PySpark Cheat Sheet

A quick reference guide to the most commonly used patterns and functions in PySpark SQL.

https://github.com/kevinschaich/pyspark-cheatsheet

https://spark.apache.org/docs/latest/api/python/index.html

https://spark.apache.org/docs/latest/api/python/reference/index.html

Table of Contents

If you can't find what you're looking for, check out the PySpark Official Documentation and add it here!

Quickstart

Install on macOS:

```
brew install apache-spark && pip install pyspark
```

Create your first DataFrame:

```
from pyspark.sql import SparkSession

spark = SparkSession.builder.getOrCreate()

# I/O options:
https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/io.ht
ml
df = spark.read.csv('/path/to/your/input/file')
```

Basics

```
# Show a preview
df.show()

# Show preview of first / last n rows
df.head(5)
df.tail(5)

# Show preview as JSON (WARNING: in-memory)
df = df.limit(10) # optional
print(json.dumps([row.asDict(recursive=True) for row in df.collect()],
indent=2))

# Limit actual DataFrame to n rows (non-deterministic)
df = df.limit(5)

# Get columns
df.columns

# Get columns + column types
df.dtypes

# Get schema
df.schema

# Get row count
df.count()

# Get column count
len(df.columns)

# Write output to disk
df.write.csv('/path/to/your/output/file')

# Get results (WARNING: in-memory) as list of PySpark Rows
df = df.collect()

# Get results (WARNING: in-memory) as list of Python dicts
dicts = [row.asDict(recursive=True) for row in df.collect()]

# Convert (WARNING: in-memory) to Pandas DataFrame
df = df.toPandas()
```

Common Patterns

Importing Functions & Types

```
# Easily reference these as F.my_function() and T.my_type() below
from pyspark.sql import functions as F, types as T
```

Filtering

```
# Filter on equals condition
df = df.filter(df.is_adult == 'Y')

# Filter on >, <, >=, <= condition
df = df.filter(df.age > 25)

# Multiple conditions require parentheses around each condition
df = df.filter((df.age > 25) & (df.is_adult == 'Y'))

# Compare against a list of allowed values
df = df.filter(col('first_name').isin([3, 4, 7]))

# Sort results
df = df.orderBy(df.age.asc()))
df = df.orderBy(df.age.desc()))
```

Joins

```
# Left join in another dataset
df = df.join(person_lookup_table, 'person_id', 'left')

# Match on different columns in left & right datasets
df = df.join(other_table, df.id == other_table.person_id, 'left')

# Match on multiple columns
df = df.join(other_table, ['first_name', 'last_name'], 'left')
```

Column Operations

```
# Add a new static column
df = df.withColumn('status', F.lit('PASS'))

# Construct a new dynamic column
df = df.withColumn('full_name', F.when(
    (df.fname.isNotNull() & df.lname.isNotNull()), F.concat(df.fname,
df.lname)
).otherwise(F.lit('N/A'))

# Pick which columns to keep, optionally rename some
df = df.select(
    'name',
    'age',
    F.col('dob').alias('date_of_birth'),
)

# Remove columns
df = df.drop('mod_dt', 'mod_username')

# Rename a column
df = df.withColumnRenamed('dob', 'date_of_birth')
```

```
# Keep all the columns which also occur in another dataset
df = df.select(*(F.col(c) for c in df2.columns))

# Batch Rename/Clean Columns
for col in df.columns:
    df = df.withColumnRenamed(col, col.lower().replace(' ', '_').replace('-
', '_'))
```

Casting & Coalescing Null Values & Duplicates

```
# Cast a column to a different type
df = df.withColumn('price', df.price.cast(T.DoubleType()))

# Replace all nulls with a specific value
df = df.fillna({
    'first_name': 'Tom',
    'age': 0,
})

# Take the first value that is not null
df = df.withColumn('last_name', F.coalesce(df.last_name, df.surname,
F.lit('N/A')))

# Drop duplicate rows in a dataset (distinct)
df = df.dropDuplicates() # or
df = df.distinct()

# Drop duplicate rows, but consider only specific columns
df = df.dropDuplicates(['name', 'height'])

# Replace empty strings with null (leave out subset keyword arg to replace
in all columns)
df = df.replace({"": None}, subset=["name"])

# Convert Python/PySpark/NumPy NaN operator to null
df = df.replace(float("nan"), None)
```

String Operations

String Filters

```
# Contains - col.contains(string)
df = df.filter(df.name.contains('o'))

# Starts With - col.startswith(string)
df = df.filter(df.name.startswith('Al'))

# Ends With - col.endswith(string)
df = df.filter(df.name.endswith('ice'))

# Is Null - col.isNull()
df = df.filter(df.is_adult.isNull())

# Is Not Null - col.isNotNull()
df = df.filter(df.first_name.isNotNull())

# Like - col.like(string_with_sql_wildcards)
df = df.filter(df.name.like('Al%'))
```

```
# Regex Like - col.rlike(regex)
df = df.filter(df.name.rlike('[A-Z]*ice$'))

# Is In List - col.isin(*cols)
df = df.filter(df.name.isin('Bob', 'Mike'))
```

String Functions

```
# Substring - col.substr(startPos, length)
df = df.withColumn('short_id', df.id.substr(0, 10))

# Trim - F.trim(col)
df = df.withColumn('name', F.trim(df.name))

# Left Pad - F.lpad(col, len, pad)
# Right Pad - F.rpad(col, len, pad)
df = df.withColumn('id', F.lpad('id', 4, '0'))

# Left Trim - F.ltrim(col)
# Right Trim - F.rtrim(col)
df = df.withColumn('id', F.ltrim('id'))

# Concatenate - F.concat(*cols)
df = df.withColumn('full_name', F.concat('fname', F.lit(' '), 'lname'))

# Concatenate with Separator/Delimiter - F.concat_ws(delimiter, *cols)
df = df.withColumn('full_name', F.concat_ws('-', 'fname', 'lname'))

# Regex Replace - F.regexp_replace(str, pattern, replacement)[source]
df = df.withColumn('id', F.regexp_replace(id, '0F1(.*)', '1F1-$1'))

# Regex Extract - F.regexp_extract(str, pattern, idx)
df = df.withColumn('id', F.regexp_extract(id, '[0-9]*', 0))
```

Number Operations

```
# Round - F.round(col, scale=0)
df = df.withColumn('price', F.round('price', 0))

# Floor - F.floor(col)
df = df.withColumn('price', F.floor('price'))

# Ceiling - F.ceil(col)
df = df.withColumn('price', F.ceil('price'))

# Absolute Value - F.abs(col)
df = df.withColumn('price', F.abs('price'))

# X raised to power Y – F.pow(x, y)
df = df.withColumn('exponential_growth', F.pow('x', 'y'))

# Select smallest value out of multiple columns – F.least(*cols)
df = df.withColumn('least', F.least('subtotal', 'total'))

# Select largest value out of multiple columns – F.greatest(*cols)
df = df.withColumn('greatest', F.greatest('subtotal', 'total'))
```

Date & Timestamp Operations

```
# Add a column with the current date
df = df.withColumn('current_date', F.current_date())

# Convert a string of known format to a date (excludes time information)
df = df.withColumn('date_of_birth', F.to_date('date_of_birth', 'yyyy-MM-
dd'))

# Convert a string of known format to a timestamp (includes time
information)
df = df.withColumn('time_of_birth', F.to_timestamp('time_of_birth', 'yyyy-
MM-dd HH:mm:ss'))

# Get year from date:       F.year(col)
# Get month from date:      F.month(col)
# Get day from date:        F.dayofmonth(col)
# Get hour from date:       F.hour(col)
# Get minute from date:     F.minute(col)
# Get second from date:     F.second(col)
df = df.filter(F.year('date_of_birth') == F.lit('2017'))

# Add & subtract days
df = df.withColumn('three_days_after', F.date_add('date_of_birth', 3))
df = df.withColumn('three_days_before', F.date_sub('date_of_birth', 3))

# Add & Subtract months
df = df.withColumn('next_month', F.add_month('date_of_birth', 1))

# Get number of days between two dates
df = df.withColumn('days_between', F.datediff('start', 'end'))

# Get number of months between two dates
df = df.withColumn('months_between', F.months_between('start', 'end'))

# Keep only rows where date_of_birth is between 2017-05-10 and 2018-07-21
df = df.filter(
    (F.col('date_of_birth') >= F.lit('2017-05-10')) &
    (F.col('date_of_birth') <= F.lit('2018-07-21'))
)
```

Array Operations

```
# Column Array - F.array(*cols)
df = df.withColumn('full_name', F.array('fname', 'lname'))

# Empty Array - F.array(*cols)
df = df.withColumn('empty_array_column', F.array([]))

# Get element at index – col.getItem(n)
df = df.withColumn('first_element', F.col("my_array").getItem(0))

# Array Size/Length – F.size(col)
df = df.withColumn('array_length', F.size('my_array'))

# Flatten Array – F.flatten(col)
df = df.withColumn('flattened', F.flatten('my_array'))
```

```
# Unique/Distinct Elements - F.array_distinct(col)
df = df.withColumn('unique_elements', F.array_distinct('my_array'))

# Map over & transform array elements - F.transform(col, func: col -> col)
df = df.withColumn('elem_ids', F.transform(F.col('my_array'), lambda x:
x.getField('id')))

# Return a row per array element - F.explode(col)
df = df.select(F.explode('my_array'))
```

Struct Operations

```
# Make a new Struct column (similar to Python's `dict()`) - F.struct(*cols)
df = df.withColumn('my_struct', F.struct(F.col('col_a'), F.col('col_b')))

# Get item from struct by key - col.getField(str)
df = df.withColumn('col_a', F.col('my_struct').getField('col_a'))
```

Aggregation Operations

```
# Row Count:                F.count()
# Sum of Rows in Group:     F.sum(*cols)
# Mean of Rows in Group:    F.mean(*cols)
# Max of Rows in Group:     F.max(*cols)
# Min of Rows in Group:     F.min(*cols)
# First Row in Group:       F.alias(*cols)
df = df.groupBy('gender').agg(F.max('age').alias('max_age_by_gender'))

# Collect a Set of all Rows in Group:       F.collect_set(col)
# Collect a List of all Rows in Group:      F.collect_list(col)
df = df.groupBy('age').agg(F.collect_set('name').alias('person_names'))

# Just take the lastest row for each combination (Window Functions)
from pyspark.sql import Window as W

window = W.partitionBy("first_name", "last_name").orderBy(F.desc("date"))
df = df.withColumn("row_number", F.row_number().over(window))
df = df.filter(F.col("row_number") == 1)
df = df.drop("row_number")
```

Advanced Operations

Repartitioning

```
# Repartition - df.repartition(num_output_partitions)
df = df.repartition(1)
```

UDFs (User Defined Functions

```
# Multiply each row's age column by two
times_two_udf = F.udf(lambda x: x * 2)
df = df.withColumn('age', times_two_udf(df.age))

# Randomly choose a value to use as a row's name
import random
```

```
random_name_udf = F.udf(lambda: random.choice(['Bob', 'Tom', 'Amy',
'Jenna']))
df = df.withColumn('name', random_name_udf())
```

Useful Functions / Transformations

```
def flatten(df: DataFrame, delimiter="_") -> DataFrame:
    '''
    Flatten nested struct columns in `df` by one level separated by
`delimiter`, i.e.:

    df = [ {'a': {'b': 1, 'c': 2} } ]
    df = flatten(df, '_')
    -> [ {'a_b': 1, 'a_c': 2} ]
    '''
    flat_cols = [name for name, type in df.dtypes if not
type.startswith("struct")]
    nested_cols = [name for name, type in df.dtypes if
type.startswith("struct")]

    flat_df = df.select(
        flat_cols
        + [F.col(nc + "." + c).alias(nc + delimiter + c) for nc in
nested_cols for c in df.select(nc + ".*").columns]
    )
    return flat_df


def lookup_and_replace(df1, df2, df1_key, df2_key, df2_value):
    '''
    Replace every value in `df1`'s `df1_key` column with the corresponding
value
    `df2_value` from `df2` where `df1_key` matches `df2_key`

    df = lookup_and_replace(people, pay_codes, id, pay_code_id,
pay_code_desc)
    '''
    return (
        df1
        .join(df2[[df2_key, df2_value]], df1[df1_key] == df2[df2_key],
'left')
        .withColumn(df1_key, F.coalesce(F.col(df2_value), F.col(df1_key)))
        .drop(df2_key)
        .drop(df2_value)
    )
```