Apache SPARK

2CEIT702: Big Data Analytics

Big Data Streaming Platforms

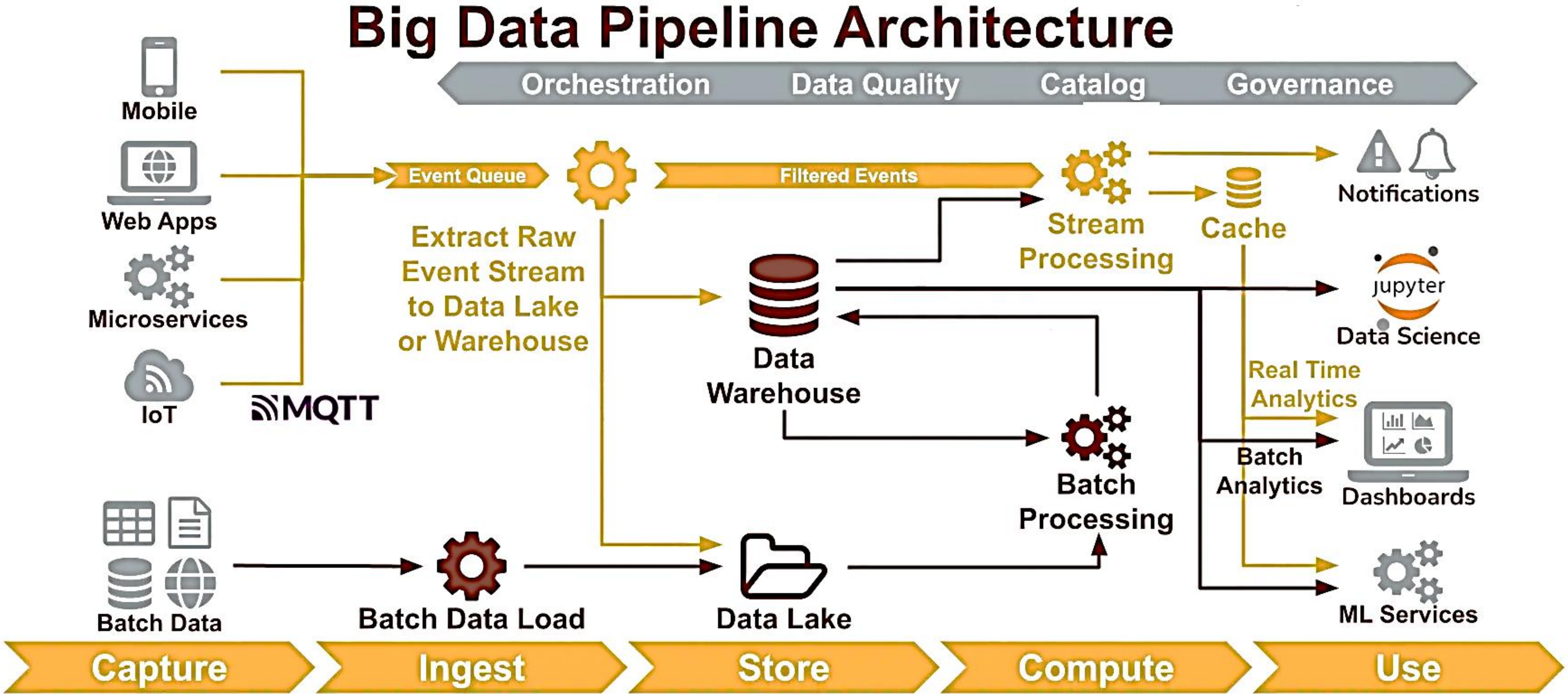
- Traditional Business Intelligence tools are not suitable for analysing streaming data in real time, because it is processed in batch processing.

- So, a large number of big data streaming platforms have been developed based on the requirements/features of the companies.

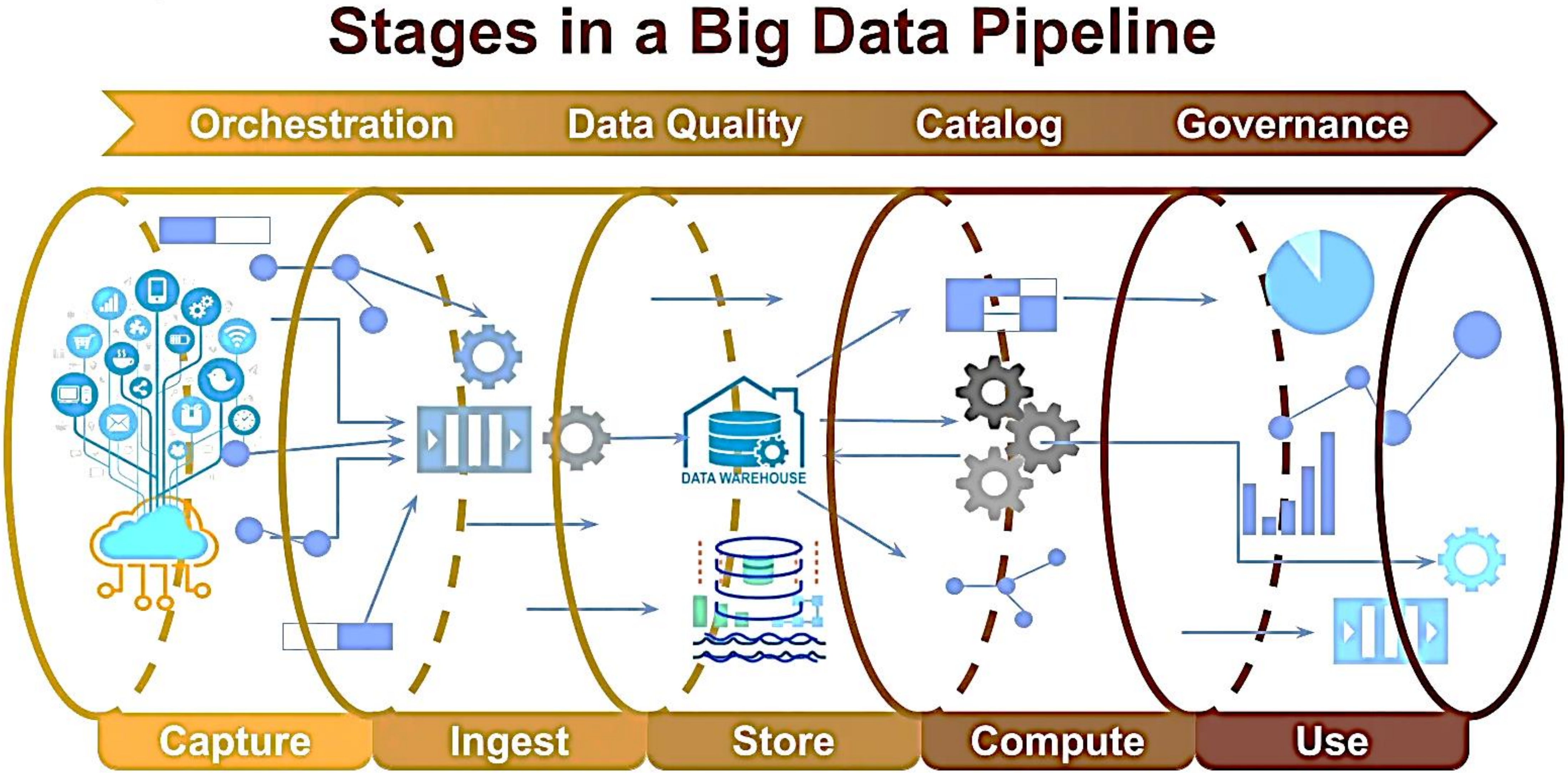
- Example: Spark, Kafka, Samza, Flink and Storm

Data Pipeline (Real-time computing Big data)

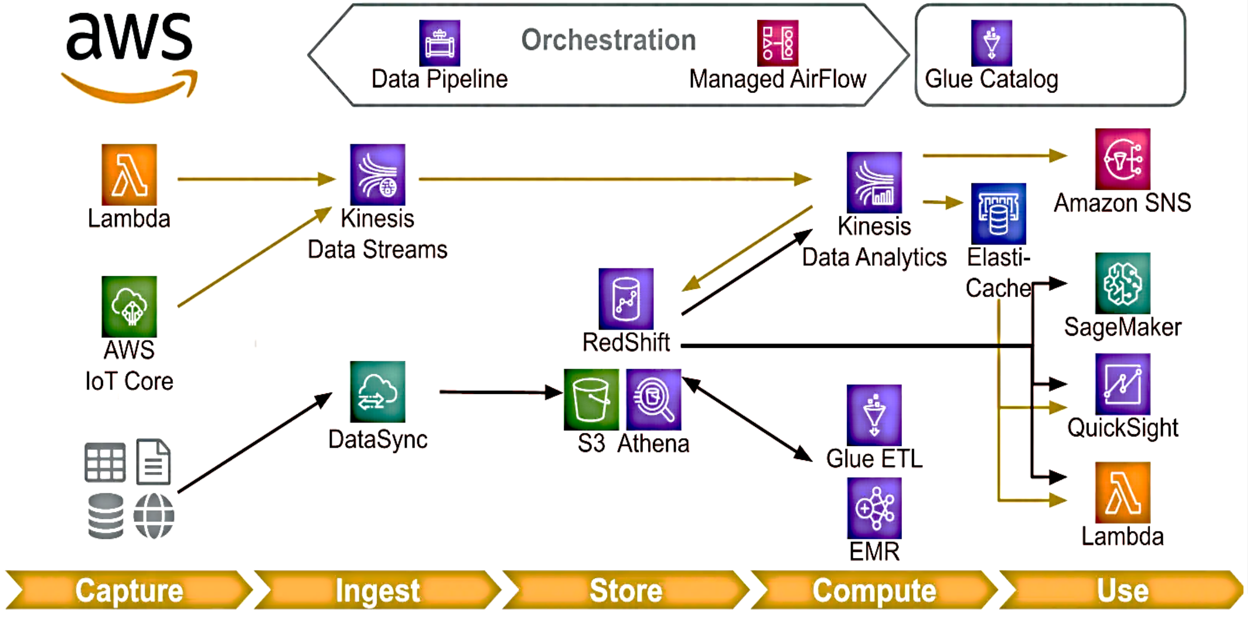
- It is a sequence of data processing steps. Data ingested at the beginning of the pipeline if not loaded. Perform the series of steps. A data pipeline is designed to transform data into a usable format as the information flows through the system.

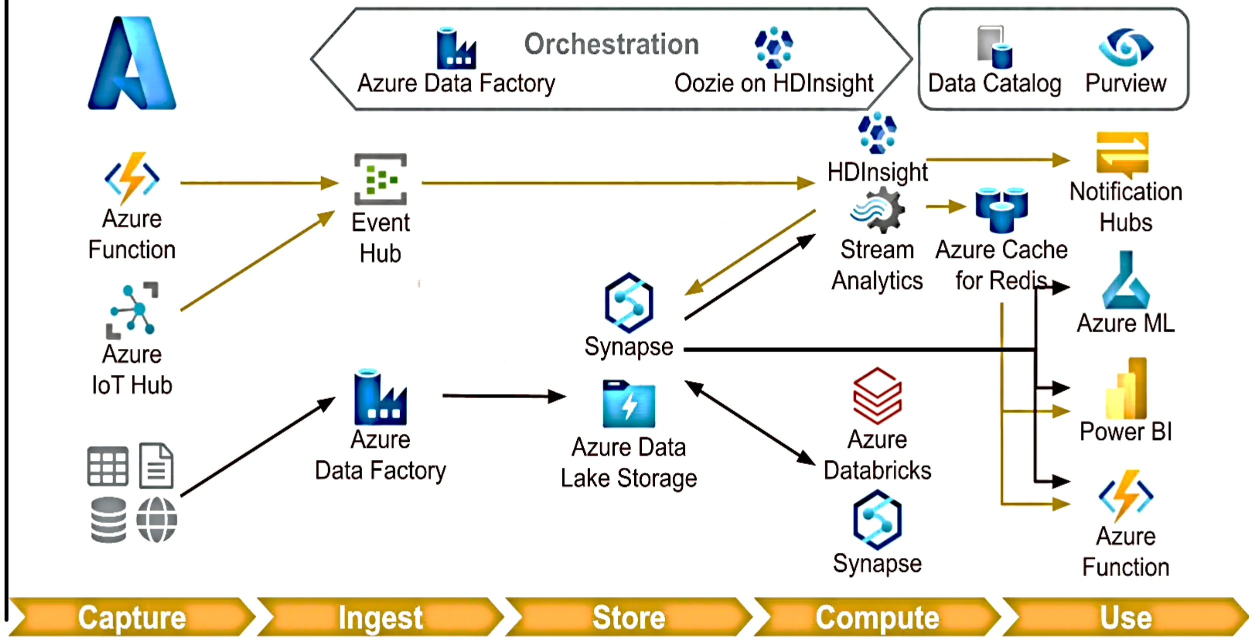


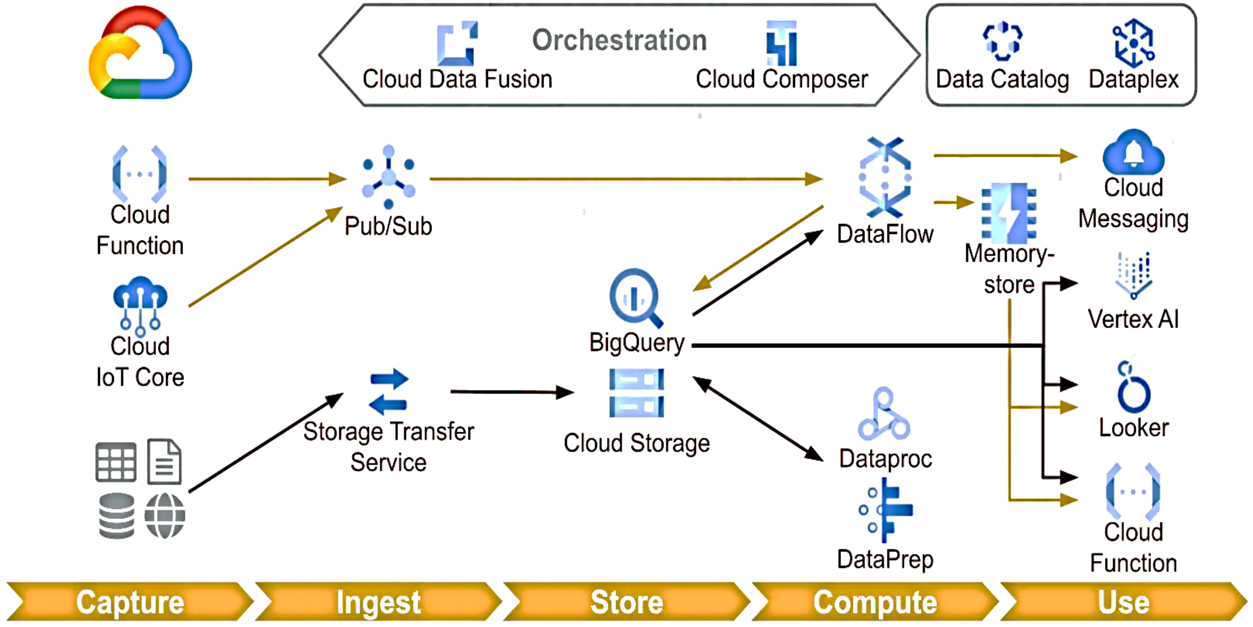
Data Pipeline (Real-time computing Big data)



Big Data Pipelines on AWS, Azure, and Google Cloud

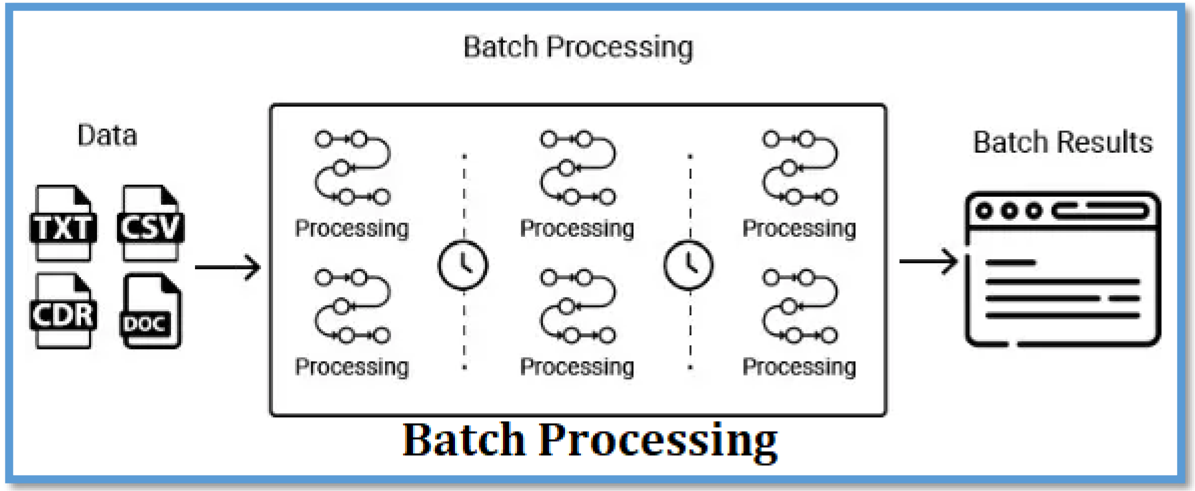


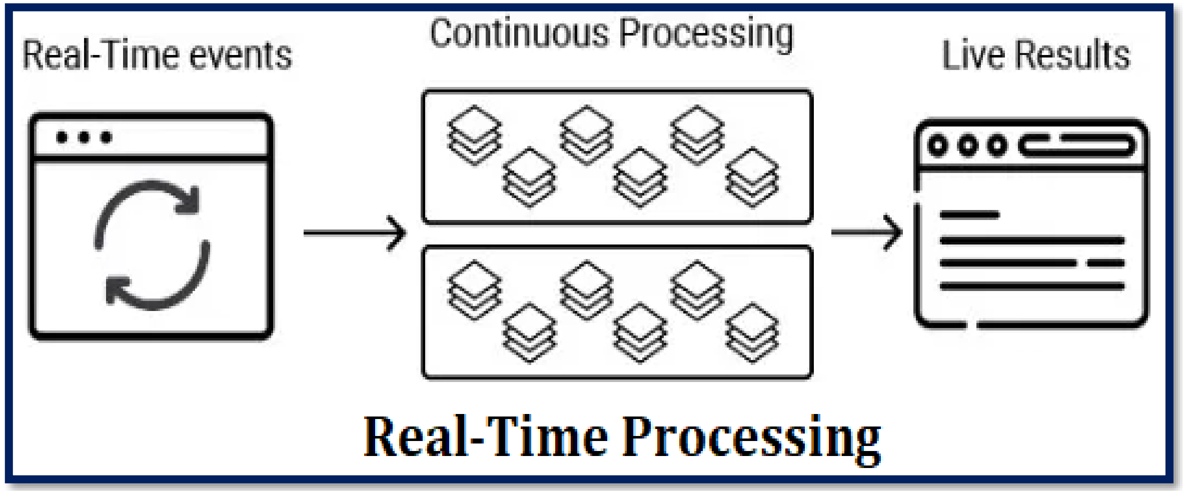




Batch Processing Vs Real-Time Processing

- **Batch processing:** process the data in batches means do not process one row at a time but, collection of row data is processed at regular intervals such as every 15min, hourly, or daily.
  - (Example: Cheque clearing, Bills generation)
- **Real-time processing:** it is continuously generated or received means that within the data stream it processes one row at a time. (Example: Reservation system, Point of Sale, Fraud Analysis, Virus found)





What Exactly is Apache Spark & How Does it Work?

Introduction

- There was no general purpose computing engine in the industry So:
  - To perform batch processing, we were using Hadoop MapReduce.
  - To perform stream processing, we were using Apache Storm / S4.
  - For interactive processing, we were using Apache Impala / Apache Tez.
  - To perform graph processing, we were using Neo4j / Apache Giraph.

- Hence there was no powerful engine in the industry, that can process the data both in real-time and batch mode.
- Also, there was a requirement that one engine can respond in sub-second and perform in-memory processing.
- **Therefore, Apache Spark programming enters, it is a powerful open source engine**

What exactly is Apache Spark?

- **Definition:** Apache Spark is a **unified computing engine and a set of libraries for parallel data processing on computer clusters**.
  - Unified means: (combine the task all together – Data Engineers, Data Analysts, Data Scientists)
  - Computing Engine: spark is limited to a computing engine. It does not store the data. Connect with all data sources(Azure storage, HDFS, etc..).
- It offers real-time stream processing, interactive processing, graph processing, in-memory processing and batch processing.
- Spark is another parallel processing framework like Hadoop. It optimized query execution for fast analytic queries against huge data.
- It is well integrated with Hadoop as it can run on top of YARN and can access HDFS.
- Today, Spark is being adopted by: Amazon, eBay, Yahoo, and others.

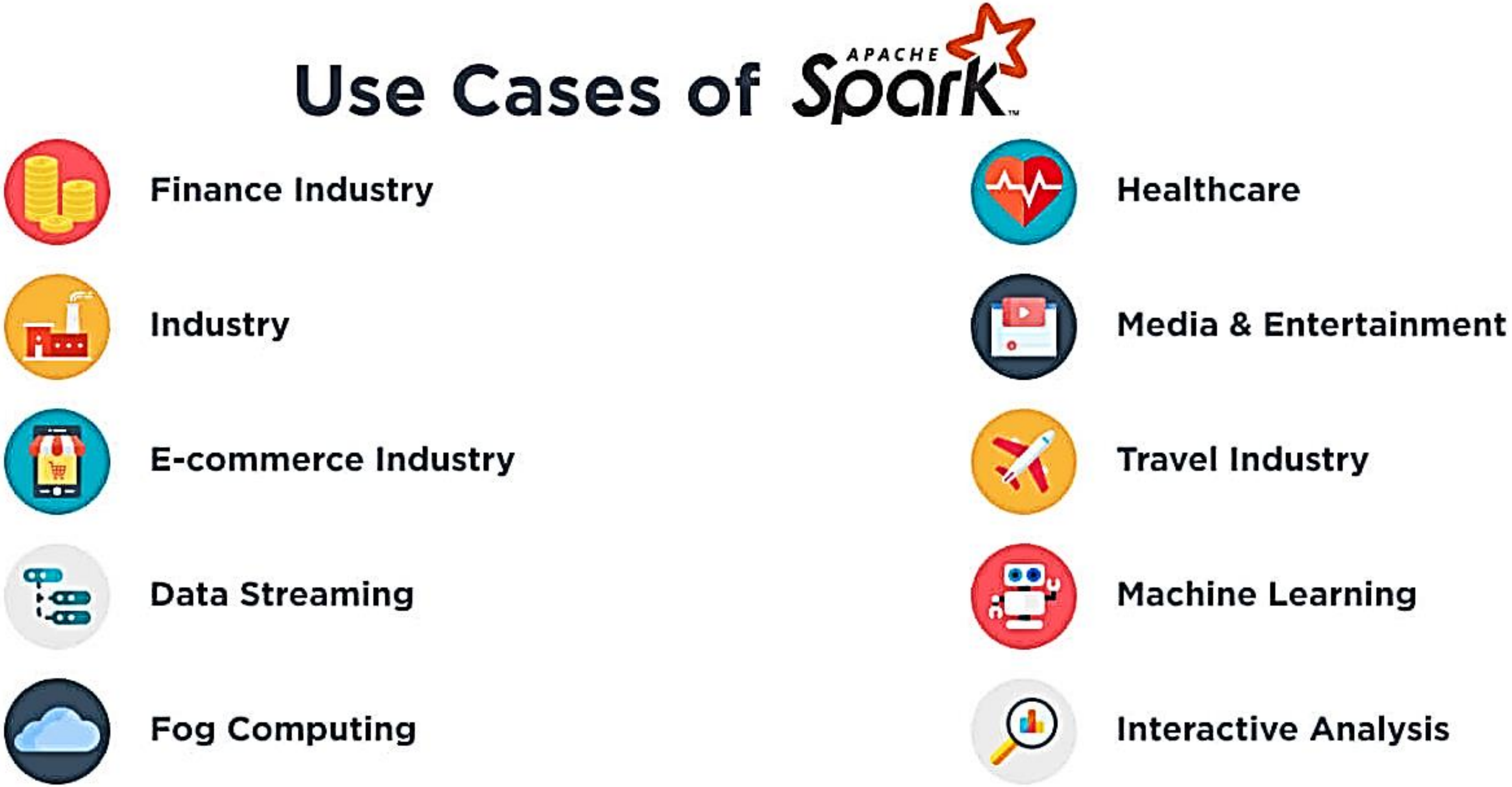
Features of Apache Spark

- **Advanced Analytics**–Spark not only supports ‘Map’ and ‘reduce’. It also supports SQL queries, Streaming data, Machine learning (ML), and Graph algorithms.

- **Supports multiple languages**–Spark provides built-in APIs in Java, Scala, or Python. Therefore, you can write applications in different languages. Spark comes up with 80 high-level operators for interactive querying.

- **Speed**– Spark works on **the concept of in-memory computation** so, it is 100 times faster than MapReduce and 10 times faster if using the Disk. This is possible by reducing number of read/write operations to disk. It stores the intermediate processing data in memory.

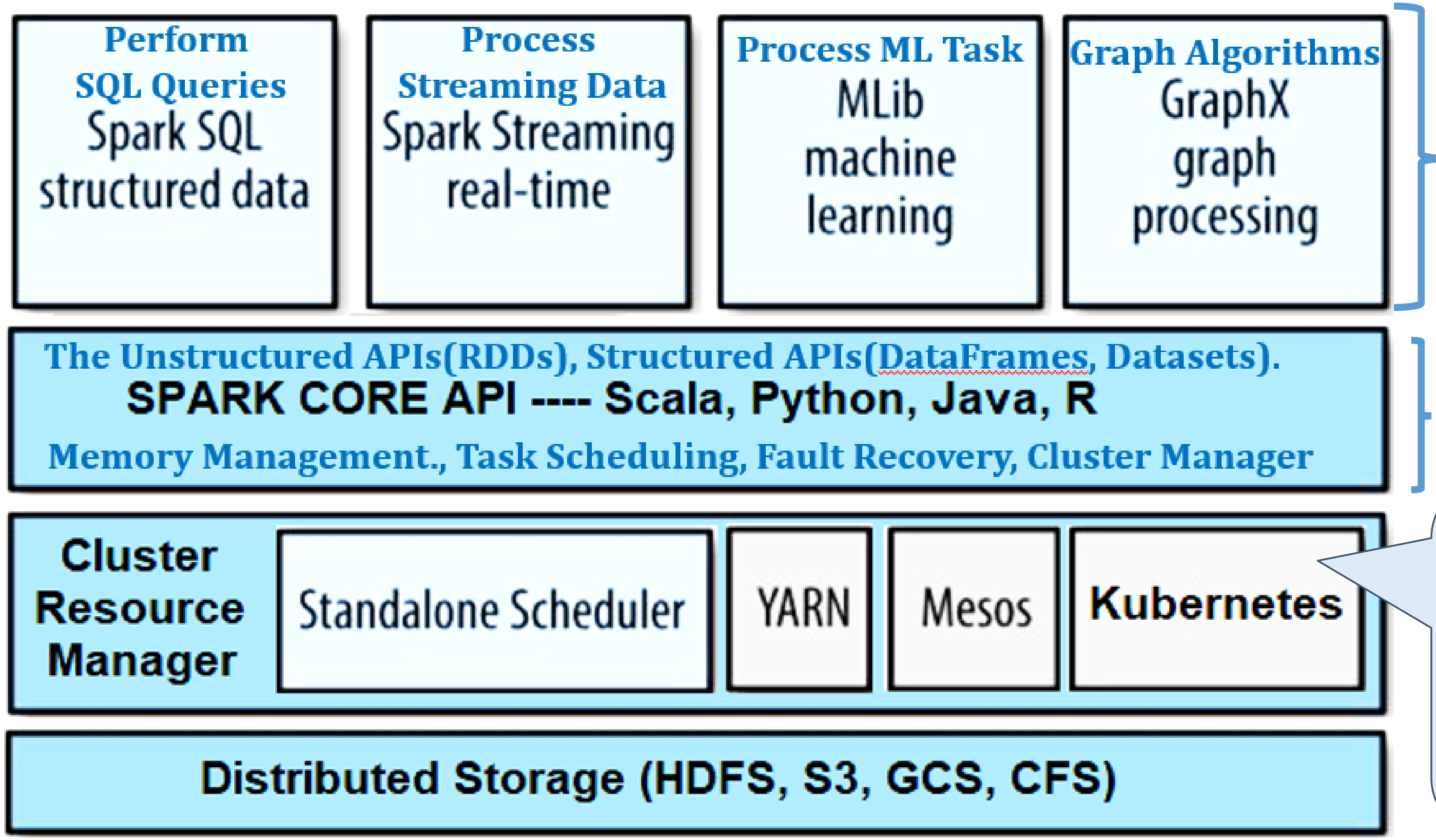
how Spark is used successfully in different industries?



Use Cases of APACHE Spark™

- Finance Industry
- Industry
- E-commerce Industry
- Data Streaming
- Fog Computing
- Healthcare
- Media & Entertainment
- Travel Industry
- Machine Learning
- Interactive Analysis

Spark Architecture



Perform SQL Queries
Spark SQL
structured data

Process Streaming Data
Spark Streaming
real-time

Process ML Task
MLib
machine learning

Graph Algorithms
GraphX
graph processing

The Unstructured APIs(RDDs), Structured APIs(DataFrames, Datasets).
SPARK CORE API ---- Scala, Python, Java, R
Memory Management., Task Scheduling, Fault Recovery, Cluster Manager

Cluster Resource Manager
Standalone Scheduler
YARN
Mesos
Kubernetes

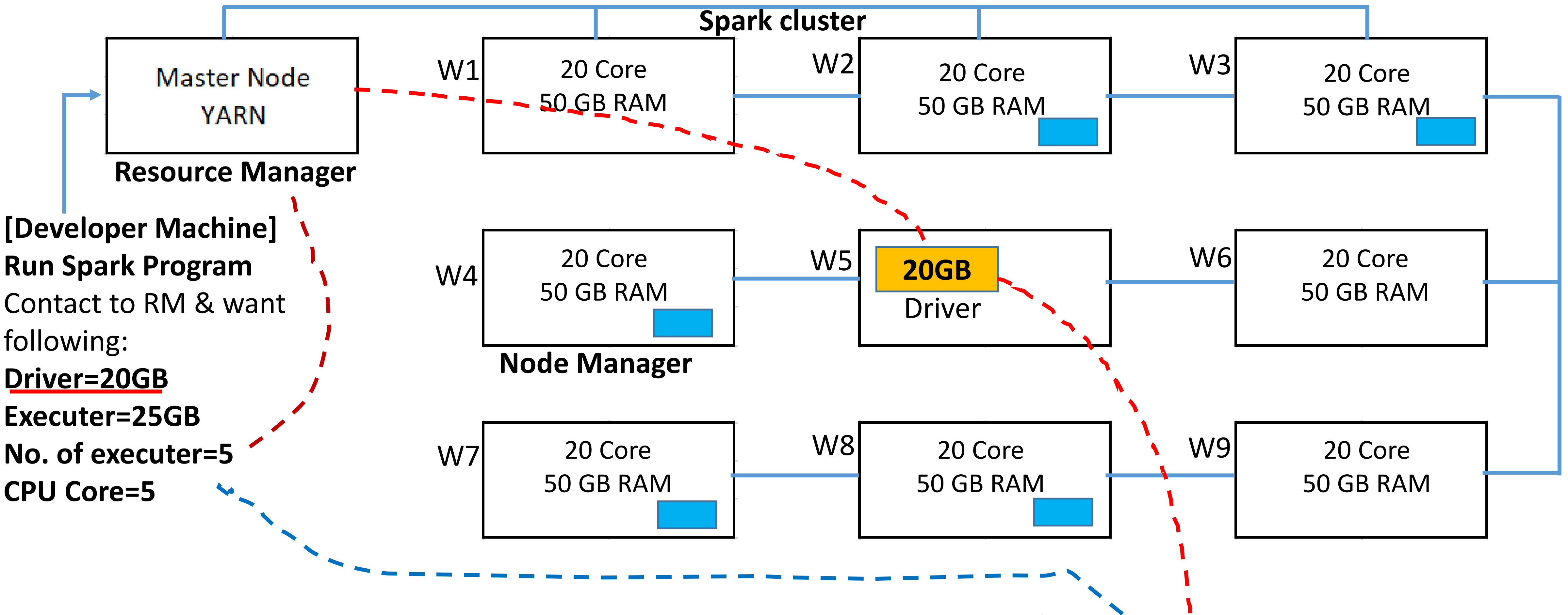
Distributed Storage (HDFS, S3, GCS, CFS)

- ❖ **High-level API**
- ❖ **Dataframe /Dataset**

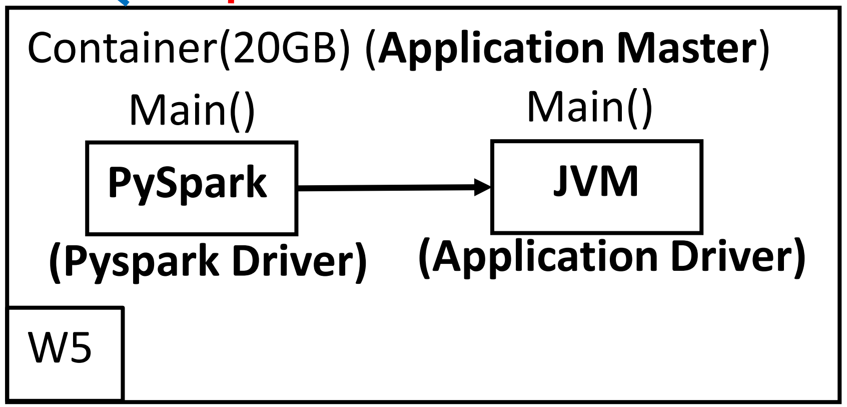
- ❖ **Low-level API**
- ❖ **RDD**

The cluster manager controls physical machines and allocates resources to Spark Applications.
This means that there can be multiple Spark Applications running on a cluster at the same time.

Spark is written in Scala
Today, Nobody write the code in RDD until it is required.

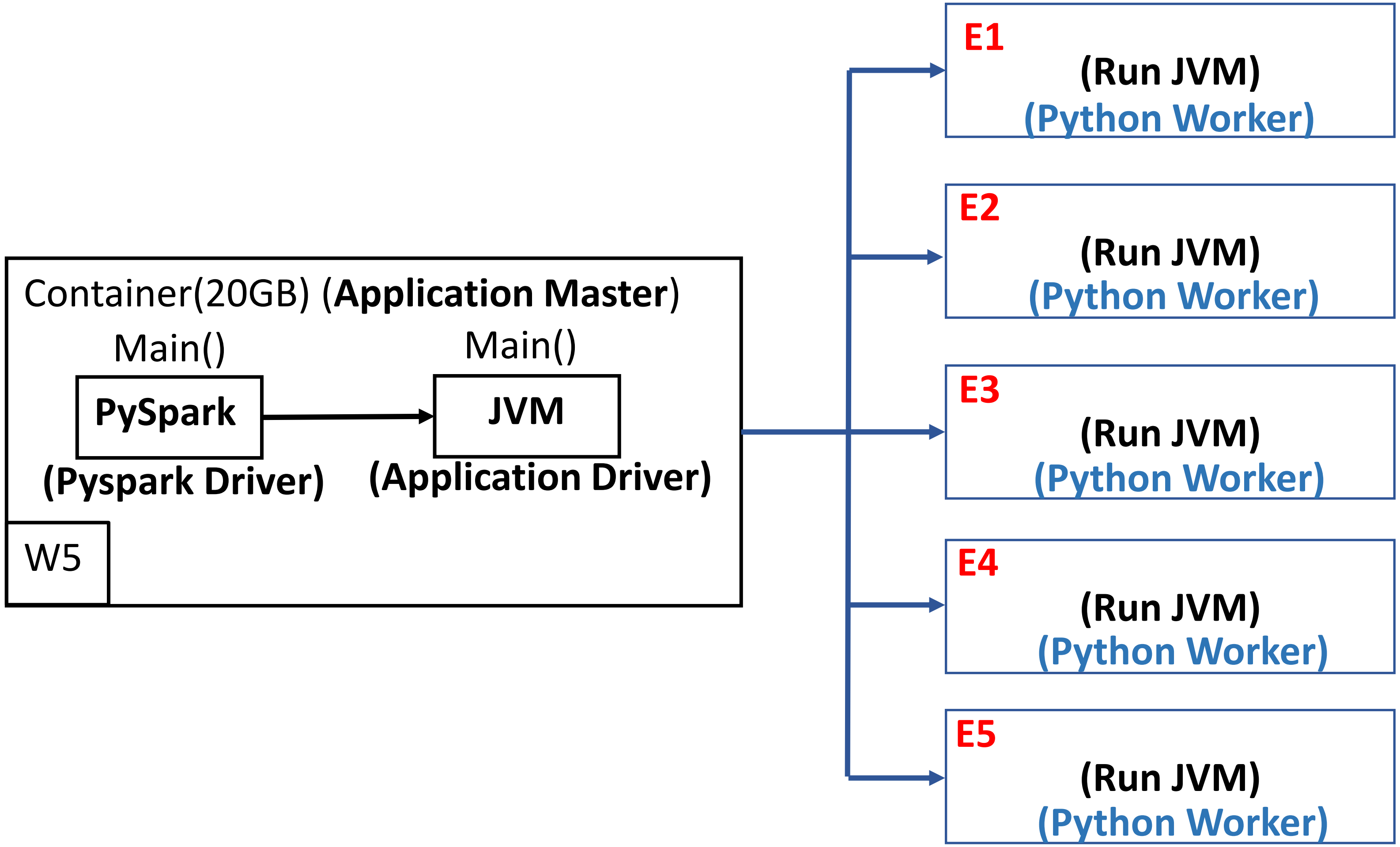


[Developer Machine]
Run Spark Program
Contact to RM & want following:
Driver=20GB
Executer=25GB
No. of executer=5
CPU Core=5



- ✓ RM contact to W5 and instruct to create container(Driver) with 20GB RAM. Where spark will run. Container known as Application Master (or Driver).
- ✓ Application Master known as Driver from where Spark code is run.
- ✓ Inside Application Master, create PySpark (if python code) and JVM Driver.
- ✓ Next, Application master ask the RM to create the executor containers-(W4,W7,W8,W2,W3)
- ✓ Next Diagram will show you: What happened in the executor containers.

Executor Containers



Python Worker is needed to execute UDF (user Define function). For better performance, do not use UDF functions.

Apache Spark’s Philosophy

- Unified
- Computing engine
- Libraries

Apache Spark's Philosophy-Unified

- Spark provide a unified platform (Single Engine) for processing different types of big data applications.
- Spark can handle batch processing, interactive queries, streaming analytics, machine learning, and graph processing.
- Spark provides consistent APIs, allowing users to reuse code and simplify their development process.
- For example, if you load data using a SQL query and then evaluate a machine learning model over it using Spark's ML library, the engine can combine these steps into one scan over the data.
- The combination of general APIs and high-performance execution, no matter how you combine them, makes Spark a powerful platform for interactive and production applications.
- Example…
    - Data scientists benefit from a unified set of libraries (e.g., Python or R) when modeling.
    - Web developers benefit from unified frameworks such as Node.js or Django

Apache Spark’s Philosophy-Computing Engine

- It carefully limits its scope to a computing engine.

- Spark handles loading data from storage systems and performing computation on it, not permanent storage as the end itself.

- You can use Spark with a wide variety of persistent storage systems, including cloud storage systems.
    - Such as Azure Storage and Amazon S3, distributed file systems such as Apache Hadoop, key-value stores such as Apache Cassandra, and message buses such as Apache Kafka.

- Spark runs well on Hadoop storage.

Apache Spark's Philosophy-Libraries

- Spark's final component is its libraries, which build on its design as a unified engine to provide a unified API for common data analysis tasks.

- Spark supports both standard libraries that ship with the engine as well as a wide array of external libraries published as third-party packages by the open-source communities.

- Spark's standard libraries are actually the bulk of the open source project: the Spark core engine itself has changed little since it was first released, but the libraries have grown to provide more and more types of functionality.

- Spark includes libraries for SQL and structured data (Spark SQL), ML (MLlib), stream processing (Spark Streaming and the newer Structured Streaming), and graph analytics (GraphX).

- Hundreds of open-source external libraries ranging from connectors for various storage systems to machine learning algorithms.

Spark!

- A *cluster* pools the resources of many machines together, giving us the ability to use all the cumulative resources as single computer.

- A group of machines alone is not powerful, you need a framework to coordinate work across them.

- Apache Spark is an open-source distributed computing system designed for big data processing and analytics. Spark is known for its speed and efficiency.

Spark!

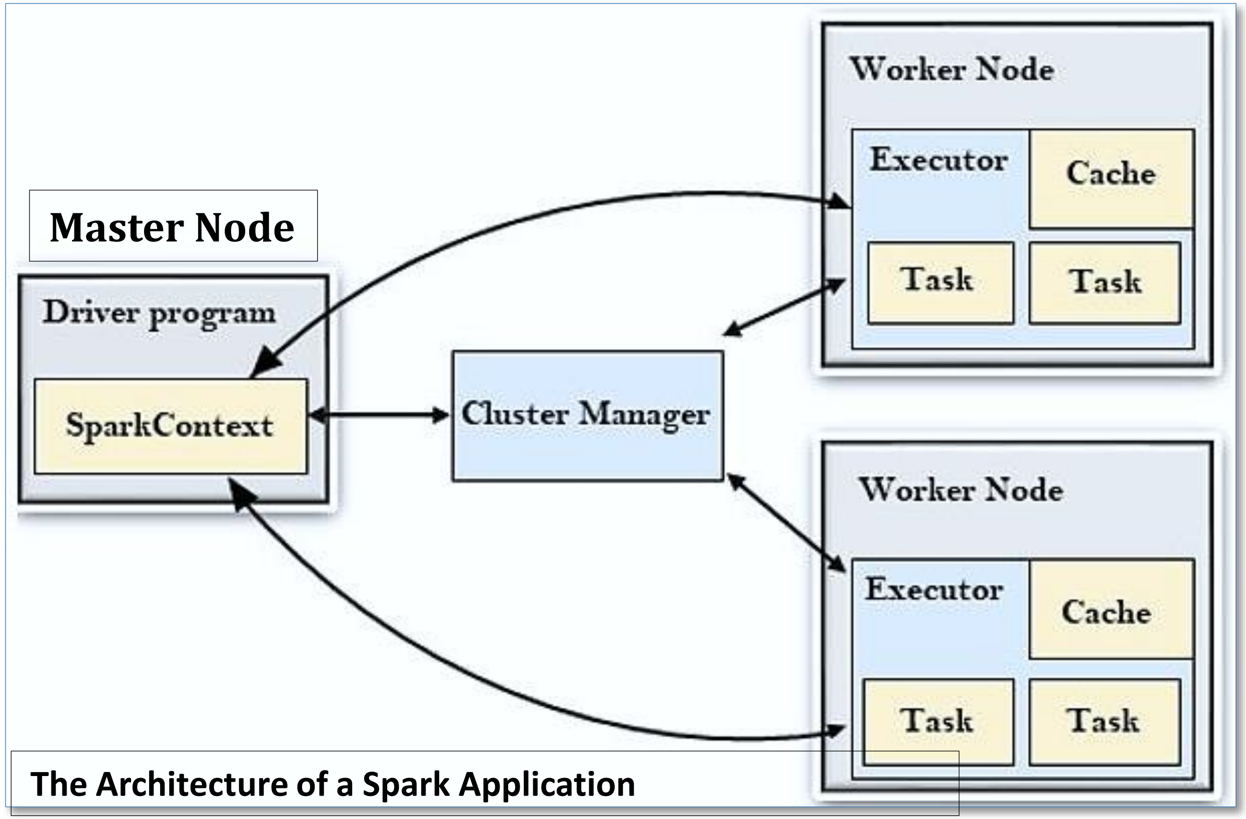
- Spark does just that, managing and coordinating the execution of tasks on data across a cluster of computers.

- The cluster of machines that Spark will use to execute tasks is managed by a cluster manager like: *Standalone cluster manager, YARN, and Mesos*

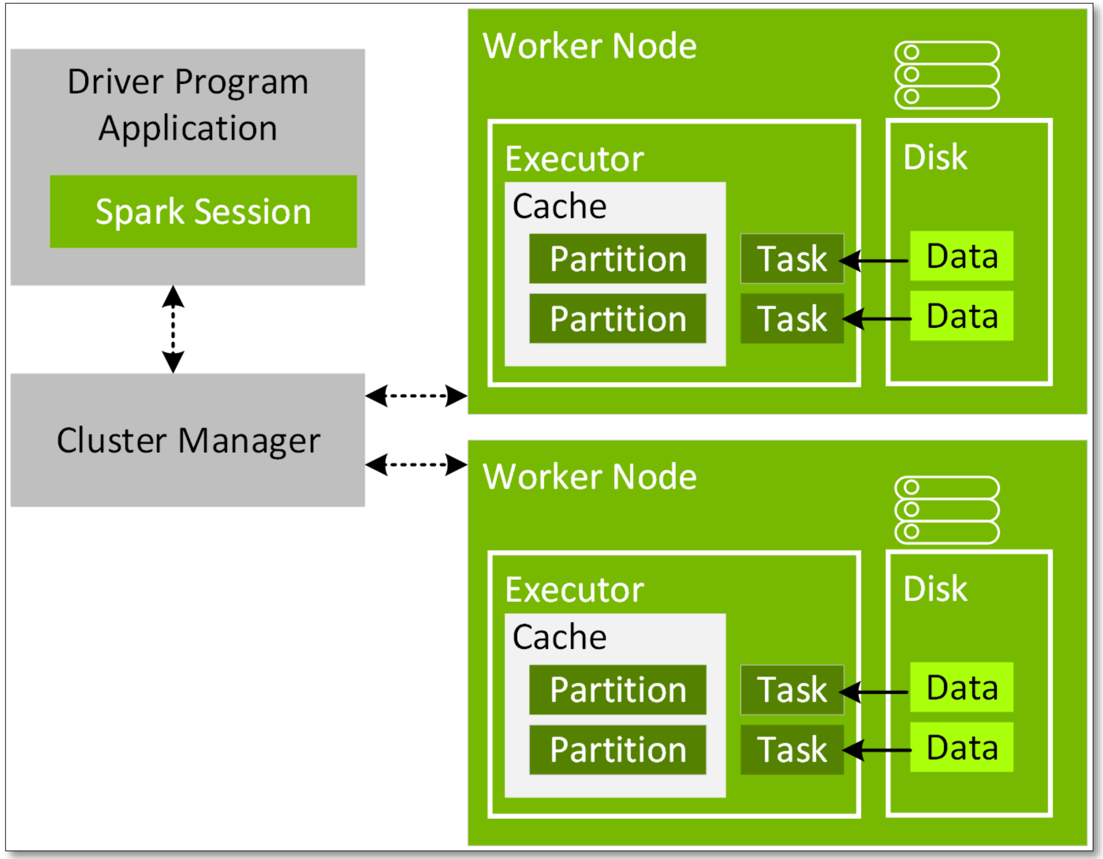
- We then submit Spark Applications to these cluster managers, which will grant resources to our application so that we can complete our work.

The high-level architecture of Spark

- When we run any job with Spark, this is how underlying execution happens.



The Architecture of a Spark Application



Cluster: A cluster is a set of computers that are viewed as a single system.

Worker: providing the physical resources to the Spark framework. A Spark application can have one or more workers.

Spark Applications

Spark Driver (Driver Program)

- Master node have the spark driver program.
- The Driver is the process that clients use to submit applications in Spark.
- It is basically a process where the main method runs and is responsible for three things:
  - Maintaining information about the Spark Application
  - Responding to a user’s program or input
  - Analysing, distributing, and scheduling work across the executors.
- It converts the user program into tasks and just after that it schedules the tasks on the executors.
- It executes the code and creates a SparkSession/ SparkContext.

Spark Applications

sparkContext / sparkSession

- Inside the driver program, create a SparkContext or SparkSession first. It is a gateway to all the Spark functionalities. It is similar to your database connection. Any command you execute in your database goes through the database connection. Likewise, anything you do on Spark goes through Sparkcontext or SparkSession.

- It creates the bridge between the cluster, executors, and driver.

- **SparkContext** takes the job, breaks the job in tasks and distribute them to the worker nodes. These tasks work on the partitioned RDD, perform operations, collect the results and return to the main SparkContext.

- **Driver program** & **SparkContext** takes care of the job execution within the cluster.

Spark Applications

Executors

- **Executors** have one core responsibility, that is to take the tasks assigned by the driver, run them, and report back their state (success or failure) and results.

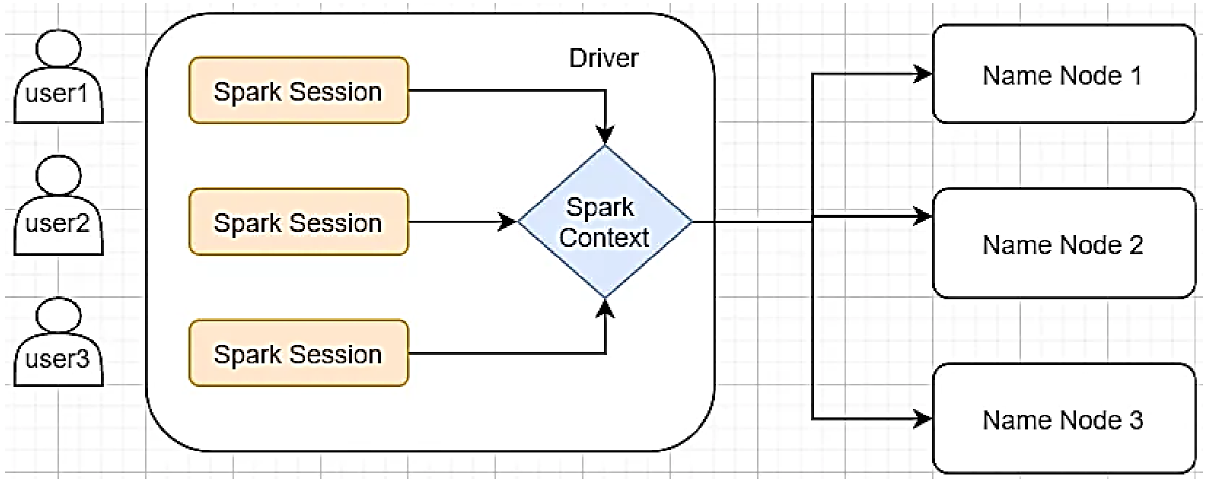
- Executor is responsible for:
  - **1)** Executing code assigned by the driver
  - **2)** Reporting the state of the computation, on that executor, back to the driver node.

- **Worker nodes** are the slave nodes whose job is to basically execute the tasks.

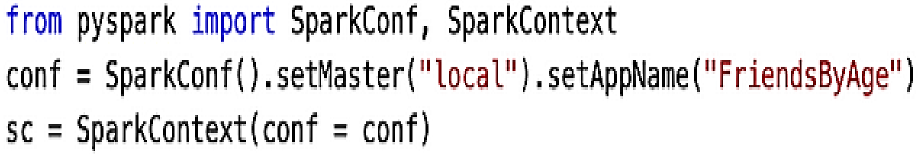
- If you increase the number of workers, then you can divide jobs into more partitions and execute them parallel over multiple systems. It will be a lot faster.

SparkContext Vs SparkSession

SparkContext	SparkSession
Spark Context is the only entry point for spark driver program to the cluster. (Spark 1.0) (Traditional approach)	SparkSession is the unified entry point for spark driver program to the cluster.(Spark 2.0) (Modern approach)
SparkContext is designed for low-level programming	SparkSession is designed for high-level programming
spark context only gives access to RDD API. To access other API, you have to create separate different contexts such as SQLContext, HiveContext etc.	It encapsulates functionality from SparkContext, SQLContext, and HiveContext into a single interface.. Support RDD, Dataframe, and Dataset

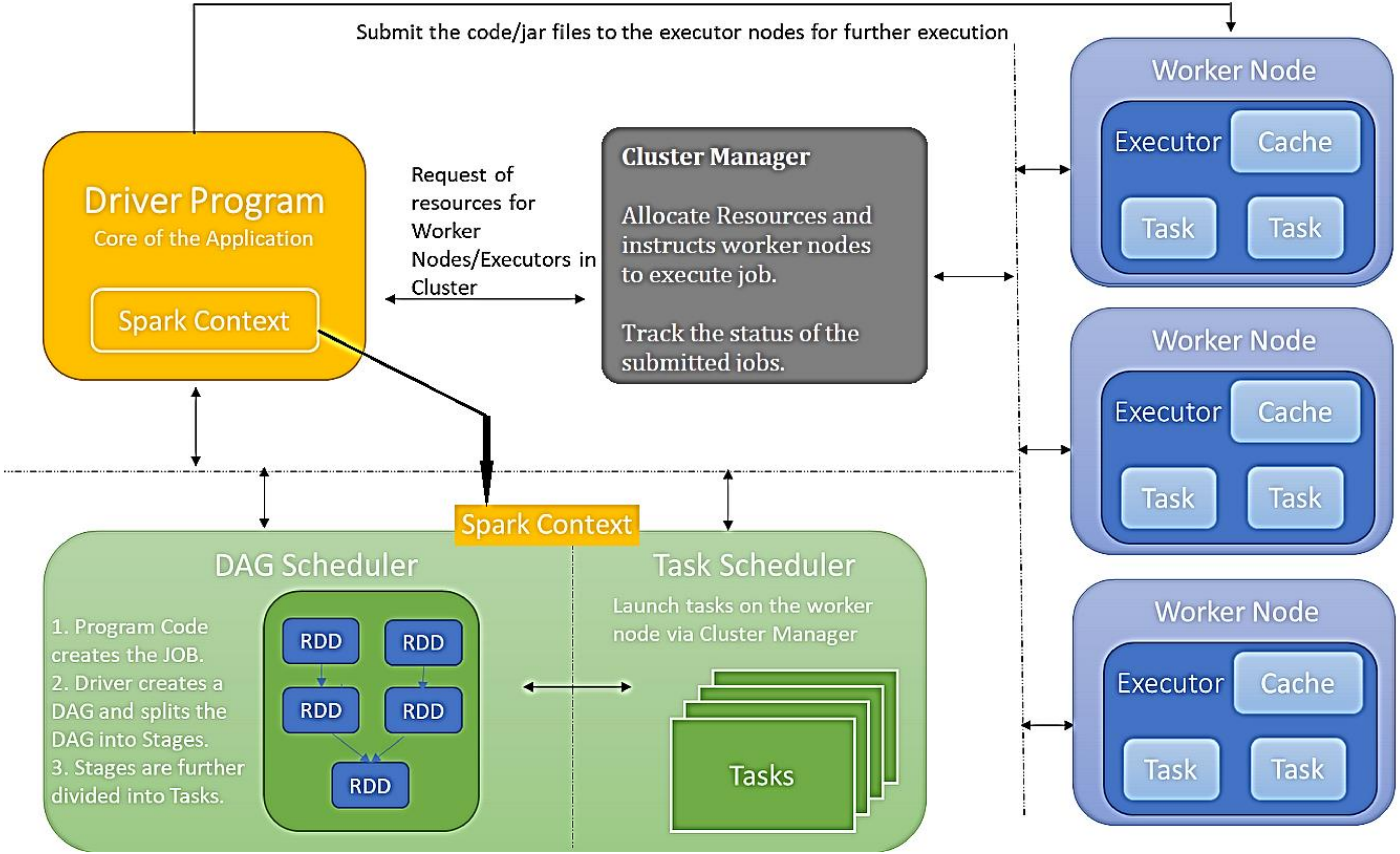


Separate Spark Session for each user and only one Spark Context



```
from pyspark import SparkConf, SparkContext
conf = SparkConf().setMaster("local").setAppName("FriendsByAge")
sc = SparkContext(conf = conf)
```

Internals of Job execution in Spark



Submit the code/jar files to the executor nodes for further execution

Request of resources for Worker Nodes/Executors in Cluster

Cluster Manager

Allocate Resources and instructs worker nodes to execute job.

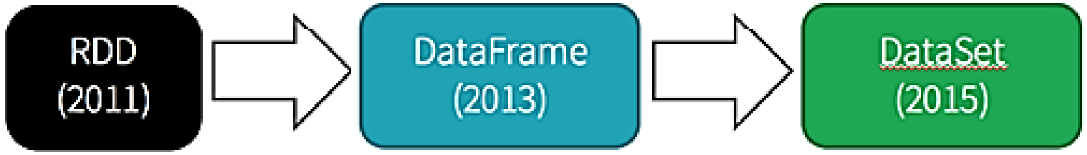
Track the status of the submitted jobs.

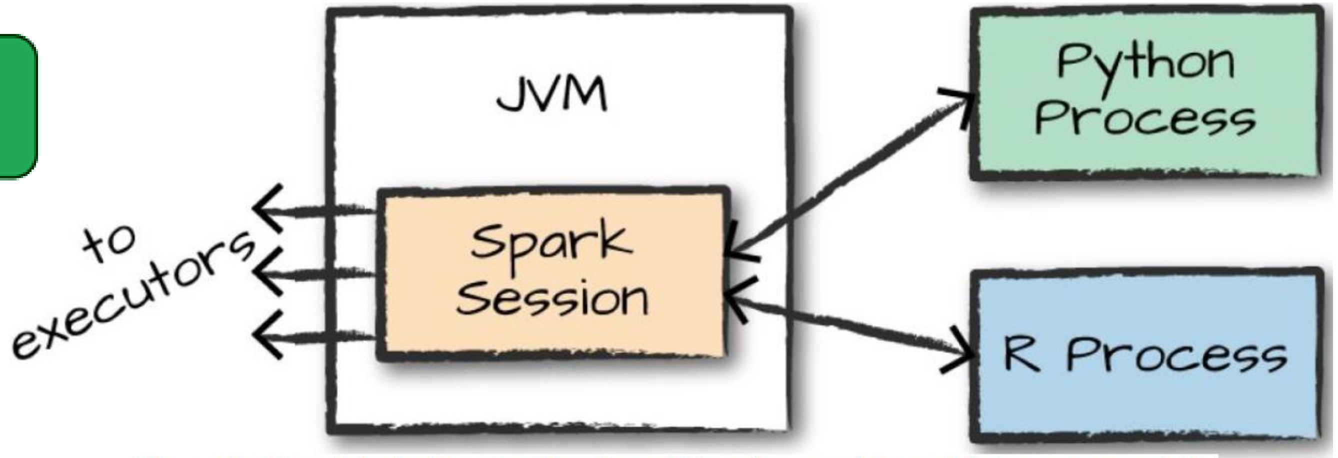
1. Program Code creates the JOB.
2. Driver creates a DAG and splits the DAG into Stages.
3. Stages are further divided into Tasks.

Launch tasks on the worker node via Cluster Manager

Spark’s Language APIs

- Spark’s language APIs make it possible for you to run Spark code using various programming languages.
- Spark presents some core “concepts” in every language and these concepts are translated into Spark code that runs on the cluster of machines.
- We can drive Spark from a variety of languages. (Python, Scala, Java, SQL, R…)
- Spark has two fundamental sets of APIs:
  - 1. The low-level “unstructured” APIs (RDD)
  - 2. The higher-level “structured” APIs. (Dataframes, Datasets)





The relationship between the SparkSession and Spark’s Language API

Dataframe (High level API)

- A DataFrame is the most common Structured API. It is a distributed collection of data, which is organized into named columns. It is equivalent to relational tables with good optimization techniques.

- You can think of a DataFrame as a spreadsheet with named columns.

- A “Spreadsheet” available on one computer while, Spark DataFrame can span thousands of computers.

- A DataFrame can be constructed from an array of different sources such as Hive tables, Structured Data files, external databases, or existing RDDs.

Dataframe (High level API)

Features of Dataframe

- Ability to process the data in the size of Kilobytes to Petabytes on a single node cluster to large cluster.

- Supports different data formats (Avro, csv, elastic search, and Cassandra) and storage systems (HDFS, HIVE tables, mysql, etc).

- State of art optimization and code generation through the Spark SQL Catalyst optimizer (tree transformation framework).

- Can be easily integrated with all Big Data tools and frameworks via Spark-Core.

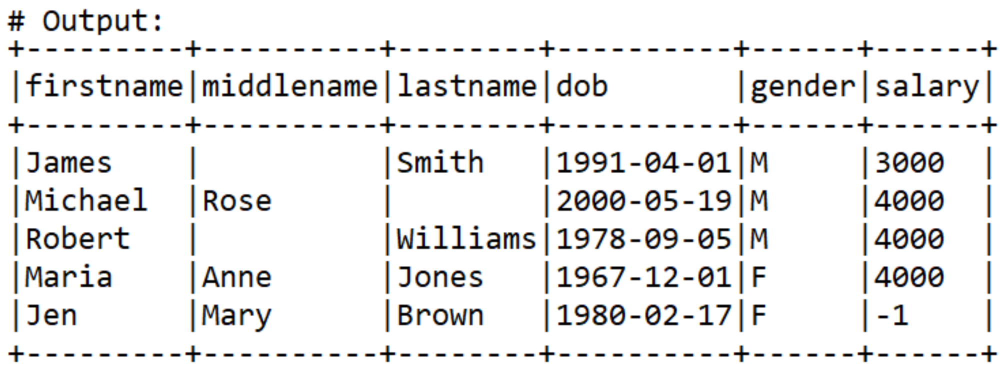
- Provides API for Python, Java, Scala, and R Programming.

Dataframe (High level API)

Example

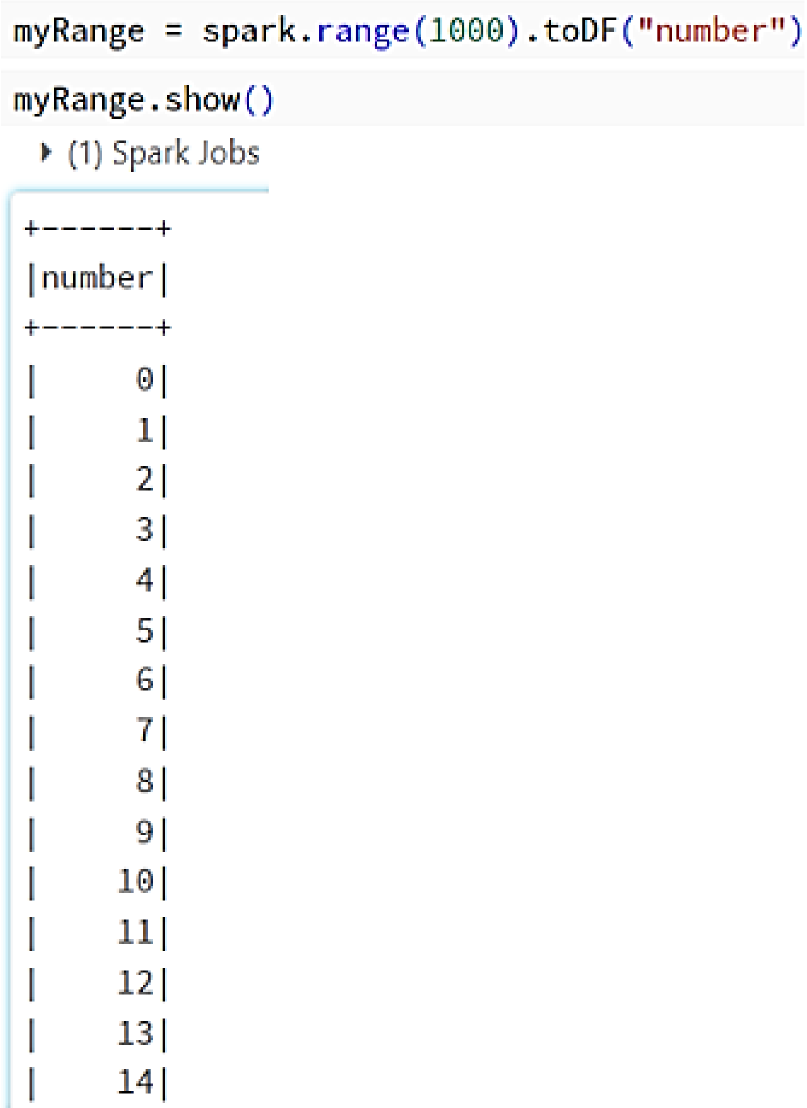
# Create DataFrame

```
data = [('James','','Smith','1991-04-01','M',3000),
            ('Michael','Rose','','2000-05-19','M',4000),
      ('Robert','','Williams','1978-09-05','M',4000),
      ('Maria','Anne','Jones','1967-12-01','F',4000),
              ('Jen','Mary','Brown','1980-02-17','F',-1)]
columns = ["firstname","middlename","lastname","dob","gender","salary"]
df = spark.createDataFrame(data=data, schema = columns)
df.show()
```



```
# Output:
+---------+----------+--------+----------+------+------+
|firstname|middlename|lastname|dob       |gender|salary|
+---------+----------+--------+----------+------+------+
|James    |          |Smith   |1991-04-01|M     |3000  |
|Michael  |Rose      |        |2000-05-19|M     |4000  |
|Robert   |          |Williams|1978-09-05|M     |4000  |
|Maria    |Anne      |Jones   |1967-12-01|F     |4000  |
|Jen      |Mary      |Brown   |1980-02-17|F     |-1    |
+---------+----------+--------+----------+------+------+
```

Dataframe (High level API)



```
myRange = spark.range(1000).toDF("number")
```

```
myRange.show()
```

```
+------+
|number|
+------+
|     0|
|     1|
|     2|
|     3|
|     4|
|     5|
|     6|
|     7|
|     8|
|     9|
|    10|
|    11|
|    12|
|    13|
|    14|
```

spark: This refers to the SparkSession, which is the entry point for using Spark functionality in PySpark. It's the main interface through which you interact with Spark.

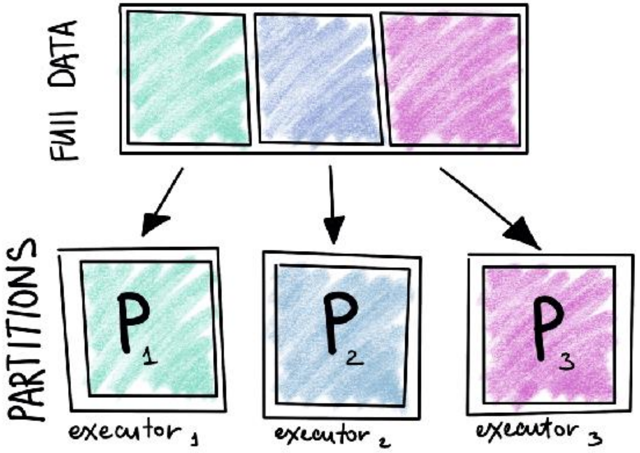
range(1000): This is a Python built-in function that generates a sequence of numbers from 0 to 999

spark.range(1000): In this context, the SparkSession's range() method is used to create a data frame with a single column named "id" (by default) containing the generated sequence of numbers from 0 to 999.

toDF("number"): The toDF() method converts the generated DataFrame into a new DataFrame with the specified column name. In this case, "number" is the name of the single column in the resulting DataFrame.

Partitions

- Partition is a logical division of the data , this idea is derived from Map Reduce (split). Logical data is specifically derived to process the data. Small chunks of data also it can support scalability and speed up the process. Input data, intermediate data, and output data everything is partitioned RDD.



- Spark uses the map-reduce API to do the partition the data. In input format, we can create a number of partitions. By default, HDFS block size is partition size (for best performance)

Partitions

- To allow every executor to perform work in parallel, Spark breaks up the data into chunks called *partitions*. A partition is a collection of rows data.
- A DataFrame’s partitions represent how the data is physically distributed across the cluster of machines during execution.
- If you have one partition, Spark will have a parallelism of only one, even if you have thousands of executors.
- If you have many partitions but only one executor, Spark will still have a parallelism of only one because there is only one computation resource.
- Once job submitted into the cluster, each partition is sent to a specific executor for further processing.
- Only one partition is processed by one executor at a time, so the size and number of partitions transferred to the executor are directly proportional to the time it takes to complete them. Thus the more partitions the more work is distributed to executors, with a smaller number of partitions the work will be done in larger pieces (and often faster).

RDD (Resilient Distributed Datasets)

- It is a fundamental data structure of spark. (Low level API). Main logical data units in spark.
- RDDs are stored in memory or on disks of different machines of a cluster and also allowing parallel processing.
- A single RDD can be divided into multiple logical partitions so that these partitions can be stored and processed on different machines of a cluster.
- It is fault-tolerant. In case of node fail, the RDD is capable of recovering automatically.
- RDDs are immutable (read-only) in nature. You cannot change an original RDD, but you can create new RDDs by performing transformations, on an existing RDD.
- RDD can be process both structured as well as unstructured data.
- RDD provides an OOP-style API, and here we tell the Spark engine "How to do" basically, how to achieve any particular task.

When to Use the Low-Level APIs?

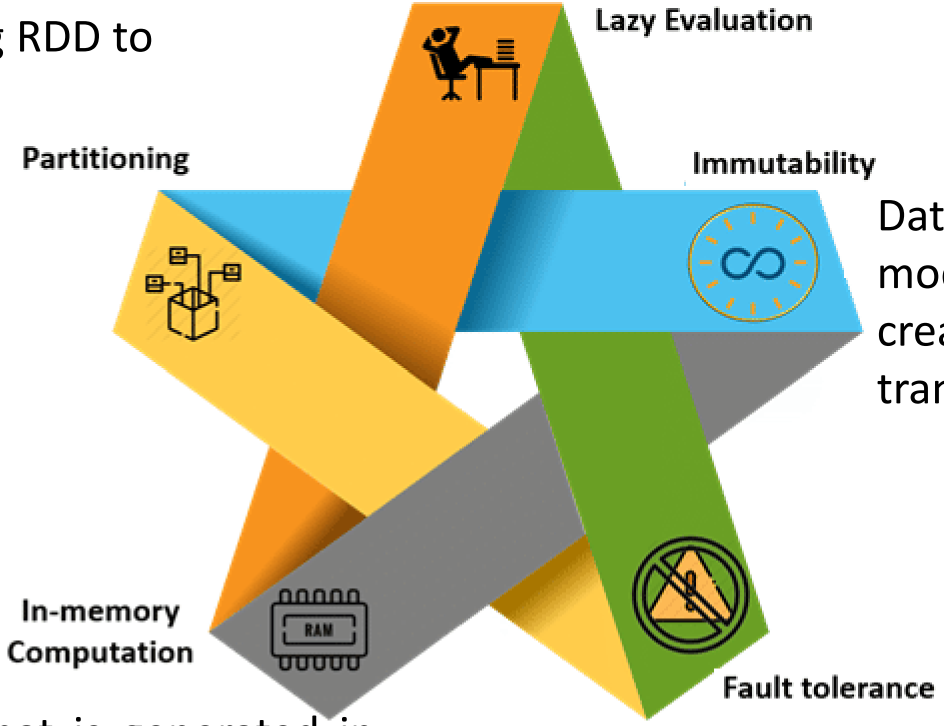
You should generally use the lower-level APIs in three situations:

- You need some functionality that you cannot find in the higher-level APIs; for example, if you need very tight control over physical data placement across the cluster.
- You need to maintain some legacy codebase written using RDDs.
- You need to do some custom shared variable manipulation.

Features of an RDD in Spark

Data does not get loaded in an RDD even if you define it. Transformations are actually computed when you call action, such as count or collect, or save the output to a file system.

Partitions can be done on any existing RDD to create logical parts that are mutable.



Data stored in an RDD is in the read-only mode. You cannot edit it. But, you can create new RDDs by performing transformations on the existing RDDs.

An RDD stores any immediate data that is generated in the memory (RAM) than on the disk so that it provides faster access.

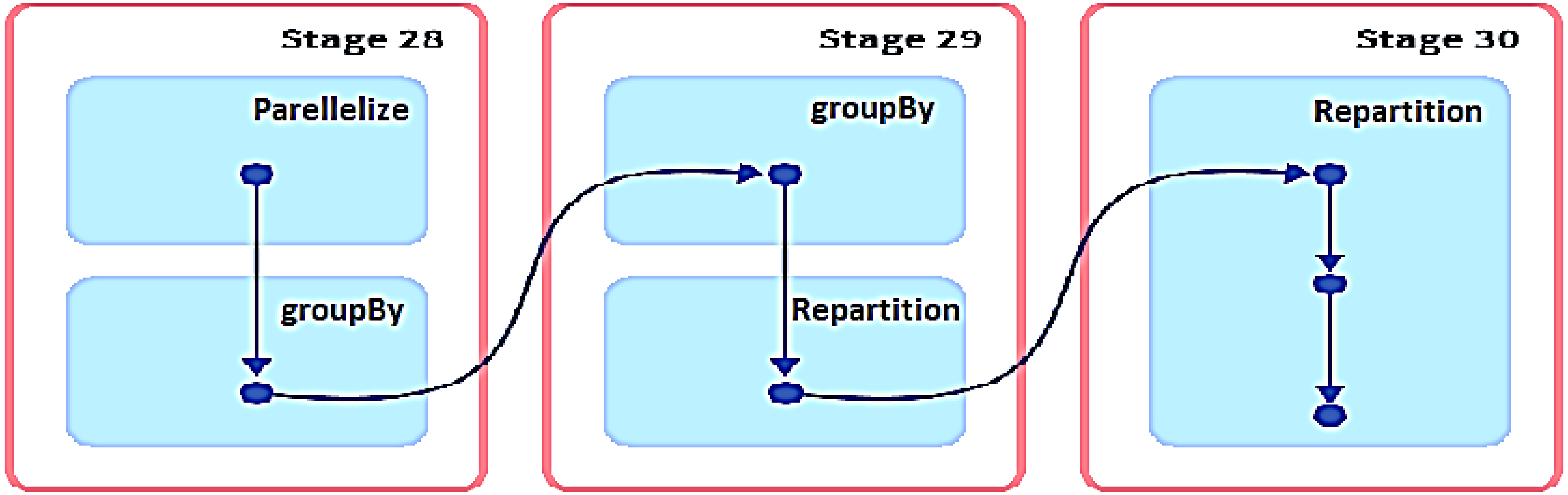
During RDD operations, if any node fail then In that case, the RDD is capable of recovering automatically.

Limitation of RDD

- There is no input optimization available in RDDs
- One of the biggest limitations of RDDs is that the execution process does not start instantly.
- No changes can be made in RDD once it is created.
- RDD lacks enough storage memory.
- The run-time type safety is absent in RDDs.

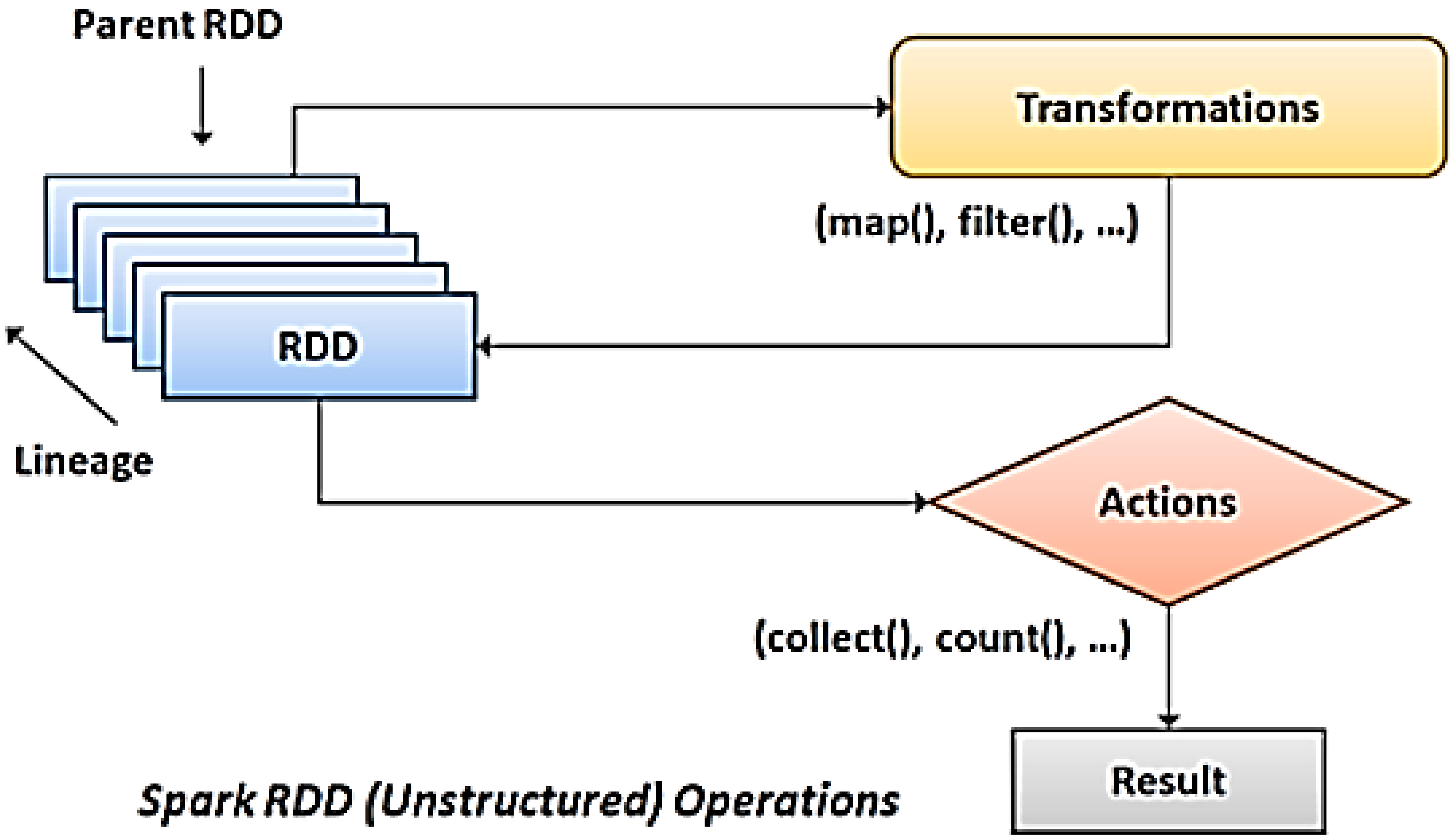
Most essential part of Apache Spark: Spark RDD and DAG

- **DAG (**Directed Acyclic Graph**):**
  - DAG in Spark is a fundamental concept that plays a crucial role in the Spark execution model.
  - DAG is “directed” because the operations are executed in a specific order.
  - DAG is “acyclic” because there are no loops or cycles in the execution plan. This means that each stage depends on the completion of the previous stage, and each task within a stage can run independently of the other.



Two main functions of Spark

- Spark performs two main functions called **transformations** and **actions**.

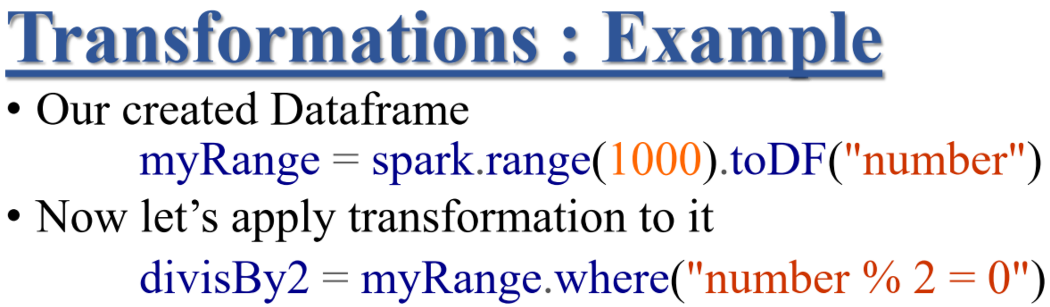


Spark RDD (Unstructured) Operations

Two main functions of Spark

Transformations

- In Spark, the core data structures(RDD) are immutable, cannot be changed after created.
- To “change” a RDD, you need to instruct Spark how you would like to modify it to do what you want.
- These instructions are called **transformations.**



Transformations : Example

- Our created Dataframe

```
myRange = spark.range(1000).toDF("number")
```

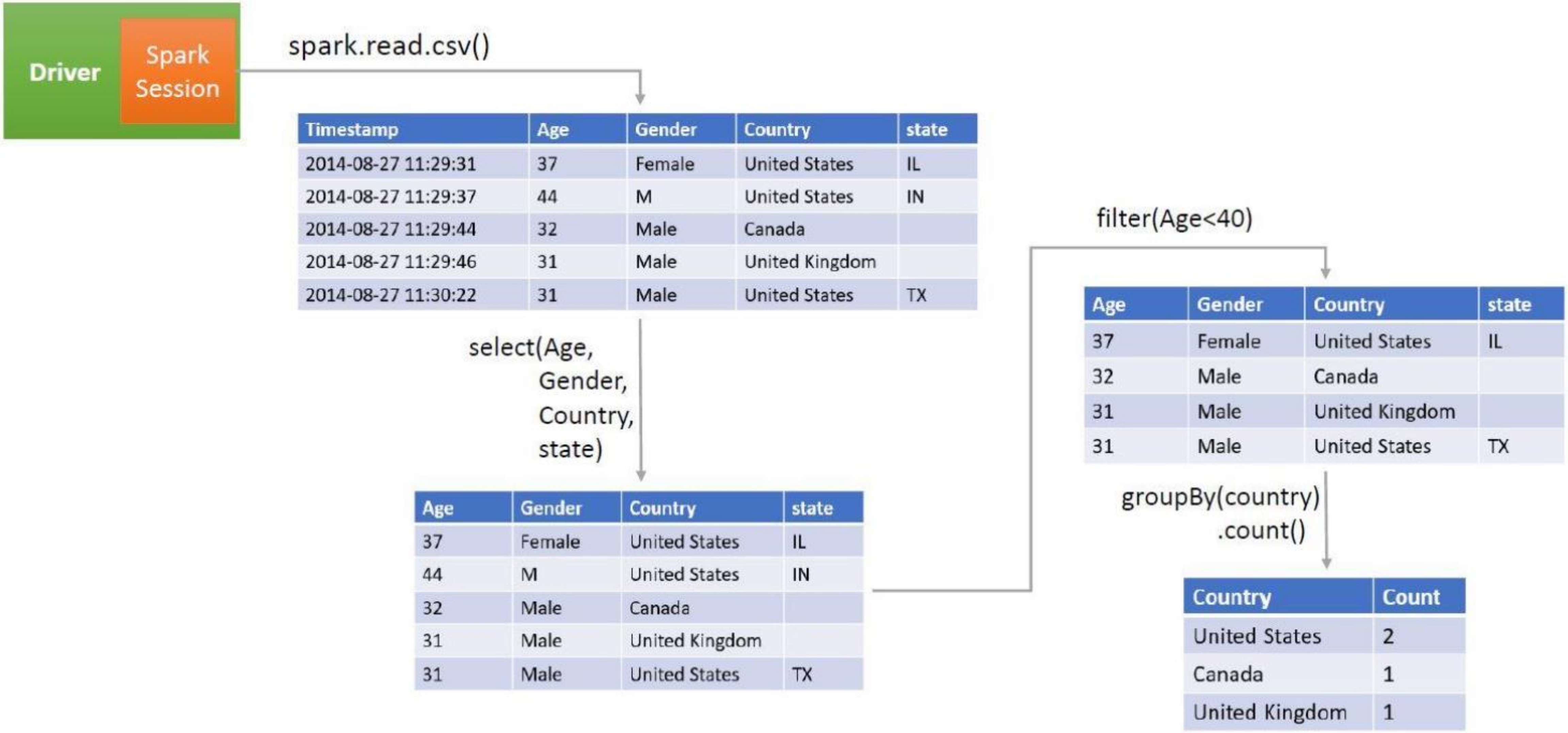
- Now let’s apply transformation to it

```
divisBy2 = myRange.where("number % 2 = 0")
```

- Notice that these return no output. because we specified only an abstract transformation, and Spark will not act on transformations until we call an “action”
- There are two types of transformations:
  - 1. Narrow transformations
  - 2. Wide transformations

Two main functions of Spark

Transformations



spark.read.csv()

Timestamp	Age	Gender	Country	state
2014-08-27 11:29:31	37	Female	United States	IL
2014-08-27 11:29:37	44	M	United States	IN
2014-08-27 11:29:44	32	Male	Canada	
2014-08-27 11:29:46	31	Male	United Kingdom	
2014-08-27 11:30:22	31	Male	United States	TX

select(Age, Gender, Country, state)

Age	Gender	Country	state
37	Female	United States	IL
44	M	United States	IN
32	Male	Canada	
31	Male	United Kingdom	
31	Male	United States	TX

filter(Age<40)

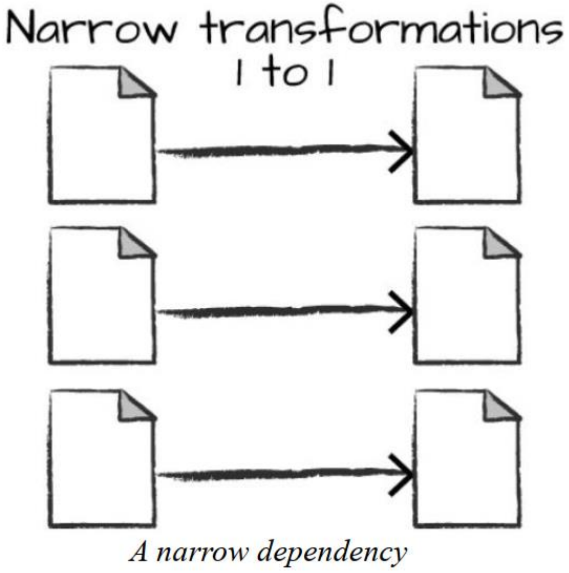
Age	Gender	Country	state
37	Female	United States	IL
32	Male	Canada	
31	Male	United Kingdom	
31	Male	United States	TX

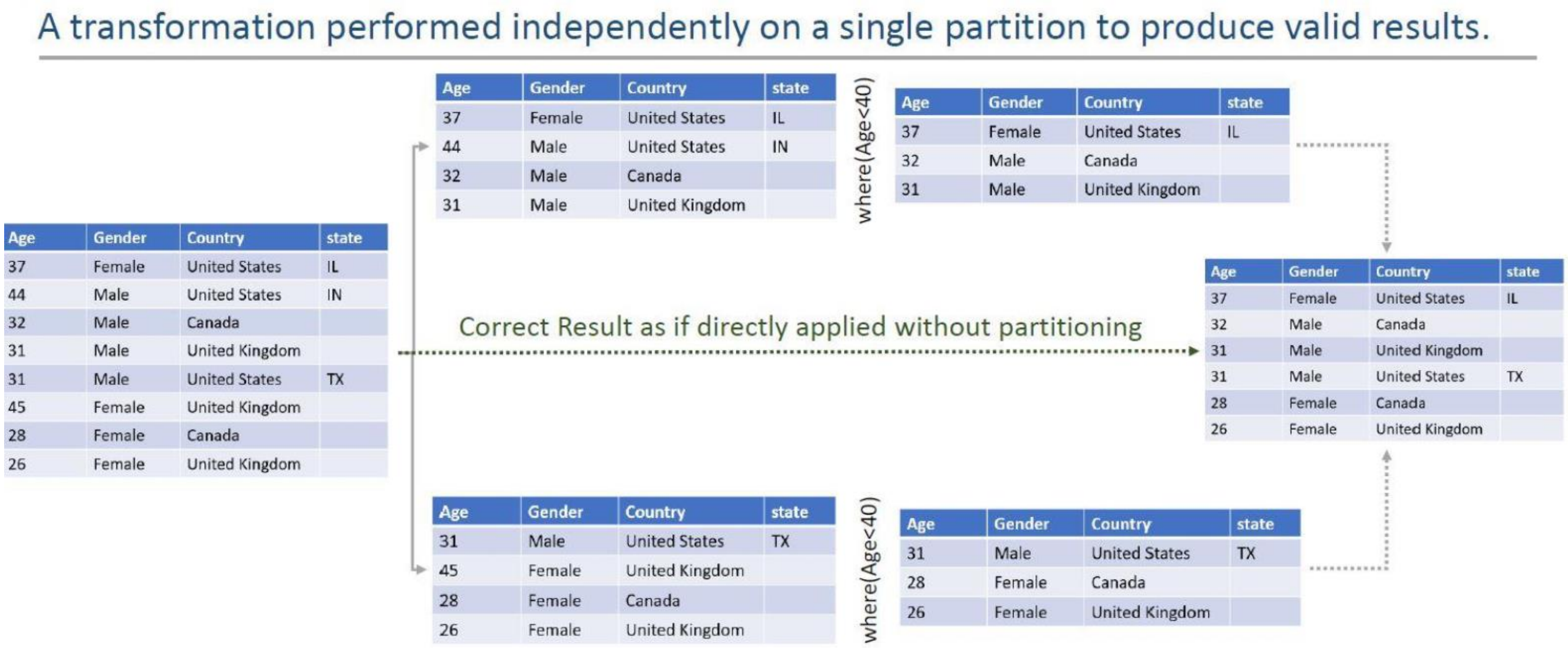
groupBy(country).count()

Country	Count
United States	2
Canada	1
United Kingdom	1

Two main functions of Spark

Narrow Transformations




A transformation performed independently on a single partition to produce valid results.

Age	Gender	Country	state
37	Female	United States	IL
44	Male	United States	IN
32	Male	Canada	
31	Male	United Kingdom	
31	Male	United States	TX
45	Female	United Kingdom	
28	Female	Canada	
26	Female	United Kingdom	

Age	Gender	Country	state
37	Female	United States	IL
44	Male	United States	IN
32	Male	Canada	
31	Male	United Kingdom	

Age	Gender	Country	state
37	Female	United States	IL
32	Male	Canada	
31	Male	United Kingdom	

Age	Gender	Country	state
31	Male	United States	TX
45	Female	United Kingdom	
28	Female	Canada	
26	Female	United Kingdom	

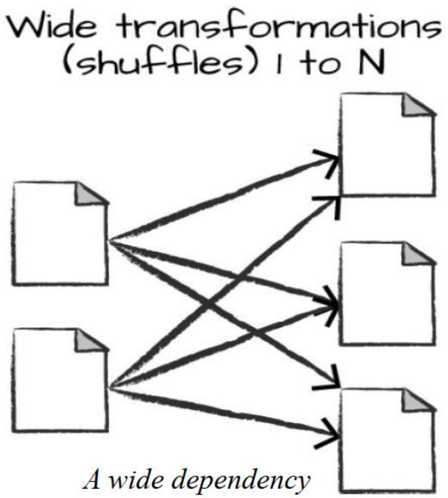
Age	Gender	Country	state
31	Male	United States	TX
28	Female	Canada	
26	Female	United Kingdom	

Age	Gender	Country	state
37	Female	United States	IL
32	Male	Canada	
31	Male	United Kingdom	
31	Male	United States	TX
28	Female	Canada	
26	Female	United Kingdom	

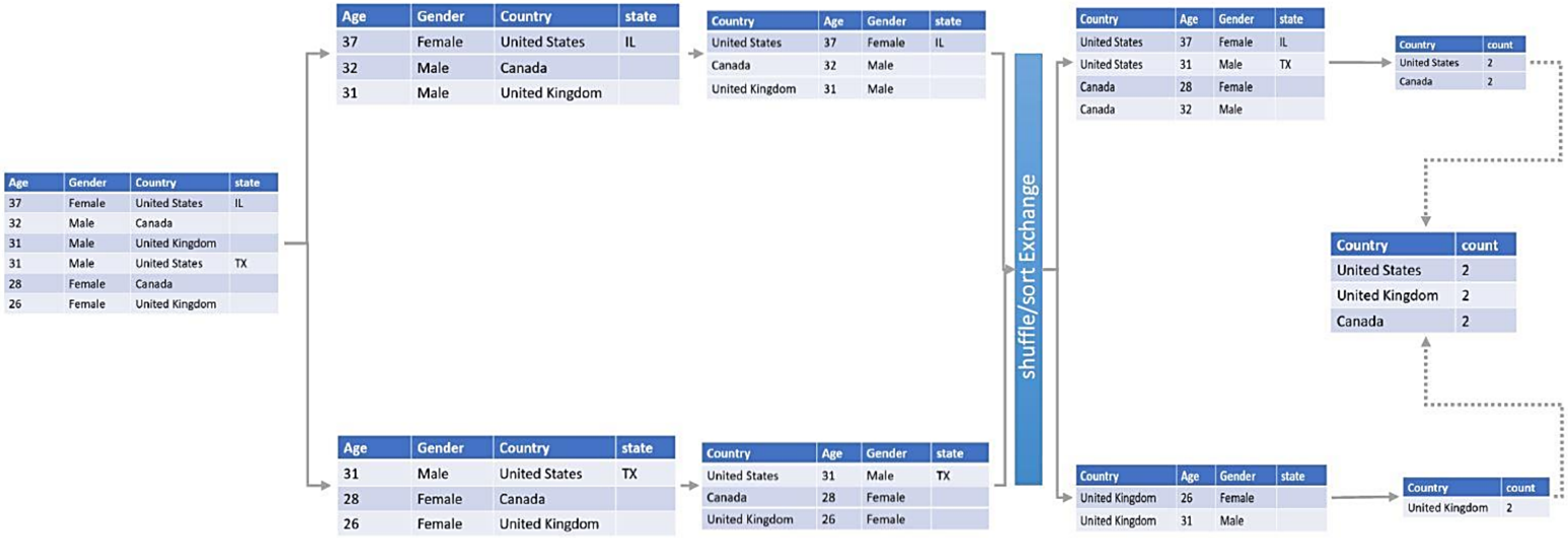
Examples of Narrow Transformations: map(), filter(), union(), coalesce(), repartition()

Two main functions of Spark

Wide Transformations



A transformation that requires data from other partitions to produce valid results.



Age	Gender	Country	state
37	Female	United States	IL
32	Male	Canada	
31	Male	United Kingdom	
31	Male	United States	TX
28	Female	Canada	
26	Female	United Kingdom	

Age	Gender	Country	state
37	Female	United States	IL
32	Male	Canada	
31	Male	United Kingdom	

Country	Age	Gender	state
United States	37	Female	IL
Canada	32	Male	
United Kingdom	31	Male	

Age	Gender	Country	state
31	Male	United States	TX
28	Female	Canada	
26	Female	United Kingdom	

Country	Age	Gender	state
United States	31	Male	TX
Canada	28	Female	
United Kingdom	26	Female	

Country	Age	Gender	state
United States	37	Female	IL
United States	31	Male	TX
Canada	28	Female	
Canada	32	Male	

Country	Age	Gender	state
United Kingdom	26	Female	
United Kingdom	31	Male	

Country	count
United States	2
Canada	2

Country	count
United Kingdom	2

Country	count
United States	2
United Kingdom	2
Canada	2

Examples of Wide Transformations: groupByKey(), reduceByKey(), join(), cogroup(), distinct()

Transformations in RDD

Some of the transformation operations are provided in the table below:

Function	**Description**
map()	Returns a new RDD by applying the function on each data element
filter()	Returns a new RDD formed by selecting those elements which the function returns true
reduceByKey()	Aggregates the values of a key using a function
groupByKey()	Converts a (key, value) pair into a (key, <iterable value>) pair
union()	Returns a new RDD that contains all elements and arguments from the source RDD
intersection()	Returns a new RDD that contains an intersection of the elements in the datasets

Actions in RDD

- Actions in Spark are functions that return the end result of RDD computations.
- It uses a lineage graph to load data onto the RDD in a particular order.
- After all of the transformations are done, actions return the final result to the Spark Driver.
- Actions are operations that provide non-RDD values.
- Some of the common actions used in Spark are given below:

Function	**Description**
count()	Gets the number of data elements in an RDD
collect()	Gets all the data elements in an RDD as an array
reduce()	Aggregates data elements into an RDD by taking two arguments and returning one
take(n)	Fetches the first *n* elements of an RDD
foreach(operation)	Executes the operation for each data element in an RDD
first()	Retrieves the first data element of an RDD

Spark: Job, Stage, and Task

Job

- A job in Spark refers to a sequence of transformations on data.
- Whenever an action like count(), first(), collect(), and save() is called on RDD (Resilient Distributed Datasets), a job is created.
- Job could be thought of as the total work that your Spark application needs to perform.

Stage

- A stage in Spark represents a sequence of transformations that can be executed in a single pass, i.e., without any shuffling of data.
- When a job is divided, it is split into stages. Each stage comprises tasks, and all the tasks within a stage perform the same computation.
- The boundary between two stages is drawn when transformations cause data shuffling across partitions.
- Narrow transformations (**map(), filter(),** and **union()**), can be done within a single partition. Wide transformations (**groupByKey(), reduceByKey(),** or **join()**), can be done in different partitions (so, working in stages).

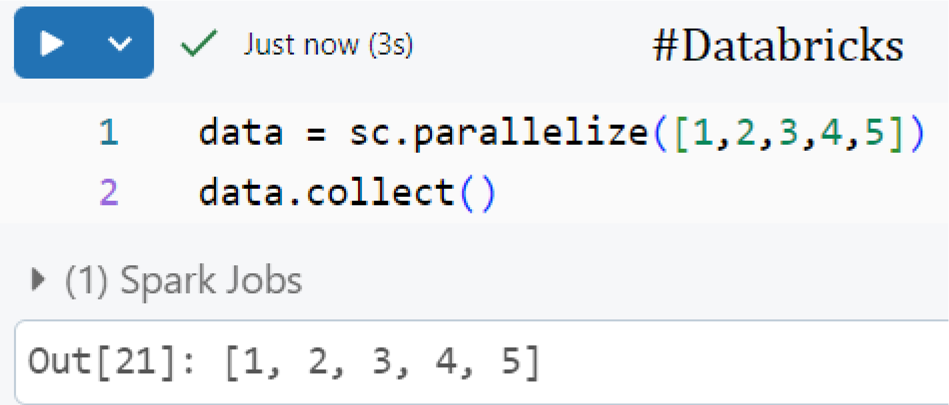
Spark: Job, Stage, and Task

Task

- A task in Spark is the smallest unit of work that can be scheduled.
- Each stage is divided into tasks.
- A task is a unit of execution that runs on a single machine. When a stage comprises transformations on an RDD, those transformations are packaged into a task to be executed on a single executor.
- For example, if you have a Spark job that is divided into two stages and you're running it on a cluster with two executors, each stage could be divided into two tasks. Each executor would then run a task in parallel, performing the transformations defined in that task on its subset of the data.

Spark RDD

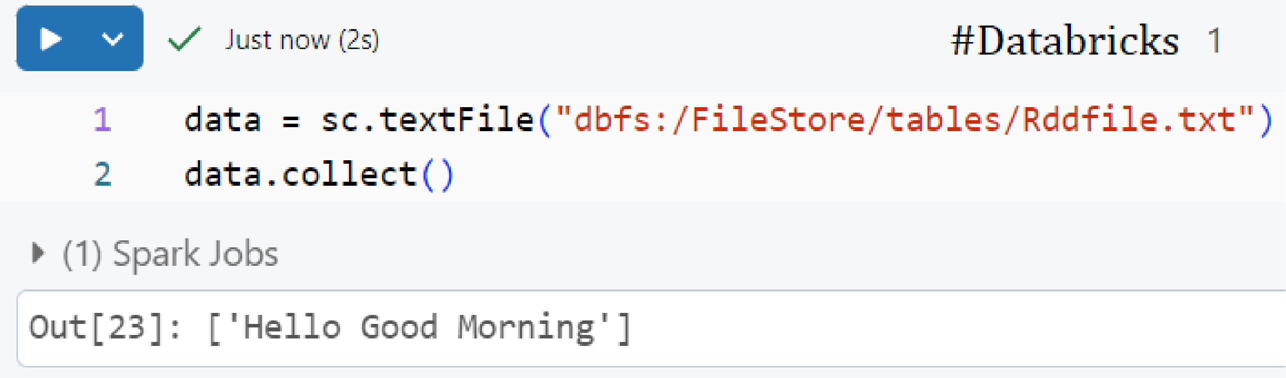
- **There are 3 ways of creating an RDD:**
  - Parallelizing an existing collection of data



```
data = sc.parallelize([1,2,3,4,5])
data.collect()
```

```
Out[21]: [1, 2, 3, 4, 5]
```

  - Referencing to the external data file stored

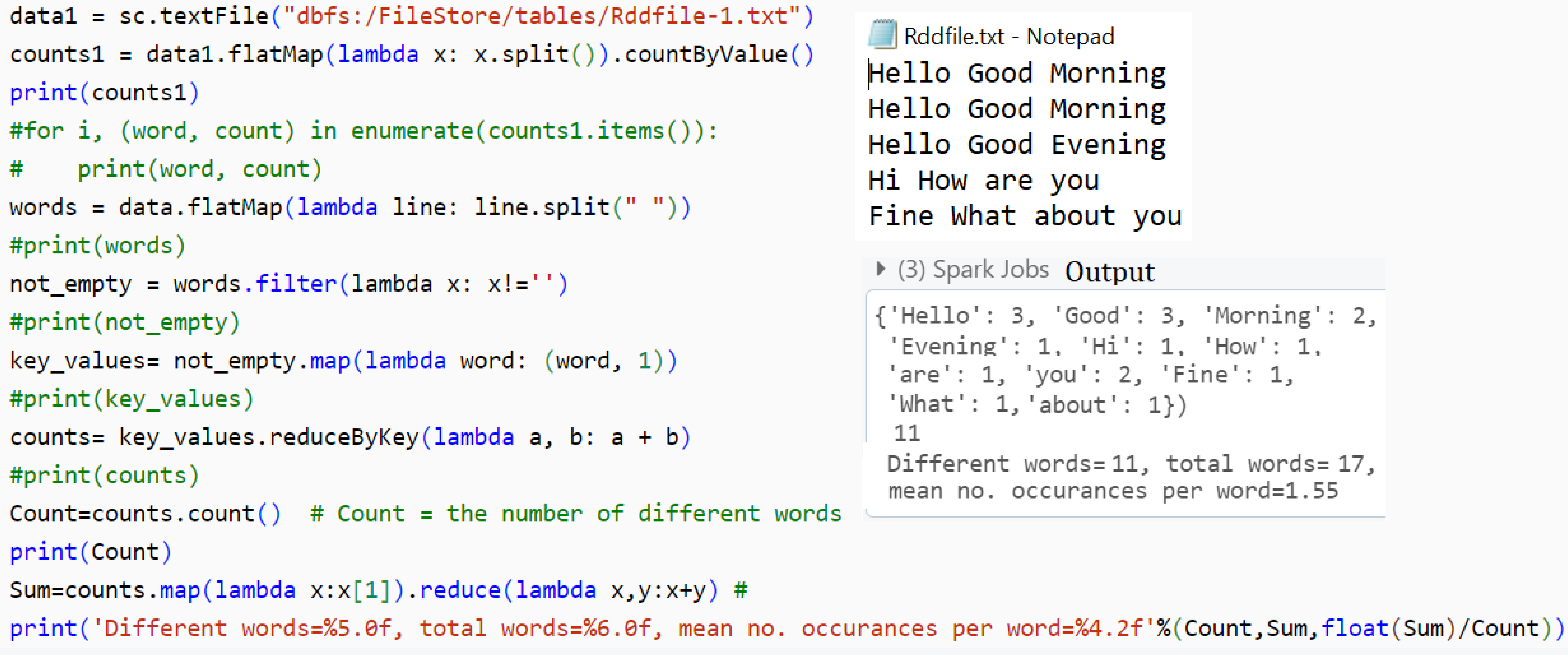


```
data = sc.textFile("dbfs:/FileStore/tables/Rddfile.txt")
data.collect()
```

```
Out[23]: ['Hello Good Morning']
```

Spark RDD

- **There are 3 ways of creating an RDD:**
    - Creating RDD from an already existing RDD



```
data1 = sc.textFile("dbfs:/FileStore/tables/Rddfile-1.txt")
counts1 = data1.flatMap(lambda x: x.split()).countByValue()
print(counts1)
#for i, (word, count) in enumerate(counts1.items()):
#    print(word, count)
words = data.flatMap(lambda line: line.split(" "))
#print(words)
not_empty = words.filter(lambda x: x!='')
#print(not_empty)
key_values= not_empty.map(lambda word: (word, 1))
#print(key_values)
counts= key_values.reduceByKey(lambda a, b: a + b)
#print(counts)
Count=counts.count()  # Count = the number of different words
print(Count)
Sum=counts.map(lambda x:x[1]).reduce(lambda x,y:x+y) #
print('Different words=%5.0f, total words=%6.0f, mean no. occurances per word=%4.2f'%(Count,Sum,float(Sum)/Count))
```

Rddfile.txt - Notepad

```
Hello Good Morning
Hello Good Morning
Hello Good Evening
Hi How are you
Fine What about you
```

▸ (3) Spark Jobs **Output**

```
{'Hello': 3, 'Good': 3, 'Morning': 2,
 'Evening': 1, 'Hi': 1, 'How': 1,
 'are': 1, 'you': 2, 'Fine': 1,
 'What': 1,'about': 1})
 11
 Different words=11, total words=17,
 mean no. occurances per word=1.55
```

Spark RDD

- **When to use RDD?**
  - Data is unstructured.
  - Transformations are low-level.
  - Data manipulation should be fast and straightforward when nearer to the data source.
  - Schema is unimportant. Since RDDs do not impose schemas, use them when accessing specific data by column or attribute is not relevant.

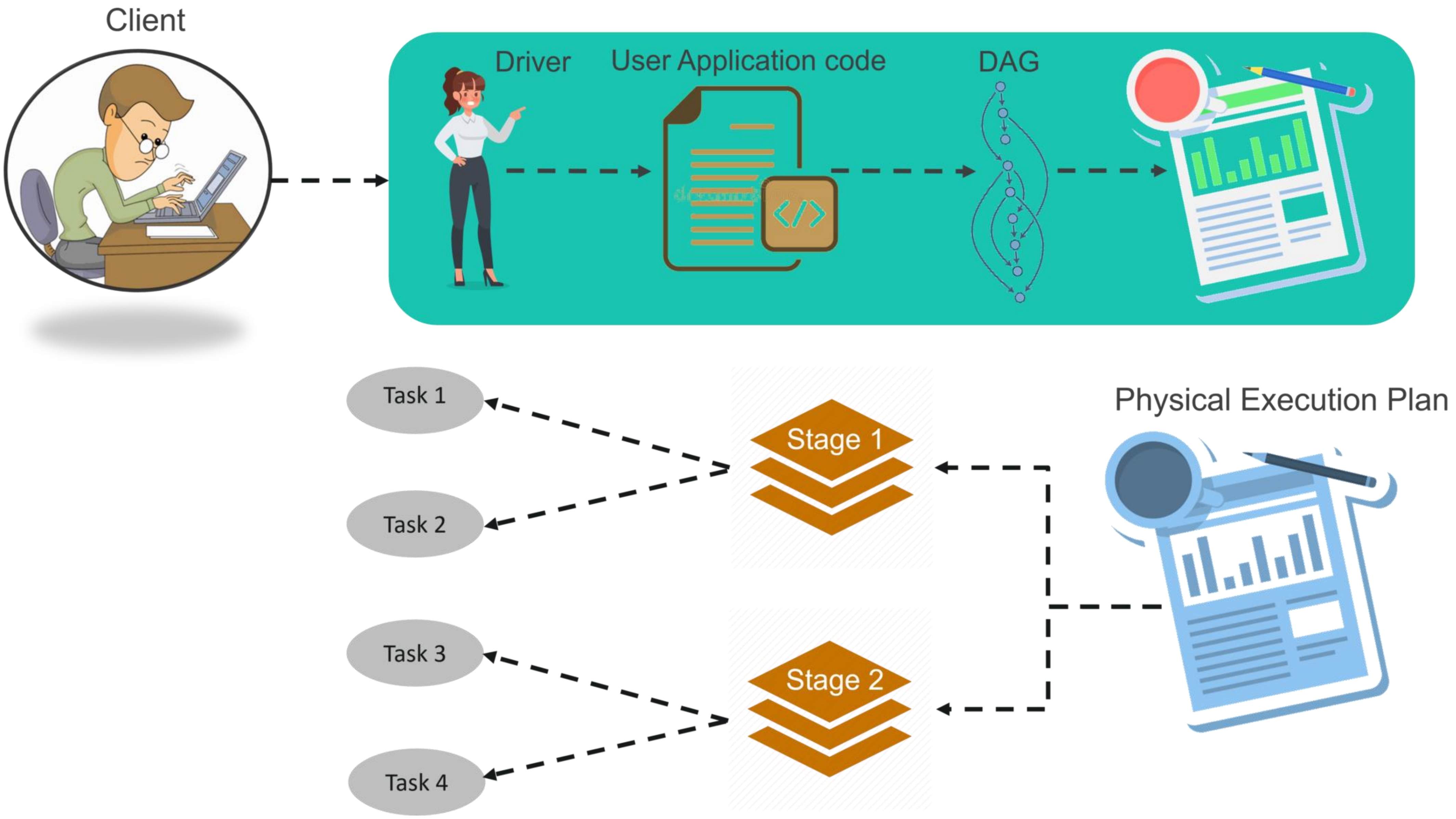
DataFrames / Datasets

- DataFrames used when dealing with structured data, similar to an SQL table, to improve performance through optimized execution plans and built-in optimizations.

- DataFrames provides SQL style API and here, we tell spark engine "What to do" and, spark engine will use optimization through the Spark SQL Catalyst optimizer to achieve the cost-effective way to accomplish the task.

- Datasets: These are the newest data abstraction provided by spark. It is the combination of best of dataframes and best of datasets. It possess best of RDDs(OOP style + type safety) and best of Dataframes(structured format+optimization+memory management).

Workflow of Spark Architecture



Workflow of Spark Architecture

- **STEP 1:** The client submits spark user application code. When an application code is submitted, the driver implicitly converts user code that contains transformations and actions into a logically *directed acyclic graph* called ***DAG.*** At this stage, it also performs optimizations such as pipelining transformations.

- **STEP 2:** After that, it converts the logical graph called DAG into physical execution plan with many stages. After converting into a physical execution plan, it creates physical execution units called tasks under each stage. Then the tasks are bundled and sent to the cluster.

- **STEP 3:** Now the driver talks to the cluster manager and negotiates the resources. Cluster manager launches executors in worker nodes on behalf of the driver. At this point, the driver will send the tasks to the executors based on data placement. When executors start, they register themselves with drivers. So, the driver will have a complete view of executors that are executing the task.

What is SparkContext?

- Since Spark 1.x, SparkContext is an entry point to Spark and is defined in org.apache.spark package.

- It is used to programmatically create Spark RDD, accumulators, and broadcast variables on the cluster.

- Its object “sc” is default variable available in spark-shell and it can be programmatically created using SparkContext class.

- NOTE: you can create only one active SparkContext per JVM. You should stop() the active SparkContext before creating a new one.
- The Spark driver program creates and uses SparkContext to connect to the cluster manager to submit Spark jobs, and know what resource manager (YARN, Mesos or Standalone) to communicate to. It is the heart of the Spark application.

Spark Example

- Social Network Dataset is in the form (id,name,age,number_of_friends): fakefriends.csv

```
0,Will,33,385
1,Jean-Luc,26,2
2,Hugh,55,221
3,Deanna,40,465
4,Quark,68,21
5,Weyoun,59,318
6,Gowron,37,220
7,Will,54,307
8,Jadzia,38,380
9,Hugh,27,181
10,Odo,53,191
...
```

Spark Example

```
# Create SparkSession from builder (run in Jupyter Lab)

import findspark
findspark.init()

from pyspark.sql import SparkSession
spark=SparkSession.builder.master("local").appName('TestSpark').getOrCreate()
sc=spark.sparkContext

spark
```

Output:

```
SparkSession - in-memory
SparkContext
Spark UI
Version: v3.3.0 Master: local AppName: TestSpark
```

Spark Example

```
def parseLine(line):
    fields = line.split(',')
    age = int(fields[2])
    numFriends = int(fields[3])
    return (age, numFriends)

lines = sc.textFile("C:\\fakefriends.csv")
rdd = lines.map(parseLine)

groupByAge = rdd.mapValues(lambda x: (x,1))
totalsByAge = groupByAge.reduceByKey(lambda x, y: (x[0] + y[0], x[1] + y[1]))
averagesByAge = totalsByAge.mapValues(lambda x: x[0] / x[1])
results = averagesByAge.collect()
for result in results:
    print result
```