Lab 7: Understanding InferSchema and Defining Custom Schemas for CSV Files

Author: Dr. Sandeep Kumar Sharma

Learning Objective

In this lab, you will understand what **inferSchema=True** means in Spark, why it is important, and how to override Spark's automatic schema detection by defining your **own custom schema** when reading CSV files.

Learning Outcome

By the end of this lab, you will be able to: - Explain the concept of schema inference - Read CSV files with and without schema inference - Define a custom schema manually using Spark DataTypes - Compare behavior differences between inferred schema and user-defined schema - Understand scenarios where custom schemas are mandatory

---

Lab Information

We will use the same dataset:

```
/FileStore/tables/employee.csv
```

Example data (assumed):

```
employee_id,name,age,department,salary
101,John,29,IT,65000
102,Asha,31,HR,72000
103,Raj,28,Finance,68000
```

Section 1: What Does inferSchema=True Mean?

When Spark reads a CSV file, it treats every column as a **string** by default. That means numbers, dates, integers — everything is read as a string unless you tell Spark otherwise.

If you want Spark to automatically detect column data types (Integer, Float, String, etc.), you must enable:

```
inferSchema=True
```

This tells Spark:

> "Look at the data inside the file and guess the correct data type for each column."

If inferSchema=False (default): - All columns become **string** - Arithmetic operations will fail (because numbers are treated as text) - Sorting numbers becomes lexicographically wrong (e.g., 100 < 2)

So inferSchema=True is useful, but not perfect — Spark may guess incorrectly when data is inconsistent.

Section 2: Reading CSV *Without* Schema Inference

```
file_path = "/FileStore/tables/employee.csv"

df_no_schema = spark.read.csv(file_path, header=True)

display(df_no_schema)
df_no_schema.printSchema()
```

Expected output (Schema)

```
root
 |-- employee_id: string (nullable = true)
 |-- name: string (nullable = true)
 |-- age: string (nullable = true)
 |-- department: string (nullable = true)
 |-- salary: string (nullable = true)
```

Everything is a **string** here.

Section 3: Reading CSV *With* inferSchema=True

```
df_infer = spark.read.csv(file_path, header=True, inferSchema=True)
```

```
display(df_infer)
df_infer.printSchema()
```

Expected output

```
root
 |-- employee_id: integer
 |-- name: string
 |-- age: integer
 |-- department: string
 |-- salary: integer
```

Now Spark automatically assigns correct data types.

Section 4: Defining a Custom Schema (Schema Enforcement)

There are many cases where you **must not trust** Spark's inference: - When data is huge (schema inference is slow) - When first few rows are misleading - When downstream systems expect strict schema - When there are NULL values that confuse Spark

So now we define our own schema.

Step 1 — Import DataTypes

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

Step 2 — Define Schema Manually

```
employee_schema = StructType([
    StructField("employee_id", IntegerType(), True),
    StructField("name", StringType(), True),
    StructField("age", IntegerType(), True),
    StructField("department", StringType(), True),
    StructField("salary", IntegerType(), True)
])
```

Step 3 — Read CSV Using Custom Schema

```
df_custom = spark.read.csv(
    file_path,
    header=True,
    schema=employee_schema
)

display(df_custom)
df_custom.printSchema()
```

Now Spark will **strictly enforce** this schema.

Section 5: Scenario 1 — Schema Mismatch Error

Let's say your schema expects `IntegerType` for age, but your CSV contains a wrong value:

```
age
25
30
NaN
thirty-five   <-- invalid
```

If you read with custom schema:

```
df_custom_error = spark.read.csv(file_path, header=True, schema=employee_schema)
```

Spark may: - Throw an error - Return NULL for invalid rows - Move bad records into `_corrupt_record` column (in permissive mode)

This is good for **data validation**.

Section 6: Scenario 2 — Enforcing Schema to Clean Data

If a file has leading/trailing spaces or mixed formats:

```
101 , John , 29 , IT , 65000
```

Spark with inferSchema might misbehave.

But custom schema + `trim()` solves the issue:

```
from pyspark.sql.functions import trim

clean_df = (spark.read.csv(file_path, header=True, schema=employee_schema)
                   .select([trim(c).alias(c) for c in df_custom.columns]))

display(clean_df)
```

Section 7: Scenario 3 — Changing Data Types (Casting)

Suppose `salary` comes as string and you want to convert it to integer.

```
df_casted = df_no_schema.withColumn("salary", df_no_schema.salary.cast("int"))

display(df_casted)
df_casted.printSchema()
```

Section 8: When Should You Use Custom Schema?

Use custom schema when: - You want **consistent**, predictable schema - Data contains invalid or dirty values - You want fast loading without inference - You want to validate records - You work in production-grade pipelines

---

End of Lab 7

You now fully understand: - What inferSchema does - Why schema inference is not always safe - How to define your own schema - How Spark enforces schema - How to handle mismatch, casting, and cleaning scenarios

If you want, we can create the next lab on **Advanced Schema Evolution**, **Delta Schema Enforcement**, or **Handling Corrupted Records**.