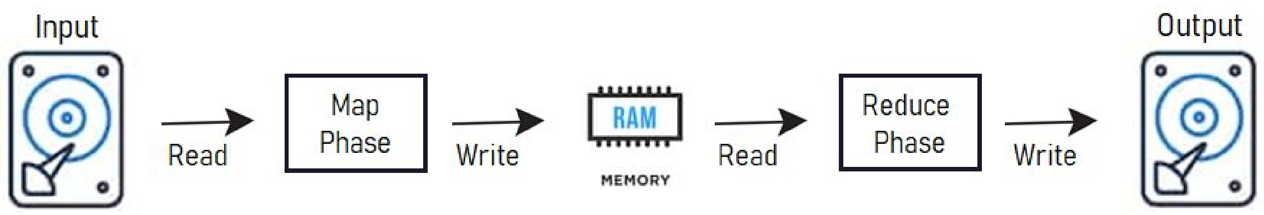
SUMMARY

Introduction to Spark

In this session, you were introduced to the concepts of Apache Spark. You began the session by establishing differences between Hadoop MapReduce and Spark. Then you learnt about the fundamental data structure of Spark i.e., RDDs. You also learnt about various operations on RDDs and about Spark architecture.

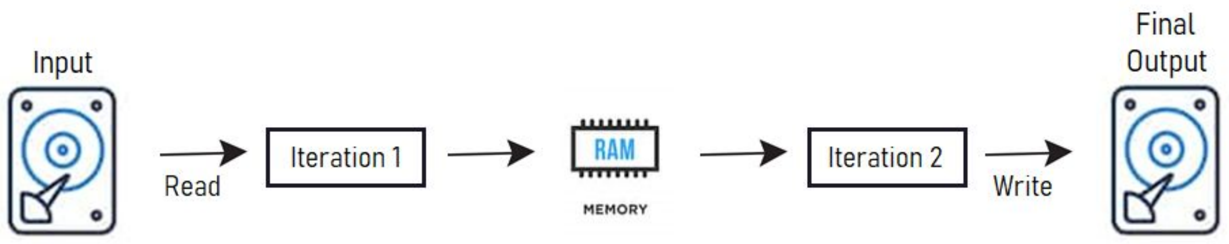
Spark vs MapReduce

HDFS (Hadoop Distributed File System) is the starting point of the MapReduce framework. The data is stored as blocks, which are distributed among different nodes in the cluster. The mapper first reads the data from the HDFS blocks and produces an output, which is then written in **memory or disk,** from where it is picked up by the reducer, and the final output is written onto an HDFS file as depicted in the diagram below:



However, in the case of Big Data, the number of map and reduce steps will be very large due to the large size of the data. Time complexity is a major factor that comes into account in the case of Big Data analytics. Therefore, there is a requirement for something faster than Hadoop MapReduce.

Apache Spark comes with the capability of **in-memory processing**. It can store data on memory for processing instead of writing the data on disk. Hence, the iterative process does not involve writing the output to disk, which could be a time-consuming step. This feature of Spark makes it a faster processing framework than the MapReduce framework. Take a look at the image below for better understanding:



Spark: Introduction

"Apache Spark™ is a unified analytics engine for large-scale data processing."

Apache Spark was started at the AMPLab at UC Berkeley by Matei Zaharia as an alternative to the MapReduce paradigm. It was later donated to the Apache Software Foundation, which made it an open source project.

Let's revise some of the features of Spark:

- **High speed**: Spark processes data at a very high speed, as it allows in-memory computation. It can work up to 10x-100x faster than MapReduce, depending on the nature of the work.
- **Support for multiple data sources**: Spark can process data in different file formats such as CSV and JSON, and it can also run over different platforms such as HSFD, S3, Cassandra etc.
- **Rich support of language APIs**: Spark enables you to run your functions in Python, R, Java, Scala or even SQL.
- **Advanced analytics**: Due to in-memory computation, you can easily perform exploratory data analysis and build machine learning models on GBs/TBs of data.

As you can see, Spark is extensively used by multiple industry domains for day-to-day activities.

1. Spark can process a large pool of data and provide the results quickly. Therefore, it aids many **business decisions** and optimises the workflow for companies.

2. The abilities of Spark also benefit the **banking and finance industry**. Banks process billions of transactions on a daily basis, as their network spreads across the globe. They have databases that collect real-time data, which is then analysed using frameworks such as Spark to detect whether any fraud is being committed in the background.

3. Several industries also rely on Spark to maximise their revenue. **Ride-sharing applications such as Ola and Uber** calculate the surge prices based on the booking and availability of drivers in that particular area. They run complex ML algorithms in real time, and the results reflected directly on their mobile application. This is possible because of the rich support of libraries offered by Spark.

4. Companies track the **browser history** of the user and provide a personalised ad on various websites or applications accessed by the user. This is possible because of the rich library support and quick analysis features of Spark.

Spark: In-Memory Storage

Unlike MapReduce, Spark enables the data to be processed in memory. Let's quickly recall how this is done:

- The map and reduce phases are merged into a unified layer, that is, **Executors**. An executor is a JVM (Java Virtual Machine) process that is responsible for executing the operations mentioned in the Spark program.
- **JVM** stands for Java Virtual Machine, and it helps to execute Java programs as well as programs that are written in other languages but are compiled to Java bytecode. It enables speedy execution.

The executors hold the data in a distributed manner in memory. When new data is defined, it gets split into smaller parts and distributed to one or more than one executors. These small chunks of data are collectively called RDDs. RDDs enable in-memory computations for a large size of data.

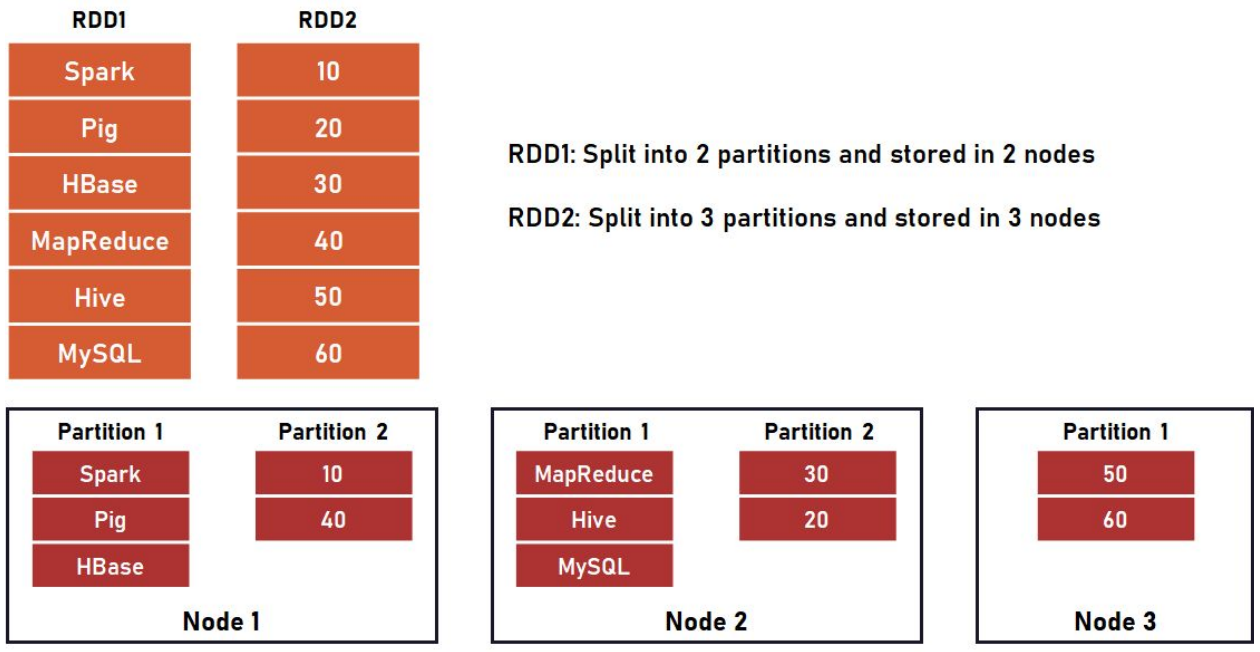
Let’s understand what the acronym RDD stands for:

- **Resilient:** This indicates that RDDs are **fault-tolerant,** which means that RDDs have the ability to recover themselves in case any data is lost during computation.
- **Distributed:** The data inside RDDs resides across multiple interconnected nodes.
- **Dataset:** It is a collection of partitioned data.

Now, let’s quickly recap the features of RDDs:

- **Immutability**: The data stored in RDDs is read-only and cannot be altered.
- **In-memory storage**: RDDs can be stored in memory, and this results in the high processing speed of Spark.
- **Parallel operations**: Each data set is divided into logical partitions that may be computed on different nodes of a cluster. Then, under the hood, Spark distributes the data contained in the RDD internally among the nodes and parallelises the operation that you perform on it.
- **Ability to use varied data sources**: RDDs can include any Python, Java or Scala object, or even user-defined classes.

This following diagram summarizes the workings of RDDs, i.e., how data is distributed over nodes and different partitions:



You can use Spark with any of the following:

- Spark Standalone - Spark’s own cluster manager
- Apache YARN - Hadoop’s resource manager
- Apache Mesos
- Apache Kubernetes

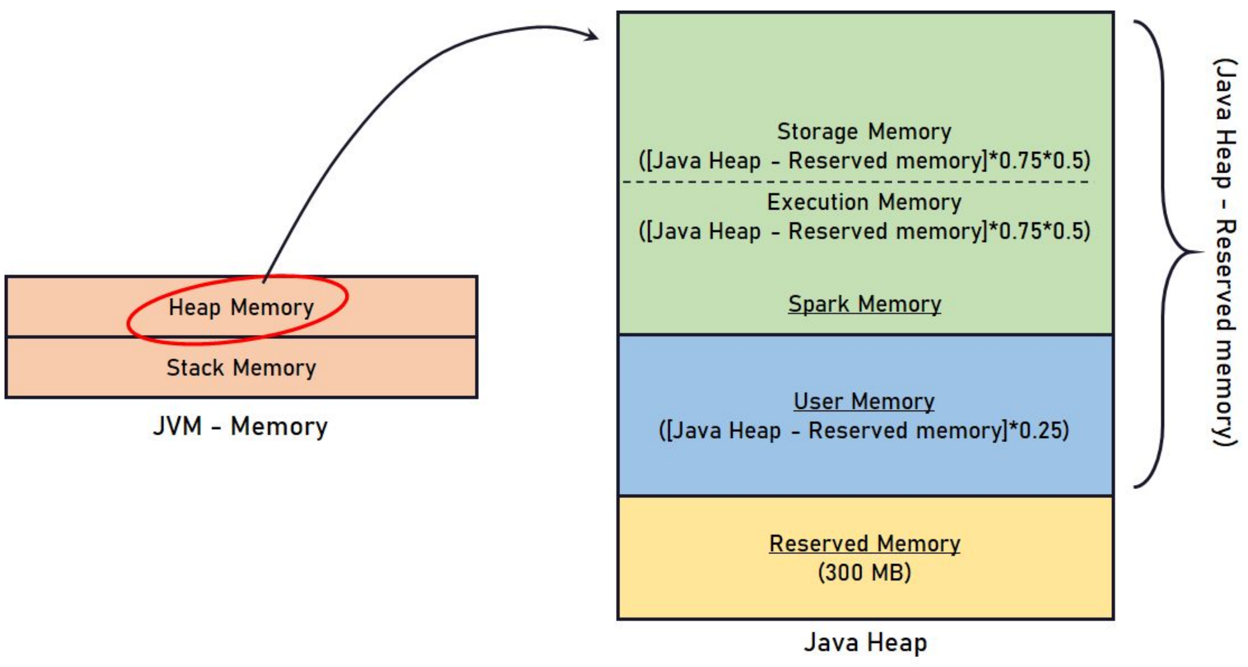
Spark: RDD Storage

For now, you need to understand that the worker nodes are the slave nodes that run the executor program. The memory inside the JVM is distributed into the following two parts:

- **Java Heap**
- **Java Stack**

Let's recall the different types of memories available in Spark:

- **Reserved memory**: By default, the size of this memory pool is around 300 MB. It is used for storing the Spark internal objects inside the JVM, which, in turn, help in the proper functioning of the executor.
- **User memory** ([Java Heap - Reserved memory]*0.25): Spark provides a fraction of memory that stores the user data structures and internal metadata in Spark. This space is reserved for the user and is not included by Spark during the storage or computation stage inside the JVM. It can also be configured to safeguard against OOM (Out of memory) errors in the case of large records. OOM errors may arise when the data is too big for the memory to hold inside Spark memory. 25% of the remaining memory is kept as user memory by default.
- **Spark Memory** ([Java Heap - Reserved memory]*0.75): The remaining memory constitutes the memory pool that is used by Spark to run the provided tasks. It is split into two parts:
  - **Spark storage memory**: This part of the memory is used for storing the cached data and propagating internal data across nodes in Spark. You will be able to understand more about it when we talk about broadcast variables in Spark.
  - **Spark execution memory**: As the name suggests, execution memory is used for computation during operations such as shuffles, joins and aggregations.


Heap Memory
Stack Memory
JVM - Memory
Storage Memory
([Java Heap - Reserved memory]*0.75*0.5)
Execution Memory
([Java Heap - Reserved memory]*0.75*0.5)
Spark Memory
User Memory
([Java Heap - Reserved memory]*0.25)
Reserved Memory
(300 MB)
Java Heap
(Java Heap - Reserved memory)

Now, let's summarise a few points about the storage of RDDs:

1. When we store RDDs in memory, they are stored inside the storage memory of the executors.
2. RDDs are stored in the form of partitions, which may be spread over single or multiple executors.
3. The number of executors can be configured based on the level of partitioning required.
4. The maximum number of executors that a worker node can host is equal to the number of processing cores in the node. The core signifies the number of parallel operations that a processor can perform.
5. One partition of an RDD can be mapped onto a single executor only. However, you can have multiple partitions in a single executor.
6. If there are multiple cores per executor, RDDs will be able to run parallel processes on each partition.
7. The ability to process data in memory makes Spark independent, and hence, you may choose to work with or without the HDFS.
8. But generally, Spark is used along with HDFS, as it integrates faster processing with reliable storage space for Big Data.

Spark: Architecture

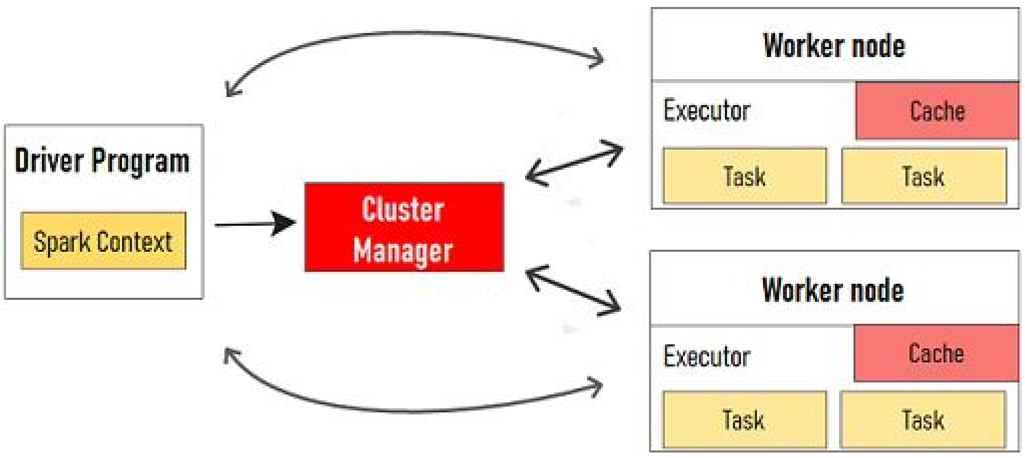
Like MapReduce, Spark also uses a master-slave architecture, except that the terms used are different.

- **Driver node**: The master node in Spark is known as the driver node, and it hosts the **driver program**.
- **Worker node**: These are slave nodes, like data nodes, where Spark hosts the **executor** program.

1. **Driver program**
    - Being the master node, the driver program is responsible for managing all the jobs that a user submits to the Spark environment.
    - The driver program breaks the code submitted by the user into smaller tasks and then distributes them among the executors. The driver program performs this task by launching **SparkContext**. SparkContext serves as the entry point for the Spark environment. It acts as the bridge between the user code and the Spark execution environment. The main function of SparkContext is to translate the code into an optimised execution plan for the executors. This is one of the key reasons why execution in Spark is faster than MapReduce.
    - The driver program is responsible for creating the RDDs and holding the metadata associated with its partitions.
    - Another key function of the driver program is to negotiate for the resources with the cluster manager. Based on the request and the available resources, the cluster manager then launches the executor program on the worker nodes.

2. **Executor program**
    - The executor program executes the task provided by the driver program. Multiple executors run completely different parts of the job received from the driver program in parallel. After execution, the executor program also sends the results to the driver program.
    - Another important role of an executor is that it provides in-memory storage for RDDs, which are used during the execution of tasks.

The image below shows the functioning of Spark architecture:



Spark has an edge over the MapReduce environment in the case of cluster managers too. Due to the lack of dependency on storage systems, Spark is able to run on multiple modes where it can accommodate different cluster managers. You can deploy Spark with the following:

- Local mode: Spark is hosted on a single machine. The machine serves the purpose of all the components in the Spark architecture. Here, the driver program plays the role of the resource manager too.
- Standalone mode: Spark uses an in-built cluster manager in this mode.
- Apache YARN: Spark can is used with Apache YARN as the cluster manager.
- Apache Mesos: Another cluster manager that can be used with Spark is Apache Mesos.

RDD Class

As you already know, RDDs form the core of Spark. RDDs are defined as a **class** in Spark because it follows object-oriented programming (OOP). In object-oriented programming, the user defines the class, which holds objects in it.

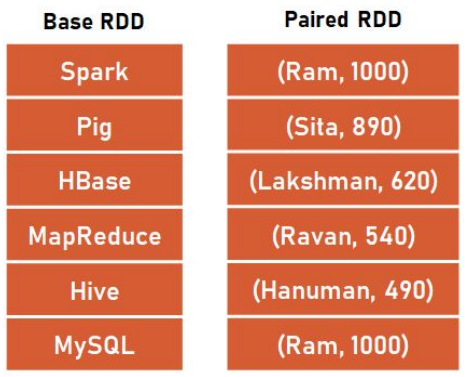
Some essential attributes of an RDD class are as follows:

- **Partitions:** RDDs are distributed in nature; this attribute denotes that data is stored as partitions in an RDD.
- **Iterator:** An iterator helps you iterate over the partitions of an RDD.
- **Parent RDD:** Since RDDs are immutable, any change in an RDD results in a new RDD. The RDD class must hold the parent RDD from which it originated. This is covered in detail in the upcoming sessions.
- **Methods:** This attribute specifies all the operations or methods that you can implement over the RDD class.

So, there are two types of RDDs, which are as follows:

1. **Basic RDDs**
2. **Paired RDDs:** Data items in a paired RDD are in the form of key-value pairs.

The image below shows the two types of RDDs:



Base RDD	Paired RDD
Spark	(Ram, 1000)
Pig	(Sita, 890)
HBase	(Lakshman, 620)
MapReduce	(Ravan, 540)
Hive	(Hanuman, 490)
MySQL	(Ram, 1000)

Spark: Ecosystem

Let’s now recall your learnings about the Spark ecosystem:

1. **Spark Core (or Apache Spark)**
    - Spark Core is the central processing engine of the Spark application and provides an execution platform for all Spark applications.
    - All the components covered (driver node, worker node, programs, RDDs, etc.) form the key elements of the Spark engine. Everything in Spark is built on top of the Spark Core.
    - Spark Core stores the data in the form of RDDs, which is the key data abstraction in Spark.
    - Some of the other critical responsibilities of Spark Core are memory management, fault recovery, and the scheduling and distribution of jobs across worker nodes.

2. **SparkSQL**
    - SparkSQL allows users to run SQL queries on top of Spark.
    - This makes the processing of structured and semi-structured data quite convenient.

3. **Spark Streaming**
    - The Spark Streaming module processes streaming data in real time. Some of the popular streaming data sources are web server logs, IoT data, data from social networking websites, such as Twitter feed.

4. **Spark MLlib**
    - MLlib is a machine learning library that is used to run machine learning algorithms on Big Data sets that are distributed across a cluster of machines.
    - It provides APIs for common machine learning algorithms such as clustering, classification and generic gradient descent. You will learn more about them in future courses.
    - Due to in-memory processing in Spark, iterative algorithms such as clustering take very little time when compared with other machine learning libraries such as Mahout, which uses the Hadoop ecosystem.

5. **Spark GraphX**
    - The GraphX API allows a user to view data as a graph and combine graphs with RDDs.

Summary

RDD Programming - I

In this session, you learnt about programming using RDDs. You also understood various operations on both basic and paired RDDs.

Creating RDDs

There are two methods of creating RDDs:

1. Parallelising an existing collection of data in the driver program:
   ```
   rdd1 = sc.parallelize(range(6), 2)
   ```

2. Loading the data set stored in an external storage system, such as a shared file system, HDFS, HBase and S3:
   ```
   rdd = sc.textFile("file_path", number_of_partitions)
   ```

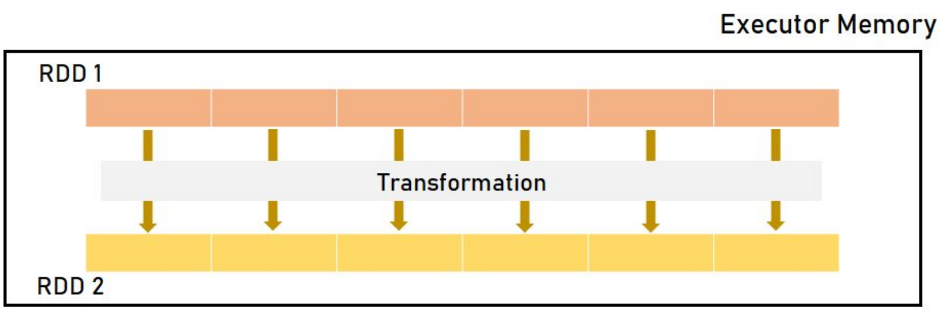
The parallelize() function works when you want to create an RDD from the data that already exists in the Spark environment. But most of the time, you will be working with the huge data stored in distributed file systems such as HDFS and S3. In such cases, you can use the sc.textFile() method.

In order to visualise the RDD and understand how data has been distributed over different partitions, use the following command:
```
rdd.saveAsTextFile('File_name')
```

Operations in RDDs

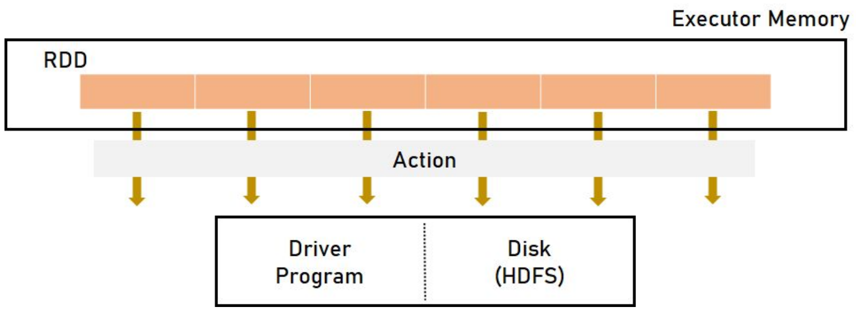
RDDs allow various methods or operations that can be performed on the data stored in them. These operations can be classified into two categories:

- **Transformations**: An operation on an RDD returns another RDD. As RDDs are immutable, every transformation results in a new RDD, as depicted in the diagram below:



- **Actions**: An operation on an RDD returns a non-RDD value. It usually involves operations such as printing, counting, adding and saving to disk and results in a non-RDD value.

The image below shows how actions are executed on a RDD:



Let's try to understand some of the important transformations and actions.

map()

- The map() transformation takes a function or a lambda expression as an argument, and this function is executed on each data item of the parent RDD.
- There is one-to-one mapping between the elements of the source RDD and those of the resultant RDD.
- The return value of the function, when applied to each data item of the parent RDD, forms the data items of the resultant RDD.

flatMap()

- The flatMap transformation maps an element of the input RDD onto one or more elements in the target RDD.
- Unlike in map(), for a single element in the input RDD, the lambda expression will return a collection of values.

filter()

- The filter() transformation takes a function or a lambda expression that returns a boolean value as the argument.
- This boolean function is executed on each element of the source RDD.
- Only those elements that returned 'true' when the lambda expression was executed on them find a place in the resultant RDD.

reduce()

- This action can be considered very similar to the reduce phase of the MapReduce framework.
- The reduce action takes a lambda expression as an input and aggregates the elements of the input RDD. You can also define a function and then use that to reduce the values stored in the RDD.
- The lambda expression passed in the reduce action denotes the operation that is used for aggregation.
- The operation denoted by the lambda expression should be commutative and associative.

glom()

- The transformation glom() helps to understand how data is split into partitions without saving the data as an external file as we do in the saveAsTextFile() method.

Some common actions that are applied to a basic RDD are as follows:

collect()

- This action collects all the data for a particular RDD distributed across partitions.
- The collected data is returned as an array to the driver program.

take()

- It takes an integer parameter "n" and returns a list of the first "n" elements of the RDD to the driver program.
- take() is more suitable than collect(), as it loads only a part of the data set in the driver program.

saveAsTextFile()

- It is useful for storing all the data items present in the RDD in persistent storage such as a local file system or HDFS.

count()

- It returns the total number of elements that exist in RDDs.

Operations on Paired RDDs

Paired RDDs are the RDDs that hold data as (Key, Value) pairs. You can easily create a paired RDD using the following methods:

- The parallelize() method
- The textFile() method
- Transforming base RDDs

Now, let’s quickly recall the important paired RDD transformations:

- The **reduceByKey()** transformation aggregates the values associated with a particular key based on the function provided.

- The **groupByKey()** transformation groups the values that have the same key.

Other important functions on paired RDDs are:

- aggregateByKey()
- countByKey()
- mapValues()

  etc.

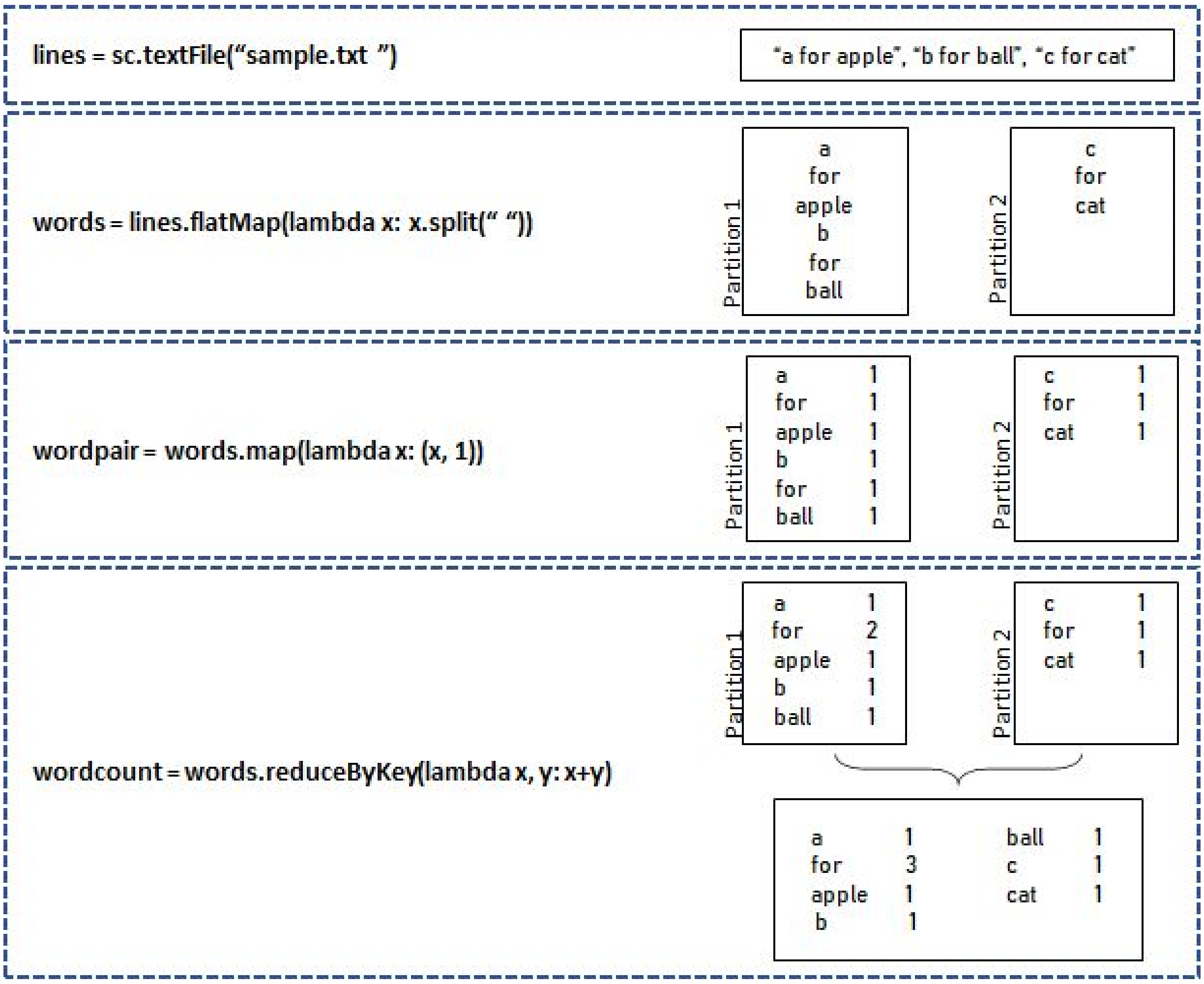
Summary

RDD Programming - II

In this session, you learnt about the Word Count problem, Lazy evaluation and Page Rank algorithm in Spark.

Word Count Problem

Let's now recall the word count problem that you have already learnt. The image below summarises the code and the entire flow of the word count problem:



lines = sc.textFile("sample.txt ")
"a for apple", "b for ball", "c for cat"
words = lines.flatMap(lambda x: x.split(" "))
Partition 1
a for apple b for ball
Partition 2
c for cat
wordpair = words.map(lambda x: (x, 1))
a 1 for 1 apple 1 b 1 for 1 ball 1
c 1 for 1 cat 1
wordcount = words.reduceByKey(lambda x, y: x+y)
a 1 for 2 apple 1 b 1 ball 1
c 1 for 1 cat 1
a 1 for 3 apple 1 b 1 ball 1 c 1 cat 1

You can clearly understand the advantages that Spark holds over the MapReduce framework through this simple example.

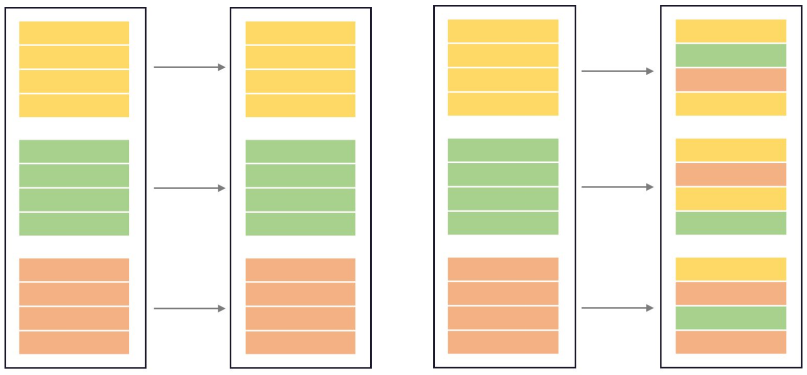
- The code written in Spark is very easy and self-interpretable. However, in MapReduce, you had to write the map and reduce jobs to execute the entire process. It involved writing longer and more complex codes in comparison with the Spark environment.
- The execution in Spark is much faster than in the MapReduce framework.

Transformations in RDD

As mentioned by professor in the video above, transformations are of two types:

- Narrow transformations: Only one partition of the child RDD will be generated from the elements of a single partition.
- Wide transformations: Multiple partitions of the child RDD will be generated from the elements of a single partition in the parent RDD.

The image below shows the two different categories of transformations:



The table below lists a few examples of both types of transformations:

Narrow Transformations	**Wide Transformations**
map()	groupByKey()
flatMap()	reduceByKey()
filter()	distinct()
union()	intersection()
sample()	

*join() can be a narrow or a wide transformation depending on the structure of the RDD over which the function is applied.

Lazy Evaluation in Spark

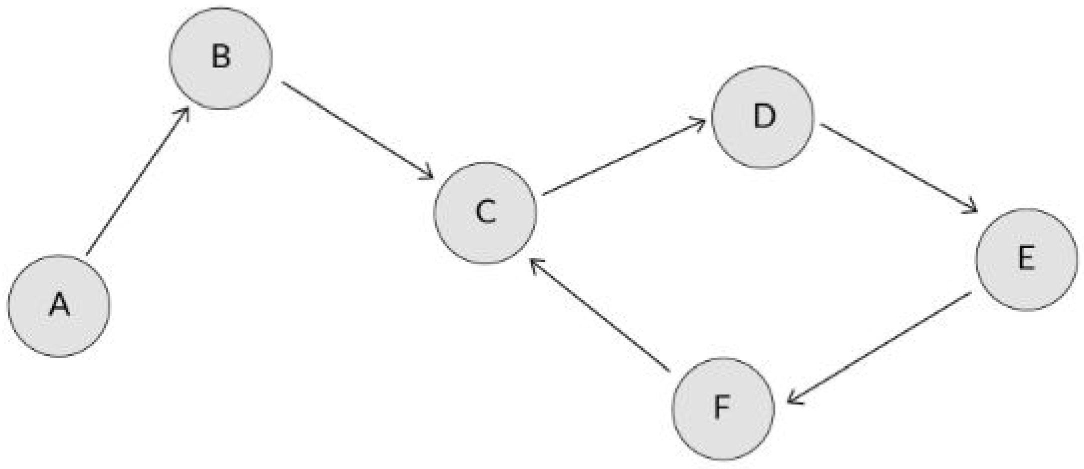
Transformations in Spark follow **lazy evaluation**, i.e., they are evaluated only when required. A transformation that you perform on an RDD is stored in the background as the metadata. Spark creates an object that stores the steps required to create a new RDD from the existing one. Spark then waits for an **action** to execute all the transformations that are stored in the metadata object.

Lazy evaluation allows Spark to analyse all the transformations specified by the user and optimise them. Here, optimisation refers to eliminating redundant steps or merging logically independent tasks into a single parallel step. Moreover, 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 that RDD is required.

When you perform a transformation, Spark creates a lineage in the background that keeps track of all the operations required to create an RDD from the parent RDD. In the previous session, it was mentioned that SparkContext first creates an optimised plan based on the code submitted by the user and then sends it for execution. It is able to do so because the RDD lineage is stored in the form of a **directed acyclic graph (DAG)**, which gets executed when SparkContext submits the optimised plan. The term 'directed acyclic graph' can be explained as follows:

- **Directed**: The arrows link one point to another in a single direction.
- **Acyclic**: All nodes have the property that if you start from any node in the graph and follow the arrows, you cannot come back to the same node. In other words, there is no cyclic loop in the graph.

The diagram below shows the basic structure of a DAG:



Partitioning

Spark data structures are expected to hold huge data sets; hence, they can be partitioned across multiple nodes in the cluster. These partitions form the basic unit of parallelism in Spark, i.e., a transformation is applied on multiple data partitions in parallel (provided that the application is being executed in multiple containers). Spark automatically distributes the data across different partitions when you create an RDD, and the user does not have any control on the partitioning scheme (i.e., which partition will hold what data).

The partitions of an RDD may lay across different executors of different nodes in the cluster. When you perform any operation, the data will be accessed from each partition and then operated based on the type of function that you have provided. So, when we create an RDD, elements are randomly distributed within the partitions. The two basic types of partitioning schemes in Spark are as follows:

1. **Hash partitioning**
   Hash partitioning calculates the hash of the key to decide the partition for a key-value pair. The concept of hash is that objects that are equal should have the same hash code. Therefore, elements with the same key are placed under the same partition. This is the default partitioning technique in Spark.

2. **Range partitioning**
   In range partitioning, tuples are first sorted and then partitioned based on the particular range of keys. Hence, tuples with keys within the same range will appear in the same partition. This is helpful when the keys of the RDD follow an order.

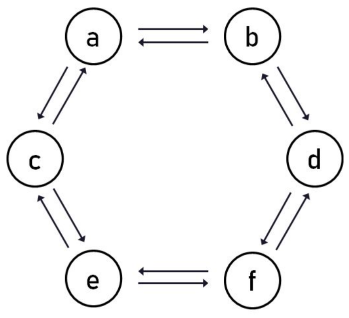
Take a look at the code below, to obtain the a better idea about different partitioning methods:

```
# Creating the RDD with 4 partitions
sample_rdd = sc.parallelize([(1,'Hadoop'), (7,'DynamoDB'), (2,'Spark'), (1,'Sqoop'), (8,'Zookeeper'), (5,'Storm'),
(7,'Flume'), (3,'Pig'), (2,'Kafka'), (4,'HBase'), (6,'MapReduce'), (4,'Hive'), (3,'Trident'), (5,'Hive'), (6,'Oozie'),
(8,'Zookeeper')], 4)

# Number of partitions
sample_rdd.getNumPartitions()
# Four partitions as defined in the parallelize command

# Check the distribution in partitions (use glom() transformation)
sample_rdd.glom().collect()

[[(1, 'Hadoop'), (7, 'DynamoDB'), (2, 'Spark'), (1, 'Sqoop')],
 [(8, 'Zookeeper'), (5, 'Storm'), (7, 'Flume'), (3, 'Pig')],
 [(2, 'Kafka'), (4, 'HBase'), (6, 'MapReduce'), (4, 'Hive')],
 [(3, 'Trident'), (5, 'Hive'), (6, 'Oozie'), (8, 'Zookeeper')]]

# By default, RDD has a random distribution of elements into the partitions.

#1. Hash Partitioning - Repartitioning into 3 partitions
sample_rdd_hash = sample_rdd.partitionBy(3)
sample_rdd_hash .glom().collect()

[[(3, 'Pig'), (6, 'MapReduce'), (3, 'Trident'), (6, 'Oozie')],
 [(1, 'Hadoop'),  (7, 'DynamoDB'),  (1, 'Sqoop'),  (7, 'Flume'),  (4, 'HBase'),  (4, 'Hive')],
 [(2, 'Spark'),  (8, 'Zookeeper'),  (5, 'Storm'),  (2, 'Kafka'),  (5, 'Hive'),  (8, 'Zookeeper')]]

# Elements with common key are now put into the same partition as the default partitioning scheme is Hash
partitioning


#2. Range Partitioning - Repartitioning into 3 partitions
sample_rdd_range = sample_rdd.sortByKey(numPartitions = 3)
sample_rdd_range.glom().collect()

[[(1, 'Hadoop'),  (1, 'Sqoop'),  (2, 'Spark'),  (2, 'Kafka'),  (3, 'Pig'),  (3, 'Trident')],
 [(4, 'HBase'),  (4, 'Hive'),  (5, 'Storm'),  (5, 'Hive'),  (6, 'MapReduce'),  (6, 'Oozie')],
 [(7, 'DynamoDB'), (7, 'Flume'), (8, 'Zookeeper'), (8, 'Zookeeper')]]

# Keys are sorted using the sortByKey() function and then repartitioned.


#3. Custom Partitioning - Repartitioning with odd and even keys in separate partitions
sample_rdd_custom = sample_rdd.partitionBy(2, lambda x: x%2)
sample_rdd_custom.glom().collect()

[[(2, 'Spark'), (8, 'Zookeeper'), (2, 'Kafka'), (4, 'HBase'), (6, 'MapReduce'), (4, 'Hive'), (6, 'Oozie'), (8,
'Zookeeper')],
 [(1, 'Hadoop'), (7, 'DynamoDB'), (1, 'Sqoop'), (5, 'Storm'), (7, 'Flume'), (3, 'Pig'), (3, 'Trident'), (5, 'Hive')]]
```

Apart from the aforementioned two techniques, Spark also allows you to customise the partitioning logic. This is termed as **custom partitioning**. Here, you can define the logic for partitioning.

PageRank Algorithm

Consider the entire internet to be a graph where every web page is a node and the links connecting the web pages are edges. If a hyperlink is present on page a, linking page b, it means that an edge from a to b exists. Web pages a and b are nodes here connected by hyperlinks that form the graph edges as shown in the diagram below:



Sample Graph

Hence, PageRank is a method for computing a ranking for every web page based on the graph of the web using the information provided by hyperlinks. The current graph of the web has billions of nodes (pages) and quadrillions of edges (links). Every page has some number of forward-links (out-edges) and back-links (in-edges).

The intuitive description of PageRank is that a page has a high rank if the sum of the ranks of its backlinks is high. This includes both these cases: when a page has many backlinks and when it has a few highly ranked backlinks. The algorithm assigns a Page Rank score to each of the web pages. The pages with a higher score are ranked higher. Google uses this algorithm in the background to display the search results on its web page.

PageRank is an iterative algorithm, and the scores keep updating after every iteration. We keep iterating over the entire graph until the difference in scores between two consecutive iterations is insignificant. Let's try to understand the mathematical side of the algorithm.

The PageRank score of a node can be calculated using the following formula:

$$V(i+1) = d*M*V(i) + [(1-d)/n]*I$$

where,
i represents the previous iteration for which PageRank scores are present,
d is the damping factor (generally taken as 0.85),
M is the transition matrix for the network of links,
V(i) represents the PageRank scores before the ith iteration,
V (i+1) represents the PageRank scores after the ith iteration,
I is the identity matrix, and
n is the total number of nodes.

The damping factor is generally kept at 85% because it is assumed that there is about a 15% chance that a typical user will not follow any of the links on the page and instead navigate to a new random URL. Therefore, a self-contribution of 0.15 is added to the score of each page, as this randomness may result in the user landing on that page.

Now, let's revise the implementation of the PageRank algorithm:

1. Create a RDD that contains the pattern of how one page links to another:
```
sc.parallelize([('a','b'),('a','c'),('b','d'),('c','e'),('d','f'),('e','f'),('f','e'),('f','d'),('e','c'),('d','b'),('b','a'),('c','a')])
```

2. Now, group the values with the same key together using the groupByKey() operation:
```
links = links.groupByKey()
```

3. Map each key with an initial value of 1:
```
ranks = links.distinct().map(lambda x: (x[0],1))
```

4. Define the transition matrix as follows:
```
def nodecontributions(nodes,rank):

        number = len(nodes)

        for node in nodes:

                yield(node,rank/number)
```

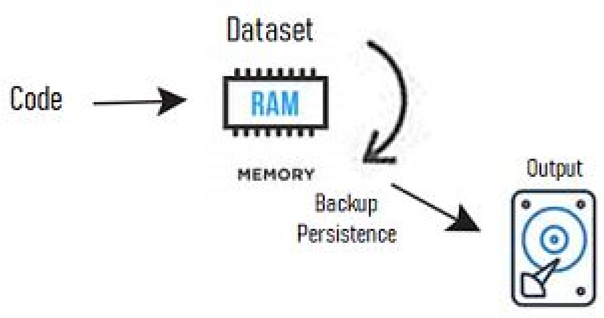
5. The following code helps you to run multiple iterations to evaluate the PageRank:
```
for i in range(3):

    contributions = links.join(ranks).flatMap(lambda x: nodecontributions(x[1][0], x[1][1]))

    ranks = contributions.reduceByKey(lambda x, y: x+y).mapValues(lambda x: 0.85 * x + 0.15)
```

6. Calculate and display the rank:
```
ranks.collect()
```

Storing RDDs in Spark

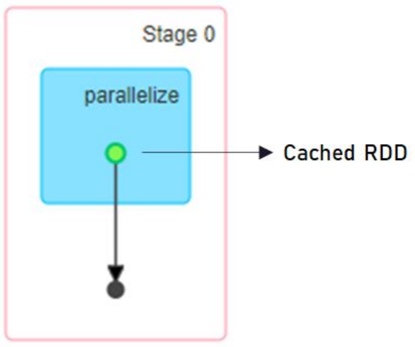
To overcome the issue of repeating the same steps, Spark provides the feature of caching and persisting. Caching and persisting Spark RDDs are common optimisation techniques that make reusing data on Spark much faster, especially in the case of iterative algorithms (machine learning algorithms) and interactive data exploration. You can store the frequently intermediate RDDs/outputs inside the Spark memory for future use.

The image below shows how the data is stored in RDDs:



You can store the intermediate RDDs using the following commands:

- **cache():** The cache() method is used when you want to store all the data inside the Spark memory. This helps in speeding up your queries, as Spark will not implement all the steps in the DAG every time. The cached RDD will be stored in the Spark storage memory for future use. The image below shows the DAG job showing a cached RDD:



However, the in-memory storage is limited, and the different tasks running over the cluster may require separate storage or computing space within the memory. This issue is resolved by the persist() method.

- **persist():** The persist() method allows you to store the intermediate RDDs in both the disk and the in-memory storage. This method offers the advantage of storing RDDs on either memory or disk, or both. The different storage levels available in the persist() method are as follows:

    1. **MEMORY_ONLY:** RDD is stored as a deserialised Java object in the memory. If the size of the RDD is more than the available memory, it will cache some partitions and recompute the remaining elements whenever needed.

2. **MEMORY_AND_DISK:** RDD is stored as a deserialised Java object in memory and disk. At this level, if the size of the RDD is greater than the available memory, Spark stores the excess partitions on the disk.
3. **MEMORY_ONLY_SER:** At this level, an RDD is stored as a serialised Java object in the memory. Hence, this level requires more space compared with the MEMORY_ONLY level, as it uses a fast serializer.
4. **MEMORY_AND_DISK_SER:** This level is similar to the MEMORY_ONLY_SER, but it spills the data partitions that do not fit in the memory to the disk, avoiding recomputing each time it is needed.
5. **DISK_ONLY:** At this storage level, RDD is stored only on disk. The space used for storage is low, the CPU computation time is high, and it makes use of on-disk storage.

If the RDDs are not removed from the memory manually, then Spark flushes out the cached RDDs automatically based on the last instance they were called. This is referred to as least recently used (LRU). Also, RDDs can only be cached at one level. If you try to store an RDD on multiple levels, then Spark will throw an error stating that the storage level of an RDD cannot be changed. You will have to unpersist the RDD from the previous level before you store it in a new one.

It is recommended that you use caching in the following situations:

- When an RDD is used in iterative machine learning applications
- When RDD computation is expensive (Here, caching can help reduce the cost of recovery in the case of executor failure.)