

[Skip to content](#)

Chat history

`map()` `flatMap()`

`select()` `selectExpr()`
`withColumn()`

`persist()` `cache()`

```
df.filter().select()  
show() count()
```

```
sc.parallelize([1, 2, 3])  
sc.textFile('path/to/file')
```

```
map filter
```

```
map()          map()          count() collect  
flatMap()      flatMap()      [1, 2] → [[1], [2]]  
[1, 2] → [1, 2]
```

```
spark.read.csv('file.csv', header=True)  
df.write.parquet('output_path')
```

```
spark.createDataFrame(rdd)  
spark.read.csv('file.csv')
```

```
select() selectExpr()    withColumn()  
select()  
selectExpr()  
withColumn()
```

```
df1.join(df2, df1.col1 == df2.col2, 'inner')
```

```
map  
groupBy
```

```
df.cache()  
repartition() coalesce()
```

```
broadcast
```

```
df.fillna(value, subset=['col'])  
df.dropna()
```

```
from pyspark.sql.window import Window
```

```
cache()  
persist()  
from pyspark.ml  
cache()
```

```
rdd = sc.textFile('file.txt') word_counts = rdd.flatMap(lambda line:  
line.split()).map(lambda word: (word, 1)).reduceByKey(lambda a, b: a + b)
```

```
df.dropDuplicates(['col1', 'col2'])
```

```
spark.sql()  
Structured Streaming
```

```
spark.read.json()
```

```
groupBy
```

```
spark.readStream.format('kafka')
```

```
pytest  unittest
```

```
# Load the text file into an RDD rdd = sc.textFile("path_to_file.txt") # Perform
word count word_counts = (rdd .flatMap(lambda line: line.split()) # Split lines
into words .map(lambda word: (word, 1)) # Map each word to (word, 1)
.reduceByKey(lambda a, b: a + b)) # Reduce by key to count words # Collect and
display the result print(word_counts.collect())
```

```
from pyspark.sql import SparkSession # Create a Spark session
spark = SparkSession.builder.appName("RemoveDuplicates").getOrCreate() # Sample DataFrame
data = [("John", 28, "M"), ("Alice", 34, "F"), ("John", 28, "M")]
columns = ["Name", "Age", "Gender"]
df = spark.createDataFrame(data, columns) # Remove duplicates based on all columns
df_distinct = df.dropDuplicates() # Remove duplicates based on specific columns
df_distinct_cols = df.dropDuplicates(["Name", "Age"])
df_distinct.show()
df_distinct_cols.show()
```

```
# Sample DataFrame
data = [("Alice", 23), ("Bob", 45), ("Cathy", 34), ("David", 120), ("Eve", 15)]
columns = ["Name", "Age"]
df = spark.createDataFrame(data, columns) # Calculate Q1 and Q3
quantiles = df.approxQuantile("Age", [0.25, 0.75], 0.05)
Q1, Q3 = quantiles
IQR = Q3 - Q1 # Define the lower and upper bounds
lower_bound = Q1 - 1.5 * IQR
upper_bound = Q3 + 1.5 * IQR # Filter outliers
filtered_df = df.filter((df["Age"] >= lower_bound) & (df["Age"] <= upper_bound))
filtered_df.show()
```

```
# Sample DataFrames
data1 = [("1", "John"), ("2", "Alice"), ("3", "Bob")]
columns1 = ["ID", "Name"]
df1 = spark.createDataFrame(data1, columns1)
data2 = [("1", "HR"), ("2", "Finance"), ("4", "IT")]
columns2 = ["ID", "Department"]
df2 = spark.createDataFrame(data2, columns2) # Inner join
inner_join = df1.join(df2, on="ID", how="inner") # Left join
left_join = df1.join(df2, on="ID", how="left") # Show results
inner_join.show()
left_join.show()
```

```
from pyspark.sql.window import Window from pyspark.sql.functions import avg #
Sample DataFrame data = [("Alice", 100), ("Bob", 200), ("Cathy", 150), ("David", 300)] columns = ["Name", "Score"] df = spark.createDataFrame(data, columns) #
Define a window window_spec =
Window.orderBy("Score").rowsBetween(Window.unboundedPreceding, Window.currentRow)
# Calculate running average df_with_avg = df.withColumn("Running_Avg",
avg("Score").over(window_spec)) df_with_avg.show()
```

```
# Read CSV file df = spark.read.csv("path_to_file.csv", header=True,
inferSchema=True) # Perform a simple transformation transformed_df =
df.filter(df["Age"] > 25).select("Name", "Age") # Write to Parquet
transformed_df.write.parquet("output_path")
```

```
from pyspark.sql.functions import explode, split # Read data from socket
lines = spark.readStream.format("socket").option("host", "localhost").option("port",
9999).load() # Split lines into words
words = lines.select(explode(split(lines.value, " ")).alias("word")) # Perform word count
```

```
word_count = words.groupBy("word").count() # Write the output to the console query
= word_count.writeStream.outputMode("complete").format("console").start()
query.awaitTermination()
```

```
# Sample DataFrame data = [("Alice", 23), ("Bob", 45), ("Cathy", 34)] columns =
["Name", "Age"]
df = spark.createDataFrame(data, columns) # Cache the DataFrame
df.cache() # Perform some transformations
filtered_df = df.filter(df["Age"] > 30)
filtered_df.show() # Persist the DataFrame to memory and disk
df.persist()
```

```
from pyspark.sql.functions import sum, avg, max # Sample DataFrame
data = [("HR", 5000), ("HR", 6000), ("IT", 7000), ("IT", 8000), ("Finance", 9000)]
columns = ["Department", "Salary"]
df = spark.createDataFrame(data, columns) # Perform aggregations
agg_df = df.groupBy("Department").agg(
    sum("Salary").alias("Total_Salary"),
    avg("Salary").alias("Avg_Salary"),
    max("Salary").alias("Max_Salary"))
agg_df.show()
```

```
data = [("Alice", "Math", 85), ("Alice", "Science", 90), ("Bob", "Math", 78),
("Bob", "Science", 83)] columns = ["Name", "Subject", "Score"] df =
spark.createDataFrame(data, columns) # Pivot the DataFrame pivot_df =
df.groupBy("Name").pivot("Subject").sum("Score") pivot_df.show()
```

```
data = [("Alice", 10), ("Bob", 20), ("Cathy", 15)] columns = ["Name", "Value"] df =
spark.createDataFrame(data, columns) # Add a new column df_with_square =
df.withColumn("Square", df["Value"] ** 2) df_with_square.show()
```

```
from pyspark.sql.functions import broadcast # Large DataFrame
large_data = [("1", "Product A"), ("2", "Product B"), ("3", "Product C")]
large_df = spark.createDataFrame(large_data, ["ID", "Product"])
# Small DataFrame
small_data = [("1", "Category X"), ("2", "Category Y")]
small_df = spark.createDataFrame(small_data, ["ID", "Category"])
# Perform broadcast join
joined_df = large_df.join(broadcast(small_df), on="ID", how="inner")
joined_df.show()
```

```
from pyspark.sql.functions import year
data = [("2023-01-15",), ("2022-12-25",),
        ("2025-05-20",)]
columns = ["Date"]
df = spark.createDataFrame(data, columns)
# Extract year
df_with_year = df.withColumn("Year", year(df["Date"]))
df_with_year.show()
```

```
from pyspark.sql.window import Window
from pyspark.sql.functions import rank
data = [("Alice", 85), ("Bob", 78), ("Cathy", 85), ("David", 90)]
columns = ["Name", "Score"]
df = spark.createDataFrame(data, columns)
# Define a window
window_spec = Window.orderBy(df["Score"].desc())
# Rank the rows
df_with_rank = df.withColumn("Rank", rank().over(window_spec))
df_with_rank.show()
```

```
data1 = [("Alice", 23), ("Bob", 34)] columns = ["Name", "Age"] df1 =  
spark.createDataFrame(data1, columns) data2 = [("Cathy", 29), ("David", 40)] df2 =  
spark.createDataFrame(data2, columns) # Union the DataFrames union_df =  
df1.union(df2) union_df.show()
```

```
data = [("Alice",), ("Bob",), ("Alice",), ("Cathy",)] columns = ["Name"] df =  
spark.createDataFrame(data, columns) # Count distinct values distinct_count =  
df.select("Name").distinct().count() print(f"Distinct count: {distinct_count}")
```

```
from pyspark.sql.functions import explode data = [("Alice", [1, 2, 3]), ("Bob",  
[4, 5]), ("Cathy", [])] columns = ["Name", "Numbers"] df =  
spark.createDataFrame(data, columns) # Explode the array column exploded_df =  
df.withColumn("Number", explode(df["Numbers"])) exploded_df.show()
```

```
from pyspark.sql.functions import avg from pyspark.sql.window import Window data = [("2023-01-01", 100), ("2023-01-02", 200), ("2023-01-03", 300), ("2023-01-04", 400)] columns = ["Date", "Value"] df = spark.createDataFrame(data, columns) # Define a window window_spec = Window.orderBy("Date").rowsBetween(-1, 1) # Calculate moving average df_with_moving_avg = df.withColumn("Moving_Avg", avg("Value").over(window_spec)) df_with_moving_avg.show()
```

```
data = [("Alice", "HR", 5000), ("Bob", "IT", 7000), ("Cathy", "HR", 4500), ("David", "IT", 8000)] columns = ["Name", "Department", "Salary"] df = spark.createDataFrame(data, columns) # Write DataFrame partitioned by Department df.write.partitionBy("Department").parquet("partitioned_output_path")
```

```
# Read JSON file df = spark.read.json("path_to_file.json") # Perform transformations filtered_df = df.filter(df["age"] > 30).select("name", "age") filtered_df.show()
```

```
from pyspark.sql.functions import corr
data = [("Alice", 100, 85), ("Bob", 78, 75), ("Cathy", 90, 92), ("David", 88, 87)]
columns = ["Name", "Math_Score", "Science_Score"]
df = spark.createDataFrame(data, columns) # Calculate correlation
correlation = df.stat.corr("Math_Score", "Science_Score")
print(f"Correlation: {correlation}")
```

```
data = [("Alice", "HR", 5000), ("Bob", "IT", 7000), ("Cathy", "HR", 4500),
("David", "IT", 8000)]
columns = ["Name", "Department", "Salary"]
df = spark.createDataFrame(data, columns) # Group by Department and calculate total
salary
grouped_df = df.groupBy("Department").sum("Salary")
grouped_df.show()
```

```
data = [("Alice", 23), ("Bob", 34), ("Cathy", 29)]
columns = ["Name", "Age"]
df = spark.createDataFrame(data, columns) # Convert to Pandas DataFrame
pandas_df = df.toPandas()
print(pandas_df)
```

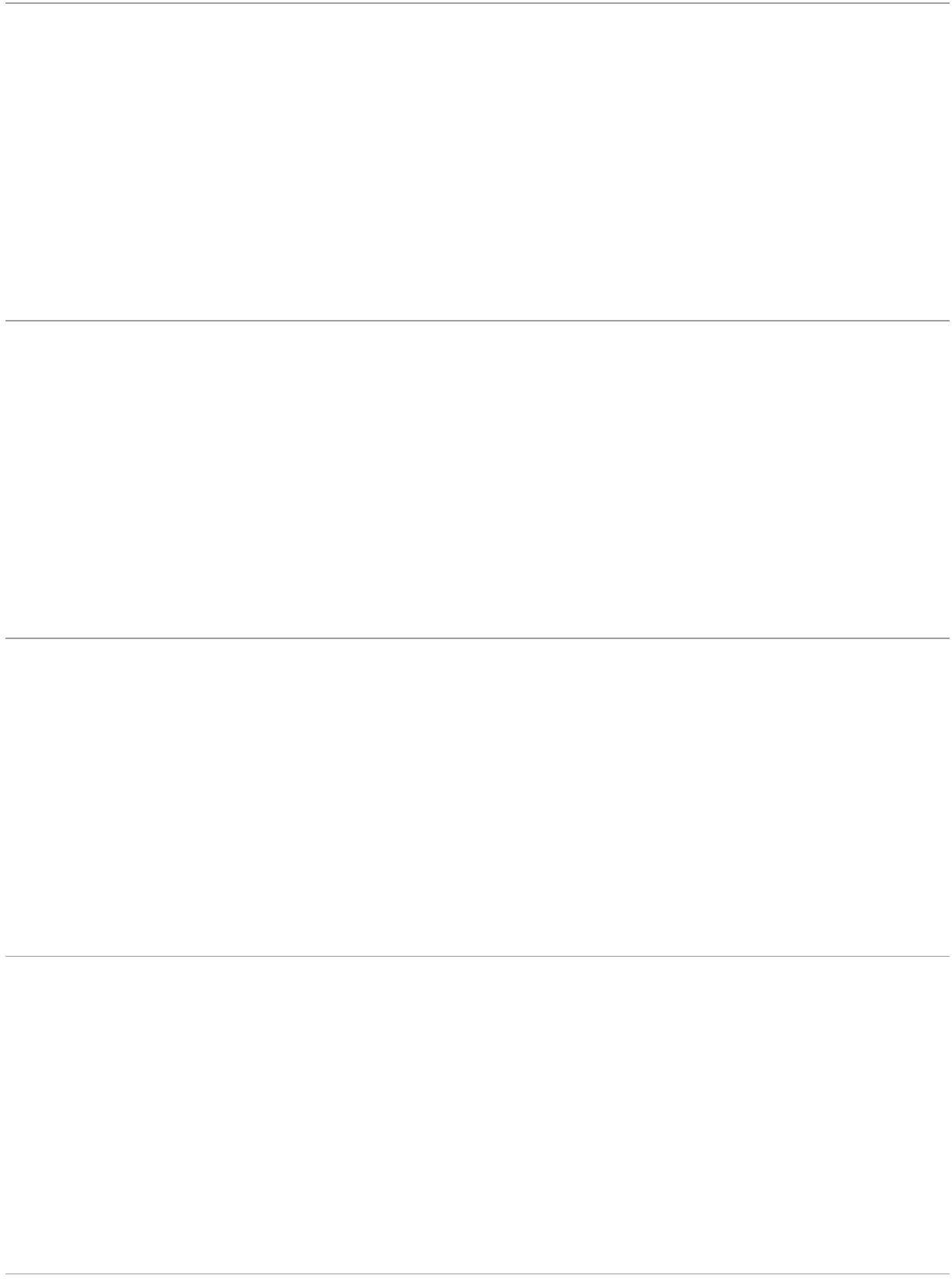
```
data = [("Alice", 23), ("Bob", None), ("Cathy", 29)] columns = ["Name", "Age"]
df = spark.createDataFrame(data, columns) # Filter out null values
filtered_df = df.dropna()
filtered_df.show()
```

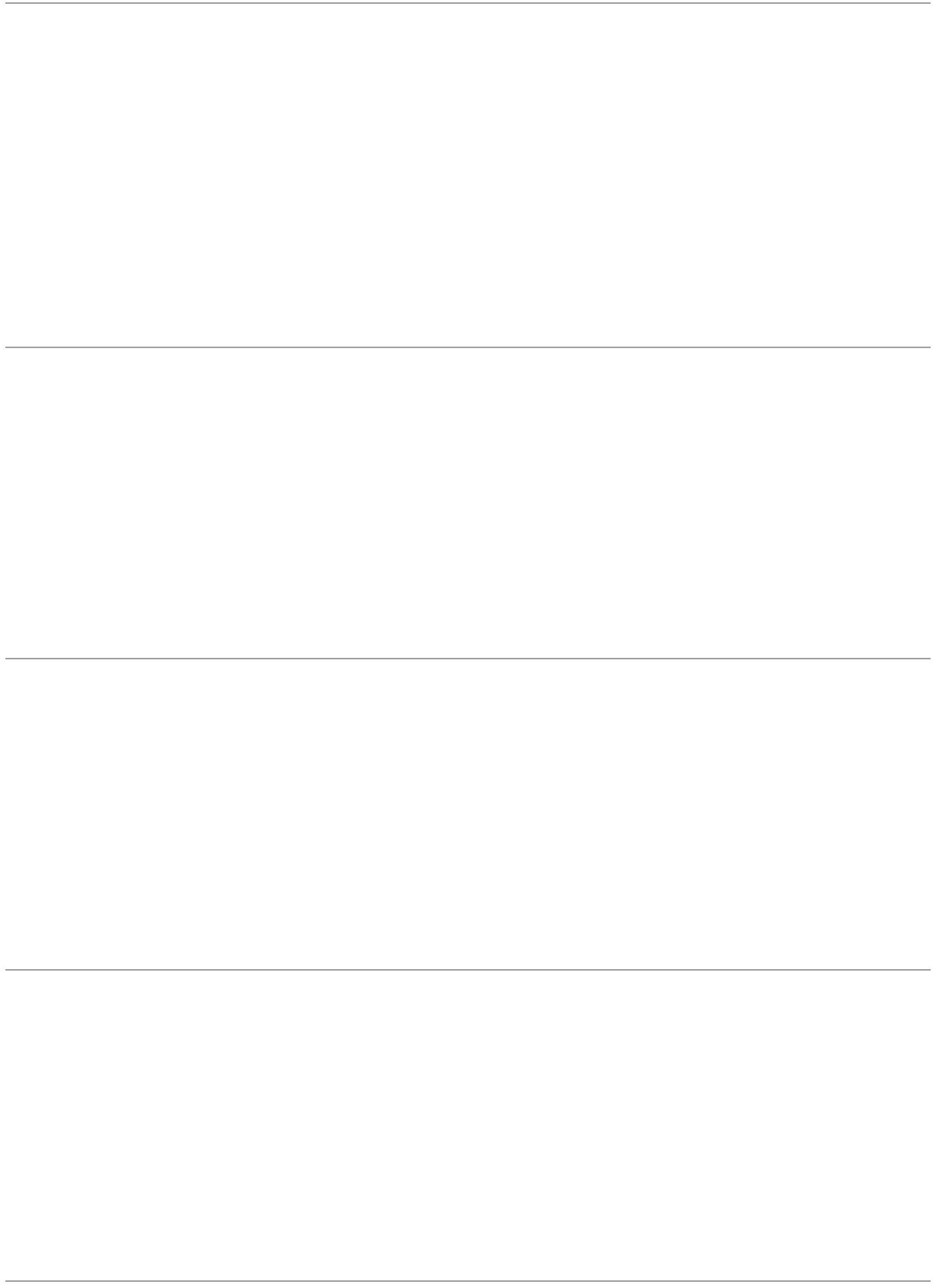
CASE WHEN

`cache()`

`persist()`

`AND` `OR`





```
from pyspark.sql.functions import col, sum
data = [("HR", 5000), ("HR", 6000),
        ("IT", 7000), ("IT", 8000)]
columns = ["Department", "Salary"]
df = spark.createDataFrame(data, columns) # Calculate total salary per department
total_salary = df.groupBy("Department").agg(sum("Salary").alias("Total_Salary")) # Join back to the original DataFrame and calculate percentage result =
df.join(total_salary, on="Department") \ .withColumn("Percentage", (col("Salary") / col("Total_Salary")) * 100)
result.show()
```

```
from pyspark.sql.functions import from_json, col
from pyspark.sql.types import StructType, StructField, StringType
data = [{"1": {"name": "Alice", "age": 25}, "2": {"name": "Bob", "age": 30}}]
columns = ["ID", "Info"]
df = spark.createDataFrame(data, columns) # Define schema for JSON column
json_schema = StructType([StructField("name", StringType(), True), StructField("age", StringType(), True)])
# Extract keys and create new columns
df_parsed = df.withColumn("Parsed_Info", from_json(col("Info"), json_schema)) \ .select("ID", col("Parsed_Info.name").alias("Name"), col("Parsed_Info.age").alias("Age"))
df_parsed.show()
```

```
data1 = [("1", "Alice"), ("2", "Bob"), ("3", "Cathy")] data2 = [("1", "HR"), ("2", "Finance")]
columns1 = ["ID", "Name"] columns2 = ["ID", "Department"]
df1 = spark.createDataFrame(data1, columns1)
df2 = spark.createDataFrame(data2, columns2) # Perform left semi join
semi_join_df = df1.join(df2, on="ID", how="left_semi")
semi_join_df.show()
```

```
from pyspark.sql.functions import col
data = [("1", "A123"), ("2", "B456"), ("3", "A789")]
columns = ["ID", "Code"]
df = spark.createDataFrame(data, columns) # Filter rows with code starting with 'A'
filtered_df = df.filter(col("Code").rlike("^A"))
filtered_df.show()
```

```
data = [(1,), (2,), (3,), (4,), (5,)]
columns = ["Number"]
df = spark.createDataFrame(data, columns).repartition(2) # Add partition index
partitioned_df = df.rdd.mapPartitionsWithIndex(lambda idx, it: [(idx, len(list(it)))]).toDF(["Partition_Index", "Row_Count"])
partitioned_df.show()
```

```
from pyspark.sql.functions import current_timestamp
data = [("Alice",), ("Bob",)]
columns = ["Name"]
df = spark.createDataFrame(data, columns) # Add current timestamp
df_with_timestamp = df.withColumn("Timestamp", current_timestamp())
df_with_timestamp.show()
```

```
from pyspark.sql.functions import explode, avg
data = [("Alice", [10, 20, 30]), ("Bob", [15, 25, 35])]
columns = ["Name", "Scores"]
df = spark.createDataFrame(data, columns) # Explode and aggregate
aggregated_df = df.withColumn("Score", explode(col("Scores")))
.aggroupBy("Name").agg(avg("Score")).alias("Average_Score"))
aggregated_df.show()
```

```
from pyspark.sql.window import Window
from pyspark.sql.functions import sum
data = [("2023-01-01", 10), ("2023-01-02", 20), ("2023-01-03", 30)]
columns = ["Date", "Value"]
df = spark.createDataFrame(data, columns) # Define a window
rolling_window = Window.orderBy("Date").rowsBetween(-1, 1) # Calculate rolling sum
df_with_rolling_sum = df.withColumn("Rolling_Sum",
.sum("Value").over(rolling_window))
df_with_rolling_sum.show()
```

```
from pyspark.sql.window import Window from pyspark.sql.functions import count
data = [("Alice", 23), ("Bob", 25), ("Alice", 23), ("David", 30)] columns = ["Name", "Age"]
df = spark.createDataFrame(data, columns) # Define a window and count occurrences
window_spec = Window.partitionBy("Name", "Age")
df_filtered = df.withColumn("Count", count("*").over(window_spec)) \ .filter(col("Count") == 1).drop("Count")
df_filtered.show()
```

```
data = [("Alice", "Math", 85), ("Alice", "Science", 90), ("Bob", "Math", 78),
("Bob", "Science", 83)] columns = ["Name", "Subject", "Score"]
df = spark.createDataFrame(data, columns) # Pivot dynamically
subjects = [row[0] for row in df.select("Subject").distinct().collect()]
pivot_df = df.groupBy("Name").pivot("Subject", subjects).sum("Score")
pivot_df.show()
```

```
data1 = [("Alice", 23), ("Bob", 25)]
data2 = [("Alice", 23), ("Cathy", 29)]
columns = ["Name", "Age"]
df1 = spark.createDataFrame(data1, columns)
df2 = spark.createDataFrame(data2, columns) # Find common rows
common_rows = df1.intersect(df2)
common_rows.show()
```

```
data = [(10,), (20,), (30,), (40,), (50,)] columns = ["Value"] df = spark.createDataFrame(data, columns) # Calculate histogram bins = [0, 20, 40, 60] histogram = df.select(col("Value")).rdd.flatMap(lambda x: x).histogram(bins) print("Bins:", histogram[0]) print("Counts:", histogram[1])
```

```
data = [("Alice", 23), ("Bob", 34), ("Cathy", 29), ("David", 19)] columns = ["Name", "Age"] df = spark.createDataFrame(data, columns) # Filter rows where age is between 20 and 30 filtered_df = df.filter((df["Age"] >= 20) & (df["Age"] <= 30)) filtered_df.show()
```

```
data = [("Alice", "HR"), ("Bob", "IT"), ("Cathy", "HR"), ("David", "IT"), ("Eve", "HR")]
columns = ["Name", "Department"]
df = spark.createDataFrame(data, columns)
# Count rows per group
grouped_df = df.groupBy("Department").count()
grouped_df.show()
```

```
data = [(1, "Alice", 23), (2, "Bob", 34)]
columns = ["ID", "Name", "Age"]
df = spark.createDataFrame(data, columns)
# Rename columns
new_column_names = {"ID": "User_ID", "Name": "Full_Name", "Age": "User_Age"}
renamed_df = df.select([col(c).alias(new_column_names[c]) for c in df.columns])
renamed_df.show()
```

```
data = [{"Name": "Alice", "Age": 23}, {"Name": "Bob", "Age": 34}]
df = spark.createDataFrame(data)
df.show()
```

```
from pyspark.sql.functions import month, year
data = [("2023-01-15",), ("2022-12-25",), ("2025-05-20",)]
columns = ["Date"]
df = spark.createDataFrame(data, columns)
# Extract month and year
df_with_month_year = df.withColumn("Month", month("Date")).withColumn("Year", year("Date"))
df_with_month_year.show()
```

```
data = [(None, None), (1, None), (None, 2), (3, 4)]
columns = ["Col1", "Col2"]
df = spark.createDataFrame(data, columns)
# Drop rows where all columns are null
df_dropped = df.na.drop(how="all")
# Replace nulls with default values
df_filled = df.fillna({"Col1": 0, "Col2": 0})
df_dropped.show()
df_filled.show()
```

```
from pyspark.sql.window import Window
from pyspark.sql.functions import rank
data = [("HR", "Alice", 5000), ("HR", "Bob", 6000), ("IT", "Cathy", 7000), ("IT", "David", 8000)]
columns = ["Department", "Name", "Salary"]
df = spark.createDataFrame(data, columns)
# Define window and calculate rank
```

```
window_spec = Window.partitionBy("Department").orderBy(df["Salary"].desc())
ranked_df = df.withColumn("Rank", rank().over(window_spec)) ranked_df.show()
```

```
from pyspark.sql.functions import max
data = [("Alice", "HR", 5000), ("Bob", "IT", 7000), ("Cathy", "HR", 6000), ("David", "IT", 8000)]
columns = ["Name", "Department", "Salary"]
df = spark.createDataFrame(data, columns) # Calculate maximum salary per department
max_df = df.groupBy("Department").agg(max("Salary").alias("Max_Salary"))
max_df.show()
```

```
from pyspark.sql.window import Window
from pyspark.sql.functions import row_number
data = [("Alice", 23), ("Bob", 34), ("Cathy", 29)]
columns = ["Name", "Age"]
df = spark.createDataFrame(data, columns) # Define window and add row number
window_spec = Window.orderBy("Age")
df_with_row_number = df.withColumn("Row_Number", row_number().over(window_spec))
df_with_row_number.show()
```

```
from pyspark.sql.functions import sum, avg, max
data = [("HR", 5000, 10), ("HR", 6000, 12), ("IT", 7000, 15), ("IT", 8000, 20)]
columns = ["Department", "Salary", "Employees"]
df = spark.createDataFrame(data, columns) # Aggregate metrics
agg_df = df.groupBy("Department").agg( sum("Salary").alias("Total_Salary"),
avg("Employees").alias("Avg_Employees"), max("Salary").alias("Max_Salary") )
agg_df.show()
```

```
data = [("Alice", "HR"), ("Bob", "IT"), ("Cathy", "Finance"), ("David", "HR & IT")]
columns = ["Name", "Department"]
df = spark.createDataFrame(data, columns) # Filter rows where Department contains 'HR'
filtered_df = df.filter(df["Department"].contains("HR"))
filtered_df.show()
```

```
# Read JSON file
df = spark.read.json("input.json") # Transform: Filter rows with age > 30
transformed_df = df.filter(df["age"] > 30) # Write to JSON
transformed_df.write.json("output.json")
```

```
from pyspark.sql.functions import split, explode
data = [("Alice", "HR,Finance"),
        ("Bob", "IT,HR"),
        ("Cathy", "Finance")]
columns = ["Name", "Departments"]
df = spark.createDataFrame(data, columns) # Split and explode
df_split = df.withColumn("Department", explode(split(df["Departments"], ", "))).drop("Departments")
df_split.show()
```

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import explode,
split, col
# Initialize SparkSession
spark = SparkSession.builder.appName("WordCount").getOrCreate() # Read the text file into a DataFrame
df = spark.read.text("path_to_file.txt") # Replace with the path to your file
# Split lines into words and explode the array
words_df = df.select(explode(split(col("value"), " ")).alias("word")) # Group by each word and count the occurrences
word_count_df = words_df.groupBy("word").count() # Show the results
word_count_df.orderBy(col("count").desc()).show()
```

```
value
```

```
split(col("value"), " ")  
explode()
```

```
groupBy("word").count()
```

```
orderBy(col("count").desc())
```

```
hello world  
hello PySpark  
hello world PySpark
```

```
+-----+-----+  
| word | count |  
+-----+-----+  
| hello|    3|  
| world|    2|  
| PySpark|    2|  
+-----+-----+
```

```
from pyspark.sql import SparkSession # Initialize SparkSession
spark = SparkSession.builder.appName("JoinExample").getOrCreate() # Create DataFrame 1
data1 = [ ("1", "Alice"), ("2", "Bob"), ("3", "Cathy") ] columns1 = ["emp_id", "name"]
df1 = spark.createDataFrame(data1, columns1) # Create DataFrame 2
data2 = [ ("101", "HR"), ("102", "Finance"), ("1", "IT"), ("2", "HR") ] columns2 = ["dept_emp_id", "department"]
df2 = spark.createDataFrame(data2, columns2) # Perform the join using different keys
joined_df = df1.join(df2, df1["emp_id"] == df2["dept_emp_id"], how="inner") # Show the result
joined_df.show()
```



df1

1	Alice
2	Bob
3	Cathy

df2

101	HR
102	Finance
1	IT
2	HR

1	Alice	1	IT
2	Bob	2	HR

inner
left

right

outer

.withColumnRenamed()

```

from pyspark.sql import SparkSession from pyspark.sql.functions import broadcast #
Initialize SparkSession spark =
SparkSession.builder.appName("BroadcastJoinExample").getOrCreate() # Create a
large DataFrame large_data = [(1, "Alice", "NY"), (2, "Bob", "LA"), (3,
"Cathy", "SF"), (4, "David", "NY")] large_columns = ["emp_id", "name", "city"]
large_df = spark.createDataFrame(large_data, large_columns) # Create a small
DataFrame small_data = [("NY", "New York"), ("LA", "Los Angeles"), ("SF", "San
Francisco")] small_columns = ["city_code", "city_name"] small_df =
spark.createDataFrame(small_data, small_columns) # Perform a broadcast join
broadcast_joined_df = large_df.join( broadcast(small_df), large_df["city"] ==
small_df["city_code"], how="inner" ) # Show the result broadcast_joined_df.show()

```



large_df

1	Alice	NY
2	Bob	LA

3	Cathy	SF
4	David	NY

small_df

NY	New York
LA	Los Angeles
SF	San Francisco

1	Alice	NY	NY	New York
2	Bob	LA	LA	Los Angeles
3	Cathy	SF	SF	San Francisco
4	David	NY	NY	New York

```
print(small_df.storageLevel)
```

```
print(small_df.storageLevel)
```

```
small_df
```

```
small_df
```

```
StorageLevel(0, 0, False, False, 1)
```

```
small_df
```

```
broadcast(small_df)
```

```
from pyspark.sql.functions import broadcast broadcast_small_df =  
broadcast(small_df) print(broadcast_small_df.storageLevel)
```

```
StorageLevel(1, 0, True, False, 1)
```

```
small_df.cache() print(small_df.storageLevel)
```

```

from pyspark.sql import SparkSession from pyspark.sql.window import Window from
pyspark.sql.functions import rank spark =
SparkSession.builder.appName("WindowFunctions").getOrCreate() data = [("Alice",
"HR", 5000), ("Bob", "HR", 6000), ("Cathy", "IT", 7000), ("David", "IT", 8000),
("Eve", "HR", 4500)] columns = ["Name", "Department", "Salary"] df =
spark.createDataFrame(data, columns) # Define a window specification window_spec =
Window.partitionBy("Department").orderBy(df["Salary"].desc()) # Apply rank
function ranked_df = df.withColumn("Rank", rank().over(window_spec))
ranked_df.show()

```

Bob	HR	6000	1
Alice	HR	5000	2
Eve	HR	4500	3
David	IT	8000	1
Cathy	IT	7000	2

```

from pyspark.sql.functions import sum data = [("East", "2023-01-01", 100),
("East", "2023-01-02", 200), ("West", "2023-01-01", 300), ("West", "2023-01-02",
400), ("East", "2023-01-03", 150)] columns = ["Region", "Date", "Sales"] df =
spark.createDataFrame(data, columns) # Define a window specification window_spec =

```

```

Window.partitionBy("Region").orderBy("Date").rowsBetween(Window.unboundedPreceding
, Window.currentRow) # Apply cumulative sum
cumulative_sum_df =
df.withColumn("Cumulative_Sales", sum("Sales").over(window_spec))
cumulative_sum_df.show()

```

East	2023-01-01	100	100
East	2023-01-02	200	300
East	2023-01-03	150	450
West	2023-01-01	300	300
West	2023-01-02	400	700

```

from pyspark.sql.functions import first, last
data = [("Alice", "2023-01-01", 100), ("Alice", "2023-01-02", 150), ("Bob", "2023-01-01", 200), ("Bob", "2023-01-03", 250), ("Alice", "2023-01-03", 200)]
columns = ["User", "Date", "Amount"]
df = spark.createDataFrame(data, columns)
# Define a window specification
window_spec = Window.partitionBy("User").orderBy("Date") # Apply first and last functions
transaction_dates_df = df.withColumn("First_Transaction",
first("Date").over(window_spec)) \ .withColumn("Last_Transaction",
last("Date").over(window_spec))
transaction_dates_df.show()

```

Alice	2023-01-01	100	2023-01-01	2023-01-03
Alice	2023-01-02	150	2023-01-01	2023-01-03

Alice	2023-01-03	200	2023-01-01	2023-01-03
Bob	2023-01-01	200	2023-01-01	2023-01-03
Bob	2023-01-03	250	2023-01-01	2023-01-03

```
from pyspark.sql.functions import avg
data = [("East", "2023-01-01", 100), ("East", "2023-01-02", 200), ("East", "2023-01-03", 300), ("West", "2023-01-01", 400), ("West", "2023-01-02", 500), ("West", "2023-01-03", 600)]
columns = ["Region", "Date", "Sales"]
df = spark.createDataFrame(data, columns) # Define a 3-day sliding window
window_spec = Window.partitionBy("Region").orderBy("Date").rowsBetween(-1, 1) # Calculate moving average
moving_avg_df = df.withColumn("Moving_Avg", avg("Sales")).over(window_spec))
moving_avg_df.show()
```

East	2023-01-01	100	150.0
East	2023-01-02	200	200.0
East	2023-01-03	300	250.0
West	2023-01-01	400	450.0
West	2023-01-02	500	500.0
West	2023-01-03	600	550.0

```
from pyspark.sql.functions import dense_rank
data = [("East", "Item1", 300), ("East", "Item2", 200), ("East", "Item3", 100), ("West", "Item4", 500), ("West", "Item5", 400), ("West", "Item6", 300)]
columns = ["Region", "Item", "Sales"]
df = spark.createDataFrame(data, columns)
# Define a window for ranking
window_spec = Window.partitionBy("Region").orderBy(df["Sales"].desc())
# Apply dense rank and filter top 2 items per region
ranked_df = df.withColumn("Rank", dense_rank().over(window_spec)).filter("Rank <= 2")
ranked_df.show()
```

Region	Item	Sales	Rank
East	Item1	300	1
East	Item2	200	2
West	Item4	500	1
West	Item5	400	2

```
from pyspark.sql.functions import sum, col
data = [("Electronics", "TV", 1000), ("Electronics", "Laptop", 2000), ("Furniture", "Table", 500), ("Furniture", "Chair", 300)]
columns = ["Category", "Product", "Sales"]
df = spark.createDataFrame(data, columns)
# Define a window for total sales by category
window_spec = Window.partitionBy("Category") # Calculate percentage of total sales
```

```
percentage_df = df.withColumn("Total_Sales", sum("Sales").over(window_spec)) \
.withColumn("Percentage", (col("Sales") / col("Total_Sales")) * 100)
percentage_df.show()
```

Electronics	TV	1000	3000	33.33
Electronics	Laptop	2000	3000	66.67
Furniture	Table	500	800	62.50
Furniture	Chair	300	800	37.50

```
from pyspark.sql.functions import sum from pyspark.sql.window import Window
data = [("Electronics", "2023-01-01", 1000), ("Electronics", "2023-01-02", 1500),
```

```

(["Furniture", "2023-01-01", 500], ["Furniture", "2023-01-03", 700]) columns =
["Category", "Date", "Sales"] df = spark.createDataFrame(data, columns) # Define a
window window_spec =
Window.partitionBy("Category").orderBy("Date").rowsBetween(Window.unboundedPreceding,
Window.currentRow) # Calculate running total running_total_df =
df.withColumn("Running_Total", sum("Sales").over(window_spec))
running_total_df.show()

```

Electronics	2023-01-01	1000	1000
Electronics	2023-01-02	1500	2500
Furniture	2023-01-01	500	500
Furniture	2023-01-03	700	1200

```

from pyspark.sql.functions import row_number data = [{"Electronics": "TV", 1000},
("Electronics", "Laptop", 2000), ("Furniture", "Table", 500), ("Furniture",
"Chair", 300)] columns = ["Category", "Product", "Sales"] df =
spark.createDataFrame(data, columns) # Define a window window_spec =
Window.partitionBy("Category").orderBy("Sales") # Add row number row_number_df =
df.withColumn("Row_Number", row_number().over(window_spec)) row_number_df.show()

```

Electronics	TV	1000	1
Electronics	Laptop	2000	2
Furniture	Chair	300	1

Furniture	Table	500	2

```
from pyspark.sql.functions import lag, lead
data = [("Electronics", "TV", 1000), ("Electronics", "Laptop", 2000), ("Electronics", "Phone", 1500), ("Furniture", "Table", 500), ("Furniture", "Chair", 300)]
columns = ["Category", "Product", "Sales"]
df = spark.createDataFrame(data, columns) # Define a window
window_spec = Window.partitionBy("Category").orderBy("Sales") # Add lag and lead values
lag_lead_df = df.withColumn("Previous_Sales", lag("Sales").over(window_spec)) \
    .withColumn("Next_Sales", lead("Sales").over(window_spec))
lag_lead_df.show()
```

Electronics	TV	1000	null	1500
Electronics	Phone	1500	1000	2000
Electronics	Laptop	2000	1500	null
Furniture	Chair	300	null	500
Furniture	Table	500	300	null

```

from pyspark.sql.functions import lag, col
data = [("Electronics", "TV", 1000),
        ("Electronics", "Laptop", 2000), ("Electronics", "Phone", 1500),
        ("Furniture", "Table", 500), ("Furniture", "Chair", 300)]
columns = ["Category", "Product", "Sales"]
df = spark.createDataFrame(data, columns) # Define a window
window_spec = Window.partitionBy("Category").orderBy("Sales") # Calculate difference from
previous row
diff_df = df.withColumn("Previous_Sales",
                        lag("Sales").over(window_spec)) \
    .withColumn("Difference", col("Sales") - col("Previous_Sales"))
diff_df.show()

```

Electronics	TV	1000	null	null
Electronics	Phone	1500	1000	500
Electronics	Laptop	2000	1500	500
Furniture	Chair	300	null	null
Furniture	Table	500	300	200

```

from pyspark.sql.functions import first, last
data = [("Electronics", "TV", 1000),
        ("Electronics", "Laptop", 2000), ("Furniture", "Table", 500),
        ("Furniture", "Chair", 300)]
columns = ["Category", "Product", "Sales"]
df = spark.createDataFrame(data, columns) # Define a window
window_spec = Window.partitionBy("Category").orderBy("Sales") # Add first and last sales
first_last_df = df.withColumn("First_Sales", first("Sales").over(window_spec)) \
    .withColumn("Last_Sales", last("Sales").over(window_spec))
first_last_df.show()

```

Electronics	TV	1000	1000	2000
Electronics	Laptop	2000	1000	2000
Furniture	Chair	300	300	500
Furniture	Table	500	300	500

```
from pyspark.sql.functions import sum, col
data = [("Electronics", "TV", 1000), ("Electronics", "Laptop", 2000), ("Furniture", "Table", 500), ("Furniture", "Chair", 300)]
columns = ["Category", "Product", "Sales"]
df = spark.createDataFrame(data, columns)
# Define a window for total sales
window_spec = Window.partitionBy("Category")
# Calculate percentage share
percentage_df = df.withColumn("Total_Sales", sum("Sales").over(window_spec)) \
    .withColumn("Percentage_Share", (col("Sales") / col("Total_Sales")) * 100)
percentage_df.show()
```

Electronics	TV	1000	3000	33.33
Electronics	Laptop	2000	3000	66.67
Furniture	Table	500	800	62.50
Furniture	Chair	300	800	37.50

```
inferSchema
```

```
spark.read.csv()
```

```
StringType inferSchema
```

```
inferSchema
```

```
inferSchema=True
```

```
IntegerType DoubleType
```

```
TimestampType
```

```
inferSchema
```

```
StringType
```

```
Name,Age,Salary  
Alice,25,5000  
Bob,30,6000  
Cathy,28,55000
```

```
inferSchema=True  
IntegerType StringType Salary  
inferSchema  
StringType
```

```
Age  
DoubleType  
Name Age Salary
```

```
Age    Salary
```

```
inferSchema
```

```
# Without inferSchema df = spark.read.csv("path_to_file.csv", header=True)
df.select(df["Age"] + 1).show() # This will throw an error because Age is
treated as StringType.
```

```
inferSchema
```

```
# With inferSchema df = spark.read.csv("path_to_file.csv", header=True,
inferSchema=True) df.select(df["Age"] + 1).show() # Works correctly as Age
is IntegerType.
```

```
inferSchema
```

```
DoubleType
```

```
inferSchema
```

```
schema()
```

```
inferSchema
```

```
from pyspark.sql.types import StructType, StructField, StringType,
IntegerType schema = StructType([ StructField("Name", StringType(), True),
StructField("Age", IntegerType(), True), StructField("Salary",
IntegerType(), True) ]) df = spark.read.csv("path_to_file.csv",
header=True, schema=schema)
```

```
inferSchema=True
```

```
distinct()
```

```
dropDuplicates()
```

```
from pyspark.sql import SparkSession # Initialize SparkSession
spark = SparkSession.builder.appName("DistinctValuesExample").getOrCreate() # Sample
DataFrame data = [("Alice", "HR"), ("Bob", "IT"), ("Alice", "HR"), ("Cathy",
"Finance"), ("Bob", "IT")]
columns = ["Name", "Department"]
df = spark.createDataFrame(data, columns)
```

```
distinct()
```

```
distinct()
```

```
# Get distinct rows
distinct_rows = df.distinct()
distinct_rows.show()
```

Alice	HR
Bob	IT
Cathy	Finance

```
dropDuplicates()
```

```
dropDuplicates()
```

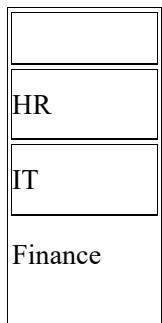
```
# Get distinct values of the 'Department' column distinct_departments =  
df.dropDuplicates(["Department"]).distinct_departments.show()
```

Alice	HR
Bob	IT
Cathy	Finance

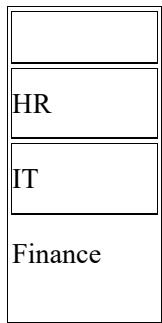
```
select()      distinct()
```

```
select()      distinct()
```

```
# Get distinct values from the 'Department' column distinct_departments =  
df.select("Department").distinct().distinct_departments.show()
```



```
# Create a temporary SQL view df.createOrReplaceTempView("employees") # Run SQL
query to select distinct values distinct_departments = spark.sql("SELECT DISTINCT
Department FROM employees") distinct_departments.show()
```



```
distinct()  
dropDuplicates()  
select()    distinct()
```

CREATE

TABLE

```
%sql CREATE TABLE active_account_samples_external USING PARQUET LOCATION  
'dbfs:/tmp/active_account_external' AS SELECT * FROM  
parquet.`dbfs:/active_account_samples`;
```

USING PARQUET

PARQUET

LOCATION

dbfs:/tmp/active_account_external

|

AS SELECT

AS SELECT

SELECT

* FROM parquet.<path>

parquet.

```
dbfs:/active_account_samples
```

```
dbutils.fs.ls("dbfs:/active_account_samples")  
dbfs:/tmp/active_account_external
```

```
cache()      persist()
```

```
cache()
```

```
df = spark.read.csv("path_to_file.csv") cached_df = df.cache()  
cached_df.show()
```

```
MEMORY_AND_DISK
```

```
cache()
```

```
from pyspark.storagelevel import StorageLevel
df = spark.read.csv("path_to_file.csv")
persisted_df = df.persist(StorageLevel.MEMORY_AND_DISK_SER)
persisted_df.show()
```

MEMORY_ONLY	Stores data in memory only. If memory is insufficient, recomputation occurs.
MEMORY_AND_DISK	Stores data in memory, spills to disk if necessary (default for <code>cache()</code>).
MEMORY_ONLY_SER	Stores serialized data in memory. More space-efficient but CPU-intensive.
MEMORY_AND_DISK_SER	Stores serialized data in memory and spills to disk.
DISK_ONLY	Stores data only on disk.
OFF_HEAP (experimental)	Stores data in off-heap memory, requires special configuration.

	cache()	persist()
	cache()	persist()
	Uses <code>MEMORY_AND_DISK</code> storage level by default.	No default; requires you to specify a <code>StorageLevel</code> explicitly.
	No control over storage levels; always uses memory/disk.	Allows fine-grained control over where and how data is stored.
Simpler to use when default storage behavior suffices.		Preferred when specific storage requirements (e.g., serialization) are needed.

You need to store intermediate results with default storage (memory/disk).	<code>cache()</code>	Simple and sufficient for most use cases.
You need control over the storage level (e.g., serialized data, disk-only).	<code>persist(StorageLevel)</code>	Offers flexibility to match the data size and available resources.
Data will be accessed multiple times, and recomputation is expensive.	<code>cache()</code> or <code>persist()</code>	Prevents recomputation and saves computational resources.
Memory is a constraint, and you want serialized storage.	<code>persist(StorageLevel.MEMORY_AND_DISK_SER)</code>	Reduces memory usage by storing serialized data.

```
cache()
df = spark.read.csv("path_to_file.csv") # Cache the DataFrame df_cached =
df.cache() # Perform actions print(df_cached.count()) # First action triggers
caching df_cached.show() # Data is reused without recomputation
```

```
from pyspark.storagelevel import StorageLevel df =
spark.read.csv("path_to_file.csv") # Persist the DataFrame with
MEMORY_AND_DISK_SER df_persisted = df.persist(StorageLevel.MEMORY_AND_DISK_SER) # Perform actions print(df_persisted.count()) # First action triggers persisting
df_persisted.show() # Data is reused without recomputation
```

```
print(df_cached.storageLevel) print(df_persisted.storageLevel)
```

```
unpersist()
```

```
df_persisted.unpersist()
```

```
cache()
```

```
persist()
```

```
MEMORY_AND_DISK
```

```
persist()
```

```
pivot()
```

```

from pyspark.sql import SparkSession from pyspark.sql.functions import sum #
Initialize SparkSession spark =
SparkSession.builder.appName("PivotExamples").getOrCreate() # Sample DataFrame
data = [("East", "Electronics", 5000), ("East", "Furniture", 2000), ("West",
"Electronics", 7000), ("West", "Furniture", 3000)] columns = ["Region",
"Category", "Sales"] df = spark.createDataFrame(data, columns) # Pivot the data
pivot_df = df.groupBy("Region").pivot("Category").sum("Sales") pivot_df.show()

```

East	5000	2000
West	7000	3000

```

from pyspark.sql.functions import avg # Sample DataFrame data = [("East",
"Electronics", 5000), ("East", "Furniture", 2000), ("West", "Electronics", 7000),
("West", "Furniture", 3000), ("East", "Electronics", 6000)] columns = ["Region",
"Category", "Sales"] df = spark.createDataFrame(data, columns) # Pivot and perform
multiple aggregations pivot_total =
df.groupBy("Region").pivot("Category").sum("Sales") pivot_avg =
df.groupBy("Region").pivot("Category").avg("Sales") print("Total Sales:")
pivot_total.show() print("Average Sales:") pivot_avg.show()

```

East	11000	2000
West	7000	3000

East	5500.0	2000.0
West	7000.0	3000.0

```
data = [("East", "Electronics", 2022, 5000), ("East", "Furniture", 2022, 2000),  
       ("West", "Electronics", 2023, 7000), ("West", "Furniture", 2023, 3000), ("East",  
       "Electronics", 2023, 6000)] columns = ["Region", "Category", "Year", "Sales"] df =  
spark.createDataFrame(data, columns) # Pivot the data on Year pivot_df =  
df.groupBy("Region", "Category").pivot("Year").sum("Sales") pivot_df.show()
```

East	Electronics	5000	6000
East	Furniture	2000	null
West	Electronics	null	7000
West	Furniture	null	3000

```
# Pivot with specified categories pivot_df =  
df.groupBy("Region").pivot("Category", ["Electronics", "Furniture"]).sum("Sales")  
pivot_df.show()
```

East	11000	2000
West	7000	3000

```
# Pivot the data pivot_df = df.groupBy("Region").pivot("Category").sum("Sales") #  
Add a total column result_df = pivot_df.withColumn("Total_Sales",  
pivot_df["Electronics"] + pivot_df["Furniture"]) result_df.show()
```

East	11000	2000	13000
West	7000	3000	10000

```
# Dynamically determine pivot columns categories = [row["Category"] for row in  
df.select("Category").distinct().collect()] # Pivot dynamically pivot_df =  
df.groupBy("Region").pivot("Category", categories).sum("Sales") pivot_df.show()
```

East	11000	2000
West	7000	3000

```
# Count the number of transactions pivot_df =  
df.groupBy("Region").pivot("Category").count() pivot_df.show()
```

East	2	1
West	1	1

```
from pyspark.sql.functions import col # Total sales by region pivot_df =  
df.groupBy("Region").pivot("Category").sum("Sales") # Calculate percentages  
percent_df = pivot_df.withColumn("Electronics_Percentage", (col("Electronics") /  
(col("Electronics") + col("Furniture")) * 100) \  
.withColumn("Furniture_Percentage", (col("Furniture") / (col("Electronics") +  
col("Furniture")) * 100) percent_df.show()
```

East	11000	2000	84.62	15.38
West	7000	3000	70.00	30.00

pivot()

pivot()

groupBy()

pivot()

pivot()
pivot()

groupBy()

groupBy

groupBy()

```
data = [("Alice", "HR", 3000), ("Bob", "IT", 4000), ("Alice", "Finance", 3500),
("Bob", "HR", 2000)] columns = ["Name", "Department", "Salary"] df =
spark.createDataFrame(data, columns)
```

```
# Invalid: Attempt to pivot without groupBy pivot_df =
df.pivot("Department").sum("Salary")
```

`pivot()
groupBy()`

```
# Correct: Use pivot with groupBy
pivot_df = df.groupBy("Name").pivot("Department").sum("Salary")
pivot_df.show()
```

Alice	3500	3000	null
Bob	null	2000	4000

`groupBy()
groupBy()
agg()`

```
from pyspark.sql.functions import when, sum # Manually create wide format using
conditional aggregation
wide_df = df.groupBy("Name").agg(
    sum(when(df["Department"] == "HR", df["Salary"])).alias("HR"),
    sum(when(df["Department"] == "Finance", df["Salary"])).alias("Finance"),
    sum(when(df["Department"] == "IT", df["Salary"])).alias("IT"))
wide_df.show()
```

Alice	3000	3500	null
Bob	2000	null	4000

pivot()

groupBy()
groupBy

groupBy()
when() agg()

pivot()

```
data = [ ("Alice", "HR", 3000), ("Bob", "HR", 4000), ("Alice", "Finance", 3500),
("Bob", "Finance", 4500), ("Alice", "IT", 5000), ("Bob", "IT", 5500) ] columns =
["Name", "Department", "Salary"] df = spark.createDataFrame(data, columns)
```

```
# GroupBy and Pivot on the same column 'Department' pivot_df =  
df.groupBy("Department").pivot("Department").sum("Salary") pivot_df.show()
```

Finance	8000	null	null
HR	null	7000	null
IT	null	null	10500

```
# GroupBy 'Name' and Pivot on 'Department' pivot_df =  
df.groupBy("Name").pivot("Department").sum("Salary") pivot_df.show()
```

Alice	3500	3000	5000
Bob	4500	4000	5500

```
groupBy      pivot
```

```
pivot(column, values)
```

```
pivot_df = df.groupBy("Department").pivot("Department").sum("Salary")
```

DataFrame

df.groupBy("Department")

Department

df

Alice	HR	3000
Bob	HR	4000
Alice	Finance	3500
Bob	Finance	4500
Alice	IT	5000
Bob	IT	5500

groupBy("Department")

HR Finance IT

.pivot("Department")

Department

Department

HR Finance IT

groupBy

HR Finance IT

```
window_spec =  
Window.partitionBy("Region").orderBy("Date").rowsBetween(Window.unboundedPreceding  
, Window.currentRow)
```

```
Window.partitionBy("Region")  
    partitionBy("Region")  
        Region  
            Region
```

East	2023-01-01	100
East	2023-01-02	200
West	2023-01-01	300
West	2023-01-02	400

```
partitionBy("Region")
```

```
          East  
2023-01-01 | 100  
2023-01-02 | 200  
          West
```

```
2023-01-01 | 300  
2023-01-02 | 400
```

```
.orderBy("Date")
```

```
Date
```

```
East
```

```
Date
```

2023-01-01	100
2023-01-02	200

```
West
```

```
Date
```

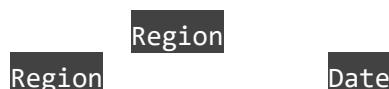
2023-01-01	300
2023-01-02	400

```
.rowsBetween(Window.unboundedPreceding, Window.CurrentRow)
```

```
Window.unboundedPreceding
```

```
Window.CurrentRow
```

```
rowsBetween(Window.unboundedPreceding, Window.currentRow)
```



```
from pyspark.sql import SparkSession from pyspark.sql.window import Window from
pyspark.sql.functions import sum spark =
SparkSession.builder.appName("WindowSpecExample").getOrCreate() # Sample Data data
= [("East", "2023-01-01", 100), ("East", "2023-01-02", 200), ("West", "2023-01-
01", 300), ("West", "2023-01-02", 400)] columns = ["Region", "Date", "Sales"] df =
spark.createDataFrame(data, columns) # Define window specification window_spec =
Window.partitionBy("Region").orderBy("Date").rowsBetween(Window.unboundedPreceding
, Window.currentRow) # Apply cumulative sum result_df =
df.withColumn("Cumulative_Sales", sum("Sales").over(window_spec)) result_df.show()
```

East	2023-01-01	100	100
East	2023-01-02	200	300
West	2023-01-01	300	300

West	2023-01-02	400	700

Region

Date

Sales

```

from pyspark.sql import SparkSession from pyspark.sql.window import Window from
pyspark.sql.functions import avg # Initialize SparkSession spark =
SparkSession.builder.appName("RunningAverageExample").getOrCreate() # Sample Data
data = [ ("East", "2023-01-01", 100), ("East", "2023-01-02", 200), ("East", "2023-
01-03", 300), ("West", "2023-01-01", 400), ("West", "2023-01-02", 500), ("West",
"2023-01-03", 600) ] columns = ["Region", "Date", "Sales"] df =
spark.createDataFrame(data, columns) # Define window specification window_spec =
Window.partitionBy("Region").orderBy("Date").rowsBetween(Window.unboundedPreceding
, Window.currentRow) # Calculate running average result_df =
df.withColumn("Running_Average", avg("Sales").over(window_spec)) # Show the result
result_df.show()

```

Region	Date	Sales	Running_Average
East	2023-01-01	100	100.0
East	2023-01-02	200	150.0
East	2023-01-03	300	200.0
West	2023-01-01	400	400.0
West	2023-01-02	500	450.0
West	2023-01-03	600	500.0

Region Date

Sales

```

Window.partitionBy("Region").orderBy("Date").rowsBetween(Window.unboundedPr
eceding, Window.currentRow)
    partitionBy("Region")
        Region
    orderBy("Date")
        Date

```

```
rowsBetween(Window.unboundedPreceding, Window.CurrentRow)
```

```
avg("Sales").over(window_spec)  
Sales
```

```
Running_Average
```

```
Region
```

```
Date
```

```
spark.sparkContext.setCheckpointDir("path/to/checkpoint/directory")
```

```
rdd = spark.sparkContext.parallelize([1, 2, 3, 4, 5]) rdd = rdd.map(lambda x: x *  
2) # Apply checkpointing rdd.checkpoint() # Perform an action to trigger  
checkpointing print(rdd.collect())  
checkpoint()  
collect()
```

```
from pyspark.sql import SparkSession # Initialize SparkSession spark =  
SparkSession.builder.appName("CheckpointingExample").getOrCreate() # Create a  
DataFrame data = [(1, "Alice"), (2, "Bob"), (3, "Cathy")] df =  
spark.createDataFrame(data, ["ID", "Name"]) # Enable checkpoint directory  
spark.sparkContext.setCheckpointDir("path/to/checkpoint/dir") # Apply  
checkpointing df = df.checkpoint() # Perform an action to trigger checkpointing  
df.show()
```

```
# Apply local checkpointing df = df.localCheckpoint() # Perform an action to  
trigger local checkpointing df.show()
```

	Stores data in memory for reuse.	Truncates lineage and stores data reliably.

	Memory (and disk if necessary).	Disk or distributed storage (e.g., HDFS).
	Retains the lineage for recomputation.	Breaks lineage; no recomputation is needed.
	Repeated access to intermediate results.	Fault tolerance for long lineage chains.
Faster (data in memory).		Slower (data is written to disk).

```
# Enable checkpointing
spark.sparkContext.setCheckpointDir("path/to/checkpoint/dir") # Create an RDD rdd
= spark.sparkContext.parallelize([1, 2, 3, 4, 5]) # Apply multiple transformations
rdd = rdd.map(lambda x: x + 1).filter(lambda x: x % 2 == 0) # Checkpoint to break
lineage rdd.checkpoint() # Trigger checkpointing print(rdd.collect())
```

map

filter

```
spark.task.maxFailures=4
```

```
spark.task.maxFailures (default: 4) spark.speculation (default: false)
```

```
spark.speculation=true
```

```
spark.sparkContext.setCheckpointDir("path/to/checkpoint")
```

```
df = df.cache()
```

```
spark.sql.shuffle.partitions
```

```
spark.speculation=true
```

```
spark.task.maxFailures=8
```

```
append overwrite checkpoint
```

partitionBy

```
data = [ ("Alice", "HR", 5000), ("Bob", "IT", 7000), ("Cathy", "HR", 4500),
        ("David", "Finance", 6000) ] columns = ["Name", "Department", "Salary"] df =
spark.createDataFrame(data, columns)
```

```
# Write data with dynamic partitioning by "Department"
df.write.partitionBy("Department").parquet("output_path")
```

```
output_path/
├── Department=Finance/
│   └── part-00000-*.parquet
├── Department=HR/
│   ├── part-00000-*.parquet
│   └── part-00001-*.parquet
└── Department=IT/
    └── part-00000-*.parquet
```

	Partitions are created dynamically based on column values.	Partitions are predefined, and values must be specified.
	No need to specify exact partition values in advance.	Requires exact partition values at runtime.
	Data with unknown or varying partition keys (e.g., user IDs).	Data with fixed partition keys (e.g., fixed time periods).
	May be slower due to runtime partition creation.	Typically faster since partitions are already defined.

```
spark.sql("SET hive.exec.dynamic.partition=true") spark.sql("SET
hive.exec.dynamic.partition.mode=nonstrict")
    hive.exec.dynamic.partition
    hive.exec.dynamic.partition.mode
        strict
        nonstrict
```

```
# Assuming the Hive table 'employee_partitioned' exists
df.write.mode("overwrite").insertInto("employee_partitioned")
```

`employee_partitioned`

`Department`

```
SELECT * FROM sales JOIN regions ON sales.region_id = regions.id WHERE  
regions.country = 'USA';
```

USA

```
from pyspark.sql import SparkSession
spark = SparkSession.builder \
    .appName("AQEExample") \
    .config("spark.sql.adaptive.enabled", "true") \
    .getOrCreate()
```

<code>spark.sql.adaptive.enabled</code>	<code>false</code>	Enables/disables AQE.
<code>spark.sql.adaptive.coalescePartitions.enabled</code>	<code>true</code>	Dynamically coalesces shuffle partitions.
<code>spark.sql.adaptive.skewJoin.enabled</code>	<code>true</code>	Enables handling of skewed joins.
<code>spark.sql.adaptive.join.enabled</code>	<code>true</code>	Enables runtime switching of join strategies (e.g., broadcast join).

<code>spark.sql.adaptive.coalescePartitions.minPartitionSize</code>	64MB	Minimum size of shuffle partitions after coalescing.

```
from pyspark.sql import SparkSession # Initialize SparkSession with AQE enabled
spark = SparkSession.builder \ .appName("AQEExample") \
.config("spark.sql.adaptive.enabled", "true") \ .getOrCreate() # Sample DataFrames
data1 = [(1, "Alice"), (2, "Bob"), (3, "Cathy")] data2 = [(1, "HR"), (2, "IT"),
(3, "Finance"), (4, "Sales")]
df1 = spark.createDataFrame(data1, ["id", "name"])
df2 = spark.createDataFrame(data2, ["id", "department"]) # Perform a join result =
df1.join(df2, "id")
result.show()
```

df2

```
rdd = spark.parallelize([1, 2, 3, 4]) rdd.map(lambda x: x *  
2).collect() # Data is serialized during the mapping operation
```

map filter

```
multiplier = 2 rdd.map(lambda x: x * multiplier).collect() # The  
lambda function and the `multiplier` variable are serialized to be  
sent to executors.
```

```
from pyspark import SparkContext sc = SparkContext() rdd =  
sc.parallelize([1, 2, 3, 4]) rdd.map(lambda x: x * 2).collect() # Python  
objects are serialized using Pickle
```

```
from pyspark import SparkConf, SparkContext conf =  
SparkConf().set("spark.serializer",  
"org.apache.spark.serializer.KryoSerializer") sc = SparkContext(conf=conf)
```

```
map filter reduce
```

```
rdd = sc.parallelize([1, 2, 3, 4]) rdd.map(lambda x: x *  
2).collect() # The lambda function is serialized
```

```
from pyspark.sql.functions import udf @udf("int") def  
multiply_by_two(x): return x * 2 df = spark.createDataFrame([(1,),  
(2,), (3,)], ["value"]) df.withColumn("result",  
multiply_by_two("value")).show() # The `multiply_by_two` function is  
serialized to be executed on worker nodes.
```

```
broadcast_var = sc.broadcast([1, 2, 3]) print(broadcast_var.value) #  
Serialized and sent to executors
```

```
class NonSerializableClass: pass rdd.map(lambda x:  
NonSerializableClass()).collect() # Will throw a serialization error
```

```
conf.set("spark.serializer",
"org.apache.spark.serializer.KryoSerializer")
```

```
broadcast()
```

```
sc.setLogLevel("DEBUG")
```

```
ProductID RegionID  
WHERE
```

```
RegionID
```

```
TransactionID Timestamp
```

```
TransactionDate  
SELECT * FROM SalesFact WHERE TransactionDate BETWEEN '2023-01-01'  
AND '2023-01-31';
```

ProductID RegionID

```
(ProductID, RegionID)
SELECT SUM(Sales) FROM SalesFact WHERE ProductID = 101 AND RegionID
= 2;
```

TransactionID

TransactionID

```
SELECT * FROM SalesFact WHERE TransactionID = 12345;
```

CustomerID	ProductID
------------	-----------

CustomerID

```
SELECT SUM(Sales) FROM SalesFact WHERE CustomerID = 567;
```

ProductID	Sales
-----------	-------

```
SELECT ProductID, SUM(Sales) FROM SalesFact GROUP BY ProductID;
```

`TransactionID`

`ProductID` `CustomerID` `RegionID`
`Date` `SalesCategory`

	Low cardinality columns	Compact, efficient for filtering	<code>RegionID</code> , <code>ProductID</code>
	High cardinality columns, range queries	Fast for point lookups and ranges	<code>TransactionDate</code>
	Queries filtering on multiple columns	Optimized for multi-column filters	<code>(ProductID, RegionID)</code>

	Primary keys, range-based queries	Faster range queries, sorted storage	<code>TransactionID</code>
	Secondary filters, join keys	Flexible, multiple indexes allowed	<code>CustomerID, ProductID</code>
	Query-specific optimizations	Eliminates lookups, reduces I/O	<code>ProductID, Sales</code>

`spark.read`

`DataFrameReader`

```
# Basic reading from a data source df =
spark.read.format("<file_format>").option("<key>", "<value>").load("<path>")
```

	<code>spark.read.csv(path)</code>	Structured data in text format.
	<code>spark.read.json(path)</code>	Semi-structured data.
	<code>spark.read.parquet(path)</code>	Columnar storage format.

	<code>spark.read.orc(path)</code>	Optimized columnar format.
	<code>spark.read.text(path)</code>	Plain text data.
	<code>spark.read.jdbc(...)</code>	Connects to relational databases.
<code>spark.sql("SELECT * FROM table")</code>		Reads from Hive tables.

```
header
inferSchema
```

```
delimiter
schema
```

```
df = spark.read.format("csv") \ .option("header", "true") \ .option("inferSchema",
"true") \ .option("delimiter", ",") \ .load("path/to/data.csv") df.show()
```

```
DataFrameWriter      df.write
```

```
df.write.format("<file_format>").option("<key>",
"<value>").mode("<mode>").save("<path>")
```

	<code>df.write.csv(path)</code>	Write structured data to text.
	<code>df.write.json(path)</code>	Write semi-structured data.

	<code>df.write.parquet(path)</code>	Write in a columnar format.
	<code>df.write.orc(path)</code>	Write optimized columnar format.
	<code>df.write.text(path)</code>	Write plain text data.
	<code>df.write.jdbc(...)</code>	Write to relational databases.
	<code>df.write.insertInto(table)</code>	Insert data into Hive tables.

mode

<code>append</code>	Adds new data to the existing data at the target location.
<code>overwrite</code>	Overwrites the existing data at the target location.
<code>ignore</code>	Does not write the data if the target already exists (no error).
<code>error</code> or <code>errorIfExists</code>	(Default) Throws an error if the target already exists.

overwrite

```
df.write.format("parquet") \ .mode("overwrite") \ .save("path/to/output.parquet")
```

```
append  
df.write.format("csv") \ .option("header", "true") \ .mode("append") \  
.save("path/to/output.csv")
```

```
df.write.mode("overwrite").insertInto("hive_table_name")
```

overwrite

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

- output/
- └ Region=East/
- └ part-00000.parquet
- └ Region=West/
- └ part-00001.parquet

```
df.write.bucketBy(4, "CustomerID").saveAsTable("bucketed_table")
```

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

```
from pyspark.sql.types import StructType, StructField, StringType, IntegerType
schema = StructType([ StructField("Name", StringType(), True), StructField("Age", IntegerType(), True), StructField("City", StringType(), True) ])
df = spark.read.csv("path", schema=schema, header=True)
```

```
df.write.option("compression", "gzip").csv("path")
```

```
df.write.option("delimiter", "|").csv("path")
```

```
df.write.option("header", "true").csv("path")
```

```
df.coalesce(1).write.csv("path")
```

```
df.repartition(10).write.csv("path")
```

	Schema is inferred or explicitly defined.	Output schema matches the DataFrame schema.
	Used for delimiter, header, etc.	Used for compression, partitioning, etc.
	Determines how data is loaded.	Determines how existing data is handled.
Not applicable.		Supports partitioning and bucketing.

readStream

```
df = spark.readStream.format("<source>").option("<key>", "<value>").load()
```

<code>format</code>	Specifies the source format (e.g., <code>csv</code> , <code>parquet</code> , <code>kafka</code> , <code>socket</code>).
<code>load()</code>	Loads data from the specified source (e.g., directory path, Kafka topic).
<code>schema</code>	Defines the schema of the streaming data explicitly (required for file-based sources like CSV).
<code>startingOffsets</code>	(Kafka) Determines where to start reading (<code>earliest</code> , <code>latest</code> , or specific offsets).
<code>maxFilesPerTrigger</code>	(Files) Limits the number of new files read per trigger, useful for throttling.
<code>includeTimestamp</code>	(Socket) Adds a timestamp column to include the message's ingestion time.

```
# Read from a file source (e.g., new CSV files in a directory)
schema = "Name STRING, Age INT, City STRING"
df = spark.readStream \
    .format("csv") \
    .schema(schema) \
    .option("header", "true") \
    .option("maxFilesPerTrigger", 1) \
    .load("path/to/directory")
```

```
# Read from Kafka source
df = spark.readStream \
    .format("kafka") \
    .option("kafka.bootstrap.servers", "localhost:9092") \
    .option("subscribe", "topic_name") \
    .option("startingOffsets", "earliest") \
    .load()
```

writeStream

```
df.writeStream \ .format("<sink>") \ .outputMode("<output_mode>") \ .option("<key>", "<value>") \ .start("<path_or_endpoint>")
```

<code>append</code>	Only new rows in the result table are written to the sink.	Best for sources that do not update data (e.g., Kafka, file sources).
<code>complete</code>	The entire result table is written to the sink every time (recomputes the whole result).	Useful for aggregations (e.g., total counts).
<code>update</code>	Only rows with updates in the result table are written to the sink.	Useful for stateful operations like aggregations but avoids full recomputation.

<code>console</code>	Prints output to the console (useful for debugging).
<code>memory</code>	Writes to an in-memory table for debugging (non-persistent).
<code>file</code>	Writes data to a directory (e.g., in CSV or Parquet format).

kafka	Writes data to a Kafka topic.
foreach	Custom sink allowing user-defined operations for each row.
delta	Writes to a Delta Lake table for ACID-compliant data storage.

```
# Write data to the console query = df.writeStream \ .format("console") \  
.outputMode("append") \ .start() query.awaitTermination()
```

```
# Write data to Kafka query = df.selectExpr("CAST(value AS STRING)") \  
.writeStream \ .format("kafka") \ .option("kafka.bootstrap.servers",  
"localhost:9092") \ .option("topic", "output_topic") \ .outputMode("append") \  
.start()
```

```
# Write data to Parquet files query = df.writeStream \ .format("parquet") \  
.option("path", "path/to/output/directory") \ .option("checkpointLocation",  
"path/to/checkpoint") \ .outputMode("append") \ .start() query.awaitTermination()
```

```
query = df.writeStream \ .format("parquet") \ .option("path", "path/to/output") \  
.option("checkpointLocation", "path/to/checkpoint") \ .outputMode("append") \  
.start()
```

	Processes data as soon as it arrives in micro-batches.
<code>trigger(once)</code>	Processes all available data once and stops.
<code>trigger(continuous)</code>	(Experimental) Continuous processing mode for low-latency requirements.
<code>trigger(processingTime='5 seconds')</code>	Executes the query at fixed intervals (e.g., every 5 seconds).

```
query = df.writeStream \ .format("console") \ .outputMode("append") \ .trigger(processingTime="5 seconds") \ .start()
```

	Defines how data is ingested into Spark.	Defines how processed data is written to sinks.
	Not explicitly defined; controlled by sources and triggers.	Output modes (<code>append</code> , <code>complete</code> , <code>update</code>).

	Checkpointing tracks data read progress.	Checkpointing tracks data write progress and ensures recovery.
	Controls batch frequency (e.g., micro-batch or continuous).	Executes writes based on trigger intervals.

```
from pyspark.sql import SparkSession # Initialize SparkSession
spark = SparkSession.builder \
    .appName("StructuredStreamingExample") \
    .getOrCreate() # Read stream from Kafka
input_df = spark.readStream \
    .format("kafka") \
    .option("kafka.bootstrap.servers", "localhost:9092") \
    .option("subscribe", "input_topic") \
    .option("startingOffsets", "earliest") \
    .load() # Transform the data
transformed_df = input_df.selectExpr("CAST(value AS STRING) AS message") #
Write to console
query = transformed_df.writeStream \
    .format("console") \
    .outputMode("append") \
    .trigger(processingTime="10 seconds") \
    .option("checkpointLocation", "path/to/checkpoint") \
    .start()
query.awaitTermination()
```

cache() persist()

	<code>cache()</code>	<code>persist()</code>
	<code>cache()</code>	<code>persist()</code>
	Uses <code>MEMORY_AND_DISK</code> by default.	Allows you to specify the storage level.
	No control over the storage level.	Flexible, supports various storage levels like memory, disk, and serialization.
Best for default caching requirements.		Suitable for fine-tuned storage needs.

`cache()`

`persist(StorageLevel.MEMORY_AND_DISK)`

`cache()`

```
from pyspark.sql import SparkSession # Initialize SparkSession
spark = SparkSession.builder.appName("CacheExample").getOrCreate() # Create a DataFrame
data = [("Alice", 25), ("Bob", 30), ("Cathy", 29)] columns = ["Name", "Age"]
df = spark.createDataFrame(data, columns) # Cache the DataFrame
cached_df = df.cache() # Trigger caching with an action
cached_df.show() # Reuse the cached DataFrame
print(f"Count: {cached_df.count()}")
```

`show()`
`count()`

```
persist()
```

```
cache()
```

```
MEMORY_ONLY  
MEMORY_AND_DISK
```

```
DISK_ONLY  
MEMORY_ONLY_SER  
OFF_HEAP
```

```
persist()  
persist()
```

```
from pyspark import StorageLevel # Persist the DataFrame with MEMORY_AND_DISK  
persisted_df = df.persist(StorageLevel.MEMORY_AND_DISK) # Trigger persistence with  
an action persisted_df.show() # Reuse the persisted DataFrame print("Count:  
{persisted_df.count()}")
```

```
persist(StorageLevel.MEMORY_AND_DISK)
```

```
persist()
```

	Stores data in memory. If memory is insufficient, data is recomputed.
	Stores data in memory. If memory is insufficient, spills to disk (default for <code>cache()</code>).
	Stores data in memory in serialized format (reduces memory usage but increases CPU usage).
	Stores data in serialized format in memory and disk.
	Stores data on disk only.

Stores data in off-heap memory (requires Spark to be configured with off-heap memory).

```
from pyspark import StorageLevel # Persist with DISK_ONLY storage level
persisted_df_disk = df.persist(StorageLevel.DISK_ONLY) # Persist with
MEMORY_ONLY_SER storage level persisted_df_memory =
df.persist(StorageLevel.MEMORY_ONLY_SER) # Trigger persistence with an action
persisted_df_disk.show() persisted_df_memory.show()
```

```
print(f"Storage Level: {persisted_df.storageLevel}")
```

```
Storage Level: Disk Memory Deserialized 1x Replicated
```

```
# Unpersist the DataFrame persisted_df.unpersist()
```

`cache()` `persist()`

```
# Without caching or persisting df_transformed = df.filter(df["Age"] > 25).groupBy("Name").count() df_transformed.show() # Transformation recomputed  
df_transformed.show() # Transformation recomputed again # With caching cached_df = df.filter(df["Age"] > 25).groupBy("Name").count().cache() cached_df.show() # Caching triggered cached_df.show() # Cached data reused # With persisting  
persisted_df = df.filter(df["Age"] > 25).groupBy("Name").count().persist(StorageLevel.MEMORY_ONLY) persisted_df.show()  
# Persistence triggered persisted_df.show() # Persisted data reused
```

`cache()`

`persist()`

`storageLevel`

```
spark.default.parallelism
```

```
# Create an RDD with 4 partitions rdd = spark.sparkContext.parallelize(range(1, 11), numSlices=4) # Check the number of partitions print(f"Number of partitions: {rdd.getNumPartitions()}") # View partition contents print(rdd.glom().collect()) # Displays data in each partition
```

```
4  
[[1, 2], [3, 4], [5, 6], [7, 8, 9, 10]]
```

```
rdd_repartitioned = rdd.repartition(6) print(f"Repartitioned to: {rdd_repartitioned.getNumPartitions()}")
```

```
rdd_coalesced = rdd.coalesce(2) print(f"Coalesced to: {rdd_coalesced.getNumPartitions()}")
```

```
spark.sql.shuffle.partitions
```

```
groupBy()    join()
```

```
df = spark.range(0, 100) # Default partitions print(f"Number of partitions: {df.rdd.getNumPartitions()}"
```

```
repartition()
```

```
df_repartitioned = df.repartition(5) print(f"Repartitioned to: {df_repartitioned.rdd.getNumPartitions()}"  
coalesce()
```

```
df_coalesced = df.coalesce(2) print(f"Coalesced to: {df_coalesced.rdd.getNumPartitions()}"
```

```
df = spark.createDataFrame([ ("Alice", "HR", 3000), ("Bob", "IT", 4000), ("Cathy", "HR", 3500), ("David", "Finance", 6000) ], [ "Name", "Department", "Salary" ]) # Write with partitioning by 'Department'  
df.write.partitionBy("Department").parquet("output_path")
```

```
output_path/  
├── Department=Finance/  
│   └── part-00000.parquet  
├── Department=HR/  
│   └── part-00001.parquet  
└── Department=IT/
```

```
└── part-00002.parquet
```

```
small_df = spark.read.csv("small.csv") large_df =  
spark.read.csv("large.csv") result = large_df.join(small_df.broadcast(),  
"key")
```

```
repartition()
```

```
repartition()  
coalesce()
```

```
partitionBy
```

```
df.groupBy(spark_partition_id()).count().show()
```

```
spark.sql.shuffle.partitions
```

<code>repartition(n)</code>	Increase partitions	Full shuffle; evenly distributes data into <code>n</code> partitions.
<code>coalesce(n)</code>	Reduce partitions	Minimizes shuffle; merges partitions into <code>n</code> .
<code>partitionBy(col)</code>	Write partitioned files	Dynamically creates output folders based on the values of <code>col</code> .
<code>spark.sql.shuffle.partitions</code>	Default shuffle partitions	Configures the number of partitions created during shuffles (default: 200).

`repartition` `coalesce` `partitionBy`

`coalesce()` `partitionBy()`

`coalesce()` `partitionBy()`

`coalesce()`

`repartition()`

```
coalesce()
from pyspark.sql import SparkSession # Initialize SparkSession
spark = SparkSession.builder.appName("CoalesceExample").getOrCreate() # Create a DataFrame
with 4 partitions
df = spark.range(0, 100).repartition(4)
print(f"Initial Partitions: {df.rdd.getNumPartitions()}")
# Output: 4
# Reduce partitions using coalesce
df_coalesced = df.coalesce(2)
print(f"Coalesced Partitions: {df_coalesced.rdd.getNumPartitions()}")
# Output: 2
# Write output with fewer files
df_coalesced.write.csv("output_path")
```

`partitionBy()`

partitionBy

partitionBy()

```
# Sample DataFrame data = [("Alice", "HR", 3000), ("Bob", "IT", 4000), ("Cathy", "HR", 3500), ("David", "Finance", 6000)] columns = ["Name", "Department", "Salary"] df = spark.createDataFrame(data, columns) # Write output partitioned by "Department" df.write.partitionBy("Department").parquet("output_path")
```

```
output_path/
├── Department=Finance/
│   └── part-00000.parquet
├── Department=HR/
│   ├── part-00001.parquet
│   └── part-00002.parquet
└── Department=IT/
    └── part-00003.parquet
```

	coalesce()	partitionBy()
	Reduces the number of partitions.	Dynamically partitions data based on column values.

	<code>coalesce()</code>	<code>partitionBy()</code>
	Works on an existing DataFrame or RDD.	Used during data writing (e.g., Parquet, CSV).
	No full shuffle (narrow transformation).	Requires a full shuffle (wide transformation).
	Optimizing partition count for processing or file writing.	Organizing output files into structured directories.
	No effect on directory structure.	Creates directories based on partition columns.
Reduces computational overhead when fewer partitions are needed.		Enables partition pruning for faster querying.

Reducing the number of partitions.	<code>coalesce()</code>	Avoids full shuffle; reduces computational cost.
Preparing output with structured directories.	<code>partitionBy()</code>	Organizes data into subdirectories for query optimization.
Reducing small output files.	<code>coalesce()</code>	Limits the number of output files without redistributing data.
Querying large datasets with filtering.	<code>partitionBy()</code>	Enables partition pruning to improve query performance by reducing the amount of data read.

`coalesce()`

`partitionBy()`

`coalesce()`

`partitionBy()`

```
# Reduce partitions before writing df_coalesced = df.coalesce(4) # Write
partitioned data to structured directories
df_coalesced.write.partitionBy("Department").parquet("output_path")
```

```
coalesce()
```

```
partitionBy()
```

```
coalesce()
```

```
partitionBy()
```

```
'DataFrame' object has no attribute 'getNumPartitions'
```

```
getNumPartitions
```

```
getNumPartitions
```

```
getNumPartitions
```

```
num_partitions = df.rdd.getNumPartitions() print(f"Number of partitions:
{num_partitions}")
```

```
df.rdd
```

```
getNumPartitions
```

```
from pyspark.sql import SparkSession # Initialize SparkSession
spark = SparkSession.builder.appName("PartitionExample").getOrCreate() # Create a
DataFrame data = [(1, "Alice"), (2, "Bob"), (3, "Cathy"), (4, "David")]
columns = ["ID", "Name"]
df = spark.createDataFrame(data, columns) # Check the number of
partitions
num_partitions = df.rdd.getNumPartitions()
print(f"Number of
partitions: {num_partitions}")
```

```
parallelize
```

```
spark.default.parallelism
```

```
df_repartitioned = df.repartition(4) # Creates 4 partitions
print(f"Number of
partitions after repartition: {df_repartitioned.rdd.getNumPartitions()}")
```

```
df_coalesced = df.coalesce(2) # Reduces partitions to 2 print(f"Number of  
partitions after coalesce: {df_coalesced.rdd.getNumPartitions()}")
```

getNumPartitions

```
df.rdd.getNumPartitions()
```

```
from pyspark.sql import SparkSession # Initialize SparkSession
spark = SparkSession.builder.appName("BroadcastExample").getOrCreate() # A small lookup
dictionary lookup_dict = {"HR": "Human Resources", "IT": "Information Technology",
"FIN": "Finance"} # Broadcast the dictionary
broadcast_lookup = spark.sparkContext.broadcast(lookup_dict) # Sample DataFrame
data = [("Alice", "HR"), ("Bob", "IT"), ("Cathy", "FIN")]
columns = ["Name", "DepartmentCode"]
df = spark.createDataFrame(data, columns) # Use the broadcast variable in a
transformation
def lookup_department(code):
    return broadcast_lookup.value.get(code, "Unknown")
from pyspark.sql.functions import udf
from pyspark.sql.types import StringType # Register the UDF
lookup_department_udf = udf(lookup_department, StringType()) # Apply the UDF to add a new column
result_df = df.withColumn("Department",
lookup_department_udf(df["DepartmentCode"]))
result_df.show()
```

Alice	HR	Human Resources
Bob	IT	Information Technology
Cathy	FIN	Finance

AccumulatorV2

```
from pyspark.sql import SparkSession # Initialize SparkSession
spark = SparkSession.builder.appName("AccumulatorsExample").getOrCreate() # Create an
accumulator
records_processed = spark.sparkContext.accumulator(0) # Sample data
data = [1, 2, 3, 4, 5]
rdd = spark.sparkContext.parallelize(data) # Use the
accumulator in an action
def process_record(x):
    global records_processed
    records_processed += 1
return x * 2
result = rdd.map(process_record).collect() # Print the accumulator value on the driver
print(f"Number of records processed:
{records_processed}")
```

```
Number of records processed: 5
```

```
# Create an accumulator for errors
error_count = spark.sparkContext.accumulator(0)
# Sample data
data = ["valid", "invalid", "valid", "invalid", "valid"]
rdd = spark.sparkContext.parallelize(data) # Use the accumulator to count errors
def validate_record(record):
    global error_count
    if record == "invalid":
        error_count += 1
return record
result = rdd.map(validate_record).collect() # Print the number of
errors on the driver
print(f"Number of errors: {error_count}")
```

```
Number of errors: 2
```

```
from pyspark.accumulators import AccumulatorParam # Define a custom accumulator
class SetAccumulator(AccumulatorParam): def zero(self, value): return set() def
addInPlace(self, value1, value2): return value1.union(value2) # Create the custom
accumulator unique_errors = spark.sparkContext.accumulator(set(),
SetAccumulator()) # Sample data data = ["error1", "error2", "error1", "error3",
"error2"] rdd = spark.sparkContext.parallelize(data) # Use the accumulator to
track unique errors def track_errors(error): global unique_errors unique_errors +=
set([error]) rdd.foreach(track_errors) # Print the unique errors print(f"Unique
errors: {unique_errors.value}")
```

```
Unique errors: {'error1', 'error2', 'error3'}
```

	Aggregate data across tasks.	Share read-only data with all tasks.
	Write-only for executors; read-only for driver.	Immutable (read-only) for executors and driver.
	Counting or summing across the cluster.	Sharing lookup tables, configurations, etc.
Driver can read the final value.		Driver cannot modify broadcasted data.

	Fully managed, user-friendly.	Requires manual setup.	Complex to configure.	Moderate complexity.
	Fully automated and native to cloud.	Requires manual configuration.	Limited support.	Requires custom setup.
	Native for AWS, Azure, GCP.	No direct integration.	Limited integration.	Custom integrations.
	Optimized for cloud-based workloads.	Not inherently cost-efficient.	May involve overhead.	Resource-efficient.
	Highly fault-tolerant.	Depends on setup.	High (YARN HA).	Moderate.

	Native for AWS, Azure, GCP.	No native cloud support.	No native cloud support.	No native cloud support.
	Fully automated and native to cloud.	Limited support.	Limited support.	Requires manual setup.
	Fully managed by Databricks.	Complex setup for Spark.	Complex setup.	Simple but limited.
	Built-in, cloud-native recovery.	High (with Hadoop HA).	Moderate.	Moderate.
Dynamically manages cloud instances.		Managed by YARN RM.	Managed by Mesos Master.	Managed by Spark Master.

```

from pyspark.sql import SparkSession from pyspark.sql.types import StructType,
StructField, StringType, IntegerType # Initialize SparkSession
spark = SparkSession.builder.appName("NullValuesExample").getOrCreate() # Define schema
schema = StructType([ StructField("Name", StringType(), True), StructField("Age",
IntegerType(), True), StructField("City", StringType(), True) ]) # Data with some
null values
data = [ ("Alice", 25, "New York"), ("Bob", None, "Los Angeles"),
(None, 29, "Chicago"), ("David", 32, None), ("Eve", None, None) ] # Create
DataFrame
df = spark.createDataFrame(data, schema=schema) # Show DataFrame
df.show()

```

Name	Age	City
Alice	25	New York
Bob	null	Los Angeles
null	29	Chicago
David	32	null
Eve	null	null

Name	null
Age	null
City	null

None null

orderBy()

orderBy()

orderBy()

orderBy()

ascending

False

desc()

```
orderBy()
```

```
sortWithinPartitions()
```

```
DataFrame.orderBy(*cols, ascending=True)
```

cols

ascending

True

```
from pyspark.sql import SparkSession # Initialize SparkSession
spark = SparkSession.builder.appName("OrderByExample").getOrCreate() # Sample DataFrame
data = [("Alice", 25), ("Bob", 30), ("Cathy", 29), ("David", 32)] columns =
[ "Name", "Age" ] df = spark.createDataFrame(data, columns) # Sort by Age in
ascending order (default) df_ordered = df.orderBy("Age") df_ordered.show()
```

+----+----+
Name Age
+----+----+
Alice 25
Cathy 29
Bob 30
David 32
+----+----+

```
# Sort by Age in descending order df_ordered = df.orderBy(df[ "Age" ].desc())
df_ordered.show()
```

```
+----+---+
| Name|Age|
+----+---+
|David| 32|
| Bob | 30|
|Cathy| 29|
|Alice| 25|
+----+---+
```

```
# Sample DataFrame with duplicate ages data = [("Alice", 25), ("Bob", 30),
("Cathy", 30), ("David", 32)] df = spark.createDataFrame(data, columns) # Sort by
Age (ascending) and Name (descending) df_ordered = df.orderBy("Age",
df[ "Name" ].desc()) df_ordered.show()
```

```
+----+---+
| Name|Age|
+----+---+
|Alice| 25|
|Cathy| 30|
| Bob | 30|
|David| 32|
+----+---+
```

ascending

ascending

```
# Sort by Age in descending order using ascending parameter df_ordered =
df.orderBy("Age", ascending=False) df_ordered.show()
```

```
+----+---+
| Name|Age|
+----+---+
|David| 32|
| Bob | 30|
|Cathy| 30|
|Alice| 25|
+----+---+
```

```
df["column"].desc()    df["column"].asc()
```

```
# Sort using column expressions df_ordered = df.orderBy(df[ "Age"].desc(),
df[ "Name"].asc()) df_ordered.show()
```

orderBy() **sort()**

orderBy() **sort()**

	orderBy()	sort()
	Specifically for global sorting.	Alias for orderBy() .
	Slightly more descriptive.	Simpler for basic usage.
Best practice for complex sorting.		Use for straightforward sorting.

`orderBy()`

`sortWithinPartitions()`

`orderBy()`

`orderBy()`

`ascending`

`asc()` `desc()`

```
from pyspark.sql import SparkSession # Initialize SparkSession
spark = SparkSession.builder.appName("BroadcastExample").getOrCreate() # Create a small
lookup dictionary
department_lookup = { "HR": "Human Resources", "IT": "Information Technology", "FIN": "Finance" } # Broadcast the dictionary
broadcasted_dept = spark.sparkContext.broadcast(department_lookup) # Sample
DataFrame data = [ ("Alice", "HR"), ("Bob", "IT"), ("Cathy", "FIN"), ("David", "HR") ] columns = ["Name", "DepartmentCode"] df = spark.createDataFrame(data, columns) # Define a function to use the broadcast variable
def map_department(code): return broadcasted_dept.value.get(code, "Unknown") # Register the function as a UDF
from pyspark.sql.functions import udf from pyspark.sql.types import StringType
map_department_udf = udf(map_department, StringType()) # Add a new column with the mapped department names
df_with_department = df.withColumn("Department", map_department_udf(df["DepartmentCode"])) # Show the resulting DataFrame
df_with_department.show()
```

Name	DepartmentCode	Department
Alice	HR	Human Resources
Bob	IT	Information Technology
Cathy	FIN	Finance
David	HR	Human Resources

```
department_lookup
spark.sparkContext.broadcast(department_lookup)
```

```
map_department
broadcasted_dept.value
```

`map_department_udf`
`map_department`

`Department`

`rank()` `dense_rank()`

<code>rank()</code>	Assigns ranks with gaps if there are ties.	Example ranks: 1, 2, 2, 4 (skips rank 3 due to tie).
<code>dense_rank()</code>	Assigns consecutive ranks without gaps, even if there are ties.	Example ranks: 1, 2, 2, 3 (no gaps in ranks).

```
from pyspark.sql import SparkSession from pyspark.sql.window import Window from pyspark.sql.functions import rank, dense_rank # Initialize SparkSession spark = SparkSession.builder.appName("RankExample").getOrCreate() # Sample DataFrame data = [("Alice", 100), ("Bob", 200), ("Cathy", 200), ("David", 300)] columns = ["Name", "Score"] df = spark.createDataFrame(data, columns) # Define a window specification window_spec = Window.orderBy(df["Score"].desc()) # Add rank and dense_rank columns df_ranked = df.withColumn("Rank", rank().over(window_spec)) \ .withColumn("Dense_Rank", dense_rank().over(window_spec)) df_ranked.show()
```

Name	Score	Rank	Dense_Rank
David	300	1	1
Bob	200	2	2
Cathy	200	2	2
Alice	100	4	3

rank()
dense_rank()

rank() dense_rank()

rank()

```
300, 200, 200, 100
```

```
1, 2, 2, 4
```

```
dense_rank()
```

```
300, 200, 200, 100
```

```
1, 2, 2, 3
```

```
rank()  
dense_rank()
```

```
concat()
```

```
pyspark.sql.functions
```

```
concat(*cols)  
cols
```

```
concat()
```

```
concat_ws()
```

```
from pyspark.sql import SparkSession from pyspark.sql.functions import concat #  
Initialize SparkSession spark =  
SparkSession.builder.appName("ConcatExample").getOrCreate() # Sample DataFrame  
data = [("Alice", "HR"), ("Bob", "IT"), ("Cathy", "Finance")] columns = ["Name",  
"Department"] df = spark.createDataFrame(data, columns) # Concatenate Name and  
Department df_with_concat = df.withColumn("FullInfo", concat(df["Name"],  
df["Department"])) df_with_concat.show()
```

```
+----+-----+-----+  
| Name|Department| FullInfo|  
+----+-----+-----+  
|Alice|      HR| AliceHR |  
| Bob|      IT| BobIT  |  
|Cathy| Finance|CathyFinance|  
+----+-----+-----+
```

```
lit()
```

```
from pyspark.sql.functions import lit # Add a separator or literal string  
df_with_concat = df.withColumn("FullInfo", concat(df["Name"], lit(" - "),  
df["Department"])) df_with_concat.show()
```

```
+----+-----+-----+
```

```
+-----+-----+-----+
| Name|Department|      FullInfo|
+-----+-----+-----+
| Alice|        HR|  Alice - HR   |
|  Bob|        IT|  Bob - IT    |
| Cathy|  Finance|Cathy - Finance|
+-----+-----+-----+
```

```
# Concatenate multiple columns
data = [("Alice", "HR", "New York"), ("Bob", "IT", "San Francisco"),
        ("Cathy", "Finance", "Chicago")]
columns = ["Name", "Department", "City"]
df = spark.createDataFrame(data, columns)
df_with_concat = df.withColumn("FullInfo", concat(df["Name"], lit(", "), df["Department"], lit(", "), df["City"]))

df_with_concat.show()
```

```
+-----+-----+-----+-----+
| Name|Department|      City|      FullInfo|
+-----+-----+-----+-----+
| Alice|        HR| New York|Alice, HR, New York   |
|  Bob|        IT|San Francisco|Bob, IT, San Francisco |
| Cathy|  Finance|     Chicago|Cathy, Finance, Chicago|
+-----+-----+-----+-----+
```

`concat_ws()`

`concat_ws()`

, - |

```
from pyspark.sql.functions import concat_ws # Concatenate with a delimiter
df_with_concat_ws = df.withColumn("FullInfo", concat_ws(", ", df["Name"],
df["Department"], df["City"]))
df_with_concat_ws.show()
```

```
+-----+-----+-----+-----+
| Name|Department|      City|      FullInfo|
+-----+-----+-----+-----+
| Alice|        HR| New York|Alice, HR, New York   |
|  Bob|        IT|San Francisco|Bob, IT, San Francisco |
| Cathy|  Finance|     Chicago|Cathy, Finance, Chicago|
+-----+-----+-----+-----+
```

```
+-----+-----+-----+-----+
```

```
concat()  
concat_ws()  
lit()
```

```
else  
when() otherwise()  
withColumn()  
if-  
pyspark.sql.functions
```

```
withColumn()  
when()  
otherwise()
```

```
from pyspark.sql import SparkSession from pyspark.sql.functions import when # Initialize SparkSession spark = SparkSession.builder.appName("AddColumnExample").getOrCreate() # Sample DataFrame data = [("Alice", 6000), ("Bob", 4000), ("Cathy", 2500), ("David", 7000)] columns = ["Name", "Salary"] df = spark.createDataFrame(data, columns) # Add a new column with conditional logic df_with_category = df.withColumn( "Category", when(df["Salary"] > 5000, "High") .when((df["Salary"] >= 3000) & (df["Salary"] <= 5000), "Medium") .otherwise("Low") ) # Show the result df_with_category.show()
```

```
+----+----+-----+
| Name|Salary|Category|
+----+----+-----+
|Alice| 6000|    High|
| Bob| 4000|  Medium|
|Cathy| 2500|     Low|
|David| 7000|    High|
+----+----+-----+
```

```
when()
```

```
when(condition, value)
  when()
otherwise()
      when()
```

```
withColumn()
```

```
from pyspark.sql.functions import col # Sample DataFrame data = [("Alice", "HR"), ("Bob", "IT"), ("Cathy", "Finance")] columns = ["Name", "Department"] df = spark.createDataFrame(data, columns) # Add a new column with conditional logic df_with_dept_category = df.withColumn("DeptCategory", when(col("Department") == "HR", "Human Resources").when(col("Department") == "IT", "Technology").otherwise("Other")) # Show the result df_with_dept_category.show()
```

Name	Department	DeptCategory
Alice	HR	Human Resources
Bob	IT	Technology
Cathy	Finance	Other

```
when()  
otherwise()
```

```
col()
```

```
between()
```

```
BETWEEN
```

```
between()
```

```
between()
```

```
pyspark.sql.Column
```

```
Column.between(lowerBound, upperBound)
```

```
lowerBound  
upperBound
```

```
between()  
between()
```

```
from pyspark.sql import SparkSession # Initialize SparkSession
spark = SparkSession.builder.appName("BetweenExample").getOrCreate() # Sample DataFrame
data = [("Alice", 6000), ("Bob", 4000), ("Cathy", 2500), ("David", 5000)] columns = ["Name", "Salary"]
df = spark.createDataFrame(data, columns) # Filter rows where Salary is between 3000 and 5000
df_filtered = df.filter(df["Salary"].between(3000, 5000)) # Show the result
df_filtered.show()
```

```
+---+----+
| Name|Salary|
+---+----+
| Bob| 4000|
| David| 5000|
+---+----+
```

between() withColumn

```
from pyspark.sql.functions import when # Add a new column with conditional logic
using between() df_with_category = df.withColumn( "Category",
when(df[ "Salary" ].between(3000, 5000), "Medium").otherwise("Other") ) # Show the
result df_with_category.show()
```

```
+---+----+----+
| Name|Salary|Category|
+---+----+----+
| Alice| 6000| Other|
| Bob| 4000| Medium|
| Cathy| 2500| Other|
| David| 5000| Medium|
+---+----+----+
```

between()

between()

```
df_filtered = df.filter((df["Salary"].between(3000, 5000)) &  
(df["Name"].startswith("D"))).show()
```

```
+----+-----+  
| Name|Salary|  
+----+-----+  
|David| 5000|  
+----+-----+
```

BETWEEN

```
# Register the DataFrame as a temporary view  
df.createOrReplaceTempView("employees") # Use SQL to filter rows with BETWEEN  
result = spark.sql("SELECT * FROM employees WHERE Salary BETWEEN 3000 AND 5000")  
result.show()
```

```
+----+-----+  
| Name|Salary|  
+----+-----+  
| Bob| 4000|  
|David| 5000|  
+----+-----+
```

```
between()  
BETWEEN 3000 AND 5000
```

```
between()  
  &      |
```

BETWEEN

groupBy() count() count

```
from pyspark.sql import SparkSession from pyspark.sql.functions import col #
Initialize SparkSession spark =
SparkSession.builder.appName("GroupByExample").getOrCreate() # Sample DataFrame
data = [("Alice", "HR"), ("Bob", "IT"), ("Cathy", "HR"), ("David", "Finance"),
("Eve", "HR"), ("Frank", "IT")] columns = [ "Name", "Department"] df =
spark.createDataFrame(data, columns) # Group by Department, count the number of
employees, and rename the count column df_grouped = df.groupBy("Department") \
.count() \ .withColumnRenamed("count", "EmployeeCount") # Sort the results by
EmployeeCount in descending order df_sorted =
df_grouped.orderBy(col("EmployeeCount").desc()) # Show the results
df_sorted.show()
```

```
+-----+-----+
|Department|EmployeeCount|
+-----+-----+
|      HR|          3|
|      IT|          2|
| Finance|          1|
+-----+-----+
```

```
groupBy("Department").count()
```

```
    Department
```

```
withColumnRenamed("count", "EmployeeCount")
```

```
    count
```

```
    EmployeeCount
```

```
orderBy(col("EmployeeCount").desc())
```

```
    EmployeeCount
```

```
alias()
```

```
withColumnRenamed()
```

```
    select()
```

```
alias()
```

```
alias()
```

```
alias()
```

```
from pyspark.sql import SparkSession from pyspark.sql.functions import col #
Initialize SparkSession spark =
SparkSession.builder.appName("GroupByAliasExample").getOrCreate() # Sample
DataFrame data = [("Alice", "HR"), ("Bob", "IT"), ("Cathy", "HR"), ("David",
"Finance"), ("Eve", "HR"), ("Frank", "IT")] columns = ["Name", "Department"]
df = spark.createDataFrame(data, columns) # Group by Department and count the number of
employees, renaming count column using alias df_grouped = df.groupBy("Department")
\ .count() \ .select(col("Department"), col("count").alias("EmployeeCount")) #
```

```
Sort the results by EmployeeCount in descending order df_sorted =  
df_grouped.orderBy(col("EmployeeCount").desc()) # Show the results  
df_sorted.show()
```

```
+---+---+  
|Department|EmployeeCount|  
+---+---+  
|      HR|          3|  
|      IT|          2|  
| Finance|          1|  
+---+---+
```

```
alias()  
col("count").alias("EmployeeCount")  
    count           EmployeeCount  
    select()  
        alias()  
        alias()           select()  
            withColumnRenamed()  
    withColumnRenamed()  
alias()
```

`substr()`

`pyspark.sql.functions`

```
from pyspark.sql import SparkSession from pyspark.sql.functions import col #
Initialize SparkSession spark =
SparkSession.builder.appName("FilterExample").getOrCreate() # Sample DataFrame
data = [("Alice123",), ("Bob4567",), ("Cath789",), ("Dav1234",), ("Eve5678",)]
columns = ["Name"] df = spark.createDataFrame(data, columns) # Filter rows where
the first 4 characters of "Name" are "Bob4" df_filtered =
df.filter(col("Name").substr(1, 4) == "Bob4") # Show the results
df_filtered.show()
```

```
+----+
|  Name|
+----+
|Bob4567|
+----+
```

`substr(1, 4)`

`1`

`4`

`col("column_name").substr(start_pos, length)`

`filter()`

`True`

`"Bob4"`

```
# Filter rows where the first 4 characters are stored in a variable prefix =  
"Cath" df_filtered_dynamic = df.filter(col("Name").substr(1, 4) == prefix)  
df_filtered_dynamic.show()
```

```
+----+  
| Name|  
+----+  
|Cath789|  
+----+
```

like

like

```
# Filter rows where the first 4 characters are "Dav1" df_filtered_like =  
df.filter(col("Name").like("Dav1%")) df_filtered_like.show()
```

```
+----+  
| Name|  
+----+  
|Dav1234|  
+----+
```

substr()

like()

%

substr() like()

IN

when()

~col().isin()

NOT

```
NOT IN
from pyspark.sql import SparkSession from pyspark.sql.functions import when, col #
Initialize SparkSession spark =
SparkSession.builder.appName("NotInConditionExample").getOrCreate() # Sample
DataFrame data = [("Alice", "abc"), ("Bob", "def"), ("Cathy", "xyz"), ("David",
"ghi")] columns = ["Name", "Category"] df = spark.createDataFrame(data, columns) # List of values to exclude exclude_list = ["abc", "xyz"] # Add a new column with "Valid" if Category is NOT in the list, otherwise "Invalid" df_with_condition =
df.withColumn( "Status", when(~col("Category").isin(*exclude_list),
"Valid").otherwise("Invalid") ) # Show the results df_with_condition.show()
```

Name	Category	Status
Alice	abc	Invalid
Bob	def	Valid
Cathy	xyz	Invalid
David	ghi	Valid

```
~col("Category").isin(*exclude_list)
    isin()
  ~
when()

      "Valid"
      "Invalid"
withColumn()
      Status
exclude_list
      NOT
IN
```

```
SELECT Name, Category, CASE WHEN Category NOT IN ('abc', 'xyz') THEN 'Valid' ELSE
'Invalid' END AS Status FROM your_table;
```

```
NOT IN
```

```
NOT IN
```

```
# Filter rows where Category is NOT in the exclude_list df_filtered =
df.filter(~col("Category").isin(*exclude_list)) df_filtered.show()
```

```
+---+---+
| Name|Category|
+---+---+
| Bob|    def|
| David|    ghi|
+---+---+
```

```
isIn()
~
when()      withColumn()
```

NOT IN

```
groupBy()
```

```
groupBy()
```

```
from pyspark.sql import SparkSession from pyspark.sql.functions import col #
Initialize SparkSession spark =
SparkSession.builder.appName("SumColumnsExample").getOrCreate() # Sample DataFrame
data = [("Alice", 10, 20, 30), ("Bob", 5, 15, 25), ("Cathy", 8, 18, 28)] columns =
["Name", "Value1", "Value2", "Value3"] df = spark.createDataFrame(data, columns) #
Add a new column that sums Value1, Value2, and Value3 df_with_sum =
df.withColumn("Total", col("Value1") + col("Value2") + col("Value3")) # Show the
result df_with_sum.show()
```

Name	Value1	Value2	Value3	Total
Alice	10	20	30	60
Bob	5	15	25	45
Cathy	8	18	28	54

```
col("Value1") + col("Value2") + col("Value3")
          col()
```

```
withColumn("Total", ...)
          Total
```

```
groupBy()
```

```
select()
```

```
select()
```

```
df_summed = df.select( col("Name"), col("Value1"), col("Value2"), col("Value3"),
(col("Value1") + col("Value2") + col("Value3")).alias("Total") ) df_summed.show()
```

```
+---+---+---+---+
| Name|Value1|Value2|Value3|Total|
+---+---+---+---+
| Alice|    10|     20|     30|    60|
|   Bob|      5|     15|     25|    45|
| Cathy|      8|     18|     28|    54|
+---+---+---+---+
```

```
withColumn()
```

```
select()           alias()
```

```
SELECT SUM(col1), SUM(col2), SUM(col3) FROM table_a;
```

```
agg()
```

```
from pyspark.sql import SparkSession from pyspark.sql.functions import sum #  
Initialize SparkSession spark =  
SparkSession.builder.appName("SumExample").getOrCreate() # Sample DataFrame  
(table_a) data = [(10, 20, 30), (5, 15, 25), (8, 18, 28)] columns = ["col1",  
"col2", "col3"] df = spark.createDataFrame(data, columns) # Perform the  
aggregation to calculate the sum of each column result_df = df.agg(  
sum("col1").alias("sum_col1"), sum("col2").alias("sum_col2"),  
sum("col3").alias("sum_col3") ) # Show the result result_df.show()
```

```
+-----+-----+-----+  
|sum_col1 |sum_col2 |sum_col3 |  
+-----+-----+-----+  
|      23|      53|      83|  
+-----+-----+-----+
```

```
agg()  
    agg()  
        sum avg count  
    sum("col1").alias("sum_col1")  
        col1  
    sum_col1  
  
    agg()
```

```
# Register the DataFrame as a temporary view df.createOrReplaceTempView("table_a")  
# Use SQL to calculate the sum result_sql = spark.sql("SELECT SUM(col1) AS  
sum_col1, SUM(col2) AS sum_col2, SUM(col3) AS sum_col3 FROM table_a") # Show the  
result result_sql.show()
```

```
groupBy()  
    groupBy()
```

```
from pyspark.sql import SparkSession from pyspark.sql.functions import count #
Initialize SparkSession spark =
SparkSession.builder.appName("GroupByTwoColumns").getOrCreate() # Sample DataFrame
data = [ ("Alice", "HR", "New York"), ("Bob", "HR", "San Francisco"), ("Cathy",
"IT", "New York"), ("David", "IT", "San Francisco"), ("Eve", "HR", "New York") ]
columns = ["Name", "Department", "City"] df = spark.createDataFrame(data, columns)
# Group by Department and City, and count the number of employees result_df =
df.groupBy("Department", "City").agg(count("*").alias("EmployeeCount")) # Show the
result result_df.show()
```

Department	City	EmployeeCount
HR	New York	2
HR	San Francisco	1
IT	New York	1
IT	San Francisco	1

```
groupBy("Department", "City")
agg(count("*").alias("EmployeeCount"))

count("*")
EmployeeCount
```

City

Department

sum avg max min

groupBy()

```
from pyspark.sql.functions import sum, avg # Add an example column for salaries
data = [ ("Alice", "HR", "New York", 5000), ("Bob", "HR", "San Francisco", 6000),
        ("Cathy", "IT", "New York", 7000), ("David", "IT", "San Francisco", 8000), ("Eve",
        "HR", "New York", 5500) ] columns = ["Name", "Department", "City", "Salary"] df =
spark.createDataFrame(data, columns) # Group by Department and City, calculate
total and average salary result_df = df.groupBy("Department", "City").agg(
sum("Salary").alias("TotalSalary"), avg("Salary").alias("AverageSalary") ) # Show
the result result_df.show()
```

Department	City	TotalSalary	AverageSalary
HR	New York	10500	5250.0
HR	San Francisco	6000	6000.0
IT	New York	7000	7000.0
IT	San Francisco	8000	8000.0

groupBy("col1", "col2")

groupBy()

.agg()
count() sum() avg()

```
.alias()
```

[Spark Download](#)

```
conf/spark-env.sh  
export SPARK_MASTER_HOST=<master-node-IP> export  
SPARK_WORKER_CORES=4 # Number of cores for each worker export  
SPARK_WORKER_MEMORY=8G # Memory allocated to each worker
```

```
./sbin/start-master.sh  
spark://<master-node-IP>:7077
```

```
./sbin/start-worker.sh spark://<master-node-IP>:7077
```

```
./bin/pyspark --master spark://<master-node-IP>:7077
```

```
conf/spark-defaults.conf  
spark.master=yarn spark.submit.deployMode=cluster
```

```
./bin/spark-submit \ --master yarn \ --deploy-mode cluster \  
your_script.py
```

```
spark-submit
```

```
pyspark
```

```
spark-submit  
spark-submit your_script.py
```

```
pyspark    spark-submit
```

```
k8s
```

```
./bin/spark-submit \ --master k8s://<kubernetes-cluster-api> \ --  
deploy-mode cluster \ --conf  
spark.kubernetes.container.image=<spark-image> \ your_script.py
```

```
pyspark --master local[*]  
local[*]
```

	Runs on the cluster's master node.	Runs on the client machine (local machine).
	For production workloads in distributed environments.	For development or debugging purposes.

```
spark-submit pyspark
```

```
from pyspark.sql import SparkSession # Initialize SparkSession
spark = SparkSession.builder.appName("DuplicateRowsExample").getOrCreate() # Create a sample DataFrame with 5 columns and some duplicate rows
data = [ (1, "Alice", 25, "HR", 5000), (2, "Bob", 30, "IT", 6000), (3, "Cathy", 29, "Finance", 7000), (4, "David", 35, "IT", 8000), (1, "Alice", 25, "HR", 5000), # Duplicate row (3, "Cathy", 29, "Finance", 7000), # Duplicate row (5, "Eve", 28, "HR", 5500) ]
columns = ["ID", "Name", "Age", "Department", "Salary"] # Create the DataFrame
df = spark.createDataFrame(data, columns) # Show the DataFrame
df.show()
```

```
+---+----+----+-----+----+
| ID| Name|Age|Department|Salary|
+---+----+----+-----+----+
| 1|Alice| 25|      HR| 5000|
| 2| Bob| 30|      IT| 6000|
| 3|Cathy| 29| Finance| 7000|
| 4|David| 35|      IT| 8000|
| 1|Alice| 25|      HR| 5000|      # Duplicate row
| 3|Cathy| 29| Finance| 7000|      # Duplicate row
| 5| Eve| 28|      HR| 5500|
+---+----+----+-----+----+
```

```
(1, "Alice", 25, "HR", 5000)      (3, "Cathy", 29,
"Finance", 7000)
```

ID	Name	Age	Department	Salary
----	------	-----	------------	--------

groupBy() agg()

```
from pyspark.sql import SparkSession from pyspark.sql.functions import count #
Initialize SparkSession spark =
SparkSession.builder.appName("SelectDuplicatesExample").getOrCreate() # Sample
DataFrame with duplicate rows data = [ (1, "Alice", 25, "HR", 5000), (2, "Bob",
30, "IT", 6000), (3, "Cathy", 29, "Finance", 7000), (4, "David", 35, "IT", 8000),
(1, "Alice", 25, "HR", 5000), # Duplicate row (3, "Cathy", 29, "Finance", 7000), #
Duplicate row (5, "Eve", 28, "HR", 5500) ] columns = ["ID", "Name", "Age",
"Department", "Salary"] df = spark.createDataFrame(data, columns) # Group by all
columns and count occurrences duplicate_rows = df.groupBy("ID", "Name", "Age",
"Department", "Salary") \ .agg(count("*").alias("Count")) \ .filter("Count > 1") \
.drop("Count") # Drop the count column if not needed # Show duplicate rows
duplicate_rows.show()
```

```
+---+----+----+-----+----+
| ID| Name|Age|Department|Salary|
+---+----+----+-----+----+
|  1|Alice| 25|        HR|  5000|
|  3|Cathy| 29| Finance|  7000|
+---+----+----+-----+----+
```

groupBy()

agg(count("*").alias("Count"))

```
filter("Count > 1")
```

```
drop("Count")  
Count
```

```
# Include the count column for reference duplicates_with_count = df.groupBy("ID",  
"Name", "Age", "Department", "Salary") \  
.agg(count("*").alias("Count")) \  
.filter("Count > 1") duplicates_with_count.show()
```

ID	Name	Age	Department	Salary	Count
1	Alice	25	HR	5000	2
3	Cathy	29	Finance	7000	2

```
count()
```

```

from pyspark.sql import SparkSession from pyspark.sql.window import Window from
pyspark.sql.functions import count, col # Initialize SparkSession spark =
SparkSession.builder.appName("DuplicatesWithWindow").getOrCreate() # Sample
DataFrame with duplicate rows data = [ (1, "Alice", 25, "HR", 5000), (2, "Bob",
30, "IT", 6000), (3, "Cathy", 29, "Finance", 7000), (4, "David", 35, "IT", 8000),
(1, "Alice", 25, "HR", 5000), # Duplicate row (3, "Cathy", 29, "Finance", 7000), #
Duplicate row (5, "Eve", 28, "HR", 5500) ] columns = ["ID", "Name", "Age",
"Department", "Salary"] df = spark.createDataFrame(data, columns) # Define a
window specification to partition by all columns window_spec =
Window.partitionBy("ID", "Name", "Age", "Department", "Salary") # Add a count
column to count duplicates df_with_count = df.withColumn("Count",
count("*").over(window_spec)) # Filter rows where Count > 1 (duplicates)
duplicate_rows = df_with_count.filter(col("Count") > 1).drop("Count") # Show the
duplicate rows duplicate_rows.show()

```

```

+---+-----+-----+-----+
| ID| Name|Age|Department|Salary|
+---+-----+-----+-----+
| 1|Alice| 25|        HR| 5000|
| 1|Alice| 25|        HR| 5000|
| 3|Cathy| 29|    Finance| 7000|
| 3|Cathy| 29|    Finance| 7000|
+---+-----+-----+-----+

```

`Window.partitionBy()`

ID	Name	Age	Department	Salary
----	------	-----	------------	--------

`count("*").over(window_spec)`

`filter(col("Count") > 1)`

`drop("Count")`

Count

groupBy()

	groupBy()	
	Aggregates the entire group into a single result.	Preserves all rows, even duplicates.
	Only shows distinct rows with their count.	Shows each duplicate row multiple times.
When you need a compact summary of duplicates.		When you need to retain all duplicate rows.

dropDuplicates()

```
distinct_duplicates = duplicate_rows.dropDuplicates() distinct_duplicates.show()
```

```
+---+-----+-----+-----+
| ID| Name|Age|Department|Salary|
+---+-----+-----+-----+
| 1|Alice| 25|        HR|  5000|
| 3|Cathy| 29|    Finance|  7000|
+---+-----+-----+-----+
```

```
spark.read.json()
```

```
from pyspark.sql import SparkSession # Initialize SparkSession
spark = SparkSession.builder.appName("JSONExample").getOrCreate() # Read JSON file
df = spark.read.json("path/to/jsonfile.json") # Show the DataFrame
df.show()
```

```
col("nestedField.subField")      selectExpr()
```

```
{ "id": 1, "name": "Alice", "address": { "city": "New York", "zip": 10001 } }
```

```
from pyspark.sql.functions import col # Read JSON file
df = spark.read.json("path/to/nestedfile.json") # Access nested fields
df.select(col("id"), col("name"), col("address.city").alias("city"),
          col("address.zip").alias("zip")).show()
```

```
write.json()
```

```
df.write.json("path/to/output.json")
```

```
multiline
```

```
True
```

```
df = spark.read.option("multiline", "true").json("path/to/multilinefile.json")
df.show()
```

```
filter()    where()
```

```
df_filtered = df.filter(df["age"] > 30) df_filtered.show()
```

```
spark.read.json()
```

```
# Sample JSON string json_data = ['{"id":1,"name":"Alice","age":25}', 
'{"id":2,"name":"Bob","age":30}'] # Create an RDD of JSON strings rdd =
spark.sparkContext.parallelize(json_data) # Read the JSON string as a DataFrame df
= spark.read.json(rdd) df.show()
```

```
select()
```

```
# Flatten a nested structure df_flat = df.select( col("id"), col("name"),
col("address.city").alias("city"), col("address.zip").alias("zip") )
df_flat.show()
```

```
explode()
```

```
{ "id": 1, "name": "Alice", "hobbies": ["reading", "swimming"] }
```

```
from pyspark.sql.functions import explode # Explode the array field df_exploded =
df.select("id", "name", explode(col("hobbies")).alias("hobby"))
df_exploded.show()
```

```
+---+----+-----+
| id| name|    hobby|
+---+----+-----+
|  1|Alice| reading|
|  1|Alice| swimming|
+---+----+-----+
```

```
read.json()
```

```
df = spark.read.json("path/to/json/files/") df.show()
```

StructType StructField

```
from pyspark.sql.types import StructType, StructField, StringType, IntegerType #  
Define schema schema = StructType([ StructField("id", IntegerType(), True),  
StructField("name", StringType(), True), StructField("age", IntegerType(), True)  
]) # Read JSON with schema df =  
spark.read.schema(schema).json("path/to/jsonfile.json") df.show()
```

```
toJSON()
```

```
json_rdd = df.toJSON() json_rdd.collect() # Collect JSON strings as a list
```

badRecordsPath mode

```
mode="PERMISSIVE"
    corrupt_record
mode="DROPMALFORMED"
mode="FAILFAST"
```

```
df = spark.read.option("mode", "DROPMALFORMED").json("path/to/malformed.json")
df.show()
```

```
from_json
```

```
from_json
```

```
from_json
```

```
from_json
```

```
StructType          from_json
```

```
from pyspark.sql import SparkSession from pyspark.sql.functions import from_json
from pyspark.sql.types import StructType, StructField, StringType, IntegerType # Initialize SparkSession
spark = SparkSession.builder.appName("FromJsonExample").getOrCreate() # Sample DataFrame
with JSON string column data = [ ("1", '{"name": "Alice", "age": 25}'), ("2", '{"name": "Bob", "age": 30}') ] columns = ["ID", "json_data"] df =
spark.createDataFrame(data, columns) # Define the schema
schema = StructType([
StructField("name", StringType(), True), StructField("age", IntegerType(), True)
]) # Parse the JSON string column into a structured column
df_parsed = df.withColumn("parsed_data", from_json(df["json_data"], schema))
df_parsed.show(truncate=False)
```

```
+---+-----+-----+
| ID|json_data           |parsed_data      |
+---+-----+-----+
| 1 |{"name": "Alice", "age": 25}|{Alice, 25}       |
| 2 |{"name": "Bob", "age": 30}|{Bob, 30}       |
+---+-----+-----+
```

```
parsed_data.field    col("parsed_data.field")
```

```
df_extracted = df_parsed.select( "ID", "json_data",
col("parsed_data.name").alias("name"), col("parsed_data.age").alias("age") )
df_extracted.show()
```

```
+---+-----+-----+
| ID|      json_data| name|age|
+---+-----+-----+
| 1 |{"name": "Alice",...|Alice| 25|
| 2 |{"name": "Bob", "...| Bob| 30|
+---+-----+-----+
```

from_json

null

```
# Sample DataFrame with malformed JSON data = [("1", '{"name": "Alice", "age": 25}), ("2", '{"name": "Bob", age: 30}')] df_malformed = spark.createDataFrame(data, columns) df_parsed_malformed = df_malformed.withColumn("parsed_data", from_json(df_malformed["json_data"], schema)) # Filter out malformed rows df_valid = df_parsed_malformed.filter(df_parsed_malformed["parsed_data"].isNotNull()) df_valid.show(truncate=False)
```

ID	json_data	parsed_data
1	{"name": "Alice", "age": 25}	{Alice, 25}

from_json

StructType

ArrayType

```
from pyspark.sql.types import ArrayType # Sample DataFrame with JSON array data = [("1", '[{"name": "Alice", "age": 25}, {"name": "Bob", "age": 30}')] df_array = spark.createDataFrame(data, ["ID", "json_data"]) # Define schema for JSON array array_schema = ArrayType(StructType([ StructField("name", StringType(), True), StructField("age", IntegerType(), True) ])) # Parse JSON array into a structured column df_parsed_array = df_array.withColumn("parsed_data", from_json(df_array["json_data"], array_schema)) df_parsed_array.show(truncate=False)
```

ID	json_data	parsed_data
1	[{"name": "Alice", "age": 25}, {"name": "Bob", "age": 30}]	[{Alice, 25}, {Bob, 30}]

explode()

```
from pyspark.sql.functions import explode # Explode the parsed JSON array
df_exploded = df_parsed_array.withColumn("exploded_data",
    explode(col("parsed_data"))). # Extract fields from the exploded rows
df_exploded.select( "ID", col("exploded_data.name").alias("name"),
    col("exploded_data.age").alias("age") ) df_exploded.show()
```

```
+---+---+---+
| ID| name|age|
+---+---+---+
| 1|Alice| 25|
| 1| Bob| 30|
+---+---+---+
```

from_json

null

```
# JSON data with missing fields
data = [{"1": {"name": "Alice"}}, {"2": {"name": "Bob", "age": 30}}]
df_missing = spark.createDataFrame(data, columns)
df_parsed_missing = df_missing.withColumn("parsed_data",
from_json(df_missing["json_data"], schema)) df_parsed_missing.show(truncate=False)
```

```
+---+-----+-----+
| ID|json_data |parsed_data |
+---+-----+-----+
| 1|{"name": "Alice"} |{Alice, null} |
| 2|{"name": "Bob", "age": 30}|{Bob, 30} |
+---+-----+-----+
```

to_json

```
from pyspark.sql.functions import to_json, struct # Convert structured fields back  
to JSON df_to_json = df_parsed.withColumn("json_back",  
to_json(struct("parsed_data.*"))).df_to_json.show(truncate=False)
```

ID	json_data	parsed_data	json_back
1	{"name": "Alice", "age": 25}	{Alice, 25}	{"name": "Alice", "age": 25}
2	{"name": "Bob", "age": 30}	{Bob, 30}	{"name": "Bob", "age": 30}

from_json

mode

mode

PERMISSIVE	Default mode. Puts corrupt records in a special column (e.g., <code>_corrupt_record</code>).
DROPMALFORMED	Drops all rows that contain corrupt records.
FAILFAST	Fails immediately if corrupt records are encountered.

PERMISSIVE

```
from pyspark.sql import SparkSession # Initialize SparkSession
spark = SparkSession.builder.appName("HandleCorruptedData").getOrCreate() # Sample JSON
file with corrupted data
data = [ '{"name": "Alice", "age": 25}', '{"name": "Bob", "age": 30}', '{"name": "Cathy", "age": "invalid"}', # Corrupted record
'{"name": "David"}' # Missing field ]
# Save as a file with open("sample.json", "w") as f:
f.write("\n".join(data)) # Read JSON file with `PERMISSIVE` mode
df = spark.read.option("mode", "PERMISSIVE").option("columnNameOfCorruptRecord",
"_corrupt_record").json("sample.json")
df.show(truncate=False)
```

```
+---+---+-----+
| name| age| _corrupt_record |
+---+---+-----+
| Alice| 25| null
| Bob| 30| null
| null| null| {"name": "Cathy", "age": "invalid"} |
| David| null| null
+---+---+-----+
```

`_corrupt_record`

`_corrupt_record`

```
valid_records = df.filter(df["_corrupt_record"].isNull())
valid_records.show()
```

```
+---+---+-----+
| name| age| _corrupt_record |
```

```
+---+---+---+
|Alice| 25|null|
| Bob| 30|null|
|David|null|null|
+---+---+---+
```

DROPMALFORMED

DROPMALFORMED

```
df_dropped = spark.read.option("mode", "DROPMALFORMED").json("sample.json")
df_dropped.show()
```

```
+---+---+
| name| age|
+---+---+
|Alice| 25|
| Bob| 30|
+---+---+
```

FAILFAST

FAILFAST

```
try: df_failfast = spark.read.option("mode", "FAILFAST").json("sample.json")
df_failfast.show() except Exception as e: print(f"Job failed: {e}")
```

```
Job failed: java.lang.RuntimeException: Malformed records are detected.
```

```
PERMISSIVE DROPMALFORMED FAILFAST
```

```
df_csv = spark.read.option("mode", "PERMISSIVE").option("header",  
"true").csv("sample.csv") df_csv.show()
```

```
from pyspark.sql.types import StructType, StructField, StringType, IntegerType #  
Define schema schema = StructType([ StructField("name", StringType(), True),  
StructField("age", IntegerType(), True) ]) # Read JSON with schema df_schema =  
spark.read.schema(schema).json("sample.json") df_schema.show()
```

```
corrupt_records = df.filter(df["_corrupt_record"].isNotNull())  
corrupt_records.write.json("path/to/save/corrupt_records")
```

```
from pyspark.sql.functions import udf from pyspark.sql.types import IntegerType #  
UDF to handle invalid age def validate_age(age): try: return int(age) except:  
return None validate_age_udf = udf(validate_age, IntegerType()) # Apply UDF  
df_cleaned = df.withColumn("age", validate_age_udf(df["age"])) df_cleaned.show()
```

<code>mode="PERMISSIVE"</code>	Keeps corrupt records in a special column (default behavior).
<code>mode="DROPMALFORMED"</code>	Drops all rows with corrupted data.
<code>mode="FAILFAST"</code>	Fails the job immediately on encountering corrupted data.
	Ensures that only records matching the schema are loaded.
<code>_corrupt_record</code>	Allows filtering out or logging invalid records for debugging or recovery.
	Custom logic to validate and clean data programmatically.

```
from pyspark.sql import SparkSession from pyspark.sql.functions import collect_list from pyspark.sql.window import Window # Initialize SparkSession spark = SparkSession.builder.appName("CumulativeListExample").getOrCreate() # Input DataFrame data = [(1, "Alice"), (2, "Bob"), (3, "Cathy"), (4, "David")] columns = ["ID", "Name"] df = spark.createDataFrame(data, columns) # Define a window specification to collect cumulative names window_spec = Window.orderBy("ID").rowsBetween(Window.unboundedPreceding, Window.currentRow) # Add the cumulative names as a new column df_with_cumulative = df.withColumn("Combine_name", collect_list("Name").over(window_spec)) # Show the result df_with_cumulative.show(truncate=False)
```

ID	Name	Combine_name
1	Alice	[Alice]
2	Bob	[Alice, Bob]
3	Cathy	[Alice, Bob, Cathy]
4	David	[Alice, Bob, Cathy, David]

```
Window.orderBy("ID").rowsBetween(Window.unboundedPreceding,  
Window.currentRow)
```

ID

```
Window.unboundedPreceding  
Window.currentRow
```

Combine_name

ID

Combine_name

repartition()

coalesce()

	<code>repartition(n)</code>	<code>coalesce(n)</code>
	Yes (full shuffle)	No (avoids full shuffle)
	Expensive due to data movement	Optimized for reducing partitions
When increasing partitions or balancing skew		When decreasing partitions efficiently

```
df_repartitioned = df.repartition(5) # Increases partitions (full shuffle)
df_coalesced = df.coalesce(2) # Decreases partitions (avoids full shuffle)
```

```
lookup_dict = {"HR": "Human Resources", "IT": "Information Technology"}
broadcast_lookup = spark.sparkContext.broadcast(lookup_dict)
```

```
error_count = spark.sparkContext.accumulator(0)
```

`cache()` `persist()`

	<code>cache()</code>	<code>persist(StorageLevel)</code>
	Stores in	Can store in
	No control over storage level	Allows different storage levels (e.g., <code>MEMORY_AND_DISK</code>)
	When you need fast access to frequently used data	When data is too large to fit in memory

```
df.cache() # Stores in memory
df.persist(storageLevel=StorageLevel.MEMORY_AND_DISK) # Stores in memory & disk
```

repartition()
broadcast()

cache()
persist()

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

repartition()

broadcast()

```
spark.conf.set("spark.sql.adaptive.skewJoin.enabled", "true")
```

```
rank() dense_rank() row_number()
```

```
from pyspark.sql.window import Window from pyspark.sql.functions import sum
window_spec =
Window.partitionBy("Department").orderBy("Salary").rowsBetween(Window.unboundedPreceding, Window.currentRow) df.withColumn("CumulativeSalary",
sum("Salary").over(window_spec)).show()
```

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

```
df.write.json("output.json")
```

```
df.select("id", "name", col("address.city"), col("address.zip")).show()
```

```
explode()
```

```
from pyspark.sql.functions import explode df = spark.createDataFrame([(1, ["a", "b", "c"]), (2, ["d", "e"])], ["ID", "Values"])
df_exploded = df.withColumn("ExplodedValues", explode("Values"))
df_exploded.show()
```

ID	Values	ExplodedValues
1	[a, b, c]	a
1	[a, b, c]	b

```
| 1 | [a, b, c] | c
| 2 | [d, e]   | d
| 2 | [d, e]   | e
+---+-----+-----+
```

```
from pyspark.sql.functions import broadcast
df_large.join(broadcast(df_small), "key")

spark.conf.set("spark.sql.adaptive.skewJoin.enabled", "true")

df.write.bucketBy(4, "key").saveAsTable("bucketed_table")
```

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

mode="PERMISSIVE"

DROPMALFORMED

FAILFAST

```
df = spark.read.option("mode", "PERMISSIVE").json("corrupted_data.json")
```

```
_corrupt_record
```

```
df_valid = df.filter(df["_corrupt_record"].isNull())
```

```
explain()
```

```
cache() broadcast()
```

```
spark.eventLog.enabled
```

```
from pyspark.sql.functions import broadcast df_result =  
df_large.join(broadcast(df_small), "customer_id")
```

```
df_large.write.bucketBy(10, "customer_id").saveAsTable("large_table")  
df_small.write.bucketBy(10, "customer_id").saveAsTable("small_table")
```

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

```
from pyspark.sql.functions import monotonically_increasing_id, expr  
df_skewed = df_skewed.withColumn("salt", (monotonically_increasing_id() %  
10)) df_large = df_large.withColumn("salt", expr("floor(rand() * 10)"))  
df_result = df_skewed.join(df_large, ["customer_id", "salt"])
```

```
spark.conf.set("spark.sql.adaptive.skewJoin.enabled", "true")
```

```
{ "id": 1, "name": "Alice", "address": { "city": "New York", "zip": 10001 },  
"orders": [ {"order_id": "A123", "amount": 500}, {"order_id": "B456", "amount":  
1000} ] }
```

```
from pyspark.sql.functions import col df_flattened = df.select( col("id"),  
col("name"), col("address.city").alias("city"),  
col("address.zip").alias("zip") ) df_flattened.show()  
|-----|  
| id | name | city | zip |  
|---|---|---|---|  
| 1 | Alice | New York | 10001 |  
  
from pyspark.sql.functions import explode df_exploded =  
df.withColumn("orders", explode("orders")) df_orders =  
df_exploded.select("id", "name", "orders.order_id", "orders.amount")  
df_orders.show()
```

```
df_new_data = df.filter(col("timestamp") >= "2024-01-01")
```

```
from delta import DeltaTable delta_table = DeltaTable.forName(spark,  
"path/to/delta_table") df_new_data =  
spark.read.format("delta").load().filter("timestamp >= '2024-01-01'")
```

```
df.write.partitionBy("year", "month", "day").parquet("data/")
```

```
df.explain(True) # Check query plan
```

```
df = df.repartition(100) # Increase for better parallelism
```

```
from pyspark.sql.functions import upper df = df.withColumn("upper_name",  
upper(col("name"))) # Faster than a Python UDF
```

```
df.cache()
```

```
from pyspark.sql.functions import count df.groupBy("id",  
"name").agg(count("*").alias("count")).filter(col("count") > 1).show()
```

```
df = df.dropDuplicates()
```

```
df.select([count(when(col(c).isNull(), c)).alias(c) for c in df.columns]).show()
```

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

```
df_filled = df.na.fill({"age": 0, "name": "Unknown"})
```

```
df.coalesce(10).write.parquet("output/")
```

```
df.write.partitionBy("year", "month").parquet("output/")
```

```
from delta.tables import DeltaTable from pyspark.sql.functions import
current_date # Load existing data delta_table = DeltaTable.forPath(spark,
"path/to/delta_table") # Incoming new records df_new =
spark.read.parquet("new_data.parquet") # Perform Merge for SCD2
delta_table.alias("old").merge( df_new.alias("new"), "old.customer_id =
new.customer_id AND old.end_date IS NULL" ).whenMatchedUpdate(
set={"end_date": current_date()}).whenNotMatchedInsert(
values={"customer_id": "new.customer_id", "start_date": current_date(),
"end_date": None}).execute()
```

```
from pyspark.sql.types import StringType df_stream = spark.readStream \
    .format("kafka") \ .option("kafka.bootstrap.servers", "localhost:9092") \ \
    .option("subscribe", "topic_name") \ .load() df_transformed = \
df_stream.selectExpr("CAST(value AS \
STRING)").select(col("value").cast(StringType()))
```

```
df_transformed.writeStream \ .format("delta") \ \
.option("checkpointLocation", "path/to/checkpoint") \ \
.start("path/to/delta_table")
```

```
df.explain(True) # Check query plan for shuffles
```

```
df = df.repartition(100)
```

```
df = df.coalesce(10)
```

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

```
df.write.option("mergeSchema",  
"true").format("delta").save("path/to/delta_table")
```

```
df = spark.read.option("inferSchema", "true").json("data.json")
```

```
df1 = df1.withColumn("new_column", lit(None).cast(StringType())) df_final =  
df1.union(df2) # Align schema before merging
```

```
badRecordsPath  
df = spark.read.option("badRecordsPath",  
"path/to/bad_records").csv("large_data.csv")
```

```
from pyspark.sql.functions import when, col df_cleaned =  
df.withColumn("age", when(col("age").cast("int").isNotNull(),  
col("age")).otherwise(None))
```

```
from pyspark.sql.types import IntegerType df_valid =  
df.filter(col("age").cast(IntegerType()).isNotNull())
```

```
df.groupBy("customer_id").count().filter("count > 1").show()
```

```
df = df.dropDuplicates()
```

```
from pyspark.sql.window import Window from pyspark.sql.functions import  
row_number window_spec =  
Window.partitionBy("customer_id").orderBy(col("timestamp").desc())  
df_latest = df.withColumn("row_num",  
row_number().over(window_spec)).filter("row_num = 1").drop("row_num")
```

```
df = spark.read.parquet("data/").filter("year = 2023 AND month = 5")
```

```
df.write.partitionBy("year", "month").parquet("output/")
```

```
spark.conf.set("hive.exec.dynamic.partition", "true")  
spark.conf.set("hive.exec.dynamic.partition.mode", "nonstrict")
```

```
from pyspark.sql.types import StructType, StructField, DoubleType,  
StringType from pyspark.sql.functions import window, avg schema =
```

```
StructType([ StructField("transaction_id", StringType(), True),  
StructField("amount", DoubleType(), True), StructField("timestamp",  
StringType(), True) ]) df_stream = spark.readStream \\.format("kafka") \\  
.option("kafka.bootstrap.servers", "localhost:9092") \\.option("subscribe",  
"transactions") \\.load() \\.selectExpr("CAST(value AS STRING) as json") \\  
.selectExpr("from_json(json, schema) as data") \\.select("data.*")
```

```
from pyspark.sql.functions import col windowed_avg = df_stream \\  
.groupBy(window(col("timestamp"), "5 minutes")) \\  
.agg(avg("amount").alias("avg_amount")) df_anomalies =  
df_stream.join(windowed_avg, "window") \\.filter(df_stream["amount"] >  
windowed_avg["avg_amount"])
```

```
df_anomalies.selectExpr("CAST(transaction_id AS STRING) AS key",  
"to_json(struct(*)) AS value") \\.writeStream \\.format("kafka") \\  
.option("kafka.bootstrap.servers", "localhost:9092") \\.option("topic",  
"anomalies") \\.option("checkpointLocation", "path/to/checkpoint") \\  
.start()
```

```
df = spark.read.parquet("s3://my-bucket/data/").filter("year = 2023 AND  
month = 5")
```

```
df.coalesce() df.coalesce(50).write.parquet("output/")
```

```
spark.conf.set("spark.sql.parquet.filterPushdown", "true")
```

```
df = spark.read.parquet("data/").select("customer_id", "amount")
```

```
groupBy("user_id").sum("amount")
```

```
from pyspark.sql.functions import expr, rand df = df.withColumn("salt",
expr("floor(rand() * 10)")) df_grouped = df.groupBy("user_id",
"salt").sum("amount").groupBy("user_id").sum("sum(amount)")
```

```
spark.conf.set("spark.sql.adaptive.skewJoin.enabled", "true")
```

```
from delta.tables import DeltaTable from pyspark.sql.functions import
current_timestamp delta_table = DeltaTable.forPath(spark, "s3://my-delta-
table") df_updates =
spark.read.parquet("new_data.parquet").withColumn("updated_at",
current_timestamp()) delta_table.alias("old").merge(
df_updates.alias("new"), "old.id = new.id"
).whenMatchedUpdate(set={"old.amount": "new.amount", "old.updated_at": "
new.updated_at"}) \ .whenNotMatchedInsert(values={"id": "new.id",
"amount": "new.amount", "updated_at": "new.updated_at"}) \ .execute()
```

```
java.lang.OutOfMemoryError
```

```
--executor-memory 8G --driver-memory 4G
df.explain(True)
df = df.coalesce(10) # Reduce memory pressure on executors

spark.conf.set("spark.sql.shuffle.partitions", "200")
    .persist(StorageLevel.DISK_ONLY)    cache()
from pyspark import StorageLevel df.persist(StorageLevel.DISK_ONLY)
```

```
from airflow import DAG from
airflow.providers.apache.spark.operators.spark_submit import
SparkSubmitOperator from datetime import datetime default_args = {"owner": "airflow", "start_date": datetime(2024, 1, 1)} dag = DAG("daily_spark_job",
default_args=default_args, schedule_interval="@daily") spark_task =
SparkSubmitOperator( task_id="run_spark_job",
application="s3://path/to/job.py", conn_id="spark_default", dag=dag )

spark.conf.set("spark.eventLog.enabled", "true")
```

```
from pyspark.sql.functions import mean, stddev
stats = df.select(mean("amount").alias("mean"),
                  stddev("amount").alias("stddev")).collect()[0]
df_filtered = df.filter(((df["amount"] - stats["mean"]) / stats["stddev"]).between(-3, 3))
df_filtered.show()
```

r5d i3

```
spark.conf.set("spark.dynamicAllocation.enabled", "true")
```

```
spark.read.parquet("s3://my-bucket/data/")
```

```
spark.read.option("fs.s3.select.enabled", "true").parquet("s3://my-  
bucket/data/")
```

```
spark.conf.set("fs.s3.consistent", "true")
```

```
df.write.format("parquet").save("gs://my-bucket/output/")
```

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

```
df_stream = spark.readStream \ .format("kafka") \  
.option("kafka.bootstrap.servers", "localhost:9092") \ .option("subscribe",  
"iot_sensor_data") \ .load()
```

```
from pyspark.sql.functions import col, from_json from pyspark.sql.types  
import StructType, StringType, DoubleType schema =  
StructType().add("sensor_id", StringType()).add("temperature",  
DoubleType()) df_parsed =  
df_stream.select(from_json(col("value").cast("string"),  
schema).alias("data")).select("data.*")
```

```
df_parsed.writeStream \ .format("delta") \ .outputMode("append") \  
.option("checkpointLocation", "s3://path/to/checkpoint") \  
.start("s3://path/to/delta_table")
```

```
from pyspark.ml.feature import VectorAssembler assembler =  
VectorAssembler(inputCols=["transaction_amount", "transaction_frequency"],  
outputCol="features") df_ml = assembler.transform(df).select("features",  
"label")
```

```
from pyspark.ml.classification import LogisticRegression lr =  
LogisticRegression(featuresCol="features", labelCol="label") model =  
lr.fit(df_ml)
```

```
predictions = model.transform(df_ml) predictions.show()
```

```
java.lang.OutOfMemoryError
```

```
--executor-memory 8G --driver-memory 4G
```

```
spark.conf.set("spark.memory.offHeap.enabled", "true")  
spark.conf.set("spark.memory.offHeap.size", "4g")
```

```
from pyspark import StorageLevel df.persist(StorageLevel.DISK_ONLY)
```

```
spark.conf.set("spark.sql.shuffle.partitions", "200")
```

```
df_new_data =  
spark.read.format("delta").load("s3://delta_table").filter("date =  
current_date()")
```

```
df_stream.withWatermark("event_time", "1  
day").dropDuplicates(["transaction_id"])
```

```
coalesce()  
df.coalesce(10).write.parquet("s3://my-bucket/data/")  
  
spark.conf.set("spark.databricks.delta.optimizeWrite.enabled", "true")  
spark.conf.set("spark.databricks.delta.autoCompact.enabled", "true")
```

```
df.groupBy("customer_id").count().filter("count > 1").show()  
  
df = df.dropDuplicates()  
  
from pyspark.sql.window import Window from pyspark.sql.functions import  
row_number window_spec =  
Window.partitionBy("customer_id").orderBy(col("timestamp").desc())  
df_latest = df.withColumn("row_num",  
row_number().over(window_spec)).filter("row_num = 1").drop("row_num")
```

```
count()      when()
```

```
from pyspark.sql import SparkSession from pyspark.sql.functions import col, when, count # Initialize SparkSession spark = SparkSession.builder.appName("CountNulls").getOrCreate() # Sample DataFrame with NULL values data = [ (1, "Alice", None, 5000), (2, "Bob", "IT", None), (3, None, "Finance", 7000), (4, "David", "HR", 8000), (5, None, None, None) ] columns = ["ID", "Name", "Department", "Salary"] df = spark.createDataFrame(data, columns) # Count NULL values in each column null_counts = df.select([count(when(col(c).isNull(), c)).alias(c) for c in df.columns]) # Show the result null_counts.show()
```

ID	Name	Department	Salary
0	2	2	2

```
Name  
Department
```

Salary
ID

```
when(col(c).isNull(), c)  
count()  
alias(c) NULL
```

```
df.select( [count(when(col(c).isNull(), c)).alias(f"{c}_nulls") for c in  
df.columns] + [count(when(col(c).isNotNull(), c)).alias(f"{c}_non_nulls") for c in  
df.columns] ).show()
```

```
from pyspark.sql.functions import round total_rows = df.count() df.select([  
round((count(when(col(c).isNull(), c)) / total_rows) * 100,  
2).alias(f"{c}_null_percent") for c in df.columns ]).show()
```

	count(when(col(c).isNull(), c))
	count(when(col(c).isNotNull(), c))
	(count(when(col(c).isNull(), c)) / total_rows) * 100

```
from pyspark.sql import SparkSession from pyspark.sql.functions import col, when, count # Initialize SparkSession spark = SparkSession.builder.appName("CountNulls").getOrCreate() # Sample DataFrame with NULL values data = [ (1, "Alice", None, 5000), (2, "Bob", "IT", None), (3, None, "Finance", 7000), (4, "David", "HR", 8000), (5, None, None, None) ] columns = ["ID", "Name", "Department", "Salary"] df = spark.createDataFrame(data, columns) # Specify the columns to check for null values columns_to_check = ["Name", "Salary"] # Count NULL values in the specified columns null_counts = df.select([count(when(col(c).isNull(), c)).alias(c) for c in columns_to_check]) # Show the result null_counts.show()
```

Name	Salary
2	2

Name

Salary

```
df.select( [count(when(col(c).isNull(), c)).alias(f"{c}_nulls") for c in columns_to_check] + [count(when(col(c).isNotNull(), c)).alias(f"{c}_non_nulls") for c in columns_to_check] ).show()
```

```
+-----+-----+-----+
|Name_nulls|Salary_nulls|Name_non_nulls|Salary_non_nulls|
+-----+-----+-----+
|    2    |    2    |     3     |     3     |
+-----+-----+-----+
```

```
from pyspark.sql.functions import round total_rows = df.count() df.select([
round((count(when(col(c).isNull(), c)) / total_rows) * 100,
2).alias(f"{c}_null_percent") for c in columns_to_check ]).show()
```

```
+-----+-----+
|Name_null_percent|Salary_null_percent|
+-----+-----+
|      40.0       |      40.0       |
+-----+-----+
```

40.0% Name
40.0% Salary

	count(when(col(c).isNull(), c))
	count(when(col(c).isNotNull(), c))
(count(when(col(c).isNull(), c)) / total_rows) * 100	

WHEN SUM() IS NULL COUNT() CASE

```
COUNT(CASE WHEN column IS NULL THEN 1 END)
```

```
SELECT COUNT(CASE WHEN Name IS NULL THEN 1 END) AS Name_Null_Count, COUNT(CASE WHEN Department IS NULL THEN 1 END) AS Department_Null_Count, COUNT(CASE WHEN Salary IS NULL THEN 1 END) AS Salary_Null_Count FROM employees;
```

Name_Null_Count	Department_Null_Count	Salary_Null_Count
2	2	2

SUM()

```
SELECT SUM(CASE WHEN Name IS NULL THEN 1 ELSE 0 END) AS Name_Null_Count, SUM(CASE WHEN Department IS NULL THEN 1 ELSE 0 END) AS Department_Null_Count, SUM(CASE WHEN Salary IS NULL THEN 1 ELSE 0 END) AS Salary_Null_Count FROM employees;
```

COUNT(CASE WHEN...)

```
SELECT COUNT(CASE WHEN Name IS NULL THEN 1 END) AS Name_Null_Count, COUNT(CASE WHEN Name IS NOT NULL THEN 1 END) AS Name_NonNull_Count, COUNT(CASE WHEN Department IS NULL THEN 1 END) AS Department_Null_Count, COUNT(CASE WHEN Department IS NOT NULL THEN 1 END) AS Department_NonNull_Count, COUNT(CASE WHEN Salary IS NULL THEN 1 END) AS Salary_Null_Count, COUNT(CASE WHEN Salary IS NOT NULL THEN 1 END) AS Salary_NonNull_Count FROM employees;
```

Name_Null_Count	Name_NonNull_Count	Department_Null_Count	Department_NonNull_Count	Salary_Null_Count	Salary_NonNull_Count
2	3	2	3		

```
SELECT ROUND((COUNT(CASE WHEN Name IS NULL THEN 1 END) * 100.0) / COUNT(*), 2) AS Name_Null_Percent, ROUND((COUNT(CASE WHEN Department IS NULL THEN 1 END) * 100.0) / COUNT(*), 2) AS Department_Null_Percent, ROUND((COUNT(CASE WHEN Salary IS NULL THEN 1 END) * 100.0) / COUNT(*), 2) AS Salary_Null_Percent FROM employees;
```

```
+-----+-----+-----+
| Name_Null_Percent | Department_Null_Percent | Salary_Null_Percent |
+-----+-----+-----+
|      40.0          |        40.0          |       40.0          |
+-----+-----+-----+
```

INFORMATION_SCHEMA.COLUMNS

```
SELECT COLUMN_NAME, SUM(CASE WHEN COLUMN_NAME IS NULL THEN 1 ELSE 0 END) AS  
Null_Count FROM employees GROUP BY COLUMN_NAME;
```

	COUNT(CASE WHEN col IS NULL THEN 1 END)
	COUNT(CASE WHEN col IS NULL THEN 1 END), COUNT(CASE WHEN col IS NOT NULL THEN 1 END)
	ROUND((COUNT(CASE WHEN col IS NULL THEN 1 END) * 100.0) / COUNT(*), 2)
Use INFORMATION_SCHEMA.COLUMNS	

```
pivot()
```

```
NULL
```

	Sales	Jan	Feb	Mar
Mar				NULL

```
from pyspark.sql import SparkSession from pyspark.sql.functions import sum #  
Initialize SparkSession spark =  
SparkSession.builder.appName("PivotIssue").getOrCreate() # Sample DataFrame (No  
NULL values) data = [ ("Alice", "Jan", 100), ("Alice", "Feb", 200), ("Bob", "Jan",  
150), ("Bob", "Mar", 300) # No "Feb" sales for Bob ] columns = [ "Name", "Month",  
"Sales" ] df = spark.createDataFrame(data, columns) df.show()
```

Name	Month	Sales
Alice	Jan	100
Alice	Feb	200
Bob	Jan	150
Bob	Mar	300

```
df_pivot = df.groupBy("Name").pivot("Month").agg(sum("Sales")) df_pivot.show()
```

|--|--|--|--|

```
+---+---+---+---+
| Name|Jan|Feb|Mar |
+---+---+---+---+
| Alice|100|200|NULL|
| Bob |150|NULL|300|
+---+---+---+---+
```



```
fillna()
```

```
df_pivot.fillna(0).show()
```

```
+---+---+---+---+
| Name|Jan|Feb|Mar |
+---+---+---+---+
| Alice|100|200| 0|
| Bob |150| 0|300|
+---+---+---+---+
```

```
values
```

```
.pivot(column, values)
```

```
df_pivot_fixed = df.groupBy("Name").pivot("Month", ["Jan", "Feb",
"Mar"]).agg(sum("Sales")) df_pivot_fixed.show()
```

```
Jan Feb Mar
```

```
fillna()
```

```
coalesce()
```

```
from pyspark.sql.functions import coalesce df_pivot_no_nulls = df.groupBy("Name").pivot("Month").agg(coalesce(sum("Sales"), 0)) df_pivot_no_nulls.show()
```

	Replace NULLs with <code>fillna(0)</code>	<code>df_pivot.fillna(0)</code>
	Define all possible pivot values	<code>df.pivot("Month", ["Jan", "Feb", "Mar"])</code>
Use <code>coalesce()</code> in aggregation		<code>agg(coalesce(sum("Sales"), 0))</code>

```
pivot('a', ['b', 'c'])
```

```
pivot(column, values)
```

```
df.groupBy("some_column").pivot("a", ["b", "c"]).agg(...)
```

a
["b", "c"]

```
    pivot('a', ['b', 'c'])  
from pyspark.sql import SparkSession from pyspark.sql.functions import sum #  
Initialize SparkSession spark =  
SparkSession.builder.appName("PivotExample").getOrCreate() # Sample DataFrame data  
= [ ("Alice", "Jan", 100), ("Alice", "Feb", 200), ("Bob", "Jan", 150), ("Bob",  
"Mar", 300) # No "Feb" sales for Bob ] columns = ["Name", "Month", "Sales"] df =  
spark.createDataFrame(data, columns) df.show()
```

```
+---+---+---+  
| Name|Month|Sales|  
+---+---+---+  
| Alice| Jan| 100 |  
| Alice| Feb| 200 |  
| Bob| Jan| 150 |  
| Bob| Mar| 300 |  
+---+---+---+
```

```
    pivot('Month', ['Jan', 'Feb'])  
df_pivot = df.groupBy("Name").pivot("Month", ["Jan", "Feb"]).agg(sum("Sales"))  
df_pivot.show()
```

```
+---+---+---+  
| Name|Jan|Feb|  
+---+---+---+  
| Alice|100|200|  
| Bob|150|NULL|  
+---+---+---+
```

"Jan" "Feb"

"Mar"
"Feb"
Bob

"Feb"

["Jan", "Feb"]

```
    fillna(0)
df_pivot.fillna(0).show()
```

```
+----+---+---+
| Name|Jan|Feb|
+----+---+---+
|Alice|100|200|
| Bob|150| 0|
+----+---+---+
```

	<pre>df.groupBy("Name").pivot("Month").agg(sum("Sales"))</pre>	Includes all unique values dynamically
	<pre>df.groupBy("Name").pivot("Month", ["Jan", "Feb"]).agg(sum("Sales"))</pre>	Only includes ["Jan", "Feb"]
<pre>df_pivot.fillna(0)</pre>		Fills missing values

```
pivot(column, values)
```

```
distinct()      select().distinct()
```

```
from pyspark.sql import SparkSession # Initialize SparkSession
spark = SparkSession.builder.appName("DistinctExample").getOrCreate() # Sample DataFrame
data = [ (1, "Alice"), (2, "Bob"), (3, "Alice"), (4, "David"), (5, "Bob"), (6, "Cathy") ]
columns = ["ID", "Name"]
df = spark.createDataFrame(data, columns) # Select distinct values from "Name" column
df.select("Name").distinct() # Show result
df.distinct().show()
```

```
+----+  
| Name |  
+----+  
| Alice|  
| Bob |  
| David|  
| Cathy |  
+----+
```

Name

dropDuplicates()

```
dropDuplicates(["column_name"])
```

```
df_distinct_alt = df.dropDuplicates(["Name"])
df_distinct_alt.show()
```

```
+---+----+  
| ID| Name |  
+---+----+  
| 1|Alice |  
| 2| Bob |  
| 4|David |  
| 6|Cathy |  
+---+----+
```

Name

<code>distinct()</code>	<code>df.select("Name").distinct()</code>	Extracts unique values from a single column
<code>dropDuplicates(["column_name"])</code>	<code>df.dropDuplicates(["Name"])</code>	Keeps the first occurrence of each unique value

	No changes, keeps original data	Reference Data (e.g., Product Categories)
	Overwrites old data	No history required (e.g., Customer Phone Number)
	Keeps history by adding new rows	Customer Address, Employee Job Title
	Keeps limited history using additional columns	Previous & Current Address
Stores historical data in a separate table		For audits and compliance

	Overwrites existing records	Creates a new row for each change
	No history maintained	History is maintained
	Only current values	Uses <code>Start_Date</code> , <code>End_Date</code> , and <code>Active_Flag</code>
Customer Phone Number		Employee Salary Change, Customer Address Change

`merge()` `overwrite`

```
from pyspark.sql import SparkSession from delta.tables import DeltaTable #
Initialize SparkSession spark =
SparkSession.builder.appName("SCD_Type1").getOrCreate() # New incoming data
(incremental updates) df_updates = spark.createDataFrame([ (1, "Alice", "HR", "New
York"), (2, "Bob", "IT", "San Francisco"), # Location change (3, "Cathy",
"Finance", "Chicago") ], ["id", "name", "department", "location"]) # Load existing
Delta table delta_table = DeltaTable.forPath(spark, "path/to/delta_table") # Merge
for SCD Type 1 (Update existing records) delta_table.alias("existing").merge(
df_updates.alias("new"), "existing.id = new.id" ).whenMatchedUpdate(
set={"existing.location": "new.location"} ).whenNotMatchedInsert( values={"id": "new.id",
"name": "new.name", "department": "new.department", "location": "new.location"} ).execute()
```

`Start_Date` `End_Date` `Active_Flag`

```
from pyspark.sql.functions import col, lit, current_date from delta.tables import DeltaTable # New data df_updates = spark.createDataFrame([ (1, "Alice", "HR", "Boston"), # Changed location (2, "Bob", "IT", "San Francisco"), # No change (3, "Cathy", "Finance", "Los Angeles") # Changed location ], ["id", "name", "department", "location"]) # Load existing data from Delta Table delta_table = DeltaTable.forPath(spark, "path/to/delta_table") # Step 1: Mark existing active records as inactive (end_date = current date) delta_table.alias("existing").merge( df_updates.alias("new"), "existing.id = new.id AND existing.active_flag = 'Y'" ).whenMatchedUpdate( condition="existing.location <> new.location", set={"existing.end_date": current_date(), "existing.active_flag": "'N'"} ).execute() # Step 2: Insert new records with a start date df_new_records = df_updates.withColumn("start_date", current_date()) \ .withColumn("end_date", lit(None).cast("string")) \ .withColumn("active_flag", lit("Y")) df_new_records.write.format("delta").mode("append").save("path/to/delta_table")
```

Previous_Location

```
from pyspark.sql.functions import col # New data with location change df_updates = spark.createDataFrame([ (1, "Alice", "HR", "Boston"), # Location changed (2, "Bob", "IT", "San Francisco"), # No change (3, "Cathy", "Finance", "Los Angeles") # Location changed ], ["id", "name", "department", "current_location"]) # Load existing data delta_table = DeltaTable.forPath(spark, "path/to/delta_table") # Merge logic for SCD Type 3 (track only previous location) delta_table.alias("existing").merge( df_updates.alias("new"), "existing.id = new.id" ).whenMatchedUpdate( condition="existing.current_location <> new.current_location", set={"existing.previous_location": "existing.current_location", "existing.current_location": "new.current_location"} ).whenNotMatchedInsert( values={"id": "new.id", "name": "new.name", "department": "new.department", "previous_location": "NULL", "current_location": "new.current_location"} ).execute()
```

```
active_flag      start_date
```

```
spark.sql("OPTIMIZE delta_table ZORDER BY (id)")
```

```
df.write.format("hudi").option("hoodie.datasource.write.operation",  
"upsert").save("s3://hudi_table/")
```

```
merge()
```

`sales_id`
`date_id`
`product_id`
`customer_id`
`sales_amount`

`date_id` `year` `month` `day`
`product_id` `product_name` `category`
`customer_id` `customer_name` `region`



Store Dimension

	Stores numerical/measurable data (e.g., sales, revenue).
	Stores descriptive attributes (e.g., customer, date, product).
	Reduces joins, improves query performance.
	Optimized for aggregations (SUM, AVG, COUNT).
Simple structure, widely used in BI tools (Tableau, Power BI).	

```
SELECT d.year, p.category, SUM(f.sales_amount) AS total_sales FROM sales_fact f
JOIN date_dim d ON f.date_id = d.date_id JOIN product_dim p ON f.product_id =
p.product_id GROUP BY d.year, p.category;
```

	Denormalized (flat)	Normalized (hierarchical)

	Faster queries, fewer joins	Slower queries, more joins
	Uses more space	Uses less space
	Simple, easy to understand	Complex, harder to maintain
OLAP, reporting, BI tools		OLAP with normalized data

```
join()      update()
```

```
from pyspark.sql import SparkSession # Initialize SparkSession spark =  
SparkSession.builder.appName("SCD_Type1").getOrCreate() # Existing data in  
dimension table (no history) df_existing = spark.createDataFrame([ (1, "Alice",  
"New York"), (2, "Bob", "San Francisco"), (3, "Cathy", "Chicago") ],  
["customer_id", "customer_name", "city"]) # Incoming new data (updates) df_updates  
= spark.createDataFrame([ (1, "Alice", "Boston"), # Change in city (2, "Bob", "San  
Francisco"), # No change (3, "Cathy", "Los Angeles") # Change in city ],  
["customer_id", "customer_name", "city"]) # Perform overwrite using join (SCD Type  
1 - No history, just update) df_scd1 = df_existing.alias("old").join(  
df_updates.alias("new"), "customer_id", "left").select( "customer_id",  
"new.customer_name", # Take updated name "new.city" # Overwrite city )  
df_scd1.show()
```

```
start_date  
end_date
```

```
active_flag
```

```
union() filter()
from pyspark.sql.functions import col, lit, current_date # Existing Data (Active
Records) df_existing = spark.createDataFrame([ (1, "Alice", "New York", "2023-01-
01", None, "Y"), (2, "Bob", "San Francisco", "2023-01-01", None, "Y"), (3,
"Cathy", "Chicago", "2023-01-01", None, "Y") ], ["customer_id", "customer_name",
"city", "start_date", "end_date", "active_flag"]) # New Incoming Data (Updates)
df_updates = spark.createDataFrame([ (1, "Alice", "Boston"), # Change in city (2,
"Bob", "San Francisco"), # No change (3, "Cathy", "Los Angeles") # Change in city
], ["customer_id", "customer_name", "city"]) # Step 1: Identify changed records
df_changes = df_existing.alias("old").join( df_updates.alias("new"), "customer_id"
).filter(col("old.city") != col("new.city")) # Step 2: Mark old records as
inactive df_old_inactive = df_changes.withColumn("end_date",
current_date()).withColumn("active_flag", lit("N")) # Step 3: Insert new records
with new start_date df_new_records = df_updates.withColumn("start_date",
current_date()) \ .withColumn("end_date", lit(None).cast("string")) \
.withColumn("active_flag", lit("Y")) # Step 4: Combine updated old records and new
records df_scd2 = df_existing.union(df_old_inactive).union(df_new_records)
df_scd2.show()
```

```
active_flag = 'N'
```

```
previous_city
current_city
```

```

join()      update()
from pyspark.sql.functions import when # Existing Data df_existing =
spark.createDataFrame([ (1, "Alice", "New York", "New York"), (2, "Bob", "San
Francisco", "San Francisco"), (3, "Cathy", "Chicago", "Chicago") ],
["customer_id", "customer_name", "previous_city", "current_city"]) # Incoming New
Data df_updates = spark.createDataFrame([ (1, "Alice", "Boston"), # Change in city
(2, "Bob", "San Francisco"), # No change (3, "Cathy", "Los Angeles") # Change in
city ], ["customer_id", "customer_name", "new_city"]) # Step 1: Merge Data - Keep
Previous City, Update Current City df_scd3 = df_existing.alias("old").join(
df_updates.alias("new"), "customer_id").select( "customer_id",
"old.customer_name", when(col("old.current_city") != col("new.new_city"),
col("old.current_city")).alias("previous_city"),
col("new.new_city").alias("current_city") ) df_scd3.show()

```

	Overwrites records using <code>join()</code>	When history is (e.g., phone number updates)
	Maintains history using <code>start_date</code> , <code>end_date</code> , <code>active_flag</code>	When (e.g., employee salary changes)
	Stores previous & current values in extra columns	When (e.g., tracking last two addresses)

	(overwrite old data)
	(track full history)

(keep limited history)	

`join()` `union()` `filter()`

	A complete (e.g., <code>show()</code> , <code>collect()</code> , <code>count()</code>).
	A in a job, based on .
	A running on a partition in an executor.

<http://localhost:4040>

```
from pyspark.sql import SparkSession from pyspark.sql.functions import col from
pyspark import SparkContext from pyspark.sql import DataFrame from
pyspark.sql.functions import count # Initialize SparkSession spark =
SparkSession.builder.appName("CountJobsStagesTasks").getOrCreate() sc =
spark.sparkContext # Track Jobs, Stages, and Tasks job_count =
sc.statusTracker().getJobIdsForGroup(None) # Get Job IDs stage_count =
len(sc.statusTracker().getActiveStageIds()) # Get Active Stages tasks =
```

```

sc.statusTracker().getExecutorInfos() # Get Task Executor Info # Print Counts
print(f"Total Jobs: {len(job_count)}") print(f"Total Stages: {stage_count}")
print(f"Total Tasks: {len(tasks)}") # Sample PySpark Action (Triggers a Job) df =
spark.createDataFrame([(1, "Alice"), (2, "Bob"), (3, "Cathy")], ["ID", "Name"])
df.groupBy("Name").count().show() # Check counts again after job execution
job_count = sc.statusTracker().getJobIdsForGroup(None) stage_count =
len(sc.statusTracker().getActiveStageIds()) tasks =
sc.statusTracker().getExecutorInfos() print(f"Total Jobs (After Execution):
{len(job_count)})") print(f"Total Stages (After Execution): {stage_count}")
print(f"Total Tasks (After Execution): {len(tasks)})")

```

```

Total Jobs: 0
Total Stages: 0
Total Tasks: 1

+----+----+
| Name|count|
+----+----+
| Alice|   1|
| Bob|   1|
| Cathy|   1|
+----+----+

Total Jobs (After Execution): 1
Total Stages (After Execution): 2
Total Tasks (After Execution): 4

```

```

from pyspark.sql import SparkSession from pyspark import SparkContext, SparkConf
from pyspark.scheduler import SparkListener, SparkListenerJobStart,
SparkListenerStageCompleted class CustomSparkListener(SparkListener): def
__init__(self): self.jobs = 0 self.stages = 0 def onJobStart(self, jobStart:
SparkListenerJobStart): self.jobs += 1 print(f"Job Started: {self.jobs}") def
onStageCompleted(self, stageCompleted: SparkListenerStageCompleted): self.stages
+= 1 print(f"Stage Completed: {self.stages}") # Initialize Spark conf =
SparkConf().setAppName("SparkJobCounter") sc = SparkContext(conf=conf) # Register
Listener listener = CustomSparkListener() sc._jsc.sc().addSparkListener(listener)
# Execute Some Actions df = spark.createDataFrame([(1, "Alice"), (2, "Bob"), (3,

```

```
"Cathy")], ["ID", "Name"]) df.groupBy("Name").count().show() print(f"Total Jobs: {listener.jobs}") print(f"Total Stages: {listener.stages}")
```

```
spark.conf.set("spark.eventLog.enabled", "true")
spark.conf.set("spark.eventLog.dir", "file:///tmp/spark-events")
```

	Open http://localhost:4040	Quick monitoring
<code>sc.statusTracker()</code>	Get job, stage, task counts programmatically	Small-scale tracking
	Custom listeners to track job execution	Real-time monitoring
	Store & analyze job execution logs	Large-scale job tracking

```
sc.statusTracker()
```

localhost:4040

```
new_df = df.select(F.from_json(F.col("payload"), schema).alias('v')).select('v.*')
```

payload

F.from_json(F.col("payload"), schema).alias('v')	Converts the payload column (which contains JSON strings) into a (v) based on schema.
.select("v.*")	Extracts all fields inside the parsed JSON (v) as separate columns.

df "payload"

```
from pyspark.sql import SparkSession from pyspark.sql.functions import col, from_json from pyspark.sql.types import StructType, StructField, StringType, IntegerType # Initialize SparkSession spark = SparkSession.builder.appName("FromJsonExample").getOrCreate() # Sample JSON DataFrame data = [ ('{"id": 1, "name": "Alice", "age": 25}, ), ('{"id": 2, "name":
```

```
"Bob", "age": 30}, ), ('{"id": 3, "name": "Cathy", "age": 28}, ) ] df =  
spark.createDataFrame(data, ["payload"]) # Define Schema for JSON schema =  
StructType([ StructField("id", IntegerType(), True), StructField("name",  
StringType(), True), StructField("age", IntegerType(), True) ]) # Apply  
from_json() to parse the JSON string new_df = df.select(from_json(col("payload"),  
schema).alias('v')).select("v.*") # Show result new_df.show()
```

```
+---+----+---+  
| id| name|age|  
+---+----+---+  
| 1|Alice| 25|  
| 2| Bob| 30|  
| 3|Cathy| 28|  
+---+----+---+
```

payload

```
payload  
df.select(F.col("payload")).show(truncate=False)  
+-----+  
|payload  
+-----+  
| {"id": 1, "name": "Alice", "age": 25}|  
| {"id": 2, "name": "Bob", "age": 30} |  
| {"id": 3, "name": "Cathy", "age": 28}|  
+-----+
```

"payload"

```
from_json()  
df_parsed = df.select(from_json(col("payload"), schema).alias("v"))  
df_parsed.show(truncate=False)  
+-----+
```

```
| v  
+---+  
| {1, Alice, 25} |  
| {2, Bob, 30} |  
| {3, Cathy, 28} |  
+---+
```

"payload"

v

```
.select("v.*")  
new_df = df_parsed.select("v.*") new_df.show()  
+---+---+---+  
| id| name|age|  
+---+---+---+  
| 1|Alice| 25|  
| 2|Bob| 30|  
| 3|Cathy| 28|  
+---+---+---+
```

id name age

from_json()

v.*

IntegerType() StringType()

```
v.*  
new_df = df.select(from_json(col("payload"),  
schema).alias("v")).select(col("v.name"), col("v.age")) new_df.show()
```

```
+----+---+
| name|age|
+----+---+
|Alice| 25|
| Bob| 30|
|Cathy| 28|
+----+---+
```

"name" "age"

```
schema = StructType([ StructField("id", IntegerType(), True), StructField("name", StringType(), True), StructField("address", StructType([ StructField("city", StringType(), True), StructField("zip", StringType(), True) ])) ]) df_parsed = df.select(from_json(col("payload"), schema).alias("v")) df_parsed.select("v.name", "v.address.city", "v.address.zip").show()
```

address.city address.zip

	<pre>from_json(col("payload"), schema)</pre>	Parses JSON strings into structured format
	<pre>.alias("v")</pre>	Stores parsed JSON into a temporary column (M)
	<pre>.select("v.*")</pre>	Flattens JSON into individual columns

```
from_json()  
    .select("v.*")  
    StructType
```

```
withColumnRenamed()
```

```
withColumnRenamed()
```

```
from pyspark.sql import SparkSession # Initialize SparkSession  
spark = SparkSession.builder.appName("RenameColumn").getOrCreate() # Sample DataFrame  
data = [(1, "Alice"), (2, "Bob"), (3, "Cathy")] df = spark.createDataFrame(data, ["ID", "Name"]) # Rename column "Name" to "Customer_Name"  
df_renamed = df.withColumnRenamed("Name", "Customer_Name") df_renamed.show()
```

```
+---+-----+  
| ID|Customer_Name|  
+---+-----+  
| 1|        Alice|  
| 2|         Bob|  
| 3|       Cathy|  
+---+-----+
```

```
withColumnRenamed("old_name", "new_name")
```

```
withColumnRenamed()
```

```
withColumnRenamed()
```

```
df_renamed = df.withColumnRenamed("ID", "Customer_ID") \  
.withColumnRenamed("Name", "Customer_Name") df_renamed.show()
```

```
+-----+-----+  
|Customer_ID|Customer_Name|  
+-----+-----+  
|          1|      Alice|  
|          2|       Bob|  
|          3|     Cathy|  
+-----+-----+
```

```
toDF()
```

```
.toDF()
```

```
# Rename columns using toDF() df_renamed = df.toDF("Customer_ID", "Customer_Name")  
df_renamed.show()
```

```
selectExpr()
```

```
df_renamed = df.selectExpr("ID as Customer_ID", "Name as Customer_Name")  
df_renamed.show()
```

```
alias()    select()
from pyspark.sql.functions import col df_renamed =
df.select(col("ID").alias("Customer_ID"), col("Name").alias("Customer_Name"))
df_renamed.show()
```

```
rename_mapping = {"ID": "Customer_ID", "Name": "Customer_Name"} # Rename columns
dynamically df_renamed = df.select([col(c).alias(rename_mapping.get(c, c)) for c
in df.columns]) df_renamed.show()
```

withColumnRenamed()	Rename one column	df.withColumnRenamed("ID", "Customer_ID")
toDF()	Rename multiple columns	df.toDF("NewCol1", "NewCol2")
selectExpr()	Rename and transform columns	df.selectExpr("ID as Customer_ID")
select().alias()	Rename a few columns	df.select(col("ID").alias("Customer_ID"))

Dictionary Mapping	Rename many columns dynamically	<pre>df.select([col(c).alias(rename_mapping.get(c, c)) for c in df.columns])</pre>

```
withColumnRenamed()  
toDF()  
selectExpr()
```

```
SELECT | INSERT | UPDATE | DELETE
```

```
WITH cte_name AS ( SELECT column1, column2 FROM table_name WHERE condition )  
SELECT * FROM cte_name;
```

```
from pyspark.sql import SparkSession # Initialize SparkSession spark = SparkSession.builder.appName("CTEExample").getOrCreate() # Sample Data data = [(1, "Alice", "HR", 5000), (2, "Bob", "IT", 7000), (3, "Cathy", "Finance", 6000), (4, "David", "IT", 8000), (5, "Emma", "HR", 5500)] columns = ["ID", "Name", "Department", "Salary"] # Create DataFrame df = spark.createDataFrame(data, columns) # Register as SQL Table df.createOrReplaceTempView("employees")
```

```
query = """ WITH high_salary_employees AS ( SELECT ID, Name, Department, Salary FROM employees WHERE Salary > 6000 ) SELECT * FROM high_salary_employees; """ # Execute SQL Query df_cte = spark.sql(query) df_cte.show()
```

```
+---+----+-----+----+  
| ID|Name |Department|Salary|  
+---+----+-----+----+  
| 2| Bob | IT      | 7000 |  
| 4|David| IT     | 8000 |  
+---+----+-----+----+
```

```
WITH high_salary AS ( SELECT ID, Name, Department, Salary FROM employees WHERE Salary > 6000 ), avg_salary AS ( SELECT Department, AVG(Salary) AS avg_dept_salary FROM employees GROUP BY Department ) SELECT h.*, a.avg_dept_salary FROM high_salary h JOIN avg_salary a ON h.Department = a.Department;
```

```
high_salary  
avg_salary
```

sql()

```
query = """ WITH high_salary AS ( SELECT ID, Name, Department, Salary FROM employees WHERE Salary > 6000 ), avg_salary AS ( SELECT Department, AVG(Salary) AS avg_dept_salary FROM employees GROUP BY Department ) SELECT h.*, a.avg_dept_salary FROM high_salary h JOIN avg_salary a ON h.Department = a.Department; """ df_cte = spark.sql(query) df_cte.show()
```

```
+---+-----+-----+-----+
| ID|Name |Department|Salary|avg_dept_salary|
+---+-----+-----+-----+
|  2| Bob | IT      | 7000 |    7500.0 |
|  4|David| IT      | 8000 |    7500.0 |
+---+-----+-----+-----+
```

	Breaks long queries into modular parts.
	Define a temporary table and reference it multiple times.
	Avoids repeating subqueries multiple times.
CTEs support recursive queries (not in PySpark).	

```
WITH cte_name AS (...)
```

0

0

fillna()

```
from pyspark.sql import SparkSession # Initialize SparkSession
spark = SparkSession.builder.appName("FillNullExample").getOrCreate() # Sample DataFrame
with NULL values
data = [ (1, "Alice", None, 5000), (2, "Bob", "IT", None), (3, None, "Finance", 7000), (4, "David", "HR", 8000), (5, None, None, None) ]
columns = ["ID", "Name", "Department", "Salary"]
df = spark.createDataFrame(data, columns)
# Fill NULL values in all columns with 0
df_filled = df.fillna(0)
df_filled.show()
```

```
+---+----+-----+----+
| ID|Name |Department|Salary|
+---+----+-----+----+
| 1|Alice|          0| 5000|
| 2| Bob|          IT|    0|
| 3|  0| Finance|  7000|
| 4|David|         HR| 8000|
| 5|  0|          0|    0|
+---+----+-----+----+
```

NULL

0

Salary

```
df_filled = df.fillna({"Salary": 0})
df_filled.show()
```

```
+---+-----+-----+-----+
| ID|Name |Department|Salary|
+---+-----+-----+-----+
| 1|Alice|      NULL| 5000|
| 2| Bob|        IT|    0|
| 3| NULL|  Finance| 7000|
| 4|David|        HR| 8000|
| 5| NULL|      NULL|    0|
+---+-----+-----+-----+
```

Salary

na.fill()

na.fill()

fillna()

```
df_filled = df.na.fill(0) df_filled.show()
```

fillna()

```
df_filled = df.fillna({"Name": "Unknown", "Department": "Not Assigned", "Salary": 0}) df_filled.show()
```

```
+---+-----+-----+-----+
| ID|  Name | Department|Salary|
+---+-----+-----+-----+
| 1| Alice |        0| 5000|
| 2| Bob |        IT|    0|
| 3| Unknown|  Finance| 7000|
| 4| David |        HR| 8000|
| 5| Unknown| Not Assigned|    0|
+---+-----+-----+-----+
```

fillna()

```
from pyspark.sql.functions import avg # Compute mean of Salary column mean_salary
= df.select(avg("Salary")).collect()[0][0] # Fill NULLs in Salary with the mean
value df_filled = df.fillna({"Salary": mean_salary}) df_filled.show()
```

Salary

	Replaces NULL in all columns	df.fillna(0)
	Only replaces NULLs in a column	df.fillna({"Salary": 0})
	Assigns unique values	df.fillna({"Name": "Unknown", "Salary": 0})
	Replaces NULLs with an aggregate function	df.fillna({"Salary": mean_salary})

```
fillna(0)
fillna({"column": value})
fillna()
na.fill()
```

	Different columns for different datasets	Different CSVs for customers & products
	Records get appended/modified frequently	Real-time financial transactions
	Users define which columns to export	Exporting reports with selected fields
CSVs created dynamically from databases/APIs		Automated data dumps from SQL

```
from pyspark.sql import SparkSession import pandas as pd # Initialize Spark
spark = SparkSession.builder.appName("DynamicCSV").getOrCreate() # Sample Data
```

```
(Different CSVs may have different structures) data_1 = [(1, "Alice", 25), (2, "Bob", 30)] data_2 = [(3, "Cathy", "New York"), (4, "David", "San Francisco")] # Define different schemas columns_1 = ["ID", "Name", "Age"] columns_2 = ["ID", "Name", "City"] # Dynamically choose the dataset data_choice = "dataset_1" # Change this to dataset_2 for a different CSV if data_choice == "dataset_1": df = spark.createDataFrame(data_1, columns_1) else: df = spark.createDataFrame(data_2, columns_2) # Save as CSV df.write.option("header", True).csv(f"output/{data_choice}.csv")
```

```
selected_columns = ["ID", "Name"] # User-defined column selection df_selected = df.select(selected_columns) # Save dynamically selected columns as CSV df_selected.write.option("header", True).csv("output/filtered_output.csv")
```

```
from pyspark.sql.functions import lit # New DataFrame with incoming data new_data = spark.createDataFrame([(5, "Emma", 35)], ["ID", "Name", "Age"]) # Append new data to the existing CSV new_data.write.mode("append").option("header", True).csv("output/dynamic_data.csv")
```

```
df_dynamic = spark.read.format("jdbc").option("url",  
"jdbc:mysql://localhost:3306/db") \ .option("dbtable", "customers").option("user",  
"root").option("password", "password").load() df_dynamic.write.option("header",  
True).csv("output/dynamic_database_export.csv")
```

	Different columns based on data type
	Users decide what to export
	Data gets appended dynamically
Scheduled data dumps from databases	

```
dropDuplicates()    distinct()
```

```
dropDuplicates()    distinct()
```

```
distinct()
```

```
distinct()
```

```
from pyspark.sql import SparkSession # Initialize Spark
spark = SparkSession.builder.appName("DistinctVsDropDuplicates").getOrCreate() # Sample
DataFrame data = [ (1, "Alice", "HR"), (2, "Bob", "IT"), (1, "Alice", "HR"), # Duplicate row
(3, "David", "HR"), (2, "Bob", "IT") ] # Duplicate row ] columns =
[ "ID", "Name", "Department" ] df = spark.createDataFrame(data, columns) # Apply
distinct() df_distinct = df.distinct() df_distinct.show()
```

```
+---+---+-----+
| ID|Name |Department|
+---+---+-----+
|  1|Alice| HR
|  2| Bob | IT
|  3|David| HR
+---+---+-----+
```

```
dropDuplicates()
```

```
dropDuplicates(["column_name"])
```

```
# Drop duplicates based on "Name" column df_drop_dup = df.dropDuplicates(["Name"])
df_drop_dup.show()
```

```

+---+----+-----+
| ID|Name |Department|
+---+----+-----+
| 1|Alice| HR
| 2| Bob | IT
| 3|David| HR
+---+----+-----+

```

Name

```

dropDuplicates(["ID", "Name"])
# Drop duplicates based on ID and Name df_drop_dup = df.dropDuplicates(["ID",
"Name"]) df_drop_dup.show()

```

```

+---+----+-----+
| ID|Name |Department|
+---+----+-----+
| 1|Alice| HR
| 2| Bob | IT
| 3|David| HR
+---+----+-----+

```

ID Name

Department

dropDuplicates() distinct()

	distinct()	dropDuplicates()

	<code>distinct()</code>	<code>dropDuplicates()</code>
	Removing	Removing duplicates based on

Remove duplicate rows from the dataset	<code>distinct()</code>
Remove duplicates	<code>dropDuplicates(["column_name"])</code>
Remove duplicates but	<code>dropDuplicates(["column_name"])</code>
Ensure	<code>dropDuplicates(["col1", "col2"])</code>

```
distinct()  
dropDuplicates(["col1", "col2"])
```

```
dropDuplicates()
```

```
coalesce()
```

```
coalesce()
```

```
coalesce(N)
```

```
repartition()
```

```
coalesce(N)
from pyspark.sql import SparkSession # Initialize SparkSession
spark = SparkSession.builder.appName("CoalesceExample").getOrCreate() # Create a DataFrame
with multiple partitions df = spark.range(1000).repartition(10) # 10 partitions
initially # Reduce partitions using coalesce() df_coalesced = df.coalesce(2)
print("Initial Partitions:", df.rdd.getNumPartitions()) # Output: 10 print("After
Coalesce:", df_coalesced.rdd.getNumPartitions()) # Output: 2
```

```
10    2
```

coalesce(N)	repartition(N)
coalesce(N)	the number of partitions
repartition(N)	with partitions
	When you need to balance data distribution

coalesce(N)	the number of partitions	When reducing partitions for writing large files
repartition(N)	with partitions	When you need to balance data distribution

```
coalesce(col1, col2, ...)
```

```
coalesce()
```

```

from pyspark.sql.functions import coalesce, lit # Sample DataFrame with NULL
values data = [(1, None, "Default Name"), (2, "Alice", None), (3, None, None)]
columns = ["ID", "Name", "Fallback_Name"] df = spark.createDataFrame(data,
columns) # Use coalesce() to fill missing values df_filled =
df.withColumn("Final_Name", coalesce(df["Name"], df["Fallback_Name"],
lit("Unknown"))) df_filled.show()

```

```

+---+----+-----+-----+
| ID| Name|Fallback_Name|Final_Name |
+---+----+-----+-----+
| 1| NULL| Default Name|Default Name|
| 2|Alice|          NULL|      Alice|
| 3| NULL|          NULL|    Unknown |
+---+----+-----+-----+

```

coalesce()		
	coalesce()	
Reduce partitions before saving	<code>df.coalesce(N)</code>	<code>df.coalesce(2).write.parquet("output")</code>
Handle NULL values in columns	<code>coalesce(col1, col2, ...)</code>	<code>df.withColumn("Final", coalesce(df["A"], df["B"], lit(0)))</code>

```

df.coalesce(N)
coalesce(col1, col2, ...)
coalesce()
coalesce()

```

COALESCE() SELECT

coalesce() SELECT

coalesce() select()

COALESCE()
coalesce() select()

```
from pyspark.sql import SparkSession from pyspark.sql.functions import coalesce,  
lit # Initialize SparkSession spark =  
SparkSession.builder.appName("CoalesceInSelect").getOrCreate() # Sample DataFrame  
with NULL values data = [(1, None, "Backup Name"), (2, "Alice", None), (3, None,  
None)] columns = ["ID", "Name", "Fallback_Name"] df = spark.createDataFrame(data,  
columns) # Use coalesce() in select() df_selected = df.select("ID",  
coalesce(df["Name"], df["Fallback_Name"], lit("Unknown")).alias("Final_Name"))  
df_selected.show()
```

ID	Final_Name
1	Backup Name
2	Alice
3	Unknown

coalesce() Final_Name

coalesce()

coalesce()

```
COALESCE()    spark.sql()
```

```
# Register DataFrame as SQL Table df.createOrReplaceTempView("people") # Use
COALESCE in SQL Query query = """ SELECT ID, COALESCE(Name, Fallback_Name,
'Unknown') AS Final_Name FROM people """
df_sql = spark.sql(query) df_sql.show()
```

```
+---+-----+
| ID| Final_Name |
+---+-----+
| 1| Backup Name|
| 2|      Alice |
| 3|    Unknown |
+---+-----+
```

```
coalesce()
```

select()	coalesce(col("A"), col("B"), [] lit("Default"))	df.select("ID", coalesce(df["A"], df["B"], [] lit("Unknown")).alias("Final"))
	COALESCE(A, B, 'Default')	SELECT ID, COALESCE(A, B, 'Unknown') FROM table

```
coalesce()
```

```
select()
```

```
COALESCE(A, B, 'Default')
```

```
agg() select() groupBy()    orderBy()
```

```
agg(max())
```

```
max() orderBy()    groupBy()
```

```
agg(max())
```

```
from pyspark.sql import SparkSession from pyspark.sql.functions import max #  
Initialize Spark spark =  
SparkSession.builder.appName("MaxDateExample").getOrCreate() # Sample DataFrame  
with Date Column data = [("Alice", "2024-01-10"), ("Bob", "2024-03-05"), ("Cathy",  
"2024-02-20")] columns = ["Name", "Transaction_Date"] df =  
spark.createDataFrame(data, columns) # Convert to DateType from  
pyspark.sql.functions import col from pyspark.sql.types import DateType df =  
df.withColumn("Transaction_Date", col("Transaction_Date").cast(DateType())) # Find  
the Maximum Date max_date = df.agg(max("Transaction_Date").alias("Max_Date"))  
max_date.show()
```

```
+-----+  
| Max_Date |  
+-----+  
| 2024-03-05 |  
+-----+
```

```
agg(max("column_name"))
```

```
orderBy()
```

```
orderBy(descending)
```

```
from pyspark.sql.functions import desc df_latest =  
df.orderBy(col("Transaction_Date").desc()).limit(1) df_latest.show()
```

```
+----+-----+  
| Name|Transaction_Date|  
+----+-----+  
| Bob | 2024-03-05|  
+----+-----+
```

```
orderBy(desc()).limit(1)
```

```
groupBy()
```

```
df_grouped = df.groupBy("Name").agg(max("Transaction_Date").alias("Max_Date"))  
df_grouped.show()
```

```
+----+-----+  
| Name | Max_Date |  
+----+-----+  
|Alice | 2024-01-10|  
| Bob | 2024-03-05|  
|Cathy | 2024-02-20|  
+----+-----+
```

```
groupBy()
```

	<code>agg(max())</code>	<code>df.agg(max("date_column"))</code>
	<code>orderBy(desc()).limit(1)</code>	<code>df.orderBy(col("date_column").desc()).limit(1)</code>
	<code>groupBy().agg(max())</code>	<code>df.groupBy("category").agg(max("date_column"))</code>

```
agg(max("column"))
orderBy(desc()).limit(1)
groupBy().agg(max("column"))
```

```
to_date()  to_timestamp()    cast()
```

```
yyyy-MM-dd
```

```
to_date()
```

```
DateType
```

```
from pyspark.sql import SparkSession from pyspark.sql.functions import to_date
from pyspark.sql.types import StringType # Initialize Spark
spark = SparkSession.builder.appName("StringToDate").getOrCreate() # Sample DataFrame with
String Dates data = [("2024-03-05",), ("2024-02-20",), ("2024-01-10",)] df =
spark.createDataFrame(data, ["date_str"]) # Convert String to DateType df =
df.withColumn("date_col", to_date(df["date_str"], "yyyy-MM-dd")) df.show()
```

```
+-----+-----+
| date_str | date_col |
+-----+-----+
| 2024-03-05 | 2024-03-05 |
| 2024-02-20 | 2024-02-20 |
| 2024-01-10 | 2024-01-10 |
+-----+-----+
```

DateType

yyyy-MM-dd HH:mm:ss

to_timestamp()

TimestampType

```
from pyspark.sql.functions import to_timestamp # Sample DataFrame with String
Timestamps data = [("2024-03-05 15:30:00",), ("2024-02-20 10:45:00",), ("2024-01-
10 23:59:59",)] df = spark.createDataFrame(data, ["timestamp_str"]) # Convert
String to TimestampType df = df.withColumn("timestamp_col",
to_timestamp(df["timestamp_str"], "yyyy-MM-dd HH:mm:ss")) df.show(truncate=False)
```

```
+-----+-----+
| timestamp_str | timestamp_col |
+-----+-----+
| 2024-03-05 15:30:00 | 2024-03-05 15:30:00 |
| 2024-02-20 10:45:00 | 2024-02-20 10:45:00 |
| 2024-01-10 23:59:59 | 2024-01-10 23:59:59 |
+-----+-----+
```

`cast()`

```
df = df.withColumn("timestamp_cast", df["timestamp_str"].cast("timestamp"))
df.show(truncate=False)
```

`yyyy-MM-dd HH:mm:ss`

`to_date()` `to_timestamp()` `MM/dd/yyyy`

```
df = df.withColumn("date_fixed", to_date(df["timestamp_str"], "MM/dd/yyyy"))
```

<code>(yyyy-MM-dd)</code>	<code>to_date()</code>	<code>to_date(df["col"], "yyyy-MM-dd")</code>
<code>(yyyy-MM-dd HH:mm:ss)</code>	<code>to_timestamp()</code>	<code>to_timestamp(df["col"], "yyyy-MM-dd HH:mm:ss")</code>
<code>cast("timestamp")</code>		<code>df["col"].cast("timestamp")</code>

```
to_date()          DateType  
to_timestamp()    TimestampType  
cast("timestamp") yyyy-MM-dd HH:mm:ss
```

```
AttributeError: 'str' object has no attribute 'alias'  
    .alias()
```

```
df = df.withColumn("new_column", "old_column".alias("alias_name"))
```

```
"old_column"
```

```
col()      pyspark.sql.functions  
          .alias()  
from pyspark.sql.functions import col df = df.withColumn("new_column",  
col("old_column").alias("alias_name"))
```

```
col("old_column") .alias()
```

```
select().alias()  
df.select(col("old_column").alias("new_column")).show()
```

```
col("old_column")
```

```
withColumn()
```

```
from pyspark.sql.functions import upper df = df.withColumn("upper_column",  
upper(col("old_column")).alias("new_alias"))
```

```
withColumn()
```

"column_name".alias("new_name")	col("column_name").alias("new_name")
df.withColumn("new_col", "old_col".alias("alias"))	df.withColumn("new_col", col("old_col"))
"column_name".upper()	col("column_name").upper()

```
col("column_name")  
.alias()
```

```
max("InvoiceDate")
```

```
current_date = orders.agg(F.max("InvoiceDate").alias("Max_Date"))
```

InvoiceDate

InvoiceDate

InvoiceDate

```
from pyspark.sql import SparkSession from pyspark.sql import functions as F from
pyspark.sql.types import TimestampType # Initialize Spark spark =
SparkSession.builder.appName("MaxDateExample").getOrCreate() # Sample Data data =
[("2024-03-05 15:30:00",), ("2024-02-20 10:45:00",), ("2024-01-10 23:59:59",)] df =
spark.createDataFrame(data, ["InvoiceDate"]) # Convert to TimestampType df =
df.withColumn("InvoiceDate", df["InvoiceDate"].cast(TimestampType())) # Get
Maximum Date-Time current_date = df.agg(F.max("InvoiceDate").alias("Max_Date"))
current_date.show()
```

```
+-----+
| Max_Date      |
+-----+
| 2024-03-05 15:30:00 |
+-----+
```

InvoiceDate

TimestampType

InvoiceDate

to_timestamp()

InvoiceDate

to_timestamp()

```

from pyspark.sql.functions import to_timestamp # Convert string to timestamp
before aggregation df = df.withColumn("InvoiceDate",
to_timestamp(df["InvoiceDate"], "yyyy-MM-dd HH:mm:ss")) # Get Maximum Date
current_date = df.agg(F.max("InvoiceDate").alias("Max_Date")).current_date.show()

```

TimestampType	Use <code>.cast(TimestampType())</code>	<code>df.withColumn("InvoiceDate", df["InvoiceDate"].cast(TimestampType()))</code>
	Use <code>to_timestamp()</code>	<code>to_timestamp(df["InvoiceDate"], "yyyy-MM-dd HH:mm:ss")</code>
Use <code>agg(F.max("column"))</code>		<code>df.agg(F.max("InvoiceDate"))</code>

```

InvoiceDate      TimestampType
to_timestamp()
agg(F.max("InvoiceDate"))

```

```
collect()
```

```
collect()
```

`collect()`

```
from pyspark.sql import SparkSession # Initialize SparkSession
spark = SparkSession.builder.appName("CollectExample").getOrCreate() # Sample DataFrame
data = [(1, "Alice", 5000), (2, "Bob", 7000), (3, "Cathy", 6000)] columns = ["ID", "Name", "Salary"]
df = spark.createDataFrame(data, columns) # Collect all rows
collected_data = df.collect() # Print collected data
for row in collected_data:
    print(row)
```

```
Row(ID=1, Name='Alice', Salary=5000)
Row(ID=2, Name='Bob', Salary=7000)
Row(ID=3, Name='Cathy', Salary=6000)
```

Row

`collect()`

`collect()`

Row

`collect()`

```
# Extract Names only
names = [row.Name for row in df.collect()]
print(names)
```

```
['Alice', 'Bob', 'Cathy']
```

Name

`collect()`

`.collect()[0]`

```
from pyspark.sql.functions import max # Get Max Salary max_salary = df.agg(max("Salary")).collect()[0][0] print(max_salary)
```

```
7000
```

```
first()
```

```
collect()[0]
```

```
.first()
```

```
.collect()[0]
```

```
first_row = df.first() print(first_row.Name) # Output: Alice
```

```
collect()[0]
```

```
take(N)
```

```
.take(N)
```

```
limited_data = df.take(2) for row in limited_data: print(row)
```

```
Row(ID=1, Name='Alice', Salary=5000)
Row(ID=2, Name='Bob', Salary=7000)
```

```
take(N)
```

```
collect()
```

```
collect()
```

	.collect()	df.collect()
	[row.col_name for row in df.collect()]	[row.Name for row in df.collect()]
	.collect()[0][0]	df.agg(max("col")).collect()[0][0]
	.first()	df.first().Name
	.take(N)	df.take(2)

```
collect()                                Row
 [row.col_name for row in df.collect()]
    first()      collect() [0]
take(N)          collect()
```

```
datetime.timedelta
```

```
import datetime df = df.withColumn("new_date", df["InvoiceDate"] -  
datetime.timedelta(days=30)) # ✗ This will fail
```

datetime.timedelta

date_sub()	timestamp_sub()
<u>date_sub()</u>	

InvoiceDate	date_sub()
-------------	------------

```
from pyspark.sql.functions import date_sub df = df.withColumn("new_date",  
date_sub(df["InvoiceDate"], 30)) df.show()
```

InvoiceDate

expr()

InvoiceDate	expr()
-------------	--------

```
from pyspark.sql.functions import expr df = df.withColumn("new_date",  
expr("InvoiceDate - INTERVAL 30 DAYS")) df.show()
```

TimestampType

datetime.timedelta

datetime

```
lit()      current_date()  
from pyspark.sql.functions import lit, current_date df = df.withColumn("new_date",  
current_date() - lit(30)) df.show()
```

current_date()

	date_sub()	df.withColumn("new_date", date_sub(df["InvoiceDate"], 30))
	expr()	df.withColumn("new_date", expr("InvoiceDate - INTERVAL 30 DAYS"))
datetime.timedelta	lit()	df.withColumn("new_date", current_date() - lit(30))

datetime.timedelta

date_sub() DateType expr() TimestampType
timedelta lit()

timedelta

timedelta datetime

```
datetime
```

timedelta

```
from datetime import datetime, timedelta # Current date and time now =  
datetime.now() print("Now:", now) # Subtract 10 days past_date = now -  
timedelta(days=10) print("10 days ago:", past_date) # Add 5 hours future_time =  
now + timedelta(hours=5) print("5 hours later:", future_time)
```

```
Now: 2024-02-13 15:30:00  
10 days ago: 2024-02-03 15:30:00  
5 hours later: 2024-02-13 20:30:00
```

```
timedelta(days=10)  
timedelta(hours=5)
```

timedelta

```
delta = timedelta(days=2, hours=3, minutes=30) print("Days:", delta.days)  
print("Seconds:", delta.seconds) print("Total Seconds:", delta.total_seconds())
```

```
Days: 2  
Seconds: 12600 # (3 hours + 30 minutes converted to seconds)  
Total Seconds: 183000.0
```

```
days hours minutes seconds
```

```
timedelta
```

```
date1 = datetime(2024, 2, 1) date2 = datetime(2024, 2, 13) difference = date2 -  
date1 print("Difference:", difference) print("Days:", difference.days)
```

```
Difference: 12 days, 0:00:00  
Days: 12
```

datetime

timedelta

timedelta

```
today = datetime.today() # 1 Week Later next_week = today + timedelta(weeks=1)  
print("Next Week:", next_week) # 30 Days Before last_month = today -  
timedelta(days=30) print("30 Days Ago:", last_month)
```

```
timedelta(weeks=1)  
timedelta(days=30)
```

timedelta

```
delta = timedelta(days=1, hours=5, minutes=30) # Convert to Hours hours =  
delta.total_seconds() / 3600 print("Total Hours:", hours) # Convert to Minutes  
minutes = delta.total_seconds() / 60 print("Total Minutes:", minutes)
```

```
Total Hours: 29.5  
Total Minutes: 1770.0
```

total_seconds()

timedelta

timedelta

	now = datetime.now()	2024-02-13 15:30:00

	<code>now + timedelta(days=5)</code>	<code>2024-02-18 15:30:00</code>
	<code>now - timedelta(days=10)</code>	<code>2024-02-03 15:30:00</code>
	<code>now + timedelta(hours=3)</code>	<code>2024-02-13 18:30:00</code>
	<code>date2 - date1</code>	<code>timedelta(days=12)</code>
<code>timedelta</code>	<code>delta.total_seconds() / 3600</code>	<code>29.5</code>

`timedelta`



`.toPandas()`

`.toPandas()`

```

from pyspark.sql import SparkSession import pandas as pd # Initialize SparkSession
spark = SparkSession.builder.appName("PySparkToPandas").getOrCreate() # Sample PySpark DataFrame
data = [(1, "Alice", 5000), (2, "Bob", 7000), (3, "Cathy", 6000)]
columns = ["ID", "Name", "Salary"]
df_spark = spark.createDataFrame(data, columns) # Convert to Pandas DataFrame
df_pandas = df_spark.toPandas() # Display Pandas DataFrame
print(df_pandas)

```

```
ID  Name  Salary
0   1    Alice  5000
1   2    Bob   7000
2   3    Cathy 6000
```

```
.toPandas()
```

```
df_pandas = df_spark.limit(10000).toPandas() # Convert only 10,000 rows
```

```
Arrow
```

```
spark.conf.set("spark.sql.execution.arrow.enabled", "true") df_pandas =
df_spark.toPandas()
```

	<code>.toPandas()</code>	<code>df_spark.toPandas()</code>
	<code>.limit(N).toPandas())</code>	<code>df_spark.limit(10000).toPandas()</code>
Enable Arrow		<code>spark.conf.set("spark.sql.execution.arrow.enabled", "true")</code>

```
.toPandas()
```

```
TypeError: Casting to unit-less dtype 'datetime64' is not  
supported
```

```
TimestampType    DateType           .toPandas()
```

```
df_pandas = df_spark.toPandas() # May cause the error if TimestampType exists
```

```
TypeError: Casting to unit-less dtype 'datetime64' is not supported. Pass e.g.  
'datetime64[ns]' instead.
```

```
datetime64[ns]
```

```
.toPandas()
```

```
df_pandas = df_spark.toPandas() # Convert specific columns to datetime  
df_pandas["InvoiceDate"] = pd.to_datetime(df_pandas["InvoiceDate"])  
print(df_pandas.dtypes) # Verify the data types
```

```
InvoiceDate
```

```
datetime64[ns]
```

```
.toPandas()
```

```
StringType
```

```
from pyspark.sql.functions import col # Convert timestamp to string in PySpark  
before converting to Pandas df_spark = df_spark.withColumn("InvoiceDate",  
col("InvoiceDate").cast("string")) # Convert to Pandas df_pandas =  
df_spark.toPandas() print(df_pandas.dtypes)
```

	Convert in Pandas after <code>.toPandas()</code>	<code>df_pandas["InvoiceDate"] = pd.to_datetime(df_pandas["InvoiceDate"])</code>
<code>.toPandas()</code>	Cast timestamp to string in PySpark	<code>df_spark.withColumn("InvoiceDate", col("InvoiceDate").cast("string"))</code>

`TimestampType`
`pd.to_datetime()`
`StringType`

`datetime64[ns]`

`when`

`when`

```
from pyspark.sql import SparkSession from pyspark.sql.functions import when, col #
Initialize Spark session spark =
```

```

SparkSession.builder.appName("WhenExample").getOrCreate() # Sample DataFrame data
= [(1, "Alice", 5000), (2, "Bob", 7000), (3, "Cathy", 3000)] columns = ["ID",
"Name", "Salary"] df = spark.createDataFrame(data, columns) # Apply 'when'
function df = df.withColumn( "Salary_Category", when(col("Salary") > 5000,
"High").otherwise("Low") ) df.show()

```

ID	Name	Salary	Salary_Category
1	Alice	5000	Low
2	Bob	7000	High
3	Cathy	3000	Low

when

when

.otherwise()

```

df = df.withColumn( "Salary_Category", when(col("Salary") > 6000, "Very High")
.when(col("Salary") > 4000, "High") .otherwise("Low"), ) df.show()

```

ID	Name	Salary	Salary_Category
1	Alice	5000	High
2	Bob	7000	Very High
3	Cathy	3000	Low

when otherwise()

when isin()

```
~col().isin()      when
```

```
df = df.withColumn( "Is_Important", when(~col("Name").isin(["Alice", "Bob"]),
"Yes").otherwise("No"), ) df.show()
```

```
+---+-----+-----+-----+
| ID| Name|Salary|Salary_Category|Is_Important|
+---+-----+-----+-----+
| 1|Alice| 5000 | High          | No        |
| 2| Bob| 7000 | Very High    | No        |
| 3|Cathy| 3000 | Low           | Yes       |
+---+-----+-----+-----+
```

```
[ "Alice", "Bob" ]
```

```
when
```

```
df = df.withColumn( "Bonus", when((col("Salary") > 5000) & (col("Name") == "Bob"),
1000).otherwise(500), ) df.show()
```

```
+---+-----+-----+-----+-----+
| ID| Name|Salary|Salary_Category|Is_Important|Bonus|
+---+-----+-----+-----+-----+
| 1|Alice| 5000 | High          | No        | 500  |
| 2| Bob| 7000 | Very High    | No        | 1000 |
| 3|Cathy| 3000 | Low           | Yes       | 500  |
+---+-----+-----+-----+-----+
```

```
when
```

	<code>when(col("Salary") > 5000, "High").otherwise("Low")</code>
	<code>.when(col("Salary") > 6000, "Very High").when(col("Salary") > 4000, "High").otherwise("Low")</code>
	<code>when(~col("Name").isin(["Alice", "Bob"])), "Yes").otherwise("No")</code>
	<code>when((col("Salary") > 5000) & (col("Name") == "Bob"), 1000).otherwise(500)</code>

when CASE WHEN
 when
 ~col().isin()
 & |

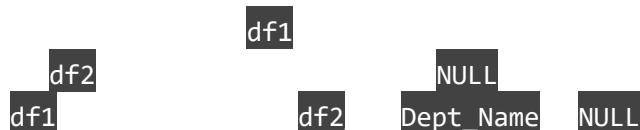
 join()

```

from pyspark.sql import SparkSession from pyspark.sql.functions import col #
Initialize Spark session spark =
SparkSession.builder.appName("LeftJoinExample").getOrCreate() # Left DataFrame
  
```

```
(employees) data1 = [(1, "Alice", 1001), (2, "Bob", 1002), (3, "Cathy", 1005)]
columns1 = ["ID", "Name", "Dept_ID"]
df1 = spark.createDataFrame(data1, columns1)
# Right DataFrame (departments)
data2 = [(1001, "HR"), (1002, "Finance"), (1003, "IT")]
columns2 = ["Dept_ID", "Dept_Name"]
df2 = spark.createDataFrame(data2, columns2)
# Perform Left Join
df_left_join = df1.join(df2, on="Dept_ID", how="left")
df_left_join.show()
```

```
+-----+---+-----+
|Dept_ID| ID| Name |Dept_Name|
+-----+---+-----+
| 1001 | 1| Alice |      HR |
| 1002 | 2|   Bob | Finance |
| 1005 | 3|Cathy |      NULL |
+-----+---+-----+
```



```
df_left_join = df1.join(df2, (df1.Dept_ID == df2.Dept_ID) & (df1.Name == "Alice"),
"left")
df_left_join.show()
```

Dept_ID

```
df_left_join = df1.join(df2, on="Dept_ID", how="left").select(df1["*"],  
df2["Dept_Name"]) df_left_join.show()
```

	<code>df1.join(df2, "Dept_ID", "left")</code>
	<code>df1.join(df2, (df1.Dept_ID == df2.Dept_ID) & (df1.Name == "Alice"), "left")</code>
	<code>.select(df1["*"], df2["Dept_Name"])</code>



```
coalesce()
```

```
coalesce()
```

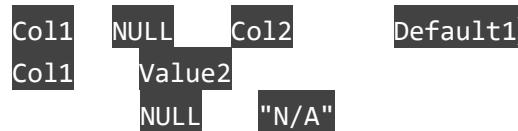
```
coalesce()
```

```
NULL
```

```
coalesce()
```

```
from pyspark.sql import SparkSession from pyspark.sql.functions import coalesce # Initialize Spark session spark = SparkSession.builder.appName("CoalesceExample").getOrCreate() # Sample data with NULL values data = [ (1, None, "Default1"), (2, "Value2", None), (3, None, None) ] columns = ["ID", "Col1", "Col2"] # Create DataFrame df = spark.createDataFrame(data, columns) # Apply coalesce: Pick the first non-null value df = df.withColumn("Final_Column", coalesce(df["Col1"], df["Col2"], "N/A")) df.show()
```

```
+---+----+----+-----+
| ID| Col1 | Col2 | Final_Column |
+---+----+----+-----+
| 1| NULL | Default1 | Default1 |
| 2| Value2 | NULL | Value2 |
| 3| NULL | NULL | N/A |
+---+----+----+-----+
```



```
coalesce()
```

```
coalesce()
```

```
df.createOrReplaceTempView("temp_table") spark.sql(""" SELECT ID, Col1, Col2,
COALESCE(Col1, Col2, 'N/A') AS Final_Column FROM temp_table """).show()
```

```
from pyspark.sql import SparkSession from pyspark.sql.functions import col, sum # Initialize Spark session spark = SparkSession.builder.appName("DynamicAggregation").getOrCreate() # Sample Data data = [ (1, 10, 20, 30), (2, 15, 25, 35), (3, 20, 30, 40) ] columns = ["ID", "Sales_A", "Sales_B", "Sales_C"] # Create DataFrame df = spark.createDataFrame(data, columns) # Define columns to aggregate dynamically agg_columns = ["Sales_A", "Sales_B", "Sales_C"] # Generate dynamic aggregation expressions agg_exprs = [sum(col(c)).alias(f"Total_{c}") for c in agg_columns] # Apply aggregation df_result = df.agg(*agg_exprs) df_result.show()
```

Total_A	Total_B	Total_C
45	75	105

```
+-----+-----+-----+-----+
```

```
"Sales_A" "Sales_B"  
"Sales_C"  
agg()
```

```
from pyspark.sql.functions import avg, max # Define dynamic aggregation  
expressions agg_exprs = [] for col_name in agg_columns:  
    agg_exprs.append(sum(col(col_name)).alias(f"Total_{col_name}"))  
    agg_exprs.append(avg(col(col_name)).alias(f"Avg_{col_name}"))  
    agg_exprs.append(max(col(col_name)).alias(f"Max_{col_name}")) # Apply aggregation  
df_result = df.agg(*agg_exprs) df_result.show()
```

```
+-----+-----+-----+-----+-----+-----+-----+  
|-----+-----+-----+-----+-----+-----+-----+  
| Total_A | Avg_A      | Max_A      | Total_B   | Avg_B      | Max_B      | Total_C   | Avg_C  
| Max_C  |  
|-----+-----+-----+-----+-----+-----+-----+  
|     45  |  15.0      |  20        |     75     |  25.0      |  30        |    105     |  35.0     |  
| 40     |  
|-----+-----+-----+-----+-----+-----+-----+
```

```
# Define grouping column and aggregation columns
group_column = "ID"
agg_columns = ["Sales_A", "Sales_B", "Sales_C"]
# Generate dynamic expressions for aggregation
agg_expressions = [sum(col(c)).alias(f"Total_{c}") for c in agg_columns]
# Apply GroupBy and Aggregation
df_grouped = df.groupby(group_column).agg(*agg_expressions)
df_grouped.show()
```

sum() avg() max()

QUALIFY
HAVING

QUALIFY

```
SELECT EmployeeID, Department, Salary, RANK() OVER (PARTITION BY Department ORDER BY Salary DESC) AS Rank FROM Employees QUALIFY Rank = 1;
```

```
RANK() OVER (PARTITION BY Department ORDER BY Salary DESC)
```

```
QUALIFY Rank = 1
```

```
SELECT Region, SalesPerson, TotalSales, DENSE_RANK() OVER (PARTITION BY Region  
ORDER BY TotalSales DESC) AS SalesRank FROM SalesData QUALIFY SalesRank <= 3;
```

	WHERE	HAVING	QUALIFY
WHERE	Filters	aggregation (raw data level).	
HAVING	Filters	aggregation (works with GROUP BY).	
QUALIFY	Filters	window functions are applied.	

QUALIFY

HAVING

QUALIFY

QUALIFY

QUALIFY

QUALIFY

```
SELECT EmployeeID, Department, Salary, RANK() OVER (PARTITION BY Department ORDER BY Salary DESC) AS SalaryRank FROM Employees QUALIFY SalaryRank = 1;
```

RANK() OVER (PARTITION BY Department ORDER BY Salary DESC)

QUALIFY SalaryRank = 1

```
SELECT Region, SalesPerson, TotalSales, DENSE_RANK() OVER (PARTITION BY Region ORDER BY TotalSales DESC) AS SalesRank FROM SalesData QUALIFY SalesRank <= 3;
```

QUALIFY

QUALIFY

```
SELECT * FROM ( SELECT Region, SalesPerson, TotalSales, RANK() OVER (PARTITION BY Region ORDER BY TotalSales DESC) AS SalesRank FROM SalesData ) t WHERE SalesRank <= 3;
```

QUALIFY
BY
HAVING
GROUP
INTO
INSERT

```
from pyspark.sql import SparkSession from pyspark.sql.types import StructType, StructField, IntegerType, StringType, DateType # Initialize Spark session
spark = SparkSession.builder.appName("CreateDataFrame").getOrCreate() # Define Schema
schema = StructType([ StructField("ProjectID", IntegerType(), False), StructField("ProjectName", StringType(), False), StructField("EmployeeID", IntegerType(), False), StructField("StartDate", DateType(), False), StructField("EndDate", DateType(), True) # Nullable field ])
# Create Data
data = [(1, "Website Redesign", 2, "2023-01-01", "2023-03-31"), (2, "Financial Analysis", 3, "2023-02-01", "2023-04-30"), (3, "HR Onboarding", 1, "2023-01-15", "2023-02-28"), (4, "Cloud Migration", 4, "2023-03-01", None) # NULL in EndDate ]
# Convert StartDate and EndDate to proper date format
from pyspark.sql.functions import col, to_date
df = spark.createDataFrame(data, schema)
df = df.withColumn("StartDate", to_date(col("StartDate")))
df = df.withColumn("EndDate", to_date(col("EndDate")))
# Show DataFrame
df.show()
```

ProjectID	ProjectName	EmployeeID	StartDate	EndDate
1	Website Redesign	2	2023-01-01	2023-03-31
2	Financial Analysis	3	2023-02-01	2023-04-30
3	HR Onboarding	1	2023-01-15	2023-02-28
4	Cloud Migration	4	2023-03-01	NULL

```
StructType  
[to_date(col("column_name"))]
```

```
None NULL
```

```
INSERT
```

```
INTO
```

```
from pyspark.sql import SparkSession from pyspark.sql.types import StructType,  
StructField, IntegerType, StringType, DateType from pyspark.sql.functions import  
col, to_date # Initialize Spark Session spark =  
SparkSession.builder.appName("CreateDataFrame").getOrCreate() # Define Schema  
schema = StructType([ StructField("ProjectID", IntegerType(), False),  
StructField("ProjectName", StringType(), False), StructField("EmployeeID",  
IntegerType(), False), StructField("StartDate", StringType(), False), # Initially  
as String to convert later StructField("EndDate", StringType(), True) # Nullable  
field ]) # Create Data data = [ (1, "Website Redesign", 2, "2023-01-01", "2023-03-  
31"), (2, "Financial Analysis", 3, "2023-02-01", "2023-04-30"), (3, "HR  
Onboarding", 1, "2023-01-15", "2023-02-28"), (4, "Cloud Migration", 4, "2023-03-  
01", None) # NULL in EndDate ] # Create DataFrame df = spark.createDataFrame(data,  
schema=schema) # Convert String Dates to DateType df = df.withColumn("StartDate",
```

```
to_date(col("StartDate"))) \ .withColumn("EndDate", to_date(col("EndDate"))) #  
Show DataFrame df.show()
```

ProjectID	ProjectName	EmployeeID	StartDate	EndDate
1	Website Redesign	2	2023-01-01	2023-03-31
2	Financial Analysis	3	2023-02-01	2023-04-30
3	HR Onboarding	1	2023-01-15	2023-02-28
4	Cloud Migration	4	2023-03-01	NULL

```
StructType  
StringType DateType to_date()  
None NULL
```

```
from pyspark.sql import SparkSession from pyspark.sql.types import StructType,  
StructField, IntegerType, StringType, FloatType # Initialize Spark Session spark =  
SparkSession.builder.appName("CreateDataFrame").getOrCreate() # Define Schema
```

```

schema = StructType([ StructField("EmployeeID", IntegerType(), False),
StructField("FirstName", StringType(), False), StructField("Department",
StringType(), False), StructField("Salary", IntegerType(), False),
StructField("TotalDeptSalary", IntegerType(), False), StructField("AvgDeptSalary",
FloatType(), False), StructField("SalaryRank", IntegerType(), False) ]) # Create
Data data = [ (1, "John", "HR", 5000, 10200, 5100.0, 2), (5, "Sarah", "HR", 5200,
10200, 5100.0, 1), (4, "Michael", "IT", 6500, 12500, 6250.0, 1), (2, "Jane", "IT",
6000, 12500, 6250.0, 2), (3, "Emily", "Finance", 5500, 5500, 5500.0, 1) ] # Create
DataFrame df = spark.createDataFrame(data, schema=schema) # Show DataFrame
df.show()

```

EmployeeID	FirstName	Department	Salary	TotalDeptSalary	AvgDeptSalary	SalaryRank
1	John	HR	5000	10200	5100.0	2
5	Sarah	HR	5200	10200	5100.0	1
4	Michael	IT	6500	12500	6250.0	1
2	Jane	IT	6000	12500	6250.0	2
3	Emily	Finance	5500	5500	5500.0	1

StructType

IntegerType()

FloatType()

.filter()

NULL
.where()

isNotNull()

```
from pyspark.sql import SparkSession from pyspark.sql.functions import col #
Initialize Spark Session spark =
SparkSession.builder.appName("FilterNotNull").getOrCreate() # Sample Data data = [
(1, "John", 5000), (2, "Jane", None), (3, "Michael", 6500), (4, "Emily", None),
(5, "Sarah", 5200) ] # Define Schema columns = ["EmployeeID", "Name", "Salary"] #
Create DataFrame df = spark.createDataFrame(data, schema=columns) # Show Original
DataFrame print("Original DataFrame:") df.show() # **Filter where Salary is NOT
NULL** df_filtered = df.filter(col("Salary").isNotNull()) # Show Filtered
DataFrame print("Filtered DataFrame (Salary NOT NULL):") df_filtered.show()
```

```
+-----+-----+-----+
|EmployeeID| Name |Salary|
+-----+-----+-----+
|      1 | John | 5000 |
|      2 | Jane | NULL  |
|      3 | Michael | 6500 |
|      4 | Emily | NULL  |
|      5 | Sarah | 5200 |
+-----+-----+-----+
```

```
+-----+-----+-----+
|EmployeeID| Name |Salary|
+-----+-----+-----+
|      1 | John | 5000 |
|      3 | Michael | 6500 |
|      5 | Sarah | 5200 |
+-----+-----+-----+
```

```
NULL
.where()
df_filtered = df.where(col("Salary").isNotNull())
df_filtered = df.filter("Salary IS NOT NULL")
```

```
.withColumn()  
F.lit() F.when()
```

```
from pyspark.sql import SparkSession from pyspark.sql.functions import lit #  
Initialize Spark Session spark =  
SparkSession.builder.appName("AddColumn").getOrCreate() # Sample Data data = [(1,  
"John", 5000), (2, "Jane", 6000), (3, "Michael", 7000)] columns = ["EmployeeID",  
"Name", "Salary"] # Create DataFrame df = spark.createDataFrame(data, columns) #  
Add a New Static Column df_new = df.withColumn("Country", lit("USA"))  
df_new.show()
```

```
+-----+-----+-----+  
|EmployeeID| Name |Salary|Country|  
+-----+-----+-----+  
| 1 | John | 5000 | USA |  
| 2 | Jane | 6000 | USA |  
| 3 | Michael | 7000 | USA |  
+-----+-----+-----+
```

```
F.when()  
from pyspark.sql.functions import when df_new = df.withColumn("Salary_Status",  
when(df.Salary > 6000, "High").otherwise("Low")) df_new.show()
```

```
+-----+-----+-----+  
|EmployeeID| Name |Salary|Salary_Status|  
+-----+-----+-----+-----+  
| 1 | John | 5000 | Low |  
| 2 | Jane | 6000 | Low |  
| 3 | Michael | 7000 | High |  
+-----+-----+-----+
```

```
from pyspark.sql.functions import expr df_new = df.withColumn("Bonus",  
expr("Salary * 0.10")) # 10% Bonus df_new.show()
```

```
from pyspark.sql.functions import udf from pyspark.sql.types import StringType #  
Define a UDF def categorize_salary(salary): return "High" if salary > 6000 else  
"Low" salary_udf = udf(categorize_salary, StringType()) # Add Column Using UDF  
df_new = df.withColumn("Salary_Category", salary_udf(df.Salary)) df_new.show()
```

```
lit()  
when().otherwise()  
expr()
```

split()

pyspark.sql.functions

split()

```
from pyspark.sql import SparkSession from pyspark.sql.functions import split #
Initialize Spark Session spark =
SparkSession.builder.appName("SplitString").getOrCreate() # Sample Data data =
[("PUSH:Mob:::1250255:MHPP:XDDG",)] columns = [ "input_string"] # Create DataFrame
df = spark.createDataFrame(data, columns) # Split String by Colon ":" df_split =
df.withColumn("split_values", split(df["input_string"], ":")) # Show Results
df_split.show(truncate=False)
```

```
+-----+-----+
| input_string | split_values |
+-----+-----+
| PUSH:Mob:::1250255:MHPP:XDDG | [PUSH, Mob, , , 1250255, MHPP, XDDG] |
+-----+-----+
```

filter()
split()

⋮

```
from pyspark.sql.functions import expr df_clean = df.withColumn("split_values",
expr("filter(split(input_string, ':'), x -> x != '')"))
df_clean.show(truncate=False)
```

```
+-----+-----+
| input_string | split_values |
+-----+-----+
| PUSH:Mob:::1250255:MHPP:XDDG | [PUSH, Mob, 1250255, MHPP, XDDG] |
+-----+-----+
```

```
split(df["input_string"], ":")  
":"  
  
filter(split(input_string, ":"), x -> x != '')
```

split() getItem(index)

```
from pyspark.sql import SparkSession from pyspark.sql.functions import split, expr  
# Initialize Spark Session spark =  
SparkSession.builder.appName("SplitString").getOrCreate() # Sample Data data =  
[("PUSH:Mob:::1250255:MHPP:XDDG",)] columns = ["input_string"] # Create DataFrame  
df = spark.createDataFrame(data, columns) # Split String and Remove Empty Strings  
df_split = df.withColumn("split_values", expr("filter(split(input_string, ':'), x  
-> x != '')")) # Extract Specific Elements df_final = (  
df_split.withColumn("col1", df_split["split_values"].getItem(0)) # "PUSH"  
.withColumn("col2", df_split["split_values"].getItem(1)) # "Mob"  
.withColumn("col3", df_split["split_values"].getItem(2)) # "1250255"  
.withColumn("col4", df_split["split_values"].getItem(3)) # "MHPP"  
.withColumn("col5", df_split["split_values"].getItem(4)) # "XDDG" ) # Show Result  
df_final.select("input_string", "col1", "col2", "col3", "col4",  
"col5").show(truncate=False)
```

```
+-----+-----+-----+-----+-----+  
| input_string | col1| col2| col3 | col4| col5|  
+-----+-----+-----+-----+-----+  
| PUSH:Mob:::1250255:MHPP:XDDG | PUSH| Mob| 1250255| MHPP| XDDG|  
+-----+-----+-----+-----+-----+
```

```
split(input_string, ':')  
filter(..., x -> x != '')  
getItem(index)
```

⋮⋮

isin()

isin()

isin()

```
from pyspark.sql import SparkSession from pyspark.sql.functions import split, expr  
# Initialize Spark Session spark =  
SparkSession.builder.appName("IsInExample").getOrCreate() # Sample Data data =  
[("PUSH:Mob:::1250255:MHPP:XDDG",), ("TEST:Web:::789456:ABCD:QWER",),  
("PUSH:Tab:::123456:XYZ:LMNO",)] columns = ["input_string"] # Create DataFrame df =  
spark.createDataFrame(data, columns) # Split String and Remove Empty Strings  
df_split = df.withColumn("split_values", expr("filter(split(input_string, ':'), x  
-> x != '')")) # Extract First Column for Filtering df_filtered =  
df_split.withColumn("col1", df_split["split_values"].getItem(0)) # Apply isin() to  
Filter Data Where col1 is in a Given List df_filtered =  
df_filtered.filter(df_filtered["col1"].isin("PUSH", "TEST")) # Show Result  
df_filtered.select("input_string", "col1").show(truncate=False)
```

input_string	col1
--------------	------

```

| PUSH:Mob:::1250255:MHPP:XDDG | PUSH|
| TEST:Web:::789456:ABCD:QWER | TEST|
| PUSH:Tab:::123456:XYZ:LMNO | PUSH|
+-----+-----+

```

```

split(input_string, ':')
filter(..., x -> x != '')
getItem(0) :::
df.filter(df["col1"].isin("PUSH", "TEST"))
    col1 :::

```

col1

when()	isin()
CASE WHEN	

```

when()      isin()
from pyspark.sql import SparkSession from pyspark.sql.functions import col, when,
split, expr # Initialize Spark Session spark =
SparkSession.builder.appName("WhenIsInExample").getOrCreate() # Sample Data data =
[("PUSH:Mob:::1250255:MHPP:XDDG",), ("TEST:Web:::789456:ABCD:QWER",),
("PULL:Tab:::123456:XYZ:LMNO",)] columns = ["input_string"] # Create DataFrame df
= spark.createDataFrame(data, columns) # Split String and Remove Empty Strings
df_split = df.withColumn("split_values", expr("filter(split(input_string, ':'), x
-> x != '')")) # Extract First Column df_transformed = df_split.withColumn("col1",
df_split["split_values"].getItem(0)) # Apply "when" and "isin" to Create a New
Column df_final = df_transformed.withColumn( "Category",
when(col("col1").isin("PUSH", "TEST"), "Allowed") # If col1 is PUSH or TEST, mark
as Allowed .otherwise("Blocked") # Otherwise, mark as Blocked ) # Show Result
df_final.select("input_string", "col1", "Category").show(truncate=False)

```

input_string	col1	Category
PUSH:Mob:::1250255:MHPP:XDDG	PUSH	Allowed
TEST:Web:::789456:ABCD:QWER	TEST	Allowed
PULL:Tab:::123456:XYZ:LMNO	PULL	Blocked

```
split(input_string, ':')
filter(..., x -> x != '')
getItem(0) :::
when(col("col1").isin("PUSH", "TEST"), "Allowed")      col1
    "Allowed"
.otherwise("Blocked")      "Blocked"

when()      isin()
```

while

while

while

while

```
i = 1 while i <= 3: # Outer while loop j = 1 while j <= 2: # Inner while loop
print(f"Iteration: i={i}, j={j}") j += 1 i += 1
```

```
Iteration: i=1, j=1
Iteration: i=1, j=2
Iteration: i=2, j=1
Iteration: i=2, j=2
Iteration: i=3, j=1
Iteration: i=3, j=2
```

while

while

while

```
from pyspark.sql import SparkSession from pyspark.sql.functions import col #
Initialize Spark Session spark =
SparkSession.builder.appName("WhileExample").getOrCreate() # Sample DataFrame data
= [(1, "Alice"), (2, "Bob"), (3, "Cathy"), (4, "David")] df =
spark.createDataFrame(data, ["ID", "Name"]) # Example of modifying DataFrame using
a while loop iteration = 0 while iteration < 3: df = df.withColumn("Iteration",
col("ID") + iteration) # Modify DataFrame in each loop print(f"Iteration
{iteration}:") df.show() iteration += 1
```

while

withColumn() select() filter()

```
while
```

```
F.when()      when()
CASE WHEN    pyspark.sql.functions
```

```
when()
from pyspark.sql import SparkSession from pyspark.sql.functions import col, when #
Initialize Spark Session spark =
SparkSession.builder.appName("MultipleWhenExample").getOrCreate() # Sample Data
data = [(1, "John", 5000), (2, "Jane", 7000), (3, "Mike", 4000), (4, "Sara",
9000), (5, "David", 3000)] columns = ["ID", "Name", "Salary"] # Create DataFrame
df = spark.createDataFrame(data, columns) # Apply multiple "when" conditions
df_new = df.withColumn( "Salary_Category", when(col("Salary") > 8000, "High")
.when((col("Salary") > 5000) & (col("Salary") <= 8000), "Medium")
.when((col("Salary") > 3000) & (col("Salary") <= 5000), "Low") .otherwise("Very
Low") # Default case ) # Show result df_new.show()
```

ID	Name	Salary	Salary_Category
1	John	5000	Low
2	Jane	7000	Medium
3	Mike	4000	Low
4	Sara	9000	High
5	David	3000	Very Low

```
when(col("Salary") > 8000, "High")
    "High"
.when((col("Salary") > 5000) & (col("Salary") <= 8000), "Medium")
    "Medium"
.when((col("Salary") > 3000) & (col("Salary") <= 5000), "Low")
    "Low"
.otherwise("Very Low")
    "Very Low"
```

```
when()
.otherwise()
> < >= <=
& |
```

when() isin()

```
array_contains
compaignTag
array_contains()
```

pyspark.sql.functions
"SMS" isin()

```
array_contains()
from pyspark.sql import SparkSession from pyspark.sql.functions import col,
array_contains # Initialize Spark Session spark =
SparkSession.builder.appName("FilterArrayColumn").getOrCreate() # Sample Data data
= [ (1, ["SMS", "Trnx", "orderFready"]), (2, ["Trnx", "SMS", "orderFready"]),
(3, ["LM", "XY", "SMS"]), (4, ["LM", "XY"])] columns = ["ID", "compaignTag"] # Create DataFrame df = spark.createDataFrame(data, columns) # Apply filter where "SMS" is present in compaignTag column filtered_df =
```

```
df.filter(array_contains(col("compaignTag"), "SMS")) # Show result  
filtered_df.show(truncate=False)
```

```
+---+-----+  
| ID | compaignTag          |  
+---+-----+  
| 1 | [SMS, Trnx, orderFready] |  
| 2 | [Trnx, SMS, orderFready] |  
| 3 | [LM, XY, SMS]           |  
+---+-----+
```

```
array_contains(col("compaignTag"), "SMS")  
"SMS"  
.filter()  
  
isin()
```

```
when()      isin()  
"SMS"       isin()  
array_contains()
```

```
when()      array_contains()
```

```
from pyspark.sql import SparkSession from pyspark.sql.functions import col, when,  
array_contains # Initialize Spark Session spark =  
SparkSession.builder.appName("WhenIsinExample").getOrCreate() # Sample Data data =  
[ (1, ["SMS", "Trnx", "orderFready"]), (2, ["Trnx", "SMS", "orderFready"]), (3,  
["LM", "XY", "SMS"]), (4, ["LM", "XY"])], columns = ["ID", "compaignTag"] #
```

```
Create DataFrame df = spark.createDataFrame(data, columns) # Apply when() and  
array_contains() to create a new column df_new = df.withColumn( "Contains_SMS",  
when(array_contains(col("compaignTag"), "SMS"), "Yes").otherwise("No") ) # Show  
result df_new.show(truncate=False)
```

ID	compaignTag	Contains_SMS
1	[SMS, Trnx, orderFready]	Yes
2	[Trnx, SMS, orderFready]	Yes
3	[LM, XY, SMS]	Yes
4	[LM, XY]	No

```
array_contains(col("compaignTag"), "SMS")
```

```
"SMS"
```

```
when(..., "Yes").otherwise("No")
```

```
"Yes"
```

```
"SMS"
```

```
"No"
```

```
array_contains()
```

```
"SMS" "Trnx"
```

```
when()
```

```
df_new = df.withColumn( "Contains_SMS_Or_Trnx",  
when(array_contains(col("compaignTag"), "SMS"), "Has SMS")  
.when(array_contains(col("compaignTag"), "Trnx"), "Has Trnx") .otherwise("No  
Match") ) df_new.show(truncate=False)
```

```
explode()
```

```
explode()
```

```
explode()
```

```
compaignTag
```

```
from pyspark.sql import SparkSession from pyspark.sql.functions import col,
explode # Initialize Spark Session spark =
SparkSession.builder.appName("ExplodeExample").getOrCreate() # Sample Data data =
[ (1, ["SMS", "Trnx", "orderFready"]), (2, ["Trnx", "SMS", "orderFready"]),
(3, ["LM", "XY", "SMS"]),
(4, ["LM", "XY"])] columns = ["ID", "compaignTag"] # Create DataFrame df = spark.createDataFrame(data, columns) # Explode the array column df_exploded = df.withColumn("Tag", explode(col("compaignTag")))) # Show result df_exploded.show(truncate=False)
```

ID	compaignTag	Tag
1	[SMS, Trnx, orderFready]	SMS
1	[SMS, Trnx, orderFready]	Trnx
1	[SMS, Trnx, orderFready]	orderFready
2	[Trnx, SMS, orderFready]	Trnx
2	[Trnx, SMS, orderFready]	SMS
2	[Trnx, SMS, orderFready]	orderFready
3	[LM, XY, SMS]	LM
3	[LM, XY, SMS]	XY
3	[LM, XY, SMS]	SMS
4	[LM, XY]	LM
4	[LM, XY]	XY

"SMS"

"SMS"

```
df_filtered = df_exploded.filter(col("Tag") == "SMS")
df_filtered.show(truncate=False)
```

ID	comaignTag	Tag
1	[SMS, Trnx, orderFready]	SMS
2	[Trnx, SMS, orderFready]	SMS
3	[LM, XY, SMS]	SMS

when()

explode()

Tag

"SMS"

```
from pyspark.sql.functions import when
df_with_flag = df_exploded.withColumn(
    "Is_SMS", when(col("Tag") == "SMS", "Yes").otherwise("No"))
df_with_flag.show(truncate=False)
```

ID	comaignTag	Tag	Is_SMS
1	[SMS, Trnx, orderFready]	SMS	Yes
1	[SMS, Trnx, orderFready]	Trnx	No
1	[SMS, Trnx, orderFready]	orderFready	No
2	[Trnx, SMS, orderFready]	Trnx	No
2	[Trnx, SMS, orderFready]	SMS	Yes
2	[Trnx, SMS, orderFready]	orderFready	No
3	[LM, XY, SMS]	LM	No
3	[LM, XY, SMS]	XY	No
3	[LM, XY, SMS]	SMS	Yes
4	[LM, XY]	LM	No
4	[LM, XY]	XY	No

```
explode()
```

```
when()
```

```
filter()
```

```
array_contains()  
array_contains()  
expr()      array_intersect()
```

```
pyspark.sql.functions  
array_contains()
```

```
array_contains()  
from pyspark.sql import SparkSession from pyspark.sql.functions import col,  
array_contains # Initialize Spark Session spark =  
SparkSession.builder.appName("ArrayContainsExample").getOrCreate() # Sample Data  
data = [ (1, ["SMS", "Trnx", "orderFready"]), (2, ["Trnx", "SMS", "orderFready"]),
(3, ["LM", "XY", "SMS"]), (4, ["LM", "XY"]) ] columns = ["ID", "compaignTag"] #  
Create DataFrame df = spark.createDataFrame(data, columns) # Filter rows where  
'compaignTag' contains either 'SMS' OR 'Trnx' df_filtered =  
df.filter(array_contains(col("compaignTag"), "SMS") |  
array_contains(col("compaignTag"), "Trnx")) df_filtered.show(truncate=False)
```

```
+---+-----+  
| ID| compaignTag |  
+---+-----+  
| 1| [SMS, Trnx, orderFready] |  
| 2| [Trnx, SMS, orderFready] |  
| 3| [LM, XY, SMS] |  
+---+-----+
```

```
array_intersect()
```

```
array_intersect()
```

```
from pyspark.sql.functions import expr # Define the values to check
values_to_check = ["SMS", "Trnx"] # Filter using array_intersect()
df_filtered = df.filter(expr("size(array_intersect(compaignTag, array{values_to_check})) > 0"))
df_filtered.show(truncate=False)
```

```
+---+-----+
| ID| compaignTag           |
+---+-----+
| 1 | [SMS, Trnx, orderFready] |
| 2 | [Trnx, SMS, orderFready] |
| 3 | [LM, XY, SMS]          |
+---+-----+
```

```
array_contains(col, "value")  
||
```

```
array_contains()
```

```
array_intersect()
```

```
size(array_intersect()) > 0
```

```
when    array_contains
```

```
array_contains()  
      ["SMS", "XY"]  
      array_contains(col("compaignTag"), "SMS") |  
      array_contains(col("compaignTag"), "XY")
```

```
from pyspark.sql import SparkSession from pyspark.sql.functions import col, when, array_contains # Initialize Spark Session spark = SparkSession.builder.appName("ArrayContainsExample").getOrCreate() # Sample Data data = [ (1, ["SMS", "Trnx", "orderFready"]), (2, ["Trnx", "SMS", "orderFready"]),
(3, ["LM", "XY", "SMS"]), (4, ["LM", "XY"]) ] columns = ["ID", "compaignTag"] # Create DataFrame df = spark.createDataFrame(data, columns) # Fix: Use OR (`|`) to check multiple values separately df_new = df.withColumn( "Contains_SMS_Or_Trnx",
when(array_contains(col("compaignTag"), "SMS") | array_contains(col("compaignTag"), "XY"), "Has SMS")
.when(array_contains(col("compaignTag"), "Trnx"), "Has Trnx") .otherwise("No Match") ) df_new.show(truncate=False)
```

ID	compaignTag	Contains_SMS_Or_Trnx
1	[SMS, Trnx, orderFready]	Has SMS
2	[Trnx, SMS, orderFready]	Has SMS
3	[LM, XY, SMS]	Has SMS
4	[LM, XY]	Has SMS

```
array_contains(col("compaignTag"), ["SMS", "XY"])
```

```
    array_contains(col("compaignTag"), "SMS") |
    array_contains(col("compaignTag"), "XY")|
.otherwise("No Match")
```

```
F.expr()
```

```
array_contains
```

```
array_intersect()
```

```
compaignTag
```

```
from pyspark.sql import SparkSession from pyspark.sql.functions import col, when, expr, array_intersect # Initialize Spark Session spark = SparkSession.builder.appName("ArrayContainsExample").getOrCreate() # Sample Data data = [ (1, ["SMS", "Trnx", "orderFready"]), (2, ["Trnx", "SMS", "orderFready"]), (3, ["LM", "XY", "SMS"]), (4, ["LM", "XY"]), (5, ["AB", "CD"]) # No match case ] columns = ["ID", "compaignTag"] # Create DataFrame df = spark.createDataFrame(data, columns) # Define list of values to check search_values = ["SMS", "XY", "Trnx"] # Use array_intersect to find matches df_new = df.withColumn( "Contains_SMS_Or_Trnx", when(expr(f"size(array_intersect(compaignTag, array({', '.join([f'\"{v}\\"' for v in search_values]})))) > 0"), "Has Match") .otherwise("No Match") ) df_new.show(truncate=False)
```

ID	compaignTag	Contains_SMS_Or_Trnx
1	[SMS, Trnx, orderFready]	Has Match
2	[Trnx, SMS, orderFready]	Has Match
3	[LM, XY, SMS]	Has Match
4	[LM, XY]	Has Match
5	[AB, CD]	No Match

```
array_intersect(compaignTag, array(...))
```

```
compaignTag  
search_values  
size(array_intersect(...)) > 0  
"Has Match"
```

```
search_values = ["SMS", "XY", "Trnx"]
```

```
struct      array
```

```
struct
```

```
struct
```

```
from pyspark.sql import Row
data = [ Row(id=1, person=Row(name="Alice", age=30)),
Row(id=2, person=Row(name="Bob", age=25)) ]
df = spark.createDataFrame(data)
df.createOrReplaceTempView("people")
```

```
SELECT id, person.name AS name, person.age AS age FROM people;
```

```
array
```

```
from pyspark.sql import Row
data = [ Row(id=1, tags=["big", "fast"]),
Row(id=2, tags=["small", "slow"]),
Row(id=3, tags=["fast", "silent"]) ]
df = spark.createDataFrame(data)
df.createOrReplaceTempView("items")
```

```
SELECT id, tag FROM items LATERAL VIEW explode(tags) t AS tag;
```

```
SELECT * FROM items WHERE array_contains(tags, 'fast');
```

```
from pyspark.sql import Row
data = [ Row(id=1, details=Row(name="John", phones=["123", "456"])), Row(id=2, details=Row(name="Jane", phones=["789"])) ]
df = spark.createDataFrame(data)
df.createOrReplaceTempView("contacts")
```

```
SELECT id, details.name AS name, details.phones[0] AS primary_phone FROM contacts;
```

```
SELECT id, details.name, phone FROM contacts LATERAL VIEW explode(details.phones)
t AS phone;
```

array_contains	Check if array has a specific value
explode()	Flatten array to rows
LATERAL VIEW	Used with explode in SQL
struct_field.subfield	Access subfields of structs

from_json

```
from pyspark.sql.functions import from_json, col from pyspark.sql.types import StructType, StringType # Sample data data = [ '{"name": "Alice", "age": 30 }',) df = spark.createDataFrame(data, ["json_string"]) # Define schema schema = StructType().add("name", StringType()).add("age", StringType()) # Parse JSON into struct df_parsed = df.select(from_json(col("json_string"), schema).alias("parsed")) df_parsed.select("parsed.name", "parsed.age").show()
```

```
from pyspark.sql import Row data = [ Row(id=1, items=[Row(name="apple", price=2), Row(name="banana", price=1)]), Row(id=2, items=[Row(name="orange", price=3)]) ] df = spark.createDataFrame(data) df.createOrReplaceTempView("orders")
```

```
SELECT id, item.name AS item_name, item.price FROM orders LATERAL VIEW  
explode(items) exploded AS item;
```

```
from pyspark.sql.functions import explode df.select("id",  
explode("items").alias("item")) \ .select("id", "item.name", "item.price") \  
.show()
```

```
data = [ (1, [{"type": "SMS", "active": True}, {"type": "Email", "active": False}]), (2, [{"type": "Push", "active": True}]) ] schema = "id INT, channels ARRAY<STRUCT<type:STRING, active:BOOLEAN>>" df = spark.createDataFrame(data, schema) df.createOrReplaceTempView("campaigns")
```

```
SELECT id, ch.type FROM campaigns LATERAL VIEW explode(channels) t AS ch WHERE ch.active = true;
```

```
df.select("id", explode("channels").alias("channel")) \  
.filter(col("channel.active") == True) \  
.select("id", "channel.type") \  
.show()
```

from_json	Parse string to struct
explode	Flatten arrays to rows
array_contains	Check value in array
LATERAL VIEW	Used with explode in Spark SQL
struct.field	Access fields inside a struct

```
LATERAL VIEW explode
```

```
LATERAL VIEW explode
```

id	colors
----	--------

```
from pyspark.sql import SparkSession from pyspark.sql import Row spark =  
SparkSession.builder.appName("LateralViewExplodeExample").getOrCreate() data = [  
Row(id=1, colors=["red", "blue", "green"]), Row(id=2, colors=["yellow",  
"orange"]), Row(id=3, colors=[])] df = spark.createDataFrame(data)  
df.createOrReplaceTempView("color_table")
```

```
LATERAL VIEW explode
```

```
SELECT id, color FROM color_table LATERAL VIEW explode(colors) t AS color;
```

```
explode(colors)  
LATERAL VIEW  
AS color
```

id	color
----	-------

1	red

1	blue
1	green
2	yellow
2	orange

```
from pyspark.sql.functions import explode df_exploded = df.select("id",  
explode("colors").alias("color")) df_exploded.show()
```

LATERAL VIEW OUTER explode

Pyspark Realworld Qna

Pyspark Realworld Qna

Pyspark Realworld Qna

```
python
# PySpark Real-World DataFrame Use Cases (Part 2)

```python
from pyspark.sql import SparkSession, Window
from pyspark.sql.functions import *
from pyspark.sql.types import *
from datetime import datetime, timedelta

spark = SparkSession.builder.appName("RealWorldAnalytics").getOrCreate()
```

---

### 1. Daily user retention for a 30-day cohort window

```python
data = [(1, '2023-01-01'), (1, '2023-01-02'), (2, '2023-01-01'), (2, '2023-01-03'), (3, '2023-01-02')]
df = spark.createDataFrame(data, ["user_id", "login_date"]).withColumn("login_date", to_date("login_date"))

cohort = df.groupBy("user_id").agg(min("login_date").alias("cohort_date"))
df = df.join(cohort, on="user_id")
df = df.withColumn("days_since_signup", datediff("login_date", "cohort_date"))
df.groupBy("cohort_date", "days_since_signup").agg(countDistinct("user_id").alias("retained_users")).show()
```

---

### 2. Latest 3 events per user

```python
events = [(1, 'login', '2023-01-01 10:00'), (1, 'click', '2023-01-01 10:05'),
 (1, 'logout', '2023-01-01 10:10'),
 (1, 'login', '2023-01-02 10:00'), (2, 'login', '2023-01-01 09:00')]
df = spark.createDataFrame(events, ["user_id", "event", "ts"]).withColumn("ts", to_timestamp("ts"))

w = Window.partitionBy("user_id").orderBy(desc("ts"))
df.withColumn("rn", row_number().over(w)).filter("rn <= 3").show()
```

```

```
mergeSchema
```

```
mergeSchema=True
```

```
df_new.write.mode("append").option("mergeSchema",  
"true").parquet("/mnt/data/events")
```

```
df_new.write.format("delta") \ .mode("append") \ .option("mergeSchema", "true") \  
.save("/mnt/data/delta/events")
```

```
from pyspark.sql import DataFrame def union_all_by_name(dfs): return reduce(lambda df1, df2: df1.unionByName(df2, allowMissingColumns=True), dfs) # Example usage:  
df1 = spark.read.csv("/mnt/data/batch1.csv", header=True, inferSchema=True) df2 =  
spark.read.csv("/mnt/data/batch2.csv", header=True, inferSchema=True) final_df =  
union_all_by_name([df1, df2])  
final_df.write.mode("append").parquet("/mnt/data/final")
```

```
df_new.write.format("delta") \ .mode("append") \ .option("mergeSchema", "true") \  
.save("/mnt/delta/events")
```

```
df_new.write.mode("append").parquet("/mnt/data/events")
```

```
.mode("overwrite")
```

```
# Simulating schema drift df1 = spark.createDataFrame([(1, "Alice")], ["id", "name"]) df2 = spark.createDataFrame([(2, "Bob", 30)], ["id", "name", "age"]) # Merge and append safely df1.write.mode("overwrite").option("mergeSchema", "true").parquet("/mnt/data/users") df2.write.mode("append").option("mergeSchema", "true").parquet("/mnt/data/users") # Check merged result merged = spark.read.option("mergeSchema", "true").parquet("/mnt/data/users") merged.show()
```

```
+---+-----+---+
| id| name| age|
+---+-----+---+
| 1|Alice| null|
| 2| Bob|  30|
+---+-----+---+
```

```
/mnt/delta/sales_data
```

```
new_df = spark.read.format("csv").option("header",  
True).load("/mnt/raw/new_sales/")
```

```
merge
```

```
from delta.tables import DeltaTable # Load target Delta table target_path =  
"/mnt/delta/sales_data" delta_table = DeltaTable.forName(spark, target_path) #  
Assume 'id' is the unique key for identifying new data ( delta_table.alias("t")  
.merge( new_df.alias("s"), "t.id = s.id" # matching condition )  
.whenNotMatchedInsertAll() # only insert new records .execute() )
```

id
id

```
target_df = spark.read.format("delta").load("/mnt/delta/sales_data") # Identify  
only new records based on a key column (e.g., id) new_records =  
new_df.join(target_df, on="id", how="left_anti") # Append only new records  
new_records.write.format("delta").mode("append").save("/mnt/delta/sales_data")
```

left_anti

MERGE INTO

```
MERGE INTO delta.`/mnt/delta/sales_data` AS target USING  
delta.`/mnt/delta/new_data` AS source ON target.id = source.id WHEN NOT MATCHED  
THEN INSERT *
```

.merge()

```
( delta_table.alias("t") .merge( new_df.alias("s"), "t.id = s.id" )
    .whenNotMatchedInsertAll() .execute() )
```

```
new_df.write.format("delta") \ .mode("append") \ .option("mergeSchema", "true") \
    .save("/mnt/delta/sales_data")
```

| | |
|------------------------------|--|
| | |
| Only append new records | <code>merge</code> with <code>whenNotMatchedInsertAll()</code> |
| Some records need update too | Add <code>.whenMatchedUpdateAll()</code> |
| Detect duplicates | Use <code>dropDuplicates()</code> before write |
| Schema changes over time | Use <code>.option("mergeSchema", "true")</code> |
| High frequency ingestion | Use + Delta MERGE |

```
from delta.tables import DeltaTable # Create or read target target_path =
"/mnt/delta/sales_data" new_df = spark.createDataFrame( [(1, "A", 100), (2, "B",
200), (3, "C", 300)], ["id", "product", "amount"] ) # If table doesn't exist,
create try: delta_table = DeltaTable.forPath(spark, target_path) except:
new_df.write.format("delta").save(target_path) delta_table =
DeltaTable.forPath(spark, target_path) # New arriving batch new_batch =
spark.createDataFrame( [(2, "B", 200), (3, "C", 350), (4, "D", 400)], ["id",
"product", "amount"] ) # Merge only new records delta_table.alias("t") \
```

```
.merge(new_batch.alias("s"), "t.id = s.id") \\ .whenNotMatchedInsertAll() \\  
.execute()
```

| | | |
|---|---|-----|
| | | |
| 1 | A | 100 |
| 2 | B | 200 |
| 3 | C | 300 |
| 4 | D | 400 |

ChatGPT can make mistakes. Check important info. See [Cookie Preferences](#).