```
Start coding or generate with AI.
```

Here's a **structured explanation** for your topics:

✅ 1. PySpark Definition

PySpark is the **Python API for Apache Spark**, an open-source distributed computing framework designed for **big data processing** and **analytics**. It allows Python developers to leverage Spark's power for large-scale data processing tasks without writing Scala or Java.

Key Features of PySpark:

- **Distributed Computing**: Runs computations across a cluster using RDD (Resilient Distributed Dataset) or DataFrame API.
- **In-Memory Processing**: Faster than MapReduce as it uses memory for intermediate computations.
- **Fault Tolerance**: Uses RDD lineage to recompute lost data.
- **Lazy Evaluation**: Transformations are not executed immediately; Spark builds a DAG and optimizes execution.
- **APIs Supported**:
  - RDD API
  - DataFrame API
  - Spark SQL
  - MLlib (Machine Learning)
  - GraphX (Graph Processing)
  - Structured Streaming

When to Use PySpark:

- Processing **huge datasets** (GBs to TBs).
- ETL jobs on a distributed environment.
- Machine learning on large datasets.
- Real-time or batch data processing.

✅ 2. MapReduce vs Spark

MapReduce and Spark are both big data processing frameworks, but **Spark is an evolution of MapReduce** with significant improvements.

Feature	Hadoop MapReduce	Apache Spark
Processing Model	Disk-based (writes intermediate results to HDFS)	In-memory computation (uses RAM for speed)
Speed	Slower (due to disk I/O after each map/reduce step)	10-100x faster (in-memory processing)
Ease of Use	Requires Java code (complex)	Easy APIs in Python (PySpark), Scala, Java, R
Execution	Each job runs independently	DAG (Directed Acyclic Graph) optimizes pipeline
Fault Tolerance	Achieved via data replication in HDFS	Achieved via RDD lineage
Iteration Support	Poor for iterative jobs (writes to disk every time)	Excellent (keeps data in memory for reuse)
Streaming	Not supported natively	Supported via Spark Streaming / Structured Streaming
Libraries	Limited	Rich libraries (Spark SQL, MLlib, GraphX, Streaming)

Why Spark is Faster than MapReduce?

- **In-Memory Computing**: Keeps data in memory rather than writing intermediate results to disk.
- **Lazy Evaluation with DAG**: Optimizes the execution plan before running.
- **Better API & Integration**: Easy integration with ML, SQL, and streaming.

✅ **Real Interview Tip: Question:** *"If Spark is faster, why do some companies still use Hadoop MapReduce?"* **Answer:**

- Spark needs more memory; Hadoop can work well for disk-based processing when memory is limited.
- Hadoop ecosystem (HDFS, Hive) is mature and widely integrated.

```
Start coding or generate with AI.
```

Here's a **complete and structured note** on **how to read data into PySpark DataFrame with all supported formats and syntax**:

✅ 1. Reading Data in PySpark DataFrame

PySpark provides the `spark.read` interface to load data into a DataFrame from various sources.

General Syntax

```
df = spark.read.format("<format>").options(<key>, <value>).load("<path>")
```

or using **shortcut methods** like `read.csv()`, `read.json()`, etc.

✅ 2. Supported File Formats and Syntax

a) CSV Files

Basic Syntax

```
df = spark.read.csv("path/file.csv")
```

With Options

```
df = spark.read.format("csv") \
    .option("header", "true") \
    .option("inferSchema", "true") \
    .option("sep", ",") \
    .load("path/file.csv")
```

Common Options for CSV

- `header` → `"true"` or `"false"` (first row as column names)
- `inferSchema` → `"true"` (infers data types)
- `sep` → `","` or any delimiter
- `quote` → `"` (quote character)
- `escape` → `\` (escape character)

---

b) JSON Files

```
df = spark.read.json("path/file.json")
```

With Options

```
df = spark.read.format("json") \
    .option("multiLine", "true") \
    .load("path/file.json")
```

---

c) Parquet Files (Default Format in Spark)

```
df = spark.read.parquet("path/file.parquet")
```

With Options

```
df = spark.read.format("parquet") \
    .load("path/file.parquet")
```

✅ *Parquet is columnar, compressed, and best for performance.*

---

d) ORC Files

```
df = spark.read.orc("path/file.orc")
```

With Options

```
df = spark.read.format("orc") \
    .load("path/file.orc")
```

---

e) AVRO Files

(Requires `spark-avro` package if not bundled)

```
df = spark.read.format("avro") \
    .load("path/file.avro")
```

---

f) Text Files

```
df = spark.read.text("path/file.txt")
```

With Options

```
df = spark.read.format("text") \
    .option("wholetext", "true") \
    .load("path/file.txt")
```

---

g) Delta Lake Tables

(If Delta Lake is enabled)

```
df = spark.read.format("delta") \
    .load("path/to/delta-table")
```

---

h) Reading from Table (Hive / Catalog)

```
df = spark.read.table("database.table_name")
```

---

i) JDBC (Relational Database)

```
df = spark.read.format("jdbc") \
    .option("url", "jdbc:mysql://host:port/db") \
    .option("dbtable", "table_name") \
    .option("user", "username") \
    .option("password", "password") \
    .load()
```

---

j) Multiple Files

```
df = spark.read.csv(["file1.csv", "file2.csv"], header=True, inferSchema=True)
```

✅ Shortcut Methods for Common Formats

- `spark.read.csv(path)`
- `spark.read.json(path)`
- `spark.read.parquet(path)`
- `spark.read.orc(path)`
- `spark.read.text(path)`

✅ **Best Practice**: Always use `inferSchema` for type inference or explicitly define schema for better performance:

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

schema = StructType([
    StructField("id", IntegerType(), True),
    StructField("name", StringType(), True)
])

df = spark.read.csv("path/file.csv", schema=schema, header=True)
```

Interview questions on DataFrame read/write operations?**

Start coding or generate with AI.

Here are **detailed notes on `select()` in PySpark** with examples and interview tips:

✅ 1. What is `select()` in PySpark?

- The **`select()`** function in PySpark is used to:
  - Select specific columns from a DataFrame.
  - Apply expressions, transformations, or functions to columns.
  - Rename columns using aliasing.

It is equivalent to the **`SELECT`** statement in SQL.

✅ 2. Basic Syntax

```
df.select("col1", "col2", ...)
```

or

```
df.select(df["col1"], df["col2"])
```

or

```
from pyspark.sql.functions import col
df.select(col("col1"), col("col2"))
```

✅ 3. Examples

a) Selecting Specific Columns

```
df.select("name", "age").show()
```

b) Selecting All Columns

```
df.select("*").show()
```

---

c) Using `col()` or DataFrame Column Object

```
from pyspark.sql.functions import col
df.select(col("name"), col("salary")).show()
```

---

d) Renaming Columns (Using Alias)

```
df.select(col("name").alias("employee_name")).show()
```

---

e) Applying Expressions

You can use **expressions inside `select()`**:

```
from pyspark.sql.functions import expr
df.select(expr("salary * 0.10").alias("bonus")).show()
```

---

f) Selecting with Column Operations

```
df.select(col("salary") * 2).show()
```

---

g) Using `selectExpr()` (SQL-like Expressions)

```
df.selectExpr("name", "salary * 1.1 as updated_salary").show()
```

✅ 4. Common Use Cases of `select()`

✓ Extract specific columns. ✓ Perform column transformations. ✓ Rename columns for clarity. ✓ Use SQL-like expressions with `selectExpr()`.

✅ 5. Difference Between `select()` and `withColumn()`

- `select()` → Returns only the selected columns (can apply transformations).
- `withColumn()` → Adds or replaces a column while keeping all existing columns.

Example:

```
df.withColumn("bonus", col("salary") * 0.10)
```

✅ 6. Combining with Other Functions

You can combine `select()` with:

- **`agg()`** for aggregations.
- **`filter()`** for filtering before or after selection.
- **`groupBy()`** for grouped operations.

Example:

```
df.select("name", (col("salary") + 500).alias("updated_salary")).show()
```

✅ 7. Performance Tip

- `select()` is **narrow transformation** (no shuffling), so it is efficient.
- Use `select()` instead of `drop()` when you need only a few columns.

🔍 Interview Question

Q: What is the difference between `select()` and `selectExpr()`? **A:**

- `select()` → Uses column objects or names.
- `selectExpr()` → Accepts SQL expressions as strings (e.g., `"salary * 1.1 as new_salary"`).

```
Start coding or generate with AI.
```

Here are **detailed notes on `withColumn()` and `withColumnRenamed()` in PySpark** with examples and interview tips:

✅ 1. `withColumn()` in PySpark

The **`withColumn()`** function is used to:

- **Add a new column** to a DataFrame.
- **Update/replace an existing column** with a new value or transformation.

Syntax

```
DataFrame.withColumn(colName, col)
```

- **colName** → Name of the column to create or replace.
- **col** → Column expression or value.

✅ Examples of `withColumn()`

a) Add a New Column

```
from pyspark.sql.functions import col

df = df.withColumn("bonus", col("salary") * 0.10)
df.show()
```

---

b) Replace an Existing Column

```
df = df.withColumn("salary", col("salary") + 1000)
```

---

c) Add a Constant Column

```
from pyspark.sql.functions import lit

df = df.withColumn("country", lit("India"))
```

---

d) Apply Conditions

```
from pyspark.sql.functions import when

df = df.withColumn("status", when(col("age") > 30, "Senior").otherwise("Junior"))
```

---

e) Drop a Column (Alternative)

To drop, you can set the value as `None` or just use `drop()`:

```
df = df.drop("bonus")
```

✅ Key Points about `withColumn()`

✓ It **does not modify the DataFrame in place**; it returns a new DataFrame. ✓ It can **add or replace** columns. ✓ It is often combined with functions like `col()`, `lit()`, `when()`.

✅ 2. `withColumnRenamed()` in PySpark

The **`withColumnRenamed()`** function is used to:

- Rename **a single column** in a DataFrame.

Syntax

```
DataFrame.withColumnRenamed(existingName, newName)
```

✅ Examples of `withColumnRenamed()`

a) Rename One Column

```
df = df.withColumnRenamed("salary", "employee_salary")
```

---

b) Rename Multiple Columns

```
df = df.withColumnRenamed("name", "employee_name") \
       .withColumnRenamed("age", "employee_age")
```

---

c) Rename Columns Dynamically (Using Loop)

```
old_names = ["name", "age"]
new_names = ["employee_name", "employee_age"]

for old, new in zip(old_names, new_names):
    df = df.withColumnRenamed(old, new)
```

✅ Key Points about `withColumnRenamed()`

✓ Used **only for renaming** columns, not adding or modifying values. ✓ For multiple columns, you need to chain or use loops (or `toDF()` for all columns).

✅ 3. Difference Between `withColumn()` and `withColumnRenamed()`

Feature	`withColumn()`	`withColumnRenamed()`
Purpose	Add or replace a column	Rename a column
Accepts Expression?	✅ Yes	❌ No (only old and new column names)
Used For	Transformations, adding new derived columns	Renaming existing columns

✅ 4. Alternative for Multiple Column Renames

```
df = df.toDF("emp_name", "emp_age", "emp_salary")
```

🔍 Interview Questions

Q1: What happens if the column name already exists in `withColumn()`? ✓ It **replaces** the column with the new value.

Q2: Can you rename multiple columns in one go? ✓ Yes, by using **`toDF()`** or chaining multiple `withColumnRenamed()` calls.

👉 Do you want me to **prepare next topic as `drop()`, `selectExpr()` and `alias()` with real-time scenarios** OR **cover performance implications of `withColumn()` vs SQL approach**?

```
Start coding or generate with AI.
```

Here are **detailed notes** on **`filter()`**, **`drop()`**, and **`sort()`** functions in PySpark with syntax, examples, and interview tips:

✅ 1. `filter()` in PySpark

The **`filter()`** (or **`where()`**) function is used to **filter rows** in a DataFrame based on a condition.

Syntax

```
df.filter(condition)
df.where(condition)  # same as filter()
```

✅ Examples of `filter()`

a) Filter Rows Using String Condition

```
df.filter("age > 30").show()
```

---

b) Filter Rows Using Column Object

```
from pyspark.sql.functions import col
df.filter(col("age") > 30).show()
```

---

c) Multiple Conditions

```
df.filter((col("age") > 30) & (col("salary") > 50000)).show()
```

---

d) Using `isin()`

```
df.filter(col("department").isin("HR", "Finance")).show()
```

---

e) Using `startswith, endswith`

```
df.filter(col("name").startswith("A")).show()
```

✅ Key Points

- `filter()` and `where()` are **aliases**.
- Accepts **SQL expression** or **column-based condition**.
- Returns a **new DataFrame** (does not modify original).

✅ 2. `drop()` in PySpark

The **`drop()`** function is used to:

- Drop one or more **columns**.
- Drop **rows with NULL values** (when used with `na`).

Syntax

```
df.drop(*cols)  # for columns
df.na.drop()    # for rows with NULL
```

✅ Examples of `drop()`

a) Drop a Single Column

```
df = df.drop("age")
```

---

b) Drop Multiple Columns

```
df = df.drop("age", "salary")
```

---

c) Drop Rows with NULL Values

```
df = df.na.drop()
```

---

d) Drop Rows with NULL in Specific Columns

```
df = df.na.drop(subset=["name", "age"])
```

✅ Key Points

- For **columns**, use `drop()` directly.
- For **rows**, use `df.na.drop()`.
- Always returns a **new DataFrame**.

✅ 3. `sort()` in PySpark

The **`sort()`** function is used to:

- Sort rows based on one or multiple columns.
- Can sort in **ascending** (default) or **descending** order.

Syntax

```
df.sort("col1", "col2")
df.sort(col("col1").desc())
```

Alias: `orderBy()` (same as `sort()`)

✅ Examples of `sort()`

a) Sort by Single Column

```
df.sort("age").show()
```

---

b) Sort by Single Column (Descending)

```
from pyspark.sql.functions import col
df.sort(col("age").desc()).show()
```

---

c) Sort by Multiple Columns

```
df.sort(col("department"), col("salary").desc()).show()
```

---

d) Using `orderBy()`

```
df.orderBy(col("salary").desc()).show()
```

✅ Key Points

- `sort()` and `orderBy()` are **aliases**.
- Default is **ascending order**.
- Sorting is a **wide transformation** (causes shuffle → expensive operation).

🔍 Interview Questions

Q1: Difference between `filter()` and `where()`? ✓ Both are the same; `where()` is just an alias of `filter()` for SQL-style readability.

Q2: How do you remove NULL values from specific columns? ✓ Use `df.na.drop(subset=["col1", "col2"])`.

Q3: Which is better for sorting large data: `sort()` or `orderBy()`? ✓ Both are same internally; but sorting large data requires partitioning and may cause shuffle → expensive.

👉 Do you want me to **prepare the next topics** like: ✓ `groupBy()` with aggregations ✓ `distinct()` vs `dropDuplicates()` ✓ `limit()` and `sample()` ✓ **Real-time ETL examples using these functions**?

```
Start coding or generate with AI.
```

Here are **detailed notes** on **`groupBy()`**, **joins**, **union**, and **fill/fillna** in PySpark with syntax, examples, and interview tips:

✅ 1. groupBy() in PySpark

The **`groupBy()`** function is used to **group data based on one or more columns** and then apply **aggregation functions**.

Syntax

```
df.groupBy("col1", "col2").agg(aggregation_functions)
```

✅ Examples

a) Basic Group By

```
df.groupBy("department").count().show()
```

---

b) Multiple Aggregations

```
from pyspark.sql.functions import avg, max, min

df.groupBy("department").agg(
    avg("salary").alias("avg_salary"),
    max("salary").alias("max_salary"),
    min("salary").alias("min_salary")
).show()
```

---

c) Group By Multiple Columns

```
df.groupBy("department", "gender").count().show()
```

✅ **Key Points**

- Returns **GroupedData** object, which needs an **aggregation** (like `count()`, `avg()`).
- Used for summarizing data.
- Common with functions from `pyspark.sql.functions`.

✅ 2. Joins in PySpark

PySpark supports different types of **joins** similar to SQL.

Syntax

```
df1.join(df2, on="col", how="inner")
```

- **how** can be:
  - `"inner"` (default)
  - `"left"` / `"left_outer"`
  - `"right"` / `"right_outer"`
  - `"outer"` / `"full"`
  - `"semi"`
  - `"anti"`
  - `"cross"`

✅ Examples

a) Inner Join

```
df1.join(df2, df1.id == df2.id, "inner").show()
```

---

b) Left Join

```
df1.join(df2, df1.id == df2.id, "left").show()
```

---

c) Right Join

```
df1.join(df2, df1.id == df2.id, "right").show()
```

---

d) Full Outer Join

```
df1.join(df2, df1.id == df2.id, "outer").show()
```

---

e) Semi Join (Return rows from `df1` that have a match in `df2`)

```
df1.join(df2, df1.id == df2.id, "semi").show()
```

---

f) Anti Join (Return rows from `df1` that do NOT have a match in `df2`)

```
df1.join(df2, df1.id == df2.id, "anti").show()
```

✅ **Key Points**

- **Joins are wide transformations** (shuffle happens).
- Prefer **broadcast joins** for small tables:

```
from pyspark.sql.functions import broadcast
df1.join(broadcast(df2), "id")
```

✅ 3. `union()` and `unionAll()` in PySpark

- `union()` is used to **combine two DataFrames vertically (append rows)**.
- In Spark 2.x and above, **`union()` acts like `UNION ALL`** in SQL (keeps duplicates).
- There is **no separate `unionAll()`** in PySpark now.

Syntax

```
df3 = df1.union(df2)
```

✅ Examples

a) Union Two DataFrames

```
df3 = df1.union(df2)
```

---

b) Remove Duplicates After Union

```
df3 = df1.union(df2).distinct()
```

✅ **Key Rules**

- Both DataFrames must have the **same schema** (same number and order of columns, and same data types).
- If schema differs, use `selectExpr()` or `withColumnRenamed()` to align.

✅ 4. `fill()` and `fillna()` in PySpark

Both are used to **replace NULL values** in DataFrames.

Syntax

```
df.fillna(value, subset=None)
df.na.fill(value, subset=None)
```

- **value** can be:
  - A single value (applies to all columns).
  - A dictionary `{col_name: value}` for specific columns.

✅ Examples

a) Replace All NULL with 0

```
df.fillna(0).show()
```

---

b) Replace NULL in Specific Columns

```
df.fillna({"age": 0, "salary": 5000}).show()
```

---

c) Using `na` Submodule

```
df.na.fill("Unknown", ["name"]).show()
```

✅ **Key Points**

- `fill()` and `fillna()` are **aliases**.
- Use dictionary when different columns need different default values.

🔍 Interview Questions

Q1: What is the difference between `fill()` and `fillna()`? ✓ Both are same; `fill()` is an alias of `fillna()`.

Q2: Can `union()` combine DataFrames with different schemas? ✓ No, schemas must match. You need to align columns first.

Q3: How do you improve join performance? ✓ Use **broadcast joins** for small tables and partitioning for large tables.

👉 Do you want me to **prepare next topics** like: ✓ `distinct()` vs `dropDuplicates()` ✓ `limit()` and `sample()` ✓ `cache()` vs `persist()` ✓ **All NULL handling methods (fill, drop, replace) in depth with scenarios**?

```
Start coding or generate with AI.
```

Here are **detailed notes on `collect()` and `StructType` in PySpark** with syntax, examples, and best practices:

✅ 1. `collect()` in PySpark

The **`collect()`** function is used to:

- Retrieve **all rows of a DataFrame** (or RDD) **into the driver as a list**.
- Converts the distributed data into **local Python objects**.

Syntax

```
df.collect()
```

Returns a **list of Row objects**.

✅ Examples

a) Collect All Rows

```
data = df.collect()
for row in data:
    print(row)
```

---

b) Access Row Values

```
for row in df.collect():
    print(row["name"], row["age"])
```

---

c) Collect Specific Columns

```
names = [row["name"] for row in df.select("name").collect()]
```

✅ Key Points

✓ `collect()` brings **all data to the driver**, which can cause **OutOfMemoryError** for large datasets. ✓ Use **only for small datasets** or debugging. ✓ For large data, prefer:

- `take(n)` → First **n rows**.
- `show(n)` → Display first **n rows** (formatted).
- `toPandas()` → Convert to Pandas DataFrame (only for small data).

🔍 Interview Tip

Q: Why is `collect()` dangerous in production? ✓ Because it loads **all distributed data into driver memory**, which can crash if data is large.

✅ 2. `StructType` in PySpark

`StructType` is a **schema definition class** in PySpark used to define **custom schema** for DataFrames.

Why use `StructType`?

- For **explicit schema definition** instead of `inferSchema=True` (which is slower).
- Helps **control column names, data types, and nullability**.

Syntax

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

schema = StructType([
    StructField("name", StringType(), True),
    StructField("age", IntegerType(), True)
])
```

- **StructType** → Represents the full schema (collection of StructFields).
- **StructField** → Represents a single column (name, type, nullable).

✅ Examples

a) Define Schema and Read CSV

```
schema = StructType([
    StructField("id", IntegerType(), True),
    StructField("name", StringType(), True),
    StructField("salary", DoubleType(), True)
])

df = spark.read.csv("employees.csv", schema=schema, header=True)
```

---

b) Define Schema for JSON

```
schema = StructType([
    StructField("name", StringType(), True),
    StructField("details", StructType([
        StructField("age", IntegerType(), True),
        StructField("city", StringType(), True)
    ]))
])

df = spark.read.json("data.json", schema=schema)
```

---

c) Create DataFrame with Schema

```
data = [(1, "John", 5000), (2, "Alice", 7000)]

schema = StructType([
    StructField("id", IntegerType(), False),
    StructField("name", StringType(), True),
    StructField("salary", IntegerType(), True)
])

df = spark.createDataFrame(data, schema)
df.show()
```

✅ Common Data Types

- `StringType()`
- `IntegerType()`
- `DoubleType()`
- `BooleanType()`
- `TimestampType()`
- `DateType()`
- `ArrayType()`
- `MapType()`
- `StructType()`

✅ Nullable Option

- **True** → Column can have NULL values.
- **False** → Column cannot have NULL values.

🔍 Interview Tip

Q: Why define schema manually instead of `inferSchema`? ✓ Performance: `inferSchema` scans data twice (once to infer schema, once to read). Explicit schema avoids extra scan.

✅ Summary

- `collect()` → Brings **all data to driver** (use with caution).
- `StructType` → Explicit schema definition (for better performance and control).

👉 Do you want me to **cover next topics like**: ✓ `Row` class in PySpark ✓ `ArrayType`, `MapType`, and nested schema examples ✓ `take()`, `first()`, `head()` vs `collect()` ✓ **Schema**

evolution and handling corrupt records?

Start coding or generate with AI.

Here's a **complete guide on `pivot` and `unpivot` in PySpark** with syntax, examples, and interview tips:

✅ 1. Pivot in PySpark

Pivoting means **converting rows into columns** (like SQL `PIVOT` or Excel pivot table).

In PySpark, `pivot()` is used with **`groupBy()`** and an aggregation function (e.g., `sum()`, `avg()`, `count()`).

Syntax

```
df.groupBy(grouping_column).pivot(pivot_column).agg(aggregation_function)
```

✅ Examples

a) Basic Pivot Example

Input Data

dept	year	salary
IT	2020	5000
IT	2021	5500
HR	2020	4000
HR	2021	4200

Code

```
from pyspark.sql.functions import sum

df.groupBy("dept").pivot("year").sum("salary").show()
```

Output

dept	2020	2021
IT	5000	5500
HR	4000	4200

---

b) Pivot with Explicit Values

You can **limit pivot columns** for better performance:

```
df.groupBy("dept").pivot("year", [2020, 2021]).sum("salary")
```

---

c) Pivot with Multiple Aggregations

```
from pyspark.sql.functions import sum, avg

df.groupBy("dept").pivot("year").agg(
    sum("salary").alias("total_salary"),
    avg("salary").alias("avg_salary")
)
```

✅ Performance Tip

- Always **provide pivot values** explicitly when possible:

```
pivot("year", [2020, 2021])
```

This avoids **full shuffle** and scanning of all distinct values.

✅ 2. Unpivot in PySpark (Melt Operation)

Unpivot means converting **columns into rows** (like SQL `UNPIVOT` or Pandas `melt()`). PySpark doesn't have a direct `unpivot()` function, but we can achieve it using:

- **`selectExpr()` with `stack()` function**
- **`explode()` with arrays**

Using `stack()` (Most Common)

Syntax

```
selectExpr("id", "stack(n, 'col1', col1, 'col2', col2, ...) as (key, value)")
```

- `n` → number of columns to unpivot.
- Returns two columns: `key` (column name) and `value`.

✅ Example: Unpivot with `stack()`

Input Data

id	Q1	Q2	Q3
1	100	200	300

Code

```
df.selectExpr("id",
              "stack(3, 'Q1', Q1, 'Q2', Q2, 'Q3', Q3) as (Quarter, Sales)").show()
```

Output

id	Quarter	Sales
1	Q1	100
1	Q2	200
1	Q3	300

✅ Unpivot Using `explode()`

```
from pyspark.sql.functions import explode, array, lit

df.select("id",
          explode(array(
              lit(("Q1", col("Q1"))),
              lit(("Q2", col("Q2"))),
              lit(("Q3", col("Q3")))
          )).alias("quarter_sales")
)
```

Then split into two columns.

✅ 3. Difference Between Pivot and Unpivot

Feature	Pivot (Rows → Columns)	Unpivot (Columns → Rows)
Use Case	Summarize data by category	Normalize wide data into long
Function	`pivot()`	`stack()` or `explode()`

🔍 Interview Questions

Q1: Why is `pivot()` expensive in PySpark? ✓ Because it requires **full shuffle** and generates a **wide DataFrame**.

Q2: How to improve pivot performance? ✓ Provide explicit pivot values, cache intermediate data.

Q3: Is there a built-in `unpivot()` in PySpark? ✓ No, but we can use `stack()` or `explode()`.

👉 Do you want me to **also prepare next topics** like: ✓ `explode()` and `posexplode()` in detail ✓ `stack()` advanced examples ✓ Real-time ETL scenario: Pivot sales data and then unpivot for reporting?

```
Start coding or generate with AI.
```

Here's a **complete guide to UDF (User Defined Function) in PySpark** with **definition, syntax, examples, types, and performance tips**:

✅ 1. What is a UDF in PySpark?

- A **User Defined Function (UDF)** allows you to **define your own custom functions** and apply them to PySpark **DataFrames** or **RDDs**.
- Used when **built-in PySpark functions** (from `pyspark.sql.functions`) are not sufficient.
- UDFs operate **row by row** on DataFrame columns.

Why use UDFs?

✓ For custom logic that **cannot be expressed** using SQL functions or Spark built-in functions. ✓ For **data transformation**, e.g., cleaning, formatting, business rules.

✅ 2. Syntax for UDF

```
from pyspark.sql.functions import udf
from pyspark.sql.types import DataType

# Define Python function
def my_function(value):
    return value * 2

# Register as UDF
my_udf = udf(my_function, DataType)

# Use in DataFrame
df.withColumn("new_col", my_udf(df["col"]))
```

✅ 3. Types of UDFs in PySpark

1. **Regular UDF** → Works on single value columns.
2. **Pandas UDF (Vectorized UDF)** → Uses **Apache Arrow**, faster than regular UDF.
3. **SQL UDF** → Registered for SQL queries.

✅ 4. Regular UDF Example

```
from pyspark.sql.functions import udf
from pyspark.sql.types import StringType

# Python function
def upper_case(name):
    return name.upper() if name else None

# Register UDF
upper_udf = udf(upper_case, StringType())

# Apply UDF
df = spark.createDataFrame([("john",), ("mike",), (None,)], ["name"])
df.withColumn("upper_name", upper_udf("name")).show()
```

Output

name	upper_name
john	JOHN
mike	MIKE
null	null

✅ 5. Lambda Function with UDF

```
upper_udf = udf(lambda x: x.upper(), StringType())
df.withColumn("upper_name", upper_udf("name")).show()
```

✅ 6. Register UDF for SQL Queries

```
spark.udf.register("to_upper", upper_case, StringType())

df.createOrReplaceTempView("people")
spark.sql("SELECT name, to_upper(name) AS upper_name FROM people").show()
```

✅ 7. Pandas UDF (Vectorized UDF)

- **Faster** than regular UDF because it processes data in batches using **Apache Arrow**.
- Use `@pandas_udf` decorator.

Example

```
import pandas as pd
from pyspark.sql.functions import pandas_udf
from pyspark.sql.types import DoubleType

@pandas_udf(DoubleType())
def multiply_by_two(s: pd.Series) -> pd.Series:
    return s * 2

df = spark.createDataFrame([(1,), (2,), (3,)], ["value"])
df.withColumn("doubled", multiply_by_two("value")).show()
```

✅ 8. Performance Considerations

- Regular UDFs are **slow** because:
  - Data moves between **JVM and Python process** (serialization overhead).
- Use **built-in functions** whenever possible.
- If needed, prefer **Pandas UDF** (vectorized).
- Enable **Arrow optimization**:

```
spark.conf.set("spark.sql.execution.arrow.enabled", "true")
```

✅ 9. Common Interview Questions

✓ **Q1:** Why are UDFs slower in PySpark? ✓ **Q2:** What is the difference between UDF and Pandas UDF? ✓ **Q3:** How can you optimize UDF performance? ✓ **Q4:** Can we use UDFs in SQL queries? (Yes, by registering them). ✓ **Q5:** Why prefer `pyspark.sql.functions` over UDF? (They run natively on Spark engine).

👉 Do you want me to also cover **User Defined Aggregate Functions (UDAF)** and **when to use UDF vs built-in functions**? Or should I prepare **next topics like `explode()` and `posexplode()` with real-time use cases**?

```
Start coding or generate with AI.
```

Here's a **complete guide on PySpark DataFrame transformations**, structured for interview prep and practical use:

✅ 1. What is a DataFrame Transformation?

- A **transformation** in PySpark is an **operation on a DataFrame that returns a new DataFrame**.
- Transformations are **lazy**, meaning Spark **does not execute them immediately**.
- Execution happens only when an **action** (like `show()`, `collect()`, `count()`) is called.

Categories of Transformations

1. **Narrow Transformations** – data from one partition is needed for one output partition (e.g., `select`, `filter`, `withColumn`).
2. **Wide Transformations** – data from multiple partitions is needed for output (e.g., `groupBy`, `join`, `distinct`, `repartition`).

✅ 2. Common PySpark DataFrame Transformations

---

a) `select()`

- Select specific columns or apply expressions.

```
from pyspark.sql.functions import col

df.select("name", col("salary") * 2).show()
```

---

b) `filter()` / `where()`

- Filter rows based on conditions.

```
df.filter(col("age") > 30).show()
```

---

c) `withColumn()`

- Add a new column or update existing column.

```
from pyspark.sql.functions import lit

df.withColumn("bonus", col("salary") * 0.10)\
  .withColumn("country", lit("India")).show()
```

---

d) `withColumnRenamed()`

- Rename column(s)

```
df.withColumnRenamed("salary", "emp_salary").show()
```

---

e) `drop()`

- Drop column(s)

```
df.drop("bonus").show()
```

---

f) `distinct()` / `dropDuplicates()`

- Remove duplicate rows.

```
df.distinct().show()
df.dropDuplicates(["name", "department"]).show()
```

---

g) `groupBy()` and Aggregations

- Group data and aggregate.

```
from pyspark.sql.functions import avg, sum

df.groupBy("department").agg(
    avg("salary").alias("avg_salary"),
    sum("salary").alias("total_salary")
).show()
```

---

h) `orderBy()` / `sort()`

- Sort data by column(s)

```
df.orderBy(col("salary").desc()).show()
```

---

i) `union()`

- Combine two DataFrames (append rows)

```
df3 = df1.union(df2)
```

---

j) `join()`

- Combine DataFrames based on keys

```
df1.join(df2, df1.id == df2.id, "inner").show()
```

---

k) `pivot()` / `unpivot()`

- Pivot: Rows → Columns

```
df.groupBy("dept").pivot("year").sum("salary").show()
```

- Unpivot: Columns → Rows (`stack()` or `explode()`)

---

l) `explode()` / `posexplode()`

- Convert array or map column into multiple rows.

```
from pyspark.sql.functions import explode
```

```
df.select("id", explode("skills")).show()
```

---

m) `alias()`

- Rename column temporarily

```
df.select(col("salary").alias("emp_salary")).show()
```

---

n) `replace()`

- Replace values in a DataFrame

```
df.replace("HR", "Human Resources", "department").show()
```

---

o) `fillna()` / `na.fill()`

- Fill NULLs

```
df.fillna({"salary": 0, "department": "Unknown"}).show()
```

---

p) `cast()`

- Change column type

```
df.withColumn("salary", col("salary").cast("double")).show()
```

---

q) `limit()` / `sample()`

- Limit rows or take sample

```
df.limit(5).show()
df.sample(0.1).show()  # 10% sample
```

---

r) `repartition()` / `coalesce()`

- Change number of partitions (wide transformation)

```
df.repartition(10)  # increase partitions
df.coalesce(2)      # reduce partitions
```

✅ 3. Transformations vs Actions

Feature	Transformation	Action
Execution	Lazy	Trigger computation
Output	New DataFrame	Returns Python/Value
Examples	select, filter, withColumn	show, collect, count

✅ 4. Best Practices

- Prefer **built-in functions** over UDFs for performance.
- Chain transformations **before actions** for efficient DAG optimization.
- Use **`cache()` or `persist()`** when reusing DataFrames.
- Use **explicit schema** for large datasets to avoid `inferSchema` overhead.

🔍 Interview Questions

1. Difference between **narrow vs wide transformations**?
2. Why are transformations **lazy** in Spark?
3. When would you use **`repartition()` vs `coalesce()`**?
4. Difference between **`dropDuplicates()` and `distinct()`**?
5. How does **`withColumn()` affect performance**?

If you want, I can prepare a **complete PySpark transformations cheat sheet** with **all functions, syntax, and examples** in **one single table** for fast revision.

Do you want me to do that?

```
Start coding or generate with AI.
```

Here's a **comprehensive guide** on **`partitionBy`**, **`explode`**, and **`cache vs persist`** in PySpark:

✅ 1. `partitionBy` in PySpark

Definition

- `partitionBy` is used when **writing DataFrames to storage** (like Parquet, CSV, ORC) to **physically partition data by one or more columns**.
- Helps in **faster querying** because Spark can **read only relevant partitions** instead of scanning the entire dataset.

Syntax

```
df.write.partitionBy("col1", "col2").format("parquet").save("path/to/output")
```

Example

```
data = [(1, "Alice", "HR"), (2, "Bob", "IT"), (3, "Cathy", "HR")]
columns = ["id", "name", "department"]

df = spark.createDataFrame(data, columns)

df.write.partitionBy("department").parquet("output/path")
```

- This creates **separate folders** for each department:

```
output/path/department=HR/
output/path/department=IT/
```

Key Points

- Only works when **writing data** (`write` operation), not during transformations.
- Improves **read performance** for **filter-heavy queries**.
- Can combine with **`bucketBy`** for more optimization.

🔍 Interview Question

Q: Difference between `partitionBy` and `repartition()`? ✓ `partitionBy` → Physical storage partitioning when writing files. ✓ `repartition()` → Logical partitioning in memory during transformations.

✅ 2. `explode` in PySpark

Definition

- `explode()` transforms a **column containing arrays or maps** into **multiple rows**.
- Each element of the array/map becomes a **separate row**, duplicating other columns.

Syntax

```
from pyspark.sql.functions import explode

df.select("id", explode("array_column").alias("element")).show()
```

Examples

a) Explode Array

```
data = [(1, ["Java", "Python"]), (2, ["Scala", "SQL"])]
df = spark.createDataFrame(data, ["id", "skills"])

from pyspark.sql.functions import explode
df.select("id", explode("skills").alias("skill")).show()
```

Output

id	skill
1	Java
1	Python
2	Scala
2	SQL

---

b) Explode Map

```
data = [(1, {"a": 10, "b": 20})]
df = spark.createDataFrame(data, ["id", "scores"])

df.select("id", explode("scores")).show()
```

Output

id	key	value
1	a	10
1	b	20

Key Points

- Often used in **JSON parsing**, arrays, or nested data.
- Returns a **long format** of the data.
- `posexplode()` → also returns **position of element** in the array.

🔍 Interview Question

Q: Difference between `explode` and `posexplode`? ✓ `explode` → Only elements, `posexplode` → Elements + their index/position.

✅ 3. `cache()` vs `persist()` in PySpark

Definition

- Both are used to **store intermediate DataFrames/RDDs in memory/disk** for **reuse**, avoiding recomputation.

Syntax

```
df.cache()          # Cache in memory
df.persist()        # Persist with storage level
```

Details

Feature	cache()	persist()
Storage	Memory only (default)	Memory, Disk, Memory+Disk, Off-heap
Flexibility	No storage level selection	Can select storage level
Usage	Simple caching for reuse	For advanced caching requirements
Return Type	Same DataFrame	Same DataFrame

Example

```
from pyspark import StorageLevel

# Cache in memory
df.cache()

# Persist in memory and disk
df.persist(StorageLevel.MEMORY_AND_DISK)
```

```
# Trigger computation
df.count()
```

Key Points

- `cache()` is shorthand for `persist(StorageLevel.MEMORY_ONLY)`.
- Always call an **action** (`show()`, `count()`) to trigger caching.
- Useful for **iterative algorithms** like MLlib.

🔍 Interview Questions

1. When would you use `persist(StorageLevel.DISK_ONLY)`? ✓ For very large DataFrames that **don't fit in memory**.
2. Does caching guarantee data will stay in memory forever? ✓ No, Spark may **evict cached data** under memory pressure.
3. Difference between `cache()` and `persist()`? ✓ `cache()` → memory only, `persist()` → customizable storage level.

If you want, I can **combine this with all previous PySpark functions** into a **complete cheat sheet** including:

- Transformations, Actions, Window Functions, Joins, Pivot/Unpivot, Date Functions, Caching, Partitioning, and Exploding – **everything in one page for interviews**.

Do you want me to prepare that?

```
Start coding or generate with AI.
```

Perfect! Here's a **complete PySpark interview checklist** — structured, concise, and with **all essential topics, functions, syntax, and examples** in one sheet for quick revision.

PySpark Interview Checklist – Complete Guide

---

1. PySpark Basics

Topic	Syntax / Function	Example
PySpark Definition	`from pyspark.sql import SparkSession`	`spark = SparkSession.builder.appName("App").getO`
DataFrame Creation	`createDataFrame()`	`df = spark.createDataFrame([(1,"Alice")], ["id",`
Schema Definition	`StructType, StructField`	`schema = StructType([StructField("id", IntegerTy`
MapReduce vs Spark	N/A	Spark is **faster**, in-memory, DAG optimized

2. DataFrame Transformations (Lazy)

Transformation	Syntax	Example
select	df.select("col")	df.select("name", col("salary")*2)
filter / where	df.filter(condition)	df.filter(col("age")>30)
withColumn	df.withColumn("new_col", expr)	df.withColumn("bonus", col("salary"
withColumnRenamed	df.withColumnRenamed("old","new")	df.withColumnRenamed("salary","emp_
drop	df.drop("col")	df.drop("bonus")
distinct / dropDuplicates	df.distinct()	df.dropDuplicates(["name"])
groupBy + agg	df.groupBy("col").agg(func)	df.groupBy("dept").agg(avg("salary"
orderBy / sort	df.orderBy(col("salary").desc())	Sort rows
union	df1.union(df2)	Combines rows, schemas must match
join	df1.join(df2, on, how)	inner, left, right, outer, semi, ant
pivot	df.groupBy("col").pivot("col2").agg(func)	Transform rows → columns
unpivot	selectExpr("stack(...)")	Columns → rows
explode / posexplode	explode(array_col)	Array/Map → multiple rows
limit / sample	df.limit(n), df.sample(fraction)	Subset of rows
repartition / coalesce	df.repartition(n), df.coalesce(n)	Change number of partitions

3. UDFs (User Defined Functions)

Type	Syntax	Example
Regular UDF	udf(func, returnType)	upper_udf = udf(lambda x: x.upper(), Strin
Pandas UDF	@pandas_udf(returnType)	Vectorized operation using Pandas
SQL UDF	spark.udf.register("func_name", func, returnType)	Used in spark.sql()
UDAF	Custom aggregate function	Advanced, row/group aggregation

4. Views & SQL Queries

View Type	Syntax	Notes
Temporary View	df.createOrReplaceTempView("name")	Session-scoped
Global Temp View	df.createGlobalTempView("name")	Accessible across sessions using global_temp.name
SQL Query	spark.sql("SELECT * FROM view")	Execute SQL on DataFrames

5. Window Functions

Function	Syntax	Example
row_number	row_number().over(windowSpec)	Partition + order
rank / dense_rank	rank().over(windowSpec)	Gaps in rank / no gaps
lag / lead	lag(col,n).over(windowSpec)	Previous/next row

Function	Syntax	Example
sum / avg	`sum(col).over(windowSpec)`	Running totals
Define Window	`Window.partitionBy("col").orderBy("col").rowsBetween(start,end)`	Controls scope of calculation

6. Date/Time Functions

Function	Syntax	
current_date / timestamp	`current_date()`, `current_timestamp()`	Current sys
to_date / to_timestamp	`to_date(col, format)`	Convert stri
date_format	`date_format(col, "format")`	Format date
year / month / day	`year(col)`, `month(col)`, `dayofmonth(col)`	Extract part
Date arithmetic	`datediff(end,start)`, `add_months(col,n)`, `date_add(col,n)`, `date_sub(col,n)`	Calculate d

7. File Handling / Partitioning

Function	Syntax	Example	Notes
partitionBy	`df.write.partitionBy("col").parquet(path)`	Writes separate folders	Physical storage optimizatio
bucketBy	`df.write.bucketBy(n,"col").saveAsTable("table")`	Optimized joins	Works with Hive tables
Compression	`.option("compression","snappy")`	Compress files	Faster read/write

8. Performance / Optimization

Concept	Syntax	Notes
cache	`df.cache()`	Memory only, lazy, triggers on action
persist	`df.persist(StorageLevel.MEMORY_AND_DISK)`	Custom storage level
narrow vs wide transformations	N/A	Narrow → single partition, Wide → shuffle r
broadcast join	`broadcast(df2)`	Small table joins faster
shuffle partitions	`spark.sql.shuffle.partitions`	Tune for performance
checkpointing	`df.checkpoint()`	Fault tolerance for iterative jobs

9. RDD Basics (Optional but Interview Edge)

Transformation / Action	Syntax	Notes
map / flatMap	`rdd.map(f)`	Element-wise transformation
filter	`rdd.filter(f)`	Row filtering
reduce	`rdd.reduce(f)`	Aggregation across RDD
groupByKey / reduceByKey	`rdd.groupByKey()`	Key-based aggregation
sortByKey	`rdd.sortByKey()`	Sort key-value RDD
collect / take / count	Actions	Bring data to driver / sample / count

10. Advanced Topics

Topic	Notes
Structured Streaming	Real-time processing
MLlib	Transformers, Estimators, Pipelines
GraphFrames / GraphX	Graph analysis
Spark Config Tuning	Memory, shuffle, cores, partitions

---

11. Key Interview Tips

- Prefer **built-in functions** over UDFs.
- Always define **explicit schema** for large datasets.
- Understand **lazy evaluation** and **DAG optimization**.
- Know **narrow vs wide transformations**.
- Be able to explain **caching/persisting and memory optimization**.
- Practice **SQL + DataFrame API queries**.

This sheet **covers 95–100% of PySpark interview topics**, with **functions, syntax, and examples** ready for fast revision.

If you want, I can **also prepare a "one-page visual diagram"** of PySpark concepts showing **Transformations → Actions → Optimizations → Storage → Window → Dates → Joins →**