Higher-Level APIs: DataFrames & Spark SQL

Why Not Use RDDs?

- **Verbose, lacks schema support, and has no query optimization.**

What are Higher-Level APIs in Spark?

- Higher-level APIs include **DataFrame** and **SparkSQL**.
- These APIs provide schema-aware processing and are easier to use compared to RDDs.

Advantages of DataFrames & Spark SQL

1. **Schema Awareness** – Provides metadata for structured data.
2. **Performance Optimization** – Uses **Catalyst Optimizer & Tungsten Execution Engine**.
3. **Ease of Use** – Supports **SQL-like querying**.
4. **Integration** – Works with **JSON, Parquet, CSV, Avro**.

DataFrame

- A **distributed collection of data** organized into named columns.
- Similar to a **table** in a relational database or a **data frame** in Python's pandas or R.
- Supports SQL-like operations and is optimized for large-scale data processing.

Spark DataFrame Reading Process

Steps in `spark.read`

1. **Format** → Specifies the file format (e.g., CSV, JSON, Parquet).
2. **Header** → Determines whether the first row should be treated as column names.
3. **Infer Schema** → Automatically detects column data types (**Avoid This**).
4. **Load** → Reads data from a specified source (e.g., HDFS, S3, local storage).

Best Practices for Data Reading

- Avoid `inferSchema=True` (triggers extra jobs).
- **Manually define schema** for efficiency.

Behavior of `spark.read` in Different Scenarios

1. Without `inferSchema`, with `header=True`

- **Behavior:** Eager Evaluation
- **Explanation:**
  - Reads only the first line of the file to determine column names.
  - Triggers a lightweight job using `collect` with `limit 1`.

2. With `inferSchema=True`

- **Behavior:** Eager Evaluation
- **Explanation:**
  - Scans the entire dataset to infer the schema.
  - Triggers **two jobs**:
    a. Reads column names.
    b. Infers data types for each column.

3. With Explicit Schema Definition

- **Behavior:** Lazy Evaluation
- **Explanation:**
  - No upfront job is triggered.
  - Schema validation occurs only when an **action** is performed on the DataFrame.

Key Takeaways

- **Eager Evaluation:** Using `inferSchema=True` triggers jobs immediately.
- **Lazy Evaluation:** Defining the schema explicitly avoids upfront computation.
- **Performance Tip:**
  - For large datasets, explicitly defining the schema **improves performance** by preventing unnecessary scans.

SparkSQL

- Enables **querying structured data** using SQL.
- Provides **integration** between SQL and Python/Scala/Java APIs.

Schema Enforcement in Spark DataFrame

Challenges with `inferSchema=True`

1. **Incorrect Inference** → Spark might detect incorrect data types.
2. **Performance Issues** → Inferring schema requires scanning the dataset, increasing overhead.
3. **Not Suitable for Production** → Unreliable schema detection can lead to **inconsistencies** in data processing.

Issues with Headers & Schema Inference

- Headers are mapped to generic column names (`c1` , `c2` , etc.).
- Schema inference may produce **incorrect** or **inconsistent** results.

Schema Enforcement Techniques

To avoid the drawbacks of `inferSchema` , enforce schema using the following methods:

1. `StructType` (Recommended)

- **Why?**
  - Avoids incorrect type inference.
  - Ensures a consistent schema across datasets.
  - Enables **stricter validation**, preventing unexpected `NULL` values.

2. DDL String

- Schema can also be defined using a **Data Definition Language (DDL) string** for flexibility.

Creating Temporary, Global or Persistent View in Apache Spark

[View Spark SQL Notes]