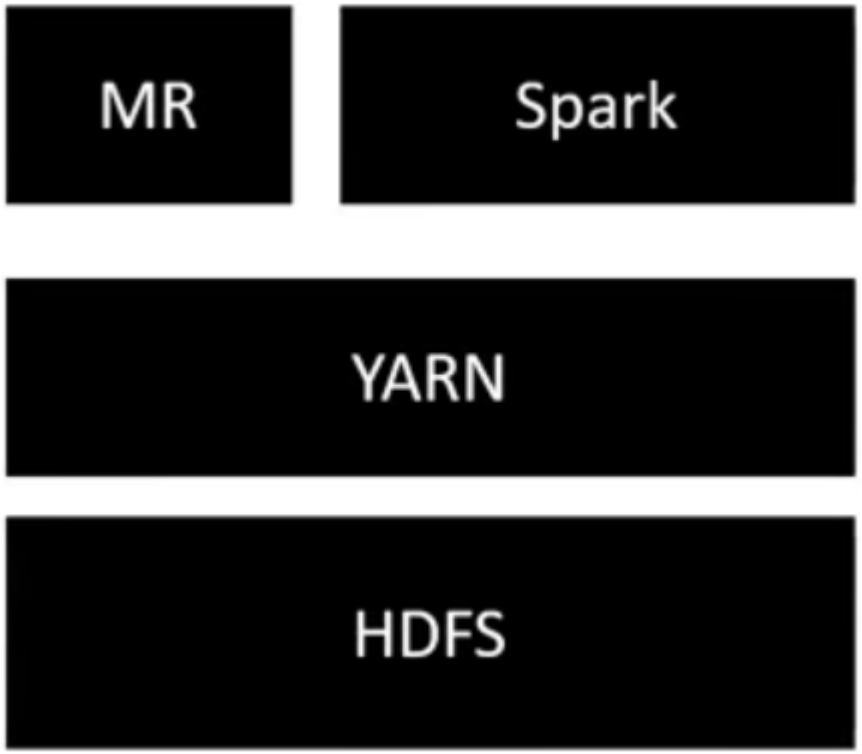

elastic
MySQL
APACHE kafka
And more...
DELTA LAKE
APACHE Spark™
PostgreSQL
redis
hadoop HDFS
mongoDB
APACHE HBASE
cassandra

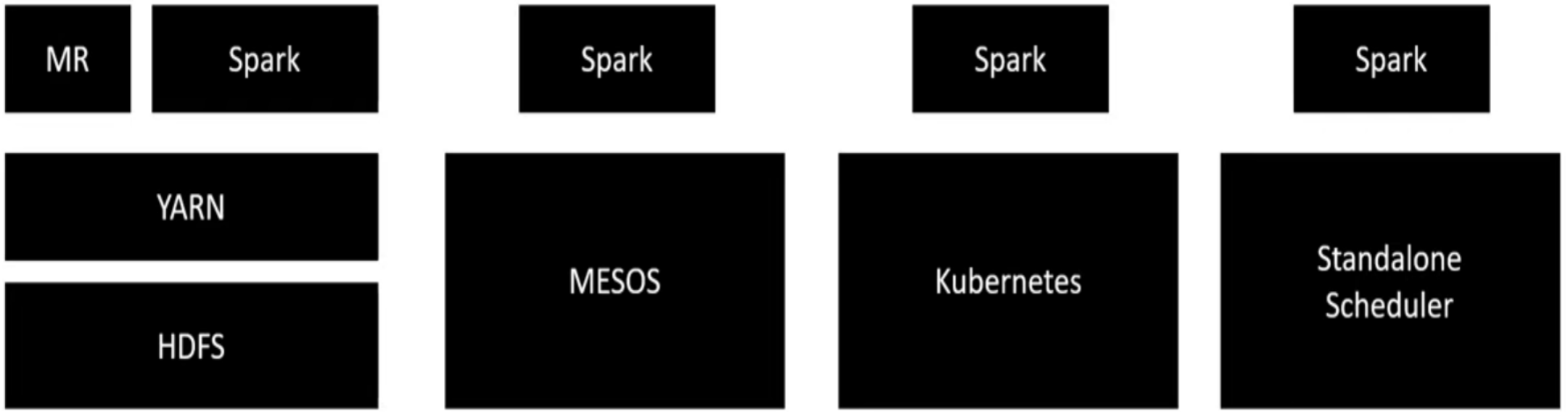

hadoop
vs
Spark
hadoop
Map Reduce
Spark
YARN
Mesos
hadoop
HDFS
Tachyon
Spark
SQL
mahout
Spark
MLlib
STORM
Spark
Streaming


MR APP
Spark APP
NN
RM
10000
Executor
Executor
Executor
Driver
Executor

Is Spark replacement of Hadoop?

Is Spark replacement of Hadoop?





Is Spark replacement to Hadoop MR ? : True

What Is Hadoop?

Open source

Distributed storage and computing

Can scale to petabytes of data

Consists of:

- Hadoop Distributed File System (HDFS)
- MapReduce

HDFS: Current State

- Still a very good option for cheap storage of large quantities of data
- Suitable for enterprises with in-house data centers
- Cloud alternatives like Amazon S3, Google Cloud Storage, Azure Blob

MapReduce: Current State

- Scales horizontally
- Very slow as it uses disk for data storage
- Being replaced by faster alternatives like Apache Spark

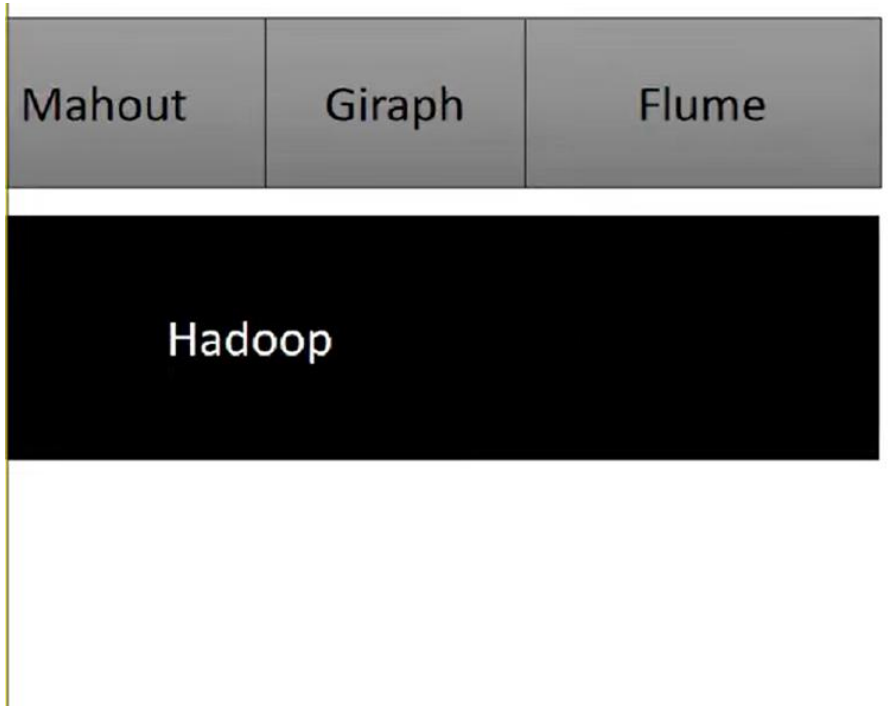
Why Spark is faster than MR

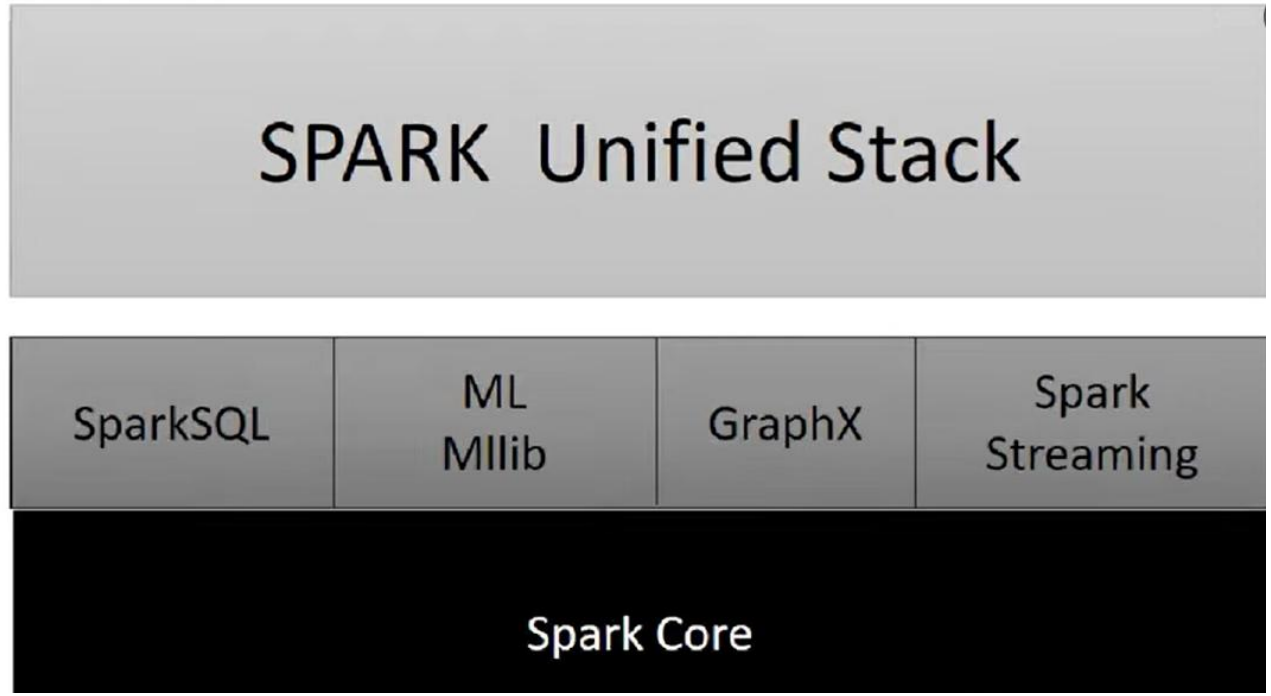
- In memory Processing
- Spark job: stages and tasks | Not limited to only map and Reduce phases
- Catalyst Optimizer
  - Rule Based Optimization
  - Cost Model
- Adaptive Query Execution
- Dynamic Partition Pruning

Spark Read/Write

- Spark is not a data storage component. It is data processing Engine.
- - HDFS, LFS, S3, DataLake, DeltaLake, NoSQL, RDBMS

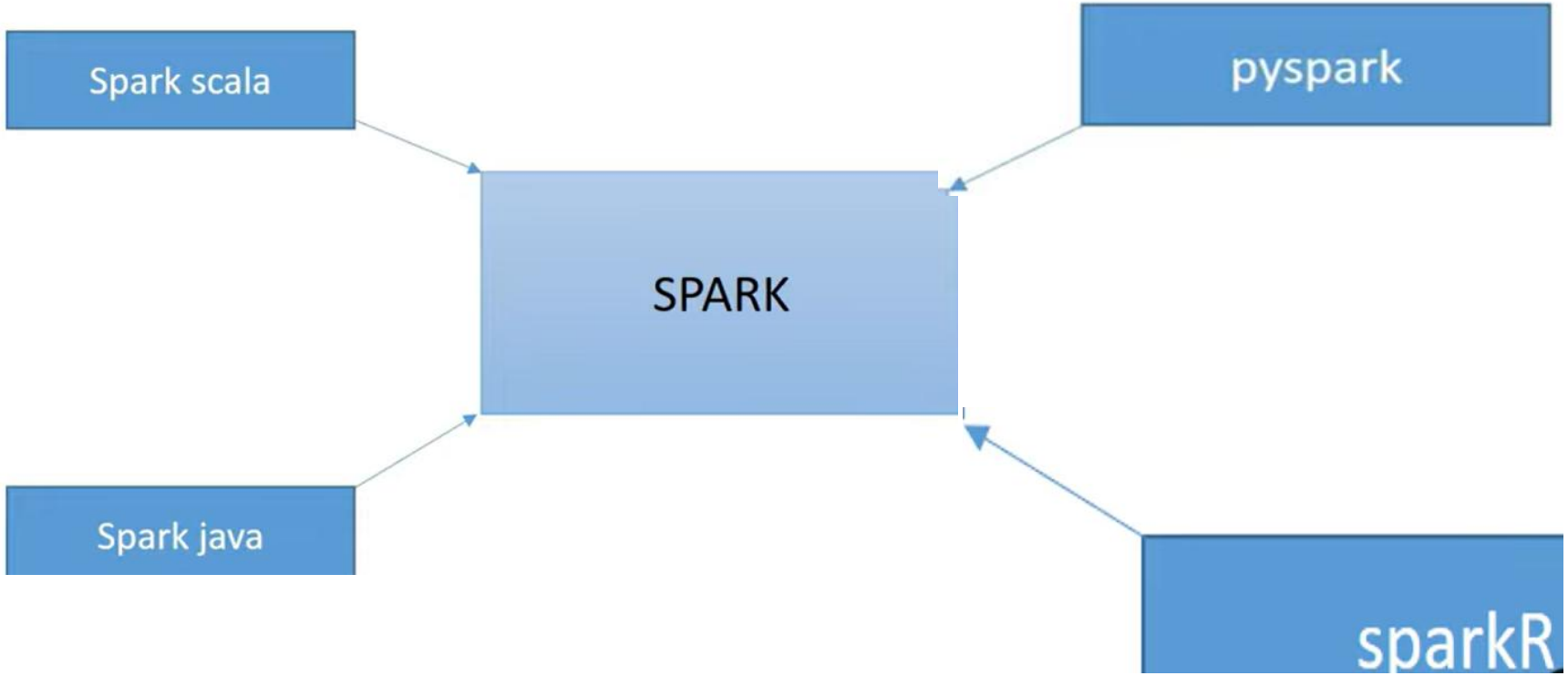
SPARK A UNIFIED STACK


Mahout
Giraph
Flume
Hadoop


SPARK Unified Stack
SparkSQL
ML
MIlib
GraphX
Spark
Streaming
Spark Core

SPAKR LANGUAGES

- - Scala
- - Python
- - R
- - Java
- - SQL



What Is Apache Spark?

- Open source
- Large-scale distributed data processing engine
- Uses memory to speed up computations
- Batch, streaming, ML, and graph capabilities

What Is Apache Spark?

- Scala, Java, Python, and R support
- The most popular big-data processing platform today

The Power of Spark and Hadoop

HDFS provides large-scale distributed storage

Spark provides large-scale fast processing

Spark is well integrated with Hadoop

- Parallelism
- Optimizations
- Caching
- YARN

Powerful Combination

Utilize the power of Spark processing over HDFS storage to build scalable, high-performance processing jobs.

Hadoop Storage Formats

- Raw text (blobs)
- Structured text files (CSV, XML, JSON)
- Sequence files
- Avro
- ORC
- Parquet

Text Files

- Simple to read/write
- Low performance
- More storage
- No schema

Avro

- Language-neutral data serialization
- Row format
- Self-describing schema support
- Compressible
- Splittable

Parquet

- Columnar format
- Read-only selected columns
- Schema support
- Compressible (column level) and splittable
- Supports nested data structures

Parquet for Analytics

Parquet provides overall better performance and flexibility for analytics applications.

Hadoop Compression Options

- Snappy
- LZO
- GZIP
- bzip2

LZO

- Moderate compression
- Excellent processing performance
- Splittable
- Requires separate license

GZIP

- Very good compression
- Moderate processing performance
- Not splittable
- Ideal for container-type applications

bzip2

- Excellent compression
- Slower processing performance
- Splittable
- Ideal for archival type applications

Choosing a Compression Format

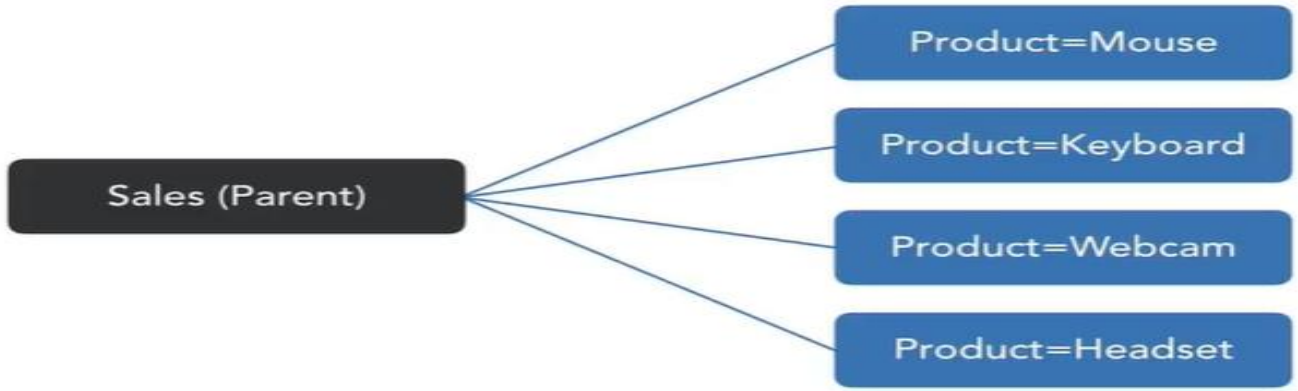
Choose a format based on the application. Do proof of concepts to understand actual performance and storage impacts.

Need for Partitioning

- HDFS does not have the concept of indexes
- Even for reading one row, the entire file should be read
- Partitioning provides a way to read only a subset of data
- Multiple attributes can be used for hierarchical partitioning

Partitioning

- Split data into directories based on individual values of attributes



Choosing Partitions

Choose attributes with a limited list of values and those that are most used in SELECT filters.

Bucketing

- Similar to partitioning
- Subdirectories based on a hash value are generated based on an attribute
- Controls the number of unique directories created
- Even distribution
- Ideal for attributes with large number of unique values

Choosing Buckets

Choose attributes with a large number of unique values and those that are most used in SELECT filters.

Storage Best Practices

- Understand the most used read and write patterns for your data
- Determine what needs optimization and what can be compromised
- Choose options carefully as they cannot be easily changed
- Run tests on actual data to understand performance and storage characteristics
- Choose partitioning and bucketing keys wisely

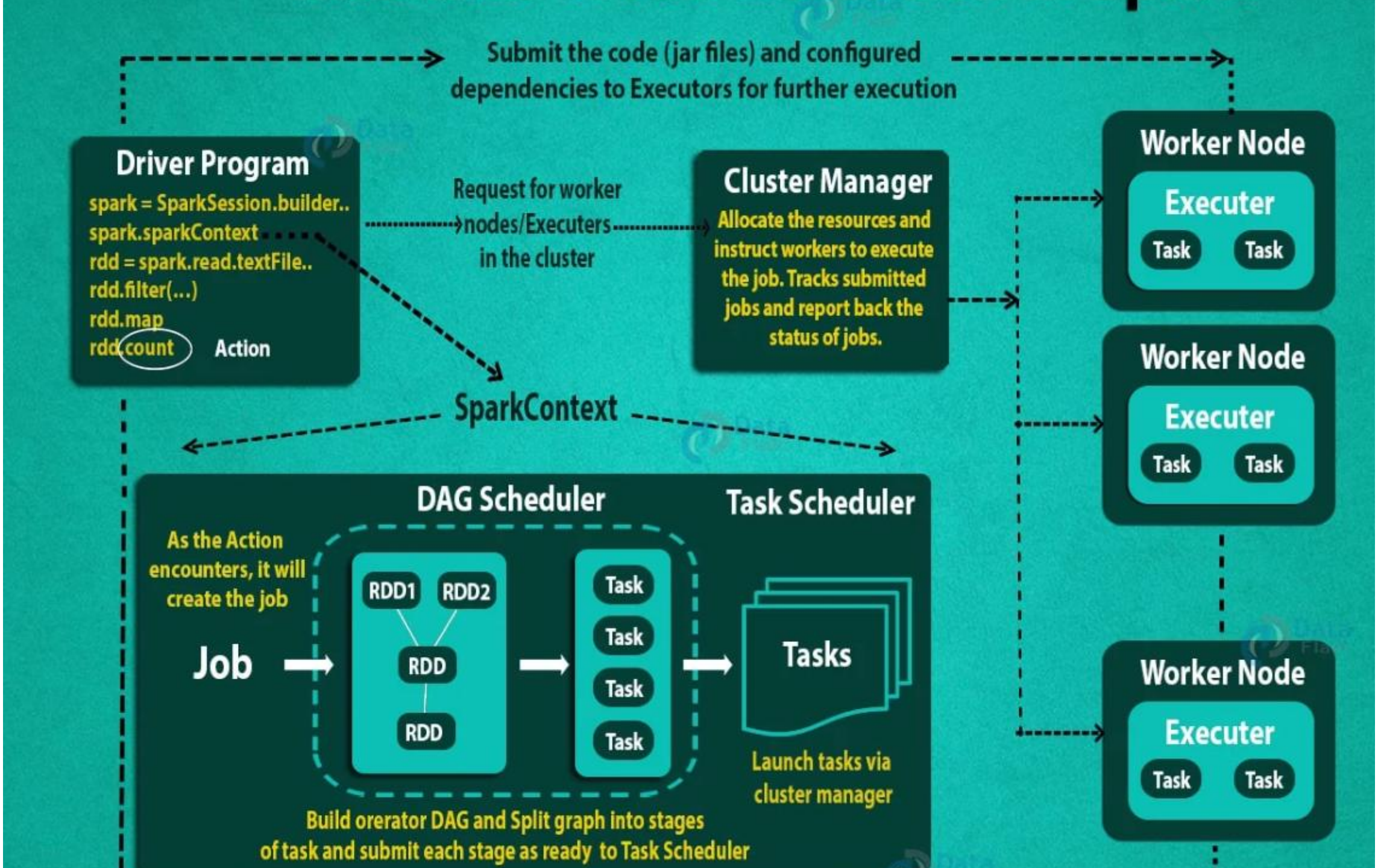
Data Ingestion Best Practices

Enable parallelism for maximum write performance

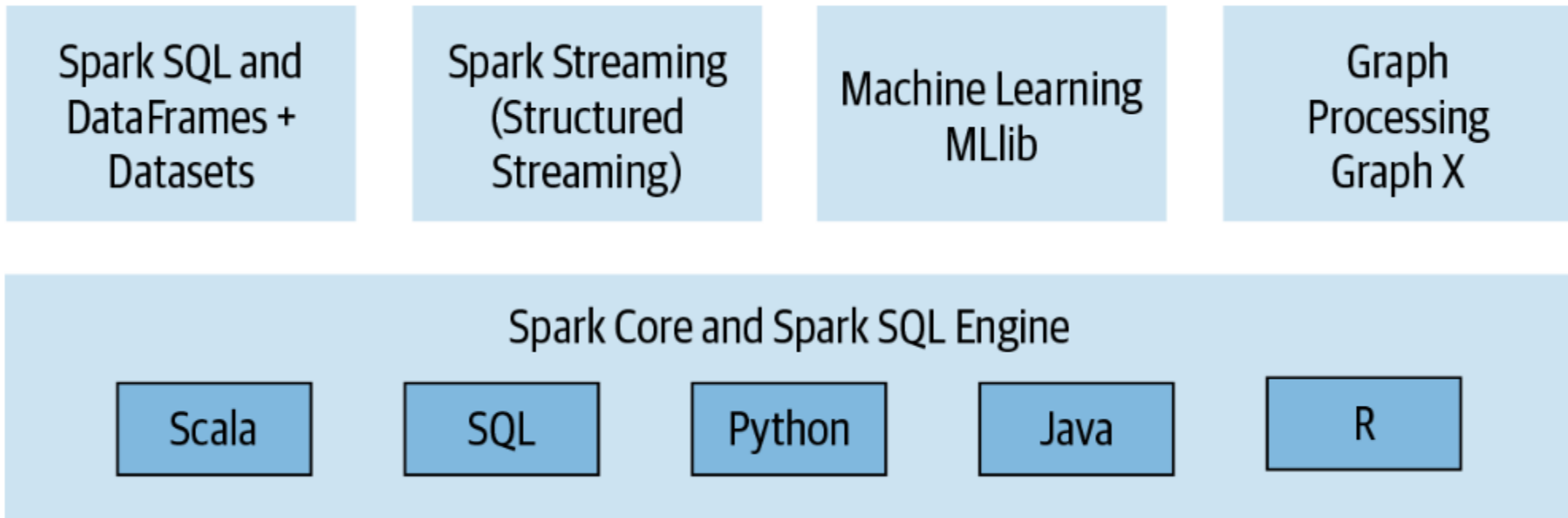
Use APPEND for incremental data ingestion

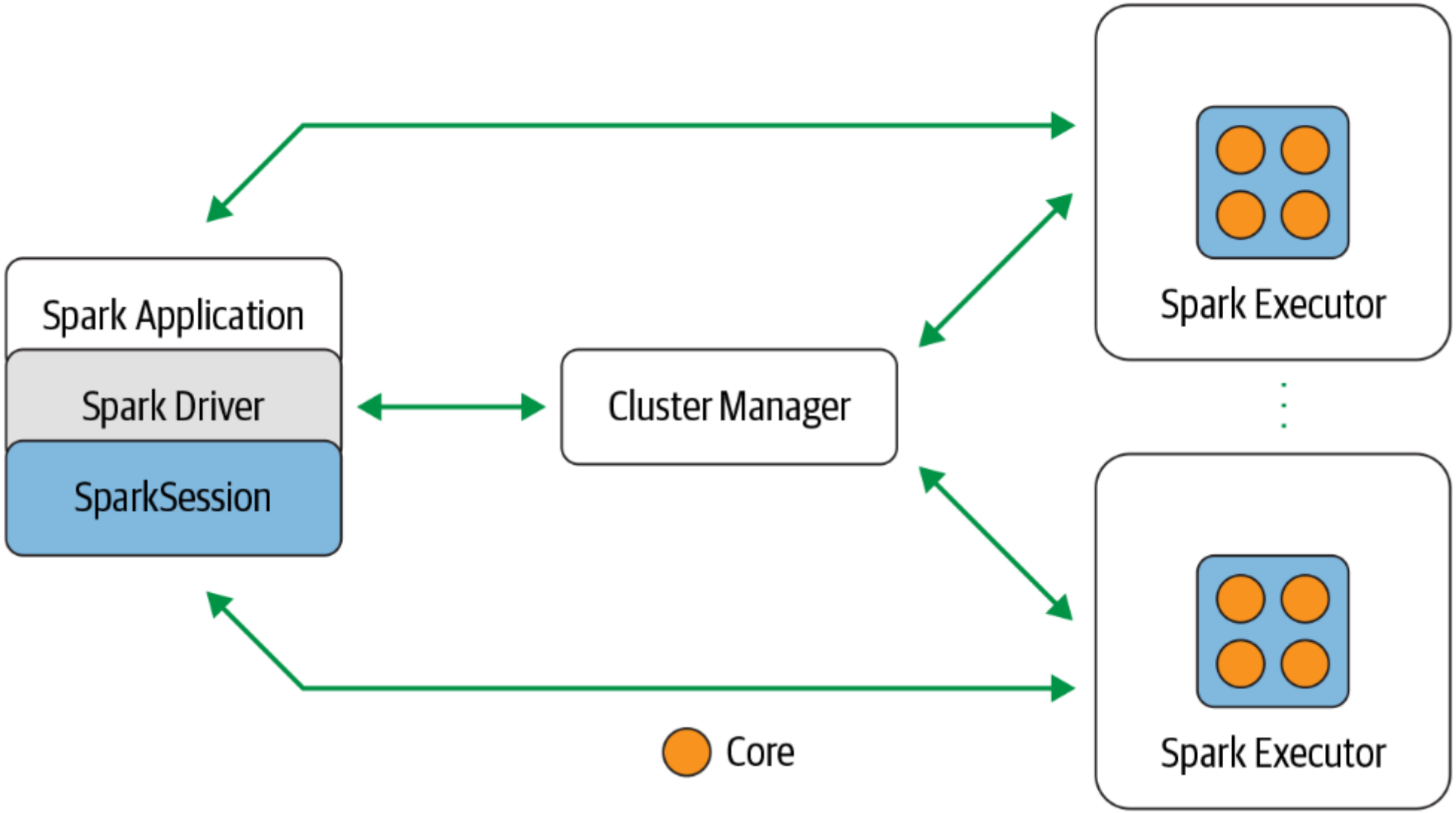
External data reads – use sources that provide parallelism

- JDBC
- Kafka
- Break down large files into smaller files


Submit the code (jar files) and configured dependencies to Executors for further execution
Driver Program
spark = SparkSession.builder..
spark.sparkContext
rdd = spark.read.textFile..
rdd.filter(...)
rdd.map
rdd.count
Action
Request for worker nodes/Executers in the cluster
Cluster Manager
Allocate the resources and instruct workers to execute the job. Tracks submitted jobs and report back the status of jobs.
Worker Node
Executer
Task
Task
Worker Node
Executer
Task
Task
Worker Node
Executer
Task
Task
SparkContext
DAG Scheduler
Task Scheduler
As the Action encounters, it will create the job
Job
RDD1
RDD2
RDD
RDD
Task
Task
Task
Task
Tasks
Launch tasks via cluster manager
Build orerator DAG and Split graph into stages of task and submit each stage as ready to Task Scheduler

User submits job → SparkContext → Cluster Manager → Worker Nodes → Executors → Task Execution → Result


Spark SQL and DataFrames + Datasets
Spark Streaming (Structured Streaming)
Machine Learning MLlib
Graph Processing Graph X
Spark Core and Spark SQL Engine
Scala
SQL
Python
Java
R


Spark Application
Spark Driver
SparkSession
Cluster Manager
Spark Executor
Spark Executor
Core

Distributed data and partitions

Actual physical data is distributed across storage as partitions residing in either HDFS or cloud storage (see Figure 1-5). While the data is distributed as partitions across the physical cluster, Spark treats each partition as a high-level logical data abstraction—as a DataFrame in memory. Though this is not always possible, each Spark executor is preferably allocated a task that requires it to read the partition closest to it in the network, observing data locality.

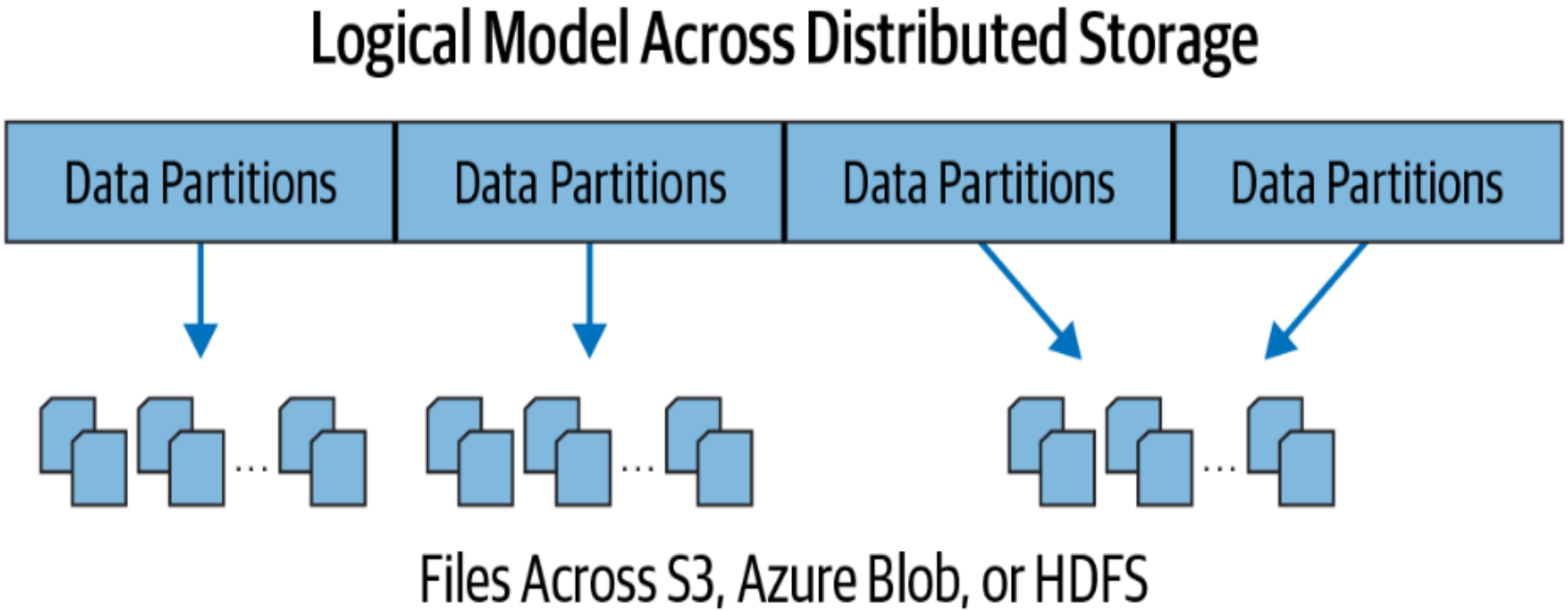
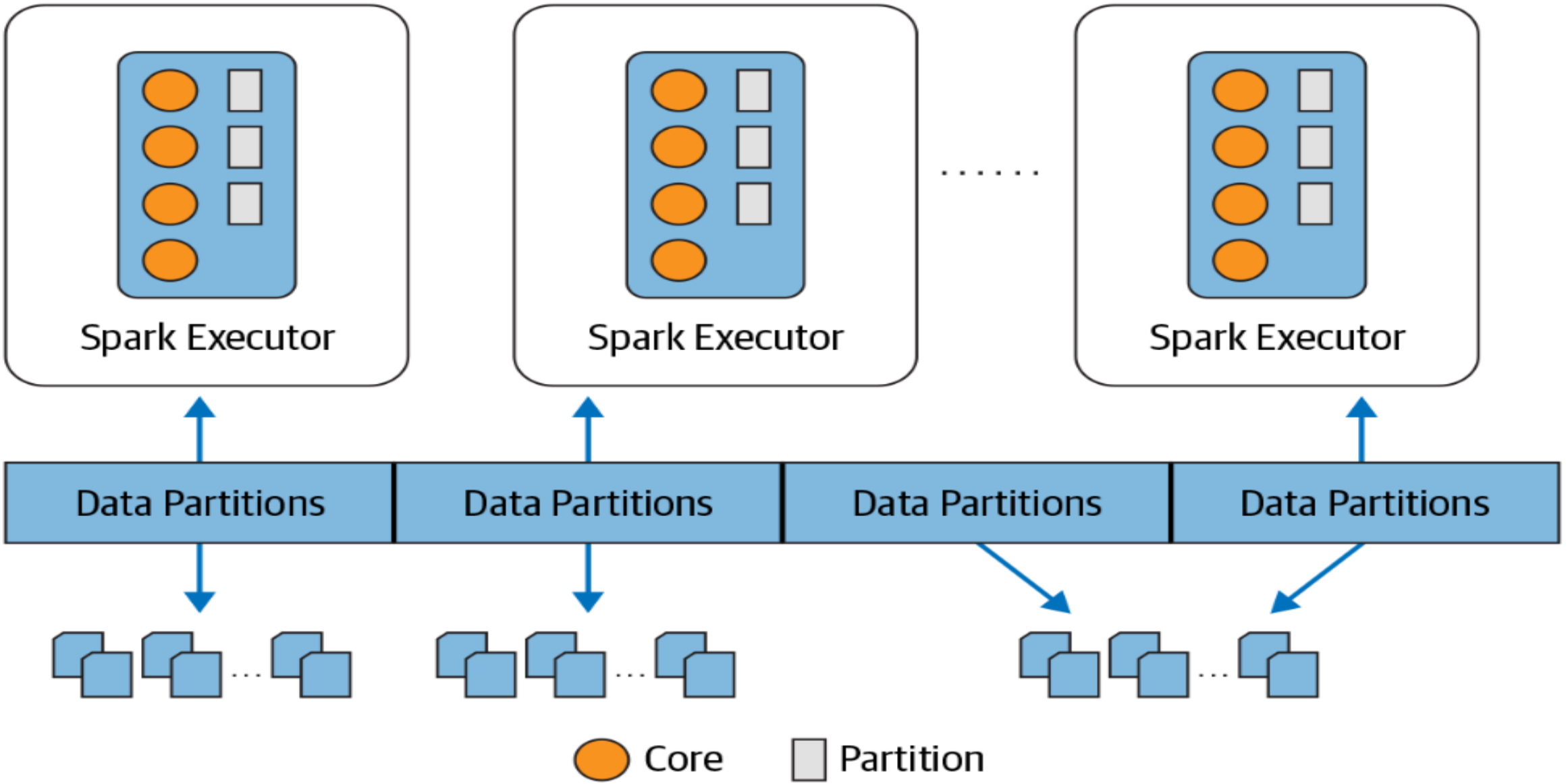
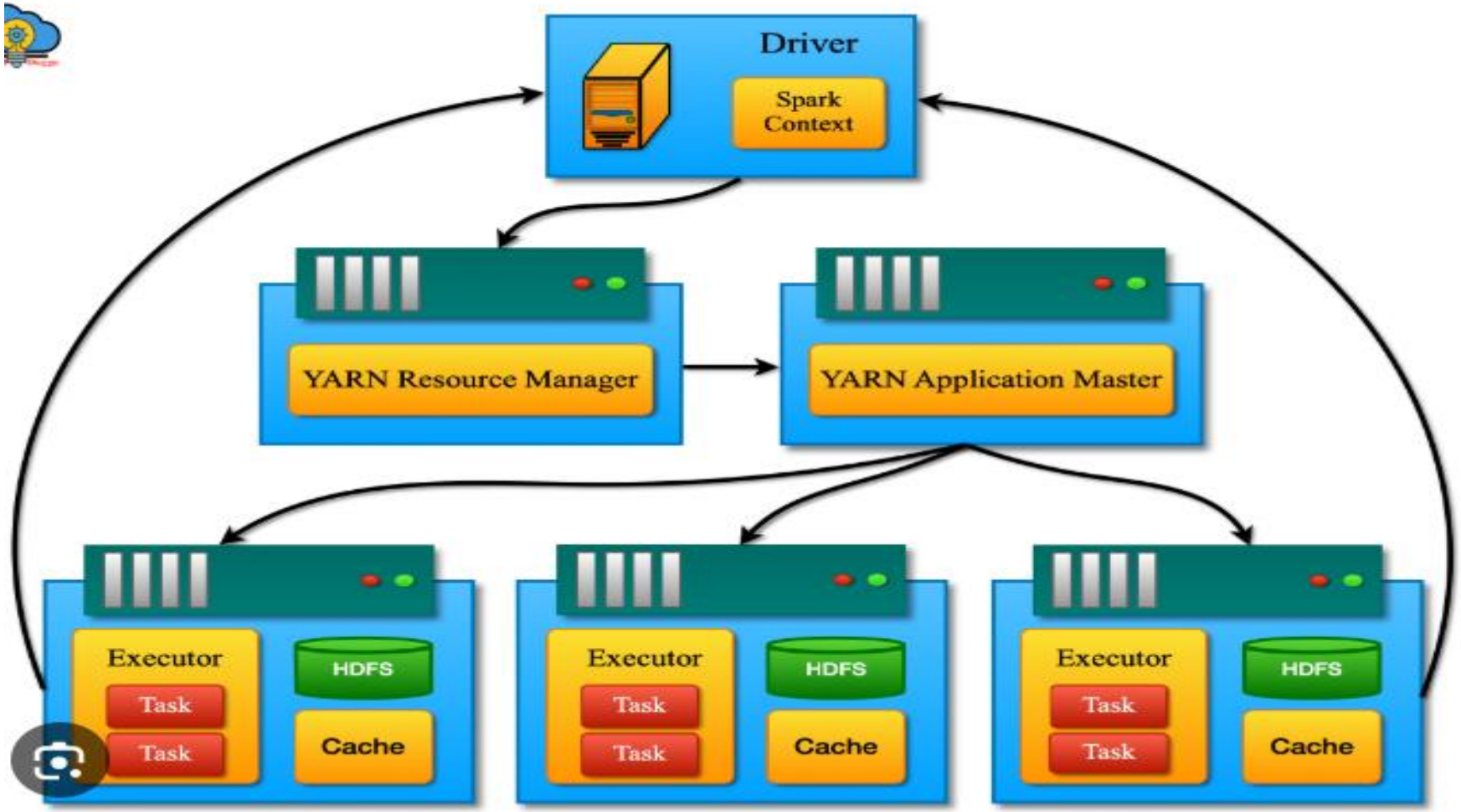
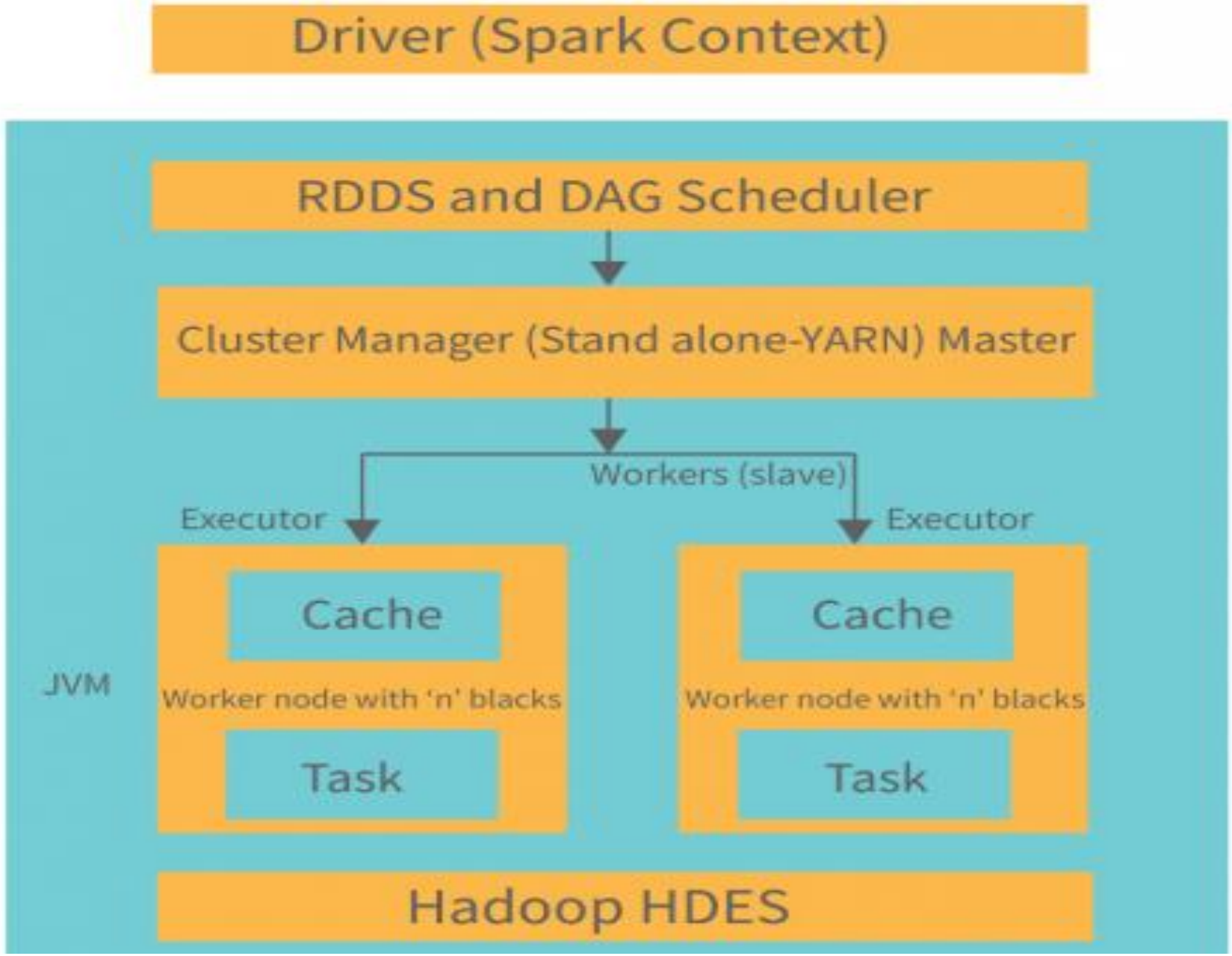


Figure 1-5. Data is distributed across physical machines

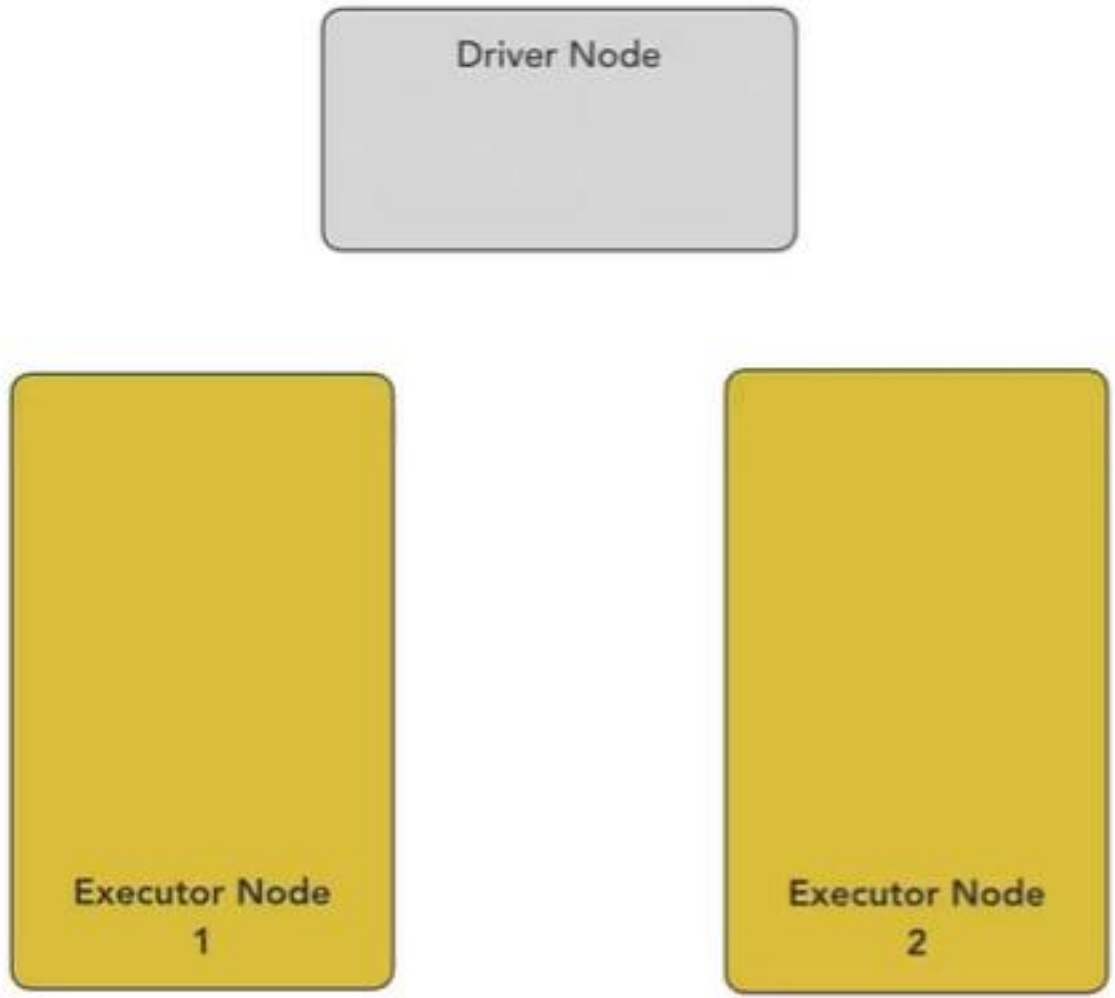
Partitioning allows for efficient parallelism. A distributed scheme of breaking up data into chunks or partitions allows Spark executors to process only data that is close to them, minimizing network bandwidth. That is, each executor's core is assigned its own data partition to work on (see Figure 1-6).



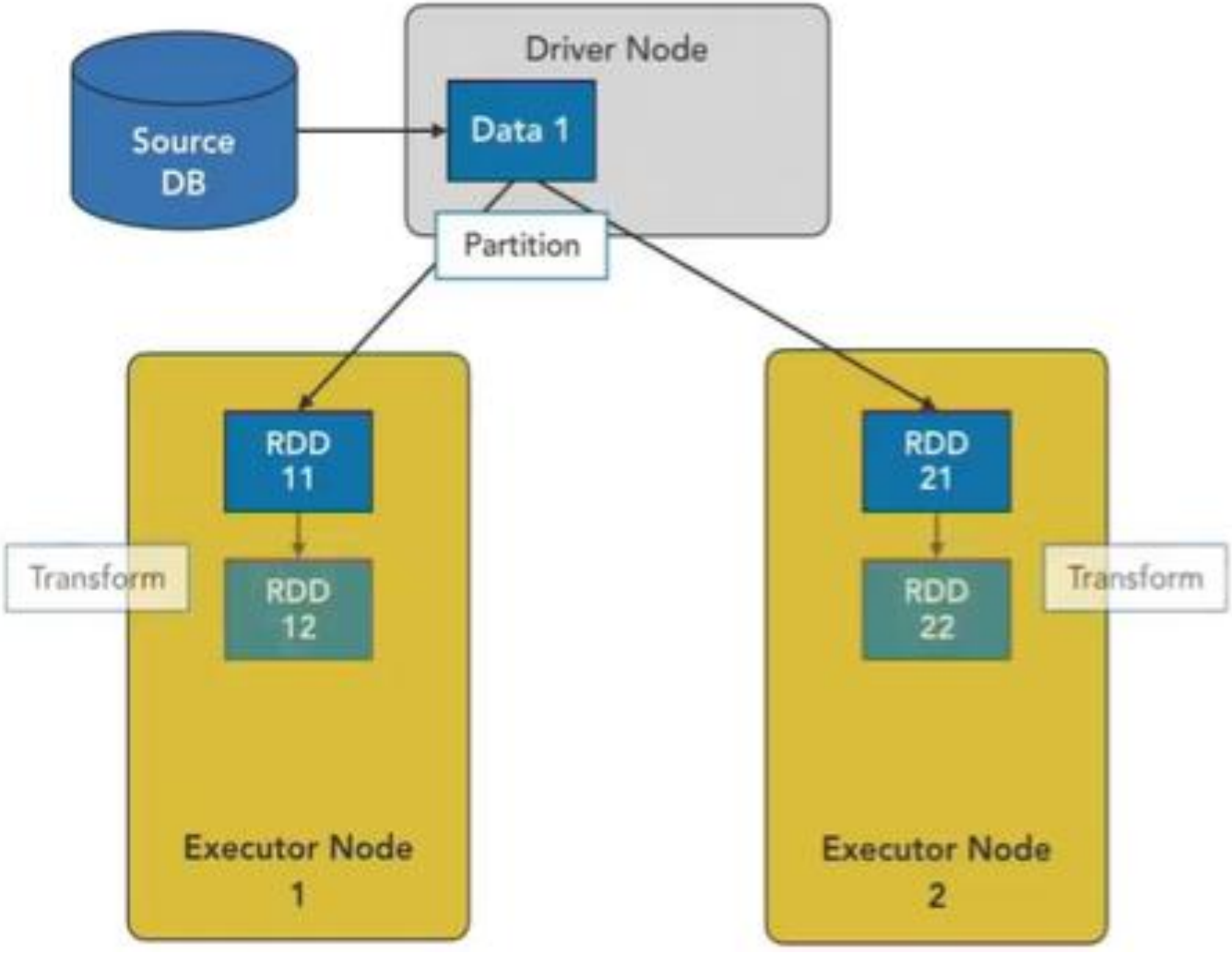

Driver
Spark Context
YARN Resource Manager
YARN Application Master
Executor
Task
Task
HDFS
Cache
Executor
Task
Task
HDFS
Cache
Executor
Task
Task
HDFS
Cache


Driver (Spark Context)
RDDS and DAG Scheduler
Cluster Manager (Stand alone-YARN) Master
Workers (slave)
Executor
Executor
Cache
Cache
JVM
Worker node with 'n' blacks
Worker node with 'n' blacks
Task
Task
Hadoop HDES

Spark Architecture and Execution

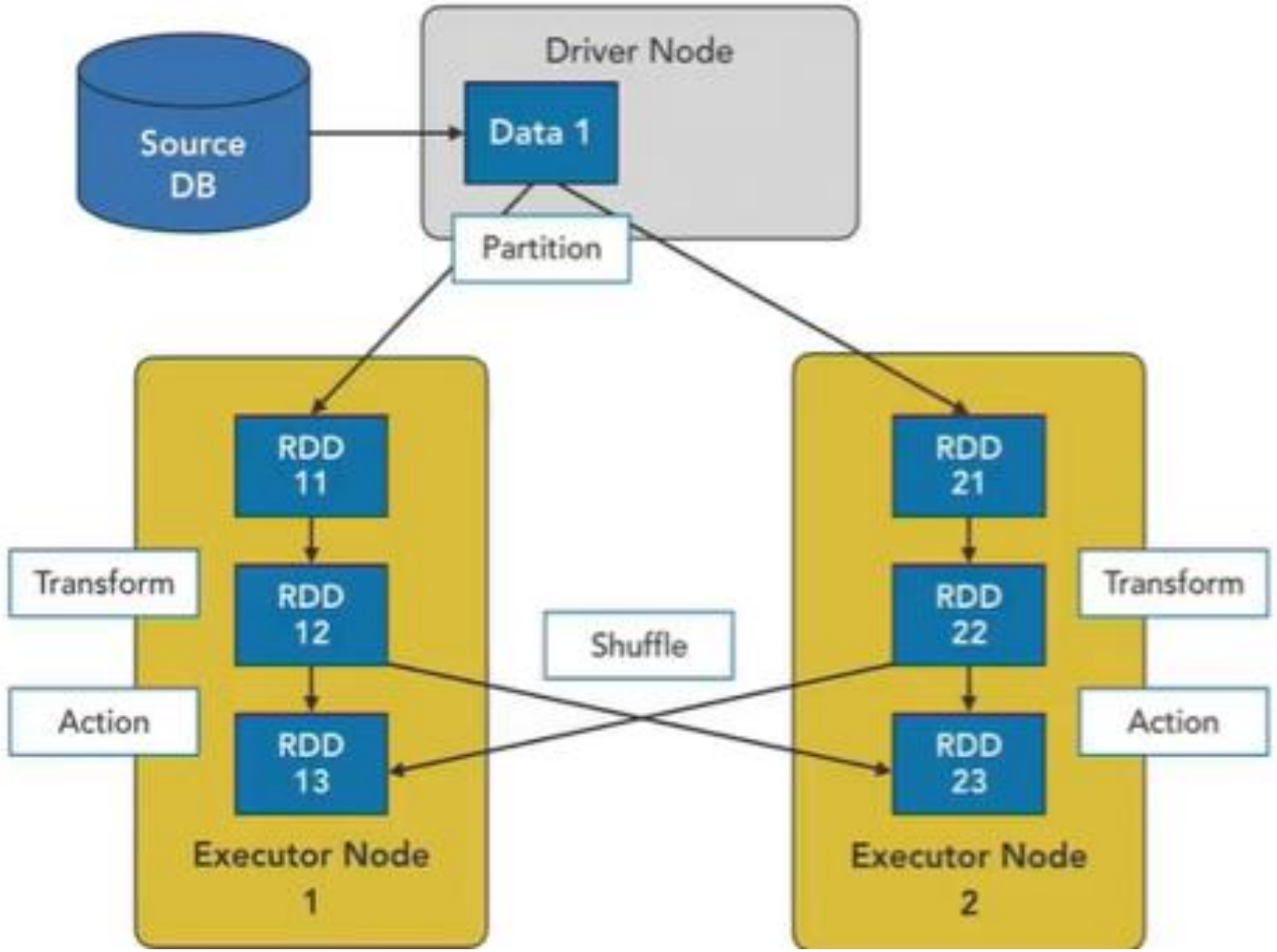


Spark Architecture and Execution



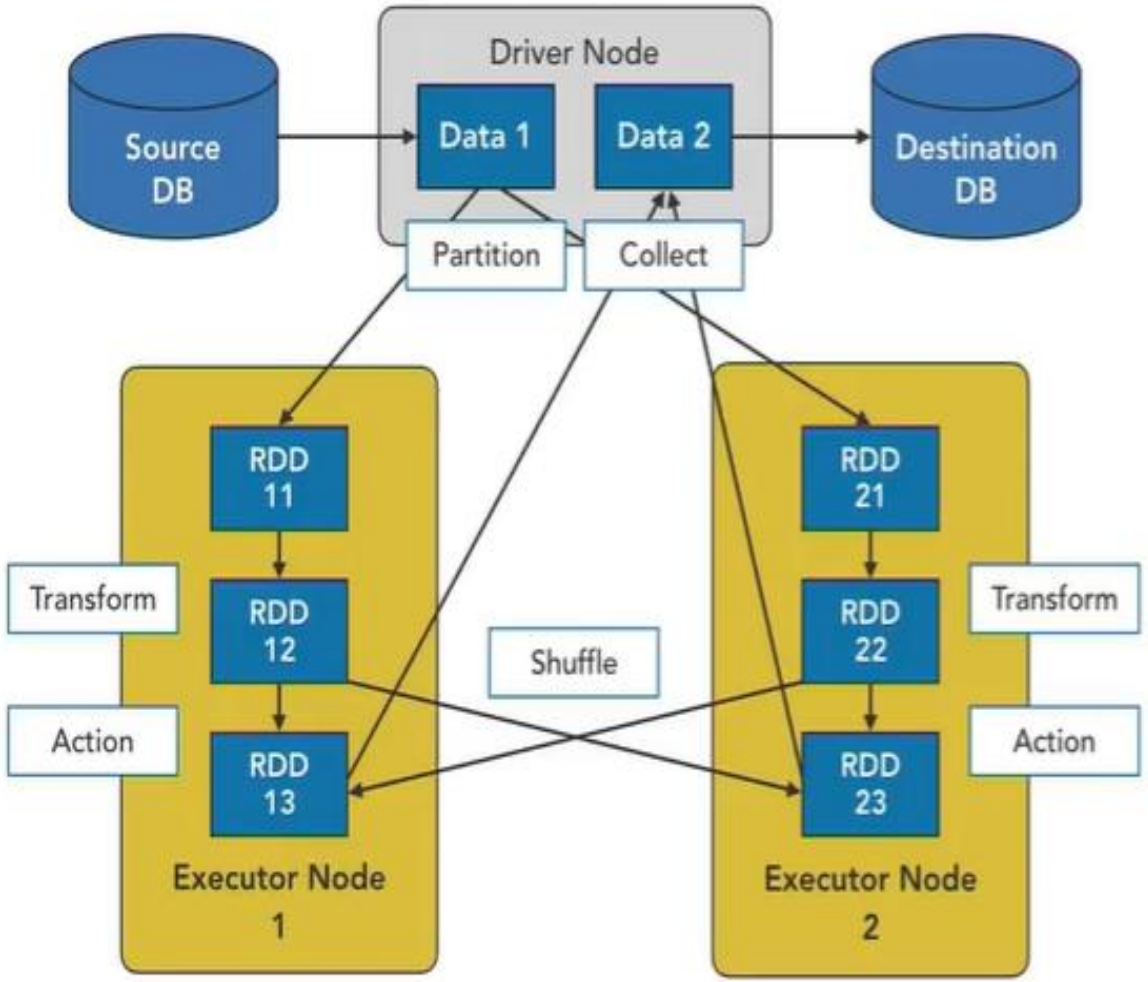
1. Driver node reads Data 1.
2. Data 1 is broken into partitions (RDDs) and distributed to executor nodes.
3. Transform operation happens locally.

Spark Architecture and Execution



1. Driver node reads Data 1.
2. Data 1 is broken into partitions (RDDs) and distributed to executor nodes.
3. Transform operation happens locally.
4. Action operation creates shuffles.

Spark Architecture and Execution



1. Driver node reads Data 1.
2. Data 1 is broken into partitions (RDDs) and distributed to executor nodes.
3. Transform operation happens locally.
4. Action operation creates shuffles.
5. Driver node collects back results.

Spark Execution Plan

Lazy execution – only an action triggers execution

Spark optimizer comes up with a physical plan

Physical plan optimizes for:

- Reduced I/O
- Reduced shuffling
- Reduced memory usage

Spark Execution Plan

Spark executors can read and write directly from external sources when they support parallel I/O (for example, HDFS, Kafka, JDBC)

Data Extraction Best Practices

Reduce data read into memory by:

- Using filters based on partition key
- Reading only required columns

Use data sources and file formats that support parallelism

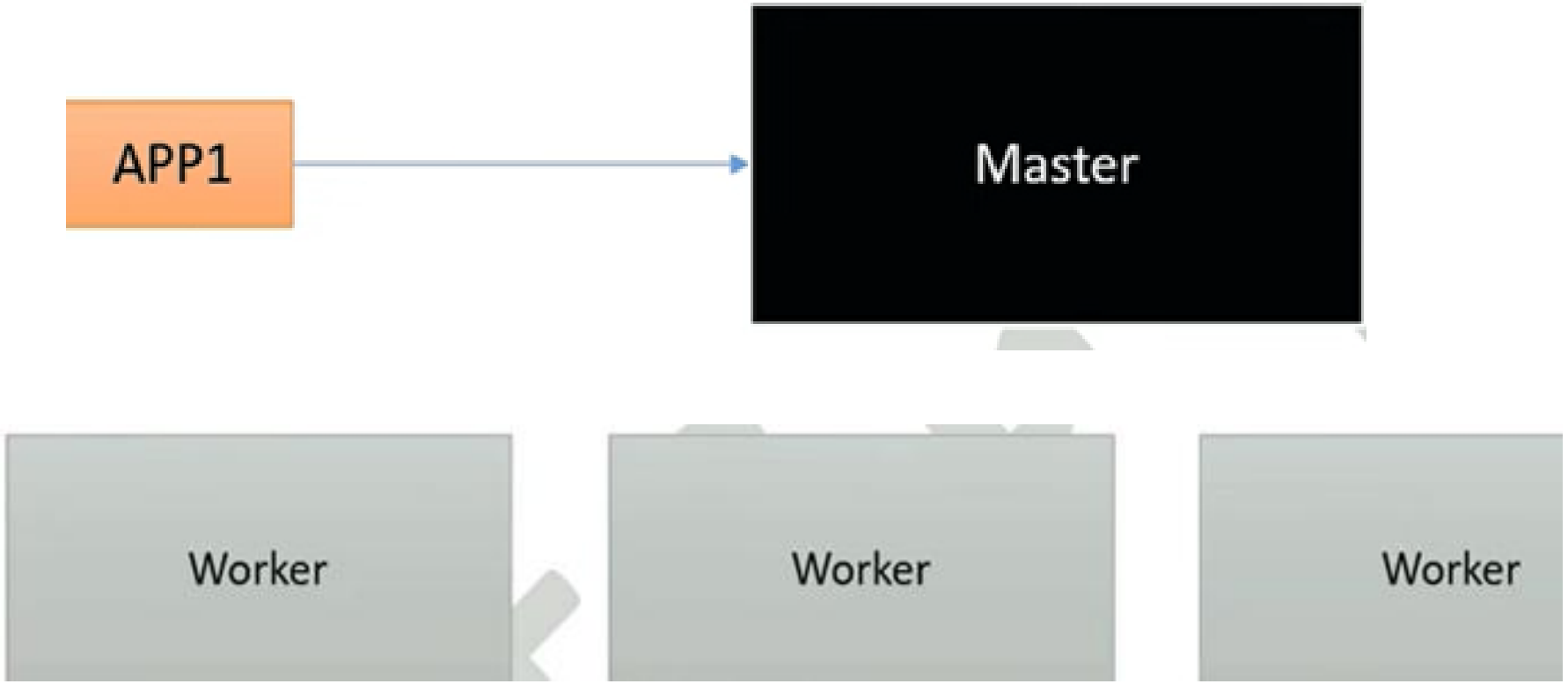
Keep number of partitions:

>= (Number of executors x Number of cores per executor)

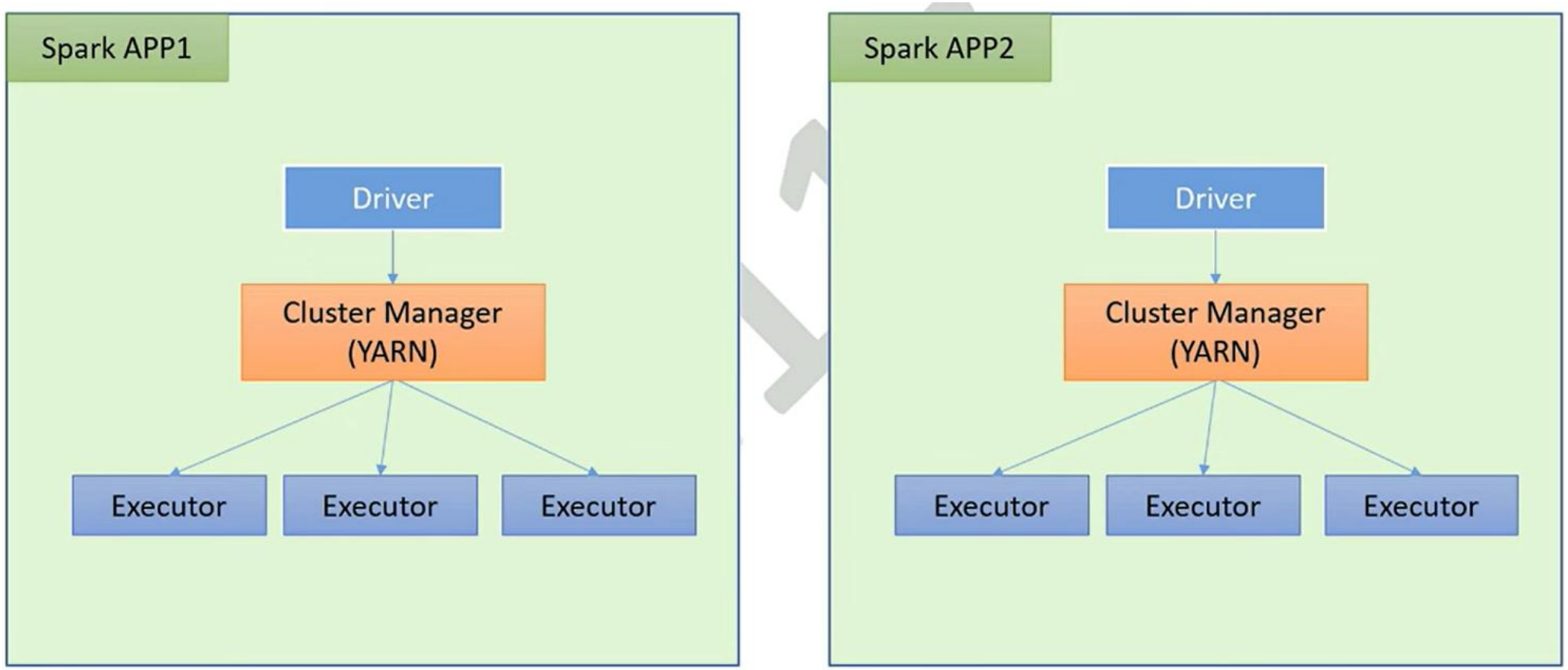
SPARK ARCHITECTURE

- Cluster view
- Application view

Cluster View

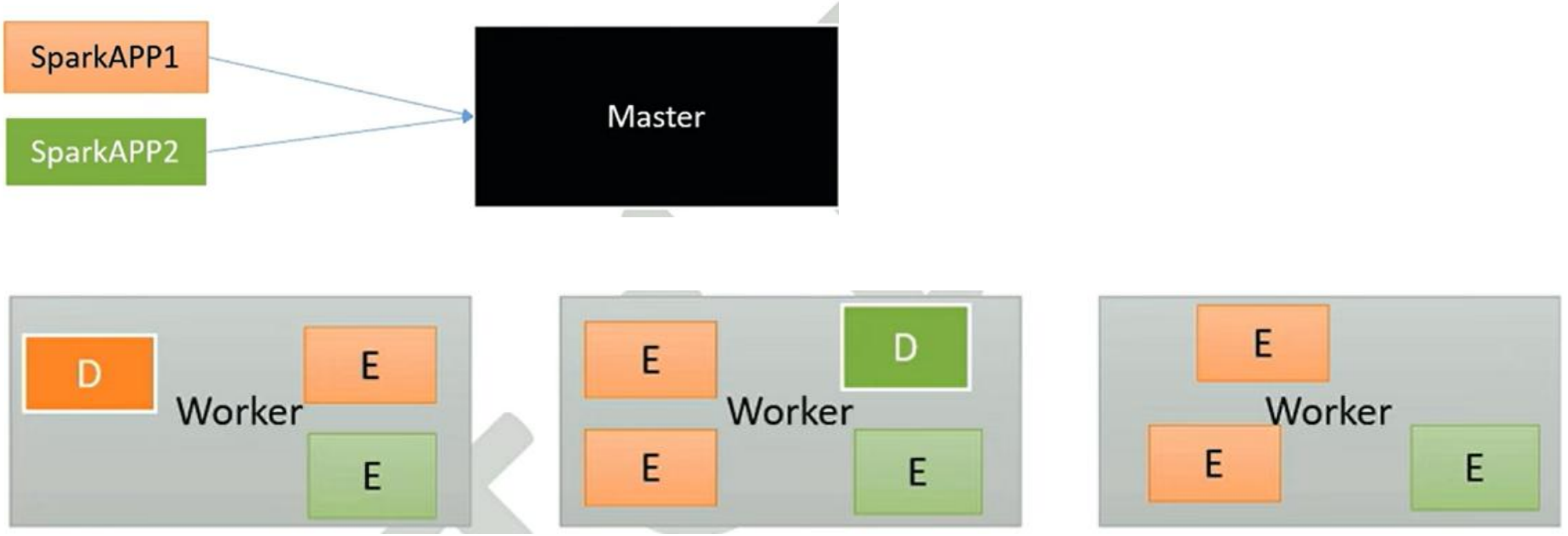


Application view

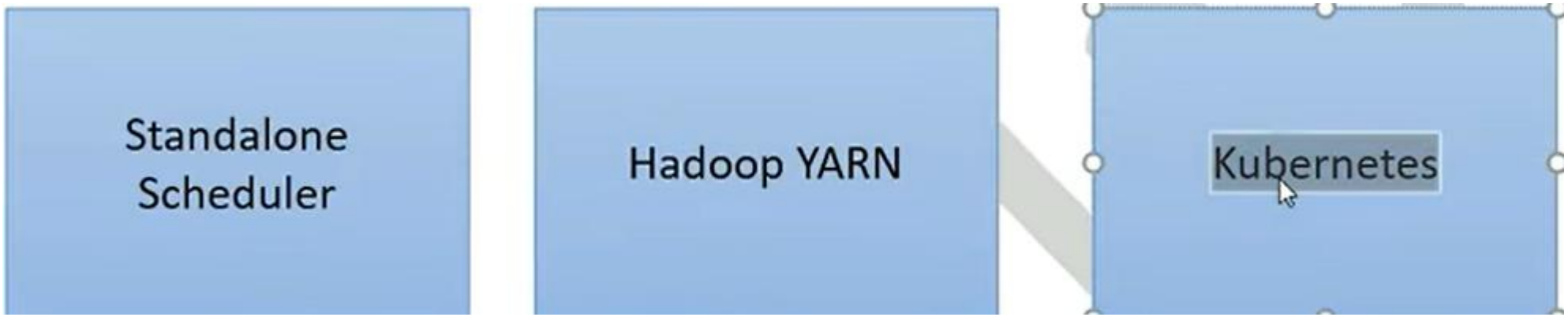


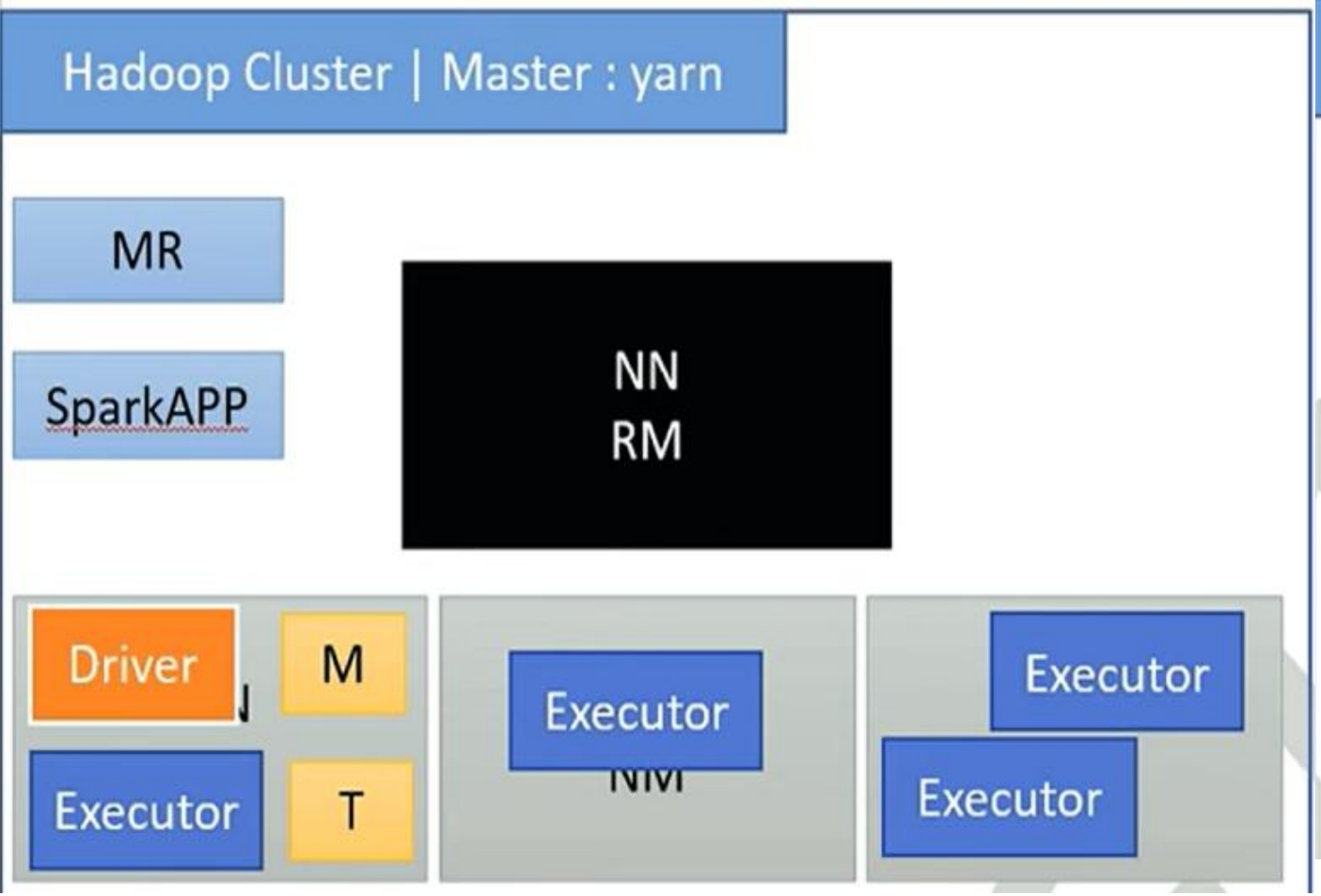
Spark application in spark cluster

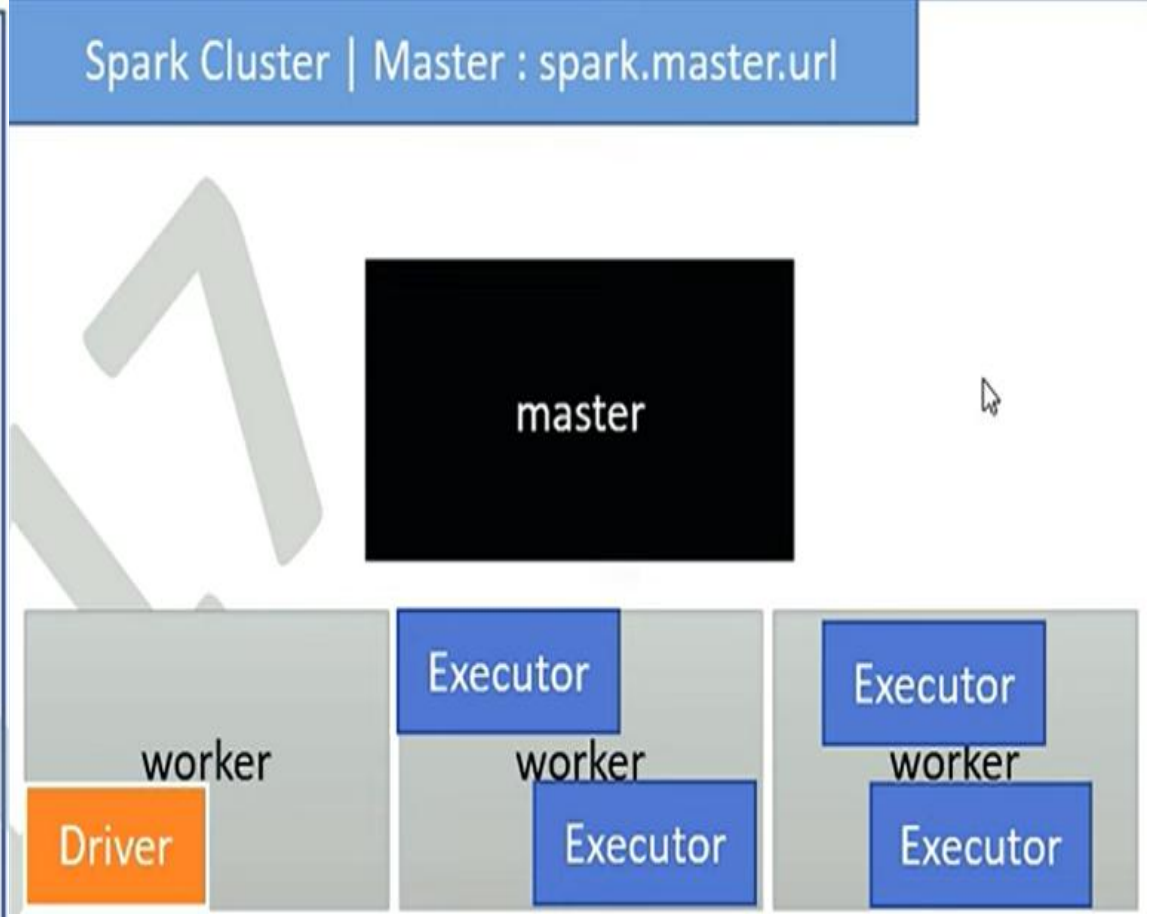
Cluster View + APP View



Spark application cluster option







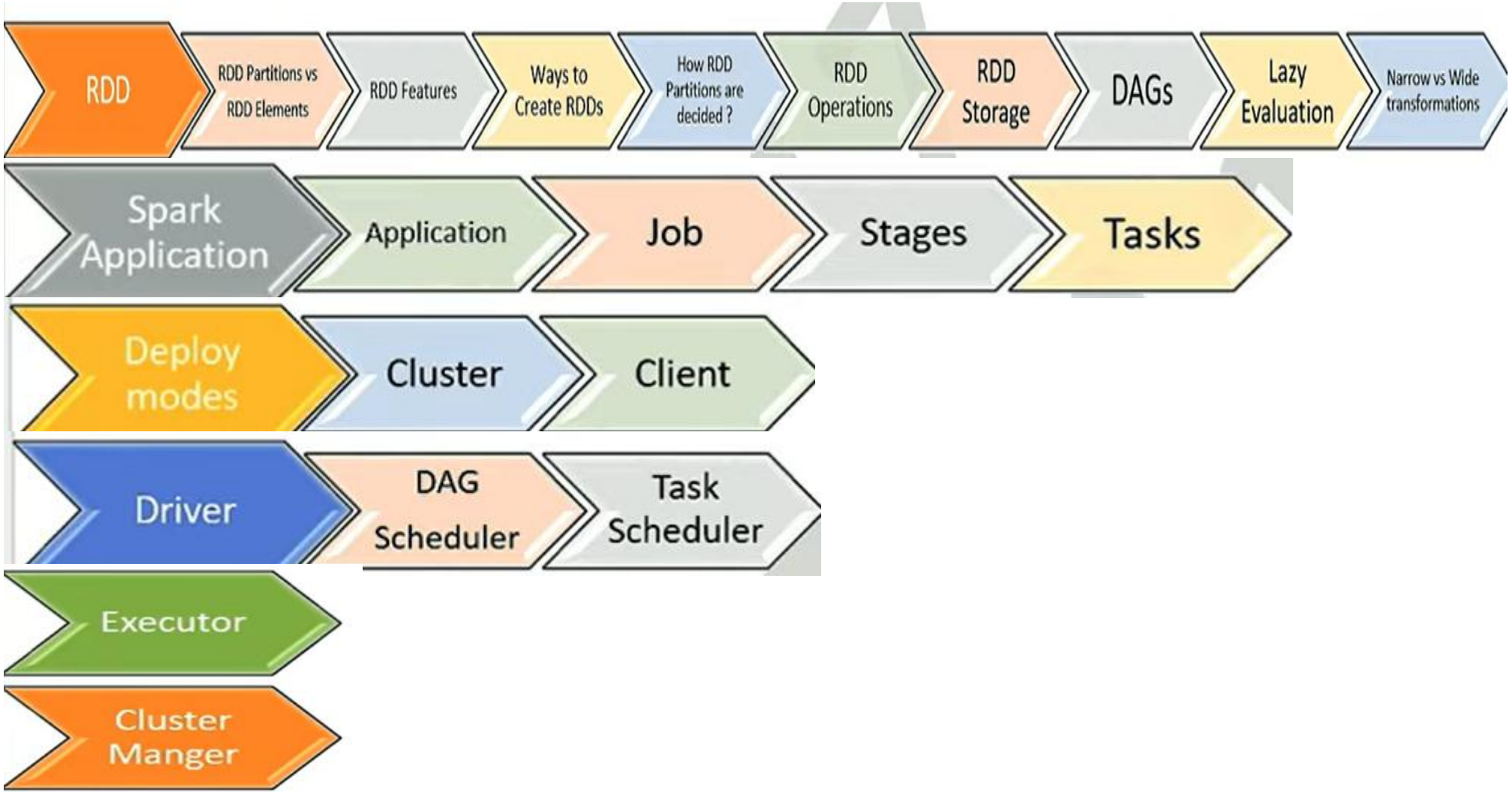
SparkAPP

SparkAPP


Kubernetes
SPARK APP
SPARK APP
SPARK APP
SPARK APP

SPARK ARCHITECTURE

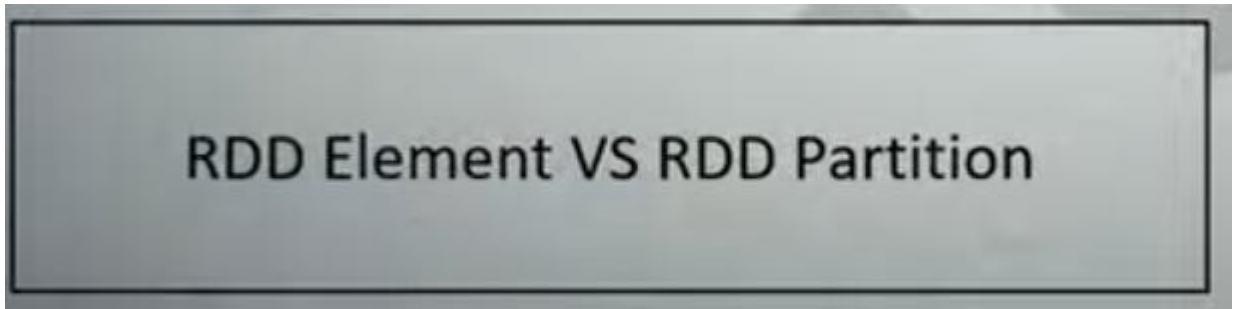
Pre requisites to understand Spark Architecture



RDD Introduction

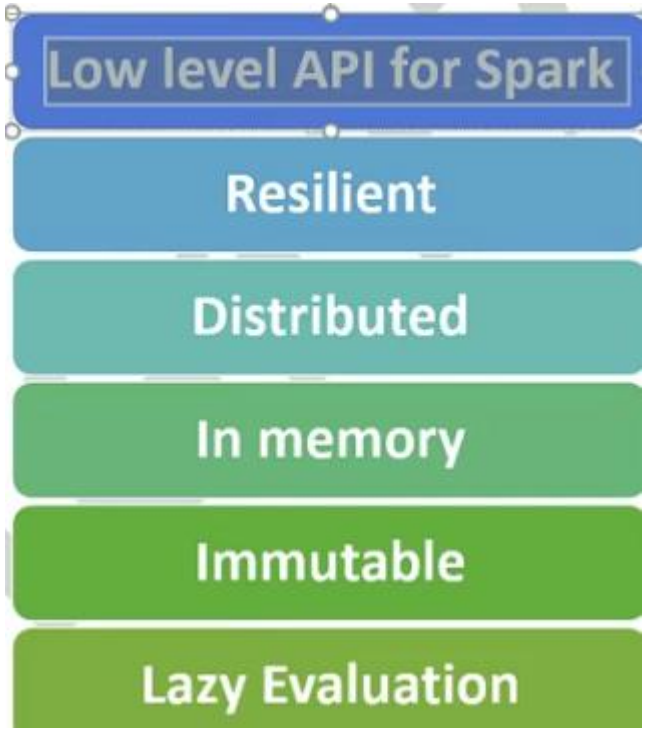
RDD: Resilient Distributed Dataset

- Founation is RDD



If parttions crashes , It heal automatically.

RDD Features



RDD (Resilient Distributed Dataset) in Apache Spark

RDD is the **fundamental data structure** in Apache Spark. It is an **immutable** (read-only) collection of objects that are **distributed** across multiple nodes in a cluster and **processed in parallel**.

Where Does RDD Run?

RDD runs **on the cluster nodes** of a Spark application. Spark operates in a **master-slave architecture**, where:

- The **Driver Program** runs on the master node.
- The **Executors** run on the worker nodes.
- RDDs are distributed across the worker nodes and are processed in parallel.

Who Initiates an RDD?

RDDs are **initiated by the user** through:

1. **Loading data from an external source** (like HDFS, S3, a database, or local files).
2. **Creating RDDs manually** (using `parallelize()` in Spark).
3. **Transforming existing RDDs** using functions like `map()`, `filter()`, `reduceByKey()`, etc.

Where is RDD Stored?

RDDs can be stored in **different storage levels**, such as:

1. **Memory (RAM)** → Default storage in Spark (`MEMORY_ONLY`).
2. **Disk** → If there is not enough memory, Spark stores RDDs on disk (`DISK_ONLY`).
3. **Both Memory and Disk** → Partial storage in memory and rest on disk (`MEMORY_AND_DISK`).
4. **Serialized format** → Compressed storage to save memory (`MEMORY_ONLY_SER`).

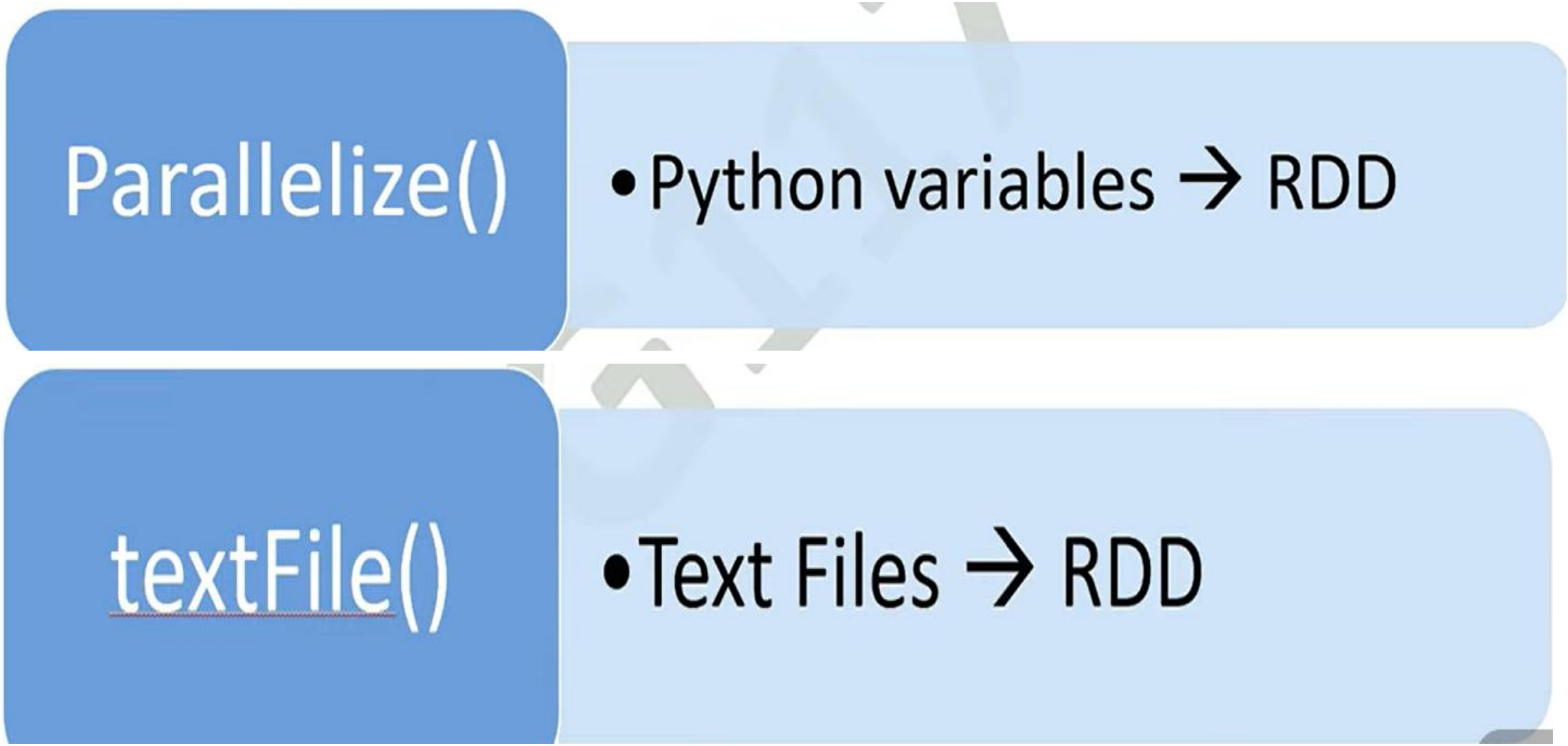
RDD storage is managed by Spark's **caching and persistence mechanisms**.

- from pyspark.sql import SparkSessionspark = SparkSession.builder.appName("test").getOrCreate()sc = spark.sparkContext

- arr=[1,2,3,4,5,6,7,8,9,10]
- arrDD=sc.parallelize(arr)
- type(arr)
- type(arrDD)
- arrDD.getNumPartitions() //we can distribute these chunks on different computers -→ distributed processing
- Arr.getNumPartitions()

- Immutable:
  arrDD.collect()
  arr[1]=50
  arrDD.map(lambda x : x+5).collect() # will not change the result only display. But you can make new arrDD
  arrDD.collect()
- sumRDD=arrDD.map(lambda x : x+5).collect() //make an rdd from existing rdd
- print(sumRDD)
-

Common way to Creat RDD



External source for applications

ON WHICH BASIS NUMBER PARTITIONS FOR RDD ARE CHOSEN ?

How partitions are decided for RDD created using Python variables

Default Parallellism

- MAX(no_of_threads , 2) : MAX(8,2) : 8

DefaultMinPartitions :

- Min(DP, 2) : Min(8,2) : 2

- arr=[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16]
- arrRDD=sc.parallelize(arr)
- arrRDD.getNumPartitions()
- sc.defaultParallelism // it is 2
  - You can change it
    from pyspark.sql import SparkSession
  -
    spark = SparkSession.builder \
  - .master("local[8]") \
  - .appName("test") \
  - .getOrCreate()
- Restrat the session

```
from pyspark.sql import SparkSession

spark = SparkSession.builder \
    .appName("test") \
    .getOrCreate()

sc = spark.sparkContext
print(sc.defaultParallelism)
```

RDD Transformations

- If change the structure of the data is called transformation

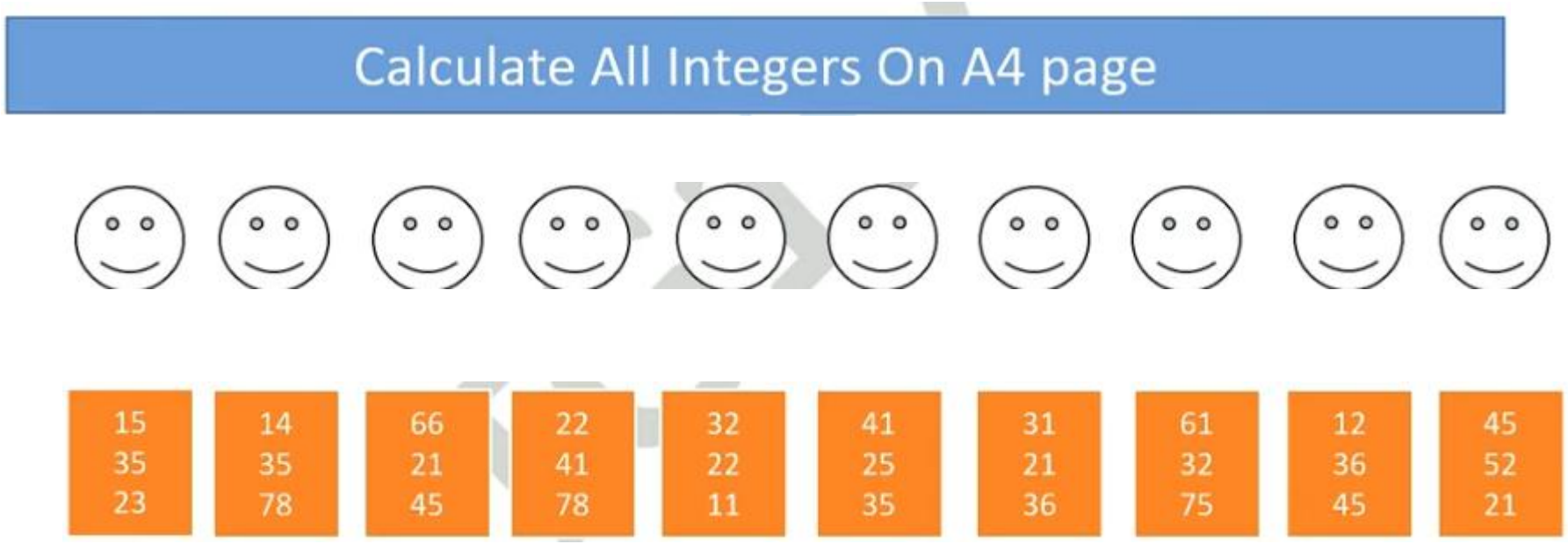
Txt File

- Load txt file
  - dataRDD=sc.textFile(r"file:///j:\sample1\Data8277.csv")
  - dataRDD.getNumPartitions()
  - dataRDD1=dataRDD.repartition(10)
  - dataRDD1.getNumPartitions()
- Lazy Evaluation (Check browser each time)
  - rdd = sc.parallelize([1,2,3,4,5])
  - rdd2 = rdd.map(lambda x: x*2)
  - rdd3 = rdd2.filter(lambda x: x > 5)
  - result = rdd3.collect()
  - print(result)
- Repartion

Types of Transformations

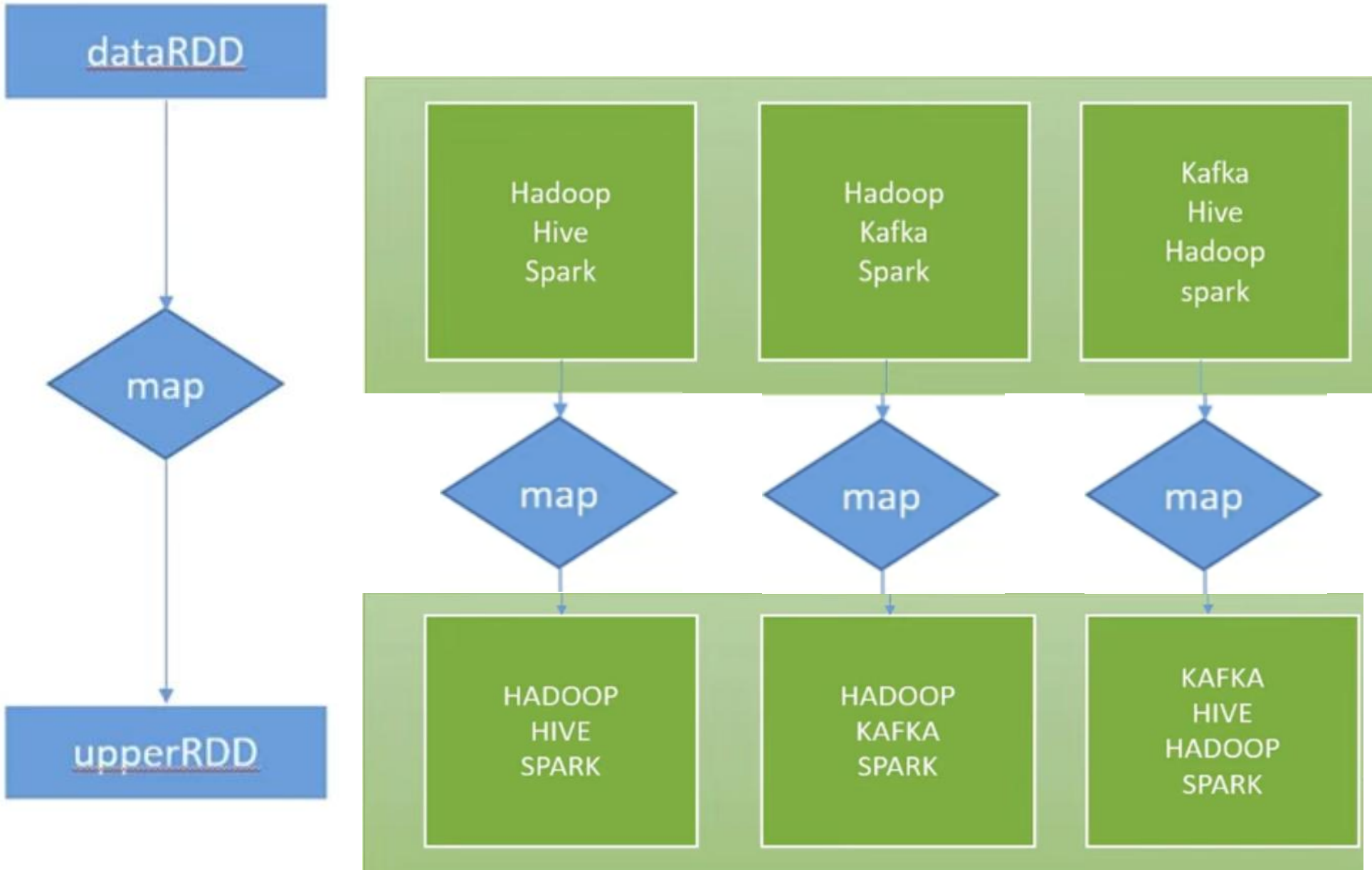
Narrow	• Map • FlatMap • Filter • union
Wide	• Intersection • groupByKey • reduceByKey • Join

Narrow: No Shuffling

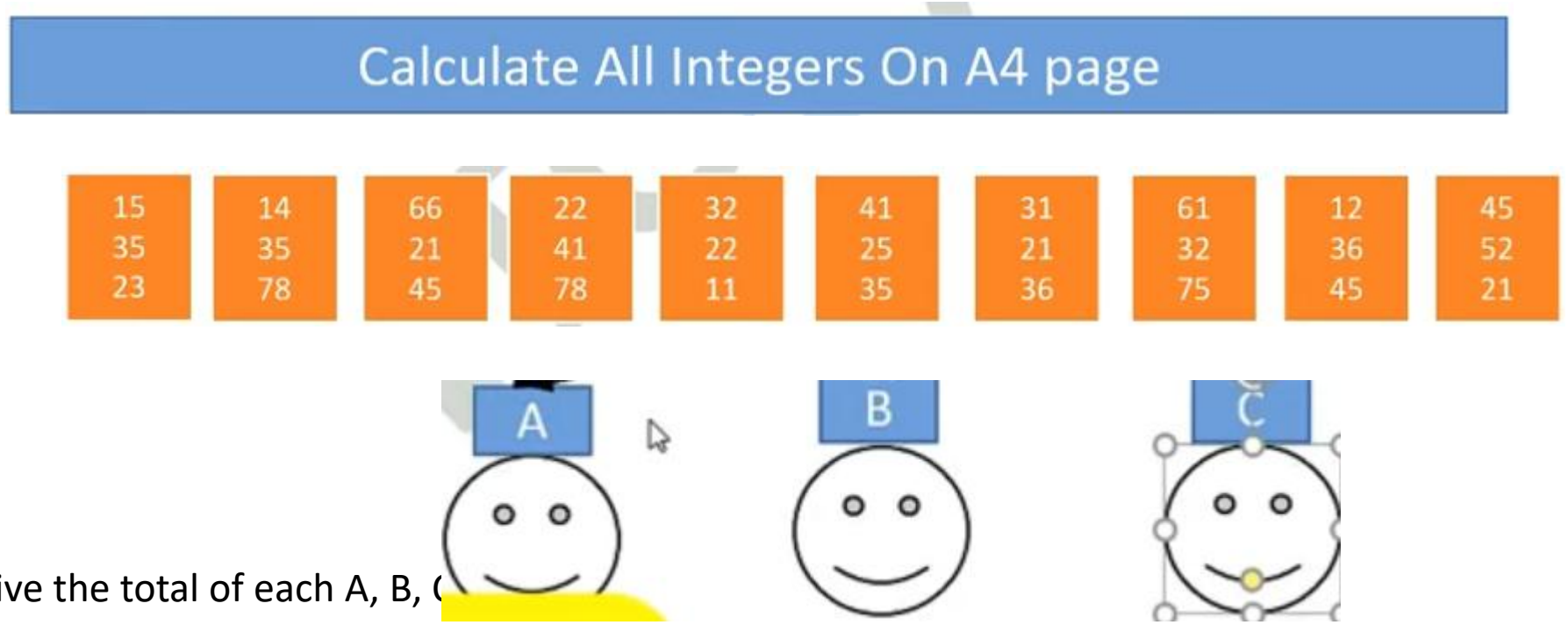


Give the total of each.
No dependency on each other

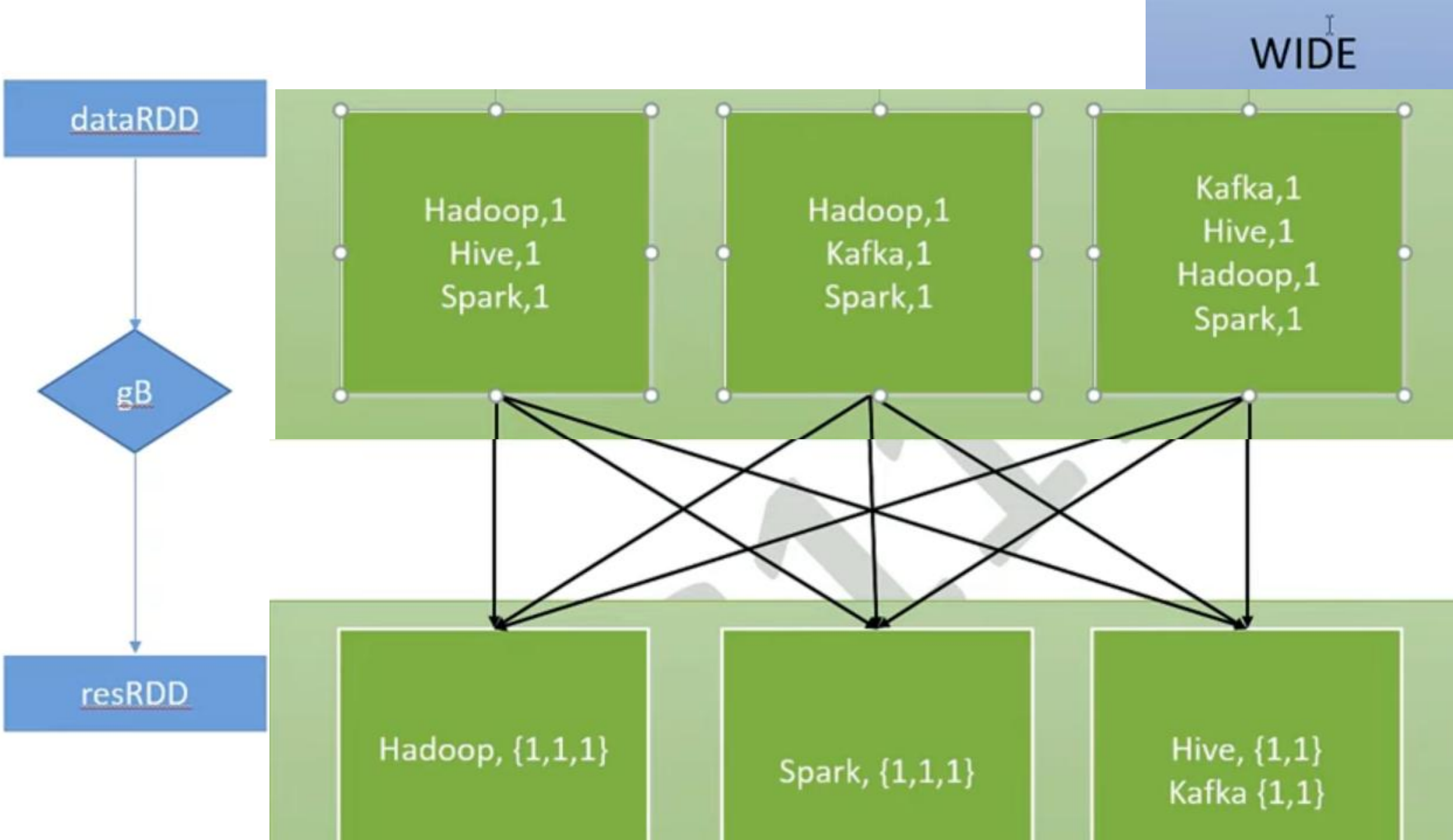
NARROW DEPENDENCY



Wide: Shuffling



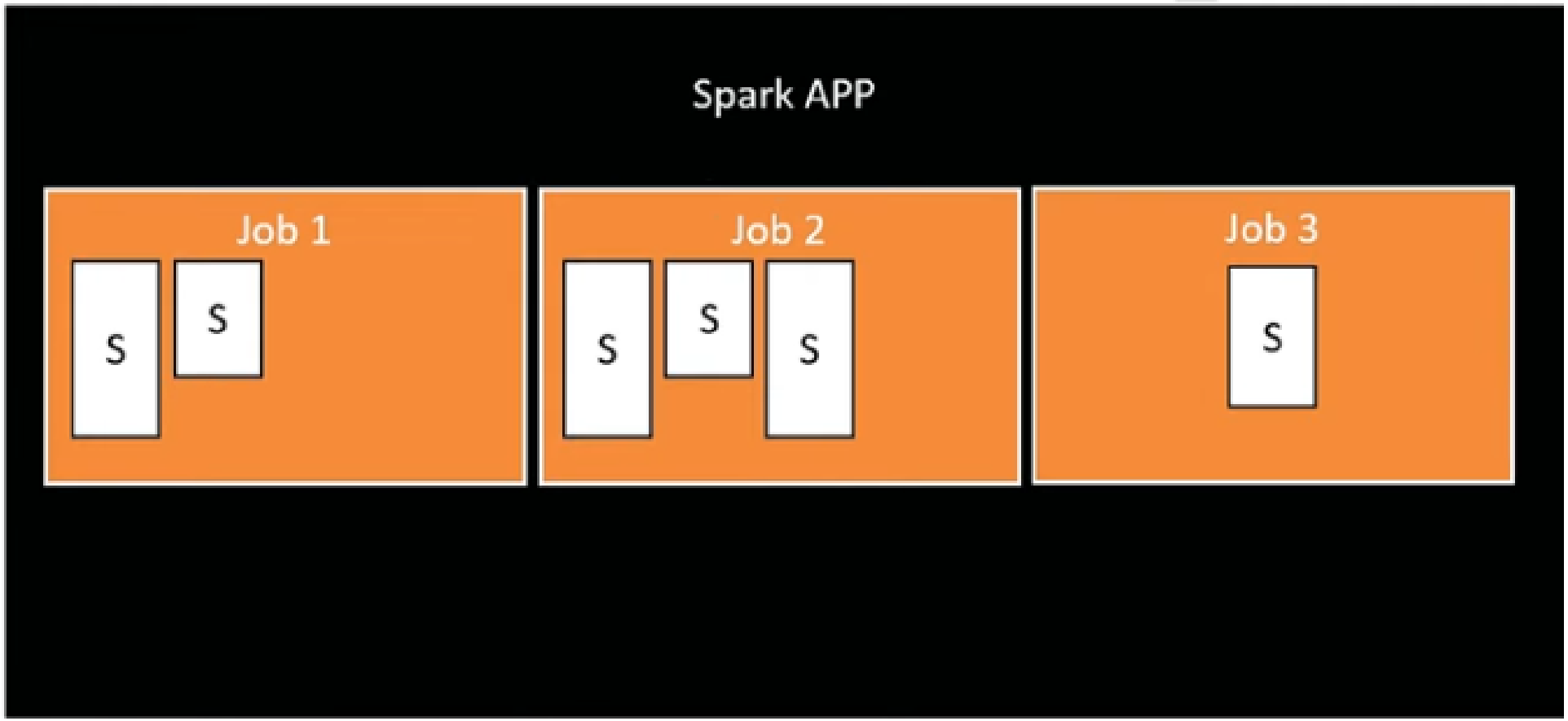
Give the total of each A, B, C


WIDE
dataRDD
gB
resRDD
Hadoop,1
Hive,1
Spark,1
Hadoop,1
Kafka,1
Spark,1
Kafka,1
Hive,1
Hadoop,1
Spark,1
Hadoop, {1,1,1}
Spark, {1,1,1}
Hive, {1,1}
Kafka {1,1}

- Data is not stored physically on machine. On each step until collect() ore reduceByKey()

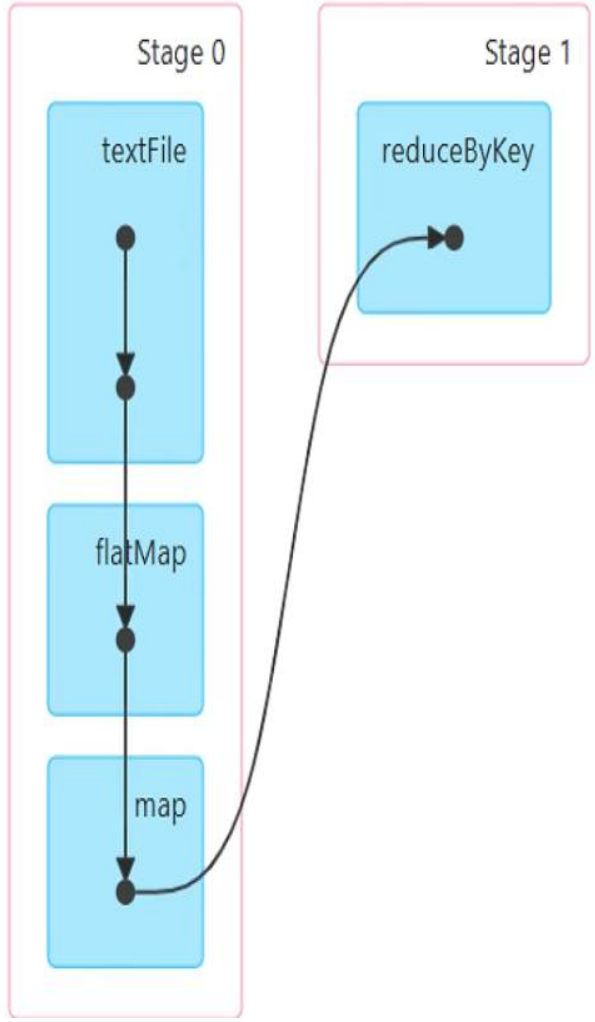
Spark Application

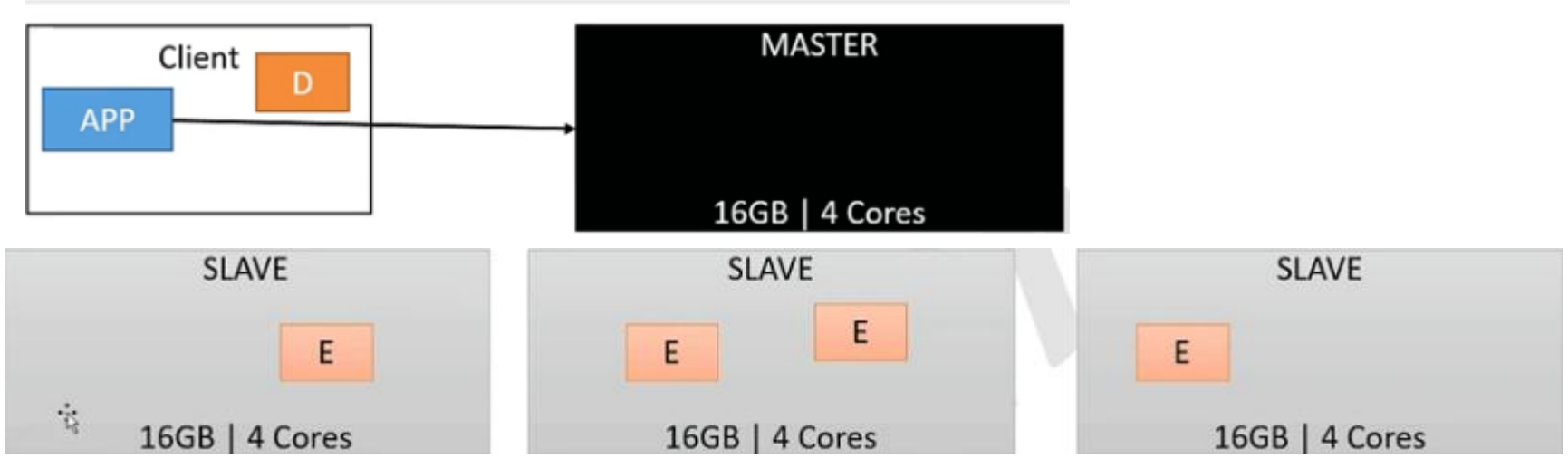
- Pyspark shell is an application
- Jupyter notebook, vs code , pycharm
- Notebook enviornement

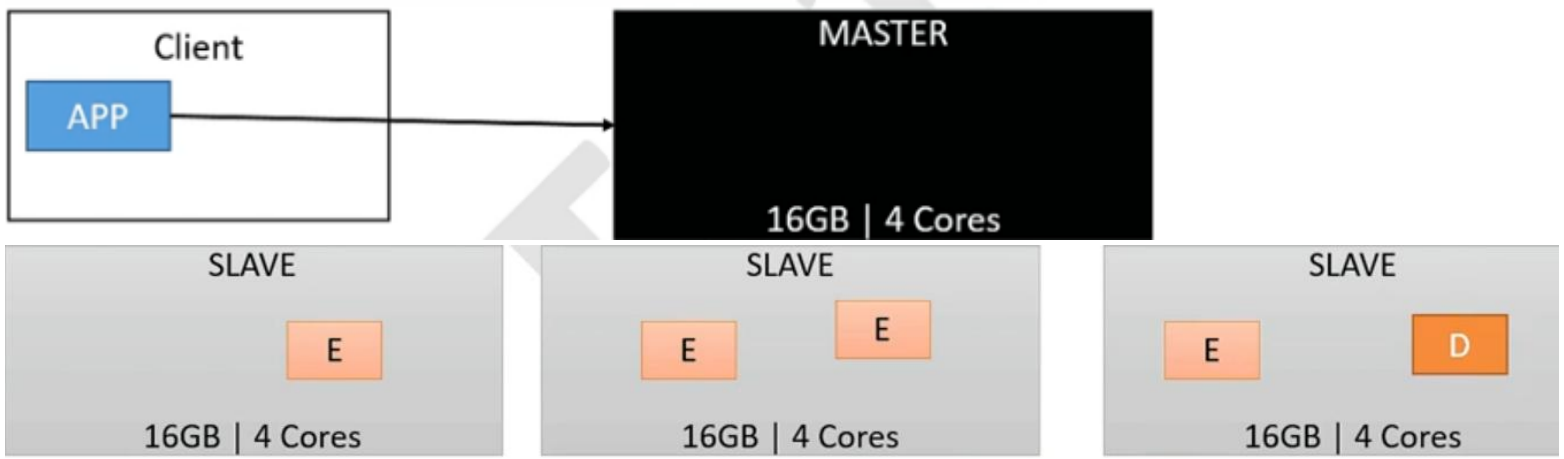

Spark APP
Job 1
S
S
Job 2
S
S
S
Job 3
S

What is job

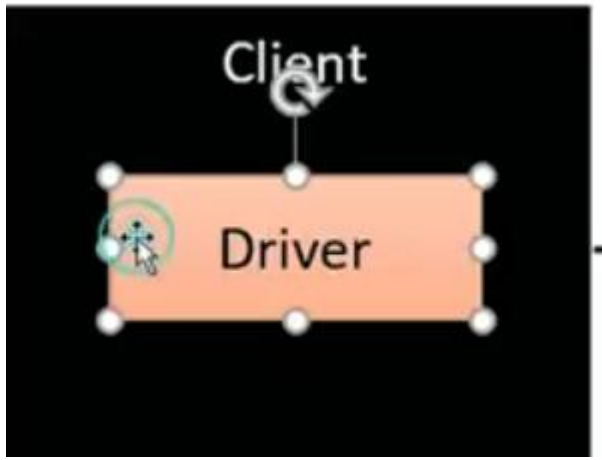
- A job has multiple stages.
- When a wide transformation is encounter stage will change.




Client
D
APP
MASTER
16GB | 4 Cores
SLAVE
E
16GB | 4 Cores
SLAVE
E
E
16GB | 4 Cores
SLAVE
E
16GB | 4 Cores


Client
APP
MASTER
16GB | 4 Cores
SLAVE
E
16GB | 4 Cores
SLAVE
E
E
16GB | 4 Cores
SLAVE
E
D
16GB | 4 Cores

Client Mode



Cluster Manager (YARN)

Executor

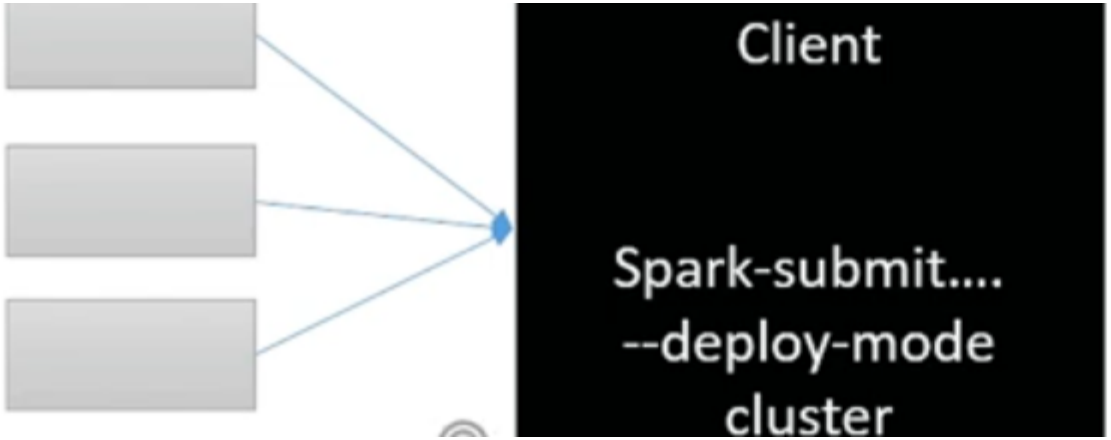
Executor

Executor

Executor

Phase : Development, Test
Using :
PySpark shell
Notebook

Cluster Mode



Cluster Manager

Executor

Executor

Executor

Driver

Executor

Phase : Production

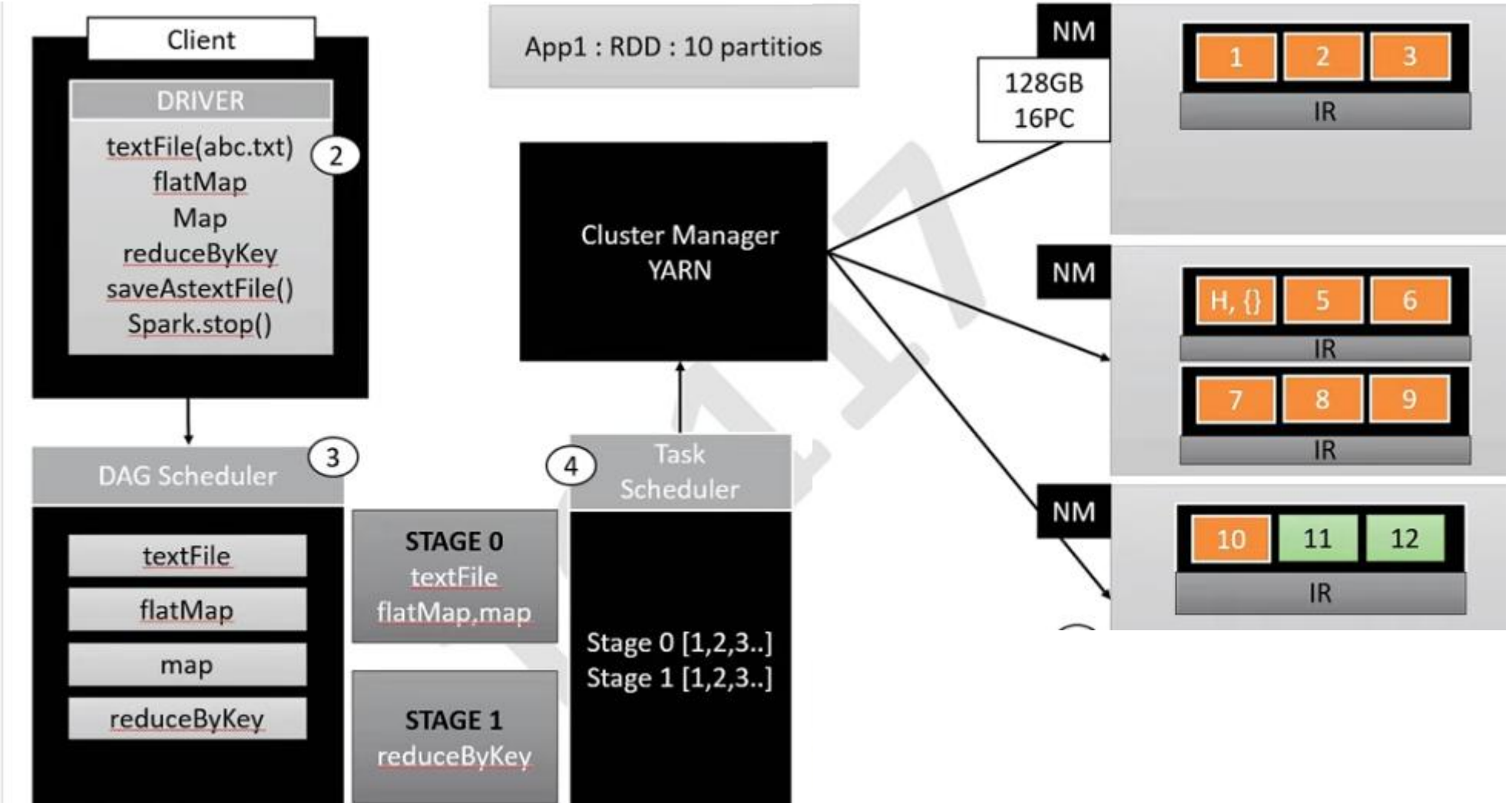
Using :

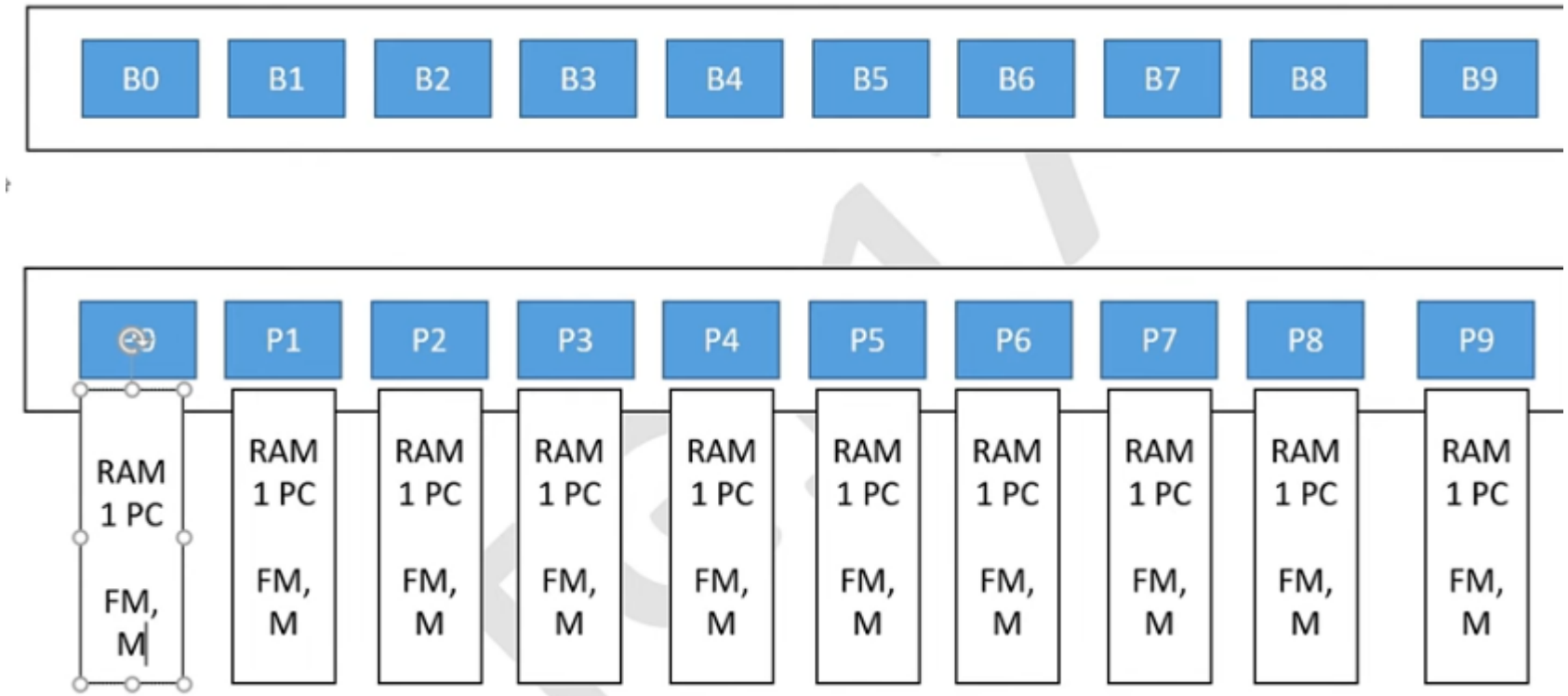
Spark-submit Deployment

Why :

1. Communication between Driver and executors becomes easy
2. Fault tolerance : If driver fails CM can handle it

Client Mode Deployment




B0
B1
B2
B3
B4
B5
B6
B7
B8
B9
P1
P2
P3
P4
P5
P6
P7
P8
P9
RAM
1 PC
FM,
M

Driver Responsibilities

1. **Job Scheduling** - Breaks user program into tasks and schedules them on executors
2. **DAG Creation** - Converts RDD transformations into a Directed Acyclic Graph (DAG) of stages
3. **Stage Division** - Splits the DAG into stages at shuffle boundaries (wide transformations)
4. **Task Assignment** - Assigns tasks to executors based on data locality
5. **Resource Management** - Communicates with cluster manager to acquire/release resources
6. **SparkContext Management** - Creates and maintains the SparkContext (entry point)
7. **Metadata Tracking** - Tracks RDD lineage, partitions, and data locations
8. **Result Collection** - Collects results from executors (e.g., during collect() action)
9. **Monitoring & Logging** - Maintains Spark UI for monitoring job progress and debugging
10. **Failure Recovery** - Handles executor failures and task re-scheduling
11. **Broadcast Variables** - Distributes broadcast variables to all executors
12. **Accumulator Management** - Collects accumulator values from executors

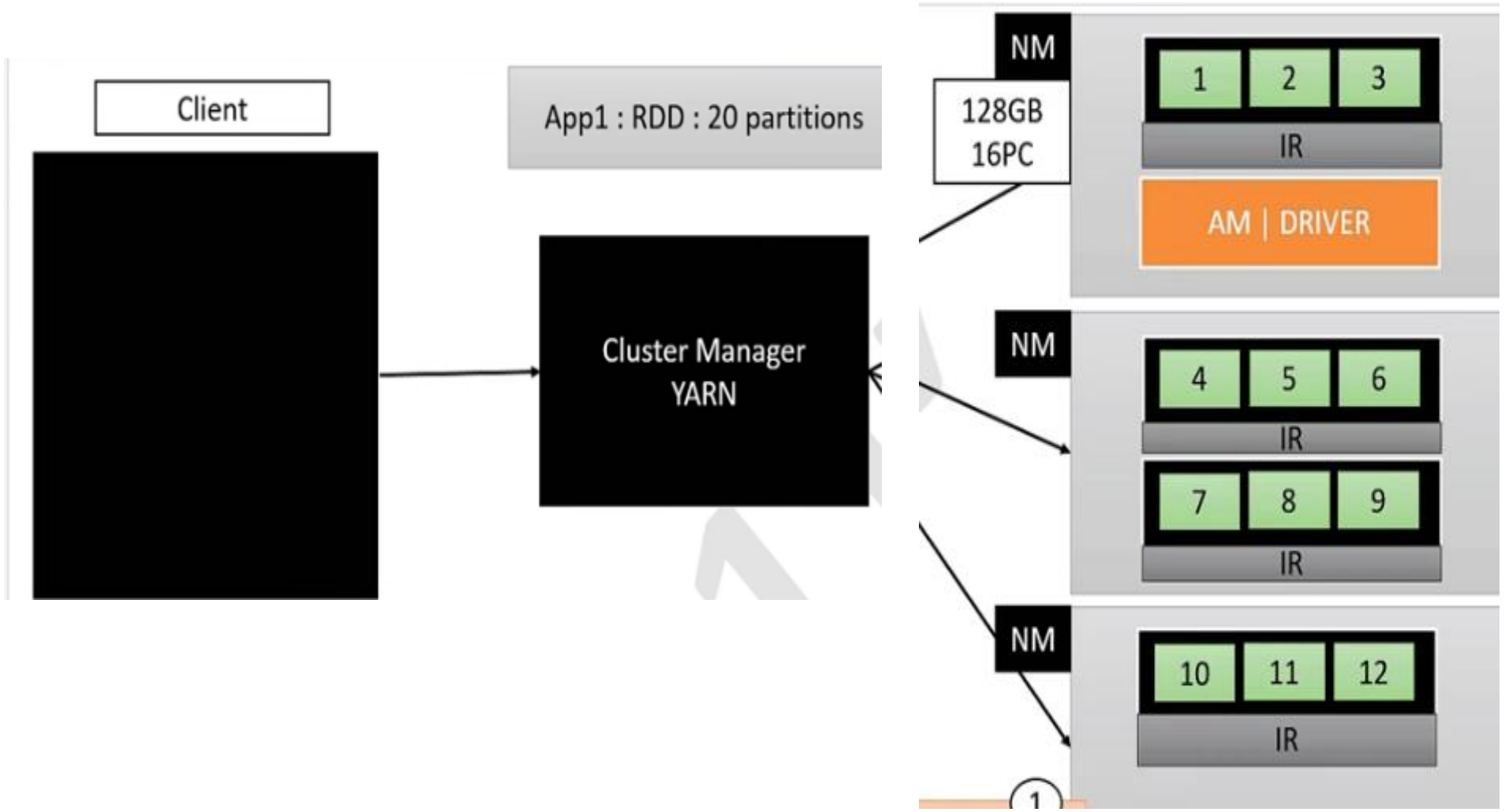
Dirver Notes

- Driver and executors are exclusive to spark application
- Multiple executors of same application can run inside one worker machine
- Multiple executors of different applications can run inside one worker machine
- executor handles multiple parallel tasks of same stage and same application
- At any given time, tasks of only one stage will be executed
- Tasks of same stage can run in parallel
- Stages are executed sequentially

YARN Runs on Multiple Machines:

- **YARN has two main components that run on different machines:**
- **1. ResourceManager (Master Node)**
- Runs on **one master machine** (dedicated server)
- Manages cluster resources globally
- Schedules applications across the cluster
- Monitors NodeManagers
- **Only 1 ResourceManager** per cluster (can have standby for HA)
- **2. NodeManager (Worker Nodes)**
- Runs on **every worker machine/slave node** in the cluster
- Manages resources (CPU, memory) on that specific machine
- Launches and monitors containers

Cluster Mode Deployment



Driver

- Master process for spark Application
- Allocates tasks and monitors them
- Tracking Tasks progress
- (In some cases) Results are travelled back to Driver

Cluster Manager

- Allocate and Deallocate Resources for spark Application
- Pluggable component
- Handling Container level Failures
- CMs : YARN, Mesos, Kubernetes, Standalone Scheduler

Broad Casting Variables

Distributed Shared variables

- Broadcasting
- Accumulator

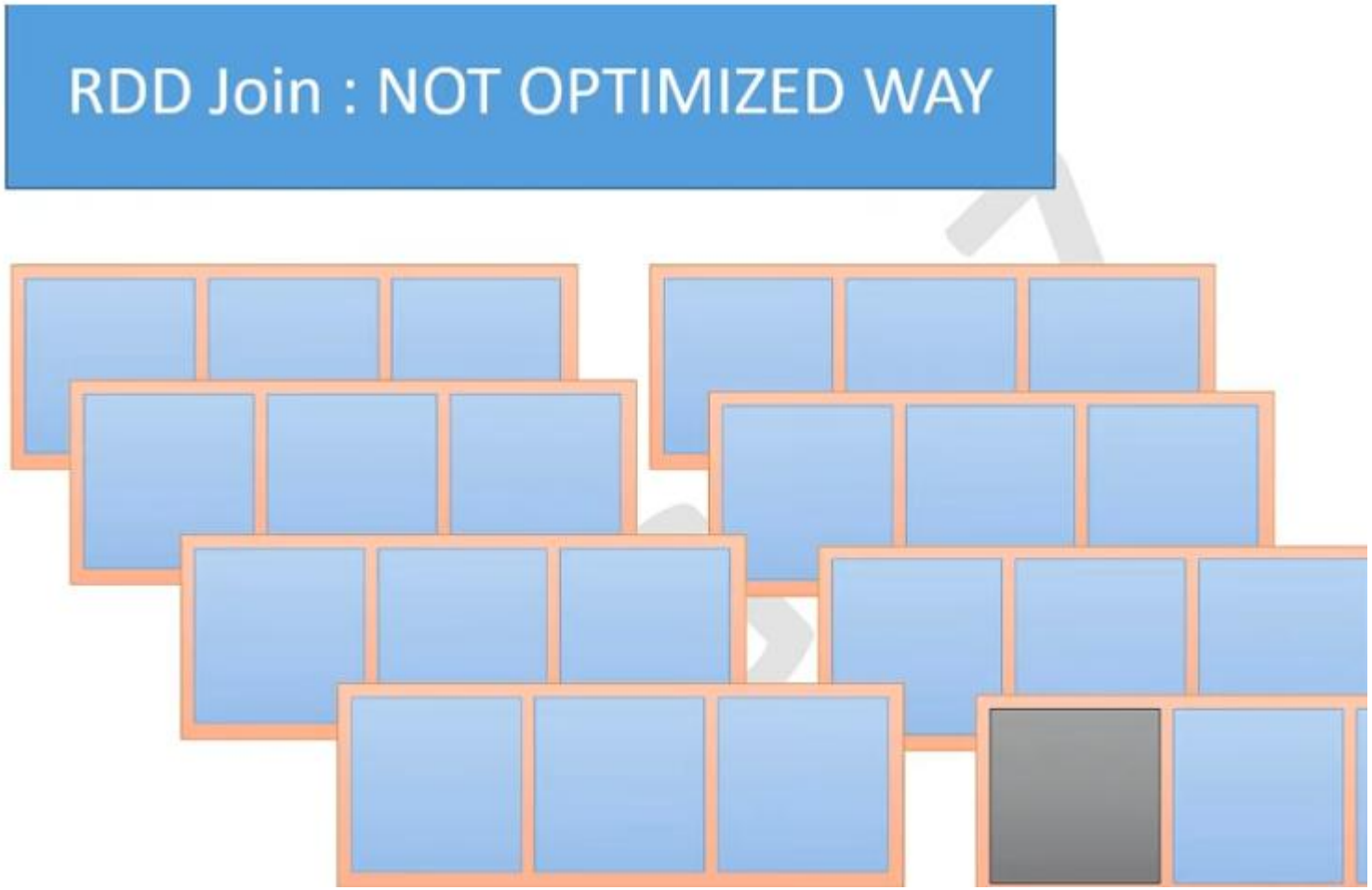
Low Level API

- RDD
- Broadcasting Variables
- Accumulator

High Level API

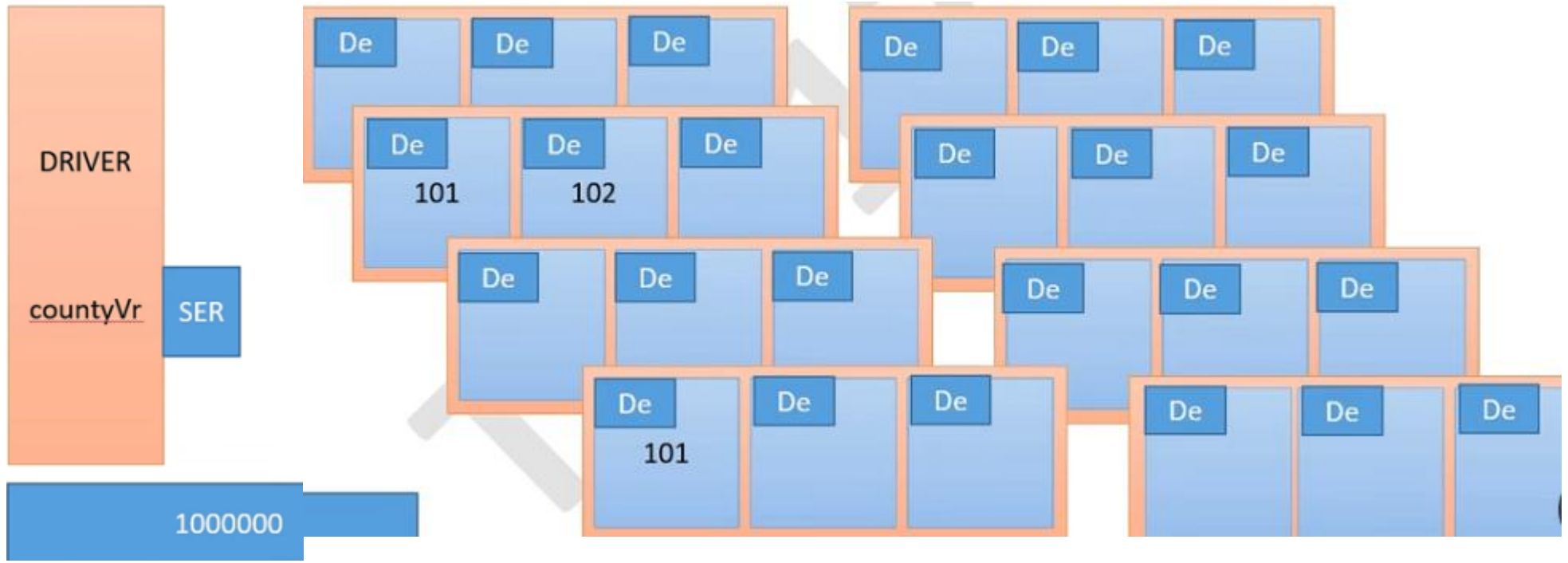
- DataFrame
- Datasets

One Driver and multiple executers

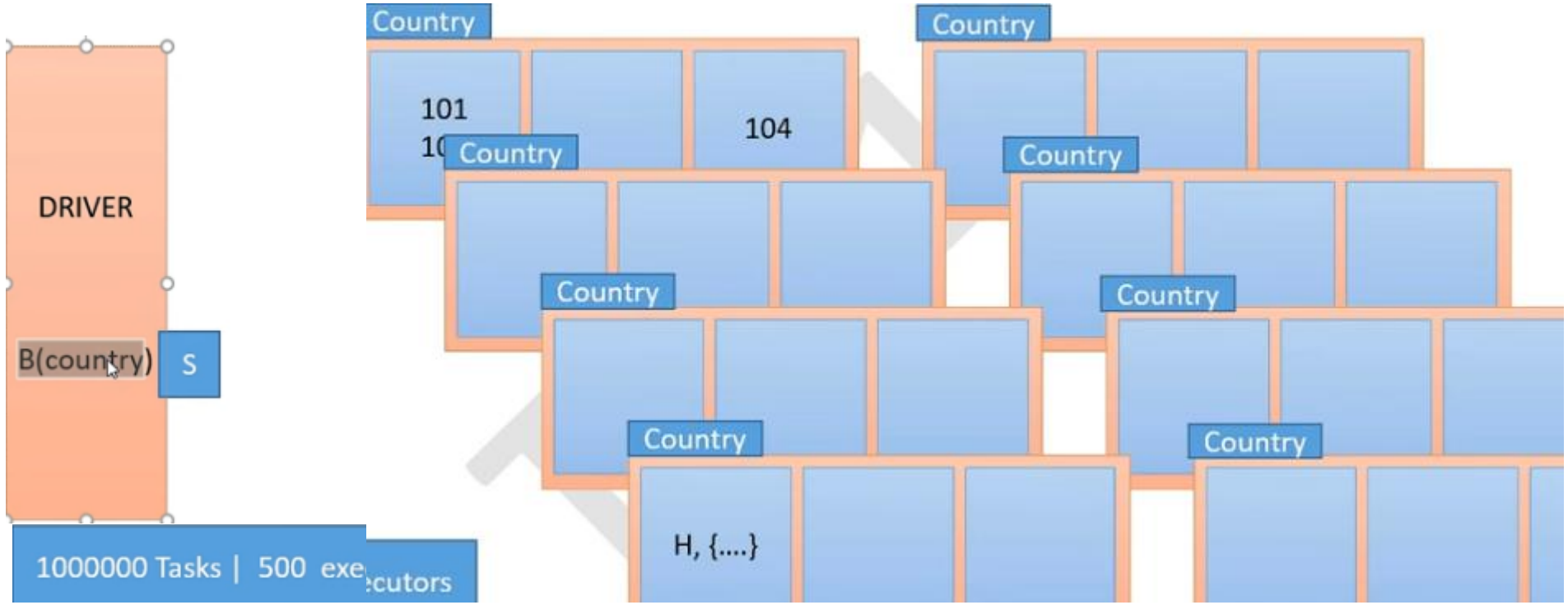


800 + 2

USING Variable: NOT OPTIMIZED WAY

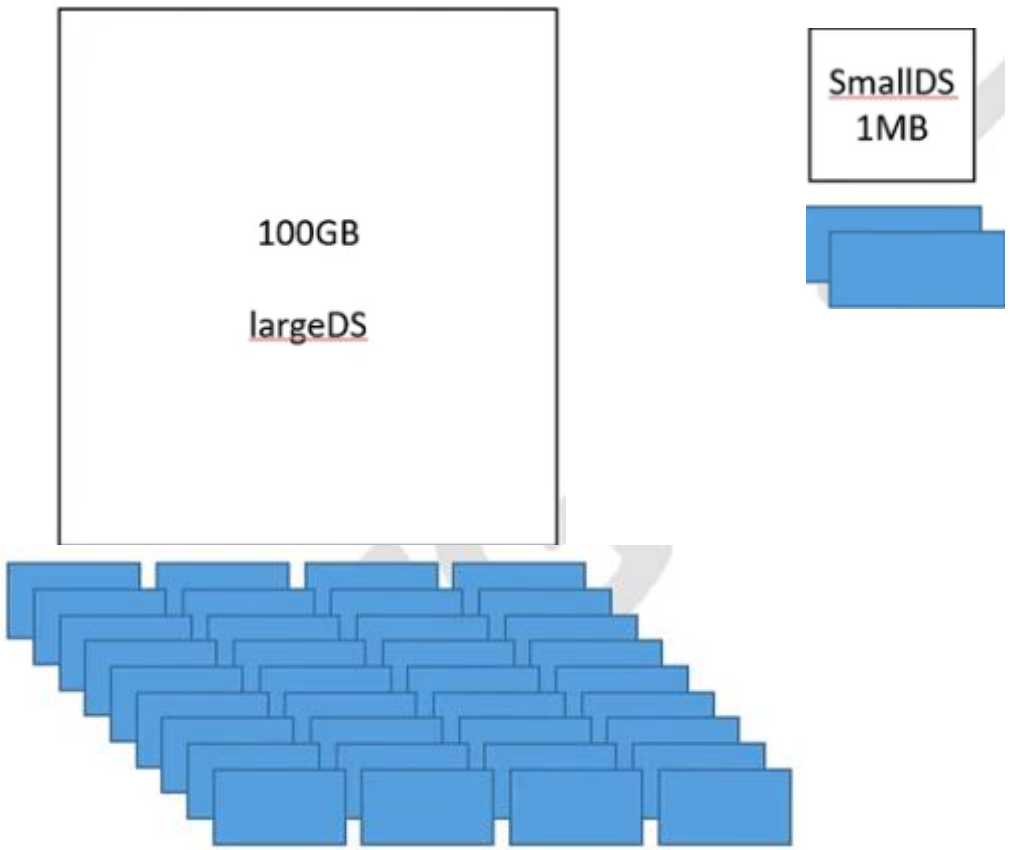


Broadcast Variable : OPTIMIZED WAY



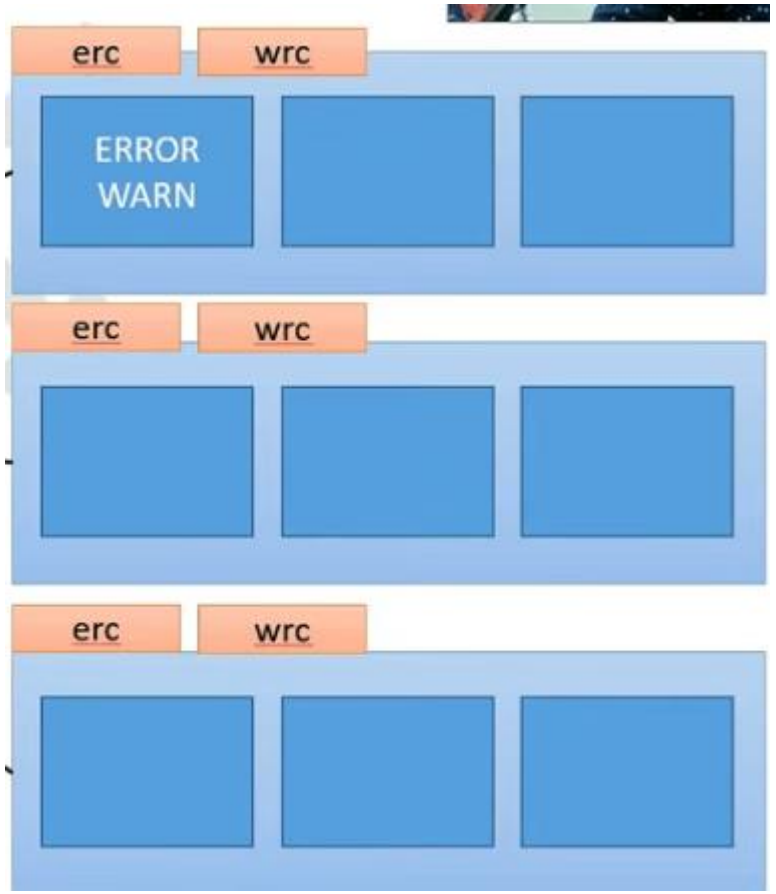
Broadcast Variables

- Read Only variables
- Immutable Variables
- Use
  - When python variables are required in RDD code
  - When we need to avoid Join operation in scenario where one file is large and other one is small
- No hard limit on size of broadcast variable
- However to select a variable/file as broadcast variable we need to consider
  - Size of Data
  - Cluster configuration (App : Driver memory and Num exec, Executor Memory)
  - Testing
- Bv.unpersist() : This method will remove BV from Driver and Executor RAM
- LRU Cache



RDD join

Accumulator

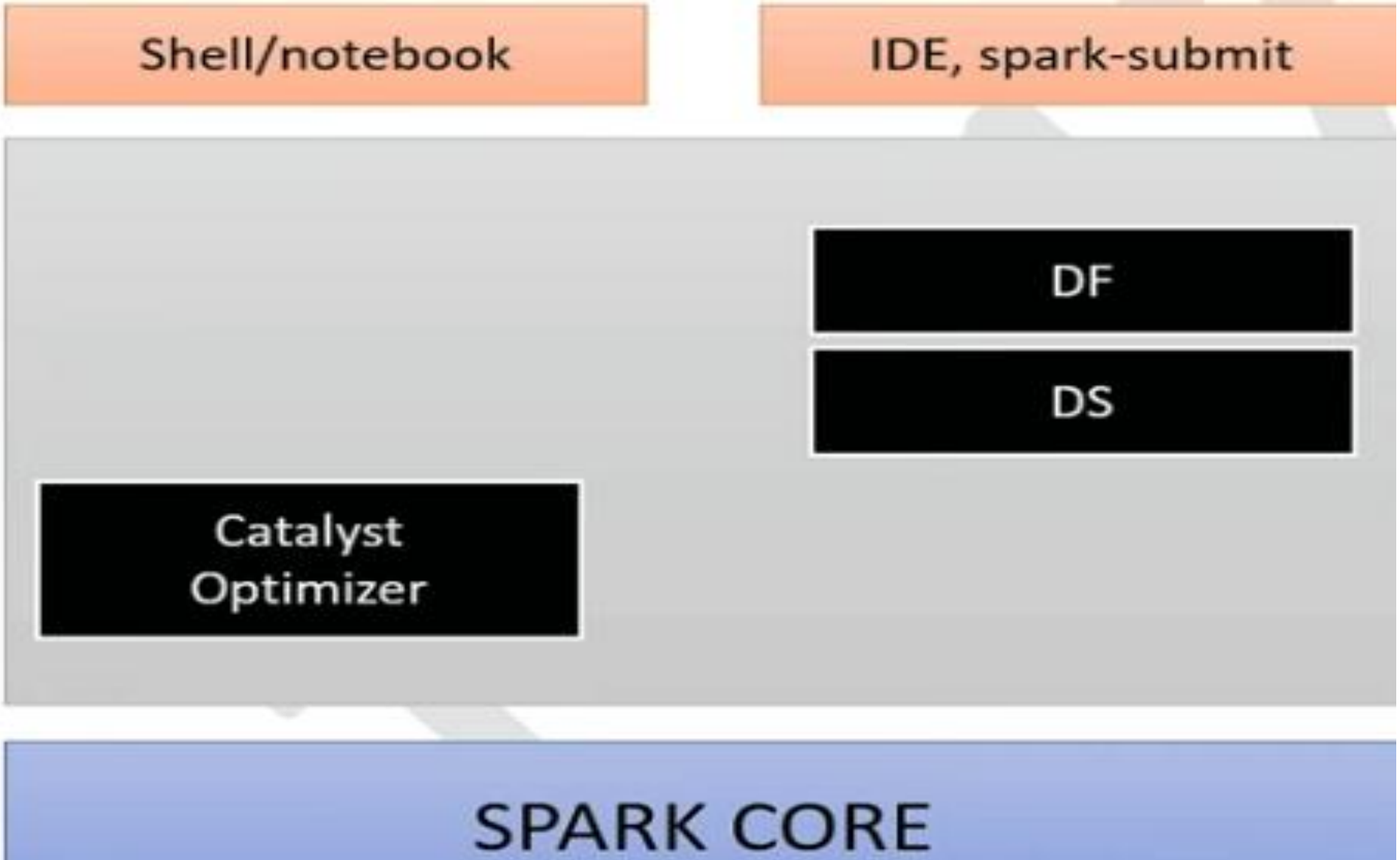


Accumulator

- Count operations
- Mutable
- Write Only
- Eliminates wide transformation for count related operations

- RDD VS DataFrame

SParkSQl Architecture



- Structual Transformation
- Literal SQL

```
datardd=sc.textFile("cr.txt")
datardd.take(2)
datardd.first()
header=datardd.first()
flrdd=datardd.filter(lambda
x: x!=header)
flrdd.take(3)
splitrdd=flrdd.map(lambda x:
x.split(","))
kvrdd.take(3)
groupbykdd=kvrdd.groupByK
ey().map(lambda x: (x[0],
```

```
dataDF =
spark.read.option("header",
"true").option("inferSchema
","true").csv(r"cr.txt")

dataDF.groupBy("cname").su
m("revenue").show()
```

- dataDF.createOrReplaceTempView("cr_view")
- sc.setLogLevel("ERROR")
- spark.sql("select cname, sum(revenue) from cr_view group by cname").show()

- dataDF.printSchema()
- dataDF.rdd.collect()

DataFrames

- **Resilient**
- **Distributed**
- **In Memory**
- **Lazy Evaluation**
- **Structured**
- **Has Schema**

empDF

org.apache.spark.sql.Row	Emp(1, Martica, Anglin, manglin0@stanford.edu, Human Resources, 9) |
org.apache.spark.sql.Row	Emp(2, Jaime, Wikey, jwikey1@clickbank.net, Human Resources, 13) |
org.apache.spark.sql.Row	Emp(3, Perice, Papes, ppapes2@trellian.com, Engineering, 12) |
org.apache.spark.sql.Row	Emp(4, Adrianne, Blissitt, ablissitt3@ebay.com, Legal, 7) |

DF

- Every row of DF is Spark's Row Object.
- DF does not have compile time type safety
- If structured transformation is applied on DF, new DF will be returned
- DF has better performance than DS

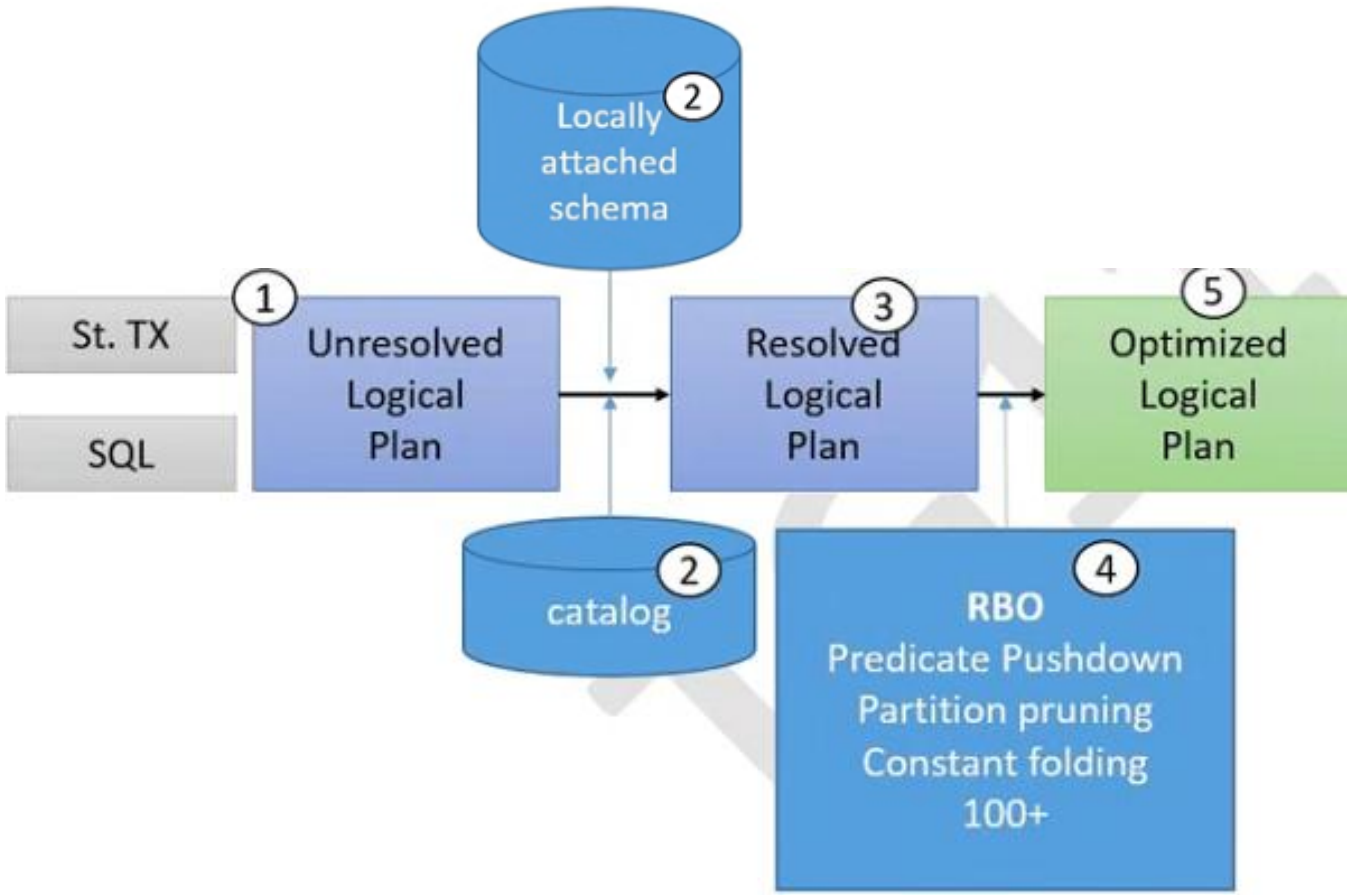
DS

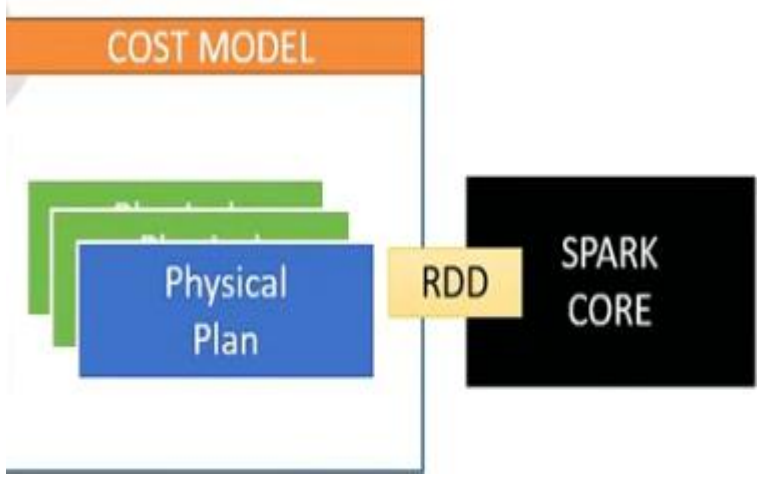
- Every row in DS is an object of class which we have defined for it.
- DS has compile time type safety with RDD transformations
- If structured transformation is applied on DS, new DF will be returned
- To maintain type safety, perform RDD transformations on DS.
- encoders

- DF : Python, R, java, scala
- DS : scala, java

Catalyst optimzer

- Refine the process of exectution on dataframe
- Work for dataframes and datasets not RDD
- It will come in , when you process structured view


Locally attached schema
2
St. TX
SQL
1
Unresolved Logical Plan
3
Resolved Logical Plan
5
Optimized Logical Plan
2
catalog
4
RBO
Predicate Pushdown
Partition pruning
Constant folding
100+


COST MODEL
Physical Plan
RDD
SPARK CORE

Unresolved Logical Plan

```
crDF = spark.read.option("inferSchema","true").option("header","true").csv(r"cr.txt")

crDF.printSchema()



crDF.groupBy("lahore").sum("gcu").show()
```

```
AnalysisException: [UNRESOLVED_COLUMN.WITH_SUGGESTION] A column, variable, or function parameter with nam
e `gcu` cannot be resolved. Did you mean one of the following? [`cid`, `cname`, `revenue`]. SQLSTATE: 427
```

- crDF.createOrReplaceTempView("cr_view")
- sc.setLogLevel("ERROR")
- spark.sql("select gcu, sum(lahore) from cr_view group by ")

```
AnalysisException: [UNRESOLVED_COLUMN.WITH_SUGGESTION] A column, variable, or function parameter with nam
e `gcu` cannot be resolved. Did you mean one of the following? [`cid`, `cname`, `revenue`]. SQLSTATE: 427
```

```
crDF.groupBy("cname").sum("revenue").where(crDF.cname != "A").explain()
```

```
== Physical Plan ==
AdaptiveSparkPlan isFinalPlan=false
+- HashAggregate(keys=[cname#75], functions=[sum(revenue#76)])
   +- Exchange hashpartitioning(cname#75, 200), ENSURE_REQUIREMENTS, [plan_id=244]
      +- HashAggregate(keys=[cname#75], functions=[partial_sum(revenue#76)])
         +- Filter (isnotnull(cname#75) AND NOT (cname#75 = A))
            +- FileScan csv [cname#75,revenue#76] Batched: false, DataFilters: [isnotnull(cname#75), NOT
(cname#75 = A)], Format: CSV, Location: InMemoryFileIndex(1 paths)[file:/C:/Users/Dr.Sahib/spark/cr.txt],
PartitionFilters: [], PushedFilters: [IsNotNull(cname), Not(EqualTo(cname,A))], ReadSchema: struct<cnam
e:string,revenue:int>
```

```
crDF.groupBy("cname").sum("revenue").where(crDF.cname != "A").explain(extended=True)
```

```
== Parsed Logical Plan ==
'Filter '`!`('`=`(cname#57, A))
+- Aggregate [cname#57], [cname#57, sum(revenue#58) AS sum(revenue)#107L]
   +- Relation [cid#56,cname#57,revenue#58] csv

== Analyzed Logical Plan ==
cname: string, sum(revenue): bigint
Filter NOT (cname#57 = A)
+- Aggregate [cname#57], [cname#57, sum(revenue#58) AS sum(revenue)#107L]
   +- Relation [cid#56,cname#57,revenue#58] csv

== Optimized Logical Plan ==
Aggregate [cname#57], [cname#57, sum(revenue#58) AS sum(revenue)#107L]
+- Project [cname#57, revenue#58]
   +- Filter (isnotnull(cname#57) AND NOT (cname#57 = A))
      +- Relation [cid#56,cname#57,revenue#58] csv

== Physical Plan ==
```

CATALYST OPTIMIZER

The Catalyst Optimizer is a query optimization engine in Apache Spark. It improves the efficiency of queries by transforming and optimizing execution plans.

Key Steps in Catalyst Optimizer:

1. **Logical Plan Creation:**
   - Spark creates an unresolved logical plan based on your DataFrame/Dataset code.
   - It contains column names, table references, and expressions but isn't validated yet.

2. **Logical Plan Resolution:**
   - Spark validates and resolves the logical plan using:
    - Schema information.
    - Catalog (if tables/views are registered).
    - Produces a resolved logical plan.

3. **Rule-Based Optimization (RBO):**
   - Applies predefined rules to optimize the resolution of logical plans.

- Examples of rules:
  - Predicate Pushdown: Moves filters closer to the data source.
  - Projection Pruning: Removes unnecessary columns.
  - Constant Folding: Precomputes constant expressions.

4. Physical Plan Generation:
  - Multiple physical plans (execution strategies) are generated for the optimized logical plan.
  - Examples: Broadcast join, shuffle join.

5. Cost-Based Optimization (CBO):
  - Spark uses a cost model to choose the most efficient physical plan.
  - Relies on data statistics (if available) for better decision-making.

6. Execution:
  - The chosen physical plan is converted to RDD operations.
  - Executors process the tasks based on these operations.

- # Why Is It Important?
- - Performance Boost: Optimizes queries to run faster.
- - Automatic: No need to manually optimize queries.
- - Flexible: Works with various data sources and query types.

- # Key Concepts to Remember:
- - Unresolved Logical Plan: Initial plan from user code.
- - Resolved Logical Plan: Validated plan with schema information.

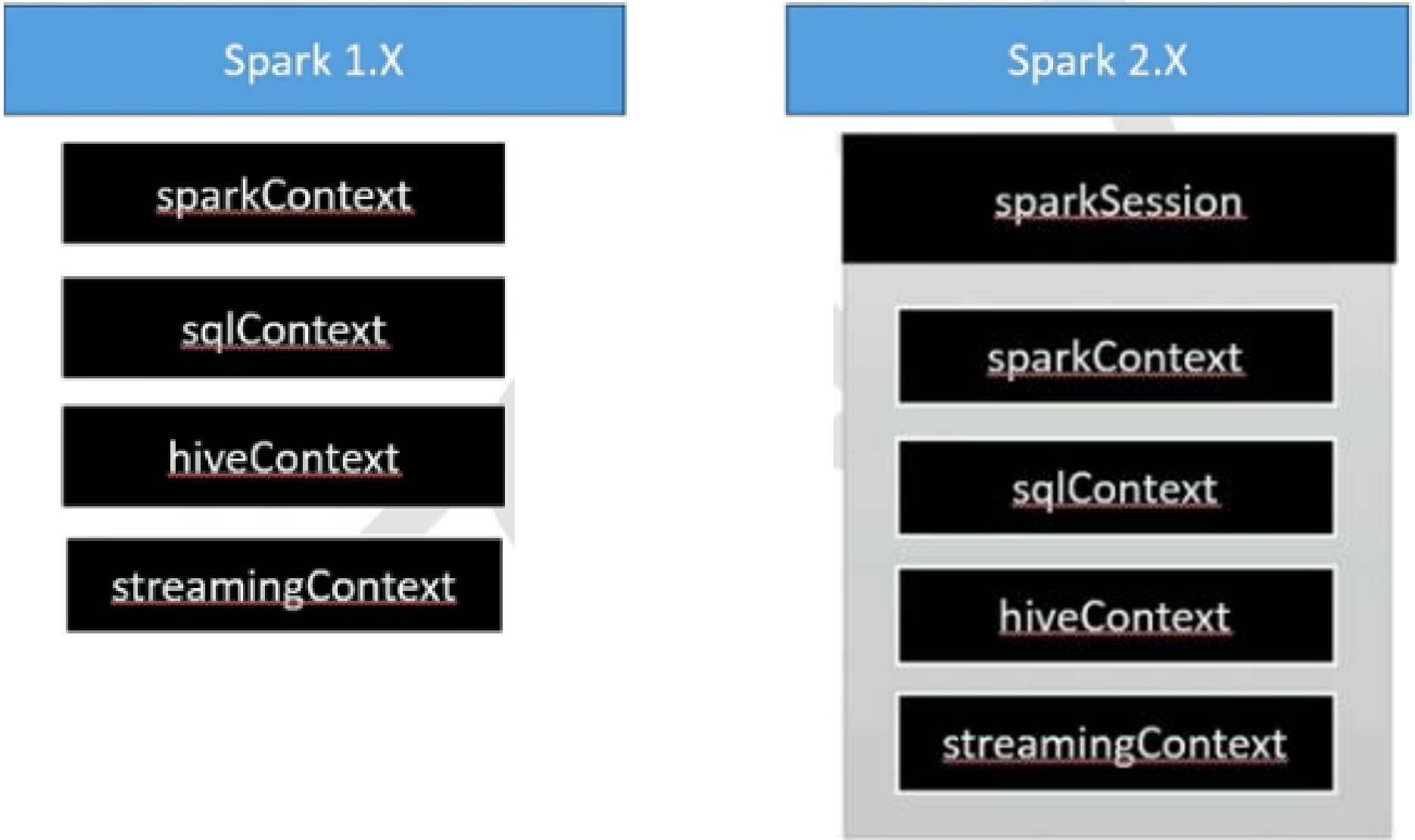
- - Optimized Logical Plan: Improved plan after applying rules.
- - Physical Plan: Execution strategy sent to executors.

- **Total Number of Rules :**
- - Spark includes dozens of rules (more than 100 in recent versions) for optimization.
- - These rules are defined in the org.apache.spark.sql.catalyst.rules.Rule class and implemented in packages like org.apache.spark.sql.catalyst.optimizer.
- - Examples of commonly applied rules:
- - Predicate Pushdown: Moves filters closer to the data source.
- - Constant Folding: Precomputes constant expressions.
- - Projection Pruning: Removes unnecessary columns from queries.
- - Reorder Join: Reorganizes joins for efficiency.
- - Eliminate No-Op Operations: Removes redundant transformations.
- - Simplify Expressions: Simplifies expressions to reduce computational complexity.

- # How Many Rules Are Applied to a Logical Plan?

- - The number of rules applied depends on:
- - The complexity of the logical plan.
- - The nature of operations in the query.
- - The transformations required to optimize the plan.
- - For a simple query, only a small subset of rules may be applied.
- - For a complex query with multiple joins, filters, and projections, many more rules may come into play.

- **Spark Context vs. Spark Session**

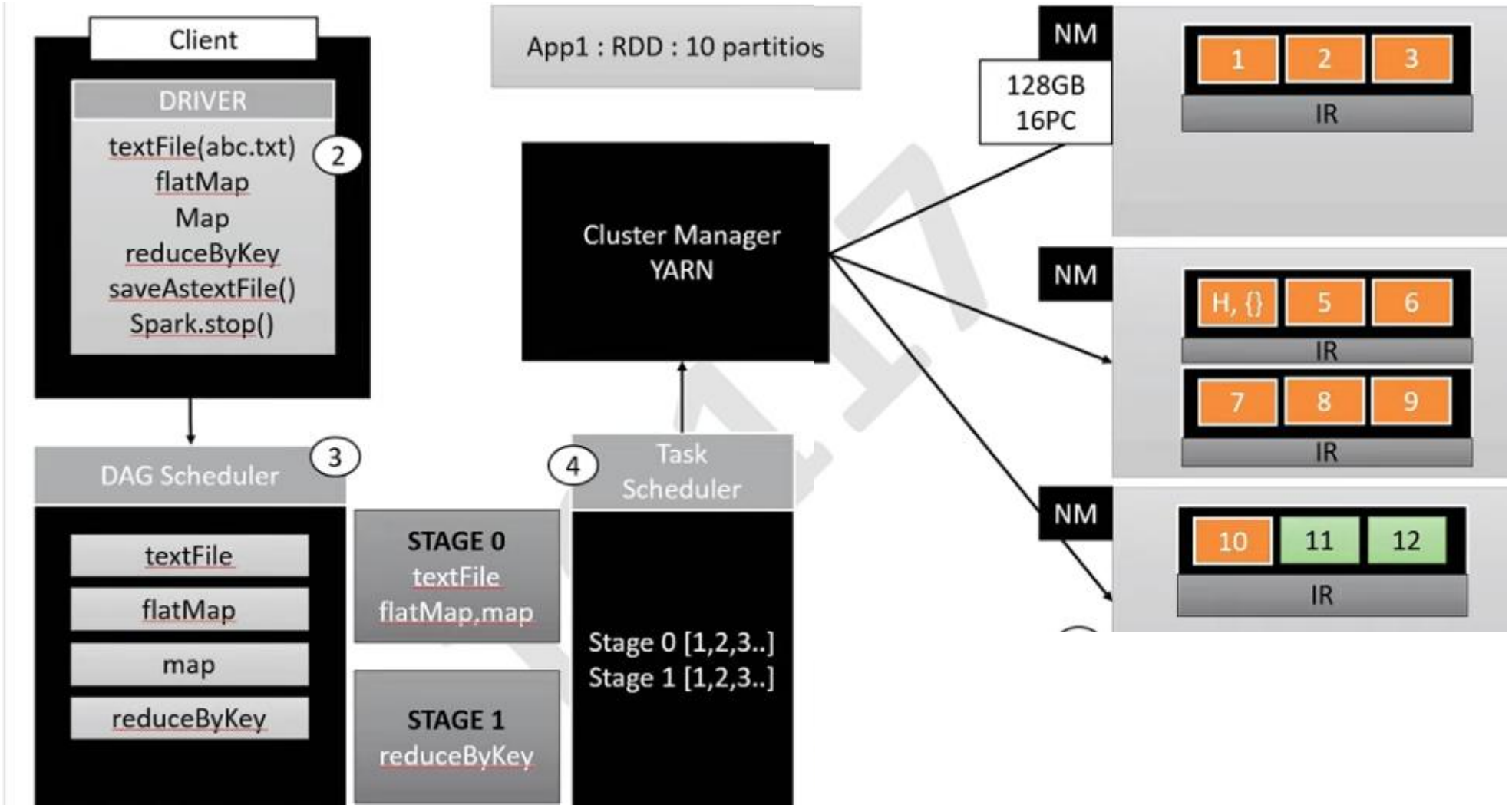

Spark 1.X
sparkContext
sqlContext
hiveContext
streamingContext
Spark 2.X
sparkSession
sparkContext
sqlContext
hiveContext
streamingContext

SparkContext

- - Low level API
- - RDD
- - Broadcast Variable
- - Accumulator

SparkSession

- - Streaming Function
- - Hive Integration
- - High Level API
- - DF
- - DS
- - Low level API
- - RDD
- - Broadcast Variable
- - Accumulator


Client
DRIVER
textFile(abc.txt)
flatMap
Map
reduceByKey
saveAstextFile()
Spark.stop()
2
App1 : RDD : 10 partitios
Cluster Manager
YARN
NM
128GB
16PC
1
2
3
IR
NM
H, {}
5
6
IR
7
8
9
IR
NM
10
11
12
IR
DAG Scheduler
3
4
Task
Scheduler
textFile
flatMap
map
reduceByKey
STAGE 0
textFile
flatMap,map
STAGE 1
reduceByKey
Stage 0 [1,2,3..]
Stage 1 [1,2,3..]

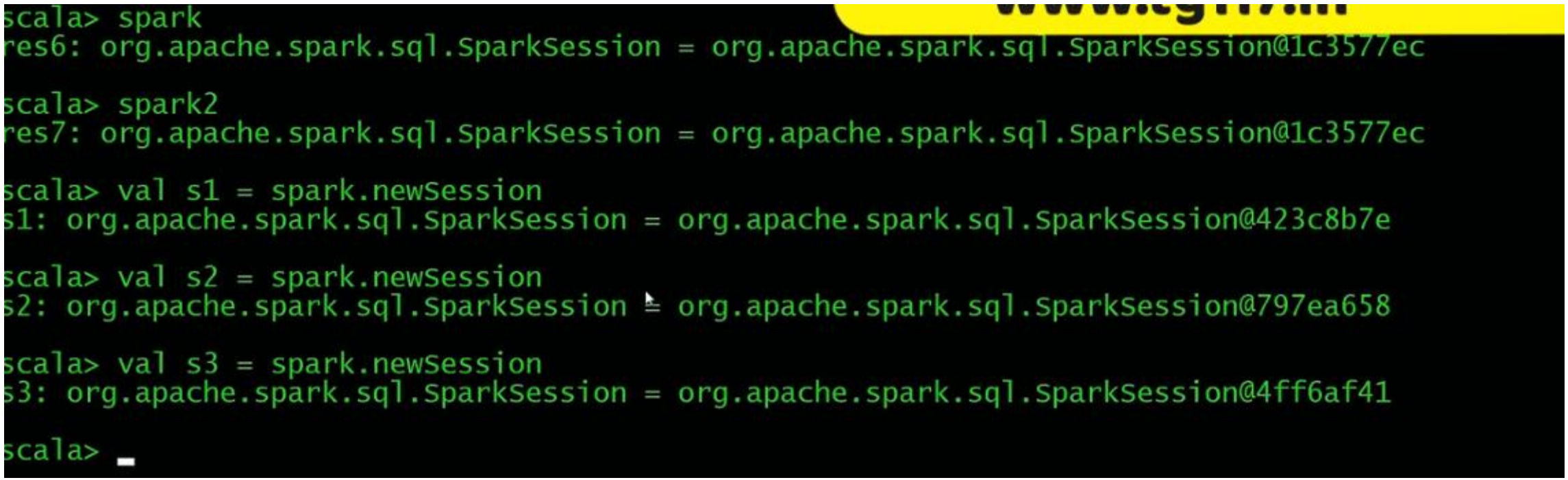
SparkContext

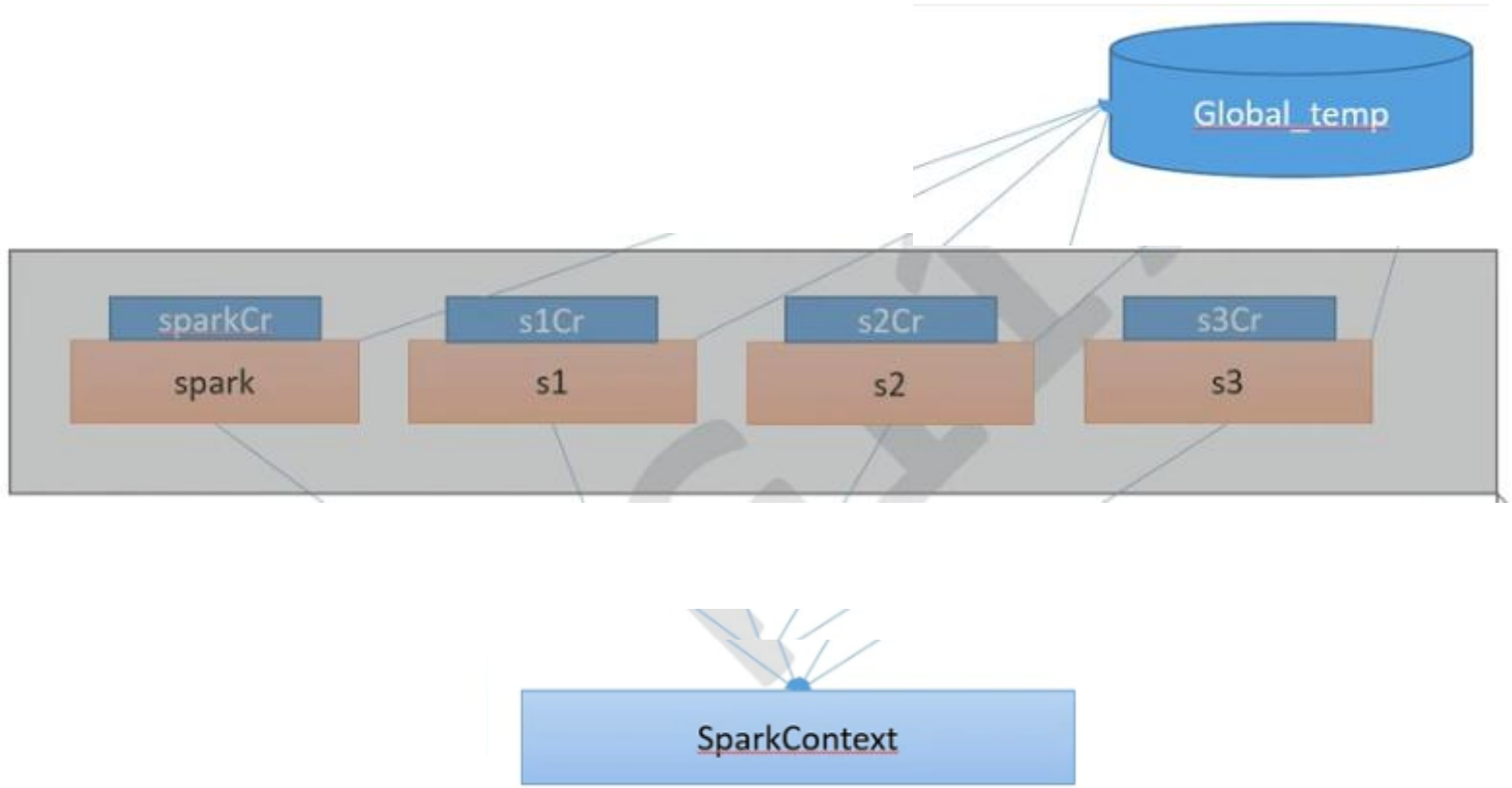
SparkContext

- Entry Point to Spark Application before Spark 2.0
- Handles low level APIs: RDD, Broadcast Variable, Accumulator
- There is only one SparkContext per Spark application
- Spark Context holds the references to DAGScheduler, TaskScheduler, ClusterManager and is useful to driver in communicating with these components
- SparkContext resides in the Driver

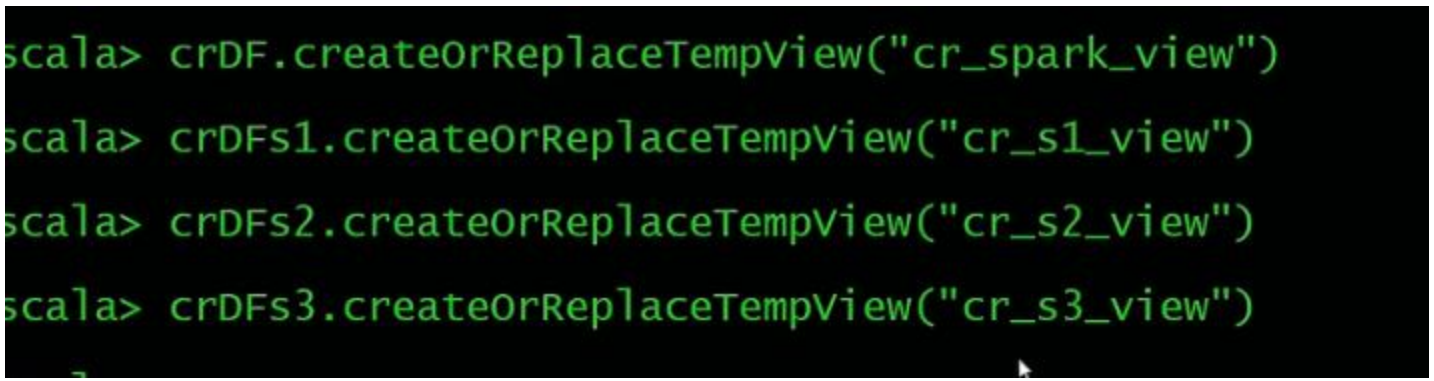
- # SparkContext

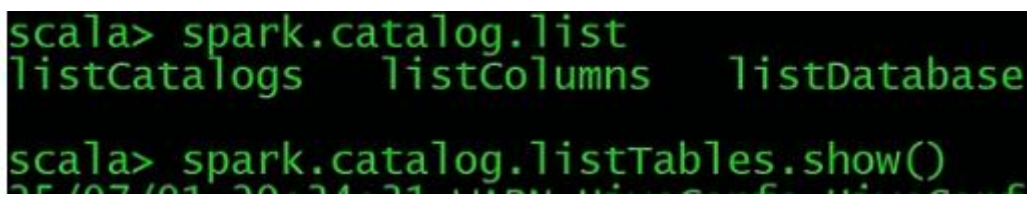
- - Entry Point to Spark Application before Spark 2.0
- - Handles low level APIs: RDD, Broadcast Variable, Accumulator
- - There is only one SparkContext per Spark application
- - Spark Context holds the references to DAGScheduler, TaskScheduler, ClusterManager and is useful to driver in communicating with these components
- - SparkContext resides in the Driver
- - SC is an interface between Driver and Cluster Manager


scala> spark
res6: org.apache.spark.sql.SparkSession = org.apache.spark.sql.SparkSession@1c3577ec
scala> spark2
res7: org.apache.spark.sql.SparkSession = org.apache.spark.sql.SparkSession@1c3577ec
scala> val s1 = spark.newSession
s1: org.apache.spark.sql.SparkSession = org.apache.spark.sql.SparkSession@423c8b7e
scala> val s2 = spark.newSession
s2: org.apache.spark.sql.SparkSession = org.apache.spark.sql.SparkSession@797ea658
scala> val s3 = spark.newSession
s3: org.apache.spark.sql.SparkSession = org.apache.spark.sql.SparkSession@4ff6af41
scala>


Global_temp
sparkCr
s1Cr
s2Cr
s3Cr
spark
s1
s2
s3
SparkContext

```
scala> val crDF = spark.read.option("inferSchema","true").option("header","true").csv("fil
e:///C:\\Users\\Sandeep\\Desktop\\data\\cr.txt")
crDF: org.apache.spark.sql.DataFrame = [cid: int, cname: string ... 1 more field]

scala> val crDFs1 = s1.read.option("inferSchema","true").option("header","true").csv("file
:///C:\\Users\\Sandeep\\Desktop\\data\\cr.txt")
crDFs1: org.apache.spark.sql.DataFrame = [cid: int, cname: string ... 1 more field]

scala> val crDFs2 = s2.read.option("inferSchema","true").option("header","true").csv("file
:///C:\\Users\\Sandeep\\Desktop\\data\\cr.txt")
crDFs2: org.apache.spark.sql.DataFrame = [cid: int, cname: string ... 1 more field]

scala> val crDFs3 = s3.read.option("inferSchema","true").option("header","true").csv("file
:///C:\\Users\\Sandeep\\Desktop\\data\\cr.txt")
crDFs3: org.apache.spark.sql.DataFrame = [cid: int, cname: string ... 1 more field]

scala>
```


scala> crDF.createOrReplaceTempView("cr_spark_view")
scala> crDFs1.createOrReplaceTempView("cr_s1_view")
scala> crDFs2.createOrReplaceTempView("cr_s2_view")
scala> crDFs3.createOrReplaceTempView("cr_s3_view")


scala> spark.catalog.list
listCatalogs listColumns listDatabase
scala> spark.catalog.listTables.show()

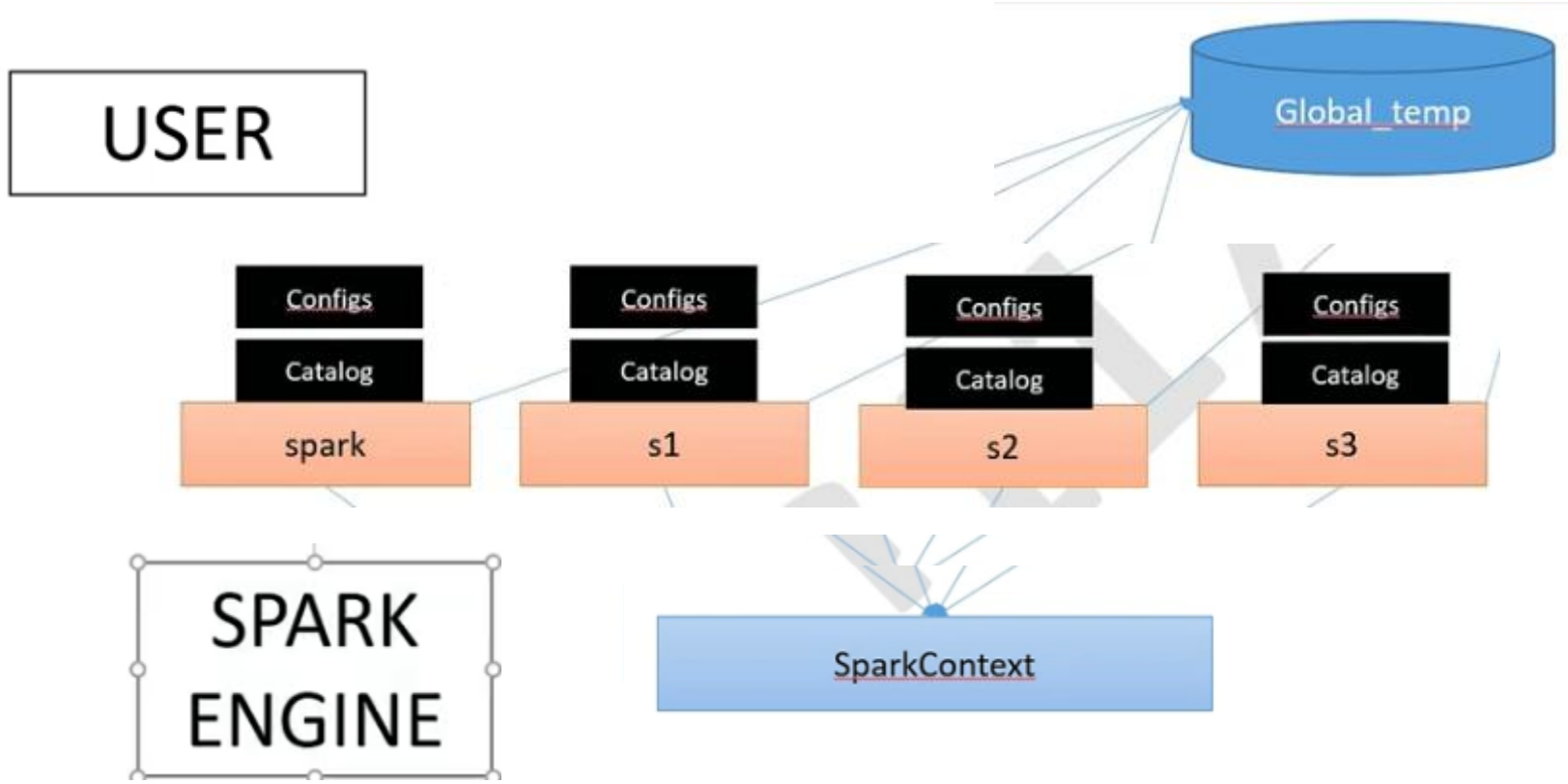
```
scala> spark.catalog.listTables.show()
+------------+-------+---------+-----------+---------+-----------+
|        name|catalog|namespace|description|tableType|isTemporary|
+------------+-------+---------+-----------+---------+-----------+
|cr_spark_view|   NULL|       []|       NULL|TEMPORARY|       true|
+------------+-------+---------+-----------+---------+-----------+


scala> s1.catalog.listTables.show()
+---------+-------+---------+-----------+---------+-----------+
|     name|catalog|namespace|description|tableType|isTemporary|
+---------+-------+---------+-----------+---------+-----------+
|cr_s1_view|   NULL|       []|       NULL|TEMPORARY|       true|
+---------+-------+---------+-----------+---------+-----------+


scala> s2.catalog.listTables.show()
+---------+-------+---------+-----------+---------+-----------+
|     name|catalog|namespace|description|tableType|isTemporary|
+---------+-------+---------+-----------+---------+-----------+
|cr_s2_view|   NULL|       []|       NULL|TEMPORARY|       true|
+---------+-------+---------+-----------+---------+-----------+
```

```
scala> s1.sql("select * from cr_s1_view").show(2)
+---+-----+-------+
|cid|cname|revenue|
+---+-----+-------+
|  1|    A|     45|
|  2|    B|     55|
+---+-----+-------+
only showing top 2 rows
```

```
scala> crDF.createOrReplaceGlobalTempView("cr_global")

scala> _
```

```
  at org.apache.spark.sql.catalyst.plans.logical.AnalysisHelper$.markInA

scala> spark.sql("select * from global_temp.cr_global").show(2)
+---+-----+-------+
|cid|cname|revenue|
+---+-----+-------+
|  1|    A|     45|
|  2|    B|     55|
+---+-----+-------+
only showing top 2 rows


scala> s1.sql("select * from global_temp.cr_global").show(2)
+---+-----+-------+
|cid|cname|revenue|
+---+-----+-------+
|  1|    A|     45|
|  2|    B|     55|
+---+-----+-------+
only showing top 2 rows


scala> s2.sql("select * from global_temp.cr_global").show(2)
+---+-----+-------+
|cid|cname|revenue|
+---+-----+-------+
|  1|    A|     45|
|  2|    B|     55|
+---+-----+-------+
only showing top 2 rows
```

```
scala> spark.conf.set("spark.sql.shuffle.partitions","5")
scala> s1.conf.set("spark.sql.shuffle.partitions","100")
scala> s2.conf.set("spark.sql.shuffle.partitions","20")
scala>
```


USER
Global_temp
Configs
Catalog
spark
Configs
Catalog
s1
Configs
Catalog
s2
Configs
Catalog
s3
SPARK ENGINE
SparkContext