Spark SQL Fundamentals

What is Spark SQL?

Spark SQL is a Spark module for:

- Processing **structured** and **semi-structured** data
- Running SQL queries on DataFrames
- Connecting to **Hive**, **JDBC**, **Parquet**, etc.
- Unifying **RDDs**, **DataFrames**, and **SQL**

SparkSession (Entry Point)

```
import org.apache.spark.sql.SparkSession
val spark = SparkSession.builder()
    .appName("SparkSQLApp")
    .master("local[*]")
    .getOrCreate()
```

DataFrame Basics

Creating DataFrames:

```
val df = spark.read.option("header", true).csv("data.csv")
```

Viewing & Inspecting:

```
df.show()
df.printSchema()
df.columns
```

SQL Queries

Register Temp Table:

```
df.createOrReplaceTempView("employees")
```

Run SQL:

```
val result = spark.sql("SELECT name, salary FROM employees
WHERE salary > 50000")
result.show()
```

Common DataFrame Operations

Category	Examples
Filtering	df.filter($"age" > 30)
Selecting	df.select("name", "salary")
Aggregation	df.groupBy("dept").agg(avg("salary"))
Sorting	df.orderBy($"salary".desc)
Joins	df1.join(df2, "id")

Data Sources

CSV:

```
spark.read.option("header", true).csv("file.csv")
```

Parquet:

```
spark.read.parquet("data.parquet")
```

JSON:

```
spark.read.json("data.json")
```

JDBC:

```
spark.read
  .format("jdbc")
  .option("url", "jdbc:mysql://localhost:3306/db")
  .option("dbtable", "table")
  .option("user", "root")
  .option("password", "pass")
  .load()
```

Spark SQL Schema Management

Schema management in Spark SQL ensures structured and consistent handling of data across diverse formats (CSV, JSON, Parquet, etc.). It's crucial for optimizing performance, managing data integrity, and reducing runtime failures.

Schema Inference vs. Schema Definition

Approach	Description
Schema Inference	Spark guesses the schema from data (default)
Manual Schema	You explicitly define the schema using StructType

Schema Inference (Default)

```
val df = spark.read.option("header", true).csv("people.csv")
df.printSchema()
```

Drawback: Type might default to StringType or incorrect types in some cases.

Manually Defining a Schema (Recommended)

```
import org.apache.spark.sql.types._
val schema = StructType(Array(
  StructField("id", IntegerType, nullable = true),
  StructField("name", StringType, nullable = true),
  StructField("salary", DoubleType, nullable = true),
  StructField("dept", StringType, nullable = true)
))

val df = spark.read
  .option("header", true)
  .schema(schema)  // 💡 Force Spark to use your schema
  .csv("employees.csv")
```

Advanced Dataset operations

The Dataset API in Apache Spark (especially in Scala and Java) is a powerful abstraction that combines the benefits of both RDDs and DataFrames.

Advantages of the Dataset API

1. Compile-Time Type Safety

- Unlike DataFrames (which are untyped), Datasets are strongly typed.
- You get compile-time error checking, reducing runtime issues.

```
case class Person(name: String, age: Int)

val ds = Seq(Person("Alice", 30)).toDS()

// You can't accidentally reference a non-existent field
```

2. Object-Oriented Programming Style

- Allows you to use custom classes and work with familiar Scala collections-style operations.

```
ds.map(_.age + 5) // Cleaner and more natural for Scala/Java devs
```

3. Optimized by Catalyst Engine

- Dataset transformations are optimized by Spark's Catalyst optimizer, just like DataFrames.
- This leads to better performance than RDDs.

4. Serialization with Tungsten Engine

- Uses Tungsten binary format for in-memory computation, reducing GC overhead and boosting performance.

5. Interoperability

- You can convert seamlessly between Datasets, DataFrames, and RDDs.

```
val ds = df.as[Person]       // DataFrame to Dataset
val df2 = ds.toDF()          // Dataset to DataFrame
val rdd = ds.rdd             // Dataset to RDD
```

6. **Rich Functional API**
   - Includes familiar operations like map, flatMap, filter, groupBy, reduce, etc.

7. **Better for Complex Transformations**
   - Especially useful when working with complex user-defined logic that's harder to express in SQL or DataFrame DSL.

8. **Better Performance than RDDs (Usually)**
   - While not always as fast as DataFrames, Datasets typically outperform RDDs due to Catalyst + Tungsten optimization.

When to Use Dataset API

Use Case	Recommended API
Strong typing & compile-time safety	Dataset
Complex business logic	Dataset
SQL-style transformations	DataFrame
Best performance & optimization	DataFrame
Fine-grained control (low-level)	RDD

What is Catalyst Optimizer?

Catalyst is a rule-based and cost-based query optimizer built into Spark's SQL engine. It applies a series of transformations to improve query performance without changing the output.

It works on:

- DataFrames
- Datasets
- Spark SQL Queries

Catalyst Optimization Phases

Here's a breakdown of the four major phases:

Phase	Description
1. Analysis	Parses your query and resolves column names, types, and references.
2. Logical Optimization	Applies rules like predicate pushdown, constant folding, etc.
3. Physical Planning	Converts the optimized logical plan to one or more physical plans.
4. Code Generation	Generates optimized Java bytecode using Tungsten engine for execution.

Internal Representation

Catalyst represents queries as a tree structure of operators called an Abstract Syntax Tree (AST).

```
val df = people.filter($"age" > 30).select("name")
```

Spark internally:

1. Builds a logical plan for filter and select
2. Applies optimizations (like pushing filter earlier)
3. Chooses the best physical plan (e.g., broadcast join vs. shuffle)
4. Generates bytecode

Catalyst + Tungsten = Spark Speed

- Catalyst optimizes queries
- Tungsten optimizes memory and execution

Together, they give Spark:

- Low-level memory management
- Reduced garbage collection
- Efficient code generation

Performance Tuning

Key Components You Should Tune

1. Data Serialization

- Use Kryo serialization instead of Java's default serializer.
- Faster and more compact.

```
sparkConf.set("spark.serializer", "org.apache.spark.serializer.KryoSerializer")
```

2. Partitioning Strategy

- Default partitions: # of cores
- Bad partitioning = too few or too many tasks = performance hit

Use:

```
rdd.repartition(n)       // Increase partitions
rdd.coalesce(n)          // Decrease partitions (less shuffle)
```

For DataFrames:

```
df.repartition($"col")  // Partition by column
```

3. Caching & Persistence

- Avoid recomputation of frequently used data:

```
val cachedDF = df.cache() // or
df.persist(StorageLevel.MEMORY_AND_DISK)
```

Use .unpersist() when done.

4. Broadcast Variables

- For small lookup tables:

val bVar = spark.sparkContext.broadcast(lookupMap)

Avoids sending same data to every task.

5. Avoid Shuffles When Possible

- Expensive! Happens in:
    - groupByKey, distinct, join, repartition
- Prefer:
    - reduceByKey over groupByKey
    - map-side joins (broadcast joins) if one side is small

6. Memory Management

- Tune executor memory via:

--executor-memory 4G

--driver-memory 2G

Avoid spilling to disk by giving enough memory
Monitor StorageMemory vs ExecutionMemory

7. Using the Right File Format

- Prefer Parquet or ORC over CSV/JSON
- Supports:
    - Columnar storage
    - Predicate pushdown
    - Compression

8. Avoid Wide Transformations

- Wide: groupBy, join, distinct → causes shuffles
- Narrow: map, filter, flatMap → no shuffle

    Keep your pipeline narrow whenever possible.

9. Use .explain() and Spark UI

- .explain(true) helps you analyze:
    - Logical Plan
    - Physical Plan
    - Shuffles and scans
- Use Spark UI at http://localhost:4040 to:
    - Monitor stages, tasks, memory, time, GC

10. Control Parallelism

- Adjust using:

    --num-executors 4

    --executor-cores 2

    --executor-memory 4G

    Balance between CPU and Memory usage.

DataFrame Operations in scala

1. Creating a DataFrame

```
import org.apache.spark.sql.SparkSession
import org.apache.spark.sql.functions._

val spark = SparkSession.builder()
```

```
  .appName("DataFrameOps")
  .master("local[*]")
  .getOrCreate()

import spark.implicits._

val data = Seq(
  (1, "Alice", 25),
  (2, "Bob", 30),
  (3, "Charlie", 28)
)

val df = data.toDF("id", "name", "age")
df.show()
```

2. Basic Operations

➢ Select Columns

```
df.select("name", "age").show()
df.select($"name", $"age" + 5).show() // with expressions
```

➢ Filtering

```
df.filter($"age" > 26).show()
df.where($"name" === "Alice").show()
```

➢ With Expressions

```
df.selectExpr("name", "age + 10 as age_plus_10").show()
```

3. Adding/Updating Columns

➢ Add Column

```
df.withColumn("age_in_5_years", $"age" + 5).show()
```

➢ Rename Column

```
df.withColumnRenamed("name", "full_name").show()
```

4. Sorting & Ordering

```
df.orderBy($"age".desc).show()
df.sort("name").show()
```

5. Aggregations & Grouping

```
df.groupBy("age").count().show()
df.agg(avg("age"), max("age")).show()
```

With groupBy and aggregations:

```
df.groupBy("age").agg(
  count("*").as("count"),
  avg("age").as("average_age")
).show()
```

6. Joins

```
val dept = Seq(
  (1, "HR"), (2, "IT"), (3, "Sales")
).toDF("id", "dept")

val joined = df.join(dept, Seq("id"), "inner")
joined.show()
```

Join types: "inner", "left", "right", "outer", "left_semi", "left_anti"

7. Null Handling

```
df.na.drop().show()                  // Drop rows with nulls
df.na.fill(0, Seq("age")).show()     // Replace null age with 0
```

8. DataFrame to Dataset

```
case class Person(id: Int, name: String, age: Int)
val ds = df.as[Person]
```

9. Reading & Writing Files

```
val newDf = spark.read.option("header", "true").csv("people.csv")
df.write.mode("overwrite").json("output/json")
df.write.parquet("output/parquet")
```

10. Schema Inspection

```
df.printSchema()
df.schema
```

UDF (User Defined Function) in Apache Spark (Scala)

A UDF (User Defined Function) allows you to define custom logic and apply it to DataFrames when built-in Spark functions aren't sufficient.

1. Basic Syntax to Create a UDF

```
import org.apache.spark.sql.functions.udf
// Define a Scala function
def upperCase(str: String): String = str.toUpperCase

// Convert it into a UDF
```

```
val upperCaseUDF = udf(upperCase)
```

2. Using UDF in a DataFrame

```
import org.apache.spark.sql.SparkSession
val spark = SparkSession.builder()
  .appName("UDF Example")
  .master("local[*]")
  .getOrCreate()

import spark.implicits._

val df = Seq(
  (1, "alice"),
  (2, "bob"),
  (3, "charlie")
).toDF("id", "name")

df.withColumn("upper_name", upperCaseUDF($"name")).show()
```

Output:

```
+---+-------+----------+
|id |name   |upper_name|
+---+-------+----------+
|1  |alice  |ALICE     |
|2  |bob    |BOB       |
|3  |charlie|CHARLIE   |
+---+-------+----------+
```

3. UDF with Multiple Arguments

```
val concatUDF = udf((name: String, id: Int) => s"$name-$id")


df.withColumn("custom_id", concatUDF($"name", $"id")).show()
```

4. Registering UDF for SQL Queries

```
spark.udf.register("toUpper", upperCase(_: String))

df.createOrReplaceTempView("people")

spark.sql("SELECT id, toUpper(name) as name_upper FROM
people").show()
```

When to Avoid UDFs

- Performance is lower compared to native Spark functions (not optimized by Catalyst).
- They break query optimization (black-box to the optimizer).
- Not portable to other languages or engines like Spark SQL or PySpark easily.

Data Cleaning

Data Cleaning and Transformation using Apache Spark DataFrames in Scala — essential steps to prepare your data for analysis or modeling.

1. Handling Missing Data

Drop rows with nulls

```
val cleanDF = df.na.drop()
```

Fill null values

```
val filledDF = df.na.fill(Map(
  "age" -> 30,
  "name" -> "Unknown"
```

```
))
```

Replace specific values

```
val replacedDF = df.na.replace("status", Map("N/A" -> "Unknown"))
```

2. Filtering and Removing Duplicates

Filter rows

```
val filteredDF = df.filter($"age" > 18)
```

Drop duplicates

```
val uniqueDF = df.dropDuplicates("id")
```

3. Type Casting

```
val castedDF = df.withColumn("age", $"age".cast("Integer"))
```

4. String Operations

```
import org.apache.spark.sql.functions._

val transformedDF = df.withColumn("name_upper", upper($"name"))
        .withColumn("name_trim", trim($"name"))
        .withColumn("name_length", length($"name"))
        .withColumn("name_substr", substring($"name", 1, 3))
```

5. Date and Time Transformation

```
val dateDF = df.withColumn("date", to_date($"date_string", "yyyy-MM-dd"))
            .withColumn("year", year($"date"))
            .withColumn("month", month($"date"))
```

6. Creating New Columns

```
val newColDF = df.withColumn("age_group", when($"age" < 18, "Minor")
```

```
                .when($"age" >= 18 && $"age" < 60, "Adult")
                .otherwise("Senior"))
```

7. Splitting and Exploding

Split a string column into array

```
val splitDF = df.withColumn("words", split($"sentence", " "))
```

Explode array into rows

```
val explodedDF = splitDF.withColumn("word", explode($"words"))
```

8. Aggregations and Grouping

```
val aggDF = df.groupBy("department")
        .agg(avg($"salary").alias("avg_salary"),
            max($"salary").alias("max_salary"),
            count("*").alias("count_employees"))
```

9. Joins for Data Enrichment

```
val enrichedDF = df.join(otherDF, Seq("id"), "left")
```

10. Chaining Transformations

```
val finalDF = df.filter($"age" > 18)
            .withColumn("age_plus_10", $"age" + 10)
            .groupBy("age_plus_10")
            .count()
```