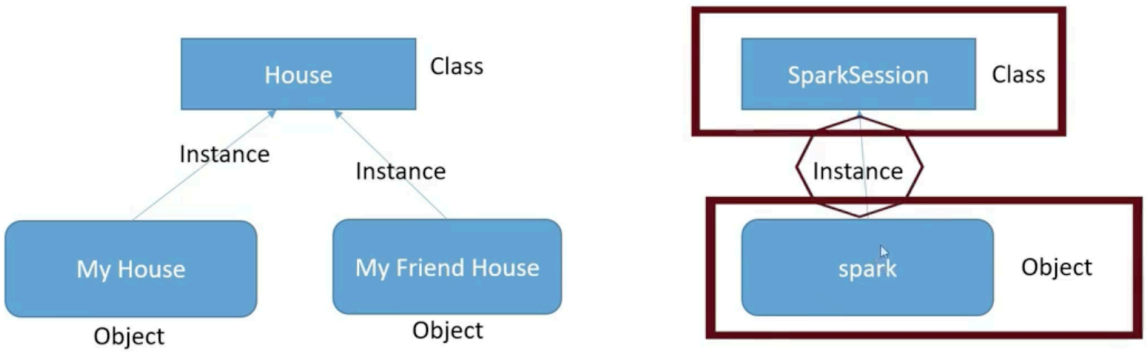
Spark:

Relationship between parent class and sub class called instance



Spark session :

```
from pyspark.sql import SparkSession

spark = SparkSession \
        .builder \
        .master('yarn') \
        .appName("Python Spark SQL basic example") \
        .getOrCreate()
```

Master can be yarn, mesos, Kubernetes or local[x] , x > 0

Spark submit:

We can pass the config using this

Spark-submit is a utility to run a pyspark application job by specifying options and configurations.

```
spark-submit \
 --master <master-url> \
 --deploy-mode <deploy-mode> \
 --conf <key<=<value> \
 --driver-memory <value>g \
 --executor-memory <value>g \
 --executor-cores <number of cores>  \
 --jars  <comma separated dependencies> \
 --packages <package name> \
 --py-files  \
 <application> <application args>
```

Client mode is more interactive and allows the user to monitor the application closely since the driver runs in the client's environment.

Cluster mode is more suitable for production deployments as the driver runs within the cluster, and the user does not need to manage it directly.

In Apache Spark, the default number of shuffle partitions is set to 200. This configuration is controlled by the parameter spark.sql.shuffle.partitions. Shuffling is a costly operation in distributed computing, involving data transfer between different nodes, so it's important to configure the number of partitions appropriately based on the size and characteristics of your data.

Setting spark.sql.shuffle.partitions to a higher value can be beneficial for improving performance in scenarios where you have large datasets or complex queries that involve shuffling data across the cluster. However, setting it too high can lead to excessive memory consumption and longer task scheduling times.

```
import org.apache.spark.sql.SparkSession

val spark = SparkSession.builder()
  .appName("YourAppName")
  .config("spark.sql.shuffle.partitions", "300")
  .getOrCreate()
```

By default driver memory is 1024 mb we can modify this as well

Apache Spark, a JAR (Java ARchive) file is a package file format typically used to aggregate many Java class files, associated metadata, and resources (such as text, images, etc.) into one file for distribution.
In the context of Spark applications, a JAR file is often used to package and distribute the code for your Spark job. When you write Spark applications using Java or Scala, you compile your code into bytecode, which is then packaged into a JAR file.
Here's how you typically use a JAR file in Spark:
Write your Spark application code: This could be in Java or Scala. You write your Spark code to perform various data processing tasks.
Compile your code: Once you've written your Spark application code, you compile it into bytecode using a Java or Scala compiler. For Scala applications, this typically involves compiling to a JAR file directly.
Package your code into a JAR file: After compiling your code, you package it into a JAR file along with any dependencies your application might have. This JAR file will contain your application code and any third-party libraries your application depends on.
—Third party libraries mean not available in spark—

Submit your Spark application using spark-submit: You use the spark-submit command-line tool to submit your Spark application to a Spark cluster. You specify the path to your JAR file containing your application code along with any necessary configuration options.

```
spark-submit --master <your-cluster-master-url> sum_measurements.py
```

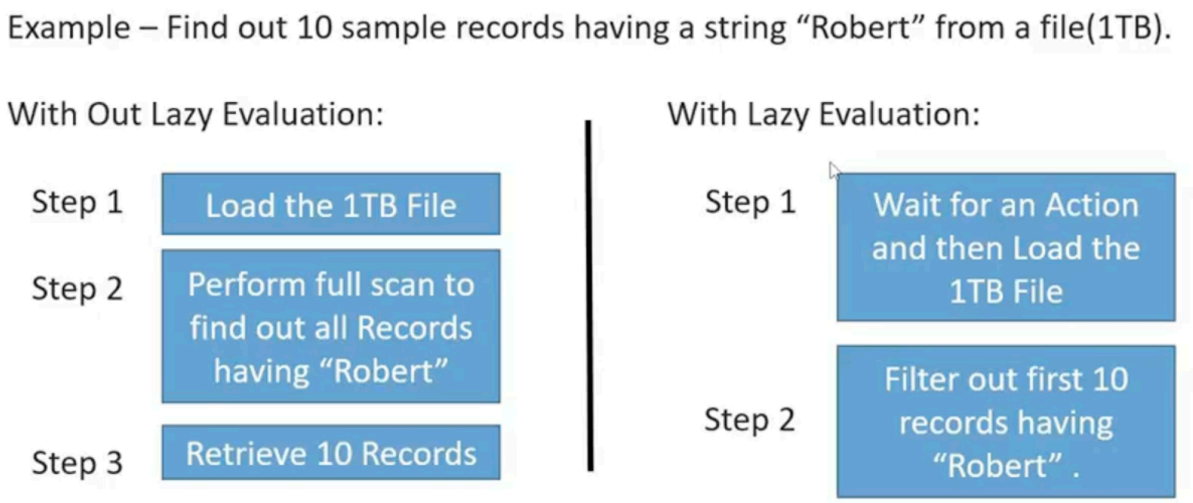
Pyspark do not support dataset but Scala support rdd data frame and dataset all three but
˘PySpark: Limited support (post Spark 2.0) version it started supported.

Lazy evaluation in Apache Spark refers to the mechanism where transformations on distributed datasets (RDDs, DataFrames, or Datasets) are not immediately executed. Instead, Spark builds up

a Directed Acyclic Graph (DAG) of the transformations, and these transformations are only executed when an action is called.

Here's how it works:
Transformation— — — — —>action
Transformations: When you apply transformations like map, filter, groupBy, etc., on an RDD or DataFrame in Spark, Spark does not immediately compute the result. Instead, it keeps track of these transformations and builds up an execution plan represented as a DAG.
Actions: Actions are operations that trigger the execution of the DAG and initiate the actual computation. Examples of actions include count, collect, saveAsTextFile, etc. When an action is called, Spark examines the DAG and optimizes it for execution, then begins the computation.

the Directed Acyclic Graph (DAG) is a fundamental concept that represents the logical execution plan of a Spark job.
The DAG is a directed graph where vertices represent RDDs (Resilient Distributed Datasets) or stages, and edges represent the operations or transformations to be applied on the data.



Problem in rdd:

Performance Overhead: RDDs have higher overhead compared to higher-level APIs like DataFrames and Datasets. Since RDDs do not provide query optimization and are not aware of the structure of the data, they may not perform as efficiently as DataFrame or Dataset operations.
Manual Optimization: Developers need to manually optimize RDD operations by considering partitioning, caching, and serialization. This requires a deep understanding of the Spark execution model and can be error-prone.

We do not much use rdd in place of that we used data set and data frame and spark sql rdd is complex here is example for simple sql how rdd looks .

in PySpark is

Main Problems in RDD ?

Problem#2 (Pretty verbose to work with)

```
emp.map(lambda x: (x.deptid, [x.age, 1])) \
    .reduceByKey(lambda x,y: [x[0] + y[0], x[1] + y[1]]) \
    . Map(lambda x: [x[0], x[1][0] / x[1][1]]) \
    .collect()

SELECT deptid, AVG(age) FROM emp GROUP BY deptid
```

straightforward.
You can create an RDD from an existing collection (such as a Python list), from a file in HDFS, or by parallelizing an existing collection. Here are a few examples:

Map vs flat map ?

The output RDD has a one-to-one correspondence with the input RDD, meaning the output RDD has the same number of elements as the input RDD.

```
rdd = sc.parallelize([1, 2, 3])
mapped_rdd = rdd.map(lambda x: [x, x*2])
# mapped_rdd: [[1, 2], [2, 4], [3, 6]]
```

The flatMap() transformation applies a function to each element of the RDD ,the output RDD. It "flattens" the results, combining all the resulting collections into a single collection.

flatMap() is useful when you need to transform each input element into multiple output elements and then flatten the result into a single RDD.

flatMap flattens the resulting lists into a single RDD. Finally, both RDDs are collected and printed.

```
rdd = sc.parallelize([1, 2, 3])
flat_mapped_rdd = rdd.flatMap(lambda x: [x, x*2])
# flat_mapped_rdd: [1, 2, 2, 4, 3, 6]
```

Data skewness handle by method salting .

Product ID	Sales Amount
1234	1000
5678	500

| 1234 | 2000 |
| 9012 | 300 |

Product ID	Product Name
1234	Product A
5678	Product B
9012	Product C

Original Product ID	Salted Product IDs
1234	1234_1, 1234_2, ...

After distributing data across partitions based on the salted product IDs, processing, and aggregating results,

Product ID	Product Name	Total Sales Amount
1234	Product A	3000
5678	Product B	500
9012	Product C	300

So after salting we did the aggregation of that amount and 1000+2000 and tagged with one id
In this the sharing data across node will equally distributed.

Map function uses ?

Parallel Processing: RDDs partition data across multiple nodes in a cluster, enabling parallel processing of data.
The map() function used to applies a transformation function to each element of the RDD in parallel, distributing the workload across the cluster's nodes.

Transformation: The map() function transforms each element of an RDD into another element based on a function provided by the user. This transformation can be anything from simple data manipulation (e.g., adding a prefix to each element) to more complex operations (e.g., parsing and processing log files).

filter()

filter() function is used to create a new RDD containing only the elements that satisfy a specified condition.

mapvalue()

This function is useful when you want to transform the values of an RDD without modifying the keys.

[('a', 2), ('b', 4), ('c', 6)]

Map vs map value

Use map() when you want to transform each element of an RDD independently, regardless of whether the RDD contains key-value pairs.

Use mapValues() when you specifically want to transform the values of key-value pairs in an RDD while preserving the keys.

cogroup() :

Cogroup is a transformation operation available in Apache Spark's Resilient Distributed Datasets (RDDs). It's used to group the values of two RDDs with the same key into an iterable of tuples. When you have two RDDs containing key-value pairs and you want to perform operations like joining them or comparing values with the same key, cogroup() can be particularly useful.
Here's a high-level explanation of how cogroup() works:

Input RDDs: You start with two RDDs, each containing key-value pairs. Let's call them RDD1 and RDD2.
Grouping by Key: cogroup() operates on these two RDDs and groups the values associated with each key into an iterable. If a key is present in one RDD but not the other, it still includes an empty iterable for that key.
Output: The output RDD contains key-value pairs where the key is the common key from both RDDs, and the value is a tuple of two iterables. The first iterable contains the values from RDD1 associated with that key, and the second iterable contains the values from RDD2 associated with the same key.

```
from pyspark import SparkContext

sc = SparkContext("local", "cogroupExample")

# Sample data for RDD1
rdd1 = sc.parallelize([(1, 'A'), (2, 'B'), (3, 'C')])

# Sample data for RDD2
rdd2 = sc.parallelize([(2, 'X'), (3, 'Y'), (4, 'Z')])

# Apply cogroup to group values by key
grouped_rdd = rdd1.cogroup(rdd2)

# Print the result
print(grouped_rdd.collect())

sc.stop()
```

```
[(1, (<pyspark.resultiterable.ResultIterable object at 0x7f91c94a2ac0>, [])),
 (2, (<pyspark.resultiterable.ResultIterable object at 0x7f91c94a2a30>,
<pyspark.resultiterable.ResultIterable object at 0x7f91c94a29d0>)),
 (3, (<pyspark.resultiterable.ResultIterable object at 0x7f91c94a2700>,
<pyspark.resultiterable.ResultIterable object at 0x7f91c94a2b20>)),
 (4, ([], <pyspark.resultiterable.ResultIterable object at 0x7f91c94a2d30>))]
```

There is two types of aggregation key aggregation and transformagrregation

- Total aggregations – reduce, count (Actions)
- By Key aggregations – reduceByKey, aggregrateByKey, groupByKey, countByKey (Transformations)

We use the parallelize() function to create an RDD rdd from the sample data.

Shuffle make the data aggregation slow and it expensive as when it is suffering the data it created multiple small data file and then sort the dats so we need to avoid the shuffle using combiner we can reduce this .

SparkContext and SparkSession are both important components in Apache Spark, but they serve different purposes.

SparkContext: It was the main entry point for Spark applications before Spark 2.0. It represented the connection to a Spark cluster and allowed you to create RDDs (Resilient Distributed Datasets), which were the primary data abstraction in earlier versions of Spark.
SparkSession: Introduced in Spark 2.0, it provides a unified entry point for interacting with Spark functionality and replaces SparkContext, SQLContext, and HiveContext. It's used to create DataFrames, which are more powerful than RDDs because they provide higher-level API methods for manipulating structured data.

Reducebykey is more efficient then group by key and aggregate by key and memory efficient.
Reason is

Group by key shuffles all the data across the network, which can lead to network overhead and performance degradation, especially with large datasets.

It's less efficient than reduceByKey() because it doesn't perform any local aggregation before the shuffle.

What is mean of local aggregation .?

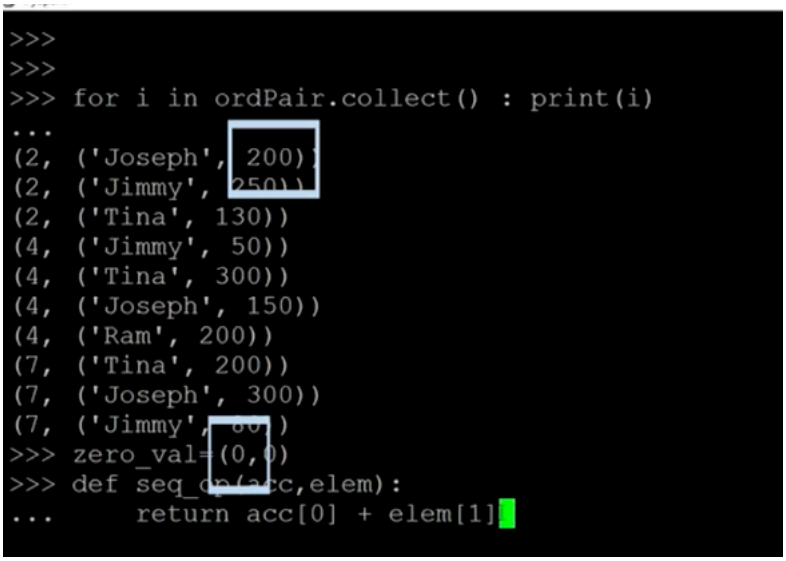
Spark provides transformations like reduceByKey, aggregateByKey, and combineByKey that perform local aggregation within partitions first.

These operations combine values with the same key within each partition, reducing the amount of data that needs to be shuffled across the network for the global aggregation phase.

Reduce by key It's more efficient than groupByKey() because it aggregates the data locally on (operations combine values with the same key)each partition before the shuffle.

Aggregationbykey()

```
aggregateByKey(initialSalesCount, seq_op, comb_op)
```



In this we are zero_val as (0,0)
After that we are creating seq_op in this we are adding up accumulator 0 and element 1 with index.
Def seq_op (acc,elem) :
    Return acc[0]+elem[1],acc[1]+1

acc[1]+1 it will add the key and increase by 1 for adding the next index in incremental manner

orderBy:

Context: orderBy is used in Spark SQL or DataFrame API.
Functionality: It sorts the rows of a DataFrame based on one or more columns.
Input: It takes column names or expressions as input.
It is more optimised as it is decided spark catalyst optimiser to optimise the execution plan .
Output: It returns a new DataFrame with rows sorted according to the specified column(s).

sortByKey:

Context: sortByKey is used in the RDD API.
Functionality: It sorts key-value pairs RDDs (PairRDD) by the key.
Input: It operates on an RDD of key-value pairs, where the key is used for sorting.
Output: It returns a new RDD with key-value pairs sorted by the key.

Execution: Spark executes the sortBy operation on each partition of the RDD independently. It sorts the data within each partition according to the custom comparator function.

In summary, sample is used for probabilistic sampling, taking a fraction of the data, while takeSample is used for deterministic sampling, taking a fixed number of elements. The choice between them depends on your specific use case and requirements

In Apache Spark, sample and takeSample are two methods used with RDDs for sampling data. They serve different purposes and are used in different scenarios.

Sample

Purpose: Used to take a random sample of data from an RDD.
Syntax: sample(withReplacement, fraction, seed=None)
Parameters:

withReplacement: Boolean flag indicating whether to sample with replacement.
fraction: Fraction of the data to sample (between 0 and 1).
seed: Optional seed for the random number generator.
Usage: Typically used for statistical analysis or quick exploration of data.

Takesample

Purpose: Used to take a deterministic sample of data from an RDD.
Syntax: takeSample(withReplacement, num, seed=None)
Parameters:
withReplacement: Boolean flag indicating whether to sample with replacement.
num: Number of elements to sample.
seed: Optional seed for the random number generator.
Usage: Often used for testing or debugging purposes when you need a specific number of samples.

To change data frame to RDD we can use this

```
rdd = df.rdd


Create an RDD
data = [('Alice', 1), ('Bob', 2), ('Cathy', 3)]
rdd = spark.sparkContext.parallelize(data)

Define a schema
schema = StructType([
    StructField("Name", StringType(), True),
    StructField("ID", IntegerType(), True)
])

Convert RDD to DataFrame
df = spark.createDataFrame(rdd, schema)

Show the DataFrame
df.show()
```

Repartion vs coalesce

repartition is used for increasing or decreasing the number of partitions with a full shuffle, while coalesce is primarily used for decreasing the number of partitions without a full shuffle, making it more efficient in many cases. Choose the appropriate method based on your specific use case and performance considerations.

Suppose you have a large dataset that you've filtered, and now you want to save it to HDFS. After filtering, the data might be spread across many small partitions, leading to inefficient I/O operations:

```
val largeDf = spark.read.parquet("path/to/large/parquet/file")
val filteredDf = largeDf.filter($"column" > 1000)
println(filteredDf.rdd.getNumPartitions)  // Check the number of partitions

// Reduce the number of partitions before saving
val optimizedDf = filteredDf.coalesce(10)
optimizedDf.write.parquet("path/to/output")
```

In this example, coalesce is used to reduce the number of partitions of filteredDf from a larger number (possibly hundreds or thousands) to 10, optimizing the write operation to HDFS.

By understanding and appropriately using coalesce and repartition, you can significantly optimize the performance of your Spark applications.

Glom

glom is a transformation operation that allows you to collect the elements of each partition of an RDD into a single list. It's useful when you need to perform operations on the entire partition as a whole, rather than on individual elements within the partition
We used this to also check how the data is distributed to each partition which help to give insight of data skewness.

Create a rdd with 2 partition :

```
# Create an RDD with 2 partitions
data = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
rdd = sc.parallelize(data, 2)
```

cache() and checkpoint() are two different mechanisms in Apache Spark, used for different purposes:

cache() is an operation that allows you to persist the data of a DataFrame, RDD, or Dataset in memory (RAM) or on disk for faster access during subsequent operations.
When you call cache() on a DataFrame or RDD, Spark will store the data partitions in memory (if enough memory is available) or spill them to disk if necessary.

Df.cache()

checkpoint() is an operation that allows you to persist the data of a DataFrame, RDD, or Dataset to a reliable storage (e.g., HDFS, S3) and truncate the lineage graph.
When you call checkpoint() on a DataFrame or RDD, Spark will save the data to the specified checkpoint directory and mark the DataFrame or RDD for truncation of the lineage graph.
Checkpointing is used to cut the lineage graph and avoid long lineage chains, which can lead to StackOverflowErrors or other issues due to excessive memory consumption.

spark.sparkContext.setCheckpointDir("hdfs://path/to/checkpoint/dir")
df.checkpoint()

NARROW VS WIDE TRANSFORMATION

Narrow Transformations:

Narrow transformations are those where each input partition contributes to only one output partition. In other words, the data is transformed within the same partition and does not require shuffling of data across the network. This makes narrow transformations more efficient and faster.

map: Applies a function to each element of the RDD or DataFrame.
filter: Selects elements that satisfy a predicate.
flatMap: Applies a function that returns an iterable to each element, then flattens the results.
mapPartitions: Similar to map, but the function is applied to each partition rather than each element.
union: Combines two RDDs or DataFrames without shuffling.

Wide transformations:

Wide transformations are those where each input partition contributes to multiple output partitions. These transformations require a shuffle phase, where data is redistributed across the network. Wide transformations can be more costly due to the overhead of shuffling data.

reduceByKey: Aggregates data with the same key across partitions.
groupByKey: Groups data with the same key across partitions.
join: Joins two RDDs or DataFrames based on keys.
distinct: Removes duplicate elements, which may require shuffling.
repartition: Changes the number of partitions, causing data to be redistributed.

Performance Considerations:

Narrow Transformations:
Faster due to no data shuffling.
More efficient in terms of resource usage.

Wide Transformations:
Slower due to the overhead of data shuffling across the network.
Higher resource usage, including CPU, memory, and network I/O.
Can cause "shuffle spill" to disk if the data does not fit in memory.

Transformation and action

Transformation

Transformations are operations on an RDD (Resilient Distributed Dataset) or DataFrame that return a new RDD or DataFrame. They are lazy, meaning they don't immediately compute their results. Instead, they build up a logical execution plan (DAG - Directed Acyclic Graph) that will be executed when an action is called. Transformations allow Spark to optimize the execution plan before actually running it.

map: Applies a function to each element of the RDD or DataFrame and returns a new RDD or DataFrame.
filter: Returns a new RDD or DataFrame containing only the elements that satisfy a predicate.
flatMap: Similar to map, but each input item can be mapped to 0 or more output items (hence the "flat").
distinct: Returns a new RDD containing the distinct elements of the original RDD.
union: Returns a new RDD that contains the union of the elements in the source dataset and the argument dataset.
groupByKey: Groups the data based on the key and returns a new RDD.
reduceByKey: Combines values with the same key using a specified associative function and returns a new RDD.
sortBy: Sorts the elements of the RDD based on a specified function.

Action

Actions are operations that trigger the execution of the transformations that are built up in the logical execution plan. They return a value (to the driver program) or write data to an external storage system.

collect: Returns all the elements of the RDD or DataFrame as an array to the driver program.
count: Returns the number of elements in the RDD or DataFrame.
take: Returns the first n elements of the RDD or DataFrame.
reduce: Aggregates the elements of the RDD or DataFrame using a specified function.
saveAsTextFile: Writes the elements of the RDD or DataFrame to a text file.

saveAsHadoopFile: Writes the elements of the RDD or DataFrame to a Hadoop file system.
foreach: Applies a function to each element of the RDD or DataFrame, typically used for side effects such as updating an accumulator or interacting with external systems.

Dataframe vs dataset

DataFrame is a higher-level abstraction designed for structured data processing and SQL-like operations, while Dataset provides type safety and performance optimizations for working with structured data in Scala and Java. The choice between DataFrame and Dataset depends on the specific requirements of the application and the programming language being
DataFrame API is available in different programming languages such as Python, Java, Scala, and R, making it widely accessible.

In Scala and Java, Dataset provides compile-time type checking, enabling early detection of type errors and better code quality.

Dataset API is available in Scala, Java, and Python, but it offers the most advanced features and type safety in Scala and Java.

Compile-time type checking: When you define a Dataset in Scala or Java, you specify the type of the elements it contains. For example, if you have a Dataset of case class objects representing Person records, the compiler ensures that operations performed on this Dataset are type-safe. If you attempt to perform operations that are not type-safe, such as accessing a field that doesn't exist or applying incompatible transformations, the compiler will raise an error at compile time, preventing runtime errors and ensuring early detection of issues.

What is serialization in Spark?

Serialization is the process of converting data structures or objects into a format that can be stored or transmitted.
What is the default serialization format used in Spark?
Java Serialization (Java objects serialized into binary format).

What is the significance of the persist method in Spark?

The persist method allows RDDs to be cached in memory or disk storage for faster access and reusability across multiple actions.

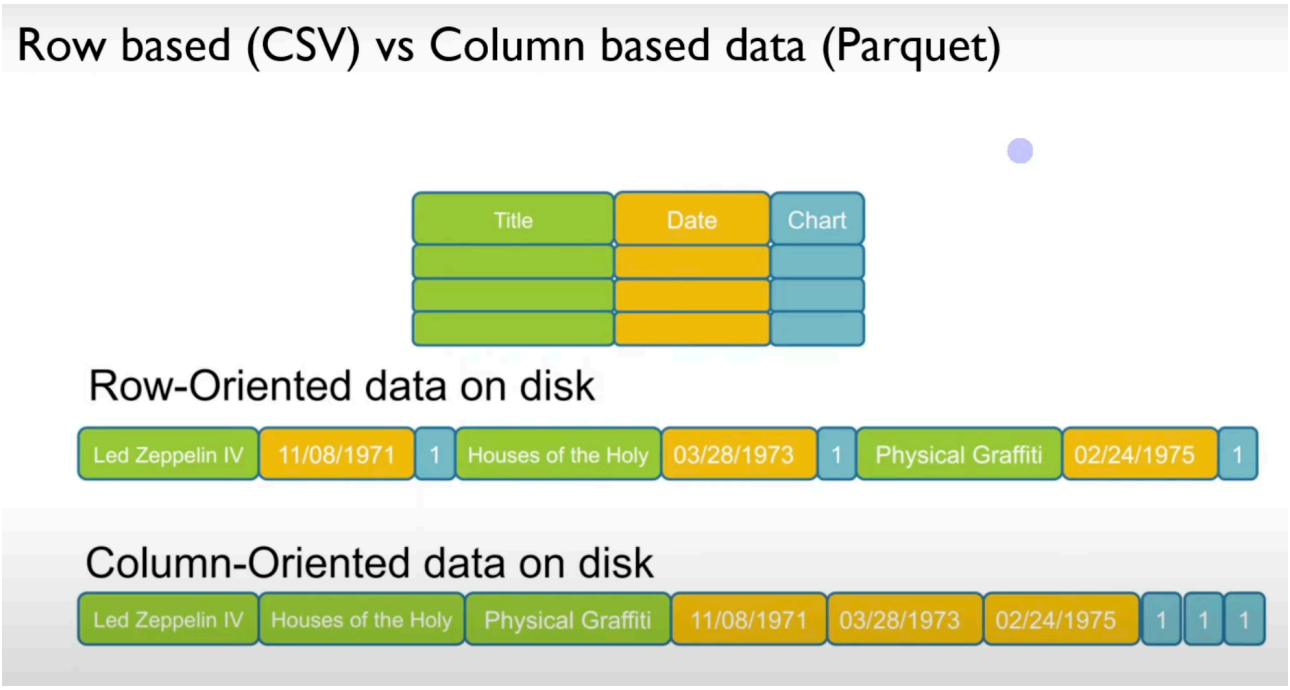
In PySpark, when writing data to external storage systems like HDFS, S3, or databases, you can specify different write modes to control how the data is handled. These write modes determine what happens if the target data already exists. Here are the common write modes available in PySpark:

1. Overwrite (mode="overwrite")
This mode overwrites the existing data at the target location with the new data being written. If the target directory does not exist, it will be created.
Use this mode when you want to completely replace existing data with new data.
2. Append (mode="append")
This mode appends the new data to the existing data at the target location. If the target directory does not exist, it will be created.
Use this mode when you want to add new data to existing data without overwriting it.

3. ErrorIfExists (mode="error" or mode="errorifexists")
This mode throws an error if the target data already exists. It prevents accidental overwriting of existing data.
Use this mode when you want to ensure that the target location is empty before writing data, avoiding accidental data loss.
4. Ignore (mode="ignore")
This mode ignores the write operation if the target data already exists. It does not overwrite or append to existing data, nor does it throw an error.
Use this mode when you want to perform the write operation only if the target location is empty.
Example Usage:
python

Copy code
# Example DataFrame df = spark.createDataFrame([(1, "Alice"), (2, "Bob")], ["id", "name"]) # Write DataFrame to Parquet format with different write modes
df.write.mode("overwrite").parquet("/path/to/output/overwrite")
df.write.mode("append").parquet("/path/to/output/append")
df.write.mode("error").parquet("/path/to/output/error")
df.write.mode("ignore").parquet("/path/to/output/ignore")
In this example, we're writing a DataFrame to Parquet format with different write modes specified.
Note:
These write modes are not specific to Parquet format; they can be used with other formats supported by PySpark, such as CSV, JSON, Avro, ORC, etc.
The default write mode in PySpark is error, meaning if you don't specify a mode explicitly, PySpark will throw an error if the target data already exists.

Csv vs parquet



Apache Spark, "layers" typically refer to different levels or components within the Spark ecosystem that work together to enable distributed data processing. Here are the common layers in Spark:

How do you optimize SQL queries in Spark?

By using techniques like query rewriting, predicate pushdown, and column pruning. ??

Predicate push down is a optmization technique in which we doing the filter then loading the data frame so in place of reading all data we are reading data on the basis of filter it helps to reduce shuffle between network .
Example :

```
# PySpark example
from pyspark.sql import SparkSession

# Initialize Spark session
spark = SparkSession.builder.appName("PredicatePushdownExample").getOrCreate()

# Read data from a Parquet file
df = spark.read.parquet("path/to/parquet/files")

# Apply a filter (predicate)
filtered_df = df.filter(df["order_date"] >= "2023-01-01")

# Perform some action to trigger the query execution
filtered_df.show()
```

in this example, Spark will push the order_date >= '2023-01-01' filter down to the Parquet file reader. The reader will only load the rows that match the predicate, reducing the amount of data read and processed.
This will print the physical plan of the query. Look for signs that the filter is being applied at the scan level, indicating that predicate pushdown is happening.

```
filtered_df.explain(True)
```

Predicate pushdown is most effective with columnar file formats like Parquet and ORC.

In Apache Spark, a job represents a complete computation task that needs to be executed to achieve a specific goal, such as executing a Spark action like collect() or saveAsTextFile(). A Spark job typically consists of multiple stages, and each stage consists of one or more tasks. Let's break down these concepts:

1. Job:
Definition: A job represents a complete computation that Spark needs to perform to execute a Spark action on an RDD or DataFrame.
Triggering Action: A job is triggered when a Spark action is called, such as collect(), saveAsTextFile(), or count().
Example: When you call collect() on an RDD or DataFrame, Spark will execute a job to collect the data from all partitions and return it to the driver program.
2. Stage:
Definition: A stage is a set of tasks that can be executed together, typically as a result of a narrow transformation (one-to-one mapping) like map() or filter().
Partitioning: Each stage corresponds to a partition of the input data, and the tasks within a stage operate on a single partition of the input data.
Example: If you perform a map() operation on an RDD, Spark will create a stage where each task executes the map() function on a single partition of the RDD.
3. Task:

Definition: A task is the smallest unit of work in Spark and represents the execution of a computation on a single partition of data.
Execution: Tasks are executed in parallel across the Spark cluster on different executor nodes.
Example: If you have an RDD with 100 partitions and you perform a map() operation, Spark will create 100 tasks, each executing the map() function on a single partition of the RDD.
Example:
Let's consider an example where we read data from a text file, perform some transformations, and then call an action like count():
python

```
from pyspark.sql import SparkSession
 # Initialize SparkSession spark = SparkSession.builder.appName("SparkExample").getOrCreate()
# Read data from text file lines = spark.read.text("/path/to/file.txt") # Perform transformations
filtered_lines = lines.filter(lines.value.contains("Spark"))
# Trigger an action count = filtered_lines.count()
```

In this example:
Reading data from the text file triggers a job.
The transformations (filter) create a stage, with each partition being processed by a task.
Calling count() triggers another job, which may involve one or more stages depending on the complexity of the computation.

To determine the number of partitions in a DataFrame in Apache Spark, you can use the rdd.getNumPartitions() method on the DataFrame's underlying RDD. Here's how you can do it:

python

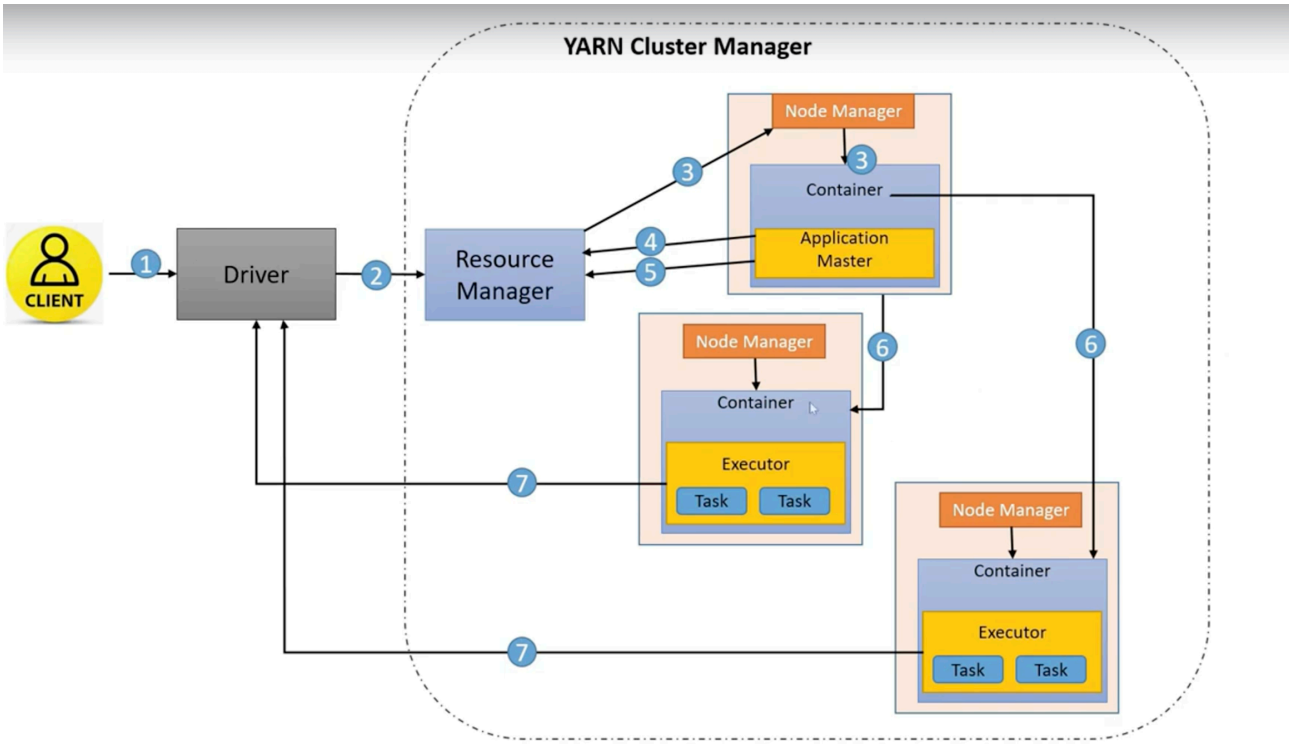
```
# Assuming 'df' is your DataFrame num_partitions = df.rdd.getNumPartitions() print("Number of partitions:", num_partitions)
```

In PySpark, when reading data into a DataFrame, you can specify various modes to control the behavior when the data source already exists. Here are the common modes used when reading a DataFrame:

1. Permissive Mode (mode="permissive")
Description: In permissive mode, when encountering corrupted records or parsing errors, Spark sets the fields to null and includes them in a "_corrupt_record" column.
Usage: Permissive mode is suitable when dealing with potentially corrupted data, allowing you to read as much data as possible while capturing errors for further investigation.
Example:

```
df = spark.read.option("mode", "permissive").csv("/path/to/data")
```

2. Drop Malformed Mode (mode="dropmalformed")
Description: In drop malformed mode, Spark skips and drops the rows with corrupted records or parsing errors.
Usage: Drop malformed mode is useful when you want to ignore or discard records that cannot be properly parsed, without including them in the DataFrame.
Example:python



Copy code
df = spark.read.option("mode", "dropmalformed").csv("/path/to/data")

3. Fail Fast Mode (mode="failfast")
Description: In fail fast mode, Spark throws an exception immediately upon encountering corrupted records or parsing errors.
Usage: Fail fast mode is suitable when you want to halt the execution and raise an error as soon as any data integrity issues are detected.
Example:python

Copy code
df = spark.read.option("mode", "failfast").csv("/path/to/data")

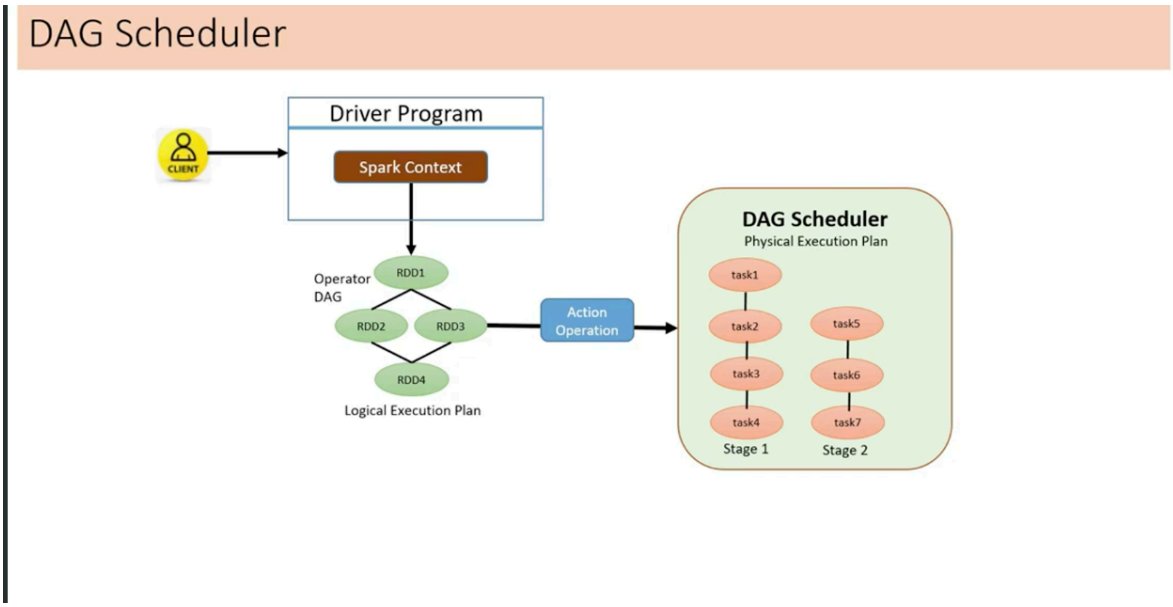
Example Usage:
python

Copy code
Example DataFrame creation with different read modes df_permissive = spark.read.option("mode", "permissive").csv("/path/to/data") df_dropmalformed = spark.read.option("mode", "dropmalformed").csv("/path/to/data") df_failfast = spark.read.option("mode", "failfast").csv("/path/to/data")
In this example, we're reading a CSV file into DataFrames using different read modes. You can replace "csv" with "parquet", "json", or any other data source format supported by PySpark.

These are the step of spark execution architecture.

So at first client submit the spark application Driver instantiate the spark context Driver talks to the resource manager and negotiate the resources.

The yarn resource manager searched for a node manager which will in turn launch an application master for that specific job in a container.
The Applicationmaster registered itself with the resource manager.
So that resource manager knows that Applicationmaster is launched and ready to use.
The Applicationmaster then negotiate with resource manager for containers.
It can request for more resources from resource manager if it is required.
Then the application master.
Would notify the node manager to launch the containers and executors and the executor, then execute the tasks.
The driver would communicate with executors to coordinate the processing of tasks of an application.
Once the job is complete, the application master unregisters with the resource manager.



So now in this step, all this narrow transmissions like filter map, etcetera, will fuse together **INTO ONE STAGE.THE NEW STAGE WILL BE FORMED ONLY WHEN IT ENCOUNTERS ONE WIDE TRANSFORMATIONS.**
With some shuffle steps.Okay, so once you get a WIDE transformation like Reducebykey, it will create the new stages.So all the stages and tasks would be created in DAG Scheduler and that we call as the physical execution plan.And after this, DAG scheduler bundles all the tasks and send it to the task scheduler.Then task Scheduler will submit the tasks to the cluster manager for execution .

It starts with an user submit application to Spark.
Then Driver takes the application and create spark context to process the application.
Then Spark Context would identify all the transformations and action operations present in the application.Then all these operations are arranged in logical flow of operations called DAG.
We can also call it logical execution plan, and it would be stopped here if Spark Context does not find any action operations.
If it identifies an action Operation Spark, submit the operator DAG to DAG Scheduler.
So the DAG scheduler converts the logical execution plan into physical execution plan and creates different stages and tasks.
Here, you would see all these narrow transformations would be fused together to form one stage.
Okay.And if it encounters any wide transformations and shuffle step, it will create new stages.
Okay.And then dark scheduler bundles all the tasks and send it to task scheduler which then submit the tasks to cluster manager for execution.

Right now, let's find out the number of partitions for this text file.And as you see here, there are two partitions.Okay.For this text file added.So that is why you see two tasks in the output.
So you see two tasks in the output.DAG because the input added has two partitions.
You can change the number of partitions using repartition in coalesce as explained in the previous chapter.Okay.But if you don't change it, you only see two tasks in the output.

The number of task submitted is depend on the number of partitions to get the partition we can use get partition and can check how many task will make.

Rdd persistence:

in PySpark, RDD persistence (or caching) is used to store RDDs in memory or on disk, which can significantly improve the performance of your Spark applications, especially for iterative algorithms and complex data pipelines. Let's go through a detailed example with code to illustrate the use of RDD persistence.
Persistence Levels in PySpark
The persistence levels available in PySpark are:
MEMORY_ONLY: Stores RDD as deserialized Python objects in the JVM. If there is not enough memory, some partitions will not be cached and will be recomputed when needed.
MEMORY_AND_DISK: Stores RDD as deserialized Python objects in the JVM. If memory is insufficient, it stores the partitions that do not fit in memory on disk.
MEMORY_ONLY_SER: Stores RDD as serialized Python objects in the JVM. This reduces memory usage but increases CPU overhead.
MEMORY_AND_DISK_SER: Similar to MEMORY_ONLY_SER, but spills partitions that do not fit in memory to disk.
DISK_ONLY: Stores RDD partitions only on disk.
OFF_HEAP: (Experimental) Stores RDD in off-heap memory.

Shared variable:

in Apache Spark, shared variables are variables that can be shared across tasks and workers. They provide a mechanism to update and read variables in a distributed manner. There are two main types of shared variables in Spark:

Broadcast Variables: Used to distribute large read-only data efficiently to all workers.

Broadcast variables allow the programmer to keep a read-only variable cached on each machine rather than shipping a copy of it with tasks. They are useful when you have a large dataset that is used across multiple stages of the computation, saving the cost of serialization and shipping the data multiple times.

Example of state full name after look from state dictonary

Accumulators: Used to perform counters and aggregators, where workers can update the variable, and the driver can read the final aggregated

Accumulators are special variables in Apache Spark that are used to perform a cumulative sum of a variable across the entire cluster. They are primarily used for counting events, sums, or other aggregate information during distributed processing. Workers can update accumulators, but only the driver program can read their value.
Key Characteristics of Accumulators:
Write-Only by Workers: Workers can only add to the accumulator and cannot read its value.

Read-Only by Driver: The driver program can read the accumulated value after all tasks are completed.
Fault Tolerance: Accumulators are resilient to task failures and will be recomputed if tasks are retried.

Example calculate how many empty line in text so in this case we can use accumulator

Logical plan vs physical plan ?

In logical plan we just check the from where data is joining and how many aggregation is happening how many file are scaning. But in physical plan we elaborate the what type of join is happening and what type of file is scanning and what aggregation is happening .

Physical Plan: A detailed, executable plan that describes how the logical plan will be executed in the Spark cluster. It includes execution strategies, optimizations, and low-level details about data processing.

Column pruning ?

The is part of query optimisation in this what is happening assume we joining two table and in the final we are selecting only two or three column so what will happen table will scan all the column then it will join so in column pruning it will select only two column which is used in the entire query .

Whole stage is used in execution or not ?

Whole stage optimised the query using tungsten and to verify it is applied in the code or not
We checked from physical plan to see we need to use this

```
# Show the logical plan
print("Logical Plan:")
result_df.explain(extended=True)
```

After that if code have * in the beginning it is using other wise its not .

```
*(1) Project [name#17, age#18L]
+- *(1) Filter (isnotnull(age#18L) AND (age#18L > 21))
   +- *(1) Scan ExistingRDD[name#17,age#18L,gender#19]
```

Catalyst optimizer ?

Catalyst optimiser help to optimise the execution plan of the query which is submitted .
And the optimized plan would then go to a tungsten execution engine, which will generate the code and then run the code in a cluster like a distributed fashion.

This is all about spark sql architecture.

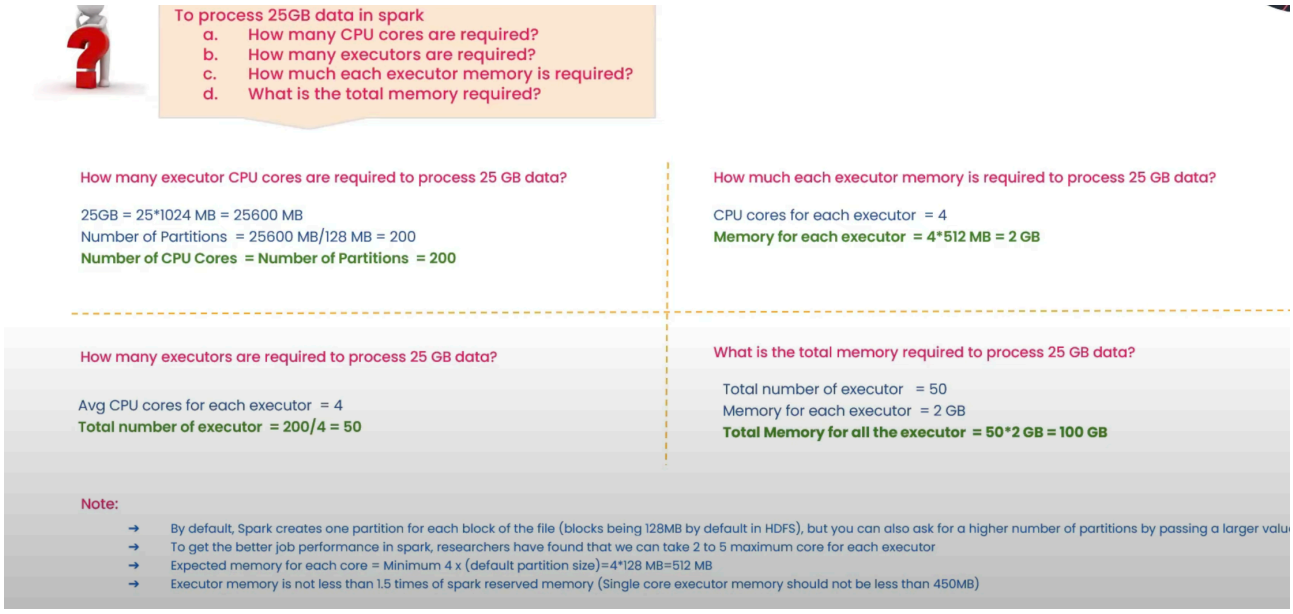
Infer schema

It help when we are load data from any path and want to detect data type of column automatically so we need to set as true it is very helpful

Files in python:

CSV, JSON and AVRO are Row-based File formats. Sample data in CSV:
ORC,PARQUET are Column-based File formats.

For few elected column columnbased files format is good , vice versa for if you want to select all



Spark creates one partition for each block of the file (default block size is 128MB in HDFS).
To achieve better performance, it is recommended to have 2 to 5 CPU cores per executor.
Expected memory for each core is at least 512 MB (i.e., 4 cores would require 4 * 128 MB = 512 MB).
Executor memory should not be less than 1.5 times the Spark reserved memory (single core executor memory should not be less than 450MB).

Catalog in spark sql:

Catalog is the interface to work with a metastore, i.e. a data catalog of database(s), local and external tables, functions, table columns, and temporary views in Spark SQL.
You can access the current catalog using SparkSession.catalog attribute.

In spark if you create a data frame then 2 partition will get created

Off heap memory :
The memory which lies outside the jvm

Serilaztion vs deserialisation

Serilaztion

At a memory level, serialization involves converting the state of an object into a sequential stream of bytes, which can then be stored in memory, on disk, or transmitted over a network. This process typically involves flattening out the object's data and converting it into a format that can be easily written to memory or disk.
Deserialization, on the other hand, reverses this process. It involves reading the serialized bytes from memory or disk and reconstructing the object's state from them. This process typically involves allocating memory for the object and populating it with the data from the serialized stream.
So, in memory, serialization involves converting an object's state into a stream of bytes, while deserialization involves reconstructing the object's state from that stream of bytes.

innerJoin vs leftantijoin vs leftsemijoin

Left anti join and left join are much faster then inner join we can use in place off inner only in case If we are looking the the data from left table and not looking for matched and unmatched from right data from then we can use left anti join but if you want to use left semi join then make sure you do not want to access any column from right table but it will behave as same only issue with select clause for left semi join

Focus on multi table join and column and self join syntax in pyspark

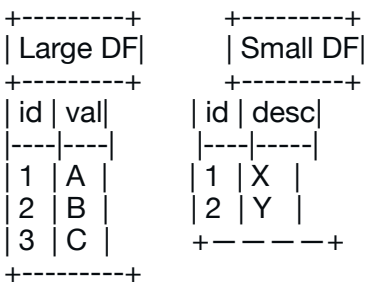
Broadcast join

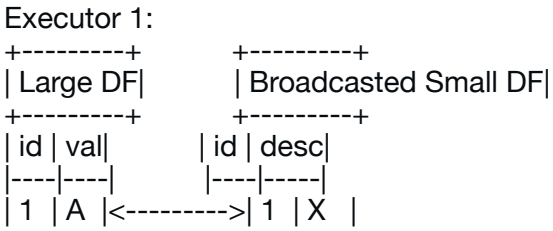
A broadcast join in Apache Spark is a join operation where the smaller DataFrame is broadcasted to all the worker nodes, so it can be joined with the larger DataFrame in parallel. This approach is highly efficient for cases where one DataFrame is much smaller than the other, as it minimizes data shuffling across the network.

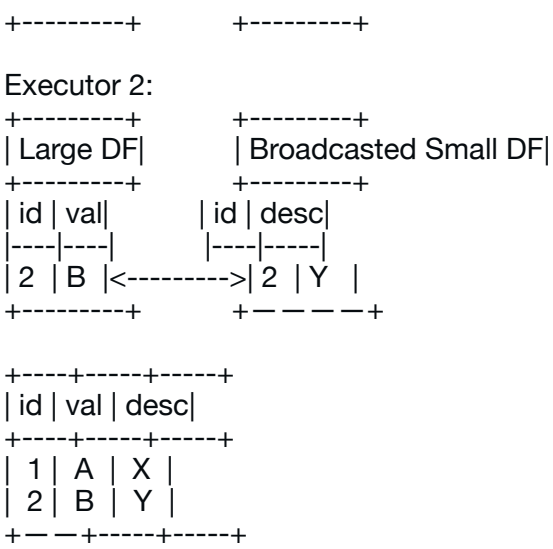
Spark sends a copy of the smaller DataFrame to all the executors (worker nodes) in the cluster. This is done using a broadcast variable, which ensures that each executor has a local copy of the small DataFrame.

Spark automatically selects the smaller DataFrame for broadcasting based on its size, or it can be manually specified using the broadcast function.

The results of the join operation from all executors are collected and aggregated to produce the final joined DataFrame.







joined_df = large_df.join(broadcast(small_df), “id”)

Hash merge join :

To execute this join we need to set the sort merge join false then it will start performing because by default sort merge join is TRUE so if set false so it will either choose any of them it mostly trigger when the data not compositely much lesser as we observed in case broadcast joined.
If you want to force optimiser of the certain data frame for spark application we can forcefully applied to the data frame using hint condition and we can observe the timing .

After broadcast join mostly prefer shuffle sort merge join after wards shuffle hash join .

Cross join is very risky and it by default set as false because it occurs so much scuffle across executors .

how we can decide. how many partition is required for any task in spark

Broadcast join and broadcast variables are related concepts in Apache Spark, but they serve different purposes and are used in different contexts. Understanding the distinction between them is key to optimizing performance in Spark jobs, particularly when dealing with large datasets and distributed computations.

Broadcast Join
A broadcast join in Spark is a type of join operation where one of the DataFrames (typically the smaller one) is broadcasted to all the worker nodes. This means that the smaller dataset is sent once to all the nodes, allowing them to perform the join locally without shuffling the larger dataset. This can significantly reduce the amount of data transferred across the network and speed up the join operation.

Use Case: Broadcast joins are useful when one of the datasets in a join operation is significantly smaller than the other. By broadcasting the smaller dataset, Spark can avoid the costly shuffle operation required to join large datasets.
Efficiency: Broadcast joins are efficient because they minimize the network traffic by sending the smaller dataset once to each node, rather than multiple times as part of the shuffle process.
Implementation: In Spark, you can use the broadcast() function to indicate that a DataFrame should be broadcasted. For example, val broadcastedDF = broadcast(smallDF) in Scala or broadcast(smallDF) in Python.
Broadcast Variables
Broadcast variables are an abstraction provided by Spark to efficiently share large, read-only data across all nodes in a Spark cluster. Broadcast variables are used for tasks other than joins, where you need to share large data across tasks without sending it repeatedly. Once a variable is broadcasted, it is available on all nodes in the cluster, allowing tasks to access it without the overhead of network communication.
Use Case: Broadcast variables are used for sharing large, read-only data like lookup tables, configurations, or other constants that need to be accessed by multiple tasks in a distributed environment.
Efficiency: They improve efficiency by reducing the amount of data transferred over the network. The variable is sent once to each node and then accessed locally.
Implementation: In Spark, you can create a broadcast variable using sparkContext.broadcast() in Python or sc.broadcast in Scala.
Comparison
Primary Focus:
Broadcast Join: Specifically used in the context of join operations to broadcast a smaller dataset to all nodes, avoiding shuffling of a large dataset.
Broadcast Variable: A more general concept used for sharing large, read-only data across tasks in a distributed environment, not limited to join operations.
Usage:
Broadcast Join: Used in SQL queries or DataFrame operations where a join is needed, and one of the datasets is small enough to be broadcasted.
Broadcast Variable: Used in various operations where large datasets or configuration data need to be accessed across multiple tasks without repeated network transfer.
Implementation:
Broadcast Join: You typically trigger a broadcast join using the broadcast() function within a join operation.
Broadcast Variable: You create a broadcast variable using broadcast() from SparkContext, and it can be accessed in multiple stages of the computation.
Understanding these concepts and when to use each can significantly impact the performance of your Spark applications, particularly when dealing with large datasets and complex transformations.

Local Mode

Description: Runs Spark on a single machine, without requiring a cluster manager. This mode is mainly used for development, testing, and debugging.
Usage: spark-submit --master local[<n>] ...
local: Use one thread.
local[K]: Use K threads.
local[*]: Use all available cores on the machine.
Advantages: Easy setup, no need for a cluster manager.
Disadvantages: Limited to the resources of a single machine.

Read write as data frame

```
input_path = 'gs://your-bucket/input/orc-file.orc'
df = spark.read.orc(input_path)
```

```
transformed_df = df.filter(df['column_name'] > some_value)
output_path = 'gs://your-bucket/output/transformed-orc-file.orc'
transformed_df.write.orc(output_path)
```

Csv

```
input_path = 'gs://your-bucket/input/csv-file.csv' csv_
df = spark.read.csv(input_path, header=True, inferSchema=True)
output_path = 'gs://your-bucket/output/transformed-csv-file.csv'
transformed_csv_df.write.csv(output_path, header=True)
```

Text

```
split_col = split(text_df['value'], ',')
# Creating a new DataFrame with columns split
text_split_df = text_df.withColumn('col1', split_col.getItem(0)) \ .
withColumn('col2', split_col.getItem(1)) \ .
withColumn('col3', split_col.getItem(2))
output_path = 'gs://your-bucket/output/transformed-text-file.txt'
transformed_text_df.write.text(output_path)
```

Here we used delimiter because text file have some must punctuation or delimiter which separate everything.