

PySpark Reference

From first DataFrame to production-grade distributed data processing.

Last reviewed 29 Sep 2026 · Lab-tested with PySpark 4.2.0 ·

Prerequisites: [Python for DE](https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/python-reference/) (https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/python-reference/) · [SQL](https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/sql-reference/) (https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/sql-reference/)

Related: [Databricks](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/data-bricks-reference/) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/data-bricks-reference/) · [Apache Iceberg](https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/apache-iceberg/) (https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/apache-iceberg/) · [Kafka](https://sarangambekar1997.github.io/data-engineering-handbook/04-streaming/kafka-reference/) (https://sarangambekar1997.github.io/data-engineering-handbook/04-streaming/kafka-reference/) · [DuckDB & Polars](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/duckdb-polars/) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/duckdb-polars/) · [Glossary](https://sarangambekar1997.github.io/data-engineering-handbook/09-reference/glossary/) (https://sarangambekar1997.github.io/data-engineering-handbook/09-reference/glossary/)

Practice: [Lab 03 — Spark Lakehouse](https://github.com/sarangambekar1997/data-engineering-handbook/tree/main/labs/03-spark-lakehouse) (https://github.com/sarangambekar1997/data-engineering-handbook/tree/main/labs/03-spark-lakehouse)

Overview

Challenge: Single-machine tools such as pandas work well until the data no longer fits in memory. A typical server has tens of gigabytes of RAM, while production datasets can reach many terabytes.

Solution: Spark runs on a **cluster of machines**. It splits data into partitions, distributes them across the cluster, processes them in parallel, and combines the results, so work that would take hours on one machine completes in minutes.

Pandas:
One machine
- 64GB RAM limit
- 1 CPU cores
- Fast to learn
- Great for < 1GB data

PySpark:
Driver + 50 Workers
- 50 × 64GB = 3.2TB RAM
- 50 × 32 cores = 1600
- Handles petabytes
- Same DataFrame API

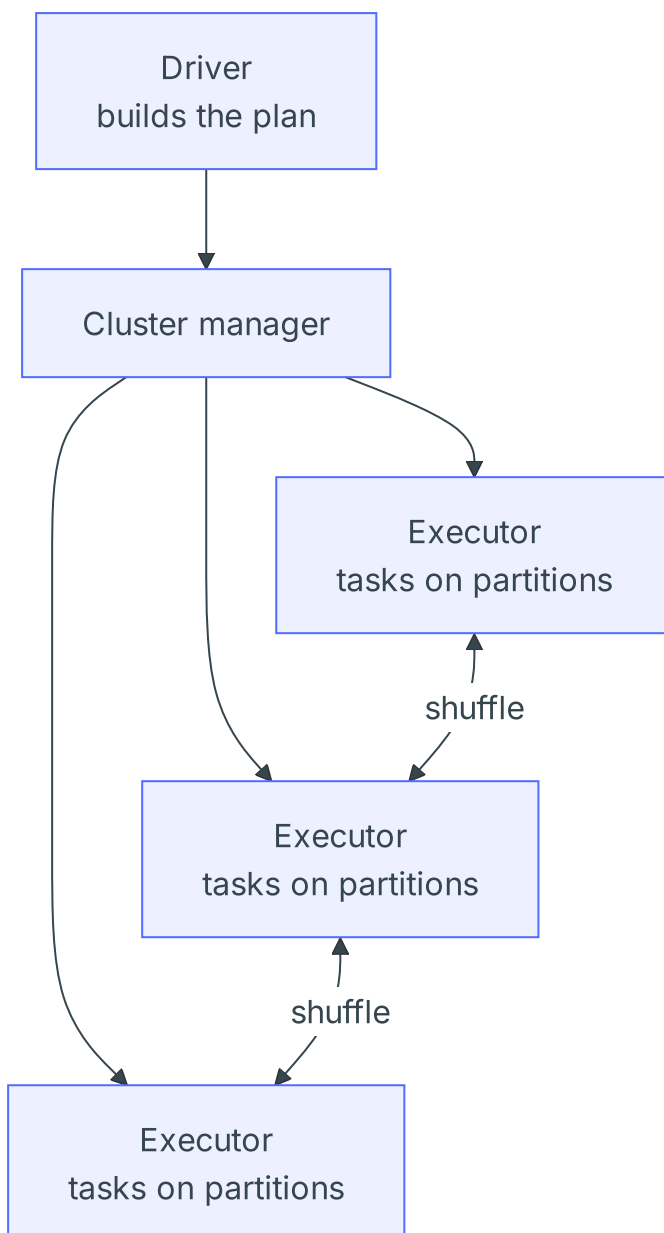


Rule of thumb:

data fits in RAM → Pandas

data is 10GB+ → PySpark (or Spark on Databricks)

Key concept — lazy evaluation: A chain such as `df.filter(...).groupBy(...).agg(...)` does not execute immediately; Spark builds a logical plan. Execution starts only when an *action* (`.show()`, `.count()`, `.write()`) is called, which allows Spark to optimize the entire chain before reading any data.



On this page

Basics - [What is Spark?](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/pyspark-reference/#what-is-spark) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/pyspark-reference/#what-is-spark) - [SparkSession](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/pyspark-reference/#sparksession) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/pyspark-reference/#sparksession) - [Reading Data](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/pyspark-reference/#reading-data) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/pyspark-reference/#reading-data) - [DataFrame Basics](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/pyspark-reference/#dataframe-basics) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/pyspark-reference/#dataframe-basics) - [Selecting & Renaming Columns](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/pyspark-reference/#selecting--renaming-columns) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/pyspark-reference/#selecting--renaming-columns) - [Filtering Rows](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/pyspark-reference/#filtering-rows) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/pyspark-reference/#filtering-rows)

Intermediate - [Transformations vs Actions](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/pyspark-reference/#transformations-vs-actions) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/pyspark-reference/#transformations-vs-actions) - [Adding & Transforming Columns](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/pyspark-reference/#adding--transforming-columns) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/pyspark-reference/#adding--transforming-columns) - [Aggregations & GroupBy](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/pyspark-reference/#aggregations--groupby) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/pyspark-reference/#aggregations--groupby) - [Joins](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/pyspark-reference/#joins) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/pyspark-reference/#joins) - [Sorting & Limiting](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/pyspark-reference/#sorting--limiting) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/pyspark-reference/#sorting--limiting) - [Handling Nulls](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/pyspark-reference/#handling-nulls) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/pyspark-reference/#handling-nulls) - [Writing Data](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/pyspark-reference/#writing-data) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/pyspark-reference/#writing-data)

Advanced - [Window Functions](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/pyspark-reference/#window-functions) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/pyspark-reference/#window-functions) - [User-Defined Functions \(UDFs\)](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/pyspark-reference/#user-defined-functions-udfs) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/pyspark-reference/#user-defined-functions-udfs) - [Spark SQL](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/pyspark-reference/#spark-sql) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/pyspark-reference/#spark-sql) - [Partitioning & Repartitioning](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/pyspark-reference/#partitioning--repartitioning) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/pyspark-reference/#partitioning--repartitioning) - [Caching & Persistence](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/pyspark-reference/#caching--persistence) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/pyspark-reference/#caching--persistence) - [Broadcast Joins](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/pyspark-reference/#broadcast-joins) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/pyspark-reference/#broadcast-joins) - [Query Optimization & EXPLAIN](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/pyspark-reference/#query-optimization--explain) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/pyspark-reference/#query-optimization--explain) - [Structured Streaming](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/pyspark-reference/#structured-streaming) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/pyspark-reference/#structured-streaming) - [Common Patterns](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/pyspark-reference/#common-patterns) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/pyspark-reference/#common-patterns)

Reference - [Common Pitfalls](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/pyspark-reference/#common-pitfalls) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/pyspark-reference/#common-pitfalls) - [Cheat Sheet](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/pyspark-reference/#cheat-sheet) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/pyspark-reference/#cheat-sheet) - [Interview Questions](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/pyspark-reference/#interview-questions) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/pyspark-reference/#interview-questions) - [Further Reading](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/pyspark-reference/#further-reading) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/pyspark-reference/#further-reading)

What is Spark?

Apache Spark is a distributed computing engine — it splits large datasets across many machines (a **cluster**) and processes them in parallel.

Architecture

```
Driver (your Python script)
├── SparkContext → cluster manager (YARN / Kubernetes / Databricks)
└── Executors (workers – where data actually lives and code runs)
    ├── Executor 1 [Partition 1] [Partition 2]
    ├── Executor 2 [Partition 3] [Partition 4]
    └── Executor 3 [Partition 5] [Partition 6]
```

- **Driver** — orchestrates the job; runs your Python code; sends tasks to executors
- **Executor** — JVM process on a worker node; holds data partitions and runs tasks
- **Partition** — a chunk of the dataset processed as a unit; parallelism = number of partitions
- **Task** — one unit of work on one partition on one executor

RDD vs DataFrame vs Dataset

	RDD	DataFrame	Dataset
API level	Low-level	High-level	High-level
Optimization	None	Catalyst optimizer	Catalyst optimizer
Type safety	Python types	Schema-based	Compile-time (JVM only)
Language	All	All	Scala/Java only
Use today?	Rarely	Yes — default	N/A in PySpark

Always use **DataFrames** in PySpark. RDDs offer no optimization and are much harder to work with. The Catalyst optimizer rewrites your DataFrame operations into an efficient execution plan automatically.

SparkSession

`SparkSession` is the entry point to all Spark functionality.

```
from pyspark.sql import SparkSession

# Local mode — uses all CPU cores on your machine
spark = SparkSession.builder \
    .appName("my_pipeline") \
    .master("local[*]") \
    .getOrCreate()

# With configuration
spark = SparkSession.builder \
    .appName("orders_pipeline") \
    .master("local[*]") \
    .config("spark.sql.shuffle.partitions", "8") \
    .config("spark.sql.adaptive.enabled", "true") \
    .getOrCreate()

# On a cluster (YARN / Kubernetes) — master is set by
# spark-submit
spark = SparkSession.builder \
    .appName("orders_pipeline") \
    .getOrCreate()

# Access SparkContext
sc = spark.sparkContext

# Stop the session when done (important in scripts, not
# notebooks)
spark.stop()
```

Key config options

Config	Default	What it controls
<code>spark.sql.shuffle.partitions</code>	200	Partitions after a shuffle (join, groupBy) — set to 2–4× your cores for local dev

Config	Default	What it controls
<code>spark.sql.adaptive.enabled</code>	true (3.x)	Auto-tunes partitions during execution
<code>spark.executor.memory</code>	1g	Memory per executor
<code>spark.executor.cores</code>	1	Cores per executor
<code>spark.default.parallelism</code>	2× cores	Default RDD parallelism

Reading Data

```
# CSV
df = spark.read \
    .option("header", "true") \
    .option("inferSchema", "true") \
    .csv("s3://my-bucket/orders/")

# With explicit schema (always prefer over inferSchema in
# production)
from pyspark.sql.types import (
    StructType, StructField, StringType, IntegerType,
    DoubleType, TimestampType
)

schema = StructType([
    StructField("order_id", StringType(),
        nullable=False),
    StructField("customer_id", IntegerType(),
        nullable=True),
    StructField("amount", DoubleType(),
        nullable=True),
    StructField("created_at", TimestampType(),
        nullable=True),
])

df = spark.read \
    .schema(schema) \
```

```

.option("header", "true") \
.csv("s3://my-bucket/orders/")

# Parquet (no schema needed – it's embedded)
df = spark.read.parquet("s3://my-bucket/orders/")

# JSON
df = spark.read \
    .option("multiline", "false") \
    .json("s3://my-bucket/events/")

# Delta Lake
df = spark.read.format("delta").load("s3://my-
bucket/delta/orders")

# Partitioned data – Spark reads partition columns from
directory names
# s3://my-bucket/orders/order_date=2024-03-15/
df = spark.read.parquet("s3://my-bucket/orders/")
# df automatically has column 'order_date' from partition
path

# JDBC (read from a database)
df = spark.read \
    .format("jdbc") \
    .option("url", "jdbc:postgresql://host:5432/db") \
    .option("dbtable", "orders") \
    .option("user", "username") \
    .option("password", "password") \
    .option("driver", "org.postgresql.Driver") \
    .load()

```

DataFrame Basics

```

# Schema and shape
df.printSchema()      # tree view of column names and
types
df.dtypes              # [("order_id", "string"), ...]
df.columns             # ["order_id", "customer_id", ...]
df.count()             # number of rows (triggers an

```

```

action)

# Preview data
df.show(5)          # print first 5 rows
df.show(5, truncate=False)
df.display()        # Databricks only – interactive table

# Summary stats
df.describe().show() # count, mean, stddev, min, max for numeric cols
df.summary().show()  # + percentiles

# Convert to Pandas (only for small DataFrames)
pdf = df.limit(1000).toPandas()

```

Selecting & Renaming Columns

```

from pyspark.sql import functions as F

# Select specific columns
df.select("order_id", "amount")
df.select(F.col("order_id"), F.col("amount"))

# Alias (rename) a column
df.select(F.col("amount").alias("order_amount"))

# Rename with withColumnRenamed
df.withColumnRenamed("amount", "order_amount")

# Select + transform in one step
df.select(
    "order_id",
    F.col("amount").cast("double").alias("amount"),
    F.upper(F.col("status")).alias("status"),
    F.to_date("created_at").alias("order_date"),
)

# Drop columns
df.drop("internal_id", "raw_payload")

```

```
# Select all except some columns
keep = [c for c in df.columns if c not in {"internal_id",
"raw_payload"}]
df.select(keep)
```

Filtering Rows

```
from pyspark.sql import functions as F

# filter() and where() are identical
df.filter(F.col("amount") > 100)
df.where(F.col("status") == "shipped")

# Multiple conditions
df.filter(
    (F.col("status") == "shipped") &
    (F.col("amount") > 100)
)
df.filter(
    (F.col("dept") == "Engineering") |
    (F.col("dept") == "Product")
)

# NOT
df.filter(~F.col("active"))

# IN
df.filter(F.col("dept").isin("Engineering", "Product",
"Design"))
df.filter(~F.col("status").isin("cancelled", "returned"))

# NULL checks
df.filter(F.col("phone").isNull())
df.filter(F.col("manager_id").isNotNull())

# String patterns
df.filter(F.col("email").endswith("@acme.com"))
df.filter(F.col("name").like("A%"))
df.filter(F.col("name").rlike(r"^[A-Z]")) # regex
```

```
# Between
df.filter(F.col("amount").between(100, 500))

# Filter on partition column for pruning
df.filter(F.col("order_date") == "2024-03-15")
```

Transformations vs Actions

This is the most important concept in Spark.

Transformations — lazy. They describe what to do but don't execute. Spark builds a logical plan.

Actions — trigger execution. Spark compiles the plan, optimizes it, and runs it.

Transformations (lazy): execution):	Actions (trigger execution):
.select()	.show()
.filter()	.count()
.withColumn()	.collect() ← never
on large data	
.groupBy()	.take(n)
.join()	.first()
.orderBy()	.write.parquet(...)
.union()	.toPandas() ← small
data only	

```
# This builds a plan but executes nothing:
result = (
    df.filter(F.col("active") == True)
      .groupBy("dept")
      .agg(F.avg("salary").alias("avg_salary"))
      .orderBy("avg_salary", ascending=False)
)

# This triggers execution:
result.show()

# Every action re-executes from scratch unless the
```

```
DataFrame is cached
# Cache if you'll use the result more than once (see
Caching section)
```

Adding & Transforming Columns

```
from pyspark.sql import functions as F

# withColumn - add or replace a column
df = df.withColumn("annual_salary", F.col("salary") * 12)
df = df.withColumn("name_upper", F.upper(F.col("name")))
df = df.withColumn("hire_year",
F.year(F.col("hire_date")))

# Type casting
df = df.withColumn("amount",
F.col("amount").cast("double"))

# Conditional - like CASE WHEN
df = df.withColumn("tier",
    F.when(F.col("salary") > 100_000, "senior")
    .when(F.col("salary") > 70_000, "mid")
    .otherwise("junior")
)

# COALESCE - first non-null
df = df.withColumn("phone", F.coalesce(F.col("phone"),
F.lit("N/A")))

# String functions
df = df.withColumn("email",
F.lower(F.trim(F.col("email"))))
df = df.withColumn("domain", F.split(F.col("email"),
"@").getItem(1))
df = df.withColumn("initials", F.concat(
    F.substring("first_name", 1, 1),
    F.substring("last_name", 1, 1)
))

# Date functions
```

```

df = df.withColumn("order_date", F.to_date("created_at"))
df = df.withColumn("order_month",
F.date_format("created_at", "yyyy-MM"))
df = df.withColumn("days_since",
F.datediff(F.current_date(), "created_at"))
df = df.withColumn("next_month",
F.add_months("created_at", 1))

# Array / Map operations
df = df.withColumn("tag_count", F.size(F.col("tags")))
df = df.withColumn("first_tag",
F.col("tags").getItem(0))
df = df.withColumn("tags_exploded", F.explode("tags")) #
one row per array element

```

Common built-in functions

```

# Math
F.round(col, 2)    F.abs(col)    F.sqrt(col)    F.log(col)
F.greatest("a","b","c")    F.least("a","b","c")

# String
F.concat(c1, F.lit("-"), c2)    F.length(col)
F.regexp_replace(col, r"\s+", "_")
F.regexp_extract(col, r"(\d+)", 1)
F.lpad(col, 8, "0")    F.rpad(col, 8, " ")

# Hashing / ID generation
F.md5(col)
F.sha2(col, 256)
F.monotonically_increasing_id() # unique but not
consecutive

```

Aggregations & GroupBy

```

from pyspark.sql import functions as F

# GroupBy + agg
df.groupBy("dept").agg(

```

```

    F.count("*").alias("headcount"),
    F.avg("salary").alias("avg_salary"),
    F.max("salary").alias("max_salary"),
    F.min("salary").alias("min_salary"),
    F.sum("salary").alias("total_salary"),
    F.countDistinct("manager_id").alias("n_managers"),
    F.collect_list("name").alias("all_names"),    # list of
values per group
    F.collect_set("dept_id").alias("dept_ids"),    # unique
values per group
)

# Multiple groupBy columns
df.groupBy("dept", "year").agg(
    F.avg("salary").alias("avg_salary")
)

# Aggregate without groupBy (whole DataFrame)
df.agg(F.avg("salary"), F.max("salary")).show()

# Pivot – like GROUP BY + column per value
df.groupBy("dept").pivot("year").agg(F.sum("revenue"))

# Filter after aggregation (equivalent to HAVING)
df.groupBy("dept") \
    .agg(F.count("*").alias("n")) \
    .filter(F.col("n") >= 5)

```

Joins

```

from pyspark.sql import functions as F

employees = spark.read.parquet("s3://data/employees/")
departments = spark.read.parquet("s3://data/departments/")

# Inner join
joined = employees.join(departments, on="dept_id",
    how="inner")

# Left join

```

```

joined = employees.join(departments, on="dept_id",
                        how="left")

# Join on multiple columns
joined = orders.join(items,
                     on=["order_id", "product_id"],
                     how="inner"
)

# Join on different column names
joined = employees.join(
    departments,
    employees["department_id"] == departments["id"],
    how="left"
)

# Disambiguate duplicate column names after join
joined = employees.join(departments,
                       employees["dept_id"] == departments["id"], "left") \
    .select(
        employees["id"].alias("emp_id"),
        employees["name"].alias("emp_name"),
        departments["name"].alias("dept_name"),
    )

# Join types
# "inner"      → only matching rows
# "left"       → all left rows + matched right
# "right"      → all right rows + matched left
# "full"       → all rows from both sides
# "left_semi"  → left rows WHERE a match exists (like EXISTS)
# "left_anti"  → left rows WHERE no match exists (like NOT EXISTS)
# "cross"      → cartesian product (every row × every row)

# Anti-join – find employees with no department
orphans = employees.join(departments,
                        employees["dept_id"] == departments["id"],
                        how="left_anti"
)

```

Sorting & Limiting

```
# orderBy / sort (identical)
df.orderBy("salary")
df.orderBy(F.col("salary").desc())
df.orderBy(F.col("dept").asc(), F.col("salary").desc())

# Nulls placement
df.orderBy(F.col("salary").desc_nulls_last())
df.orderBy(F.col("salary").asc_nulls_first())

# Limit
df.limit(10)

# Sample
df.sample(fraction=0.1, seed=42)
df.sample(withReplacement=False, fraction=0.01)
```

Handling Nulls

```
# Drop rows with any null
df.dropna()

# Drop rows with null in specific columns
df.dropna(subset=["order_id", "amount"])

# Drop only if ALL columns are null
df.dropna(how="all")

# Fill nulls
df.fillna(0) # all
numeric columns
df.fillna({"salary": 0, "phone": "N/A"}) # per column

# Replace nulls with another column value
df.withColumn("phone", F.coalesce("phone", F.lit("N/A")))

# Filter out nulls
df.filter(F.col("amount").isNotNull())
```

```
# Count nulls per column
from pyspark.sql.functions import col, count, when

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

Writing Data

```
# Parquet (default, recommended)
df.write \
    .mode("overwrite") \
    .parquet("s3://my-bucket/output/orders/")

# With partitioning
df.write \
    .mode("overwrite") \
    .partitionBy("order_date") \
    .parquet("s3://my-bucket/output/orders/")

# CSV
df.write \
    .mode("overwrite") \
    .option("header", "true") \
    .csv("s3://my-bucket/output/orders_csv/")

# Delta Lake
df.write \
    .format("delta") \
    .mode("overwrite") \
    .save("s3://my-bucket/delta/orders/")

# Upsert / Merge in Delta
from delta.tables import DeltaTable

target = DeltaTable.forPath(spark, "s3://my-
bucket/delta/orders/")
target.alias("t").merge(
```

```

        df.alias("s"),
        "t.order_id = s.order_id"
    ).whenMatchedUpdateAll() \
    .whenNotMatchedInsertAll() \
    .execute()

# Write modes
# "overwrite" – replace all existing data
# "append" – add to existing data
# "ignore" – no-op if data already exists
# "error" – raise error if data exists (default)

# JDBC
df.write \
    .format("jdbc") \
    .option("url", "jdbc:postgresql://host:5432/db") \
    .option("dbtable", "staging.orders") \
    .option("user", "username") \
    .option("password", "password") \
    .mode("append") \
    .save()

```

Window Functions

Window functions compute a value for each row using a set of surrounding rows, without collapsing them. Same concept as SQL window functions.

```

from pyspark.sql import functions as F
from pyspark.sql.window import Window

# Define the window
w_dept =
Window.partitionBy("dept").orderBy(F.col("salary").desc())
w_date =
Window.partitionBy("user_id").orderBy("event_date")

# Ranking
df = df.withColumn("rank", F.rank().over(w_dept))
df = df.withColumn("dense_rank",
F.dense_rank().over(w_dept))

```

```

df = df.withColumn("row_number",
F.row_number().over(w_dept))

# Lag / Lead — look at previous or next row
df = df.withColumn("prev_salary", F.lag("salary",
1).over(w_date))
df = df.withColumn("next_salary", F.lead("salary",
1).over(w_date))

# Running total
w_running = Window.orderBy("hire_date").rowsBetween(
    Window.unboundedPreceding, Window.currentRow
)
df = df.withColumn("running_total",
F.sum("salary").over(w_running))

# Moving average — 3-row window
w_moving = Window.orderBy("hire_date").rowsBetween(-2, 0)
df = df.withColumn("moving_avg_3",
F.avg("salary").over(w_moving))

# Percent of total
w_total = Window.partitionBy("dept")
df = df.withColumn("pct_of_dept",
    F.col("salary") / F.sum("salary").over(w_total) * 100
)

# Top-N per group
top3 = df.withColumn("rnk", F.row_number().over(w_dept)) \
    .filter(F.col("rnk") <= 3) \
    .drop("rnk")

```

User-Defined Functions (UDFs)

UDFs let you apply arbitrary Python logic to DataFrame columns — but they come with a significant cost.

```

from pyspark.sql.functions import udf
from pyspark.sql.types import StringType, DoubleType

```



```
# Define and register a UDF
def clean_phone(phone: str) -> str:
    if phone is None:
        return None
    return "".join(c for c in phone if c.isdigit())

clean_phone_udf = udf(clean_phone, StringType())

df = df.withColumn("phone_clean",
clean_phone_udf(F.col("phone")))

# Shorthand with decorator
@udf(returnType=DoubleType())
def usd_to_inr(amount):
    if amount is None:
        return None
    return round(amount * 83.5, 2)

df = df.withColumn("amount_inr", usd_to_inr("amount"))
```

Pandas UDF (Vectorized) — much faster

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

# Processes an entire column as a Pandas Series — avoids
row-by-row Python overhead
@pandas_udf(DoubleType())
def usd_to_inr_fast(amounts: pd.Series) -> pd.Series:
    return (amounts * 83.5).round(2)

df = df.withColumn("amount_inr", usd_to_inr_fast("amount"))
```

When NOT to use UDFs

UDFs are a last resort. Every UDF:

- Breaks the Catalyst optimizer — it's a black box -
- Serializes data between JVM and Python (regular UDFs) - Can be 10–100× slower than built-in functions

Always check built-in functions first: `pyspark.sql.functions` has 300+ functions covering most string, date, math, and array operations. If you can express the logic with built-ins or SQL, do that.

Spark SQL

You can write SQL directly against registered temp views.

```
# Register a DataFrame as a temp view
df.createOrReplaceTempView("orders")
departments.createOrReplaceTempView("departments")

# Run SQL
result = spark.sql("""
    SELECT
        d.name          AS dept,
        COUNT(*)        AS headcount,
        AVG(o.amount) AS avg_order
    FROM    orders o
    JOIN    departments d ON o.dept_id = d.id
    WHERE   o.status = 'shipped'
    GROUP   BY d.name
    HAVING  COUNT(*) > 10
    ORDER   BY avg_order DESC
""")

result.show()

# Global temp view – visible across SparkSessions
df.createOrReplaceGlobalTempView("orders_global")
spark.sql("SELECT * FROM global_temp.orders_global")
```

Partitioning & Repartitioning

Every DataFrame is split into **partitions**. The number of partitions determines parallelism.

```
# Check current partition count
df.rdd.getNumPartitions() # e.g. 200
```

```
# repartition – full shuffle, evenly distributes data
df = df.repartition(8) # by count
df = df.repartition(8, "dept_id") # by column –
rows with same dept_id land together

# coalesce – reduces partitions without a full shuffle
(only merges, never splits)
df = df.coalesce(4) # use to reduce before writing to
fewer files

# partitionBy on write – physical directory partitions
df.write.partitionBy("order_date").parquet("s3://output/orders/")
```

Rules of thumb

Scenario	Guidance
After join/groupBy produces too many small partitions	<code>coalesce(n)</code> before write
After reading many small files	<code>repartition(n)</code> to rebalance
Writing to partitioned storage	<code>repartitionByRange("date")</code> then write
Shuffle partitions	Set <code>spark.sql.shuffle.partitions</code> to 2–4× cores for dev, tune for prod
Target partition size	100 MB–1 GB per partition

Caching & Persistence

By default, every action recomputes the DataFrame from scratch. Cache when you use a DataFrame more than once in the same job.

```
# Cache (DataFrame default level: MEMORY_AND_DISK)
df.cache()
df.persist() # same as cache()

# Choose storage level explicitly
```

```

from pyspark import StorageLevel

df.persist(StorageLevel.MEMORY_ONLY)           # partitions
that don't fit are recomputed, not spilled
df.persist(StorageLevel.MEMORY_AND_DISK)       # spills to
disk when memory is full (the default)
df.persist(StorageLevel.DISK_ONLY)            # always on
disk

# Cache is lazy – must trigger an action to actually cache
df.cache()
df.count()  # first action materializes and stores the
cached data

# Unpersist when done – free cluster memory
df.unpersist()

```

When to cache

```

# Good: df used in multiple branches
clean_df = raw_df.filter(...).withColumn(...).cache()
clean_df.count()  # materialize

branch_a = clean_df.groupBy("dept").agg(...)
branch_b = clean_df.join(other_df, ...)
# Without cache, raw_df would be read and processed twice

```

Broadcast Joins

When one table is small (fits in executor memory), broadcast it to all executors so each executor has a full copy. Eliminates the shuffle for the large table.

```

from pyspark.sql.functions import broadcast

# Explicitly broadcast the small table
result = large_df.join(broadcast(small_lookup_df),
on="dept_id", how="left")

# Auto-broadcast threshold (default: 10 MB)

```

```
spark.conf.set("spark.sql.autoBroadcastJoinThreshold", 50 *
1024 * 1024) # 50 MB

# Disable auto-broadcast
spark.conf.set("spark.sql.autoBroadcastJoinThreshold", -1)
```

Rule of thumb: broadcast any table under ~100 MB. The large table gets no shuffle — massive speedup on joins.

Query Optimization & EXPLAIN

```
# Logical plan
df.explain()

# Full plan: parsed → analyzed → optimized → physical
df.explain(extended=True)

# Formatted (Spark 3.x)
df.explain(mode="formatted")

# What to look for in the physical plan:
# BroadcastHashJoin    — good, small table was broadcast
# SortMergeJoin        — shuffle-based join (expensive but
#                        necessary for large tables)
# HashAggregate         — efficient aggregation
# Exchange              — a shuffle is happening (network
#                        I/O)
# FileScan              — reading files; check
#                        "PushedFilters" for pushdown
# Project               — column pruning is happening (good)
```

Common performance issues

Symptom	Cause	Fix
Job runs forever	Skewed partition (one partition has 90% of data)	<code>salting</code> — add a random prefix to the join key, then strip it after

Symptom	Cause	Fix
200 tiny output files	<code>shuffle.partitions=200</code> default	<code>coalesce(n)</code> before write
OOM on executor	Partition too large, or <code>collect()</code> on huge dataset	Increase executor memory or reduce partition size
Slow joins	No broadcast on small table	<code>broadcast()</code> hint or raise auto-broadcast threshold
Reading slowly	No partition filter → full scan	Filter on partition column

Structured Streaming

Structured Streaming is Spark's incremental processing model — it treats a live data stream as an unbounded table.

```
# Read from Kafka
stream_df = spark.readStream \
    .format("kafka") \
    .option("kafka.bootstrap.servers", "kafka:9092") \
    .option("subscribe", "user-events") \
    .option("startingOffsets", "latest") \
    .load()

# Kafka data comes as binary key/value – parse it
from pyspark.sql.types import StructType, StructField,
StringType, LongType

event_schema = StructType([
    StructField("user_id", LongType()),
    StructField("event_type", StringType()),
    StructField("ts", LongType()),
])

events = stream_df.select(
    F.from_json(F.col("value").cast("string"),
event_schema).alias("data")
).select("data.*") \
    .withColumn("event_time", F.col("ts").cast("timestamp"))
```

```

# epoch seconds → timestamp

# Windowed aggregation on event time
from pyspark.sql.functions import window

# Watermark must be on a timestamp column: accept events up
to 10 min late
agg = events \
    .withWatermark("event_time", "10 minutes") \
    .groupBy(
        window(F.col("event_time"), "5 minutes"), # 5-min
        tumbling window
        F.col("event_type")
    ) \
    .count()

# Write to sink
query = agg.writeStream \
    .outputMode("append") \
    .format("delta") \
    .option("checkpointLocation", "s3://checkpoints/user-
events/") \
    .start("s3://output/user-event-counts/")

query.awaitTermination() # block until stopped

```

Output modes

Mode	Writes	Use when
append	Only new rows since last trigger	Aggregations with watermark, append-only sinks
update	Only rows that changed	Aggregations where you want incremental updates
complete	Entire result table every trigger	Small aggregation results

Checkpointing

Checkpoints save the stream's progress (offsets + aggregation state) to durable storage. Required for exactly-once processing and recovery from failures.

```
.option("checkpointLocation", "s3://my-bucket/checkpoints/stream-name/")
```

Common Patterns

Deduplicate with window function

```
from pyspark.sql.window import Window
from pyspark.sql import functions as F

w =
Window.partitionBy("order_id").orderBy(F.col("updated_at").
desc())

deduped = df \
    .withColumn("rn", F.row_number().over(w)) \
    .filter(F.col("rn") == 1) \
    .drop("rn")
```

Watermark-based incremental load

```
def incremental_load(spark, source_path, target_path,
watermark_col="updated_at"):
    try:
        existing = spark.read.parquet(target_path)
        last_ts =
existing.agg(F.max(watermark_col)).collect()[0][0]
    except Exception:
        last_ts = None    # target doesn't exist yet –
full load

    source = spark.read.parquet(source_path)

    if last_ts:
```

```

        new_data = source.filter(F.col(watermark_col) >
last_ts)
    else:
        new_data = source

    new_data.write.mode("append").parquet(target_path)
    print(f"Loaded {new_data.count()} new rows")

```

Schema evolution guard

```

EXPECTED = {"order_id", "customer_id", "amount",
"created_at"}

def validate_schema(df, expected_cols=EXPECTED):
    actual = set(df.columns)
    missing = expected_cols - actual
    extra = actual - expected_cols
    if missing:
        raise ValueError(f"Missing columns: {missing}")
    if extra:
        print(f"Warning: unexpected columns: {extra}")

```

Dynamic partition overwrite

```

# Overwrite only the partitions present in the new data –
not the whole table
spark.conf.set("spark.sql.sources.partitionOverwriteMode",
"dynamic")

df.write \
    .mode("overwrite") \
    .partitionBy("order_date") \
    .parquet("s3://output/orders/")

```

Flatten a nested / struct column

```

# Struct: expand fields with col("struct.*")
flat = df.select("id", "event.*") # expands all struct
fields

```

```
# Array: one row per element
exploded = df.withColumn("tag",
F.explode("tags")).drop("tags")

# Array of structs: explode then expand
exploded = df.withColumn("item", F.explode("line_items")) \
    .select("order_id", "item.*")
```

Common Pitfalls

Pitfall	Symptom	Fix
<code>collect()</code> / <code>toPandas()</code> on a big DataFrame	Driver OOM, job dies at the very end	Aggregate or <code>limit()</code> first; write large results to storage
<code>inferSchema=true</code> in production	Slow (reads the data twice); types change when the data changes	Pass an explicit <code>StructType</code> schema
Python UDFs for things built-ins can do	10–100× slower; the optimizer can't see inside	<code>pyspark.sql.functions</code> first, then Pandas UDFs, and Python UDFs only as a last resort
<code>count()</code> / <code>show()</code> after every step to "debug"	Every action re-runs the whole lineage from source	Inspect with <code>explain()</code> ; cache only what's reused; check small samples
Caching everything	Executors run out of memory, other jobs slow down	Cache only DataFrames used more than once, then <code>unpersist()</code>
Joining on keys with a few huge values	One task runs for hours while the rest finish in seconds	Enable AQE skew join handling; salt hot keys; broadcast the small side
Leaving <code>spark.sql.shuffle.partitions</code> at 200 for every job	Thousands of tiny tasks on small data, or giant tasks on huge data	Enable AQE (coalesces partitions automatically) or tune per job
<code>mode("overwrite")</code> on a partitioned path without dynamic overwrite	One day's rerun wipes the entire table	<code>partitionOverwriteMode=dynamic</code> , or Delta/Iceberg <code>replaceWhere</code> / <code>MERGE</code>

Pitfall	Symptom	Fix
<code>repartition(1) / coalesce(1)</code> to "get one file" on big data	All data squeezed through one task — slow or OOM	Accept multiple files; compact with the table format if needed
Chaining hundreds of <code>withColumn</code> calls in a loop	Huge query plans, slow planning, even stack overflows	One <code>select()</code> with a list of expressions, or <code>withColumns({...})</code>
Filtering on a derived expression (<code>F.to_date("ts") == ...</code>)	No partition pruning or predicate pushdown	Filter on the raw partition column, or add a proper partition column
Changing a streaming query's logic but reusing its checkpoint	Query fails to start or gives wrong state	New checkpoint location for incompatible changes; plan state migrations
Streaming aggregation without a watermark	State grows forever until the job OOMs	<code>withWatermark()</code> on an event-time column

Cheat Sheet

Task	Code
Session	<code>SparkSession.builder.appName("x").getOrCreate()</code>
Read with schema	<code>spark.read.schema(schema).option("header", True).csv(path)</code>
Read Parquet / Delta / table	<code>spark.read.parquet(p) · spark.read.format("delta").load(p) · spark.table("db.t")</code>
Select / rename / cast	<code>df.select("a", F.col("b").alias("c"), F.col("d").cast("date"))</code>
Add several columns	<code>df.withColumns({"x": expr1, "y": expr2})</code>
Filter	<code>df.filter((F.col("a") > 1) & F.col("b").isNotNull())</code>
Conditional	<code>F.when(cond, v).when(cond2, v2).otherwise(v3)</code>
Group + aggregate	<code>df.groupBy("k").agg(F.sum("amt").alias("total"), F.countDistinct("id"))</code>

Task	Code
Join	<code>a.join(b, on="id", how="left") · a.join(F.broadcast(b), "id")</code>
Anti / semi join	<code>how="left_anti" (in a, not in b) · how="left_semi" (in a, and in b)</code>
Window	<code>w = Window.partitionBy("k").orderBy(F.desc("ts")) → F.row_number().over(w)</code>
Dedupe exact rows / by key	<code>df.dropDuplicates() · df.dropDuplicates(["id"]) (arbitrary row — use a window for "latest")</code>
Nulls	<code>df.fillna({"a": 0}) · F.coalesce("a", "b")</code>
JSON string → columns	<code>F.from_json("value", schema) → .select("data.*")</code>
Explode array	<code>F.explode("items") (explode_outer keeps empty/NULL arrays)</code>
Dates	<code>F.to_date, F.date_trunc("month", c), F.datediff, F.date_add</code>
SQL	<code>df.createOrReplaceTempView("t"); spark.sql("SELECT ...")</code>
Partitions	<code>df.rdd.getNumPartitions() · repartition(n, "k") · coalesce(n)</code>
Plan	<code>df.explain(mode="formatted")</code>
Write partitioned	<code>df.write.mode("overwrite").partitionBy("dt").parquet(p)</code>
Write a table	<code>df.writeTo("db.t").append() · .overwritePartitions()</code>

Performance checklist: AQE on (default in 3.2+) → filter and select early → broadcast small tables → avoid Python UDFs → check the Spark UI for skew (one long task) and spill → right-size files on write (128 MB–1 GB)

Narrow vs wide: `select`, `filter`, `withColumn`, `union` are narrow (no shuffle) · `groupBy`, `join`, `distinct`, `orderBy`, `repartition` are wide (shuffle = a stage boundary)

Interview Questions

Q: What is the difference between a transformation and an action in Spark? A:

Transformations (`filter`, `select`, `join`, `groupBy`) are lazy — they build an execution plan but don't process data. Actions (`show`, `count`, `collect`, `write`) trigger execution. This distinction

lets Spark's Catalyst optimizer combine and reorder transformations for efficiency before running anything. Calling `.count()` after each step to debug is an anti-pattern — it forces execution at every step.

Q: What is a Spark partition and how does it relate to parallelism? A: A partition is a chunk of the data that one executor task processes. With 100 partitions and 10 executor cores, Spark processes 10 partitions at a time. Too few partitions = some cores idle; too many = too much scheduling overhead. Rule of thumb: 2-4 partitions per CPU core, each 128-256MB.

Q: What is the difference between repartition and coalesce? A: Both change the number of partitions. `repartition(n)` does a full shuffle (expensive) and can both increase and decrease partitions — use when you need an even distribution or more partitions. `coalesce(n)` merges partitions without a shuffle (cheap) but can only decrease — use when writing output to reduce the number of output files.

Q: What is a broadcast join and when should you use it? A: A broadcast join sends the smaller DataFrame to every executor so the join can happen locally without a shuffle. Use when one table is small enough to fit in executor memory (< 10MB by default, configurable). It's the single most impactful optimization for joins with a small lookup table (e.g., joining orders to a small products table). Enable with `spark.sql.autoBroadcastJoinThreshold` or `F.broadcast(small_df)`.

Q: What causes data skew and how do you fix it? A: Skew is when one partition has far more data than others — one executor does all the work while others sit idle. Common cause: joining or grouping on a column with very uneven distribution (e.g., a few customers with millions of orders). Fixes: (1) salting — add a random suffix to the key, join, then aggregate; (2) broadcast join the large-key entity; (3) filter out the skewed keys and process them separately; (4) use AQE (`spark.sql.adaptive.enabled=true`) which auto-detects and handles skew.

Q: What is the difference between Spark Structured Streaming and batch processing? A: Batch processing reads a bounded dataset, processes it, and writes results — has a clear start and end. Structured Streaming reads from an unbounded source (Kafka, S3 files) continuously, processing micro-batches or trigger-based intervals, with a checkpoint to track progress. The API is the same (DataFrame operations) but streaming adds constraints: only certain aggregations work, joins have limitations, and you must manage state and watermarks.

Q: What is Adaptive Query Execution (AQE)? A: AQE re-optimizes the query plan *during* execution using runtime statistics from completed shuffle stages. It does three main things: coalesces many small shuffle partitions into fewer, well-sized ones; switches a sort-merge join to a broadcast join when one side turns out to be small; and splits skewed partitions in joins so one hot key doesn't stall the job. It's on by default since Spark 3.2 and removes most of the need to hand-tune `spark.sql.shuffle.partitions`.

Q: What are narrow and wide transformations, and what is a stage? A: In a narrow transformation, each output partition depends on one input partition (`filter` , `select`), so no data moves between executors. In a wide transformation, output partitions depend on many input partitions (`groupBy` , `join`), which requires a shuffle over the network. Spark splits a job into stages at shuffle boundaries; tasks within a stage run in parallel, one per partition. Fewer shuffles usually means a faster job.

Q: Walk through what happens when you call `df.write.parquet(...)`. A: The write is an action, so the driver takes the logical plan built by earlier transformations, lets Catalyst optimize it (predicate pushdown, column pruning, join selection), and generates a physical plan. The DAG scheduler splits it into stages at shuffle boundaries, and the task scheduler sends one task per partition to executors. Each executor reads its input splits, processes them, and writes its own part files; the output committer then finalizes the files (which is why object stores benefit from table formats with atomic commits).

Q: How would you debug a Spark job that is slow? A: Open the Spark UI. In the Stages tab, look for a stage where one task takes far longer than the median (skew), large "spill (disk)" values (partitions too big for memory), or huge shuffle read/write sizes (an unnecessary shuffle or a missing broadcast). In the SQL tab, check the plan for full scans with no pushed filters, sort-merge joins where a broadcast would do, and Python UDF nodes. Then fix the cause: filter earlier, broadcast, salt skewed keys, replace UDFs, or right-size partitions.

Q: How does Structured Streaming achieve exactly-once results? A: Replayable sources (Kafka offsets, file lists) plus checkpointing plus idempotent or transactional sinks. Before processing each micro-batch, Spark records the offsets it will read in the checkpoint's write-ahead log; state is also checkpointed. After a failure, it replays exactly those offsets. Sinks like Delta commit each batch atomically, tagged with the batch ID, so a replayed batch isn't written twice. With a non-idempotent sink (e.g. a plain JDBC insert), you only get at-least-once.

Further Reading

- [PySpark API reference](https://spark.apache.org/docs/latest/api/python/reference/index.html) (https://spark.apache.org/docs/latest/api/python/reference/index.html)
- [Spark SQL performance tuning](https://spark.apache.org/docs/latest/sql-performance-tuning.html) (https://spark.apache.org/docs/latest/sql-performance-tuning.html) — AQE, broadcast, partitioning
- [Structured Streaming programming guide](https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html) (https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html)
- [Spark UI guide](https://spark.apache.org/docs/latest/web-ui.html) (https://spark.apache.org/docs/latest/web-ui.html) — reading stages, tasks, and SQL plans
- *Learning Spark, 2nd Edition* — Jules Damji et al. (O'Reilly; free PDF from Databricks)

- *Spark: The Definitive Guide* — Bill Chambers & Matei Zaharia (O'Reilly)
-

Previous: [DuckDB & Polars](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/duckdb-polars/) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/duckdb-polars/) • **Next:** [Databricks](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/databricks-reference/) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/databricks-reference/) • **Back to:** [Index](https://sarangambekar1997.github.io/data-engineering-handbook/) (https://sarangambekar1997.github.io/data-engineering-handbook/)

 September 29, 2026