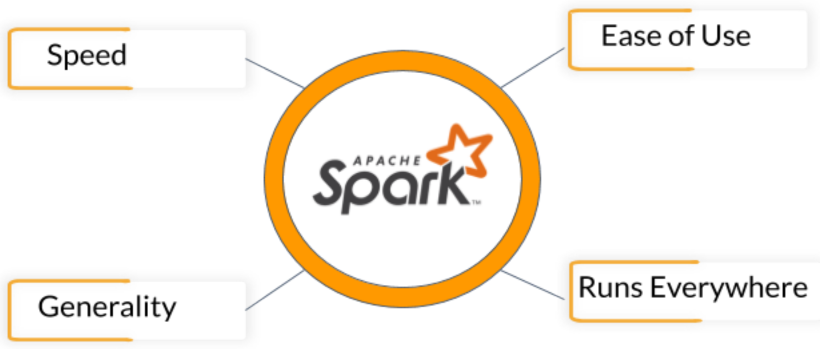
Module 01: Introduction to Apache Spark

I. Getting started with Apache Spark

▼ Spark Overview

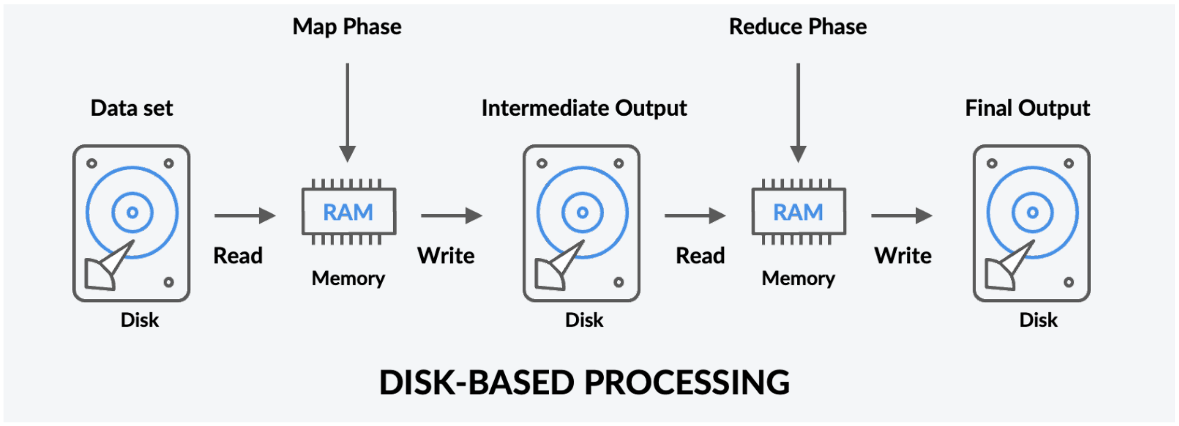
- **Apache Spark** is a multi-language engine for executing data engineering, data science, and machine learning on single-node machines or clusters**.**" as defined on the company's website. It is an **open-source, distributed computing engine** that provides a productive environment for data analysis owing to its lightning speed and support for various libraries.
- Spark use-cases:
    - To perform exploratory analysis on data sets in the scale of hundreds of GBs (or even TBs) within a realistic time frame.
    - To run near real-time reports from streaming data.
    - To develop machine learning models.
- **4 main features of Spark:**



- **Pro tip:**
    - Do not compare Spark to HDFS. Spark is a data processing layer, whereas HDFS is a data storage layer.
    - Spark was invented at the Algorithms, Machines, and People (AMP) Lab at the University of California Berkeley as an alternative to the MapReduce paradigm.
    - Spark can use HDFS as its data storage layer.

▼ Disk-based vs Memory-based

- Storage systems are primarily of the following two types:
    - **Memory, and**
    - **Disk.**
- **MapReduce follows disk-based processing systems**



- In the Map phase, first, data is processed and then split into **partitions** (mapped). The output of the Map phase is again transferred to the disk as intermediate output. The same output acts as input for the Reduce phase. The data is again read from the disk to the memory in the Reduce phase. The output of the Reduce phase is then stored in the disk.
- Two stages, Map and Reduce, are not performed together in memory. There is an intermediate output that is stored in the disk. This back-and-forth data movement between the disk and the memory creates an overhead and slows down the entire data processing cycle.

- Access time for retrieving a file from a disk
    - **Seek time**: This is the time taken by the read/write head to move from the current position to a new position.
    - **Rotational delay**: This is the time taken by the disk to rotate so that the read/write head points to the beginning of the data chunk.
    - **Transfer time**: This is the time taken to read/write the data from/to the hard disk to/from the main memory.
    - **Access Time = Seek time + Rotational delay + Transfer time**
- Why and Why not use Disk-based processing system:
    - Seek time delay analysis
        - Since hard disk always retrieves data sequentially from very start of the data chunk, the disk has to rotate so that read/write head points to the the start of chunk
        - If retrieve data randomly, it will take lots of time to access data
        - Big Files → Less time ← → Small files → More time
    - Why:
        - Suitable for storing and managing large amounts of data.
        - Hard drive → safer and fault tolerant
    - Why not
        - I/O consumes lots of job’s run time
        - Cannot use for real-time data for immediate result
- Why In-Memory Processing
    - **Real-time data processing:** Since data can be accessed very fast, in-memory processing can be used in cases where immediate results are required.
    - **Accessing data randomly in memory:** Since data is stored in the RAM, memory can be accessed randomly without scanning the entire storage.
    - **Iterative and interactive operations:** Intermediate results are stored in memory and not in disk storage so that we can use this output in other computations.

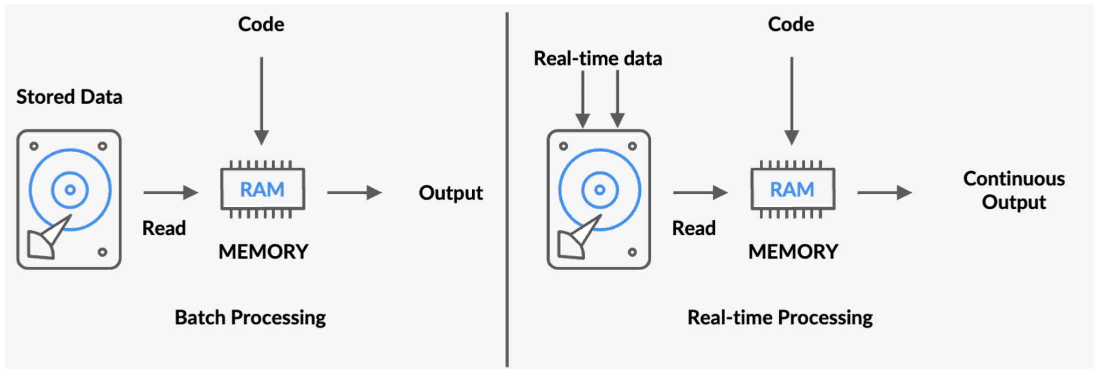
▼ Spark vs MapReduce

- Iterative Operations in MapReduce
    - Data in HDFS
    - Map: Read data → Memory
    - Reduce: Data back in HDFS

  ⇒ Multiple jobs have write back time (cost 60-70% for only I/0 time)

  ⇒ **N queries require n times of I/0**

- Interative Operations in Spark



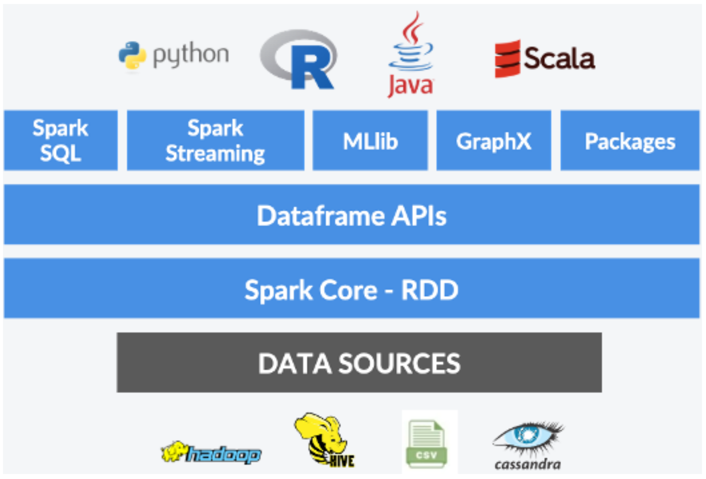
    - **Batch processing:** data is collected over a period of time and is available for analysis in a batch. Output from such tasks is not expected instantly, and, hence, you can work without a high-speed processing framework. An example of batch processing can be processing credit card bills at the end of every month.
    - **Real-time processing:** data is collected and processed continuously. Typically, for such tasks, the system needs to produce output in a very short span of time, sometimes almost instantly. As an example, payment systems such as PayPal, credit card companies like Visa, and many others use fraud detection algorithms to identify suspicious transactions.

  ⇒ **N queries, try to read once and store in memory, reduce I/0 times**

- Spark vs MapReduce

MapReduce	Spark
Disk-based → Slow	In-memory ⇒ Fast
Only batch processing	Batch + Real-time processing
Storage: HDFS	Storage: various, such as local files, HDFS, S3, etc
Language: Java, C++, Ruby, Groovy, Perl using Hadoop streaming library	Language: Scala, Java, Python, R, SQL
Raw API, not much support	Rich APIs

▼ Spark Ecosystem



- **Spark Core - RDD:**
  - Spark Core is the heart of Spark and is responsible for all kinds of processing.
  - Everything in Spark, including Spark API, is built on top of Spark Core, which provides an execution platform for every other Spark API.
  - The Spark Core engine executes all Spark jobs with RDDs as inputs.
  - Even if you use any high-level API for optimizing your Spark tasks, Spark handles the data internally in the distributed environment using RDDs. You will learn about Spark RDDs in upcoming sessions.
  - Here is a glimpse of RDD storage and representation of data:

```
[['U121', 'Mark', 'USA'], ['U150', 'Pete
r', 'UK'], ['U140', 'John', 'Germany'],
['U110', 'Roger', 'France'], ['U90', 'Raf
ael', 'Italy'], ['U50', 'Sam', 'Sweden']]
```

- **Dataframe API:**
  - When dealing with a structured form of data, you will be using the Dataframe API.
  - Dataframe API stores the data in a table structure. You will learn more about the different data types in the next module.
  - Let us have a glance at Dataframe storage and representation of data.

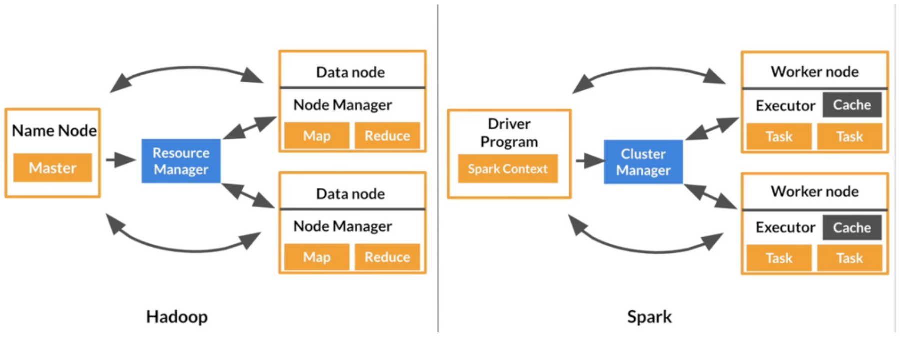
```
+-------+---------+-----------+
|User ID|User Name|User Region|
+-------+---------+-----------+
|   U121|     Mark|        USA|
|   U150|    Peter|         UK|
|   U140|     John|    Germany|
|   U110|    Roger|     France|
|    U90|   Rafael|      Italy|
|    U50|      Sam|     Sweden|
+-------+---------+-----------+
```

- **SparkSQL:**
  - **Spark SQL** is Apache Spark's module for working with structured data.
  - It is a high-level API that lets you utilize Spark's power using SQL queries.
  - You do not have to modify your SQL query code. Spark SQL functionality can run your SQL query.
  - Spark SQL supports the HiveQL syntax as well as Hive SerDes and UDFs, allowing you to access existing Hive warehouses.

- Spark Streaming:
  - Spark Streaming involves processing live data streams in Spark. **This feature of Spark is another advantage over MapReduce.**
  - Spark Streaming provides APIs that resemble the RDD API used in Spark Core. Due to this, you can easily manipulate data that is either stored on disk or is coming from live data streams.
  - It provides the same level of fault tolerance as provided by Spark Core.
  - Spark streaming is used extensively in the industry. Companies like Netflix use Kafka and Spark Streaming to run real-time engines that provide movie recommendations to their users. Banks can build real-time fraud detection systems to minimize fraud
- **MLlib:**
  - MLlib involves applying machine learning algorithms. To analyze and build models over the distributed data sets, MLlib offers a wide range of features.
  - It supports functionalities that include model evaluation and data import.
- **GraphX:**
  - GraphX is used to process large volumes of data in Spark in the form of graphs.
  - It is Spark’s tool for ETL (Extract, Transform & Load) process, exploratory analysis, and iterative graph computation.
  - This package also provides different operators for working with graphs and manipulating them.
  - The usage of graphs can be seen in LinkedIn’s connections (networks), Facebook friends, and Google maps. These are all different kinds of graphs. For example, if you want to find out which family member has the most number of connections on Facebook or which two cities in the country are located farthest from each other, you can use GraphX to get the answers
- **Packages:**
  - Different packages like PySpark, SparkR, etc., help to run queries on big data through the libraries supported by languages like Python, R, etc. This flexibility of using different languages to write code adds to Spark's ease of use.
- **Pro Tip: Spark's ecosystem adds to Spark's Run Everywhere feature by supporting so many libraries and functionalities. All these make Spark highly powerful and easy to use.**

▼ Spark Architecture

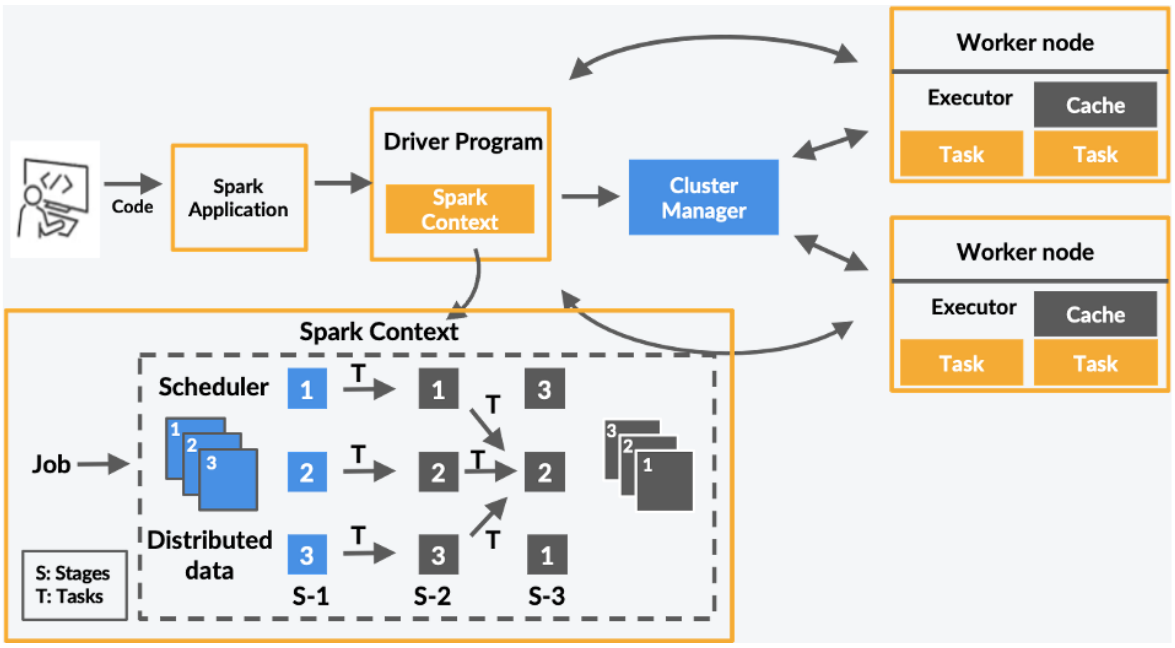
- Comparison



  - Both Hadoop and Spark follow similar distributed approach
  - Components have a similar framework but fulfill different functionalities
  - Like Hadoop MapReduce, Spark also distributes data across a cluster and processes it in multiple nodes parallelly. It also uses master-slave architecture. This architecture in Spark consists of:
    - **Driver node** (master): Runs the driver program, and

    - **Worker nodes** (slave): Runs the executor.

- The driver program and the executor program are managed by the cluster manager. [A cluster manager is a pluggable component in Spark](). Because of this, Spark can run on various cluster manager modes, which include the following:
    - In the **Standalone** mode, Spark uses its own cluster manager and does not require any external infrastructure.
    - However, at an enterprise level, for running large Spark jobs, Spark can be integrated with external cluster managers like **Apache YARN** or **Apache Mesos**. This facility allows it to be deployed on the same infrastructure as Hadoop and acts as an advantage for companies looking to use Spark as an analytics platform.
    - Apache Mesos is deprecated in the latest version of Spark, but it's still a widely used cluster in the industry.
    - Spark has also added **Kubernetes** as a cluster manager in its latest version.
- Spark job get executed in Spark



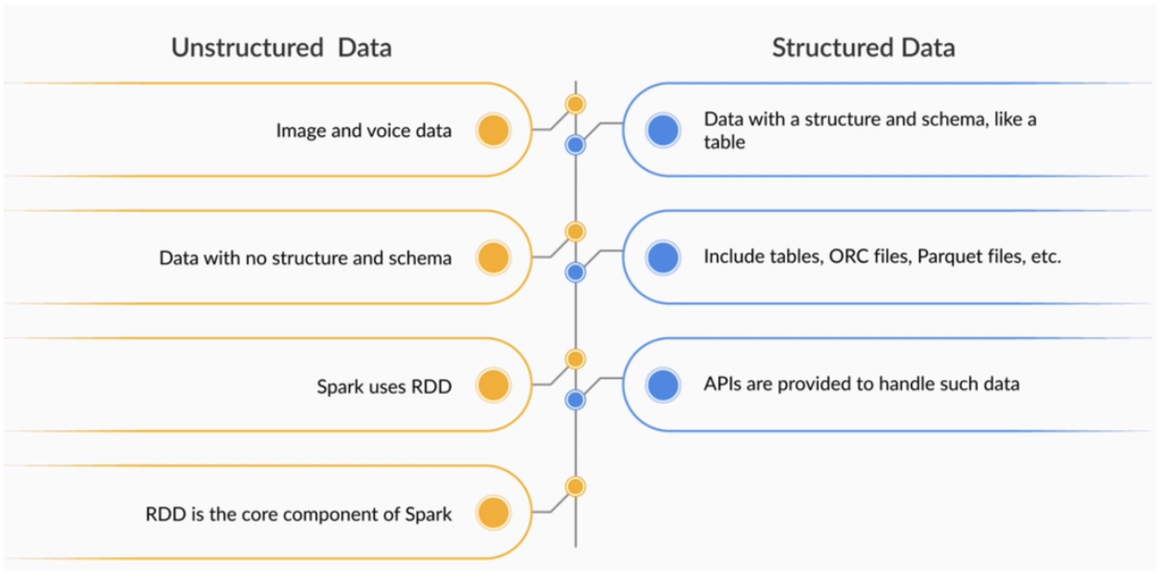
    - **Spark Context: a door allows all thing inside Spark**
    - The role of each Spark
        - **The role of Spark Context:**
            - Spark Context **does not execute** the code but **creates an optimized physical plan of the execution** within the Spark architecture. It is the initial entry point of Spark to the distributed environment.
        - **The role of the driver program:**
            - **The driver program is like the main() method** that contains the instructions and action steps to be taken on the data present in each worker node.
            - A driver program creates a general logical graph of operations. These operations mostly involve the creation of RDD from some source data, transformation functions to manipulate and filter data, and finally, some action to save the data or print it.
            - When the driver program runs, the logical graph is turned into an execution plan.
            - In Spark terminology, a process or action on data is called a **Spark job**. A job is further broken down into different **stages**. These stages help make the Spark environment reliable and fault-tolerant, which we will look at in the later sections. Finally, each stage comprises **tasks** the executors implement. A task is the most basic unit of work that each executor performs parallelly on the respective partition of data.

- Once the entire execution plan is ready, the Spark driver coordinates with executors to run various tasks.

- **The role of the executors:**
    - Executors are processes that are launched for a Spark application on worker nodes.
    - Each worker node consists of one or more executor(s) and is responsible for running the task.
    - The main task of an executor is to run the tasks and send results to the driver program.
    - It also stores the cached data that is created while running a user program.
    - One executor can consume one or more cores on a single worker node. Assume the following:
        - If one executor runs on one core, and one worker node contains eight cores, there will be eight executors in a single worker node. Since every executor has only one core to run a process or task, the operations in the program cannot be parallelized.
        - If one executor runs on two cores, and one worker node contains eight cores, there will be four executors in a single worker node. Since every executor has two cores to run a process or task, the operations in the program can be parallelized.
- **The role of cluster manager:**
    - It launches the executor programs and allocates and manages the resources allocated to each component.

▼ Spark APIs

- Spark can load data from various sources, which can be classified into the following two categories:


Unstructured Data
Structured Data
Image and voice data
Data with a structure and schema, like a table
Data with no structure and schema
Include tables, ORC files, Parquet files, etc.
Spark uses RDD
APIs are provided to handle such data
RDD is the core component of Spark

- Unstructured data
    - Unstructured data is generally free-form text that lacks a schema (which defines the organization of the data).
    - Examples of such data include text files, log files, images, videos, etc.
    - To deal with unstructured data, Spark uses an unstructured API in the form of a resilient distributed data set (RDD).
    - RDD is the core component of Spark, and it helps in working with unstructured data.
- Structured data
    - Structured data includes a schema.
    - The data could be structured in a columnar or a row format.
    - Structured data formats include ORC files, Parquet files, tables, or dataframes in SQL, Python, etc.

- To deal with this type of data, Spark provides multiple APIs, including SparkSQL, DataFrame, and data set. We will be learning about them in the next module.

II. Programming with Spark RDD

▼ Spark Installation

Major pros of Spark 3

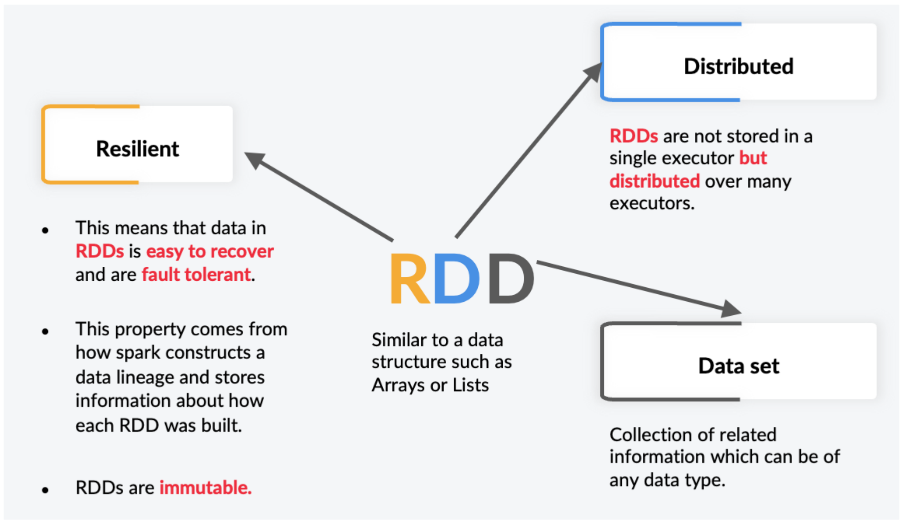
- Spark 3 is much **faster** than Spark 2 because of optimizations like adaptive query execution and dynamic partition pruning.
- Spark 3 offers additional **GPU instance support.**
- Spark 3 provides deeper **Kubernetes support.**
- **Spark 3 includes binary file support.**
- Spark 3 has new UI support for structured streaming.

Major deprecations in Spark 3

- Support for RDD-Interfaced Machine Learning (ML) Libraries
- Python 2
- Mesos cluster manager

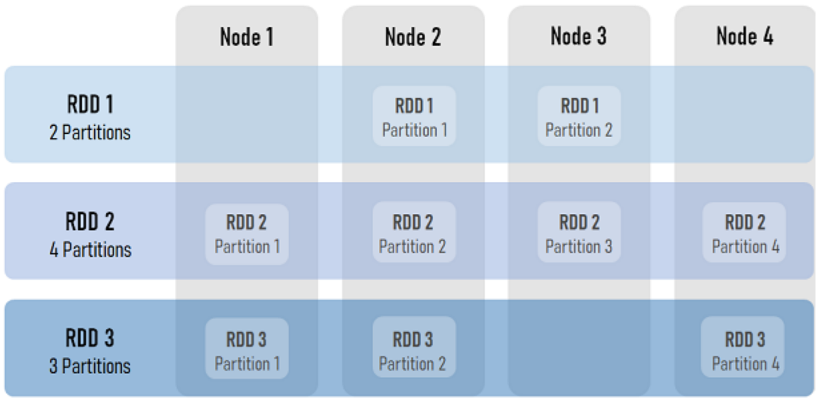
▼ Introduction to Spark RDDs

- **Resilient Distributed Data sets (RDDs)** form the core abstraction of Spark. RDDs are special data types that are tailor-made for Apache Spark. The first boost to the performance of Apache Spark came from the innovative nature of the structure of an RDD. An RDD can be considered a **distributed set of elements**


Resilient
This means that data in RDDs is easy to recover and are fault tolerant.
This property comes from how spark constructs a data lineage and stores information about how each RDD was built.
RDDs are immutable.
RDD
Similar to a data structure such as Arrays or Lists
Distributed
RDDs are not stored in a single executor but distributed over many executors.
Data set
Collection of related information which can be of any data type.

  - **Distributed collection of data**: RDDs exist in a distributed form over different worker nodes. This property helps RDDs store large data sets. The driver node is responsible for creating and tracking this distribution.
  - **Fault tolerance**: This refers to the ability to generate RDDs if they are lost during computation. Intuitively, fault tolerance implies that if somehow an RDD gets corrupted (lost due to the volatility of memory), then you can recover the uncorrupted RDD (resilient).
  - **Immutable**: RDDs are immutable, which means that once an RDD is transformed, a new RDD is formed, and you cannot edit the older RDDs. This property of RDDs helps maintain the data lineage, which you will be studying later in this session.

    - **Parallel operations**: Although RDDs exist as distributed files across worker nodes, their processing takes place in parallel. Multiple worker nodes work simultaneously to execute the complete job.
    - **Ability to use varied data sources**: RDDs are not dependent on any specific structure of an input data source. They are adaptive and can be built from different sources.
- How RDD storage



▼ Creating RDDs

- Starting a Spark application command: sc
- There are 3 ways to create RDD in Spark:
    - Parallelization
        - `rdd1 = sc.parallelize(["Bob", "Sam", "Ben", "John", "Ian", "Kate"])`
        - `rdd1.getNumPartitions()`: return a number, which indicates the default partitioning done by Spark core according to the size of rdd1.
        - `rdd1 = sc.parallelize(["Bob", "Sam", "Ben", "John", "Ian", "Kate"], 3)`: set the number of partitions while creating a RDD
        - `rdd1.collect()`: display the elements of rdd1

          ⇒ The collect() function sends all the values of an RDD back to the driver node. If the RDD is vast, then this function can result in an Out of Memory error.
        - `rdd1.count()`: count how many elements in RDD
        - `rdd1.glom().collect()`: glom() try go to ech partition and create a list of elements in each partition
    - External files
        - rddnew = sc.textFile('file_path.txt')

      ⇒ Use Livy as service to connect Jupyter notebook ⇒ file under `/user/livy` as default

        ⇒ `hadoop fs -ls /user/livy`

        ⇒ `saveAsTextFile('/user/livy/output01')`
    - Existing RDD
- From S3 file to "livy"
    - `hadoop distcp <S3 file location URI> .` : copy file from S3 to Hadoop root folder
    - `hadoop fs -put "filename" /user/livy/<folder name if any>` : from Hadoop root folder to "livy"
- Get type of RDD:
    - `type(rdd1)`

RDD Programming Guide - Spark 3.4.1 Documentation
Spark 3.4.1 programming guide in Java, Scala and Python
https://spark.apache.org/docs/latest/rdd-programming-guide.html

▼ Operations on RDDs

- 2 types of operations:
  - Transformation: operations on an RDD ⇒ result a new RDD
  - Action: operation result in an output **not stored as RDD**

▼ Transformation Operations

- **filter():** This operation is useful to filter out the contents of an RDD based on a condition.
- **map()**: It runs a function over each element of an RDD.
- **flatMap()**: It runs a function where the output of each element may not be a single element.
- **distinct():** This function will identify the unique elements in an RDD and put them in a new RDD.
- **union()**: This operation will work on two RDDs and will result in an output that contains all the elements present in both the RDDs.
- **intersection()**: This operation will work on two RDDs and will result in an output that contains only those elements that are present in both the RDDs.
- **subtract()**: This operation will work on two RDDs and will result in an output that contains all the elements present in rdd1 but not those present in rdd2.
- **cartesian()**: This operation will work on two RDDs and will result in an output that contains pairs of each element of rdd1 with each element of rdd2.
- **sorted()**: Sort all elements in a RDD

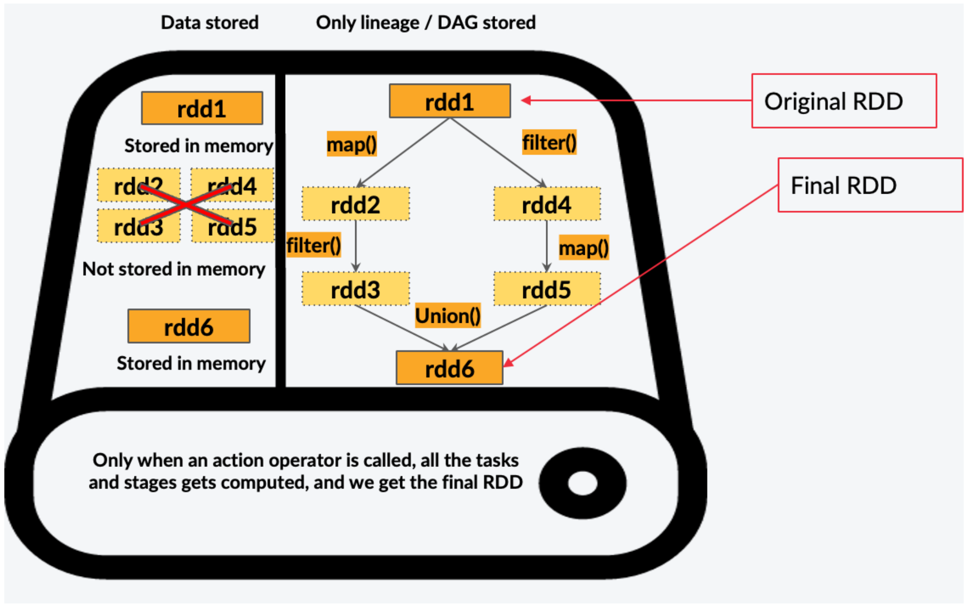
▼ Action Operations

- **collect()**: This operation collects all the elements of an RDD from every partition and returns the result as an array to the driver node.
- **count()**: This action returns the total number of elements of an RDD.
- **countByValue():** return a dictionary of unique value
- **items():** return items in dictionary output
- **take(num)**: This operation takes the first num (number) of elements of an RDD. It first scans one partition and then uses the results from that partition to estimate the number of additional partitions needed to satisfy the limit.
- **top(num):** return top #num in RDD i.e. [2,3,4,5,6] → top 2 : [6, 5]
- **first()**: It returns the first element of an RDD. There are two questions that can be asked regarding this action.
- **reduce()**: This function is useful to perform various operations on RDDs, where both the values on which the operation is performed are within the same RDD. sc.parallelize([1,2,3,4,5]).reduce(lambda x,y: x*y) → 120
- **fold()**: Aggregate the elements of each partition, and then the results for all the partitions, using a given associative function and a neutral zero value. The zero value is used as the initial value for each partition in folding. It is used to initialize the accumulator for each partition. It acts as an initial call to each partition. i.e. sc.parallelize([1,2,3,4,5]).fold(0, add) → 15; sc.parallelize([1,2,3,4,5]).fold(1, lambda x,y: x*y) → 120.
- **Aggregate() function:** Aggregates the elements of each partition and then the results for all the partitions using a given combination of functions and a neutral zero value. **Since RDDs are partitioned, the aggregate takes full advantage of it by first aggregating elements in each partition and then aggregating results of all partitions to get the final result.** Aggregate() lets you take an RDD and generate a single value that is of a different type than what was stored in the original RDD.

- **foreach()** function: Applies a function to all elements of this RDD. **It does not return any value, and it executes the input function on each element of an RDD**. **It does not return any value to the driver node, instead, the code is executed and stored in the worker node itself.**

▼ Lazy Evaluation in Spark

- Lazy Evaluation in Spark
  - Spark **dose not perform any transformation** utils an action is called on an RDD.
  - Rather Spark created a lineage that stores how each RDD can be derived from transformations.
  - Since Spark knows how to derive each RDD, in case of system failure, RDD can be recreated.
  - Lazy evaluation in Spark amkes RDD resilient and fault tolerant.
- Directed Acyclic Graph (DAG):

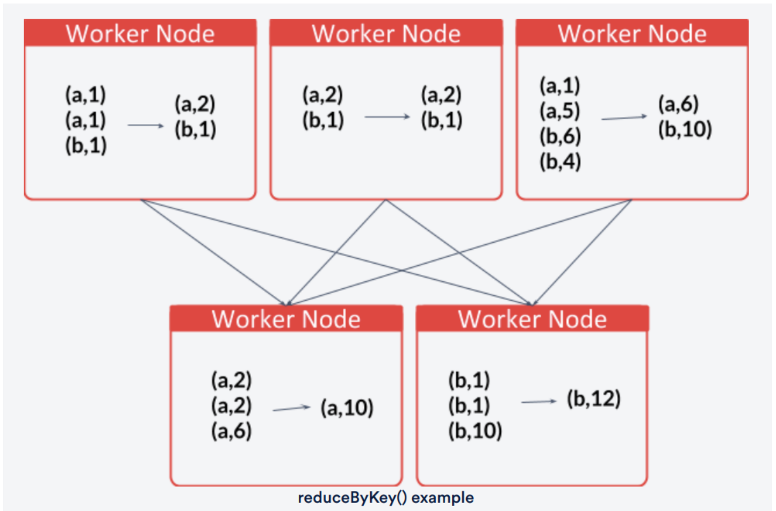


  - When a user submits a code file, SparkContext creates an optimal plan in the background. This plan (the lineage of RDD) is stored in the **Directed Acyclic Graph (DAG) Scheduler** within SparkContext.
    - **Directed**: The arrows link one point to another point in a single direction.
    - **Acyclic**: All nodes have a property that states that if you start from any node in the graph and follow the arrows, then you cannot come back to the same node. That is, there is no cyclic loop in the graph.
  - The DAG Scheduler prepares the flow of how an RDD is derived from the parent RDD. This process helps to make the RDDs fault-tolerant.
    - At any instance, if a Spark job fails, then you can use the stage flow mentioned in the graph to return to the same state.
    - This way, your progress is not lost; hence, the term resilient is associated with RDDs. Note that the **lineage stores the details of the transformations, and not the data itself.**
    - Once the transformations leading up to an RDD are known, any stage (representing a subset of the original data) can be recreate
- The advantages of lazy evaluation are as follows:
  - Due to lazy evaluation, Spark does not perform unnecessary computation.

- More specifically, it lets Spark **perform computation only when a final result is required**. It eliminates any redundant steps or merges logically independent tasks into a single parallel step.
- It also helps Spark to use the executor memory efficiently, as it will not create and store multiple RDDs that occupy the limited memory space until an RDD is required.

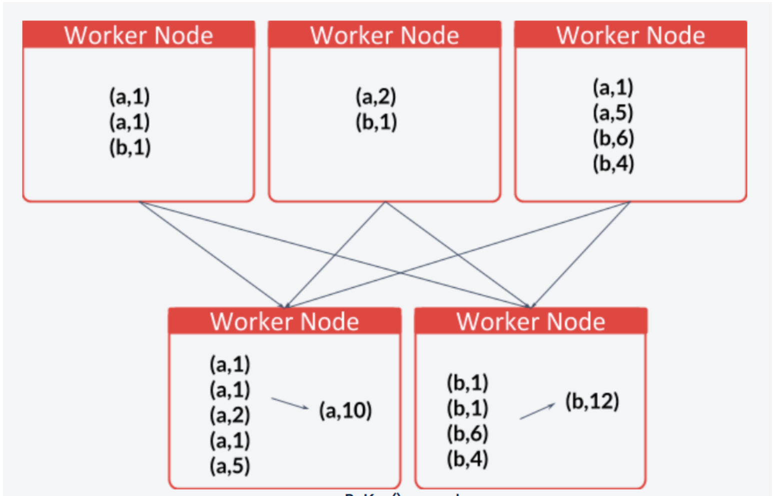
III. Paired RDDs

- **Paired RDD** is a special RDD class that holds the data as (key, value) pair. Due to the difference in structure, there are various operations associated with paired RDDs.
- Transformation functions on a paired RDD
  - **reduceByKey():** The reduceByKey() method is used to perform a particular operation on the elements of RDDs.


Worker Node
(a,1) (a,1) (b,1) → (a,2) (b,1)
Worker Node
(a,2) (b,1) → (a,2) (b,1)
Worker Node
(a,1) (a,5) (b,6) (b,4) → (a,6) (b,10)
Worker Node
(a,2) (a,2) (a,6) → (a,10)
Worker Node
(b,1) (b,1) (b,10) → (b,12)
reduceByKey() example

⇒ The reduce() method in basic RDDs is an action operation that returns a result to the driver program. However, reduceByKey() is a transformation operation that results in a new paired RDD with a key-value pair where the value is now an aggregated result for that particular key.

  - **groupByKey():** The groupByKey() method performs the same operation as reduceByKey(), but it creates an **iterable for the values for a particular key**. You get back an object which allows you to iterate over the results.



groupByKey() example

- **mapValues():** This function is used for operating on the value part of the key-value pair.

```
sales = sc.parallelize([("cosmetics", ["shampoo", "soap", "conditioner", "brush"]), ("food", ["bread", "meat"]), ("clothes", ["tshi
def f(sales): return len(sales)
sales.mapValues(f).collect()
# output: [('cosmetics', 4), ('food', 2), ('clothes', 2)]
```

- **flatMapValues():** FlatMap "breaks down" collections into the elements of the collection

```
sales = sc.parallelize([("cosmetics", ["shampoo", "soap", "conditioner", "brush"]), ("food", ["bread", "meat"]), ("clothes", ["tshi
def f(sales): return sales
sales.flatMapValues(f).collect()
# output: [('cosmetics', 'shampoo'), ('cosmetics', 'soap'), ('cosmetics', 'conditioner'), ('cosmetics', 'brush'), ('food', 'bread'
```

- **keys():** The keys() function creates a new RDD that contains only the keys from the paired RDD.

```
sales_1 = sc.parallelize([("shampoo", 1), ("soap", 1), ("conditioner", 3), ("brush", 2), ("soap", 2), ("shampoo", 2), ("bread", 2),
k = sales_1.keys();
k.collect()
# output: ['shampoo', 'soap', 'conditioner', 'brush', 'soap', 'shampoo', 'bread', 'meat', 'tshirt', 'jeans']
```

- **values():** The values() function creates a new RDD that contains only the values from the paired RDD.
- **sortByKey():** This operator is used to sort the elements of a paired RDD. The sorting is done based on the key of the paired RDD.
- **subtractByKey()**: Return each (key, value) pair in self that has no pair with matching key in other.
- **join():** Return an RDD containing all pairs of elements with matching keys in self and other. Whenever you do an inner join, the key must be present in both the paired RDDs; however, for an outer join, the key may or may not be present in both the paired RDDs
- **rightOuterJoin():** The rightOuterJoin() has the option to skip the key that is present on the left side of the operator; however, all keys that are present on the right side of the operator must be present.
- **leftOuterJoin():** The leftOuterJoin() has the option to skip the key that is present on the right side of the operator; however,all keys that are present on the left side of the operator must be present.
- **cogroup():** In the case of cogroup(), if the key is present in any one of the RDDs, it will be present in the output. For each key k in self or other, return a resulting RDD that contains a tuple with the list of values for that key in self as well as other.
- **countByKey():** Counts the number of elements for each key.
- **lookup(key):** Finds all the values associated with the key provided.
- **rightOuterJoin():** The rightOuterJoin() has the option to skip the key that is present on the left side of the operator; however, all keys that are present on the right side of the operator must be present

```
rdd1 = sc.parallelize([('Apple', 50), ('Banana', 100), ('Mango', 150), ('Carrot', 120)])
rdd2 = sc.parallelize([('Apple', 100), ('Banana', 120), ('Mango',150)])
sorted(rdd2.rightOuterJoin(rdd1).collect())
# output: [('Apple', (100, 50)), ('Banana', (120, 100)), ('Carrot', (None, 120)), ('Mango', (150, 150))]
```

- **leftOuterJoin():** The leftOuterJoin() has the option to skip the key that is present on the right side of the operator; however,all keys that are present on the left side of the operator must be present.

```
rdd1 = sc.parallelize([('Apple', 50), ('Banana', 100), ('Mango', 150), ('Carrot', 120)])
rdd2 = sc.parallelize([('Apple', 100), ('Banana', 120), ('Mango',150)])
sorted(rdd2.leftOuterJoin(rdd1).collect())
# output: [('Apple', (100, 50)), ('Banana', (120, 100)), ('Mango', (150, 150))]
```

- **cogroup():** In the case of cogroup(), if the key is present in any one of the RDDs, it will be present in the output. For each key k in self or other, return a resulting RDD that contains a tuple with the list of values for that key in self as well as other.