**Apache Spark**

1. What is Apache Spark?

* Apache spark is open-source unified computing engine which has support of multiple libraries for parallel data processing on cluster of computers
* Apache spark is processing massive amounts of data parallelly on cluster.
* Spark is limited to only computing the data it doesn’t store any data, but it provides connector support to connect multiple sources like HDFS, ODBC/JDBC and Asure storage etc.
* Let’s understand with example that how Apache spark works :

Imagine you have a large pile of books (which represents a huge amount of data) and you want to find out how many times the word "cat" appears in all the books. Doing this manually would take a long time, right? Even with a computer, if it’s just one machine processing the data, it could still be slow.

This is where **Apache Spark** comes in. Think of Spark as a team of super-fast computers that work together. Instead of one computer reading and counting the word “cat” in all the books, Spark splits the books among many computers (called a cluster), and they all work at the same time. This means the work gets done much faster!

**Key Features:**

* **Parallel Processing:** Spark divides the work and distributes it across many computers.
* **In-Memory Computing:** Spark stores data in memory (RAM), making it faster than other tools that read from disks repeatedly.
* **Supports Multiple Languages:** You can write Spark programs in languages like Python, Java, and Scala.

1. What is RDD? How RDD is created and how it is useful in Spark?

* RDD (**Resilient Distributed Dataset**) is a fundamental data structure in Spark.

It is a distributed collection of objects that can be operated on in parallel across multiple nodes in a cluster.

* RDD provides fault tolerance ,which means that they can recover from the node failure.
* This achieves through the Lineage information ,which allow spark to recompute lost data from the previous transformation.
* **Immutable**: Once created, an RDD cannot be changed. Any transformation creates a new RDD.
* **Distributed**: Data is divided into partitions and processed across different nodes.
* **Resilient**: Fault-tolerant, meaning it can recover from failures automatically.
* RDDs support two types of operations:
* **Transformations**: Functions like map(), filter(), or flatMap() that return a new RDD based on the previous one.
* **Actions**: Functions like count(), collect(), or saveAsTextFile() that return a result to the driver program or write data to storage.
* **How RDD is Created in PySpark:**

In PySpark, an RDD can be created in a few different ways:

1. **From an existing collection** (e.g., a Python list or an array).
2. **From external storage** like a file stored in HDFS, S3, or a local filesystem.
3. **From transformations** on other RDDs (e.g., by filtering or mapping over an existing RDD).

from pyspark import SparkContext

# Initialize Spark

sc = SparkContext("local", "Create RDD Example")

# **Create an RDD from a Python list**

data = [1, 2, 3, 4, 5]

rdd = sc.parallelize(data)

# **Show the data in RDD**

print(rdd.collect())

# **Create an RDD from a text file**

file\_rdd = sc.textFile("path/to/file.txt")

**# Create an RDD from a Python list**

numbers = sc.parallelize([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])

**# Apply a transformation (filter) to create a new RDD**

even\_numbers = numbers.filter(lambda x: x % 2 == 0)

**# Show the even numbers**

print(even\_numbers.collect())

**Summary of Benefits of RDD in PySpark:**

* **Parallel processing**: RDDs are split into partitions and processed simultaneously across multiple nodes.
* **Fault tolerance**: RDDs can automatically recover from node failures.
* **Lazy evaluation**: Transformations are only computed when an action is called.
* **In-memory computing**: RDDs can store intermediate results in memory, speeding up iterative algorithms.
* **Scalability**: RDDs can handle large datasets across distributed systems with minimal coding effort.

**Summary:**

* **Spark decides the number of partitions** based on factors like the file size, file system block size, and available cores.
* **By default**, Spark creates partitions based on the block size of the file system (typically 128 MB in HDFS).
* **You can manually control** the number of partitions using options like sc.textFile("file.txt", minPartitions) or adjust them later using repartition() or coalesce().

1. What are the roles and responsibilities of **Driver** in Apache spark Architecture?

* The **Spark Driver** is essentially the brain or the **master node** of a Spark application. It is the **program that runs your main code** (the one you write) and is responsible for coordinating the work across the cluster. It manages all the distributed workers (called executors) that process the data.

1. **Starting the Spark Application**: Initiate spark session and spark context.
2. **Job Scheduling** : Break down Jobs into Stage and Tasks.
3. **Task Distribution** : Send tasks to worker nodes(Executors) for parallel processing.
4. **Lazy Evaluation** : Keeps track of Transformations and Actions and create execution plan
5. **Fault Tolerance** : Track task execution and reschedules if it fails.
6. **Collecting Results** : Gather result from the worker nodes and return to the application.
7. **Managing Application Lifecycle** : Start, Monitor and Stop the Spark Application.
8. What is difference between Spark Session and Spark Context?

* In Apache Spark, **SparkContext** and **SparkSession** are two important concepts that play key roles in interacting with the Spark cluster.

**Overview**:

 **SparkContext**: It was the main entry point in Spark applications before Spark 2.0. It allows you to connect to the Spark cluster and perform operations on RDDs (Resilient Distributed Datasets).

 **SparkSession**: Introduced in Spark 2.0, it is now the **new entry point** for Spark applications. It encapsulates all the older APIs, including **SparkContext**, and provides additional functionalities like working with DataFrames and Datasets. It also simplifies interaction with different components like SQL, Streaming, and Hive.

|  |  |  |
| --- | --- | --- |
| **Feature** | **SparkContext** | **SparkSession** |
| Introduced In | Spark 1.x | Spark 2.0 |
| Primary Use | Entry point for RDD-based operations | Unified entry point for all Spark APIs (RDD, DataFrame, Dataset, SQL, Streaming) |
| Data Abstraction | RDD (low-level API) | DataFrame, Dataset, RDD (high-level APIs) |
| SQL and Hive | Requires SQLContext or HiveContext | Built-in support for SQL and Hive |
| Ease of Use | Requires managing low-level operations | Easier with high-level APIs like DataFrame and Dataset |
| Performance Optimizations | Less optimized (manual tuning needed) | Built-in optimizations (Catalyst Optimizer, Tungsten) |
| Entry Point Type | Legacy (requires multiple contexts) | Unified (single entry point for all APIs) |

1. What is the difference between GroupByKey and ReduceByKey?

* **GroupByKey**
* **What is does** : Group all values by their key
* **How it works** : it collects all the values associated with a particular key from all partitions and brings them to single place(Which can be inefficient for larger datasets)
* **Efficiency** : GroupByKey is less efficient because it transfer all values across the network before doing computation.
* **ReduceByKey**
  + **What it does :** combines the values for each key using as aggregation function
  + **How it works :** it reduces the data in-place before shuffling, meaning the aggregation happen at each partition and only the results are shuffled across the network.
  + **Efficiency :** reduceByKey is more efficient because minimize the amount of data shuffled between nodes by reducing the data before sending it across the network.

|  |  |  |
| --- | --- | --- |
| **Aspects** | **GroupByKey** | **ReduceByKey** |
| Operation | Groups values by Key | Aggregates(Reduces) values by key |
| Shuffling | Shuffles all values across the network | Shuffles only the reduced values(more efficient) |
| Memory Usage | Higher | Lower |
| Performance | Slower for large datasets | Faster for lage datasets |
| Use case | When you need all the values for each key | When you need to aggregate values by key(like sum,avg,etc) |

1. **Difference between cache() and persist()?**

* In Apache Spark, both persist() and cache() are used to store an RDD or DataFrame in memory or other storage levels to avoid recomputation in future actions.

|  |  |  |
| --- | --- | --- |
| **Aspect** | **cache()** | **persist()** |
| **Default Storage** | Stores the data **in-memory only** | Allows you to specify **different storage levels** (e.g., memory, disk, both) |
| **Storage Levels** | Uses MEMORY\_ONLY storage level | Can use various levels, such as MEMORY\_AND\_DISK, DISK\_ONLY, etc. |
| **Flexibility** | Less flexible (only in-memory) | More flexible (choose memory, disk, or both) |
| **Use Case** | Use when you only need to cache in memory | Use when you want more control over storage (like storing on disk if memory is insufficient) |

**What is cache()?**

* cache() is a shorthand for persisting the data using the default storage level of memory only (MEMORY\_ONLY). This means Spark tries to keep the entire dataset in memory (RAM).
* When to use: Use cache() when you are sure that the dataset can fit into memory, and you don’t want to recompute the RDD or DataFrame in subsequent actions.

**What is persist()?**

* persist() gives you **more control** over how the data is stored. You can choose different **storage levels**, such as:
  + **MEMORY\_ONLY**: Store data only in memory.
  + **MEMORY\_AND\_DISK**: Store data in memory, but if memory is full, spill it to disk.
  + **DISK\_ONLY**: Store data on disk only.
  + **MEMORY\_ONLY\_SER**: Store data in memory in a serialized (compressed) format.
  + **MEMORY\_AND\_DISK\_SER**: Store data in memory (compressed), and spill to disk if memory is full.
* **When to use**: Use persist() when you want more control over the storage strategy, especially for large datasets that may not fit entirely in memory.

|  |  |
| --- | --- |
| **Storage Level** | **Description** |
| **MEMORY\_ONLY** | Stores RDD in memory only. If it doesn’t fit, recomputes partitions as needed. |
| **MEMORY\_AND\_DISK** | Stores RDD in memory, and spills excess data to disk if memory is insufficient. |
| **DISK\_ONLY** | Stores RDD only on disk. |
| **MEMORY\_ONLY\_SER** | Stores RDD in memory, but in a serialized (compressed) format to save space. |
| **MEMORY\_AND\_DISK\_SER** | Stores RDD in memory (serialized), and spills to disk if memory is insufficient. |
| **OFF\_HEAP** (experimental) | Stores RDD in off-heap memory (outside the JVM heap). Requires enabling off-heap. |

1. Difference between RDD, DataFrame and DataSet?

|  |  |  |  |
| --- | --- | --- | --- |
| **Aspect** | **RDD** | **DataFrame** | **Dataset** |
| **API Level** | Low-level (more control, more code) | High-level (easy to use, less code) | High-level + Strongly Typed |
| **Type Safety** | No type safety | No type safety | Compile-time type safety (only in Scala/Java) |
| **Data Structure** | Distributed collection of objects | Distributed table with rows and columns | Distributed table with rows and columns + typing |
| **Optimization** | No automatic optimization | Optimized by Catalyst optimizer | Optimized by Catalyst optimizer |
| **Data Type** | Can handle unstructured and structured data | Best for structured data (tabular format) | Best for structured and semi-structured data |
| **Use Case** | When you need low-level transformations or raw data | Best for structured data processing and analytics | Best when type safety is needed (in Scala/Java) |

1. What are accumulator and Broadcast variable in pyspark?

**1. Accumulator:**

**What is it?**

* An **Accumulator** is a **shared, write-only variable** that multiple worker nodes can update, but the driver node can **read**.
* It's typically used for **aggregating information** (e.g., counting, summing) across multiple nodes in a **distributed fashion**.
* Think of an accumulator as a **global counter** that all worker nodes can increment, but only the driver node can access the final value.

**Use Case:**

* Counting errors or bad records during processing.
* Summing values across nodes.

A screenshot of a computer program

Description automatically generated

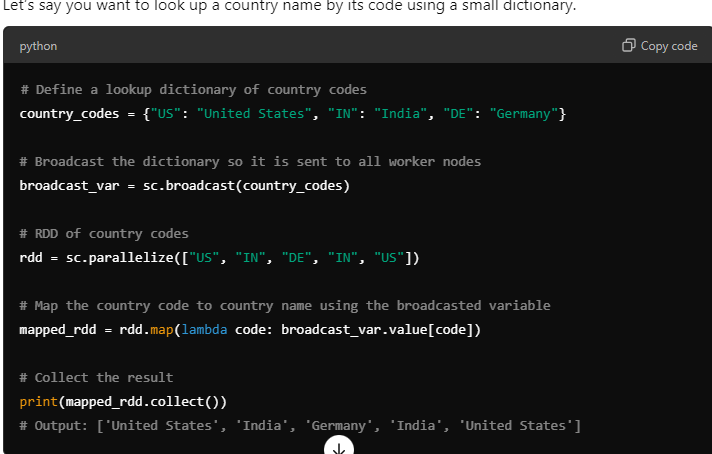
**2. Broadcast Variable:**

**What is it?**

* A **Broadcast variable** allows you to **distribute large read-only data** (like lookup tables or configuration settings) efficiently to all worker nodes.
* Spark sends the **broadcasted data** to all executors (workers) only once, avoiding the need to send large data repeatedly with every task.

**Use Case:**

* Sharing a **large lookup table** across worker nodes for faster lookups.
* Distributing common constants or configuration settings to nodes.



|  |  |  |
| --- | --- | --- |
| **Aspect** | **Accumulator** | **Broadcast Variable** |
| **Purpose** | Used for **aggregating values** across nodes | Used for **distributing large read-only data** across nodes |
| **Type of Access** | **Write-only** for workers, **read-only** for the driver | **Read-only** for all nodes (driver and workers) |
| **Typical Use Case** | Counting, summing, aggregating statistics across nodes | Sharing large lookup tables, constants, or configuration |
| **Data Flow** | Workers update, driver reads final result | Driver broadcasts, workers read |
| **Example** | Counting errors, summing values | Distributing large lookup tables |

1. What is OOM issue ? How to deal with it?

* An **Out of Memory (OOM)** issue occurs when a system or application **exceeds its available memory limit**. In the context of Apache Spark (or any other application), an OOM happens when Spark attempts to load more data or perform more operations than can fit in the available memory of the **driver** or **executors**.
* This issue can cause **task failures, job crashes**, or even the **entire Spark application to fail**, leading to performance problems and requiring retries or adjustments.
* **Why Does OOM Happen in Spark?**
* **Large Datasets**: If you're working with large datasets that can't fit into memory all at once, Spark may fail to handle the data efficiently, leading to OOM errors.
* **Shuffling**: During operations like groupByKey, reduceByKey, or join, Spark shuffles data between nodes. If too much data is shuffled and can’t fit into memory, it may result in an OOM error.
* **Inefficient Data Structures**: Using inefficient or oversized data structures can lead to higher memory consumption.
* **Insufficient Resources**: Not allocating enough memory or resources to executors can lead to OOM issues.
* **How to Deal with OOM Issues in Spark?**

1. Increase Executor/Driver Memory:

* The simplest way to handle an OOM issue is to increase the amount of memory allocated to the Spark executors and driver.
* **How:**
* You can increase memory by tuning the Spark configuration settings like **spark.executor.memory** and **spark.driver.memory.**

1. Use persist() or cache() Efficiently:

* **Persisting** or **caching** RDDs or DataFrames allows Spark to keep them in memory for future operations. However, it can lead to OOM errors if too much data is cached.

**To avoid this:**

* Cache only when necessary.
* Use **disk-based persistence** (MEMORY\_AND\_DISK), which will spill data to disk if there isn’t enough memory.
* Rdd.persist(StorageLevel.MEMORY\_AND\_DISK)

1. **Optimize Shuffles**:

 Shuffles (when Spark moves data between nodes) can cause OOM issues due to the amount of data being transferred.

 **How to mitigate**:

* Use **reduceByKey** instead of **groupByKey**. reduceByKey is more memory-efficient because it reduces data locally before shuffling.
* Avoid wide transformations that cause heavy shuffling.
* Tune shuffle parameters like **spark.sql.shuffle.partitions** (for DataFrames) or **spark.default.parallelism** (for RDDs) to ensure the data is divided into an appropriate number of partitions, reducing the memory burden on each node.

1. **Use Broadcast Variables**:

* For **large read-only datasets** that are used across many tasks (e.g., lookup tables), use **broadcast variables**. This allows Spark to send a **single copy** of the data to each executor rather than having multiple copies.

**Solutions**: Summary

1. Increase memory for executors/drivers.
2. Optimize memory usage (cache efficiently, use broadcast variables, etc.).
3. Avoid excessive shuffles (reduceByKey vs groupByKey).
4. Use optimized file formats (Parquet, ORC).
5. Monitor jobs via Spark’s Web UI.
6. What is Checkpointing? And how it is useful?

* **Checkpointing** in PySpark is a process that saves the **intermediate state** of an RDD or DataFrame to **reliable storage** (e.g., HDFS or a local disk), so that Spark can recover from failures and **avoid re-computation** of complex lineages.
* In distributed computing, operations can fail or nodes can crash, and Spark recomputes transformations from the start to ensure fault tolerance. However, with complex and long chains of transformations (lineages), recomputing everything from scratch can be **time-consuming** and **inefficient**. Checkpointing solves this by cutting the lineage and saving the data to **persistent storage**.

**How Does Checkpointing Work?**

When you **checkpoint** an RDD or DataFrame, Spark:

1. **Stops tracking the lineage** of the transformations leading to that point.
2. **Writes** the RDD or DataFrame to reliable storage, such as **HDFS**, **S3**, or local disk.
3. **Clears the lineage** history, ensuring that future computations are based on the checkpointed data and not on the original transformation chain.

This allows Spark to resume operations from the checkpoint without having to recompute all the previous transformations.

**Types of Checkpointing:**

1. **RDD-Level Checkpointing**:
   * This is used to persist RDDs to reliable storage to truncate long lineages, making them fault-tolerant.
   * Ideal when you have long and complex computations, and you want to **truncate** the lineage to improve fault tolerance or reduce re-computation time.
2. **Streaming Checkpointing**:
   * In **Spark Streaming**, checkpointing is used to maintain the **state** of the stream (like windowed operations or stateful transformations) and recover from failures.
   * Here, both the **metadata** (like offsets) and the **data** are saved to reliable storage.

To set checkpointing **: sc.setCheckpointDir("/path/to/checkpoint/dir")**

**A screen shot of a computer program

Description automatically generated**

**Difference Between Checkpointing and Caching**

* **Checkpoint**:
  + Saves data to **reliable storage** (HDFS, S3, etc.).
  + **Truncates the lineage** (no more recomputation of earlier transformations).
  + Use it for **fault tolerance** and when you need to **cut long lineages**.
* **Cache/Persist**:
  + **Keeps data in memory** (or optionally spills to disk).
  + Retains the **lineage** (can still recompute from the source if necessary).
  + Use it to **speed up** repeated computations.

**Example:**

Imagine an RDD that is built by reading a large file, applying 10 transformations (map, filter, etc.). If a failure occurs during processing, Spark will try to **re-run all 10 transformations** from the beginning.

Now, if you **checkpoint** the RDD after the 5th transformation, Spark will **save the result to disk**. If a failure happens after this point, instead of recomputing everything from scratch, Spark **starts from the checkpoint**, only redoing the transformations after the checkpoint, **saving time** and ensuring recovery is faster and more reliable.

In short, **truncating the lineage** with checkpointing means Spark doesn't need to recompute the entire history, making it easier to recover from failures efficiently.

1. What is role of Catalyst Optimizer in spark?

* <https://www.linkedin.com/pulse/turbocharging-spark-queries-how-catalyst-optimizer-tripathi-pmp-/>

1. AVRO and ORC file , which one would you prefer?

**1. AVRO (Apache Avro):**

* **Serialization format**: It is used for **row-based** data storage, meaning data is stored in the order it is written.
* **Good for streaming** or **write-heavy workloads**.
* **Schema evolution**: Avro handles schema changes very well, making it suitable for data where the structure might change over time.
* **Example**: If you are constantly writing logs or messages to a file, **Avro** would be better since it stores each row efficiently.

**2. ORC (Optimized Row Columnar):**

* **Columnar format**: Data is stored **column by column**, making it highly efficient for reading and compressing large datasets.
* **Good for analytics**: Ideal for **read-heavy** operations like querying large data sets, aggregations, or scans.
* **Compression**: ORC provides excellent compression, reducing storage costs.
* **Example**: If you are querying large datasets in a **data warehouse** for analytical purposes, **ORC** would be preferred because of its fast reading performance.
* "I prefer **ORC** for large analytical workloads where we need to read and process big datasets efficiently, like in data warehouses. It’s faster because of its columnar storage and great compression.
* On the other hand, I’d use **Avro** for **streaming** or when handling a lot of **small writes** or **schema evolution** scenarios, such as log storage or data pipelines with evolving data structures."

1. **What is difference between DAG and Lineage Graph?**

* In Spark, both Lineage Graph and DAG(Directed Acyclic Graph) are related to how the execution of tasks is tracked and optimized.
  1. **Lineage Graph :**
  + The Lineage Graph tracks the sequence of transformation applied to RDDs (Resilient Distributed Datasets) to create new RDDs. It shows how RDD are derived from the other RDDs.
  + It is a logical plan that helps in fault recovery. If part of the data is lost, Spark can recompute that data by retracing the transformation(Using the lineage)

Example:

Rdd1 = sc.textFile(“file.txt”)

Rdd2 = rdd1.map(lambda x:x.split(“ “))

Rdd3 = rdd2.filter(lambda x : len(x) > 2 )

In this example, rdd3 can be recomputed by retracing the transformation from rdd1 -> rdd2 -> rdd3 . This sequence of transformation is the **Lineage Graph.**

* 1. **DAG(Directed Acyclic Graph):**
* The DAG represents the **physical execution plan** in spark. It breaks down the logical plan(Lineage graph) into stages of tasks to be executed in parallel.
* The DAG contains stages and tasks based on shuffle boundaries and represents how spark will actually execute the job on the cluster.

Example:

**Rdd3.count()**

Spark Creates a DAG of Stages:

1. Stage 1 : Read the file(rdd1).
2. Stage 2 : Apply transformation (rdd2 -> rdd3)
3. Stage 3 : Perform the action (count)

**Key Difference :**

* **Lineage Graph :** Logical sequence of transformation(for fault recovery)
* **DAG :** Physical execution plan with stages and task(for execution)

1. **What is difference between Spark-submit and Spark shell ?**

* Summary :
  + **Spark Shell** is for interactive use, like exploring datasets, testing code, or learning Spark features.
  + **spark-submit** is for running pre-built applications in production, usually on a cluster.
* A good example to mention is:
  + Use Spark Shell when you want to analyze a small dataset interactively on your local machine.
  + Use spark-submit when you have a large ETL job to run on a cluster, processing terabytes of data in a distributed fashion.

1. **What is Data Skewness? How to deal with it ?**

* **Data Skewness** in Spark (or any distributed computing framework) refers to the **uneven distribution of data** across different partitions or nodes.
* When data is not evenly distributed, some partitions have much more data than others, causing certain tasks to take much longer to complete. This results in inefficient resource utilization, where some tasks finish quickly while others are left processing large amounts of data, causing **performance bottlenecks**.

**Example of Data Skewness:**

* Imagine you have a dataset of **orders**, and you want to group the data by the **country** where the orders were placed. If most of the orders come from just one country, like the USA, then the data will be **skewed** toward that country.

A screenshot of a computer screen

Description automatically generated

In this case, the partition containing data for the USA will have significantly more records to process compared to the partitions for India and the UK. This leads to **data skew**.

**How Data Skew Impacts Performance:**

* Some tasks (processing USA data) will take much longer to complete compared to others.
* Nodes with smaller partitions will be **underutilized**, while the nodes with more data are **overloaded**, creating an imbalance.
* The overall **job performance is slower** because Spark has to wait for the overloaded partitions to finish.
* **Dealing with Data Skewness in PySpark**

**1. Salting:**

* One of the most common techniques to deal with skewness is **salting**. This involves adding a random key to the skewed data to artificially distribute it across more partitions.

By salting, we reduce the load on a single partition by spreading the "USA" records across multiple partitions. This distributes the workload more evenly.

**2. Increasing the Number of Partitions:**

You can increase the number of partitions to reduce the amount of data each partition holds.

df\_repartitioned = df.repartition(10, 'Country')

**3. Avoiding Expensive Operations on Skewed Data:**

* Certain operations like **groupByKey()** or **reduceByKey ()** can cause problems with skewed data because they require shuffling data to group by a key. Instead, using more efficient transformations like aggregateByKey() can reduce the impact of skewness.

**4. Skewed Join Optimization:**

* In cases where you're joining skewed datasets, you can use **broadcast joins** or **skewed join optimizations** to handle the skewed key. Broadcasting a smaller dataset to all nodes can avoid shuffling large amounts of skewed data.

**Summary :**

* **Data Skewness** happens when data is unevenly distributed across partitions, leading to some partitions being overworked and others underutilized.
* It can significantly **slow down** performance in Spark jobs.
* To handle it:
  1. Use **salting** to spread skewed data across more partitions.
  2. **Increase the number of partitions** to better distribute data.
  3. Avoid expensive operations like groupByKey() by using alternatives like reduceByKey().
  4. For joins, use **broadcast joins** or optimize for skewed joins.

1. **Difference between Order by, Sort by and Cluster By ?**

**1. ORDER BY:**

* **Purpose**: Fully **sorts the data** across all partitions.
* **Global Sorting**: Ensures a complete sort of the data across all partitions. After ORDER BY, the data is sorted across the entire dataset, not just within partitions.
* **Performance**: It is the most expensive of the three because it triggers a **shuffle** and ensures that all data is fully sorted, meaning it takes more time and resources.
* **Usage**: Use ORDER BY when you need a global sort on your data.

Ex: df.orderBy(“age”).show()

This sorts the entire DataFrame by the age column globally, ensuring the result is sorted from the smallest to largest value across all partitions.

**2. SORT BY:**

* **Purpose**: Sorts the data **within each partition**.
* **Local Sorting**: Unlike ORDER BY, SORT BY sorts data only within each partition, so the results might not be globally sorted.
* **Performance**: It's less expensive than ORDER BY because it doesn’t ensure global ordering but only sorts within partitions.
* **Usage**: Use SORT BY when you just need sorted data within each partition but don’t need the data fully sorted globally.

Ex: df.sort(“age”).show()

**3. CLUSTER BY:**

* **Purpose**: Distributes data into partitions based on the values of one or more columns, and then **sorts each partition**. It is similar to **distribute by** + **sort by**.
* **Hash Partitioning**: Spark performs **hash partitioning** based on the specified column(s). All rows with the same key will end up in the same partition.
* **Performance**: It’s efficient for **partitioning** the data and ensuring rows with the same value are in the same partition (useful for joins, bucketing, etc.). It also **sorts** data within each partition but doesn’t guarantee global ordering.
* **Usage**: Use CLUSTER BY when you want to **partition data** by a specific column and **sort** within partitions. It’s useful when you need to distribute data evenly across partitions based on a key (like preparing for a join).

Ex ; df.clusterBy("age", numPartitions=5).show()

**Interview Answer:**

* **ORDER BY** globally sorts data across all partitions. It ensures that your data is in complete order from start to end but is the most expensive in terms of performance due to shuffling.
* **SORT BY** sorts data only **within** partitions, which is faster but doesn’t guarantee that the data is globally ordered.
* **CLUSTER BY** partitions the data based on a column and then sorts the data within each partition. It’s ideal for distributing data across partitions and ensures sorted data within partitions.

A good way to differentiate is that **ORDER BY** is for global sorting, **SORT BY** is for local sorting, and **CLUSTER BY** is for partitioning and sorting combined.

1. **Calculate executor? If you have 10 nodes , 16 cores each, 64 GB RAM ?**

To calculate the number of **executors** in a Spark cluster, follow these steps based on the hardware provided:

* **Cluster Details**:
  + **Nodes** = 10
  + **Cores per Node** = 16
  + **RAM per Node** = 64 GB

**Simple Calculation:**

1. **Leave 1 core per node** for **OS and system processes**:  
   Available cores per node = **16 - 1 = 15 cores**
2. **Executor Cores**:  
   Typically, you allocate **5 cores per executor** for efficient parallelism without causing overhead.
   * **Executors per Node** = 15 cores 5 cores per executor=3 executors per node\frac{15 \text{ cores}}{5 \text{ cores per executor}} = 3 \text{ executors per node}5 cores per executor15 cores​=3 executors per node
3. **Memory per Executor**:
   * Reserve **10% of RAM** for system processes and overhead.  
     Available memory per node = 64 GB×90%=57.6 GB64 \text{ GB} \times 90\% = 57.6 \text{ GB}64 GB×90%=57.6 GB
   * **Memory per executor** = 57.6 GB3 executors=19.2 GB per executor\frac{57.6 \text{ GB}}{3 \text{ executors}} = 19.2 \text{ GB per executor}3 executors57.6 GB​=19.2 GB per executor
4. **Total Executors**:
   * Total nodes = 10
   * Executors per node = 3
   * **Total Executors = 10 \times 3 = 30 executors**

**Summary:**

* **Executors per node**: 3
* **Memory per executor**: 19.2 GB
* **Total Executors**: 30 executors in the cluster

1. **Explain Dynamic partition Pruning?**

**Predicate Pushdown** reduces data read by applying filters directly at the data source.

**Partition Pruning** limits the number of partitions read, improving efficiency when data is partitioned.

*Dynamic Partition Pruning = Predicate Push Down + Broadcast Hash Join*

* 1. [*https://medium.com/geekculture/dynamic-partition-pruning-baf3270694b4*](https://medium.com/geekculture/dynamic-partition-pruning-baf3270694b4)
  2. [*https://medium.com/@prabhakaran.electric/spark-3-0-feature-dynamic-partition-pruning-dpp-to-avoid-scanning-irrelevant-data-1a7bbd006a89*](https://medium.com/@prabhakaran.electric/spark-3-0-feature-dynamic-partition-pruning-dpp-to-avoid-scanning-irrelevant-data-1a7bbd006a89)

1. **Explain AQE (Adaptive Query Execution) in spark?**

[**https://www.linkedin.com/posts/prashant-kumar-pandey\_sparkpysparkday14-activity-7227871283982524416-NmiT?utm\_source=share&utm\_medium=member\_android**](https://www.linkedin.com/posts/prashant-kumar-pandey_sparkpysparkday14-activity-7227871283982524416-NmiT?utm_source=share&utm_medium=member_android)

1. **How does data serialization work in pyspark?**

* Serialization in PySpark refers to the process of **converting data objects** (like Python objects, RDDs, or DataFrames) into a **byte stream** that can be sent across a network or stored efficiently. It’s like converting a complex object into a simpler format so it can be moved easily between Spark's **driver** and **worker nodes**.
* PySpark runs in a distributed environment, so it needs to send data and computation tasks across nodes in the cluster. Serialization ensures that **Python objects** (functions, classes, variables) can be **packaged** and sent over to **executor nodes** where they are deserialized and executed.

**Summary:**

* **Serialization** is the process of converting objects into a format (byte stream) that can be transmitted across a network.
* In PySpark, serialization allows data and computation (like functions) to be **sent from the driver to worker nodes**.
* PySpark supports **Java serialization** (default) and **Kryo serialization** (faster).
* Use **Kryo serialization** for better performance, especially with custom objects or large datasets.

1. **Can you discuss the significance of choosing the right compression codec for your PySpark Application?**

* Choosing the **right compression codec** for your PySpark application can have a **significant impact** on performance, storage efficiency, and resource utilization.
* **Why is Compression Important in PySpark?**

1. Reduced disk I/O :
2. Speedup data Transfer :
3. Saves storage costs : compress data require less storage space(i.e S3,HDFS)
4. Optimize memory usage : reducing the risk of OOM.

* **Common Compression Codecs in PySpark :**

1. **GZIP :**

* Compression: High
* Decompression Speed: Slow
* File Splittability: Not splittable (files are read entirely in one go)

1. **Snappy :**

* **Compression**: Moderate
* **Decompression Speed**: Fast
* **File Splittability**: Splittable

**How to Choose the Right Compression Codec?**

The right compression codec depends on your specific use case. Here are some factors to consider:

1. **Query Performance vs. Compression Ratio**:
   * If **fast read performance** is more important than saving space, use **Snappy** or **LZO**.
   * If **saving space** is more critical (e.g., for archival or long-term storage), use **Gzip** or **Bzip2**.
2. **Data Size and Parallel Processing**:
   * For **large datasets** that need to be processed in parallel across a cluster, choose **splittable formats** like **Snappy** or **LZO**. Avoid **Gzip** as it’s not splittable, which limits parallelism.
3. **Storage Costs**:
   * If you are storing data in cloud environments where you are charged for storage, **compression can lead to significant cost savings**. Formats like **Parquet** or **ORC** with Snappy compression are a good choice for such scenarios because they balance **speed** and **compression**.
4. **Network Transfer**:
   * If your Spark job involves a lot of **network transfer** (e.g., between nodes or reading from cloud storage), **compressed data** can reduce the amount of data transferred, speeding up the job and reducing network costs.
5. **Are there any specific strategic or functions you prefer for handling missing data in spark?**

* **Imputation** : Filling missing values with the mean or median of the column using fillna().
* **Imputation** : For categorical columns, replace nulls with the most common values or create a new category.
* **Flagging** : Add a new binary column to indicate whether the data was originally missing, helping to preserve information.

1. **How do you handle missing or null values in PySpark?**
2. Use Dropna() to remove rows with null values.
3. Use fillna() to fill null values with specified constant.
4. Apply conditional logic to replace nulls based on other column values, using when() and otherwise() functions.