SEAS 8515 - Lecture 4
Spark SQL and DataFrames

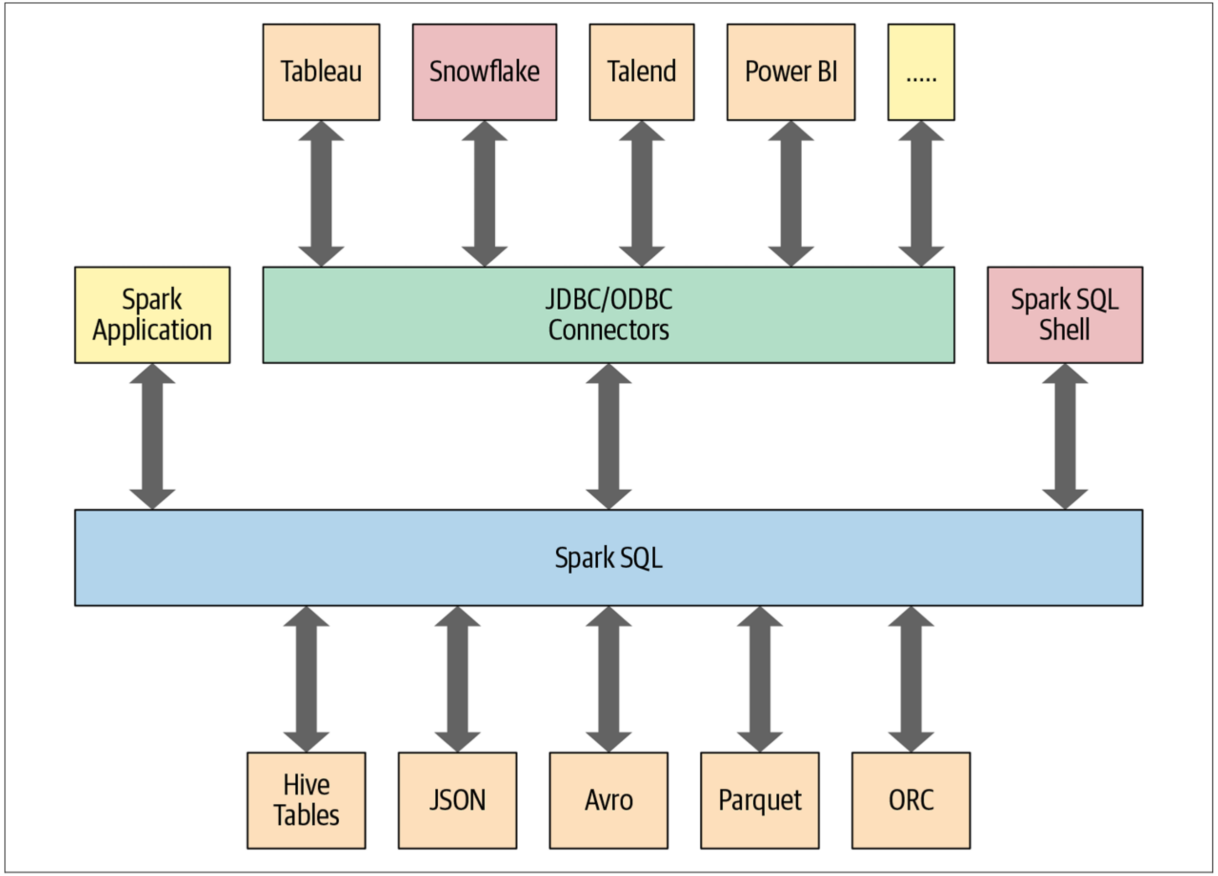
School of Engineering
& Applied Science
THE GEORGE WASHINGTON UNIVERSITY

Dr. Adewale Akinfaderin

Spark SQL

- Spark SQL serves as the foundation for the high-level Structured APIs discussed in Chapter 3.
- It supports reading and writing in various structured formats like JSON, Hive tables, Parquet, Avro, ORC, and CSV.
- Enables data querying through JDBC/ODBC connectors from external BI tools (e.g., Tableau, Power BI) and RDBMSs (e.g., MySQL, PostgreSQL).
- Offers a programmatic interface for interacting with structured data, supports ANSI SQL:2003 commands, HiveQL, and includes an interactive shell for SQL queries.

Spark SQL connectors and data sources



Using Spark SQL in Spark Applications

- **SparkSession Introduction:** Introduced in Spark 2.0 as a unified entry point for leveraging Spark's Structured APIs.
- **Usage:** Import the SparkSession class, create an instance, and utilize this instance to access all Spark functionalities.
- **SQL Queries:** Execute SQL queries using the sql() method on the SparkSession instance, e.g., spark.sql("SELECT * FROM myTableName").
- **DataFrames:** Queries return DataFrames, allowing for further operations and data manipulations as discussed in previous and upcoming chapters.

SQL Tables and Views

- **Table Metadata in Spark:** Each table in Spark is associated with metadata such as schema, table name, column names, partitions, and physical data location.
- **Central Metastore:** All table metadata is stored in a central metastore, with Spark using the Apache Hive metastore by default.
- **Default Metastore Location:** The default location for the Hive metastore is at /user/hive/warehouse.
- **Customizing Metastore Location:** Users can change the default metastore location by adjusting the spark.sql.warehouse.dir in Spark’s configuration to a new local or distributed storage path.

Managed Versus UnmanagedTables

- **Types of Tables in Spark:** Spark supports two types of tables, managed and unmanaged.
- **Managed Tables:** For managed tables, Spark manages both the metadata and the actual data, which can reside on local filesystems, HDFS, or cloud storage like Amazon S3 or Azure Blob.
- **Unmanaged Tables:** In contrast, unmanaged tables have their metadata managed by Spark, but the actual data is stored and managed externally (e.g., in Cassandra).
- **Data and Metadata Management:** Dropping a managed table via SQL (DROP TABLE table_name) removes both its metadata and data, whereas the same command for an unmanaged table only removes the metadata.

Creating SQL Databases and Tables

Tables reside within a database. By default, Spark creates tables under the `default` database. To create your own database name, you can issue a SQL command from your Spark application or notebook. Using the US flight delays data set, let's create both a managed and an unmanaged table. To begin, we'll create a database called `learn_spark_db` and tell Spark we want to use that database:

```
// In Scala/Python
spark.sql("CREATE DATABASE learn_spark_db")
spark.sql("USE learn_spark_db")
```

From this point, any commands we issue in our application to create tables will result in the tables being created in this database and residing under the database name `learn_spark_db`.

Creating a managed table

To create a managed table within the database `learn_spark_db`, you can issue a SQL query like the following:

```
// In Scala/Python
spark.sql("CREATE TABLE managed_us_delay_flights_tbl (date STRING, delay INT,
  distance INT, origin STRING, destination STRING)")
```

You can do the same thing using the DataFrame API like this:

```
# In Python
# Path to our US flight delays CSV file
csv_file = "/databricks-datasets/learning-spark-v2/flights/departuredelays.csv"
# Schema as defined in the preceding example
schema="date STRING, delay INT, distance INT, origin STRING, destination STRING"
flights_df = spark.read.csv(csv_file, schema=schema)
flights_df.write.saveAsTable("managed_us_delay_flights_tbl")
```

Both of these statements will create the managed table `us_delay_flights_tbl` in the `learn_spark_db` database.

Creating an unmanaged table

By contrast, you can create unmanaged tables from your own data sources—say, Parquet, CSV, or JSON files stored in a file store accessible to your Spark application.

To create an unmanaged table from a data source such as a CSV file, in SQL use:

```
spark.sql("""CREATE TABLE us_delay_flights_tbl(date STRING, delay INT,
  distance INT, origin STRING, destination STRING)
  USING csv OPTIONS (PATH
  '/databricks-datasets/learning-spark-v2/flights/departuredelays.csv')""")
```

And within the DataFrame API use:

```
(flights_df
  .write
  .option("path", "/tmp/data/us_flights_delay")
  .saveAsTable("us_delay_flights_tbl"))
```

DataFrameReader

- ❖ **DataFrameReader Overview:** The DataFrameReader is essential for reading data into a DataFrame, accessible only through a SparkSession instance using SparkSession.read for static data sources or SparkSession.readStream for streaming sources.
- ❖ **Usage Pattern:** Employ a chain of method calls such as DataFrameReader.format(args).option("key", "value").schema(args).load() to configure and execute data read operations, enhancing readability and consistency with Spark's API style.
- ❖ **Method Options:** Each method on DataFrameReader can take various arguments to adjust the reading process, with specific options for formatting, options, schema settings, and data source.
- ❖ **Streaming and Static Data:** read is used for static data sources, while readStream is tailored for streaming data inputs, part of Spark’s Structured Streaming capabilities to be discussed later.

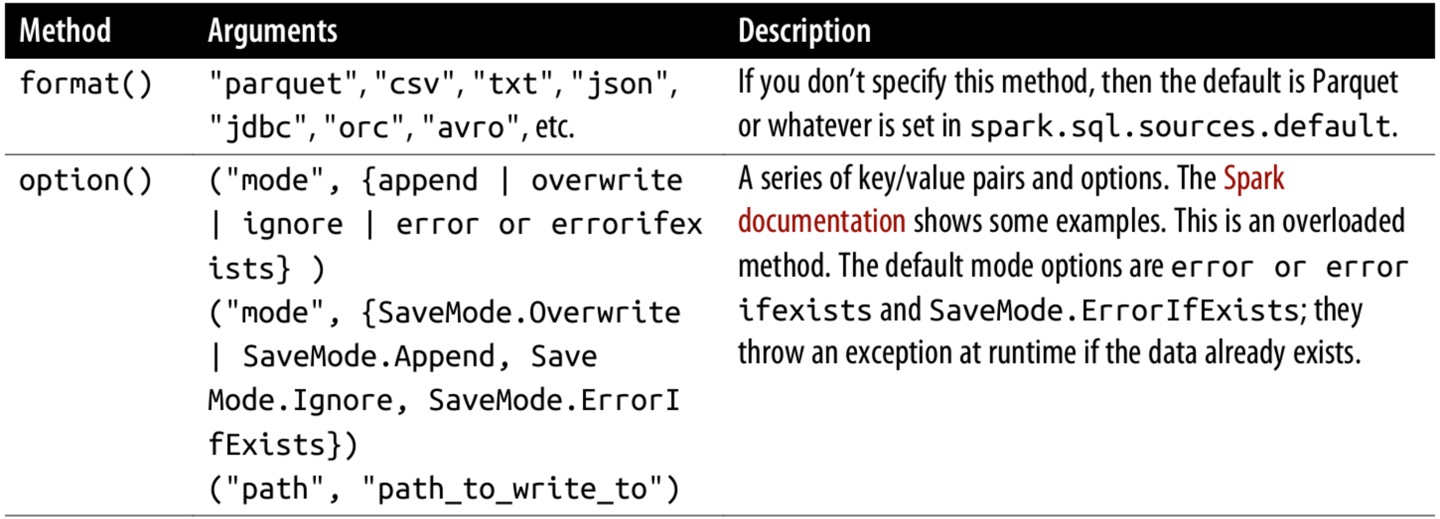
DataFrameReader methods, arguments, and options

Method	Arguments	Description
`format()`	`"parquet"`, `"csv"`, `"txt"`, `"json"`, `"jdbc"`, `"orc"`, `"avro"`, etc.	If you don't specify this method, then the default is Parquet or whatever is set in `spark.sql.sources.default`.
`option()`	`("mode", {PERMISSIVE | FAILFAST | DROPMALFORMED } )` `("inferSchema", {true | false})` `("path", "path_file_data_source")`	A series of key/value pairs and options. The Spark documentation shows some examples and explains the different modes and their actions. The default mode is `PERMISSIVE`. The `"inferSchema"` and `"mode"` options are specific to the JSON and CSV file formats.
`schema()`	DDL `String` or `StructType`, e.g., `'A INT, B STRING'` or `StructType(...)`	For JSON or CSV format, you can specify to infer the schema in the `option()` method. Generally, providing a schema for any format makes loading faster and ensures your data conforms to the expected schema.
`load()`	`"/path/to/data/source"`	The path to the data source. This can be empty if specified in `option("path", "...")`.

DataFrameWriter

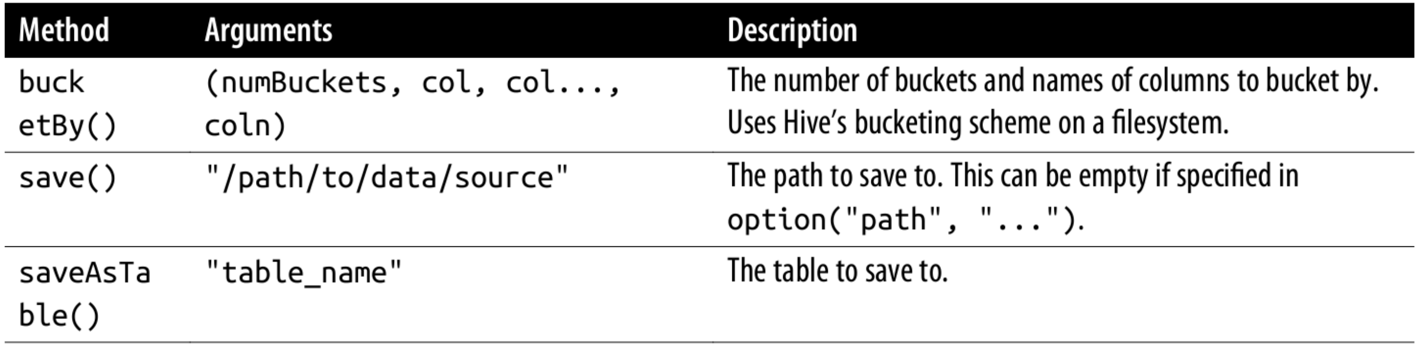
- **DataFrameWriter Functionality:** Contrary to DataFrameReader, DataFrameWriter is used for saving or writing data from a DataFrame to a specified built-in data source.
- **Accessing DataFrameWriter:** It is accessed not from a SparkSession, but directly from the DataFrame that needs to be saved using DataFrame.write for batch processes or DataFrame.writeStream for streaming processes.
- **Usage Patterns:** Common usage includes chaining methods like DataFrameWriter.format(args).option(args).partitionBy(args).save(path) for writing to files, or DataFrameWriter.format(args).option(args).sortBy(args).saveAsTable(table) for saving as a table.
- **Customization Options:** Methods such as format, option, bucketBy, partitionBy, and sortBy allow for detailed configuration of how data is written, including formatting, partitioning, and sorting data before saving.

DataFrameWriter methods, arguments, and



Method	Arguments	Description
`format()`	`"parquet"`, `"csv"`, `"txt"`, `"json"`, `"jdbc"`, `"orc"`, `"avro"`, etc.	If you don't specify this method, then the default is Parquet or whatever is set in `spark.sql.sources.default`.

`option()`	`("mode", {append | overwrite | ignore | error or errorifexists} )`
	`("mode", {SaveMode.Overwrite | SaveMode.Append, SaveMode.Ignore, SaveMode.ErrorIfExists})`
	`("path", "path_to_write_to")`
		A series of key/value pairs and options. The Spark documentation shows some examples. This is an overloaded method. The default mode options are `error or errorifexists` and `SaveMode.ErrorIfExists`; they throw an exception at runtime if the data already exists.



Method	Arguments	Description
`bucketBy()`	`(numBuckets, col, col..., coln)`	The number of buckets and names of columns to bucket by. Uses Hive's bucketing scheme on a filesystem.
`save()`	`"/path/to/data/source"`	The path to save to. This can be empty if specified in `option("path", "...")`.
`saveAsTable()`	`"table_name"`	The table to save to.

Parquet

- **Preference for Parquet:** Parquet is the default data source in Spark, recommended due to its efficiency and widespread support across big data frameworks.
- **Optimizations and Benefits:** As an open-source columnar file format, Parquet includes optimizations like compression, reducing storage requirements and enhancing access speed to column data.
- **Usage Recommendation:** After transforming and cleansing data, it's advised to save DataFrames in Parquet format to facilitate efficient downstream consumption.
- **Starting Point in Data Exploration:** The exploration of data sources in Spark often begins with Parquet due to its integral role and benefits in data processing.

Reading Parquet files into a DataFrame

Parquet files are stored in a directory structure that contains the data files, metadata, a number of compressed files, and some status files. Metadata in the footer contains the version of the file format, the schema, and column data such as the path, etc.

For example, a directory in a Parquet file might contain a set of files like this:

```
_SUCCESS
_committed_1799640464332036264
_started_1799640464332036264
part-00000-tid-1799640464332036264-91273258-d7ef-4dc7-<...>-c000.snappy.parquet
```

There may be a number of *part-XXXX* compressed files in a directory (the names shown here have been shortened to fit on the page).

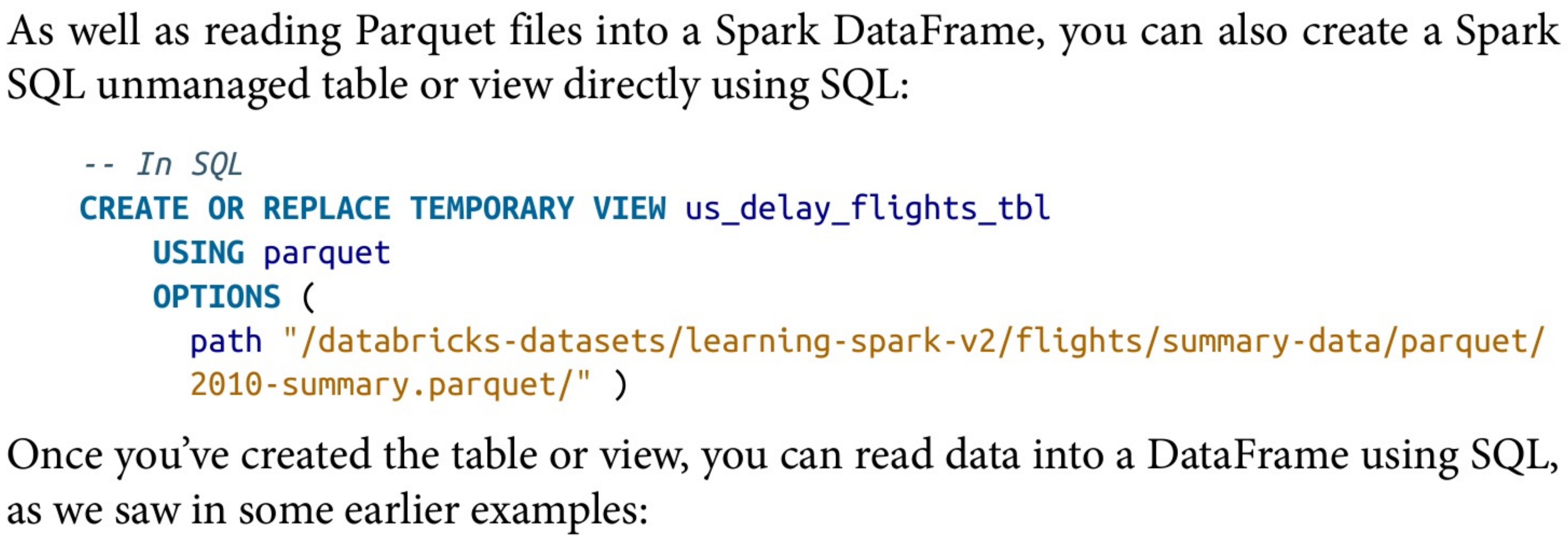
Reading Parquet files into a DataFrame

```
// In Scala
val file = """/databricks-datasets/learning-spark-v2/flights/summary-data/
  parquet/2010-summary.parquet/"""
val df = spark.read.format("parquet").load(file)
```

```
# In Python
file = """/databricks-datasets/learning-spark-v2/flights/summary-data/parquet/
  2010-summary.parquet/"""
df = spark.read.format("parquet").load(file)
```

Unless you are reading from a streaming data source there's no need to supply the schema, because Parquet saves it as part of its metadata.

Reading Parquet files into a Spark SQL table



As well as reading Parquet files into a Spark DataFrame, you can also create a Spark SQL unmanaged table or view directly using SQL:

```
-- In SQL
CREATE OR REPLACE TEMPORARY VIEW us_delay_flights_tbl
    USING parquet
    OPTIONS (
      path "/databricks-datasets/learning-spark-v2/flights/summary-data/parquet/
      2010-summary.parquet/" )
```

Once you've created the table or view, you can read data into a DataFrame using SQL, as we saw in some earlier examples:

Reading Parquet files into a Spark SQL table

```
// In Scala
spark.sql("SELECT * FROM us_delay_flights_tbl").show()

# In Python
spark.sql("SELECT * FROM us_delay_flights_tbl").show()
```

Both of these operations return the same results:

```
+-----------------+-------------------+-----+
|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count|
+-----------------+-------------------+-----+
|United States    |Romania            |1    |
|United States    |Ireland            |264  |
|United States    |India              |69   |
|Egypt            |United States      |24   |
|Equatorial Guinea|United States      |1    |
|United States    |Singapore          |25   |
|United States    |Grenada            |54   |
|Costa Rica       |United States      |477  |
|Senegal          |United States      |29   |
|United States    |Marshall Islands   |44   |
+-----------------+-------------------+-----+
only showing top 10 rows
```

Writing DataFrames to Parquet files

Writing or saving a DataFrame as a table or file is a common operation in Spark. To write a DataFrame you simply use the methods and arguments to the `DataFrameWriter` outlined earlier in this chapter, supplying the location to save the Parquet files to. For example:

```
// In Scala
df.write.format("parquet")
  .mode("overwrite")
  .option("compression", "snappy")
  .save("/tmp/data/parquet/df_parquet")
```

```
# In Python
(df.write.format("parquet")
  .mode("overwrite")
  .option("compression", "snappy")
  .save("/tmp/data/parquet/df_parquet"))
```

Writing DataFrames to Parquet files

Writing or saving a DataFrame as a table or file is a common operation in Spark. To write a DataFrame you simply use the methods and arguments to the `DataFrameWriter` outlined earlier in this chapter, supplying the location to save the Parquet files to. For example:

```
// In Scala
df.write.format("parquet")
  .mode("overwrite")
  .option("compression", "snappy")
  .save("/tmp/data/parquet/df_parquet")
```

```
# In Python
(df.write.format("parquet")
  .mode("overwrite")
  .option("compression", "snappy")
  .save("/tmp/data/parquet/df_parquet"))
```

JSON

- **Popularity of JSON:** JavaScript Object Notation (JSON) is a widely used data format known for its simplicity and ease of parsing compared to XML.
- **Representational Formats in JSON:** JSON supports two formats in Spark: single-line mode, where each line represents a single JSON object, and multiline mode, where an entire multiline object is treated as a single JSON object.
- **Support in Spark:** Both single-line and multiline modes are supported by Spark, accommodating diverse data structuring needs.
- **Reading Multiline JSON:** To read JSON in multiline mode in Spark, set the multiLine option to true using the option() method when reading data.

Reading a JSON file into a DataFrame

You can read a JSON file into a DataFrame the same way you did with Parquet—just specify `"json"` in the `format()` method:

```
// In Scala
val file = "/databricks-datasets/learning-spark-v2/flights/summary-data/json/*"
val df = spark.read.format("json").load(file)
```

```
# In Python
file = "/databricks-datasets/learning-spark-v2/flights/summary-data/json/*"
df = spark.read.format("json").load(file)
```

Reading a JSON file into a Spark SQL table

You can also create a SQL table from a JSON file just like you did with Parquet:

```
-- In SQL
CREATE OR REPLACE TEMPORARY VIEW us_delay_flights_tbl
    USING json
    OPTIONS (
      path  "/databricks-datasets/learning-spark-v2/flights/summary-data/json/*"
    )
```

Once the table is created, you can read data into a DataFrame using SQL:

```
// In Scala/Python
spark.sql("SELECT * FROM us_delay_flights_tbl").show()

+-----------------+-------------------+-----+
|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count|
+-----------------+-------------------+-----+
|United States    |Romania            |15   |
|United States    |Croatia            |1    |
|United States    |Ireland            |344  |
|Egypt            |United States      |15   |
```

Writing DataFrames to JSON files

Saving a DataFrame as a JSON file is simple. Specify the appropriate `DataFrameWriter` methods and arguments, and supply the location to save the JSON files to:

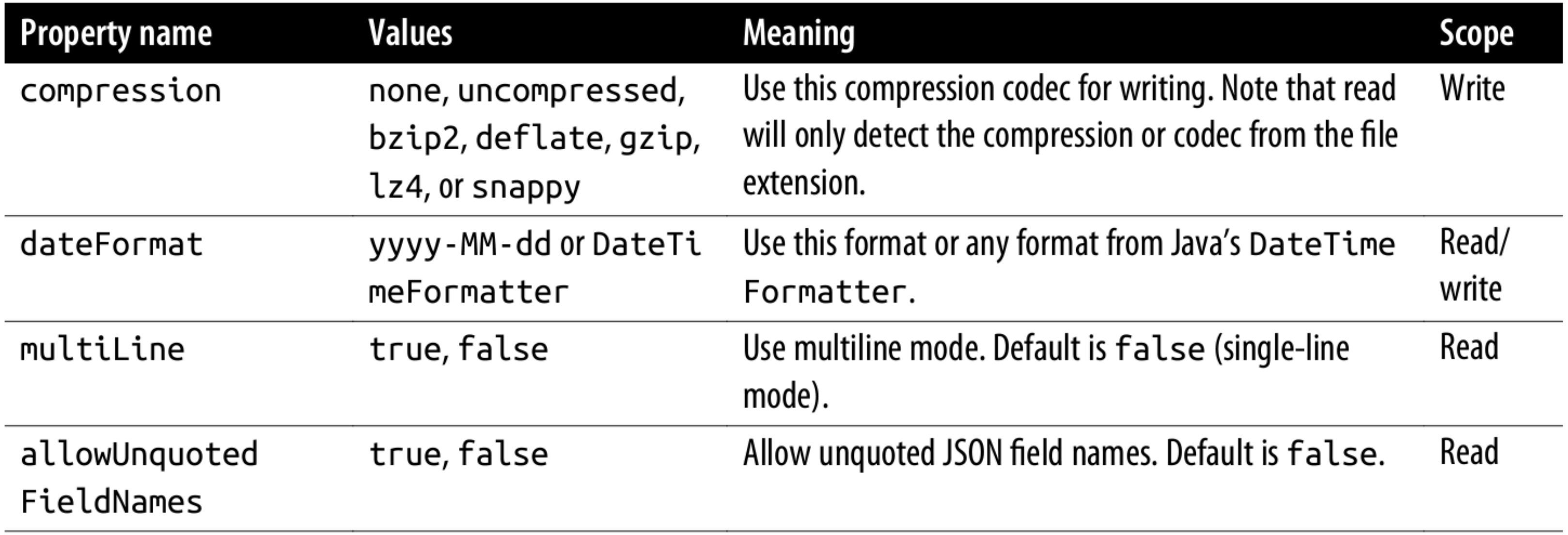
```
// In Scala
df.write.format("json")
  .mode("overwrite")
  .option("compression", "snappy")
  .save("/tmp/data/json/df_json")

# In Python
(df.write.format("json")
  .mode("overwrite")
  .option("compression", "snappy")
  .save("/tmp/data/json/df_json"))
```

This creates a directory at the specified path populated with a set of compact JSON files:

```
-rw-r--r--  1 jules  wheel   0 May 16 14:44 _SUCCESS
-rw-r--r--  1 jules  wheel  71 May 16 14:44 part-00000-<...>-c000.json
```

JSON options for DataFrameReader and DataFrameWriter



Property name	Values	Meaning	Scope
`compression`	`none`, `uncompressed`, `bzip2`, `deflate`, `gzip`, `lz4`, or `snappy`	Use this compression codec for writing. Note that read will only detect the compression or codec from the file extension.	Write
`dateFormat`	`yyyy-MM-dd` or `DateTimeFormatter`	Use this format or any format from Java's `DateTimeFormatter`.	Read/ write
`multiLine`	`true`, `false`	Use multiline mode. Default is `false` (single-line mode).	Read
`allowUnquotedFieldNames`	`true`, `false`	Allow unquoted JSON field names. Default is `false`.	Read

Reading a CSV file into a DataFrame

As with the other built-in data sources, you can use the `DataFrameReader` methods and arguments to read a CSV file into a DataFrame:

```
// In Scala
val file = "/databricks-datasets/learning-spark-v2/flights/summary-data/csv/*"
val schema = "DEST_COUNTRY_NAME STRING, ORIGIN_COUNTRY_NAME STRING, count INT"

val df = spark.read.format("csv")
  .schema(schema)
  .option("header", "true")
  .option("mode", "FAILFAST")     // Exit if any errors
  .option("nullValue", "")        // Replace any null data with quotes
  .load(file)

# In Python
file = "/databricks-datasets/learning-spark-v2/flights/summary-data/csv/*"
schema = "DEST_COUNTRY_NAME STRING, ORIGIN_COUNTRY_NAME STRING, count INT"
df = (spark.read.format("csv")
  .option("header", "true")
  .schema(schema)
  .option("mode", "FAILFAST")  # Exit if any errors
  .option("nullValue", "")     # Replace any null data field with quotes
  .load(file))
```

Writing DataFrames to CSV files

Saving a DataFrame as a CSV file is simple. Specify the appropriate `DataFrameWriter` methods and arguments, and supply the location to save the CSV files to:

```
// In Scala
df.write.format("csv").mode("overwrite").save("/tmp/data/csv/df_csv")

# In Python
df.write.format("csv").mode("overwrite").save("/tmp/data/csv/df_csv")
```

This generates a folder at the specified location, populated with a bunch of compressed and compact files:

```
-rw-r--r--  1 jules  wheel   0 May 16 12:17 _SUCCESS
-rw-r--r--  1 jules  wheel  36 May 16 12:17 part-00000-251690eb-<...>-c000.csv
```

CSV options for DataFrameReader and DataFrameWriter

Property name	Values	Meaning	Scope
`compression`	`none`, `bzip2`, `deflate`, `gzip`, `lz4`, or `snappy`	Use this compression codec for writing.	Write
`dateFormat`	`yyyy-MM-dd` or `DateTimeFormatter`	Use this format or any format from Java's `DateTimeFormatter`.	Read/write
`multiLine`	`true`, `false`	Use multiline mode. Default is `false` (single-line mode).	Read
`inferSchema`	`true`, `false`	If `true`, Spark will determine the column data types. Default is `false`.	Read
`sep`	Any character	Use this character to separate column values in a row. Default delimiter is a comma (`,`).	Read/write
`escape`	Any character	Use this character to escape quotes. Default is `\`.	Read/write
`header`	`true`, `false`	Indicates whether the first line is a header denoting each column name. Default is `false`.	Read/write

SEAS 8515

Next Class:

Spark SQL and DataFrames: Interacting with External Data Sources, Introduction to Webscraping

School of Engineering & Applied Science

THE GEORGE WASHINGTON UNIVERSITY

Dr. Adewale Akinfaderin