#0

Hello everyone,

It's been two years since I started my journey into this corporate world with core programming skills and a good understanding of computer fundamentals. My career began as a data engineer, where I found myself tasked with handling semi-structured files, transforming them, and loading them into a data warehouse.

While I initially grasped the knowledge of my project, I soon realised there was so much more to learn about the world of data engineering and big data. I was curious in knowing the background workings and architecture behind big data, eager to understand it in the bigger picture.

Through my ongoing exploration of core fundamentals, I'm continuously gaining insights into how big data operates and the intricate architecture that supports it. I'm excited to share my evolving understanding with you all here on this platform.

Here's to maintaining consistency and continuing to grow together!

Thanks,

Azzam"

has context menu

#1
What is BIG DATA..?

Well there are several informal definitions in describing Big Data.
Like - Data that cannot be stored/processed on a single computing system.

However, IBM provides a formal definition for Big Data.
"Data that is characterised by 3V's / 5V's is considered to be Big Data"
These V's are :
1. Volume
2. Variety
3. Velocity
4. Veracity
5. Value

1. Volume:

This refers to the vast amount of data generated and collected from various sources, including sensors, social media, business transactions, and more. This data is so huge that it cannot be stored/processed on a single conventional system.

2. Variety

Variety describes the diverse types and formats of data that make up big data. This includes
- Structured (e.g., databases)
- Semi - Structured (e.g., XML, JSON, CSV)
- Unstructured (e.g., text, images, videos)

3. Velocity

Velocity refers to the speed at which data is generated, collected, and processed.

E.g Amazon is running a sale and has to present the sales that have happened in the last 1 hour to its BI Team. System should be able to accommodate the new huge volumes of data coming in and generate some insights for Business Decisions, Like - How many iphones were sold in the last one hour.

4. Veracity

Veracity refers to the quality, reliability, and trustworthiness of the data. Data cannot always be in the right format and we should be able to handle this not so high quality data.

5. Value

Data having no Value is of no good to the company, unless you turn it into something useful. Data Should be able to generate some value out of the data collected by processing it for making business Decisions.

#2
Why do we have new technology stacks to handle big data?
It's because our traditional/conventional systems are not capable of handling Big Data.
To understand this, We should first know about Monolithic and Distributed System

Monolithic System:
It is a one big system holding a lot of **resources** and having high **computing** power.

Resources are nothing but
- Compute - CPU cores(e.g., 4 CPU cores)
- Memory - Ram (e.g., 8 gb RAM)
- Storage - Hard Disk (ex - 1 TB Hard Disk)

Ex- Let's assume a person is handling data with the above system specifications. After a period of time the data becomes big and the above specifications may not be able to handle this amount of data.
Then in monolithic we would usually increase resources (Vertical scaling) to.
Ex- 8 CPU cores, 16 gb RAM, 2 TB Hard Disk
And this person will repeat the same by increasing resources when the data is getting added.

Now the question is will the performance increase the same as resources get increased.? IT'S No..!
Ex- X resource gives Y Performance.
We assume that 2X resources give 2Y Performance.
But, X resource ~ Y Performance.
2X resources < 2Y Performance.
Distributed System:
A Distributed system is a network of computers with each holding its own resources that work together to achieve a common goal. Rather than relying on a single, centralised computer.
A distributed system distributes tasks across multiple computers, or nodes, connected through a network.
Ex - 5 Node cluster, with each Node/System holding 4 CPU Cores, 8 GB RAM and 1 TB Hard Disk.
Distributed System follows (Horizontal Scaling) that involves increasing the number of nodes/systems. Such a scaling is a true scaling to obtain the below result.
X resource ~ Y Performance.
2X resources ~ 2Y Performance.
Therefore, All the Big Data technology stacks are based on Distributed architecture to obtain true scaling.

#3 What is hadoop..?

Hadoop was one of the earliest and most influential Technology/FrameWork in the big data landscape.

- It is a framework because it is not just a single tool but a combination / ecosystem of several tools and technologies to solve Big Data problems.

3 Main components of Hadoop

- HDFS
- MapReduce
- YARN

⇒ **Hadoop Distributed File System (HDFS) -**

HDFS is a distributed file system designed to store large files and datasets across multiple nodes in a Hadoop cluster (Distributed System)

It is designed to store large files and datasets in a distributed manner across all datanodes of a cluster providing fault tolerance by replicating data across nodes.

⇒ **MapReduce -**

It is a distributed processing. Since the data is stored across multiple nodes, conventional programming models cannot process such data.

A MapReduce is a data processing tool which is used to process the data parallel in a distributed form.

⇒**Yet Another Resource Negotiator (YARN) -**

YARN is a resource management and job scheduling framework in Hadoop. It manages resources (CPU, memory, etc.) across the cluster and schedules jobs for execution.

Core Components of Hadoop:

Hadoop Distributed File System (HDFS)	**MapReduce**	**YARN**
For distributed Storage	For Distributed Processing	Resource Manager

** Writing MapReduce code is hard as it involves coding in JAVA. Which would require many lines of code even to perform a small task.

Therefore MapReduce is outdated at the moment and there are other alternatives that makes things easier, Which i will sharing in the upcoming posts **

#4 Hadoop Ecosystem technologies
Hadoop core components are HDFS, MapReduce, YARN

Apart from the core components of Hadoop, Ecosystem technologies comes into picture to make things easy for processing

Below are some of ecosystem technologies:
- Sqoop
- Pig
- Hive
- Oozie
- HBase

Before I start describing the above technologies.

Why do we have these ecosystems.?
- As I said in my earlier post, writing code in MapReduce is very hard, Which would require many lines of code even to perform a small task.
- Then these technologies come as a saviour.
- Instead of writing 100's of lines of code in MapReduce, you can achieve the same result by using the above technologies in a single **command** or **query** or **script**.
- Internally most of the ecosystem technologies write their code in MapReduce.

Sqoop : Used to transfer data between your HDFS and Relational Databases.
Pig : Is used to Analyze large DataSets in the Hadoop environment.
Hive : Hive allows users to read, write, and manage petabytes of data using SQL.
Oozie : it is a workflow scheduler system that manages Apache Hadoop jobs.
HBase : is a NoSQL database. It is used incases where very quick search of records from the data is required.

Don't you think learning all these EcoSystems are challenging? - Of-Course it is..!
Because all these Ecosystems have their own style of logic and interface. And each has its own learning Curve, Like for analyzing we need to learn Pig, for data querying we need to learn Hive and many more.
The Big data era started with Hadoop, But due to MapReduce is slow and hard, and has challenges. Hadoop is losing its position, But still HDFS and YARN are in use.
Since BigData technologies have evolved much better, It's up to the individual to learn the above technologies, Unless there is any requirement.

#5 What is Apache Spark

Apache spark came into picture to address the limitations of existing bigdata processing frameworks.

Concluding my previous post, MapReduce was a bottleneck as it was slow and needed a lot of coding effort even to achieve a simple task. And needed to learn different ecosystems to perform different Tasks.

This is where Apache spark comes into Rescue:

- Apache Spark is an open-source, distributed computing system designed for big data processing and analytics.
- It provides a unified framework for various data processing tasks, including batch processing, real-time streaming, machine learning, and interactive querying.

Is Apache spark replacement for Hadoop..?
Hadoop has 3 components:

- HDFS
- MapReduce
- YARN

Spark can act as a replacement for MapReduce but not Hadoop.

Therefore, Spark needs a **Storage** and **Resource Manager** to process and is not only bound to **HDFS** and **YARN**.

Spark need needs 2 components to work:

1. **Storage :** EX - HDFS | Amazon S3 | ADLS (Azure DataLake Storage) | Google cloud Storage etc.
2. **Resource Manager :** YARN | Apache Mesos | Kubernetes

Spark is 10x to 100x faster than traditional MapReduce as it stores and processes the data in-memory.

Spark supports multiple programming languages :
Scala | JAVA | Python | R

- However, Apache Spark itself is primarily implemented in Scala.

Spark with Python is called **Pyspark**

#6 Database vs data Warehouse vs Data Lake

Database, data warehouse, and data lake are all different types of data storage and management systems, each serving distinct purposes and designed to handle various types of data processing and analytics tasks.
- Here's an overview of each.

Database -(Structured Data)

- It deals with day to day transactional data.
- Meant for **OLTP**(Online Transactional Processing.
- DataBase majorly deals with Structured Data, Where the structure is in the form of rows and columns.
- It can only hold recent data for better performance. It cannot hold historical data of years.

Use Case : Online banking transactions, E-Commerce platforms..

Data Warehouse -(Structured Data)

- It deals with a lot of historical data of years to find insights.
- Data is brought in from multiple sources to the data warehouse.
- Data warehouse also deals with Structured Data and follows schema on write for validating the data.

Ex : Google BigQuery, Redshift, Teradata..

Data warehouse follows an ETL Process:

Suppose your data is in the database or in any other file format, And you have to bring it to the data warehouse for Analytical, Then the data needs to go through the following process.
Extract : Collect/Extract the data from various sources.
Transform : Once the data is extracted, it undergoes transformation to standardise, clean, and convert the data into a desired form.
Load : Load the data to the Data Warehouse.

Data Lake

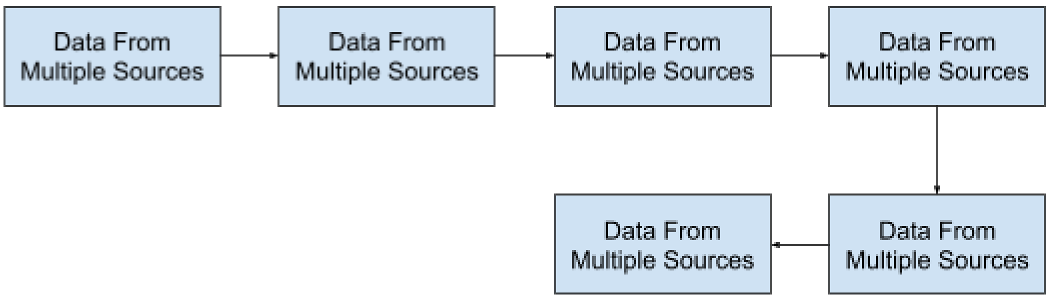
- A data lake is a centralized repository that allows storage of structured, semi-structured, and unstructured data at scale.
- Unlike data warehouses, which enforce schema-on-write, data lakes usually employ schema-on-read, meaning data is stored as-is and the schema is applied when it's queried.
- Data lakes are designed to store raw, unfiltered data from diverse sources, enabling organizations to perform advanced analytics, machine learning, and other data exploration tasks.

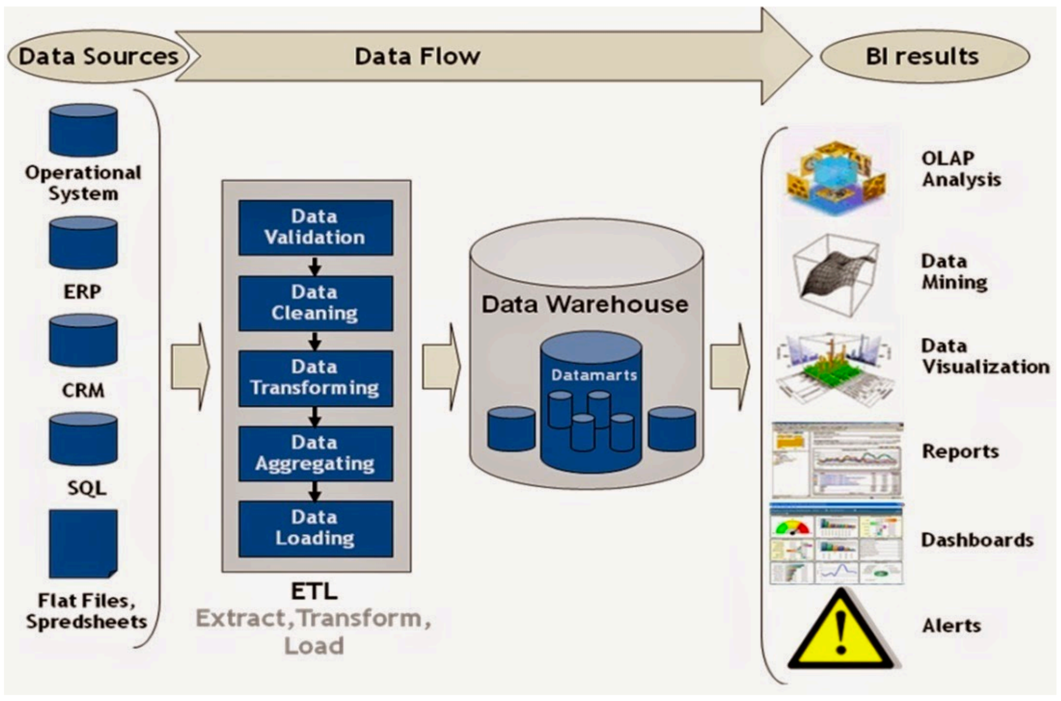
Ex : Amazon S3, ADLS (Azure Data lake storage), GCS (Google cloud Storage)
In Data lake, data has to go through **ELT** (Extract, Load, Transform) to do Analytics.

#7 Data Engineering Flow

Consider we are getting data from different sources like databases, sensor data, social media data etc..
Then the data has to go through the below steps:

1. All the data needs to be brought to the Storage from different sources. i.e.,(Data lake)
2. To bring the data to the storage, We need an ingestion framework to ingest the data from different sources. (ex : Apache kafka, Apache Sqoop)
3. After ingesting the data to storage we need to process the data, Here we are going through the ELT process. Where we load the data first and then perform various transformations (like Cleaning, Aggregations, joins etc..) as per the requirement
4. The processed results will be put into the serving layer to which the visualisation tools like Tableau, PowerBI are connected for analysing the data and viewing the results graphically.


Data From Multiple Sources
Data From Multiple Sources
Data From Multiple Sources
Data From Multiple Sources
Data From Multiple Sources
Data From Multiple Sources



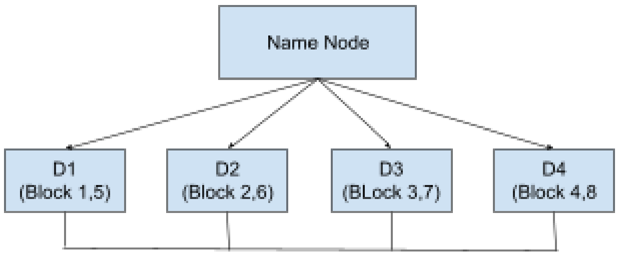
#8 HDFS

HDFS (Hadoop distributed file system) follows a master slave Architecture.
Master : Name Node
Slaves : Data Node

- **NameNode** is the centrepiece of the HDFS architecture. It manages the metadata and namespace of the file system.
- **DataNode** are the workhorses of HDFS. They store the actual data blocks of files.
- Files in HDFS are divided into fixed-size **blocks** (default block size 128 MB).
- Each block is replicated across multiple DataNodes for fault tolerance and reliability. The default replication factor is three, but it can be configured.

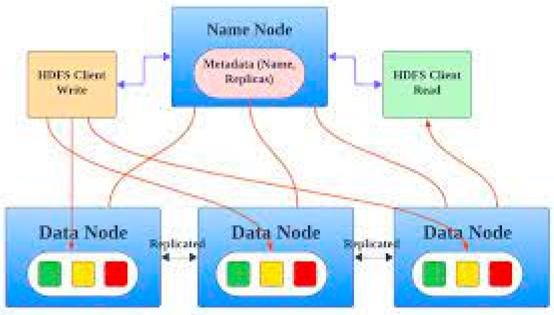
Example:
Assume we have **4 node** cluster. And we need to store file1.txt of size 1 GB.
File1.txt = 1 GB = 1024 MB
No of blocks = 1024/128 = 8 blocks

- Each Node can contain multiple blocks, Therefore, Out of 8 blocks each node stores 2 blocks, like below.



How is the client request processed? Say the client wants to read the above file1.txt

- When the client requests for the read, The request first goes to the name node.
- The name node checks its metadata table to find the location of actual data where it is stored.
- The name node passes the location information to the client,from Where the client can read the blocks.



# 9 Linux Commands

Certainly! Data engineers often work with large datasets and processing pipelines. Here's a list of essential Linux commands that every data engineer should know:

ls: List directory contents.
- Example: `ls -l` (long listing format), `ls -a` (including hidden files).

cd: Change directory.
- Example: `cd /path/to/directory`.

pwd: Print working directory.
- Example: `pwd`.

mkdir: Make directory.
- Example: `mkdir new_directory`.

rm: Remove files or directories.
- Example: `rm filename`, `rm -r directory` (recursive).

cp: Copy files or directories.
- Example: `cp file1 file2`, `cp -r directory1 directory2` (recursive).

mv: Move or rename files or directories.
- Example: `mv file1 file2`, `mv old_name new_name`.

touch: Create an empty file.
- Example: `touch new_file.txt`.

vi : Command to create a file with some content.
- Example: `vi filename`.

cat: Display file content and Concatenate.
- Example: `cat filename`.

head: Display the beginning of a file.
- Example: `head filename`.

tail: Display the end of a file.
- Example: `tail filename`.

grep: Search for patterns in files.
- Example: `grep pattern filename`.

find: Search for files or directories.
- Example: `find /path/to/search -name "pattern"`.

sort: Sort lines of text files.
- Example: `sort filename`.

uniq: Report or omit repeated lines in a file.
- Example: `uniq filename`.

wc: Display line, word, and byte count of a file.
- Example: `wc filename`.

du: Display disk usage of files and directories.
- Example: `du -sh directory`.

df: Display disk space usage for file systems.

- Example: `df -h`.

tar: Manipulate archives.

- Example: `tar -czvf archive.tar.gz directory` (create a compressed archive).

gzip/gunzip: Compress or decompress files.

- Example: `gzip filename`, `gunzip filename.gz`.

# 10 Key features of HDFS

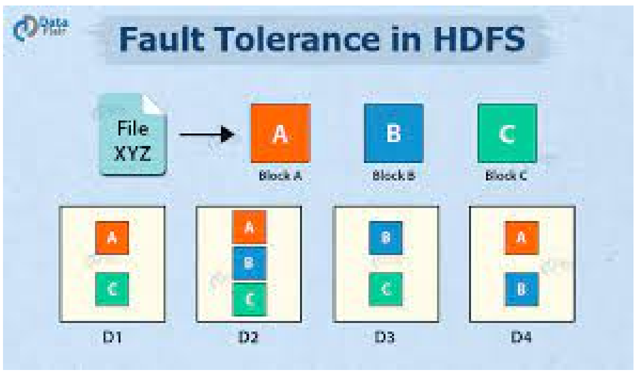
Name Node Federation:

- In the newer version of hadoop, Name node federation is introduced where there can be more than one Name mode to handle growing metadata.
- It prevents performance issues by distributing the workload across multiple NameNodes.
- This approach helps manage growing metadata more efficiently.
- Helps in achieving Scalability.

Fault Tolerance:

What if the data node fails..?

- Replication factor ensures copies of data are stored on multiple DataNodes.
- If a DataNode fails, data can still be accessed from replicated copies on other nodes.
- This redundancy helps maintain data availability and prevents loss in case of node failures.



What if the Name node fails..?

- Secondary Name node come into the picture to keep the system up and running.
- The Secondary NameNode regularly checks the state of the NameNode.
- In case of a failure, this checkpoint can be used to restore the state of the NameNode up to the last checkpoint, reducing the downtime.

#11 HDFS vs Cloud Data Lake

Cloud Alternatives of HDFS :

- AWS - Amazon S3
- Azure - ADLS Gen2 (Azure Data Lake Storage - Generation 2)

Difference b/w HDFS and Data lake in Cloud:

- **HDFS** is a distributed file system, And it stores data in block form.
- **ADLS Gen2** or **Amazon S3** is object based storage.

HDFS	Amazon S3 / ADLS Gen2
It is a distributed file system, where data is internally organized into blocks.	It is a storage system based on objects, where data is stored as discrete entities known as "Objects." Each object possesses a unique identifier, actual content or data, and metadata including access permissions and data characteristics.
HDFS lacks persistence, implying that data is lost upon termination of the Hadoop cluster due to its close integration with compute resources.	Amazon S3 and ADLS Gen2 offer persistence, ensuring that data remains intact even when the cluster is terminated.
Data is tightly coupled with compute, Means you need to pay for compute resources even if you only need storage.	In this scenario, storage is separated from compute, enabling payment solely for the utilized storage resources without the need to pay for compute.
When using HDFS, If you have 2 Hadoop clusters, Data b/w these 2 HDFS storage is not that easily possible.	With Amazon S3 / ADLS Gen2, multiple clusters have the capability to access the same data without constraints.

#12 MapReduce vs Apache Spark.

HDFS(storage)	MapReduce(Compute)	YARN(Resource Manager)

HDFS / AWS S3 / ADLS Gen 2..(storage)	SPARK(Compute)	YARN / Mesos / Kubernetes(Resource Manager)

Challenges of MapReduce:

- Less performance due to many IO disk seeks.
- Need to write many lines of Code to accomplish even a simple task.
- MapReduce Supports only Batch Processing.
- Learning curve is high.
- Constrained to always think in a Map-Reduce perspective.
- No Interactive mode

Spark is a Plug and Play Compute engine used for Distributed Processing.

Spark needs-

1. **Storage** (HDFS or Any DataLake)
2. **Resource Manager** (YARN / Mesos / Kubernetes)

Formal Definition of APache Spark :

Apache Spark is a multi language engine for executing data engineering, data science and machine learning on a single node or cluster.

Spark with Python - PySpark

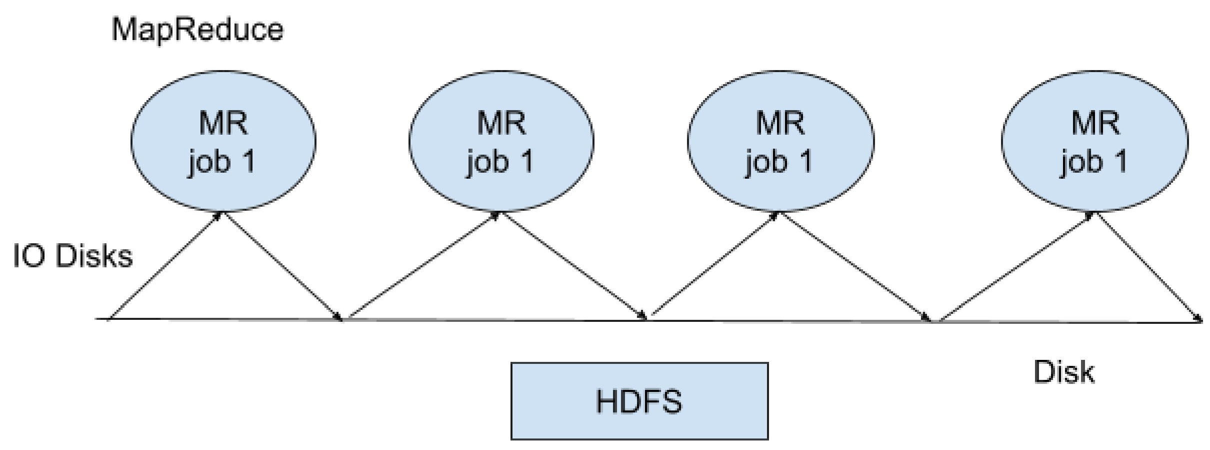
- Faster due to in-memory processing.
- Concise coding for complex tasks.
- Supports batch, streaming, and interactive processing.
- Easier learning curve.
- Offers more flexibility in problem-solving approaches.
- Well-suited for real-time and iterative processing.

#13 Why does Spark have high performance.?

The combination of in-memory processing and distributed computing in Apache Spark results in a system that offers high performance.

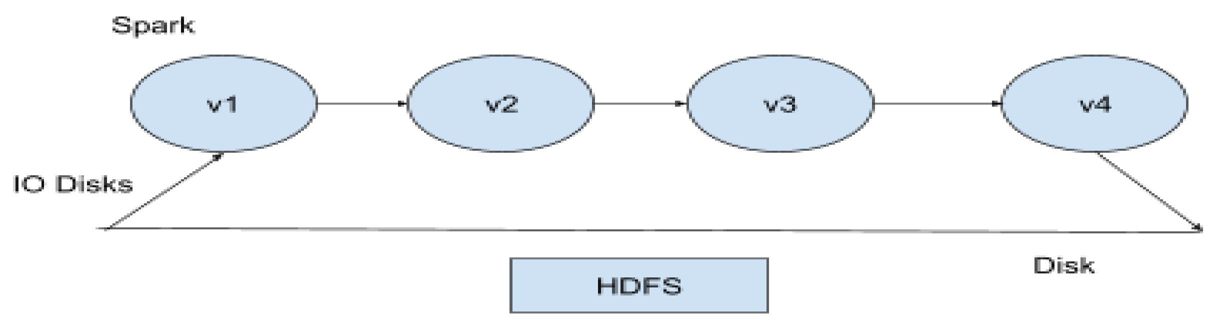
What is meant by In-Memory?

In case of MapReduce -



- Primarily relies on disk-based processing.
- Intermediate data is typically stored on disk between map and reduce phases, which can lead to disk I/O bottlenecks.

In Case of Spark -



- Spark, on the other hand, emphasizes in-memory processing.
- It keeps data in memory across multiple processing steps, which significantly reduces the need for disk I/O operations.
- This in-memory processing capability makes Spark well-suited for iterative algorithms,

SPARK

Apache Spark is an open-source cluster computing framework. Its primary purpose is to handle the real-time generated data.

- Spark was built on the top of the Hadoop MapReduce.
- It was optimized to run in memory whereas alternative approaches like Hadoop's MapReduce writes data to and from computer hard drives.
- So, Spark processes the data much quicker than other alternatives.

Features of Apache Spark

- **Fast** - It provides high performance for both batch and streaming data, using a state-of-the-art DAG scheduler, a query optimizer, and a physical execution engine.
- **Easy to Use** - It facilitates writing applications in Java, Scala, Python, R, and SQL. It also provides more than 80 high-level operators.
- **Generality** - It provides a collection of libraries including SQL and DataFrames, MLlib for machine learning, GraphX, and Spark Streaming.
- **Lightweight** - It is a light unified analytics engine which is used for large scale data processing.
- **Runs Everywhere** - It can easily run on Hadoop, Apache Mesos, Kubernetes, standalone, or in the cloud.

Before starting to learn Apache Spark, it's recommended to understand some basic concepts about Big Data and Hadoop. If you haven't explored these concepts yet, Go through the link below for basic fundamentals.

https://www.linkedin.com/feed/update/urn:li:activity:7172098113011134464/

#16 Spark Architecture A

The Spark follows the master-slave architecture. Its cluster consists of a single master and multiple slaves.

The Spark architecture depends upon two abstractions:

- Resilient Distributed Dataset (RDD)
- Directed Acyclic Graph (DAG)

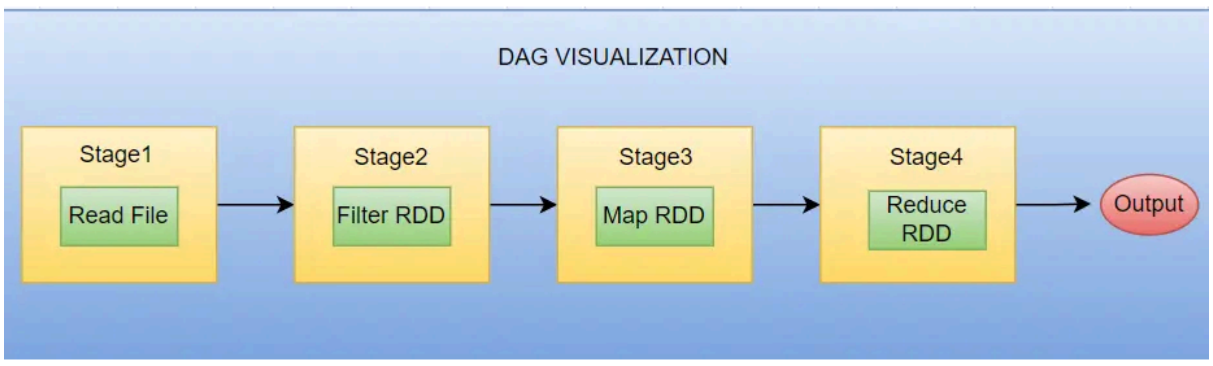
Resilient Distributed Datasets (RDD)

The Resilient Distributed Datasets are the group of data items that can be stored in-memory on worker nodes. Here,

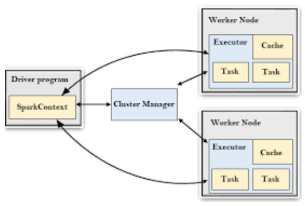
- Resilient: Restore the data on failure.
- Distributed: Data is distributed among different nodes.
- Dataset: Group of data.

Directed Acyclic Graph (DAG)

Directed Acyclic Graph is a finite directed graph that performs a sequence of computations on data. Each node is an RDD partition, and the edge is a transformation on top of data. Here, the graph refers to the navigation whereas directed and acyclic refers to how it is done.



#17 Spark Architecture - B



Cluster Manager: Spark can run on various cluster managers like Apache Mesos, Hadoop YARN, or Spark's own standalone cluster manager. The cluster manager allocates resources and manages the execution of Spark applications across the cluster.

Driver: The driver is the main process in a Spark application. It runs the user's main function and coordinates the execution of tasks in the cluster. It splits the application into smaller tasks and schedules them to be executed on worker nodes.

Worker Nodes: Worker nodes are machines in the cluster that execute tasks and store data. Each worker node is managed by a Spark Worker process, which is responsible for executing tasks assigned by the driver. Worker nodes also store data partitions in memory or on disk and communicate with the driver to report the status of tasks.

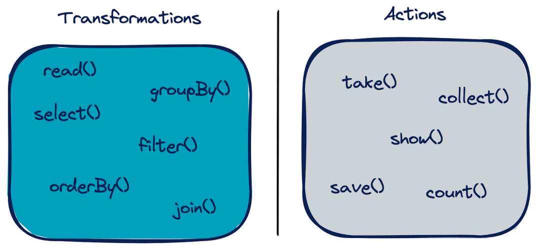
Executor: Executors are processes launched on worker nodes for executing tasks. Each executor is responsible for running tasks on a specific subset of worker nodes and caching data in memory, if needed. Executors are managed by the SparkContext and can run multiple tasks concurrently.

SparkContext: SparkContext is the entry point for Spark applications. It establishes a connection to the cluster manager and coordinates the execution of the driver program. SparkContext also manages the distribution of the code and data across the cluster, and it provides APIs for interacting with RDDs (Resilient Distributed Datasets) and other Spark features.

#18 RDD Operations

The RDD (Resilient Distributed Dataset) provides the two types of operations:

1. Transformation
2. Action



Transformation:

- Transformations are operations that create a new RDD from an existing one.
- They are lazy, meaning they don't compute their results right away but rather remember the series of transformations applied to the base RDD.

Action:

- Actions are operations that trigger computation and return results to the driver program or write data to external storage systems.
- They force the evaluation of the RDD transformations.

Why are transformations LAZY...?

Lazy evaluation in Apache Spark enables the construction of a Directed Acyclic Graph (DAG) representing the computation plan. This DAG captures the dependencies between RDDs and transformations, optimizing execution by allowing Spark to schedule and execute tasks efficiently across the cluster.

#19 Creating RDDs in Apache Spark: Various Methods and Examples

In Apache Spark, RDDs (Resilient Distributed Datasets) can be created from various data sources and through different methods. Here are some common ways of creating RDDs:

Parallelizing an Existing Collection:

You can create an RDD from an existing collection (e.g., list, tuple) by parallelizing it using the `parallelize()` method of the SparkContext (`sc`) object.

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

From External Data Sources:

RDDs can be created from external data sources such as text files, CSV files, JSON files, databases, HDFS, S3, etc. Spark provides built-in methods to read data from these sources.

```
# Create SparkContext
sc = SparkContext(appName="Read from S3")

# S3 bucket and file path
bucket = "your_bucket_name"
file_path = "s3a://{}/your_file_path".format(bucket)

# Read data from S3 into RDD
rdd = sc.textFile(file_path)
```

From Another RDD:

You can create a new RDD by transforming an existing RDD using various operations such as **`map()`**, **`filter()`**, **`flatMap()`**, etc.

```
rdd1 = sc.parallelize([1, 2, 3, 4, 5])
rdd2 = rdd1.map(lambda x: x * 2)
```

From Pair RDDs:

Pair RDDs are RDDs where each element is a key-value pair. They can be created using various transformations on existing RDDs.

```
data = [("a", 1), ("b", 2), ("c", 3)]
pair_rdd = sc.parallelize(data)
```

From Sequence of Numbers:

You can create an RDD containing a sequence of numbers using the **`range()`** function.

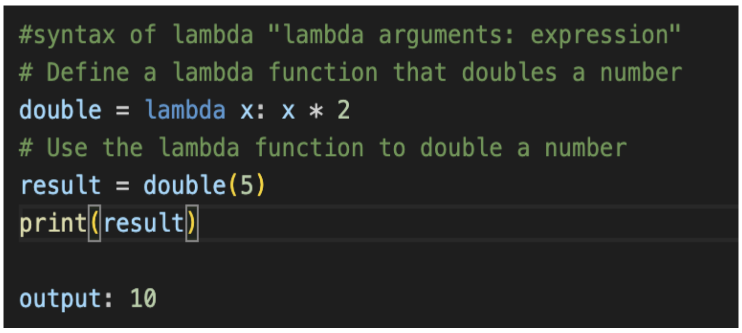
```
rdd = sc.range(1, 100, 2)  # RDD containing odd numbers from 1 to 99
```

These are some of the common ways to create RDDs in Apache Spark. Depending on your use case and the nature of your data, you can choose the appropriate method to create RDDs in your Spark application.

#20 lambda, map() and reduce() functions in python.

Lambda Functions:

- Lambda functions are anonymous functions defined using the **lambda** keyword.
- They are used to create small, inline functions without the need for a formal function definition.
- Lambda functions are particularly useful when you need a simple function for a short period of time.

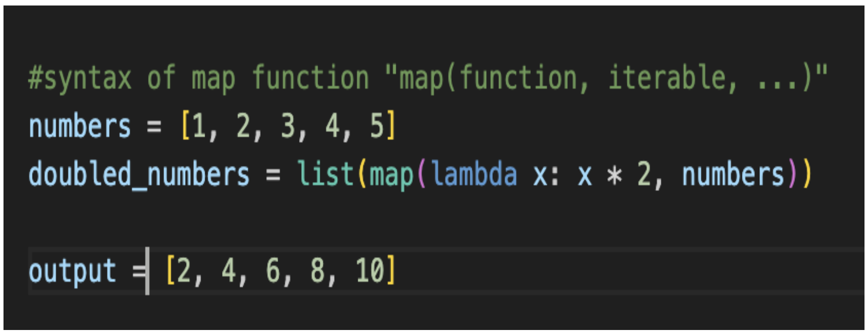


```
#syntax of lambda "lambda arguments: expression"
# Define a lambda function that doubles a number
double = lambda x: x * 2
# Use the lambda function to double a number
result = double(5)
print(result)

output: 10
```

Map Function:

- The **`map()`** function applies a given function to each item of an iterable (such as a list) and returns an iterator that yields the results.
- It takes two arguments: the function to apply and the iterable to apply it to. **Lambda** functions are commonly used with **`map()`** to apply simple transformations to every element of an iterable.

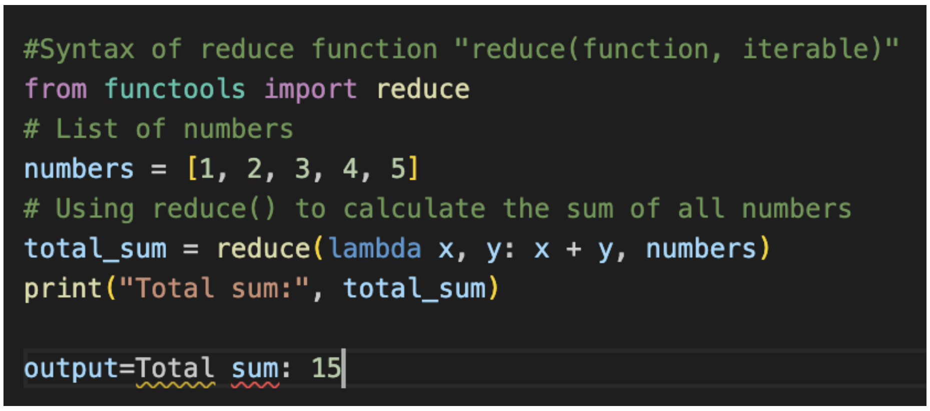


```
#syntax of map function "map(function, iterable, ...)"
numbers = [1, 2, 3, 4, 5]
doubled_numbers = list(map(lambda x: x * 2, numbers))

output = [2, 4, 6, 8, 10]
```

Reduce Function:

- The **`reduce()`** function applies a rolling computation to sequential pairs of values in an iterable.
- It takes two arguments: the function to apply and the iterable to apply it to. The function should accept two arguments and return a single value.
- Lambda functions are often used with **`reduce()`** to specify the computation to be performed.



```
#Syntax of reduce function "reduce(function, iterable)"
from functools import reduce
# List of numbers
numbers = [1, 2, 3, 4, 5]
# Using reduce() to calculate the sum of all numbers
total_sum = reduce(lambda x, y: x + y, numbers)
print("Total sum:", total_sum)

output=Total sum: 15
```

Understanding Python concepts like **`map()`**, **`reduce()`**, and lambda functions can be beneficial before diving into Spark RDDs (Resilient Distributed Datasets).

#21 map() and reduce () in RDD's

In Spark RDDs (Resilient Distributed Datasets), **`map()`** and **`reduce()`** are fundamental operations for transforming and aggregating data across distributed systems. Here's how you can use **`map()`** and **`reduce()`** in RDDs:

`map()` in RDDs:

The **`map()`** function in Spark RDDs applies a given function to each element of the RDD and returns a new RDD consisting of the results. It transforms each element in the RDD into zero or more elements using the provided function.

```
# Create an RDD
rdd = sc.parallelize([1, 2, 3, 4, 5])

# Use map() to double each element
doubled_rdd = rdd.map(lambda x: x * 2)

# Collect the results
print(doubled_rdd.collect())
```

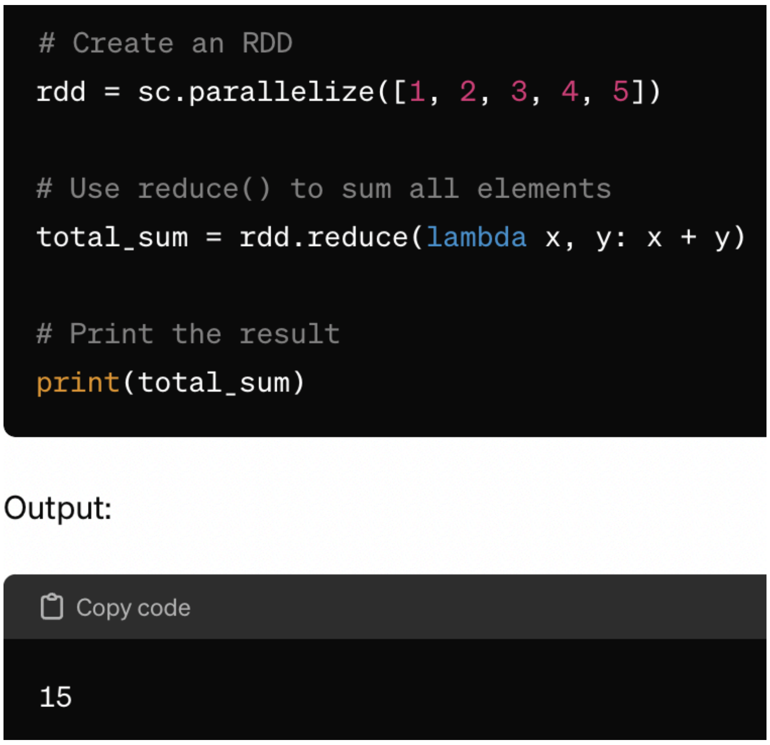
Output:

```csharp
[2, 4, 6, 8, 10]
```

`reduce()` in RDDs:

The **`reduce()`** function in Spark RDDs aggregates the elements of the RDD using a specified function. It takes each element and combines them pairwise until only a single result remains.

Example:



```python
# Create an RDD
rdd = sc.parallelize([1, 2, 3, 4, 5])

# Use reduce() to sum all elements
total_sum = rdd.reduce(lambda x, y: x + y)

# Print the result
print(total_sum)
```

Output:

```
15
```

These are basic examples, but `map()` and `reduce()` can be applied to more complex transformations and aggregations in Spark RDDs, making them powerful tools for distributed data processing.

#22 RDD Transformation and Action Operations Example with PySpark -A

Here's an example code demonstrating how to create an RDD and apply various transformations and actions

- Create a spark session and RDD (using parallelize)

```python
from pyspark.sql import SparkSession

# Create SparkSession
spark = SparkSession.builder \
    .appName("RDD Example") \
    .getOrCreate()

# Create an RDD from a list of tuples
data = [("apple", 10), ("banana", 5), ("apple", 20), ("banana", 15), ("cherry", 8)]
rdd = spark.sparkContext.parallelize(data)
```

- Take method to print the number of desired results

```python
# Take and print the first few elements of the RDD
print("First few elements of RDD:")
print(rdd.take(3))  # Take the first 3 elements
```

O/P:

```
First few elements of RDD:
[('apple', 10), ('banana', 5), ('apple', 20)]
```

- Applying **map()** and **Reduce()**

```
# Map operation to double the values
doubled_rdd = rdd.map(lambda x: (x[0], x[1] * 2))

# ReduceByKey operation to sum values with the same key
sum_rdd = doubled_rdd.reduceByKey(lambda x, y: x + y)

# Take and print the first few elements of doubled_rdd
print("First few elements of doubled_rdd:")
print(doubled_rdd.take(3))

# Take and print the first few elements of sum_rdd
print("\nFirst few elements of sum_rdd:")
print(sum_rdd.take(3))
```

O/P:

```
First few elements of doubled_rdd:
[('apple', 20), ('banana', 10), ('apple', 40)]

First few elements of sum_rdd:
[('apple', 60), ('banana', 25), ('cherry', 16)]
```

#23 RDD Transformation and Action Operations Example with PySpark -B

Continuing from the previous post by using the same RDD created. If you **haven't** gone through the post **A** here is the link **<post 21 link>**

- **sortBy()** operation to sort the RDD's by values in descending order.

```
sorted_rdd = sum_rdd.sortBy(lambda x: x[1], ascending=False)

# Take and print the first few elements of sorted_rdd
print("First few elements of sorted_rdd:")
print(sorted_rdd.take(3))
```

Output:

```less
First few elements of sorted_rdd:
[('apple', 60), ('banana', 25), ('cherry', 16)]
```

- Count number of elements in RDD

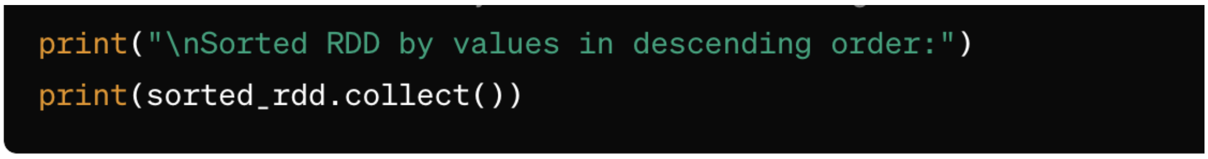
```
count = sorted_rdd.count()

# Print the count
print("Number of elements in sorted_rdd:", count)
```

Output:

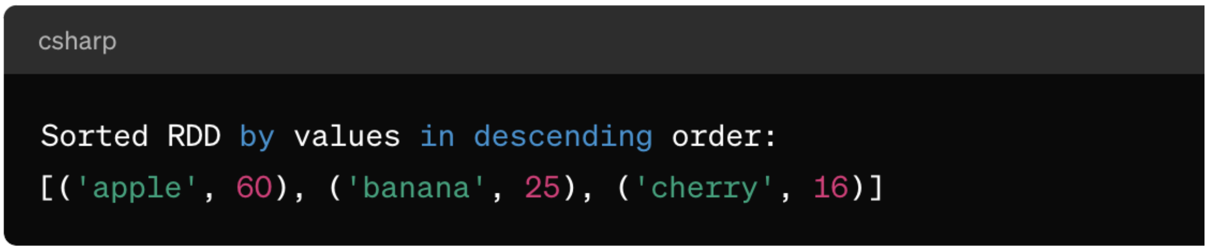
```javascript
Number of elements in sorted_rdd: 3
```

- Printing the results using collect()



```
print("\nSorted RDD by values in descending order:")
print(sorted_rdd.collect())
```

Output:



```csharp
Sorted RDD by values in descending order:
[('apple', 60), ('banana', 25), ('cherry', 16)]
```

Next week:

1. Different transformations in rdd
2. Transformations vs action
3. Shuffling and sorting
4. Narrow vs wide transformation
5. Reduce vs reduce by key()

#24: 10 Majorly Used Transformations in RDDs (Resilient Distributed Datasets)

Certainly! Here are 10 majorly used transformations in RDDs (Resilient Distributed Datasets) in Apache Spark:

map(func):

- Applies a function to each element of the RDD.
- Example: rdd.map(lambda x: x * 2)

filter(func):

- Filters elements based on a predicate function.
- Example: rdd.filter(lambda x: x % 2 == 0)

flatMap(func):

- Similar to map, but each input item can be mapped to 0 or more output items.
- Example: rdd.flatMap(lambda x: (x, x*2))

reduceByKey(func):

- Combines values with the same key using a specified reduce function.
- Example: rdd.reduceByKey(lambda x, y: x + y)

groupByKey():

- Groups the values for each key in the RDD into a single sequence.
- Example: rdd.groupByKey()

sortByKey():

- Sorts the RDD by key.
- Example: rdd.sortByKey()

join(otherRDD):

- Performs an inner join between two RDDs based on their keys.
- Example: rdd1.join(rdd2)

distinct():

- Returns a new RDD containing distinct elements from the original RDD.
- Example: rdd.distinct()

mapPartitions(func):

- Similar to map, but operates on each partition of the RDD separately.
- Example: rdd.mapPartitions(lambda partition: [x*2 for x in partition])

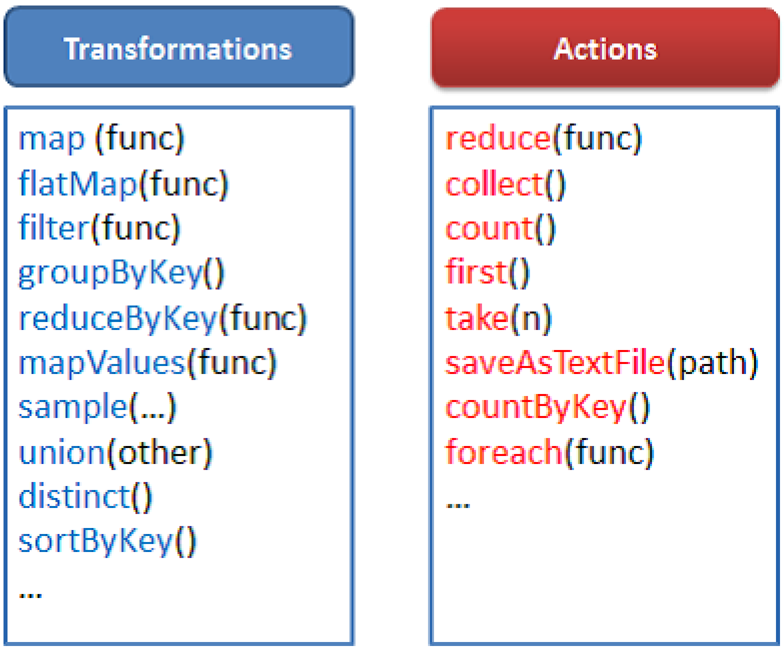
cogroup(otherRDD):

- Groups the values for each key in both RDDs and performs a cogroup operation.
- Example: rdd.cogroup(otherRDD)

These transformations are commonly used in Spark applications for various data processing tasks, such as filtering, mapping, aggregating, joining, and sorting data distributed across a cluster.

#25: Transformation vs Action in Apache Spark

In Apache Spark, there are two types of operations that can be applied to RDDs (Resilient Distributed Datasets): transformations and actions. Here's a breakdown of each:



Transformations	Actions
map (func)	reduce(func)
flatMap(func)	collect()
filter(func)	count()
groupByKey()	first()
reduceByKey(func)	take(n)
mapValues(func)	saveAsTextFile(path)
sample(...)	countByKey()
union(other)	foreach(func)
distinct()	...
sortByKey()	
...	

Transformations:

- Transformations create a new RDD from an existing RDD.
- However, transformations are lazy, meaning they do not compute their results immediately. Instead, they create a lineage graph (DAG) representing the sequence of transformations applied to the base dataset.
- Spark keeps track of these transformations and only computes them when an action is called.
- This lazy evaluation allows Spark to optimize the execution plan.

Common transformations include `map()`, `filter()`, `flatMap()`, `reduceByKey()`, `join()`, `groupByKey()`, etc. These transformations typically perform data processing tasks like filtering, mapping, aggregating, joining, and sorting data.

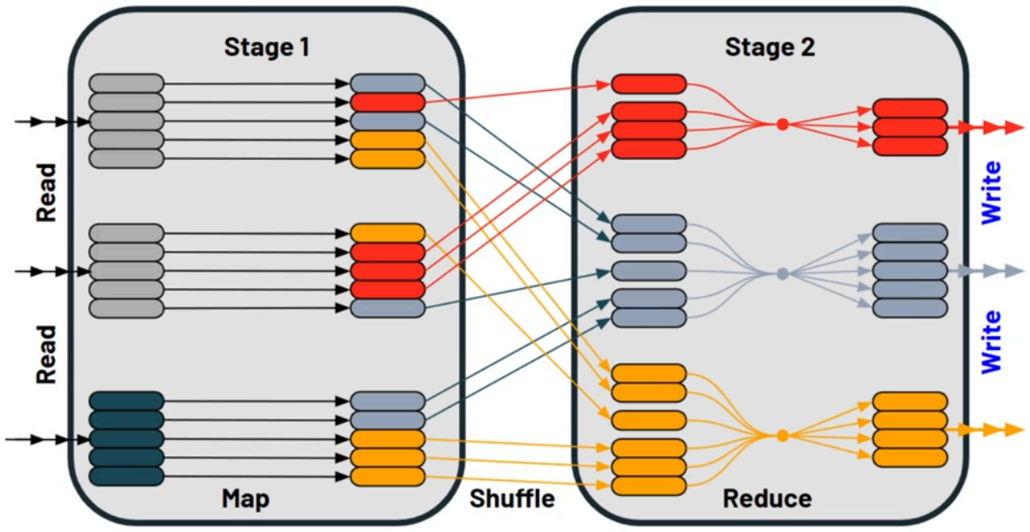
Actions:

- Actions, on the other hand, trigger the execution of the transformations and produce some output.
- When an action is called on an RDD, Spark evaluates the lineage graph and computes the result, which might involve executing the transformations on the distributed dataset across the cluster.
- Actions are the operations that initiate the actual computation and return the results to the driver program or write data to external storage.
- Examples of actions include `collect()`, `count()`, `reduce()`, `take()`, `saveAsTextFile()`, `foreach()`, etc.
- These actions perform tasks such as collecting data to the driver, counting elements in an RDD, reducing elements to a single result, taking a sample of data, saving RDDs to external storage, or executing a function on each element of the RDD.

In summary, transformations are used to build a directed acyclic graph (DAG) of computation, describing how data is transformed from one RDD to another, while actions execute the computations and produce final results or write data to external storage.

#26: Shuffling and Sorting in Apache Spark

Shuffling and sorting are fundamental operations in Apache Spark, especially in distributed data processing. They play crucial roles in various data transformations and actions.



Shuffling:

- Definition: Shuffling refers to the process of redistributing data across the partitions of an RDD. It involves moving data between nodes in the cluster to perform operations that require data from multiple partitions.
- Occurrence: Shuffling typically occurs when operations like `groupByKey()`, `reduceByKey()`, `join()`, and `sortByKey()` are executed. These operations may require data to be reorganized across the cluster to group or aggregate by key, or to perform joins across different datasets.
- Performance Impact: Shuffling incurs network overhead as data is transferred between nodes, making it one of the costliest operations in Spark. Minimizing shuffling is crucial for optimizing Spark jobs.

- Optimizations: Spark provides various optimizations to minimize shuffling, such as partitioning strategies, shuffle file consolidation, and data skew handling.

Sorting:

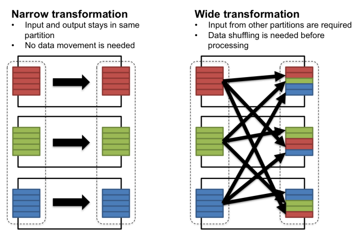
- Definition: Sorting involves arranging the elements of an RDD in a specific order, typically based on a key or a custom comparator function.
- Occurrence: Sorting is often required for operations like `sortByKey()`, `sortBy()`, or when performing certain types of joins and aggregations that necessitate ordered data.
- Performance Impact: Sorting can be computationally expensive, especially when dealing with large datasets. It requires data to be shuffled across partitions and then sorted within each partition.
- Optimizations: Spark employs various optimizations to improve sorting performance, such as using efficient sorting algorithms (e.g., Timsort), leveraging partitioning strategies to minimize data movement during sorting, and parallelizing sorting tasks across the cluster.

Conclusion:

Shuffling and sorting are essential operations in Apache Spark, enabling various transformations and actions on distributed datasets. While they are powerful, they can also have significant performance implications, particularly in terms of network and computational overhead. Understanding when and how shuffling and sorting occur, as well as employing optimization techniques, is crucial for building efficient and scalable Spark applications.

#27 Narrow vs Wide Transformations in Spark

In Apache Spark, transformations are broadly categorized into two types based on how they operate across partitions of an RDD (Resilient Distributed Dataset): narrow transformations and wide transformations.


Narrow transformation
• Input and output stays in same partition
• No data movement is needed
Wide transformation
• Input from other partitions are required
• Data shuffling is needed before processing

Narrow Transformations:

- Definition: Narrow transformations are those where each input partition contributes to only one output partition, i.e., each output partition depends on a single input partition.
- Operation: Narrow transformations operate on a single partition of the parent RDD to compute the output partition.
- Example: `map()`, `filter()`, `flatMap()`, `mapPartitions()`, `filter()`, etc.
- Characteristics:
    - Narrow transformations are executed independently on each partition, without needing to shuffle or exchange data across partitions.
    - They result in RDDs with the same number of partitions as the parent RDD.
    - Narrow transformations are more efficient and faster to execute because they don't involve data movement or shuffling across the cluster.

Wide Transformations:

- Definition: Wide transformations are those where each input partition contributes to multiple output partitions, i.e., each output partition may depend on multiple input partitions.
- Operation: Wide transformations may require data shuffling and exchange across partitions to perform operations such as groupings, aggregations, or joins.
- Example: `groupByKey()`, `reduceByKey()`, `sortByKey()`, `join()`, `cogroup()`, `sortBy()`, `distinct()`, etc.
- Characteristics:
  - Wide transformations often involve reorganizing or redistributing data across the cluster, leading to shuffling and exchanging data between partitions.
  - They can result in RDDs with a different number of partitions than the parent RDD, depending on the operation performed.
  - Wide transformations are generally more computationally expensive and time-consuming compared to narrow transformations due to the involvement of data shuffling and network communication.

Conclusion:

Understanding the distinction between narrow and wide transformations is crucial for designing efficient Spark applications. Minimizing the use of wide transformations and optimizing their performance, such as through appropriate partitioning strategies, can significantly enhance the efficiency and scalability of Spark jobs, particularly in large-scale distributed data processing scenarios.

#28: reduce VS reduceByKey in Apache Spark RDDs:

reduce and reduceByKey are two distinct operations available in Apache Spark, a distributed computing framework for big data processing.

Reduce:

- reduce is an action that collapses the elements of an RDD (Resilient Distributed Dataset) into a single result.
- It applies a function that takes two elements and returns a single element of the same type.
- The function should be associative and commutative, meaning it can be applied in any order.
- It operates on the entire RDD.

Example:

```
# Python Spark code
rdd = sc.parallelize([1, 2, 3, 4, 5])
result = rdd.reduce(lambda x, y: x + y)
print(result)  # Output: 15 (1 + 2 + 3 + 4 + 5)
```

reduceByKey:

- reduceByKey is a transformation that combines values with the same key in a Pair RDD using a specified associative and commutative function.
- It operates on Pair RDDs, where each element is a key-value pair.
- It reduces values with the same key to a single value using the provided function.
- Example:

```python
# Python Spark code
rdd = sc.parallelize([('a', 1), ('b', 2), ('a', 3), ('b', 4)])
result = rdd.reduceByKey(lambda x, y: x + y)
print(result.collect())  # Output: [('a', 4), ('b', 6)]
```

- In this example, the values for each key are summed up.

In summary, while both reduce and reduceByKey perform reduction operations, reduce operates on the entire RDD, collapsing it to a single result, whereas reduceByKey works on Pair RDDs, reducing values with the same key to a single value.

#29 ReduceBy() key vs groupBy() key in spark spark RDD

In the context of Apache Spark's Resilient Distributed Datasets (RDDs), both reduceByKey and groupByKey are transformation operations used for processing data. However, they operate differently and have distinct performance characteristics:

reduceByKey:

- reduceByKey is a transformation that performs a reduction operation (like sum, average, max, etc.) on the values of each key in the RDD.
- It aggregates the values of each key using an associative and commutative function, reducing the data before shuffling it across partitions.
- It's preferred over groupByKey when you're aggregating values for each key, as it reduces data movement during shuffling, leading to better performance.

Example: `rdd.reduceByKey(lambda x, y: x + y)`

groupByKey:

- groupByKey is a transformation that groups the values of each key in the RDD into an iterable.
- It collects all the values associated with each key and forms a list (or iterable) of those values.
- It can be inefficient when working with large datasets because it shuffles all the data across partitions regardless of the volume of data per key.
- It's typically used when you need to perform complex operations that involve iterating over all the values associated with each key.

Example: `rdd.groupByKey()`

In summary, reduceByKey is generally preferred over groupByKey due to its better performance characteristics, especially for aggregation operations. However, groupByKey might still be useful for certain scenarios where you need access to all values associated with a particular key and can't express the computation using reduceByKey.

#30 Task job and stage in spark

In Apache Spark, jobs, tasks, and stages are fundamental concepts that play a crucial role in the distributed execution of computations. Here's an overview of each:

Job:

- A job in Spark refers to a complete computation triggered by an action, such as collect(), saveAsTextFile(), or count().
- When an action is called on an RDD, Spark prepares to execute one or more jobs to fulfill that action.
- Each job consists of one or more stages.

Stage:

- A stage is a logical division of a job's computation, corresponding to a sequence of transformations that can be executed without shuffling data across the network.
- Stages are determined by the presence of shuffle operations (e.g., reduceByKey, groupByKey, sortByKey) or data partitioning operations (e.g., repartition, partitionBy).
- Stages are further divided into tasks for actual execution.

Task:

- A task is the smallest unit of work in Spark and represents the actual execution of a computation on a single partition of data.
- Each task corresponds to a single partition of an RDD and performs the transformations defined in the stage it belongs to.
- Tasks are executed in parallel across the worker nodes in the Spark cluster.
- Tasks are created for each partition of data in the RDD being operated on within a stage.

When you submit a Spark application, it is divided into multiple stages, and each stage is further divided into tasks. These tasks are then scheduled and executed across the available resources in the Spark cluster. The division into stages allows Spark to optimize the execution plan by minimizing data shuffling and maximizing parallelism.

Understanding these concepts is crucial for optimizing Spark applications, as inefficiencies in job, stage, or task execution can lead to longer processing times or resource wastage.

#31 partitions in spark

In Apache Spark, partitions are the basic units of parallelism and data distribution. When you create an RDD (Resilient Distributed Dataset) or DataFrame in Spark, it is divided into multiple partitions, with each partition containing a subset of the data. Understanding partitions is crucial for optimizing Spark jobs and improving performance. Here's a closer look at partitions:

Definition:

- A partition in Spark represents a logical division of the dataset.
- Partitions are the fundamental units of parallelism in Spark, as computations are performed independently on each partition.
- Spark ensures fault tolerance and parallelism by distributing partitions across the cluster's nodes.

Role:

- Partitions enable parallel processing of data across multiple nodes in a Spark cluster.
- They facilitate parallel execution of tasks, allowing Spark to efficiently utilize the available computational resources.
- By dividing the data into partitions, Spark can process different portions of the dataset simultaneously, improving overall performance.

Properties:

- The number of partitions in an RDD or DataFrame can be determined by calling the getNumPartitions() method.
- Partitions are immutable and typically represent subsets of data that can be processed independently.

- Spark automatically determines the default number of partitions when creating RDDs or DataFrames. However, you can also specify the number of partitions explicitly when creating RDDs or performing transformations.

Control:

- You can control the number of partitions in Spark RDDs or DataFrames using methods like repartition() or coalesce().
- repartition(n) increases or decreases the number of partitions to n by shuffling data across the cluster.
- coalesce(n) decreases the number of partitions to n without shuffling data, if possible, by merging partitions.

Optimization:

- Proper partitioning is essential for optimizing Spark jobs. It affects data locality, task distribution, and overall performance.
- Spark tries to maintain data locality by processing data on the same node where it's stored whenever possible, reducing data movement across the cluster.

In summary, partitions are the building blocks of parallelism in Apache Spark, allowing efficient distribution and processing of data across the nodes in a cluster. Understanding and optimizing partitions are critical for achieving optimal performance in Spark applications.

#32 Repartition vs coalsece

repartition() and coalesce() are both methods in Apache Spark used to manage the number of partitions in an RDD or DataFrame. However, they differ in their behavior and use cases:

repartition():

- repartition() is used to increase or decrease the number of partitions in an RDD or DataFrame. It involves a full shuffle of the data across the cluster.
- When you call repartition(n), Spark evenly redistributes the data into n partitions, regardless of the current number of partitions.
- It's typically used when you want to increase the level of parallelism or when you want to explicitly control the number of partitions, for example, before performing an operation that benefits from a specific partitioning scheme.
- Example: df.repartition(10)

coalesce():

- coalesce() is used to decrease the number of partitions in an RDD or DataFrame. It performs a narrow transformation and tries to minimize data movement by merging partitions on the same worker node if possible.
- Unlike repartition(), coalesce() does not involve a full shuffle of the data across the cluster unless you explicitly set the shuffle parameter to True.
- It's typically used when you want to reduce the number of partitions to optimize performance or when you know that the data distribution is skewed and you want to reduce the number of partitions without incurring the overhead of a full shuffle.
- Example: df.coalesce(5)

In summary, repartition() is used to increase or decrease the number of partitions with a full shuffle, while coalesce() is primarily used to decrease the number of partitions with a narrow transformation and optional shuffle. coalesce() is more efficient when reducing the number of partitions without significant data movement.

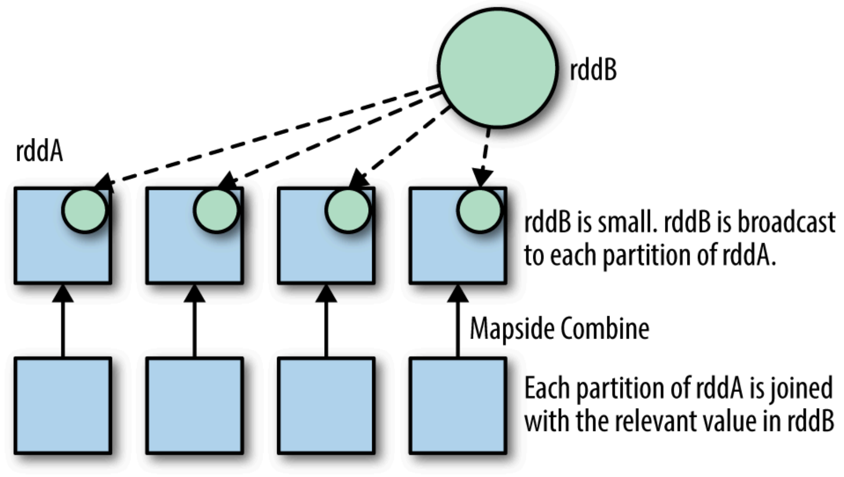
#33 what is broadcast join in spark

In Apache Spark, a "broadcast join" is a type of join operation used to optimize performance when joining large and small datasets.

When performing a join operation between two datasets, Typically normal join is a wide transformation and shuffles data across the network to ensure that matching records are brought together.

However, if one of the datasets is small enough to fit entirely in memory on each executor node, it can be more efficient to broadcast that dataset to all executor nodes rather than shuffling it across the network.

This is particularly useful when joining a large dataset with a small dataset, as it reduces the amount of data that needs to be shuffled and improves performance.


rddB
rddA
rddB is small. rddB is broadcast to each partition of rddA.
Mapside Combine
Each partition of rddA is joined with the relevant value in rddB

Here's how a broadcast join works in Spark:

1. The **larger** dataset is distributed across the cluster.
2. Whereas, the **smaller** dataset, the complete copy of the dataset, is made available on every machine on the cluster.
3. This hugely reduces the shuffling of Data and thereby optimizing the performance.

Ex: spark.sparkContext.broadcast(<dataset name>.collect())

Broadcast joins can significantly improve the performance of join operations, especially when dealing with large datasets, by minimizing data shuffling and network communication overhead. However, it's important to use broadcast joins judiciously, as broadcasting large datasets can consume a significant amount of memory and may lead to out-of-memory errors if not managed properly.