1. What is PySpark?

Answer: PySpark is the Python API for Apache Spark, allowing you to use Spark's distributed data processing capabilities with Python.

2. PySpark Architecture and its features?

Answer: PySpark is the Python API for Apache Spark, which uses a master-slave architecture and is designed for distributed data processing.

The Apache Spark framework uses a master-slave architecture that consists of a driver, which runs as a master node, and many executors that run across as worker nodes in the cluster. Apache Spark can be used for batch processing and real-time processing as well.

1. Driver Program

- The entry point of your PySpark application.
- Converts Python code into a DAG (Directed Acyclic Graph) of stages.
- Coordinates execution and manages the SparkContext.

2. Cluster Manager

- Allocates resources across the cluster.
- Types: Standalone, YARN, Mesos, Kubernetes.

3. Executors

- JVM processes running on worker nodes.
- Each executor runs tasks and stores data in memory (caching).

4. Tasks

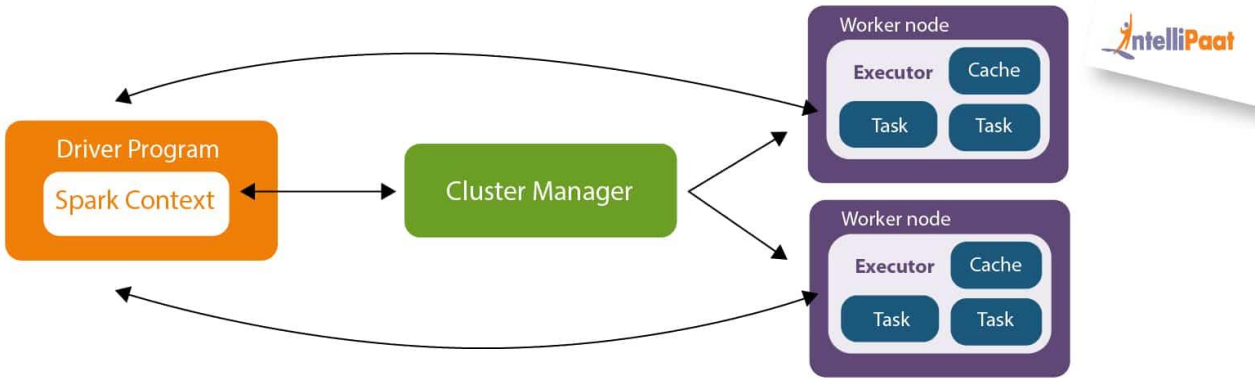
- Units of work sent to executors.
- Tasks are executed in **parallel** across partitions of the data.

5. RDDs / DataFrames

- PySpark translates DataFrame operations into RDD transformations internally.
- Optimized execution via **Catalyst Optimizer** and **Tungsten Engine**.

6. Job Execution Flow

1. User writes code using PySpark DataFrames.
2. Code is translated to logical plan → optimized plan.
3. Spark Scheduler breaks it into stages → tasks.
4. Tasks run on executors in parallel.



Key Features of PySpark:

Feature	Description
In-Memory Processing	Speeds up computation by storing intermediate results in memory.
Fault Tolerance	Uses lineage of RDDs to recompute lost data automatically.
Lazy Evaluation	Transformations are not executed until an action is triggered.
Scalability	Handles petabytes of data across thousands of nodes.
Language Support	Supports Python (via PySpark), Java, Scala, and R.
Advanced Analytics	Built-in support for SQL, machine learning (MLlib), graph processing.

Feature	Description
DataFrame & SQL APIs	Provides structured processing APIs for ease and optimization.
Integration	Connects easily with HDFS, Hive, GCS, S3, JDBC, Kafka, and more.
Catalyst Optimizer	Optimizes logical plans for query execution.
Tungsten Execution Engine	Performs memory management, code generation, and CPU optimization.

3.What are RDDs, DataFrames, and Datasets?
Answer:
RDD (Resilient Distributed Dataset): Low-level, immutable distributed collection.
DataFrame: Distributed collection of data organized into named columns, like a table.
Dataset: Type-safe interface available in Scala/Java (not supported in Python)

4. Why would you use RDDs over DataFrames in PySpark?
Answer:

RDDs allow low-level transformations and actions, ideal for custom processing logic.
RDDs are suitable when working with raw, unstructured data (e.g., parsing logs or custom file formats).
RDDs don't require a predefined schema like DataFrames. You can work with Python objects directly.
Some older PySpark operations or APIs work only with RDDs.

5. How would you handle missing or additional columns?
Answer:
1. Handling Missing Columns:
When Reading Data:
If expected columns are missing in the incoming data (e.g., schema evolution), you can:

```
from pyspark.sql.types import StructType, StructField, StringType

# Define a full schema
schema = StructType([
   StructField("id", StringType(), True),
   StructField("name", StringType(), True),
   StructField("age", StringType(), True),  # This column might be missing in data
])

# Read with schema
df = spark.read.schema(schema).json("path/to/file.json")
```

After Reading (Add Missing Columns):
If a DataFrame is missing a column and you want to align it with a target schema:

```
from pyspark.sql.functions import lit

# Add missing column
df = df.withColumn("age", lit(None).cast("integer"))
```

2. Handling Additional Columns:
If incoming data has extra columns not present in your expected schema:
Drop Unwanted Columns:

```
df = df.drop("unexpected_col1", "extra_field")
```

Select Only Required Columns:

```
required_columns = ["id", "name", "age"]
df = df.select([col for col in required_columns if col in df.columns])
```

6. What is an RDD in PySpark?

Answer: RDD stands for Resilient Distributed Dataset. It is the core abstraction in Apache Spark for working with distributed collections of data.

An RDD is an immutable, distributed collection of objects that can be processed in parallel across a cluster.

Feature	Description
Resilient	Can recover from node failures automatically using lineage info
Distributed	Data is split across multiple nodes in the cluster
Immutable	Once created, the data in an RDD cannot be changed
Lazy Evaluation	Transformations (e.g., map, filter) are not executed until an action is called
Fault Tolerant	Automatically recomputes lost data due to failures
In-memory computation	Offers fast processing through in-memory storage

7.How to Create RDDs?

Answer:

1.Parallelizing a Collection
2.Reading from a file
3.Transforming an Existing RDD

1.Parallelizing a Collection: Useful when small data such as list, tuple, arrays wants to process in it.

```
from pyspark.sql import SparkSession
spark=SparkSession.builder.getOrCreate()

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

x = sc.parallelize([1,2,3])
y = x.filter(lambda x: x%2 == 1) #keep odd values
print(x.collect()) #[1,2,3]
print(y.collect()) #([1,3])

names=["Ajay","Balu","Dinesh","Chaitu","Arun","Charan"]
x=sc.parallelize(names)
y=x.groupBy(lambda w:w[0])
print(y.collect())
print ([(k,list(v)) for (k,v) in y.collect()])
#[('B', ['Balu']), ('C', ['Chaitu', 'Charan']), ('A', ['Ajay', 'Arun']), ('D', ['Dinesh'])]

x = sc.parallelize([('B',5),('B',4),('A',3),('A',2),('A',1)])
y=x.groupByKey()
print([(j[0],list(j[1]))for j in y.collect()])
#[('B', [5, 4]), ('A', [3, 2, 1])]
```

2.Reading from a file: Reading a file such as text, CSV files in large datasets.

```
Rdd=sc. textFile("path/to/file.txt")
print(rdd.collect())
```

3.Transforming an Existing RDD: Creating a New RDD by transforming existing RDD

```
Rdd1=sc.parallelize(data)
```

*Rdd2=Rdd1.map(lambda x:x*2)*
print(Rdd2.collect())

Operations:
1.Transformations: The transformation are operations that create new rdd or df from an existing one. Not modify original data
Transformations: map(), filter(), flatMap(), union(), distinct(),groupByKey(),join()
2.Actions: Actions are operations that returns a value or write data an external system.
Actions: collect(), count(), take(), reduce(), saveAsTextFile()

8. What is DataFrames and how many ways to create?
Answer:
A DataFrame in PySpark is a distributed collection of data organized into named columns, similar to a table in a relational database or a data frame in pandas.
It is built on top of RDDs but provides a higher-level abstraction with powerful optimizations via the Catalyst optimizer and Tungsten execution engine.
Key Features of PySpark DataFrame:

- Distributed and fault-tolerant
- Schema-aware (supports data types and column names)
- Optimized query execution
- Interoperable with various sources (CSV, Parquet, JSON, Hive, etc.)
- Supports SQL-like operations

Create DataFrames:
1.Collection of Data and Schema:
from pyspark.sql import SparkSession
spark=SparkSession.builder.getOrCreate()
data=[("john",1), ("james",2), ("mike",3)]
schema = ["name","id"]
df=spark.createDataFrame(data,schema)
df.display()

2.from RDD and Schema:
rdd=sc.parallelize(data)
df1=rdd.toDF(schema)
df1.display()

3.From External Files (CSV, JSON, Parquet, etc.):
 # CSV
df_csv = spark.read.csv("file.csv", header=True, inferSchema=True)

 # JSON
df_json = spark.read.json("file.json", mergeSchema = True)

 # Parquet
df_parquet = spark.read.parquet("file.parquet", mergeSchema = True)

#Text
df_text = spark.read.text("path/to/file.txt")

4. From a SQL Query:
df.createOrReplaceTempView("people")
result_df = spark.sql("SELECT Name, Value FROM people WHERE Value > 1")

5. From a Hive Table:
*df = spark.sql("SELECT * FROM hive_table_name")*

9.What is lazy evaluation in PySpark?
Answer:

Transformations in PySpark are lazy, meaning computations aren't executed until an action (like .collect() or .show()) is called.

Lazy Evaluation means PySpark doesn't execute transformations (like map, filter, select) immediately. Instead, it **builds a logical execution plan**, and only when an **action** like count(), collect(), or show() is called, does it actually run the computation.

```
# Assume a large dataset
df = spark.read.csv("big_dataset.csv", header=True, inferSchema=True)

# Transformations (lazy)
filtered_df = df.filter(df["age"] > 25)
selected_df = filtered_df.select("name", "age")

# No computation has happened yet

# Now we trigger an action
selected_df.show()
```

PySpark combines the filter and select operations into a single stage using Catalyst optimizer, avoiding intermediate computations. This makes it faster and more efficient, especially on large data.

10. How Caching Affects Lazy Evaluation?
Answer:

Caching tells Spark to persist the result of a DataFrame or RDD after it's computed the first time so it doesn't recompute it again.

```
df = spark.read.csv("big_dataset.csv", header=True, inferSchema=True)

# Apply transformation
filtered_df = df.filter(df["age"] > 25)

# Cache it
filtered_df.cache()

# First action - triggers actual computation
filtered_df.count()  # Computation + caching

# Second action - uses cached data
filtered_df.show()   # Much faster
```

Even though the transformations are lazy, once you cache and trigger an action, the result is stored in memory (or disk if memory is limited). This can save time when the same data is reused multiple times in your workflow.

Concept	**Benefit**
Lazy Evaluation	Reduces unnecessary computation
Caching	Avoids recomputation on reuse

11. How do you handle null values in PySpark?

Answer:

Handling null values in PySpark is a common task during data preprocessing. PySpark provides a variety of ways to deal with nulls, depending on the context and data quality requirements.

1. Detect Null Values:

```
from pyspark.sql.functions import col
df.filter(col("column_name").isNull()).show()  #Show rows with nulls in a specific column
```

2. Drop Null Values:

```
df.na.drop() #Drop rows with any nulls
df.na.drop(how="all") #Drop rows only if all values are null
df.na.drop(subset=["column1", "column2"]) #Drop rows where specific columns are null
```

3.Fill Null Values:

Fill all nulls with a constant

```
df.na.fill(0)  # For numeric columns
df.na.fill("unknown")  # For string columns
```

Fill specific columns

```
df.na.fill({"age": 0, "name": "unknown"})
```

4. Replace Nulls Based on Conditions

```
from pyspark.sql.functions import when
df.withColumn("age", when(col(age).isNull(), 0).otherwise(col("age")))
```

12.What is a broadcast join and when should you use it?

Answer:

Broadcast Join:

- A broadcast join replicates a small table to all worker nodes so that the large table can be joined with it without shuffling.

```
from pyspark.sql.functions import broadcast
df1 = spark.read.csv("large_table.csv", header=True)
df2 = spark.read.csv("small_lookup_table.csv", header=True)
joined = df1.join(broadcast(df2), "key")
```

When to use it:

- One table is small enough to fit in memory (a few MBs to ~10MB depending on cluster settings).
- Avoid shuffle which is costly and slow for large datasets

Risks: Broadcast table must fit in memory of each executor. If it doesn't: Spark may throw an OOM (Out of Memory) error.

13.How can you optimize PySpark jobs?

Answer:

- Use caching wisely.
- Repartition or coalesce properly.
- Use coalesce() over repartition()
- Use broadcast joins.
- Avoid wide transformations like groupBy when possible.
- Use DataFrame/Dataset over RDD
- Avoid UDF's (User Defined Functions)
- Reduce expensive Shuffle operations

14. What is repartition() and coalesce() in PySpark?

Answer:

Both repartition() and coalesce() are used to control the number of partitions of a DataFrame or RDD in PySpark, which directly affects performance.

repartition()

- **Definition:** Redistributes the data into the specified number of partitions with a full shuffle across the cluster.
- **Usage:** Used to increase or decrease partitions.
- **When to use:** When you need even distribution of data, especially before joins or wide transformations.

coalesce()

- **Definition**: Merges existing partitions without a full shuffle; it's a more efficient way to reduce the number of partitions.
- **Usage**: Used to reduce the number of partitions.
- **When to use**: Before writing data to avoid creating too many small files.

repartition() – Example

```
df = spark.read.csv("large.csv", header=True)
df_repart = df.repartition(10)          # Increase to 10 partitions – triggers full shuffle
print(df_repart.rdd.getNumPartitions())  # Output: 10
```

coalesce() – Example

```
df = spark.read.csv("large.csv", header=True)
df_coalesced = df.coalesce(2)    # Reduce to 2 partitions – faster, less shuffle
print(df_coalesced.rdd.getNumPartitions())  # Output: 2
```

PySpark functions:

Initializing SparkSession

```
from pyspark.sql import SparkSession
spark=SparkSession.builder.getOrCreate()
```

Filter:

```
data = spark.read.csv("dbfs:/FileStore/data.csv",header=True,inferSchema=True)
data.filter(data["Pulse"]>=100).display()
data.filter(data["Calories"]=="NaN").display()
data[data.Maxpulse.isin(135,123,134)].display()
```

drop duplicates:

```
data=data.dropDuplicates()
```

GroupBy:

```
data.groupBy("Duration").count().show()
```

Sorting:

```
data.sort("Pulse").display()
data.sort(data.Date.desc()).display()
peopledf.sort(peopledf.age.desc()).collect()
df.sort(“age” , ascending=False).collect()
df.orderBy([ “age”, ”city” ],ascending=[0,1]).collect()
```

functions:

```
from pyspark.sql import functions as F
from pyspark.sql.functions import lit

data.select("Duration", F.when(data.Pulse>100,1).otherwise(0).alias("check")).display()
data.withColumn("Time",lit("")).display()
```

Updating Columns:

```
df = df.withColumnRenamed( 'old_name' 'new_name')
```

Adding Columns:

```
df.withColumn("Salary",lit(0))
```

Removing Columns:

```
df = df.drop( , "address" "phone Number")
df = df.drop(df.address).drop(df.phoneNumber)
```

Missing and replace null values:

```
df.na.fill(50).show()
df.na.drop().show()
 df.na.replace(10, 20) .show()
```

Running Queries Programmatically:

```
peopledf.createGlobalTempView(“people” )
df.createTempView( “cusmoter”)
df.createOrReplaceTempView( “customer”)

df5 = spark.sql( "SELECT * FROM customer").show()
peopledf2=spark.sql(“SELECT * FROM global_temp.people”).show()
```

Inspect Data:

```
df.dtypes  #Return df column names and data types
df.show()  #Display the content of df
df.head() #Return first n rows
df.first() #Return first row
df.describe().show() #Compute summary statistics
df.count() #Count the number of rows in df
df.printSchema() #Print the schema of df

spark.stop()
```

Usecase-1

from pyspark.sql import SparkSession
from pyspark.sql.functions import max, min, col, lit, when
spark = SparkSession.builder.getOrCreate()

1)Read the csv file into dataframe
custdf=spark.read.csv("dbfs:/FileStore/customers.csv",header=True,inferSchema=True)
custdf.display()

2) Show the max Age
custdf.orderBy("Age", ascending = False).show(1)

custdf.orderBy(custdf["Age"].desc()).show(1)

maxAge=custdf.select(max("Age")).collect()[0][0]
custdf.filter(custdf["Age"]==maxAge).display()

custdf.createOrReplaceTempView("customer")
spark.sql("select max(Age) as MaxAge from customer").show()

3)Show all the data order by Age asc
custdf.orderBy("Age",ascending=True).display()
custdf.orderBy(custdf["age"].asc()).display()

4)Show all the details of country belongs to USA
custdf.filter(custdf["CountryCode"]=="USA").display()
*spark.sql("select * from customer where CountryCode = 'USA' ").show()*

5)Make min Age as null
minAge=custdf.select(min("Age")).collect()[0][0]
custdf.filter(custdf['Age']==minAge).display()
custdf.withColumn('Age',when(col('Age')==minAge,lit(None)).otherwise(col('Age'))).display()

6) write to another csv file
custdf.write.mode("overwrite").csv("dbfs:/FileStore/customers1.csv")

custdf1 = spark.read.csv("dbfs:/FileStore/customers1.csv",header=True,inferSchema=True)
custdf1.display()
dbutils.fs.rm("dbfs:/FileStore/customers1.csv", True) #delete file

UseCase-2

from pyspark.sql import SparkSession
from pyspark.sql.functions import lit, col, when
spark=SparkSession.builder.getOrCreate()

1) Read CSV file into DataFrame
drizlydf=spark.read.csv("dbfs:/FileStore/Drizly_Customers_File.csv",header=True,inferSchema=True)
drizlydf.display()

2) Add an external column with the name salary

drizlydf.withColumn("Salary",lit(0)).display()

3) Add values into the column based on Age critiria

if the Age >= 50 then Salary is 1,00,000

if the Age >= 30 and <50 then Salary is 65,000

if the Age > 20 then Salary is 35,000

```
drizlydf.withColumn("Salary",
        when(col("Age")>=50,100000).
        when((col("Age")>=30) & (col("Age") <50),65000).when((col("Age")>20) &
(col("Age")<30),35000).otherwise(0)).display()
```

4)Remove all Null value Records

```
drizlydf1=drizlydf.withColumn("Product",when(col("Product")=="Null",None).otherwise(col("Product")))
drizlydf1=drizlydf1.withColumn("Company",when(col("Company")=="Null",None).otherwise(col("Company")))
drizlydf1.dropna().display()
```

5)Orderby Ascending order of Name

drizlydf.orderBy(drizlydf["Name"].asc()).display()

6) Groupby Location Name

drizlydf.groupBy("Location").count().display()