Global CoE Community

Sanjay Chandra

PySpark Cheatsheet 2.0

Explore More

Contents

Part 1: Connect & Read

Import

```python
# 1. Main Spark Session and Types
from pyspark.sql import SparkSession
from pyspark.sql.types import StructType, StructField, StringType, IntegerType, DoubleType, DateType, TimestampType

# 2. Functions (Standard Alias 'F' is industry best practice)
from pyspark.sql import functions as F

# 3. Window Functions (for ranking, running totals)
from pyspark.sql.window import Window

# 4. Initialize Spark Session
spark = SparkSession.builder \
    .appName("MasterCheatSheet") \
    .config("spark.sql.adaptive.enabled", "true") \
    .getOrCreate()

# 5. Check Spark Version
print(spark.version)

# 6. Set Log Level (Reduce noise in console)
spark.sparkContext.setLogLevel("WARN")
```

I/O

```python
# 1. Read Parquet (Best format for Spark)
df = spark.read.parquet("path/to/data.parquet")

# 2. Write Parquet (Overwrite existing data)
df.write.mode("overwrite").parquet("path/to/output")

# 3. Read CSV (Infer Schema = Slow; use mostly for ad-hoc)
df = spark.read.option("header", "true").option("inferSchema", "true").csv("data.csv")

# 4. Read JSON (Enable multiline for standard nested JSON)
df = spark.read.option("multiline", "true").json("data.json")

# 5. Read Delta Table (Versioning & Acid Transactions)
df = spark.read.format("delta").load("path/to/delta_table")

# 6. Write to Delta Table (Append new data)
df.write.format("delta").mode("append").save("path/to/delta_table")

# 7. Write Partitioned Data (Crucial for query speed later)
df.write.partitionBy("year", "month").parquet("path/to/output")

# 8. Save as Managed Table (Hive/Glue catalog)
df.write.saveAsTable("db_name.table_name")
```

Schema

```python
# 1. Define a Schema explicitly
schema = StructType([
    StructField("id", IntegerType(), False),       # Not Nullable
    StructField("name", StringType(), True),       # Nullable
    StructField("salary", DoubleType(), True),
    StructField("hire_date", DateType(), True)
])

# 2. Apply Schema while reading (Fastest Read Method)
df = spark.read.schema(schema).csv("employees.csv")

# 3. Print Schema to console (Tree format)
df.printSchema()

# 4. Get List of Column Names
columns = df.columns

# 5. Inspect Column Data Types
dtypes = df.dtypes  # Returns list of tuples: [('id', 'int'), ('name', 'string')]

# 6. Alter Schema: Change Nullability (Advanced)
# Creates a new schema based on existing one but forces fields to be Nullable
new_schema = StructType([
    StructField(f.name, f.dataType, True) for f in df.schema.fields
])
```

Part 2: Clean & Transform

Nulls & Duplicates

```python
# 1. Drop Duplicate Rows (Checks all columns)
df = df.dropDuplicates()

# 2. Drop Duplicates based on specific columns (e.g., only check ID)
df = df.dropDuplicates(["id"])

# 3. Drop Rows containing ANY null values
df = df.na.drop(how="any")

# 4. Drop Rows where ALL columns are null
df = df.na.drop(how="all")

# 5. Fill Nulls with a default value (e.g., 0 for integers)
df = df.na.fill(0)

# 6. Fill Nulls for specific columns using a dictionary
df = df.na.fill({"age": 0, "city": "Unknown", "salary": 1000.0})

# 7. Replace specific values (e.g., empty strings with None)
df = df.replace("", None)

# 8. Trim Whitespace from strings (Left and Right)
df = df.withColumn("name", F.trim(F.col("name")))

# 9. Standardize Case (Lower/Upper)
df = df.withColumn("email", F.lower(F.col("email")))
```

Filtering

```python
# 1. Simple Equality Filter
df_filt = df.filter(F.col("status") = = "Active")

# 2. Multiple Conditions (AND) - Note: Parentheses are mandatory in PySpark
df_filt = df.filter((F.col("age") > 25) & (F.col("dept") = = "Sales"))

# 3. Multiple Conditions (OR)
df_filt = df.filter((F.col("category") = = "A") | (F.col("category") = = "B"))

# 4. Filter using a List (isin)
df_filt = df.filter(F.col("country").isin("USA", "UK", "Canada"))

# 5. Filter for Null / Not Null
df_filt = df.filter(F.col("manager_id").isNotNull())
df_nulls = df.filter(F.col("manager_id").isNull())

# 6. Filter Strings (Contains / StartsWith)
df_filt = df.filter(F.col("email").contains("@gmail.com"))
df_filt = df.filter(F.col("name").startswith("J"))

# 7. Filter using Negation (~)
df_filt = df.filter(~(F.col("status") = = "Closed"))

# 8. Filter Between Range
df_filt = df.filter(F.col("salary").between(50000, 100000))
```

Ordering

```python
# 1. Sort Ascending (Default)
df_sorted = df.orderBy("salary")

# 2. Sort Descending (High to Low)
df_sorted = df.orderBy(F.col("salary").desc())

# 3. Sort by Multiple Columns
# Sorts by Department (Asc), then by Salary (Desc)
df_sorted = df.orderBy(F.col("department").asc(), F.col("salary").desc())

# 4. Sort using "sort()" (Alias for orderBy - does the exact same thing)
df_sorted = df.sort(F.col("last_name"))

# 5. Sort with Nulls First
# useful if you want to highlight missing data at the top
df_sorted = df.orderBy(F.col("age").desc_nulls_first())

# 6. Sort with Nulls Last (Default for asc, but explicit here)
df_sorted = df.orderBy(F.col("age").asc_nulls_last())

# 7. Sort within Partitions (sortWithinPartitions)
# More efficient than orderBy if you only need data sorted per partition
# (e.g., before writing files), avoids a massive global shuffle.
df_sorted = df.sortWithinPartitions("date")

# 8. Limit Top N Rows (Ordering is required first)
top_5_earners = df.orderBy(F.col("salary").desc()).limit(5)
```

Column Operations

```python
# 1. Add a New Column (Constant Value)
df = df.withColumn("report_date", F.lit("2023-01-01"))

# 2. Add a Calculated Column
df = df.withColumn("total_comp", F.col("salary") + F.col("bonus"))

# 3. Rename a Column
df = df.withColumnRenamed("fname", "first_name")

# 4. Conditional Logic (Case When ... Otherwise)
df = df.withColumn("grade",
    F.when(F.col("score") ≥ 90, "A")
     .when(F.col("score") ≥ 80, "B")
     .otherwise("C")
)

# 5. Cast Data Type (Change String to Integer, etc.)
df = df.withColumn("age", F.col("age").cast(IntegerType()))

# 6. Concatenate Columns (Combine First + Last Name)
df = df.withColumn("full_name", F.concat_ws(" ", F.col("first"), F.col("last")))

# 7. Drop a Column
df = df.drop("temp_column")

# 8. Select specific columns to keep (Reorder)
df = df.select(F.col("id"), F.col("full_name"), F.col("grade"))
```

String Operations

```python
# 1. Change Case (Lower / Upper / Title)
df = df.withColumn("lower_name", F.lower(F.col("name")))
df = df.withColumn("init_cap", F.initcap(F.col("description")))

# 2. Trim Whitespace (Both ends, Left, Right)
df = df.withColumn("clean_id", F.trim(F.col("id_str")))

# 3. Concatenate Strings (with Separator)
# Combines columns with a separator (ignores nulls automatically)
df = df.withColumn("full_address", F.concat_ws(", ", F.col("city"), F.col("state")))

# 4. Substring (Extract parts of a string)
# Note: PySpark substring index starts at 1, and length is the second arg
df = df.withColumn("zip_prefix", F.substring(F.col("zipcode"), 1, 3))

# 5. Split String (Convert String to Array)
df = df.withColumn("tags_array", F.split(F.col("tags"), ","))

# 6. Regex Replace (Pattern Matching)
# Remove all non-digit characters (keep only numbers)
df = df.withColumn("phone_clean", F.regexp_replace(F.col("phone"), "[^0-9]", ""))

# 7. Check String Length
df = df.withColumn("desc_len", F.length(F.col("description")))

# 8. Check if String Contains Pattern
df = df.withColumn("is_gmail", F.col("email").contains("gmail"))

# 9. Padding (Add characters to reach fixed width)
df = df.withColumn("id_padded", F.lpad(F.col("id"), 10, "0"))
```

Part 3: Joins & Aggregate

Joins

```python
# 1. Inner Join (Default - Keep only matches)
df_join = df1.join(df2, "id", "inner")

# 2. Left Join (Keep all Left, match Right)
df_join = df1.join(df2, "id", "left")

# 3. Full Outer Join (Keep everything from both)
df_join = df1.join(df2, "id", "outer")

# 4. Left Semi Join (Filter Left: Keep rows that EXIST in Right)
# Efficient alternative to IN subquery. Only keeps columns from Left.
df_join = df1.join(df2, "id", "left_semi")

# 5. Left Anti Join (Filter Left: Keep rows that DO NOT EXIST in Right)
# Efficient alternative to NOT IN. Only keeps columns from Left.
df_join = df1.join(df2, "id", "left_anti")

# 6. Join on Multiple Columns
df_join = df1.join(df2, ["id", "dept_code"], "inner")

# 7. Join with Different Column Names
# Note: This keeps both keys (id, emp_id). You often drop one after.
df_join = df1.join(df2, F.col("id") == F.col("emp_id"), "left")

# 8. Cross Join (Cartesian Product - Warning: Very Expensive!)
# Matches every row in Left with every row in Right
df_join = df1.crossJoin(df2)
```

Aggregations

```python
# 1. Simple Count (Rows per Group)
df_agg = df.groupBy("department").count()

# 2. Multiple Aggregations (Sum, Avg, Max, Min)
# Always use .agg() to perform multiple calcs at once
df_agg = df.groupBy("department").agg(
    F.sum("salary").alias("total_salary"),
    F.avg("salary").alias("avg_salary"),
    F.max("bonus").alias("max_bonus"),
    F.count("id").alias("emp_count")
)

# 3. Count Distinct (Unique values per group)
df_agg = df.groupBy("category").agg(F.countDistinct("product_id"))

# 4. Pivot (Convert Rows to Columns)
# Example: Turn 'month' values (Jan, Feb) into columns
df_pivot = df.groupBy("product").pivot("month").sum("sales")

# 5. Global Aggregation (No GroupBy - summarize entire DF)
max_val = df.agg(F.max("salary")).collect()[0][0]

# 6. Collect List (Aggregate values into an Array)
# Warning: Be careful with large lists, can cause OOM errors
df_agg = df.groupBy("dept").agg(F.collect_list("name").alias("employees"))
```

Date & Time

```python
# 1. Get Current Date & Timestamp
df = df.withColumn("now", F.current_timestamp())
df = df.withColumn("today", F.current_date())

# 2. Convert String to Date (Cast or Parse)
df = df.withColumn("date_col", F.to_date(F.col("date_str"), "yyyy-MM-dd"))

# 3. Convert Timestamp to String (Format)
df = df.withColumn("date_str", F.date_format(F.col("timestamp"), "MM/dd/yyyy"))

# 4. Date Arithmetic (Add / Subtract Days)
df = df.withColumn("next_week", F.date_add(F.col("date_col"), 7))
df = df.withColumn("last_week", F.date_sub(F.col("date_col"), 7))

# 5. Date Difference (Returns number of days)
df = df.withColumn("days_diff", F.datediff(F.col("end_date"), F.col("start_date")))

# 6. Extract Date Components
df = df.withColumn("year", F.year(F.col("date_col")))
df = df.withColumn("month", F.month(F.col("date_col")))
df = df.withColumn("day", F.dayofmonth(F.col("date_col")))

# 7. Truncate Date (Start of Month/Year)
# Useful for monthly reporting aggregations
df = df.withColumn("start_of_month", F.trunc(F.col("date_col"), "Month"))

# 8. Unix Timestamp (Seconds since Epoch)
df = df.withColumn("unix_time", F.unix_timestamp(F.col("timestamp_col")))
```

Window Functions

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

# 1. Define Window Specification (Partition & Order)
w_spec = Window.partitionBy("department").orderBy(F.col("salary").desc())

# 2. Row Number (Sequential: 1, 2, 3 ... )
df = df.withColumn("row_num", F.row_number().over(w_spec))

# 3. Rank (Gaps for ties: 1, 2, 2, 4)
df = df.withColumn("rank_val", F.rank().over(w_spec))

# 4. Dense Rank (No gaps for ties: 1, 2, 2, 3)
df = df.withColumn("dense_rank", F.dense_rank().over(w_spec))

# 5. Lead (Next Row's Value) - Shift Up
df = df.withColumn("next_salary", F.lead("salary", 1).over(w_spec))

# 6. Lag (Previous Row's Value) - Shift Down
df = df.withColumn("prev_salary", F.lag("salary", 1).over(w_spec))

# 7. Running Total (Cumulative Sum)
df = df.withColumn("running_sum", F.sum("salary").over(w_spec))

# 8. Group Average (Append average to every row - No OrderBy needed)
w_agg = Window.partitionBy("department")
df = df.withColumn("dept_avg", F.avg("salary").over(w_agg))
```

Part 4: Master & Extend

Complex Data Types

```python
# 1. Explode Array (Create new row for each element)
# [A, B] → Row 1: A, Row 2: B
df = df.withColumn("tag", F.explode(F.col("tags_array")))

# 2. Array Contains (Filter if element exists)
df_filt = df.filter(F.array_contains(F.col("tags_array"), "urgent"))

# 3. Get Array Size
df = df.withColumn("num_tags", F.size(F.col("tags_array")))

# 4. Get Specific Element (1-based index)
df = df.withColumn("first_tag", F.element_at(F.col("tags_array"), 1))

# 5. Create Map (Key-Value pairs from columns)
df = df.withColumn("properties", F.create_map(F.lit("color"), F.col("color_col")))

# 6. Get Map Keys and Values
df = df.withColumn("all_keys", F.map_keys(F.col("properties")))
df = df.withColumn("all_values", F.map_values(F.col("properties")))

# 7. Access Nested Struct Field (Dot Notation)
# Assuming 'address' is a struct column with field 'city'
df = df.withColumn("city", F.col("address.city"))

# 8. Collapse Columns into Struct
df = df.withColumn("full_address", F.struct(F.col("city"), F.col("state")))

# 9. Collect Set (Aggregation: Create Array of Unique Values)
df_agg = df.groupBy("dept").agg(F.collect_set("id").alias("id_list"))
```

UDFs

```python
from pyspark.sql.types import StringType, DoubleType
from pyspark.sql.functions import udf, pandas_udf
import pandas as pd

# 1. Standard Python UDF (Slow - Avoid for large data)
def upper_case_fn(s):
    return s.upper() if s else None

# Register as UDF with return type
upper_udf = udf(upper_case_fn, StringType())
df = df.withColumn("upper_name", upper_udf(F.col("name")))

# 2. Pandas UDF (Vectorized - Much Faster)
# Input: pd.Series, Output: pd.Series
@pandas_udf(DoubleType())
def add_one_udf(a: pd.Series) -> pd.Series:
    return a + 1

df = df.withColumn("score_plus_one", add_one_udf(F.col("score")))

# 3. Register UDF for SQL Usage
spark.udf.register("sql_upper", upper_case_fn, StringType())
df_sql = spark.sql("SELECT sql_upper(name) FROM employees")

# 4. Apply UDF with Multiple Columns
@pandas_udf(StringType())
def combine_names(first: pd.Series, last: pd.Series) -> pd.Series:
    return first + " " + last

df = df.withColumn("full_name", combine_names(F.col("first"), F.col("last")))
```

SCD Type 2

```python
# 1. Filter for active records in the target table
active = target.filter(F.col("is_active") = = "Y")

# 2. Rename source columns to avoid ambiguity in join
src = source.select(*(F.col(c).alias(f"src_{c}") for c in source.columns))

# 3. Join Source with Active Target (Full Outer to catch all cases)
joined = active.join(src, F.col("id") = = F.col("src_id"), "full")

# 4. Identify Updates (Match found BUT values changed)
# Note: Using strict inequality (≠) to detect changes
updates = joined.filter(
    F.col("id").isNotNull() & F.col("src_id").isNotNull() &
    (F.col("salary") < > F.col("src_salary"))
)

# 5. Identify New Records (No match in target)
new_recs = joined.filter(F.col("id").isNull())

# 6. Close Old Records (Expire: Set is_active=N, end_date=Today)
closed = updates.select("id", "salary", "start_date") \
    .withColumn("end_date", F.current_date()) \
    .withColumn("is_active", F.lit("N"))

# 7. Prepare New Active Records (From Updates + Brand New IDs)
# Combine, map back to original column names, and add active flags
active_new = updates.unionByName(new_recs) \
    .select("src_id", "src_salary") \
    .toDF("id", "salary") \
    .withColumn("start_date", F.current_date()) \
    .withColumn("end_date", F.lit(None)) \
    .withColumn("is_active", F.lit("Y"))

# 8. Union All: Unchanged History + Closed Records + New Active
final_df = target.join(updates, "id", "left_anti") \
    .unionByName(closed).unionByName(active_new)
```

Time Travel

```python
from delta.tables import *

# 1. Read a specific version (e.g., Version 5)
df_ver = spark.read.format("delta").option("versionAsOf", 5).load("path/to/table")

# 2. Read by Timestamp (e.g., Data as it looked yesterday)
df_time = spark.read.format("delta") \
    .option("timestampAsOf", "2023-10-25 12:00:00") \
    .load("path/to/table")

# 3. Initialize DeltaTable Object (Required for maintenance ops)
delta_tbl = DeltaTable.forPath(spark, "path/to/table")

# 4. View History (See who changed what and when)
delta_tbl.history().show()

# 5. Restore to Version (Rollback - Overwrites current state!)
delta_tbl.restoreToVersion(5)

# 6. Restore to Timestamp
delta_tbl.restoreToTimestamp("2023-10-25")

# 7. SQL Time Travel (If table is registered in Catalog)
df_sql = spark.sql("SELECT * FROM employees TIMESTAMP AS OF '2023-10-25'")

# 8. Vacuum (Cleanup old files to save space)
# Warning: You cannot time travel back past the retention period (default 7 days)
# 168 hours = 7 days
delta_tbl.vacuum(168)
```

Streaming

```python
# 1. Read Stream (Input)
# Requires a defined schema for file sources (JSON/CSV)
df_stream = spark.readStream.schema(schema).json("path/to/input_folder")

# 2. Transformations (Same as Batch!)
df_processed = df_stream.filter(F.col("status") == "error")

# 3. Write Stream to Console (Debugging only)
query = df_processed.writeStream \
    .outputMode("append") \
    .format("console") \
    .start()

# 4. Write Stream to Delta (Production)
# Checkpointing is mandatory for fault tolerance
query = df_processed.writeStream \
    .format("delta") \
    .outputMode("append") \
    .option("checkpointLocation", "path/to/checkpoints") \
    .start("path/to/output_table")

# 5. Trigger (Control batch interval)
# "availableNow=True" processes all available data then stops (Batch-like)
query = df_processed.writeStream \
    .trigger(processingTime="10 seconds") \
    .start()

# 6. Watermarking (Handle late data in aggregations)
# Drops data older than the threshold (e.g., 10 minutes)
df_agg = df_stream \
    .withWatermark("timestamp", "10 minutes") \
    .groupBy(F.window("timestamp", "5 minutes")) \
    .count()
```

Part 5: Optimize & Debug

Optimisation

```python
from pyspark import StorageLevel

# 1. Cache (Lazy: Stores in memory after first action)
df.cache()

# 2. Persist (Memory & Disk - prevents OutOfMemory errors)
df.persist(StorageLevel.MEMORY_AND_DISK)

# 3. Unpersist (Clear memory when done - Critical!)
df.unpersist()

# 4. Repartition (Increases partitions, Full Shuffle)
# Use when you need more parallelism or to fix skewed data
df_repart = df.repartition(100)

# 5. Coalesce (Decreases partitions, No Shuffle)
# Use before writing to reduce number of output files
df_coal = df.coalesce(1)

# 6. Broadcast Join (For small tables < 10MB)
# Sends copy of small table to all nodes, avoids shuffle
df_join = df_large.join(F.broadcast(df_small), "id")

# 7. Explain Plan (Check physical execution)
df.explain()

# 8. Tune Shuffle Partitions (Default is 200)
# Set equal to 2x-3x total cores, or lower for small data
spark.conf.set("spark.sql.shuffle.partitions", "200")
```

Debugging

```python
# 1. Explain Plan (Crucial for performance debugging)
# Shows the Physical Plan (indexes, join strategies used)
df.explain(mode="extended")

# 2. Show Full Content (Don't truncate long strings)
df.show(truncate=False, n=5)

# 3. Check for Skew (Count records per partition)
# If one partition has significantly more data, you have skew.
partition_counts = df.rdd.glom().map(len).collect()
print(partition_counts)

# 4. Identify Data Skew on a Key
# If one key has millions of records, joins will be slow.
df.groupBy("join_key").count().orderBy(F.col("count").desc()).show()

# 5. Check Data Types (List format)
print(df.dtypes)

# 6. Validate Nulls in Critical Columns
null_counts = df.select([
    F.count(F.when(F.col(c).isNull(), c)).alias(c)
    for c in df.columns
])
null_counts.show()

# 7. Clear Cache (If you run out of memory during iterative dev)
spark.catalog.clearCache()

# 8. Check Spark Configuration (Debug memory settings)
print(spark.conf.get("spark.sql.shuffle.partitions"))
```