Built-in Data Sources

Lesson Objectives

- After completing this lesson, you should be able to:
  - Understand Spark's built-in Data Sources
  - Use generic Load and Save functions
  - Specify options manually
  - Load and save JSON datasets
  - Load and save Parquet files
  - Load data from a database using JDBC

Data Sources

- The DataFrame interface allows Spark SQL to use various data sources
- Spark Data Sources provides general methods for loading and saving data from different sources:
  - JSON, Parquet, JDBC
- Unless specified otherwise, the default data source (Parquet) will be used
- Sample files used in the lesson can be found in Spark's `/examples/src/main/resources` folder
  - users.parquet
  - people.json

Generic Load/Save Functions

```
df = sqlc.read.load(examples_folder + "users.parquet")
df.show()
```

```
+------+--------------+----------------+
|  name|favorite_color|favorite_numbers|
+------+--------------+----------------+
|Alyssa|          null|  [3, 9, 15, 20]|
|   Ben|           red|              []|
+------+--------------+----------------+
```

```
!rm -rf namesAndFavColors.parquet/
df.select("name", "favorite_color").write.save("namesAndFavColors.parquet")
```

```
!ls -l namesAndFavColors.parquet/
```

```
total 4
-rw------- 1 dvgodoy dvgodoy 509 Set 25 17:04 part-r-00000-876c04e5-04ce-44a9-a451-ec2074838894.snappy.parquet
-rw------- 1 dvgodoy dvgodoy   0 Set 25 17:04 _SUCCESS
```

Manually Specifying Options

- You can specify the data source to be used to load/save data
- Data sources are specified by their fully qualified name

```
sqlContext.read.format("org.apache.spark.sql.parquet").
load("users.parquet")
```

- Built-in sources have short names (parquet, json, jdbc)

```
sqlContext.read.format("parquet").load("users.parquet")
```

- These sources also have a corresponding method

```
sqlContext.read.parquet("users.parquet")
```

- DataFrames of any type can be converted into other types using this syntax

Manually Specifying Options

```
sqlc.read.format("org.apache.spark.sql.parquet").load("users.parquet")
```

```
DataFrame[name: string, favorite_color: string, favorite_numbers: array<int>]
```

```
sqlc.read.format("parquet").load("users.parquet")
```

```
DataFrame[name: string, favorite_color: string, favorite_numbers: array<int>]
```

```
sqlc.read.parquet("users.parquet")
```

```
DataFrame[name: string, favorite_color: string, favorite_numbers: array<int>]
```

JSON

- Spark SQL can infer the schema of a JSON dataset and load it as a DataFrame
- The JSON dataset can be either an RDD of Strings or a JSON file
- If a JSON file is used, each line must contain a separate valid JSON object
- A DataFrame can be converted to an RDD of JSON Strings using the method `toJSON`

JSON Example (1)

```
df_json = sqlc.read.format("json").load(examples_folder + "people.json")
df_json.show()
```

```
+----+-------+
| age|   name|
+----+-------+
|null|Michael|
|  30|   Andy|
|  19| Justin|
+----+-------+
```

JSON Example (2)

```
df_json.printSchema()
```

```
root
 |-- age: long (nullable = true)
 |-- name: string (nullable = true)
```

```
json_strings = df_json.toJSON()
json_strings.collect()
```

```
[u'{"name":"Michael"}',
 u'{"age":30,"name":"Andy"}',
 u'{"age":19,"name":"Justin"}']
```

Parquet

- Columnar format supported by several data processing systems
- Spark SQL automatically preserves the schema of the original data
- All columns are converted to nullable for compatibility reasons

Parquet Example

```
!rm -rf namesAndAges.parquet/
df_json.select("name", "age").write.format("parquet").save("namesAndAges.parquet")
```

```
!ls -l namesAndAges.parquet
```

```
total 1
-rw------- 1 dvgodoy dvgodoy 522 Set 25 17:09 part-r-00000-931a292f-ba66-4df2-8f10-87a8abb42289.snappy.parquet
-rw------- 1 dvgodoy dvgodoy   0 Set 25 17:09 _SUCCESS
```

```
!rm -rf namesAndAgesCoalesced.parquet/
df_json.select("name", "age").coalesce(1).write.format("parquet").save("namesAndAgesCoalesced.parquet")
```

```
!ls -l namesAndAgesCoalesced.parquet
```

```
total 1
-rw------- 1 dvgodoy dvgodoy 522 Set 25 17:10 part-r-00000-038eb1d1-1907-4923-a0ea-d60dc1401151.snappy.parquet
-rw------- 1 dvgodoy dvgodoy   0 Set 25 17:10 _SUCCESS
```

Save Modes

- Save operation can take a `SaveMode` parameter that specifies how to handle existing data
- These modes are not atomic and do not use any kind of locking
- Actions of each `SaveMode`, if data already exists:
  - "`error`": throws an exception
  - "`append`": appends DataFrame contents to it
  - "`overwrite`": deletes original data and writes new data
  - "`ignore`": do not change original data nor save new data

Save Modes Example

```
df_json.write.mode("error").parquet("namesAndAges.parquet")
# This IS supposed to throw an exception, as it is trying to write an already existent file!
```

```
---------------------------------------------------------------------------
AnalysisException                         Traceback (most recent call last)
<ipython-input-30-5d4993300a51> in <module>()
----> 1 df_json.write.mode("error").parquet("namesAndAges.parquet")
```

```
df_json.write.mode("overwrite").parquet("namesAndAges.parquet")
```

SQL on Files

•Spark SQL allows files to be queried directly without being first loaded into DataFrames

```
df = sqlc.sql("SELECT * FROM json.`people.json` WHERE age = 19")
df.show()
```

```
+---+------+
|age|  name|
+---+------+
| 19|Justin|
+---+------+
```

JDBC

- Data source to read from databases into a DataFrame using JDBC
- MySQL Connector/J:
  - `https://dev.mysql.com/downloads/connector/j/5.0.html`
- PostgreSQL Connector:
  - `https://jdbc.postgresql.org/download.html`
- Include the JDBC driver for a given database on Spark classpath or add it to the spark.defaults.conf with spark.jars

Options

Option	Description
url	JDBC URL to connect to
driver	class name of the JDBC driver to use to connect to the URL
dbtable	JDBC table to be read (subqueries in parentheses are also valid)
partitionColumn, lowerBound, upperBound, numPartitions	if any of these are specified, ALL must be specified; they describe how to partition the table when reading in parallel from multiple workers. lowerBound and upperBound are used to decide the partition stride only, not for filtering the rows
partitionColumn	must be a numeric column from the table in question
fetchSize	JDBC fetch size (how many rows to fetch per round trip)

Example – Table and Data

- For this example, it is assumed the existence of a “test” database with a "users" table containing 3 rows

```
CREATE TABLE users (user_id int PRIMARY KEY, fname text,
lname text);
INSERT INTO users (user_id,  fname, lname) VALUES (1,
'john', 'smith');
INSERT INTO users (user_id,  fname, lname) VALUES (2,
'john', 'doe');
INSERT INTO users (user_id,  fname, lname) VALUES (3,
'john', 'smith');
```

Example (1)

```
df_mysql = sqlc.read.format("jdbc") \
                .option("url", "jdbc:mysql://localhost:3306/test") \
                .option("driver", "com.mysql.jdbc.Driver") \
                .option("dbtable", "users") \
                .option("user", "root") \
                .option("password", "").load()
```

```
df_mysql.printSchema()
```

```
root
 |-- user_id: integer (nullable = false)
 |-- fname: string (nullable = true)
 |-- lname: string (nullable = true)
```

Example (2)

```
df_mysql.registerTempTable("users")
```

```
sqlc.sql("select * from users").collect()
```

```
[Row(user_id=1744, fname=u'john', lname=u'doe'),
 Row(user_id=1745, fname=u'john', lname=u'smith'),
 Row(user_id=1746, fname=u'john', lname=u'smith')]
```

Lesson Summary

- Having completed this lesson, you should be able to:
  - Understand Spark's built-in Data Sources
  - Use generic Load and Save functions
  - Specify options manually
  - Load and save JSON datasets
  - Load and save Parquet files
  - Load data from a database using JDBC