02 Intro to DataFrames -KirkYagami 👨‍💻🕵️‍♂️

1. Introduction to Spark DataFrames

A Spark DataFrame is a distributed collection of data organized into named columns, similar to a table in a relational database. DataFrames provide a high-level API for working with structured data and integrate seamlessly with Spark's SQL capabilities.

2. Creating a Spark DataFrame

A DataFrame can be created from various data sources, including CSV files, JSON files, and databases.

- **Example: Creating a DataFrame from a CSV File**

```
from pyspark.sql import SparkSession

# Initialize SparkSession
spark = SparkSession.builder \
    .appName("Spark DataFrame Example") \
    .getOrCreate()

# Load data into DataFrame
disney_df = spark.read \
  .format("csv") \
  .option("header", "true") \
  .option("inferSchema", "true") \
  .option("multiline", "true") \
  .option("quote", "\"") \
  .option("escape", "\"") \
  .load("disney_plus_shows.csv")

# Show schema and first few rows
disney_df.printSchema()
disney_df.show(truncate=False)
```

This example demonstrates how to load a CSV file into a Spark DataFrame and view its schema and data.

3. Exploring DataFrames

Spark DataFrames provide methods to inspect and manipulate the data.

- **Example: Displaying Basic Information**

```
# Display the first few rows
disney_df.show()

# Display column names
print(disney_df.columns)
```

```
# Display the schema
disney_df.printSchema()
```

These methods allow you to view the content, column names, and schema of the DataFrame.

4. Selecting and Filtering Data

DataFrames support various operations for selecting and filtering data.

- **Example: Selecting Specific Columns**

```
# Select specific columns
selected_columns = disney_df.select("title", "genre", "imdb_rating")
selected_columns.show(truncate=False)
```

- **Example: Filtering Data**

```
# Filter data based on IMDb rating
high_rated_shows = disney_df.filter(disney_df.imdb_rating > 7.0)
high_rated_shows.show(truncate=False)
```

These examples demonstrate how to select specific columns and filter rows based on conditions.

5. Aggregations and Grouping

Spark DataFrames support SQL-like aggregation and grouping operations.

- **Example: Grouping and Aggregating Data**

```
from pyspark.sql.functions import count

# Group by genre and count the number of shows
genre_counts = disney_df.groupBy("genre").agg(count("*").alias("count"))
genre_counts.show(truncate=False)
```

This example counts the number of shows in each genre.

6. Sorting Data

You can sort DataFrames by one or more columns.

- **Example: Sorting Data**

```
# Sort by IMDb rating in descending order
sorted_shows = disney_df.orderBy(disney_df.imdb_rating.desc())
sorted_shows.show(truncate=False)
```

This example sorts the DataFrame by IMDb rating in descending order.

7. Handling Missing Data

Handling missing data is crucial for data quality and analysis.

- **Example: Dropping Rows with Null Values**

```
# Drop rows with null values in any column
clean_df = disney_df.na.drop()
clean_df.show(truncate=False)
```

- **Example: Filling Null Values**

```
# Fill null values with a default value
filled_df = disney_df.fillna({"genre": "Unknown", "imdb_rating": 0})
filled_df.show(truncate=False)
```

These examples show how to handle missing data by dropping rows or filling null values.

8. Transforming Data

DataFrames allow for various data transformations, including adding new columns and modifying existing ones.

- **Example: Adding a New Column**

```
from pyspark.sql.functions import col

# Add a new column indicating if the IMDb rating is high
transformed_df = disney_df.withColumn("is_high_rated", col("imdb_rating") > 7.0)
transformed_df.show(truncate=False)
```

- **Example: Renaming a Column**

```
# Rename a column
renamed_df = disney_df.withColumnRenamed("imdb_rating", "rating")
renamed_df.show(truncate=False)
```

These examples demonstrate how to add a new column and rename an existing column.

9. Joins

DataFrames support various types of joins for combining data from multiple DataFrames.

- **Example: Inner Join**

```
# Create another DataFrame for the join example
other_df = spark.createDataFrame([
    ("tt0147800", "10 Things I Hate About You"),
    ("tt7019028", "101 Dalmatian Street")
], ["imdb_id", "title"])

# Perform an inner join
```

```
joined_df = disney_df.join(other_df, on="imdb_id", how="inner")
joined_df.show(truncate=False)
```

- **Example: Left Join**

```
# Perform a left join
left_joined_df = disney_df.join(other_df, on="imdb_id", how="left")
left_joined_df.show(truncate=False)
```

These examples illustrate how to perform inner and left joins on DataFrames.

10. Window Functions

Window functions perform operations over a range of rows related to the current row.

- **Example: Using Window Functions**

```
from pyspark.sql.window import Window
from pyspark.sql.functions import rank

# Define a window specification
window_spec = Window.partitionBy("genre").orderBy(col("imdb_rating").desc())

# Apply window function to rank shows by IMDb rating within each genre
ranked_df = disney_df.withColumn("rank", rank().over(window_spec))
ranked_df.show(truncate=False)
```

This example ranks shows within each genre by IMDb rating.

11. Real-World Examples

1. **Data Exploration and Reporting:**
   - Use DataFrames to load and explore large datasets, perform aggregations, and generate reports for business insights.
2. **ETL Pipelines:**
   - DataFrames are used in ETL (Extract, Transform, Load) pipelines to transform raw data into a format suitable for analysis or storage.
3. **Data Cleansing:**
   - Apply transformations to clean and prepare data for further analysis, such as handling missing values and filtering out irrelevant records.
4. **Customer Segmentation:**
   - Use DataFrames to analyze customer data, segment customers based on various attributes, and generate insights for targeted marketing strategies.
5. **Real-Time Data Processing:**
   - Process streaming data in real time, applying transformations and aggregations to monitor live metrics or detect anomalies.

12. Conclusion

Spark DataFrames provide a powerful and flexible API for handling and manipulating structured data. They support a wide range of operations, including data selection, filtering, aggregation, and transformation. By leveraging DataFrames, you can efficiently perform complex data processing tasks and gain valuable insights from your datasets.