PySpark and Spark SQL Comprehensive Guide

1. Introduction

PySpark is the Python API for **Apache Spark**, a high-performance distributed computing framework for big data. In a Spark application, data is manipulated as **DataFrames** (distributed tables) using a rich library of operations and SQL-like functions. Spark SQL builds on this by allowing you to execute SQL queries on DataFrames or tables. Databricks provides a managed Spark environment; you write PySpark code in notebooks, run it on a cluster, and can mix Python DataFrame API with SQL for data processing. This guide covers PySpark (especially on Databricks) in depth, including how to load, transform, join, filter, and save data, using built-in functions and Spark SQL queries.

2. Databricks Environment and SparkSession

In Databricks, you typically run PySpark code in a notebook attached to a compute cluster. Creating a new notebook is straightforward (e.g. in the Databricks workspace click **New ➔ Notebook**). Each notebook comes with a default Spark session object, commonly named `spark`. If you were running PySpark outside Databricks, you would create a SparkSession explicitly:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder \
    .appName("MyApp") \
    .getOrCreate()
```

You also import DataFrame functions and types. A common convention is to alias `pyspark.sql.functions` as `F` and `pyspark.sql.types` as `T` to avoid conflicts and shorten code. For example:

```
from pyspark.sql.functions import col, lit, when
import pyspark.sql.functions as F
import pyspark.sql.types as T
```

Here `col("x")` creates a column reference and `lit(value)` creates a constant column. Importing modules as shown above is recommended for clarity. In Databricks notebooks, use `display(df)` or `df.show()` to trigger execution and view results. (The `display()` function is Databricks-specific but simply calls Spark actions under the hood.) For example, after defining a DataFrame `df`, calling `display(df)` will execute the transformations and show the table.

3. Loading Data into Spark

PySpark can load data from many sources (files, databases, tables, etc.) into DataFrames. The primary entry point is the `DataFrameReader` accessed via `spark.read`. Common formats include CSV, JSON, Parquet, Delta, etc.

- **CSV:** To read a CSV file with a header row and infer schema, use:

```
df_csv = (spark.read
    .format("csv")
    .option("header", True)
    .option("inferSchema", True)
    .load("/mnt/path/to/file.csv"))
```

This creates a DataFrame from the CSV file. For example, Databricks shows this pattern for CSV uploads. You can also use `spark.read.csv("file.csv", header=True, inferSchema=True)`.

- **JSON:** Similarly, JSON files can be loaded with `spark.read.json(path)` or `format("json")`. Schemas can often be inferred or provided.
- **Parquet/ORC:** For columnar data, use `spark.read.parquet("file.parquet")` or ORC. These formats are faster and preserve types.
- **Database Tables or Hive:** If you have a table registered in the catalog (Hive or Unity Catalog), use `spark.read.table("db.schema.table")`. In Databricks, you can also use `spark.table("db.schema.table")`. The example Databricks tutorial shows:

  ```
  display(spark.read.table("samples.nyctaxi.trips"))
  ```

  which runs `SELECT *` on that table. The `spark.read.table()` method simply returns the table as a DataFrame.
- **Database connections:** You can also connect to external databases via JDBC by specifying `format("jdbc")` and connection options.

After loading data, you can call actions like `df.show()` or `df.count()` to trigger computation and inspect the data.

4. Creating DataFrames

Beyond loading from files, you can directly construct DataFrames in memory:

- **From Python data:** Use `spark.createDataFrame(data, schema)` where `data` is a Python list of tuples or dicts. For example, the following creates a simple DataFrame with a schema:

  ```
  data = [("Mikhail", 15), ("Zaky", 13), ("Zoya", 8)]
  df_kids = spark.createDataFrame(data, schema=["name","age"])
  display(df_kids)
  ```

  Spark will infer column types. To specify types explicitly, define a `StructType` schema:

  ```
  from pyspark.sql.types import StructType, StructField, StringType,
  IntegerType
  ```

```
schema = StructType([StructField("name", StringType(), True),
                     StructField("age", IntegerType(), True)])
df_kids2 = spark.createDataFrame(data, schema)
```

These examples are demonstrated in Databricks documentation.

- **From existing RDD or DataFrame:** You can also convert an RDD or list of `Row` objects to a DataFrame with `createDataFrame`.
- **From queries:** You can create DataFrames by running Spark SQL directly with `spark.sql("SELECT ...")`, or by creating temporary views (see below).

Once you have a DataFrame, you can manipulate it or save it; see later sections.

5. Basic DataFrame Transformations

PySpark DataFrames support many transformations. Common operations include:

- **Selecting columns:**

  ```
  df2 = df.select("col1", "col2")
  ```

  or using expressions:

  ```
  df2 = df.select(col("col1"), (col("col2") * 2).alias("col2x2"))
  ```

  The `col()` function returns a Column object and `alias()` renames it.

- **Filtering rows:** Use `df.filter(condition)` or `df.where(condition)`. These are equivalent. For example:

  ```
  df_filtered = df.filter((col("age") > 30) & (col("country") == "US"))
  ```

  You can also pass a SQL string: `df.where("age > 30 AND country = 'US'")`. In PySpark one often uses `col` and Python operators (`&` for AND, `|` for OR). Databricks examples show chaining filters in code (e.g. `.filter(col("o_orderstatus") == "F")`).

- **Adding or transforming columns:** Use `withColumn` to add a column or transform an existing one. Example:

  ```
  df2 = df.withColumn("age_plus_5", col("age") + 5)
  ```

  or using functions:

```
df2 = df.withColumn("status", when(col("score") > 50,
"Pass").otherwise("Fail"))
```

- **Dropping columns:**

```
df2 = df.drop("unneeded_column")
```

- **Chaining transformations:** Spark transformations are lazy and return new DataFrames. You can chain them fluently. For example, the Databricks documentation demonstrates chaining filter, groupBy, and sort in one statement:

```
from pyspark.sql.functions import count
df_chained = (
    df_orders
      .filter(col("o_orderstatus") == "F")
      .groupBy(col("o_orderpriority"))
      .agg(count(col("o_orderkey")).alias("n_orders"))
      .sort(col("n_orders").desc())
)
display(df_chained)
```

  The `.filter()`, `.groupBy()`, `.agg()`, and `.sort()` calls are all lazy transformations; nothing executes until an action (like `display` or `show`) is called.

- **Sorting and limiting:** Sort by columns with `.orderBy()` or `.sort()`. For example, `df.orderBy(col("amount").desc())`. Limit rows with `.limit(n)`. The docs note that sorting can be expensive on large data. Example:

```
df_sorted = df.orderBy(col("score").desc(), col("id").asc())
display(df_sorted.limit(10))
```

  (This matches the example in Databricks docs.)

Each of these transformation methods returns a new DataFrame. Remember that Spark uses lazy evaluation: transformations build a query plan, and execution happens when an action is called.

6. Joins and Unions

Joins

To combine DataFrames by matching rows, use `join`. The syntax is `df1.join(df2, on=condition, how="...")`. The common join types are **inner**, **left**, **right**, and **outer**. For example, given DataFrames `orders` and `customers`, you might do an inner join on the customer key:

```
df_joined = df_orders.join(
    df_customers,
    on = df_orders["o_custkey"] == df_customers["c_custkey"],
    how = "inner"
)
display(df_joined)
```

Databricks documentation illustrates this: joining `orders` with `customers` on matching keys, using an inner join (the default). That example assumes every order has exactly one matching customer. You can join on multiple conditions by using boolean expressions with `&` and `|`. For example:

```
df_complex = df_orders.join(
    df_customers,
    on = ((df_orders["o_custkey"] == df_customers["c_custkey"]) &
(df_orders["o_totalprice"] > 500000)),
    how = "inner"
)
display(df_complex)
```

This will include only rows where the join keys match *and* the order total price is over 500,000.

The join operation can be memory- and shuffle-intensive. Spark supports broadcast joins (with `broadcast(df)` hint for small tables) and careful partitioning for performance (see Databricks docs on *Work with joins*). Common pitfalls include joining on non-equi conditions or very large tables (in which case consider partitioning or limiting data first).

Unions (Appending Rows)

If you simply want to stack DataFrames vertically (they must have the same schema or compatible columns), use `union` or `unionByName`. The standard `union` appends rows, requiring both DataFrames to have the same column order and types. For example:

```
df_union = df1.union(df2)
display(df_union)
```

In Databricks, `union` is demonstrated to append rows from two DataFrames. For example, if `df_one_customer` has one row and `df_filtered_customer` has two rows, `df_one_customer.union(df_filtered_customer)` yields a DataFrame with all three rows.

Spark 3.4+ also provides `unionByName`, which aligns columns by name rather than position. This is helpful when DataFrames have the same columns in different orders or even different column sets. For instance:

```
df1 = spark.createDataFrame([[1,2,3]], ["col0","col1","col2"])
df2 = spark.createDataFrame([[4,5,6]], ["col1","col2","col0"])
df_union = df1.unionByName(df2)  # columns aligned by name
```

`unionByName` can also allow missing columns by filling them with nulls (`allowMissingColumns=True`). See the [DataFrame.unionByName documentation] and examples for details.

7. Aggregations and Grouping

Aggregation (similar to SQL `GROUP BY`) is done with `groupBy()` and aggregate functions. For example, to compute the average of a column grouped by another:

```
from pyspark.sql.functions import avg

df_segment_balance = df_customers.groupBy("c_mktsegment") \
    .agg(avg("c_acctbal").alias("avg_balance"))
display(df_segment_balance)
```

This is equivalent to `SELECT c_mktsegment, AVG(c_acctbal) FROM customers GROUP BY c_mktsegment`. The Databricks docs show exactly this example for average balance by market segment. You can group by multiple columns by passing more arguments to `groupBy()`. Common aggregate functions include `count`, `sum`, `max`, `min`, `stddev`, etc., all available in `pyspark.sql.functions`. For instance, `df.count()` returns the total row count (an action).

8. Handling Missing Data (Nulls and NaNs)

Real-world data often contains nulls (or NaNs). Spark provides methods to detect, drop, or fill these values:

- **Detect nulls:** Use expressions like `col("c").isNull()` or `col("c").isNotNull()`. For example, `df.filter(col("age").isNull())` returns rows where `age` is null. Spark's built-in functions include `isnull`/`isNotNull` and `isnan` for NaN checks.

- **Drop nulls:** `df.na.drop()` returns a new DataFrame omitting any row with a null or NaN by default. You can specify `how='any'` (drop if any column is null) or `how='all'` (drop only if all columns are null), or a threshold of non-null values. For example:

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

  This matches the DataFrame API: *"Returns a new DataFrame omitting rows with null or NaN values"*. (In code, `df.na.drop()` is an alias for `df.dropna()`.) You can also restrict dropping to certain columns using the `subset` parameter.

- **Fill nulls:** To replace nulls with a constant or other value, use `fillna()`. For instance, `df.na.fill(0)` replaces all null numeric fields with 0. You can specify different values per column by passing a dict, e.g. `df.na.fill({"age":0, "city": "unknown"})`. The docs say `DataFrame.fillna(value, subset=None) returns a new DataFrame where nulls are filled with` value`:contentReference[oaicite:30]{index=30}. In the Databricks example, account balances with null were filled with` "0"`:

```
df_acct_filled = df_customer.na.fill("0", subset=["c_acctbal"])
```

Similarly, for string columns one can use `replace` (e.g. replace empty strings with `"UNKNOWN"`):

```
df_phone_filled = df_customer.na.replace([""], ["UNKNOWN"], subset=
["c_phone"])
```

This pattern is shown in Databricks docs.

- **Replace specific values:** `df.na.replace()` can replace specific values (e.g. blank string or special marker) with another value, as shown above.
- **Filter nulls:** You can explicitly filter for null or not null:

```
df_null = df.filter(col("col").isNull())
df_notnull = df.filter(col("col").isNotNull())
```

Before heavy processing, it's common to clean nulls as shown. Dropping or imputing nulls is often necessary since many operations propagate null or can error out on null comparisons.

9. Built-in Functions and Expressions

PySpark provides a vast library of built-in functions in `pyspark.sql.functions`. These include:

- **Column creation:** `col("name")`, `column("name")`, `lit(value)` to create literal columns, and `expr("...")` to parse expression strings.
- **Conditional functions:** `when(condition, value).otherwise(other)`, `coalesce(col1, col2, ...)` to return the first non-null value, `nvl`, `ifnull`, etc. For example, `coalesce(col("a"), lit(0))` returns the first non-null of `a` or 0. These allow SQL-like CASE logic.
- **Predicate functions:** `isnull`, `isnotnull`, `like`, `rlike`, etc.. Also aggregate predicates like `equal_null` and `ilike`.
- **Arithmetic functions:** `abs`, `ceil`, `floor`, `round`, `sqrt`, `exp`, `ln`, etc. For example, `sqrt(col("value"))`. The documentation lists many, e.g. `abs`, `acos`, `atan`, etc..
- **String functions:** e.g. `substring(col, start, length)`, `concat(col1, col2, ...)`, `concat_ws(sep, col1, col2, ...)`, `upper`, `lower`, `trim`, `length`, `split`, `regexp_extract`, etc. For instance, `df.withColumn("first3", F.substring(col("name"), 1, 3))`. (Spark's SQL Functions docs list these under *String Functions*, see [3], or the Spark SQL reference [33].)
- **Date/Time functions:** e.g. `current_date()`, `current_timestamp()`, `to_date`, `datediff`, `date_add`, `year`, `month`, etc.
- **Collection functions:** `array`, `map`, `explode`, etc.

- **Window and Ranking:** PySpark also supports window functions (`pyspark.sql.Window`) for operations like `row_number`, `rank`, `lag`, etc.

Most DataFrame transformations use these functions. For example, to compute a new column with uppercase name: `df.withColumn("name_upper", F.upper(col("name")))`. (Here `F.upper` refers to `pyspark.sql.functions.upper`.)

Databricks documentation notes to import what you need or alias the module for convenience. For a complete list of PySpark SQL functions, refer to the official API docs or Spark SQL built-in functions reference.

10. Spark SQL Queries

In addition to the DataFrame API, Spark allows writing SQL queries. To do so, register DataFrames as temporary views and then use `spark.sql`. For example:

```
df.createOrReplaceTempView("my_table")
result = spark.sql("SELECT name, SUM(amount) AS total FROM my_table GROUP BY
name")
display(result)
```

This runs a SQL query on the Spark engine. The method `DataFrame.createOrReplaceTempView(name)` makes a temporary view tied to the current SparkSession. Example usage from the docs:

```
df = spark.createDataFrame([(2, "Alice"), (5, "Bob")], ["age", "name"])
df.createOrReplaceTempView("people")
df2 = spark.sql("SELECT * FROM people WHERE age > 3")
```

shows exactly how to create and query a temp view. Temporary views exist only for the session's lifetime (you can drop them with `spark.catalog.dropTempView("people")`).

Spark SQL supports all standard SQL syntax and built-in functions. You can also query external tables directly via SQL (e.g. `spark.sql("SELECT * FROM database.table")`) or use `spark.read.table()` as shown earlier. In Databricks, you often mix DataFrame operations with occasional `spark.sql` calls for complex queries or when leveraging existing SQL knowledge.

11. Writing and Saving Data

After transformations, you'll want to save the DataFrame. Spark's `DataFrameWriter` API supports writing to various formats:

- **Write to Files:**
  - CSV: `df.write.csv(path, mode="overwrite", header=True)` or with format:

    ```
    df.write.format("csv").option("header",
    True).mode("overwrite").save("/mnt/output/csv_dir")
    ```

The Spark documentation example shows writing a small DataFrame to CSV and reading it back.

- Parquet:

  ```
  df.write.parquet("/mnt/output/parquet_dir", mode="overwrite")
  ```

  This is confirmed by the Spark API docs. Parquet (and ORC) are columnar formats that support compression and schema.

- JSON: `df.write.json(path)`.

- Delta (Databricks): if you have Delta Lake enabled, `df.write.format("delta").save(path)` writes a Delta table (ACID transactions)

You can specify save modes (`overwrite`, `append`, `ignore`, `errorIfExists`). The Parquet doc notes the modes and that the default `error` will fail if data exists. The CSV doc likewise shows mode options.

After writing files, you can read them back with `spark.read.format(...).load(path)` as in the examples.

- **Save as Table:** To write a DataFrame to a table in the metastore (e.g. Unity Catalog in Databricks or a Hive database), use `saveAsTable` (or `write.saveAsTable`). For example:

  ```
  df.write.saveAsTable(f"{catalog}.{schema}.{table}")
  ```

  The Databricks guide shows:

  ```
  df_joined.write.saveAsTable(f"{catalog_name}.{schema_name}.{table_name}")
  ```

  which writes `df_joined` as a managed table. This table can later be queried by SQL or via `spark.read.table()`.

- **Partitioning:** You can partition the output by certain columns using `.partitionBy("col1","col2")` before saving. This is commonly used for Parquet/Delta to speed up reads by partition pruning. E.g. `df.write.partitionBy("year","month").parquet(path)`.

- **Format options:** Many write options can be set via `.option()`. For example, for CSV you might set delimiter or compression. For Parquet/ORC you can set compression with `.option("compression","snappy")`.

In summary, Spark's writer methods are very flexible: `df.write.format("format").mode(...).option(...).save(path)` or specific shortcuts like `.parquet()`, `.csv()`, `.saveAsTable()`.

12. DataFrame Persistence (Caching)

For iterative algorithms or repeated queries on the same DataFrame, it is efficient to cache or persist the data in memory/disk. You can call `df.cache()` or `df.persist()` to keep the DataFrame in memory across operations. For example:

```
df_transformed = df.someTransforms()
df_transformed.cache()
# now multiple actions on df_transformed will reuse cached data
```

Caching is optional but can dramatically improve performance for repeated access.

13. SQL vs DataFrame API

PySpark's DataFrame API and Spark SQL are two sides of the same coin. Operations that you can do with SQL, you can also do with DataFrame methods and functions, and vice versa. For example:

- **SQL:** `spark.sql("SELECT * FROM table WHERE x > 5")`
- **DataFrame:** `spark.table("table").filter(col("x") > 5)`

All Spark SQL built-in functions (like `substring`, `sum`, `count`, etc.) are available as DataFrame functions (in `pyspark.sql.functions`) and in SQL expressions. Use whichever is more convenient. Many developers mix both in notebooks; Databricks even allows mixing languages (SQL, Python, Scala, R in the same notebook).

14. Example: Financial Data Reconciliation

Data reconciliation refers to comparing two datasets (often from different systems) to identify discrepancies and ensure consistency. In a financial scenario, imagine you have two sources of transaction data (e.g. *System A* and *System B*) and you want to verify that each transaction's amount matches in both systems.

A typical reconciliation approach in PySpark is:

1. **Load data:** Read the two sources as DataFrames, e.g. `dfA` and `dfB`, each with a common key column (e.g. `transaction_id`).

2. **Join:** Perform a full outer join on the key, so you have one row per transaction with columns from both sources. For example:

   ```
   from pyspark.sql.functions import col, when
   comparison = dfA.alias("A").join(dfB.alias("B"), on="transaction_id",
   how="full")
   ```

   (This aligns matching transaction IDs and leaves nulls where a transaction is missing from one side.) This mirrors the approach in a reconciliation blog, which uses a join on a *natural key*.

3. **Compare columns:** Create indicator columns for matches/mismatches. For example, to check if `amount` matches:

```
df_comp = comparison.withColumn(
    "amt_match",
    when(col("A.amount") == col("B.amount"), True).otherwise(False)
)
```

   You can apply this to all relevant columns (amounts, dates, accounts) using functions like `when`, `coalesce`, or direct equality. Viral Patel's reconciliation example uses expressions like `coalesce((df1.col == df2.col) | (df1.col.isNull() & df2.col.isNull()), lit(False))` to flag matches or mismatches.

4. **Analyze results:** Filter or count the mismatches. For example, `df_comp.filter(col("amt_match") == False).show()` will show transactions where the amounts differ between systems.

This workflow ensures that for each transaction key, you compare the values from System A vs System B. The goal is to have a *true/false* match indicator, as described in the reconciliation function example. In practice, you might mark matches and output mismatches to investigate further.

Real-world reconciliation can be complex (partial matches, key mismatches, etc.), but the essence is using Spark joins and column comparisons. The theory is that *"Reconciliation of DataFrame refers to the process of identifying and resolving discrepancies between two or more data sources"*. Spark's DataFrame API—with `join`, `when`, `coalesce`, etc.—provides the tools to do this at scale.

15. Additional Notes and Best Practices

- **Performance:** Use broadcast joins (`broadcast(df)`) when one table is small. Use partitioning (both Spark partitions and data partitioning) wisely. Avoid shuffling unnecessarily.

- **Schema Management:** Defining schemas up front (rather than inferring) can prevent mistakes and speed up reads.

- **Use Spark SQL Functions:** Many operations (e.g. string manipulation, date math) are easier with built-in functions (`pyspark.sql.functions`). Avoid Python UDFs when possible, as built-ins run in the JVM and are faster. Databricks encourages the pandas-on-Spark API for Pandas users for easier syntax in Spark.

- **Version Compatibility:** PySpark evolves; some functions (like `unionByName`) require newer Spark versions (>=2.3). Always check the version docs.

- **Data Quality:** Always handle nulls, duplicates, and data types explicitly. Use `df.printSchema()` and `df.show()` to inspect intermediate results frequently.

- **Testing:** Use `df.show()` or `df.count()` to test small parts of your transformation pipelines as you build them.

- **Documentation and Help:** The official PySpark documentation (api docs) and Databricks guides are invaluable. For example, the Spark API docs list every DataFrame method and function (e.g. [Functions — PySpark 4.0.0 documentation]). The Databricks Knowledge Base and community forums often have code examples for common tasks.

16. Summary

This guide has covered the key concepts and operations in PySpark (with a Databricks perspective), including data ingestion, DataFrame creation, transformations (select, filter, join, union, groupBy), handling nulls, using built-in functions, Spark SQL integration, and saving results. Throughout, we have seen that Spark's unified DataFrame API (in Python or SQL) provides a powerful framework for large-scale data processing, with examples of real-world patterns like financial data reconciliation. The combination of comprehensive built-in functions, flexible data sources, and SQL compatibility makes PySpark a versatile tool for modern data engineering and analytics.