Spark Cheat Sheet

Start Spark & Load Data

Load CSV

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName("MyApp").getOrCreate()
df = spark.read.csv("filename.csv", header=True, inferSchema=True)
```

Load JSON

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName("MyApp").getOrCreate()
df = spark.read.json("filename.json")
```

Exploring the Data

Task	Command
Show rows	`df.show()`
Show first N rows	`df.show(10)`
Show schema	`df.printSchema()`
Show column names	`df.columns`
Get number of rows	`df.count()`
Get distinct values	`df.select("column").distinct().show()`
Describe (summary stats)	`df.describe().show()`

Column operations

Task	Command
Select column(s)	`df.select("col1", "col2")`
Filter rows	`df.filter(df.col1 > 50)`
Add new column	`df.withColumn("new_col", col("col1") + 5)`
Rename column	`df.withColumnRenamed("old", "new")`
Cast column type	`df.withColumn("col", df["col"].cast("float"))`
Drop a column	`df.drop("col_to_drop")`

Row operations

Task	Command
Show first N rows	`df.show(N)`
Show top rows with full content	`df.show(truncate=False)`
Filter rows with a condition	`df.filter(df["col"] > 100)`
Filter with multiple conditions	`df.filter((df["age"] > 18) & (df["country"] == "US"))`
Filter using `where()`	`df.where("age > 30")`
Filter NULL rows	`df.filter(df["col"].isNull())`
Filter non-NULL rows	`df.filter(df["col"].isNotNull())`
Select rows with values in list	`df.filter(df["col"].isin("A", "B", "C"))`
Drop duplicate rows	`df.dropDuplicates()`
Drop duplicate rows by column(s)	`df.dropDuplicates(["col1", "col2"])`
Limit number of rows	`df.limit(100)`
Random sample of rows	`df.sample(withReplacement=False, fraction=0.1)`
Add row number	`from pyspark.sql.window import Window``from pyspark.sql.functions import row_number``df.withColumn("row_num", row_number().over(Window.orderBy("col")))`
Add conditional column	`df.withColumn("flag", when(df["score"] > 70, "Pass").otherwise("Fail"))`
Replace values conditionally	`df.withColumn("status", when(df["status"] == "old", "new").otherwise(df["status"]))`

Data clearning

Task	Command
Drop rows with nulls	`df.dropna()`
Drop nulls in specific columns	`df.dropna(subset=["col1", "col2"])`
Fill nulls	`df.fillna({'col1': 0, 'col2': 'Unknown'})`
Replace values	`df.replace("old", "new", "column")`
Drop duplicates	`df.dropDuplicates()`
Drop duplicates by column	`df.dropDuplicates(["col1"])`
Trim strings	`from pyspark.sql.functions import trim; df.withColumn("col", trim(col("col")))`
Convert to lower case	`df.withColumn("col", lower(col("col")))`
Convert to upper case	`df.withColumn("col", upper(col("col")))`
Remove special characters	`df.withColumn("col", regexp_replace(col("col"), "[^a-zA-Z0-9 ]", ""))`
Cast column type	`df.withColumn("col", col("col").cast("float"))`
Rename column	`df.withColumnRenamed("old_name", "new_name")`
Split string into array	`df.withColumn("col_array", split(col("col"), ","))`
Explode array into rows	`df.withColumn("value", explode(col("col_array")))`
Add column with condition	`df.withColumn("flag", when(col("score") > 90, "high").otherwise("low"))`
Parse date	`df.withColumn("parsed_date", to_date("Date", "M/d/yyyy"))`
Extract year/month/day from date	`df.withColumn("Year", year(col("parsed_date")))` etc.
Check for NULLs in all columns	`df.select([count(when(col(c).isNull(), c)).alias(c) for c in df.columns]).show()`
Select rows with NULLs	`df.filter(col("col").isNull())`
Select rows with non-NULLs	`df.filter(col("col").isNotNull())`
Conditional replacement	`df.withColumn("col", when(col("col") == "old", "new").otherwise(col("col")))`
Round numeric values	`df.withColumn("col_rounded", round(col("col"), 2))`

Aggregations

Task	Command
Group and count	`df.groupBy("col").count()`
Group and average	`df.groupBy("col").avg("value_col")`
Multiple aggregations	`df.groupBy("col").agg({'value_col': 'avg', 'id': 'count'})`
Sort ascending	`df.orderBy("col")`
Sort descending	`df.orderBy(df.col.desc())`

Using aggregated functions

```
from pyspark.sql.functions import avg, max, min, sum, stddev
df.select(avg("Valuation_numeric"), stddev("Valuation_numeric")).show()
```

Date functions

```
# For all options
from pyspark.sql.functions import (
    to_date, to_timestamp, year, month, dayofmonth, dayofweek, weekofyear,
    date_format, date_add, date_sub, datediff, concat_ws, current_date
)

from pyspark.sql.functions import to_date, year, month, dayofmonth

df = df.withColumn("DateParsed", to_date("Date", "M/d/yyyy"))
df = df.withColumn("Year", year("DateParsed"))
df = df.withColumn("Month", month("DateParsed"))
df = df.withColumn("Day", dayofmonth("DateParsed"))
```

Date/time operations

Task	Command
Convert string to date (MM/dd/yyyy)	`df.withColumn("parsed_date", to_date("Date", "MM/dd/yyyy"))`
Convert string to date (M/d/yyyy)	`df.withColumn("parsed_date", to_date("Date", "M/d/yyyy"))`
Convert to timestamp	`df.withColumn("ts", to_timestamp("Date", "MM/dd/yyyy HH:mm:ss"))`
Extract year	`df.withColumn("Year", year("parsed_date"))`
Extract month	`df.withColumn("Month", month("parsed_date"))`
Extract day of month	`df.withColumn("Day", dayofmonth("parsed_date"))`
Extract day of week (1=Sun)	`df.withColumn("DayOfWeek", dayofweek("parsed_date"))`
Extract week of year	`df.withColumn("WeekOfYear", weekofyear("parsed_date"))`
Format date to string (yyyy-MM)	`df.withColumn("MonthFormatted", date_format("parsed_date", "yyyy-MM"))`
Add days	`df.withColumn("Plus7Days", date_add("parsed_date", 7))`
Subtract days	`df.withColumn("Minus7Days", date_sub("parsed_date", 7))`
Calculate difference between dates	`df.withColumn("diff_days", datediff(col("end_date"), col("start_date")))`
Create date from year, month, day	`df.withColumn("full_date", to_date(concat_ws("-", "year", "month", "day")))`
Get current date/time	`from pyspark.sql.functions import current_date, current_timestamp``current_date()`

Save as `csv`

Task	Command
Save as CSV	`df.write.csv("folder", header=True)`
Save as single CSV file	`df.coalesce(1).write.csv("folder", header=True)`
Save as Parquet	`df.write.parquet("path")`

Transformations vs Actions

Category	Transformations	Actions
Definition	Define *what to do* with the data	Actually *do it* and return results
Lazy?	✅ Yes (do nothing until an action is called)	❌ No (they trigger execution)
⚙Returns	New DataFrame	Actual values or saved output
Triggers execution?	No	Yes
Examples	`select()`, `filter()`, `withColumn()`, `groupBy()`	`show()`, `collect()`, `count()`, `take()`, `write()`

Transformations

Function	Description	Example Usage
`pivot()`	Rotates rows into columns (like a pivot table)	`df.groupBy("Year").pivot("Country").sum("Valuation")`
`explode()`	Converts an array or string split into multiple rows	`explode(split(col("Investors"), ","))`
`split()`	Splits a string into an array based on a delimiter	`split(col("Investors"), ",")`
`array()`	Combines multiple columns into an array	`array("col1", "col2")`
`concat_ws()`	Concatenates multiple columns into one string (with a separator)	`concat_ws("-", col("Year"), col("Month"), col("Day"))`
`withColumn()`	Creates or replaces a column	`df.withColumn("Val", col("Valuation").cast("float"))`
`selectExpr()`	SQL-style expression for selecting and transforming columns	`df.selectExpr("Company", "Valuation * 1.2 as AdjustedVal")`
`melt()` *(custom)*	Converts wide data back to long format (inverse of pivot — requires workaround)	[Not native, use stack/union workaround]
`groupBy().agg()`	Aggregate after grouping, with multiple functions	`df.groupBy("Industry").agg(avg("Val"), count("*"))`
`stack()`	Emulates melt/unpivot by converting multiple columns into rows	`SELECT Company, stack(2, '2020', val2020, '2021', val2021) AS (Year, Value) FROM table`
`flatten()`	Flattens nested arrays (requires Spark 2.4+)	`flatten(col("nested_array"))`
`explode_outer()`	Same as `explode()`, but returns NULL if input is NULL	`explode_outer(col("arr"))`
`arrays_zip()`	Combines multiple arrays into an array of structs (zip-like behavior)	`arrays_zip("arr1", "arr2")`
`posexplode()`	Explodes array with element positions (like index + value)	`df.select(posexplode(split(col("tags"), ",")))`
`map()` (in SQL)	Create a map column from key-value pairs	`select map("k1", col1, "k2", col2) as kv_map`

Actions

Action	Description**	Returns
`show()`	Displays rows from the DataFrame	None (prints to console)
`collect()`	Collects all rows to the driver node	Python list of Rows
`take(n)`	Returns the first `n` rows	List of Row objects
`first()`	Returns the first row	Single Row
`head()`	Same as `first()` or `take(1)`	Row or list
`count()`	Counts number of rows	Integer
`describe().show()`	Summary statistics of numerical columns	Console output
`agg().show()`	Performs aggregate computations (avg, sum, etc.)	None (prints result)
`foreach()`	Runs a function on each row (no return)	None
`foreachPartition()`	Runs a function on each partition	None
`toPandas()`	Converts DataFrame to Pandas DataFrame	`pandas.DataFrame`
`write.csv(...)`	Saves DataFrame to CSV	Saves files (returns nothing)
`write.json(...)`	Saves DataFrame to JSON	Saves files
`write.parquet(...)`	Saves DataFrame to Parquet	Saves files
`write.saveAsTable(...)`	Saves DataFrame as a Hive table (if using Hive)	None
`isEmpty()` *(3.3+)*	Checks if DataFrame has no rows	Boolean
`inputFiles()` *(rdd)*	Returns paths used to construct a DataFrame from files	List of strings

Pivot

> **Pivoting** is like rotating a table: it turns **row values into column headers**. It's useful for summarizing and reshaping data (like pivot tables in Excel).

```
df.groupBy("Year").pivot("Country").sum("Valuation")
```

Explode

> **Explode** transforms an **array or map column into multiple rows**. It's like flattening a list inside a column.

For example, going from:

Company	Investors
Stripe	["Sequoia", "Tiger Global"]
Klarna	["BlackRock", "Atomico"]

To:

Company	Investor
Stripe	Sequoia
Stripe	Tiger Global
Klarna	BlackRock
Klarna	Atomico

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

df.withColumn("Investor", explode(split(col("Investors"), ",")))
```