O'REILLY®

Wydanie II

Spark

Błyskawiczna analiza danych

Jules S. Damji
Brooke Wenig
Tathagata Das
Denny Lee

Helion

Tytuł oryginału: Learning Spark: Lightning-Fast Data Analytics, 2nd Edition

Tłumaczenie: Robert Górczyński

ISBN: 978-83-283-9914-3

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *https://helion.pl* (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
https://helion.pl/user/opinie/sparb2
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

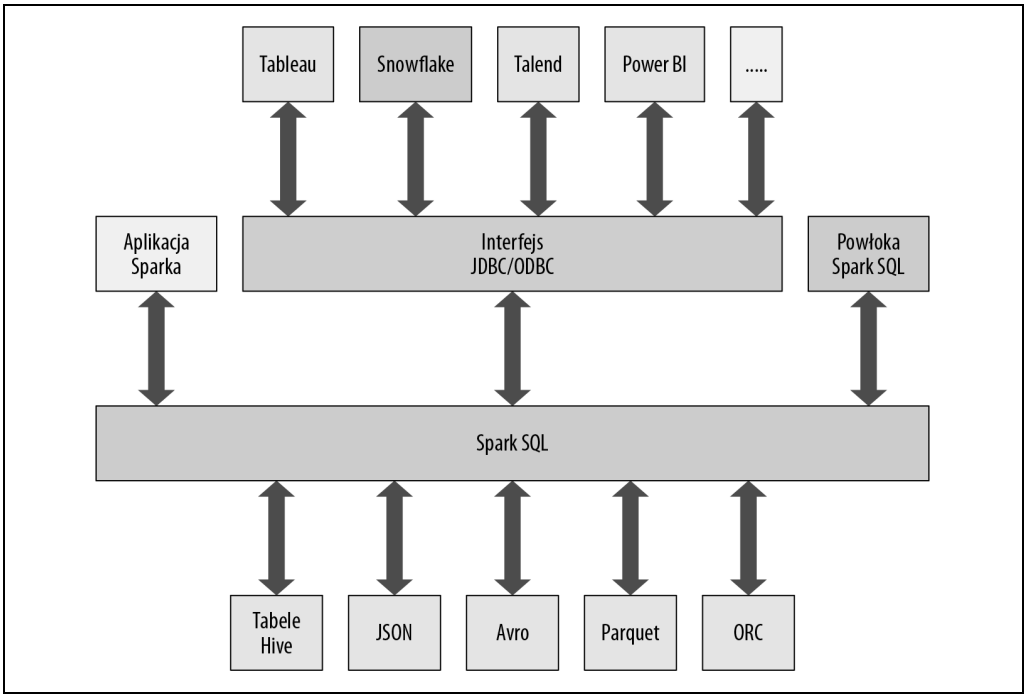
Spis treści

ROZDZIAŁ 4.

Spark SQL i DataFrame — wprowadzenie do wbudowanych źródeł danych

W poprzednim rozdziale omówiliśmy ewolucję struktury w Sparku i uzasadniliśmy jej istnienie. Przede wszystkim wyjaśniliśmy, w jaki sposób silnik Spark SQL zapewnia ujednolicone podstawy, na których opierają się wysokiego poziomu API DataFrame i Dataset. W tej części książki będziemy kontynuować omawianie DataFrame i pokażemy, na jakiej zasadzie wspomniane API może współpracować z silnikiem Spark SQL.

Ten i następny rozdział wyjaśnią, jak Spark SQL współdziała z wybranymi komponentami zewnętrznymi, które zostały pokazane na rysunku 4.1.



Rysunek 4.1. Interfejsy i źródła danych Spark SQL

Spark SQL przede wszystkim:

- Zapewnia silnik, na bazie którego zostało opracowane API strukturalne, dokładniej omówione w poprzednim rozdziale.
- Może odczytywać i zapisywać dane w wielu formatach strukturalnych (m.in. JSON, tabele Hive, Parquet, Avro, ORC, CSV).
- Pozwala wykonywać zapytania dotyczące danych, używając w tym celu interfejsów JDBC/ODBC z zewnętrznych źródeł danych, takich jak Tableau, Power BI czy Talend, bądź z systemów RDBMS, np. MySQL i PostgreSQL.
- Oferuje interfejs programistyczny przeznaczony do pracy (z poziomu aplikacji Sparka) ze strukturalnymi danymi, które są przechowywane jako tabele lub widoki w bazie danych.
- Zapewnia interaktywną powłokę, pozwalającą na wykonywanie zapytań SQL dotyczących strukturalnych danych.
- Obsługuje polecenia zgodne ze standardem ANSI SQL:2003 (*https://en.wikipedia.org/wiki/SQL:2003*) i HiveQL (*https://spark.apache.org/docs/latest/sql-data-sources-hive-tables.html*).

Na początku wyjaśnimy, jak można używać Spark SQL w aplikacji Sparka.

Używanie Spark SQL w aplikacji Sparka

Wprowadzony w Sparku 2.0 egzemplarz `SparkSession` zapewnia ujednolicony punkt wyjścia (*https://databricks.com/blog/2016/08/15/how-to-use-sparksession-in-apache-spark-2-0.html*) do programowania w Sparku z wykorzystaniem API strukturalnego. Egzemplarz ten można wykorzystać w celu uzyskania dostępu do funkcjonalności frameworka: wystarczy zaimportować klasę, a następnie utworzyć egzemplarz `SparkSession` w kodzie.

W celu wysłania dowolnego zapytania SQL należy użyć metody `sql()` egzemplarza `SparkSession`, `spark`, np. `spark.sql("SELECT * FROM nazwaTabeli")`. Wszystkie zapytania `spark.sql` wykonywane w taki sposób zwracają egzemplarz DataFrame, na którym można później przeprowadzać kolejne operacje Sparka — jak choćby te omówione w poprzednim rozdziale bądź też poznane w tym i następnym rozdziale.

Przykłady podstawowych zapytań

W tym punkcie zamierzamy omówić kilka przykładów zapytań dotyczących zbioru danych *Airline On-Time Performance and Causes of Flight Delays* (*https://catalog.data.gov/dataset/airline-on-time-performance-and-causes-of-flight-delays*), który zawiera informacje o lotach krajowych w USA, takie jak data lotu, opóźnienie, odległość, lotniska początkowe i docelowe. Dane te są dostępne w postaci pliku w formacie CSV zawierającego ponad milion rekordów. Za pomocą schematu dane zostaną wczytane do egzemplarza DataFrame, który następnie zostanie zarejestrowany jako widok tymczasowy (wkrótce dowiesz się więcej na temat widoków tymczasowych), aby można było wysyłać do niego zapytania SQL.

Przykładowe zapytania przedstawiliśmy w postaci fragmentów kodu oraz notatników Pythona i Scali zawierających cały kod omawiany w tym rozdziale, które znajdziesz w archiwum materiałów do tej książki (*ftp://ftp.helion.pl/przyklady/sparb2.zip*). Dzięki tym przykładom możesz zobaczyć, jak używać zapytań SQL w aplikacjach Sparka za pomocą interfejsu programistycznego `spark.sql` (*https://spark.apache.org/sql/*). Podobnie jak w przypadku API DataFrame w jego deklaratywnej postaci także wymieniony interfejs pozwala wykonywać zapytania dotyczące strukturalnych danych z poziomu aplikacji Sparka.

W niezależnej aplikacji Sparka egzemplarz `SparkSession` zazwyczaj tworzy się ręcznie, jak pokazaliśmy w kolejnym fragmencie kodu. Natomiast w powłoce Sparka (lub notatniku Databrick) egzemplarz `SparkSession` tworzony jest automatycznie i można uzyskać do niego dostęp za pomocą zmiennej o pasującej nazwie: `spark`.

Rozpoczynamy od wczytania zbioru danych do widoku tymczasowego.

```
// W kodzie Scali.
import org.apache.spark.sql.SparkSession
val spark = SparkSession
  .builder
  .appName("SparkSQLExampleApp")
  .getOrCreate()

// Ścieżka dostępu do zbioru danych.
val csvFile="/databricks-datasets/learning-spark-v2/flights/departuredelays.csv"

// Odczytanie danych i utworzenie widoku tymczasowego.
// Ustalenie schematu (w przypadku ogromnych plików prawdopodobnie zechcesz samodzielnie podać schemat).
val df = spark.read.format("csv")
  .option("inferSchema", "true")
  .option("header", "true")
  .load(csvFile)
// Utworzenie widoku tymczasowego.
df.createOrReplaceTempView("us_delay_flights_tbl")

# W kodzie Pythona.
from pyspark.sql import SparkSession
# Utworzenie egzemplarza SparkSession.
spark = (SparkSession
  .builder
  .appName("SparkSQLExampleApp")
  .getOrCreate())

# Ścieżka dostępu do zbioru danych.
csv_file = "/databricks-datasets/learning-spark-v2/flights/departuredelays.csv"

# Odczytanie danych i utworzenie widoku tymczasowego.
# Ustalenie schematu (w przypadku ogromnych plików
# prawdopodobnie zechcesz samodzielnie podać schemat).
df = (spark.read.format("csv")
  .option("inferSchema", "true")
  .option("header", "true")
  .load(csv_file))
df.createOrReplaceTempView("us_delay_flights_tbl")
```

Jeżeli chcesz podać schemat, skorzystaj z ciągu tekstowego sformatowanego jako DDL. Spójrz na przedstawiony tutaj przykład.

```
// W kodzie Scali.
val schema = "date STRING, delay INT, distance INT, origin STRING
➥destination STRING"

# W kodzie Pythona.
schema = "`date` STRING, `delay` INT, `distance` INT, `origin` STRING,
➥`destination` STRING"
```

Kod powoduje utworzenie widoku tymczasowego, co pozwala wykonywać zapytania SQL za pomocą silnika Spark SQL. Zapytania te nie różnią się od tych wydawanych względem tabeli SQL, np. w bazie danych MySQL bądź PostgreSQL. Naszym celem jest tutaj pokazanie, że Spark SQL oferuje interfejs SQL zgodny z ANSI:2003, oraz zademonstrowanie możliwości współdziałania między SQL i DataFrame.

Dane dotyczące opóźnień lotów w USA składają się z pięciu kolumn:

- Kolumna `date` zawiera ciąg tekstowy typu `02190925`. Po konwersji wartość ta jest mapowana na `02-19 09:25 am`.
- Kolumna `delay` podaje wyrażone w minutach opóźnienie między planowaną i faktyczną godziną odlotu. W przypadku wcześniejszego odlotu wartość w tej kolumnie jest ujemna.
- Kolumna `distance` podaje wyrażoną w milach odległość między początkowym i docelowym portem lotniczym.
- Kolumna `origin` zawiera kod IATA początkowego portu lotniczego.
- Kolumna `destination` zawiera kod IATA docelowego portu lotniczego.

Mając to wszystko na uwadze, wykonamy teraz przykładowe zapytania względem tych danych.

Zaczynamy od wyszukania wszystkich lotów na odległość większą niż 1000 mil.

```
spark.sql("""SELECT distance, origin, destination
FROM us_delay_flights_tbl WHERE distance > 1000
ORDER BY distance DESC""").show(10)

+--------+------+-----------+
|distance|origin|destination|
+--------+------+-----------+
|4330    |HNL   |JFK        |
|4330    |HNL   |JFK        |
|4330    |HNL   |JFK        |
|4330    |HNL   |JFK        |
|4330    |HNL   |JFK        |
|4330    |HNL   |JFK        |
|4330    |HNL   |JFK        |
|4330    |HNL   |JFK        |
|4330    |HNL   |JFK        |
|4330    |HNL   |JFK        |
+--------+------+-----------+
only showing top 10 rows
```

W efekcie otrzymujemy dane na temat najdłuższych rejsów, czyli tych między portami lotniczymi w Honolulu (HNL) i Nowym Jorku (JFK). W kolejnym przykładzie pobieramy dane dotyczące wszystkich rejsów między San Francisco (SFO) i Chicago (ORD), które miały opóźnienie powyżej 2 godzin.

```
spark.sql("""SELECT date, delay, origin, destination
FROM us_delay_flights_tbl
WHERE delay > 120 AND ORIGIN = 'SFO' AND DESTINATION = 'ORD'
ORDER by delay DESC""").show(10)

+--------+-----+------+-----------+
|date    |delay|origin|destination|
+--------+-----+------+-----------+
|02190925|1638 |SFO   |ORD        |
|01031755|396  |SFO   |ORD        |
|01022330|326  |SFO   |ORD        |
|01051205|320  |SFO   |ORD        |
|01190925|297  |SFO   |ORD        |
|02171115|296  |SFO   |ORD        |
|01071040|279  |SFO   |ORD        |
|01051550|274  |SFO   |ORD        |
|03120730|266  |SFO   |ORD        |
|01261104|258  |SFO   |ORD        |
+--------+-----+------+-----------+
only showing top 10 rows
```

Wygenerowane dane pokazują wiele dużych opóźnień dotyczących rejsów między wymienionymi miastami, występujących w różnych dniach. (W ramach ćwiczenia spróbuj skonwertować kolumnę `date` na format czytelny dla człowieka oraz odszukać dni lub miesiące, w których najczęściej występowały opóźnienia. Czy miały one miejsce w miesiącach zimowych, czy letnich?)

Przejdźmy do znacznie bardziej złożonego zapytania, w którym kod SQL zawiera klauzulę `CASE`. W kolejnym fragmencie kodu chcemy oznaczyć etykietami wszystkie loty w USA, niezależnie od portu początkowego i docelowego, oraz wskazać wielkość opóźnienia: ogromne (powyżej 6 godzin), duże (od 2 do 6 godzin) itd. W nowej kolumnie o nazwie `Flight_Delays` zostaną wyświetlone czytelne dla człowieka etykiety informujące o wielkości opóźnienia: `Very Long Delays` (pol. *bardzo duże opóźnienie*), `Long Delays` (pol. *duże opóźnienie*), `Short Delays` (pol. *krótkie opóźnienie*), `Tolerable Delays` (pol. *akceptowalne opóźnienie*) i `No Delays` (pol. *brak opóźnienia*).

```
spark.sql("""SELECT delay, origin, destination,
              CASE
                  WHEN delay > 360 THEN 'Very Long Delays'
                  WHEN delay >= 120 AND delay <= 360 THEN 'Long Delays'
                  WHEN delay >= 60 AND delay < 120 THEN 'Short Delays'
                  WHEN delay > 0 and delay < 60 THEN 'Tolerable Delays'
                  WHEN delay = 0 THEN 'No Delays'
                  ELSE 'Early'
               END AS Flight_Delays
               FROM us_delay_flights_tbl
               ORDER BY origin, delay DESC""").show(10)

+-----+------+-----------+-------------+
|delay|origin|destination|Flight_Delays|
+-----+------+-----------+-------------+
```

```
|333  |ABE   |ATL        |Long Delays  |
|305  |ABE   |ATL        |Long Delays  |
|275  |ABE   |ATL        |Long Delays  |
|257  |ABE   |ATL        |Long Delays  |
|247  |ABE   |DTW        |Long Delays  |
|247  |ABE   |ATL        |Long Delays  |
|219  |ABE   |ORD        |Long Delays  |
|211  |ABE   |ATL        |Long Delays  |
|197  |ABE   |DTW        |Long Delays  |
|192  |ABE   |ORD        |Long Delays  |
+-----+------+-----------+-------------+
only showing top 10 rows
```

Podobnie jak w przypadku API DataFrame i Dataset także podczas pracy z interfejsem `spark.sql` można przeprowadzać najczęstsze operacje wykonywane podczas analizy danych, np. te omówione w poprzednim rozdziale. Obliczenia przeprowadzane są dokładnie w taki sam sposób jak w silniku Spark SQL (więcej informacji na ten temat znajdziesz w podrozdziale „Optymalizator Catalyst" stanowiącym część rozdziału 3.), więc wygenerowane wyniki są identyczne.

Wszystkie trzy spośród wcześniejszych zapytań SQL mogą być wyrażone za pomocą odpowiadającego im zapytania API DataFrame. Na przykład pierwsze z nich można zdefiniować z użyciem API Python DataFrame w następujący sposób:

```
# W kodzie Pythona.
from pyspark.sql.functions import col, desc
(df.select("distance", "origin", "destination")
  .where(col("distance") > 1000)
  .orderBy(desc("distance"))).show(10)

# Ewentualnie:
(df.select("distance", "origin", "destination")
  .where("distance > 1000")
  .orderBy("distance", ascending=False).show(10))
```

Spowoduje to wygenerowanie dokładnie takich samych wyników jak w przypadku wcześniejszego zapytania SQL.

```
+--------+------+-----------+
|distance|origin|destination|
+--------+------+-----------+
|4330    |HNL   |JFK        |
|4330    |HNL   |JFK        |
|4330    |HNL   |JFK        |
|4330    |HNL   |JFK        |
|4330    |HNL   |JFK        |
|4330    |HNL   |JFK        |
|4330    |HNL   |JFK        |
|4330    |HNL   |JFK        |
|4330    |HNL   |JFK        |
|4330    |HNL   |JFK        |
+--------+------+-----------+
only showing top 10 rows
```

W ramach ćwiczenia spróbuj skonwertować dwa pozostałe zapytania SQL na postać API DataFrame.

Jak pokazują omawiane przykłady, używanie interfejsu Spark SQL do pobierania danych przebiega podobnie jak tworzenie zwykłych zapytań SQL skierowanych do tabel relacyjnych baz danych. Wprawdzie zapytania te są pisane w kodzie SQL, ale pod względem czytelności i semantyki przypominają one API DataFrame, które zostało zaprezentowane w poprzednim rozdziale i będzie jeszcze omawiane w następnym.

Aby zezwolić na wykonywanie zapytań dotyczących strukturalnych danych, jak pokazaliśmy we wcześniejszych przykładach, Spark nadzoruje wszystkie skomplikowane kwestie podczas tworzenia widoków i tabel oraz zarządzania nimi — zarówno tymi znajdującymi się w pamięci, jak i na dysku. To prowadzi nas na kolejnego zagadnienia: jak są tworzone tabele i widoki oraz jak się nimi zarządza.

Widoki i tabele SQL

Tabele przechowują dane. Z każdą tabelą w Sparku powiązane są odpowiednie metadane, czyli informacje o niej i jej danych: schemat, opis, nazwa tabeli, nazwa bazy danych, nazwy kolumn, partycje, fizyczne położenie rzeczywistych danych itd. Wszystkie te dane przechowywane są w centralnym metamagazynie danych.

Zamiast oddzielnych metamagazynów danych dla tabel Sparka framework domyślnie używa metamagazynu danych Apache Hive, położonego w */user/hive/warehouse*, w celu trwałego przechowywania wszystkich metadanych dotyczących tabel. Istnieje możliwość zmiany położenia domyślnego poprzez przypisanie zmiennej `spark.sql.warehouse.dir` wartości wskazującej na inne położenie, np. w lokalnym bądź zewnętrznym rozproszonym magazynie danych.

Tabele zarządzane kontra tabele niezarządzane

Spark pozwala tworzyć dwa typy tabel: zarządzane i niezarządzane. W przypadku tabeli *zarządzanej* framework zarządza metadanymi i danymi znajdującymi się w pliku magazynu danych. Może to być lokalny system plików, HDFS lub obiekt magazynu danych, takiego jak Amazon S3 lub Azure Blob. Natomiast w przypadku tabeli *niezarządzanej* Spark zarządza jedynie metadanymi, a Ty — danymi w zewnętrznych źródłach danych (*https://docs.databricks.com/data/data-sources/index.html*), takich jak Cassandra.

Skoro Spark zarządza całą zawartością tabeli zarządzanej, zapytanie SQL takie jak `DROP TABLE` *`nazwa_tabeli`* powoduje usunięcie zarówno metadanych, jak i danych. Z kolei w tabeli niezarządzanej to samo zapytanie spowoduje usunięcie jedynie metadanych, a nie rzeczywistych danych. W następnym punkcie pokażemy przykłady tworzenia tabel zarządzanych i niezarządzanych.

Tworzenie baz danych i tabel SQL

Tabele znajdują się w bazie danych. Domyślnie Spark tworzy je w bazie danych o nazwie `default`. W celu utworzenia własnej bazy danych można wykonać odpowiednie zapytanie SQL z poziomu aplikacji Sparka lub notatnika. Wykorzystamy teraz zbiór danych dotyczący opóźnień rejsów lotniczych w USA i przystąpimy do utworzenia dwóch tabel: zarządzanej i niezarządzanej.

Rozpoczniemy jednak od utworzenia nowej bazy danych o nazwie `learn_spark_db` i wskazania Sparkowi, że to właśnie ona ma być używana.

```
// W kodzie Scali i Pythona.
spark.sql("CREATE DATABASE learn_spark_db")
spark.sql("USE learn_spark_db")
```

Od tej chwili wszystkie wykonywane w aplikacji polecenia dotyczące tworzenia tabel będą powodowały utworzenie ich w tej bazie danych (`learn_spark_db`).

Tworzenie tabeli zarządzanej

W celu utworzenia tabeli zarządzanej w bazie danych `learn_spark_db` można wysłać zapytanie SQL podobne do tego przedstawionego poniżej:

```
// W kodzie Scali i Pythona.
spark.sql("CREATE TABLE managed_us_delay_flights_tbl (date STRING, delay INT,
  distance INT, origin STRING, destination STRING)")
```

To samo zadanie można wykonać za pomocą API DataFrame w następujący sposób:

```
# W kodzie Pythona.
# Ścieżka dostępu do pliku CSV zawierającego informacje o opóźnieniach lotów w USA.
csv_file = "/databricks-datasets/learning-spark-v2/flights/departuredelays.csv"
# Schemat taki jak ten zdefiniowany w poprzednim przykładzie.
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")
```

Wynikiem działania tych poleceń jest utworzenie tabeli zarządzanej `us_delay_flights_tbl` w bazie danych `learn_spark_db`.

Tworzenie tabeli niezarządzanej

Tabelę niezarządzaną można utworzyć na podstawie własnego źródła danych, np. pliku Parquet, CSV lub JSON przechowywanego w magazynie danych dostępnym dla aplikacji Sparka.

W celu utworzenia tabeli niezarządzanej na podstawie źródła danych takiego jak plik CSV należy użyć następującego kodu SQL:

```
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')""")
```

Natomiast w przypadku API DataFrame należy użyć kodu:

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

Aby umożliwić analizowanie omawianych tutaj przypadków, utworzyliśmy przykładowe notatniki Pythona i Scali, które znajdziesz w archiwum materiałów do tej książki (*ftp://ftp.helion.pl/przyklady/sparb2.zip*).

Tworzenie widoku

Poza tworzeniem tabel Spark umożliwia również tworzenie widoków na podstawie istniejących tabel. Widoki można podzielić na globalne (dostępne we wszystkich egzemplarzach `SparkSession` w danym klastrze) oraz mające zasięg sesji (dostępne jedynie dla pojedynczego egzemplarza `SparkSession`). Są one tymczasowe, czyli znikają po zakończeniu działania aplikacji Sparka.

Podczas tworzenia widoku (*https://docs.databricks.com/spark/latest/spark-sql/language-manual/sql-ref-syntax-ddl-create-view.html#id1*) stosowana jest podobna składnia jak w trakcie tworzenia tabeli w bazie danych. Po utworzeniu widoku można wysyłać do niego zapytania, podobnie jak w przypadku tabeli. Różnica między widokiem i tabelą polega na tym, że widok w rzeczywistości nie przechowuje danych. Po zakończeniu działania aplikacji Sparka tabela wciąż istnieje, natomiast widok znika.

Widok można utworzyć na podstawie istniejącej tabeli, wykorzystując do tego kod SQL. Na przykład jeśli chcesz pracować jedynie z podzbiorem danych dotyczących opóźnień rejsów lotniczych w USA między portami w Nowym Jorku (JFK) i San Francisco (SFO), to przedstawione tutaj zapytania utworzą widoki globalne i tymczasowe zawierające wyłącznie żądany fragment danych.

```
-- W kodzie SQL.
CREATE OR REPLACE GLOBAL TEMP VIEW us_origin_airport_SFO_global_tmp_view AS
  SELECT date, delay, origin, destination from us_delay_flights_tbl WHERE
  origin = 'SFO';

CREATE OR REPLACE TEMP VIEW us_origin_airport_JFK_tmp_view AS
  SELECT date, delay, origin, destination from us_delay_flights_tbl WHERE
  origin = 'JFK'
```

Ten sam efekt można uzyskać za pomocą API DataFrame po wykonaniu następujących poleceń:

```
# W kodzie Pythona.
df_sfo = spark.sql("SELECT date, delay, origin, destination FROM
  us_delay_flights_tbl WHERE origin = 'SFO'")
df_jfk = spark.sql("SELECT date, delay, origin, destination FROM
  us_delay_flights_tbl WHERE origin = 'JFK'")

# Utworzenie widoku tymczasowego i globalnego.
df_sfo.createOrReplaceGlobalTempView("us_origin_airport_SFO_global_tmp_view")
df_jfk.createOrReplaceTempView("us_origin_airport_JFK_tmp_view")
```

Po utworzeniu widoków można wysyłać do nich zapytania, podobnie jak w przypadku tabeli. Pamiętaj, że podczas uzyskiwania dostępu do globalnego widoku tymczasowego trzeba używać prefiksu `global_temp.<nazwa_widoku>`, ponieważ Spark tworzy tego rodzaju widoki w globalnej tymczasowej bazie danych o nazwie `global_temp`. Spójrz na kolejny fragment kodu.

```
-- W kodzie SQL.
SELECT * FROM global_temp.us_origin_airport_SFO_global_tmp_view
```

Z kolei dostęp do zwykłego widoku tymczasowego można uzyskać bez konieczności użycia prefiksu `global_temp`.

```
-- W kodzie SQL.
SELECT * FROM us_origin_airport_JFK_tmp_view

// W kodzie Scali i Pythona.
spark.read.table("us_origin_airport_JFK_tmp_view")
// Ewentualnie:
spark.sql("SELECT * FROM us_origin_airport_JFK_tmp_view")
```

Usunięcie widoku przebiega dokładnie w taki sam sposób jak usunięcie tabeli.

```
-- W kodzie SQL.
DROP VIEW IF EXISTS us_origin_airport_SFO_global_tmp_view;
DROP VIEW IF EXISTS us_origin_airport_JFK_tmp_view

// W kodzie Scali i Pythona.
spark.catalog.dropGlobalTempView("us_origin_airport_SFO_global_tmp_view")
spark.catalog.dropTempView("us_origin_airport_JFK_tmp_view")
```

Widok tymczasowy kontra globalny widok tymczasowy

Różnica między tymi dwoma widokami, *tymczasowym* i *globalnym tymczasowym*, jest subtelna, w związku z czym może być źródłem dezorientacji wśród wielu nowych programistów Sparka. Widok tymczasowy jest powiązany z pojedynczym egzemplarzem `SparkSession` w aplikacji. Z kolei globalny widok tymczasowy jest dostępny w wielu egzemplarzach `SparkSession` w aplikacji Sparka. Oczywiście istnieje możliwość utworzenia wielu egzemplarzy `SparkSession` (*https://www.waitingforcode.com/apache-spark-sql/multiple-sparksession-one-sparkcontext/read#mutliple_SparkSessions_use_cases*) w pojedynczej aplikacji Sparka. Takie rozwiązanie może być użyteczne, np. jeśli chcesz uzyskać dostęp do danych z dwóch odmiennych egzemplarzy `SparkSession` (i połączyć te dane), w przypadku gdy egzemplarze te nie współdzielą tych samych konfiguracji metamagazynu danych Hive.

Wyświetlanie metadanych

Jak wcześniej wspomnieliśmy, Spark zarządza metadanymi powiązanymi z każdą tabelą zarządzaną lub niezarządzaną. Odbywa się to za pomocą egzemplarza klasy `Catalog`, czyli abstrakcji wysokiego poziomu w Spark SQL przeznaczonej do przechowywania metadanych. W wydaniu Spark 2.x funkcjonalność tego obiektu została poszerzona o nowe metody publiczne, umożliwiające analizowanie metadanych dotyczących baz danych, tabel i widoków. Spark 3.0 jeszcze bardziej rozbudowuje te możliwości i pozwala programiście używać zewnętrznego obiektu `catalog` (co zostanie pokrótce omówione w rozdziale 12.).

Na przykład w aplikacji Sparka, po utworzeniu zmiennej `spark` egzemplarza obiektu `SparkSession`, można uzyskać dostęp do wszystkich przechowywanych metadanych za pomocą metod takich jak te poniżej:

```
// W kodzie Scali i Pythona.
spark.catalog.listDatabases()
spark.catalog.listTables()
spark.catalog.listColumns("us_delay_flights_tbl")
```

Zaimportuj notatnik znajdujący się wśród materiałów do tej książki (*ftp://ftp.helion.pl/przyklady/sparb2.zip*) i wypróbuj tę możliwość samodzielnie.

Buforowanie tabel SQL

Wprawdzie strategie buforowania tabel zostaną omówione w następnym rozdziale, ale warto w tym miejscu wspomnieć, że podobnie jak w przypadku egzemplarzy DataFrame istnieje możliwość buforowania i niebuforowania widoków i tabel SQL. W Sparku 3.0 (*https://spark.apache.org/docs/latest/sql-ref-syntax-aux-cache-cache-table.html*) jedną z opcji dotyczących tabeli jest `LAZY`, która określa, że tabela powinna być buforowana dopiero po pierwszym użyciu, a nie natychmiast.

```
-- W kodzie SQL.
CACHE [LAZY] TABLE <nazwa-tabeli>
UNCACHE TABLE <nazwa-tabeli>
```

Wczytywanie zawartości tabeli do egzemplarza DataFrame

Inżynierowie danych bardzo często tworzą potoki danych jako część swojego standardowego procesu analizy danych i przeprowadzania na nich operacji typu ETL. Wypełniają bazy danych i tabele Spark SQL oczyszczonymi danymi, przeznaczonymi do użycia przez aplikację.

Załóżmy, że mamy bazę danych `learn_spark_db` i tabelę `us_delay_flights_tbl`, które są gotowe do użycia. Zamiast wczytywać dane z zewnętrznego pliku JSON, możemy po prostu użyć zapytania SQL pobierającego dane, a otrzymany wynik przypisać egzemplarzowi DataFrame.

```
// W kodzie Scali.
val usFlightsDF = spark.sql("SELECT * FROM us_delay_flights_tbl")
val usFlightsDF2 = spark.table("us_delay_flights_tbl")

# W kodzie Pythona.
us_flights_df = spark.sql("SELECT * FROM us_delay_flights_tbl")
us_flights_df2 = spark.table("us_delay_flights_tbl")
```

W ten sposób otrzymujemy egzemplarz DataFrame z oczyszczonymi danymi, które zostały wczytane z istniejącej tabeli Spark SQL. Dane można odczytywać także w innych formatach, używając w tym celu wbudowanych w Sparku źródeł danych. Zapewnia to elastyczność podczas pracy z różnymi najczęściej używanymi formatami plików.

Źródła danych dla egzemplarzy DataFrame i tabel SQL

Jak można zobaczyć na rysunku 4.1 znajdującym się na początku tego rozdziału, Spark SQL oferuje interfejs dla różnych źródeł danych. Dostarcza także zbiór powszechnie używanych metod przeznaczonych do odczytywania danych z tych źródeł i zapisywania ich w tych źródłach za pomocą API źródeł danych (*https://databricks.com/blog/2015/01/09/spark-sql-data-sources-api-unified-data-access-for-the-spark-platform.html*).

W tym podrozdziale omówimy kilka spośród wbudowanych źródeł danych (*https://spark.apache.org/docs/latest/sql-data-sources.html#data-sources*), dostępnych formatów plików oraz sposobów wczytywania i zapisywania danych, a także określonych opcji związanych z tymi źródłami danych.

Jednak wcześniej trzeba się zapoznać z dwoma konstrukcjami API źródeł danych wysokiego poziomu, które wskazują, w jaki sposób odbywa się współpraca z różnymi źródłami danych: `DataFrameReader` i `DataFrameWriter`.

DataFrameReader

`DataFrameReader` (*https://spark.apache.org/docs/latest/api/scala/org/apache/spark/sql/DataFrameReader.html*) to podstawowa konstrukcja przeznaczona do wczytywania danych ze źródła danych do egzemplarza DataFrame. Ma ona zdefiniowany format i zalecany wzorzec użycia:

```
DataFrameReader.format(args).option("key", "value").schema(args).load()
```

Ten wzorzec łączenia metod jest powszechnie stosowany w Sparku i pozostaje łatwy do odczytania. Użyliśmy go już w rozdziale 3. podczas omawiania najczęściej stosowanych wzorców analizy danych.

Należy pamiętać, że dostęp do `DataFrameReader` można uzyskać jedynie poprzez egzemplarz `SparkSession`. Oznacza to, że nie można utworzyć egzemplarza `DataFrameReader`. W celu uzyskania uchwytu do egzemplarza trzeba skorzystać z następującego wywołania:

```
SparkSession.read
// Ewentualnie:
SparkSession.readStream
```

Podczas gdy `read` zwraca uchwyt do obiektu `DataFrameReader` w celu wczytania do egzemplarza DataFrame danych pochodzących ze statycznego źródła danych, `readStream` zwraca egzemplarz pozwalający wczytać dane ze źródła strumieniowania. (Dokładniejsze omówienie strumieniowania strukturalnego znajdziesz w dalszej części tej książki).

Argumenty poszczególnych metod publicznych `DataFrameReader` pobierają różne wartości. Zostały one wymienione w tabeli 4.1 razem z podzbiorem obsługiwanych argumentów.

Tabela 4.1. Metody, argumenty i opcje DataFrameReader

Metoda	Argumenty	Opis
`format()`	`"parquet"`, `"csv"`, `"txt"`, `"json"`, `"jdbc"`, `"orc"`, `"avro"` itd.	Jeżeli metoda ta nie zostanie użyta, to domyślnie stosowany będzie format Parquet lub inny format zdefiniowany w `spark.sql.sources.default`.

`option()`	`("mode", {PERMISSIVE | FAILFAST | DROPMALFORMED } )`
	`("inferSchema", {true | false})`
	`("path", "`*`ścieżka_do_pliku_danych_źródłowych`*`")`
		Seria par klucz-wartość i opcji. W dokumentacji Sparka (*https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrameReader.csv.html*) znajdują się przykłady i wyjaśnienia dotyczące różnych trybów i sposobu ich działania. Trybem domyślnym jest `PERMISSIVE`. Opcje `inferSchema` i `mode` są związane z formatami plików JSON i CSV.

`schema()`	*`Ciąg tekstowy DDL`* lub `StructType`, np. `'A INT, B STRING'` lub `StructType(...)`	W przypadku formatów JSON i CSV w metodzie `option()` można wybrać opcję automatycznego ustalenia schematu. Ogólnie rzecz biorąc, dostarczenie schematu dla dowolnego formatu oznacza, że wczytywanie danych będzie odbywało się szybciej. Gwarantuje to również zgodność danych z oczekiwanym schematem.
`load()`	*`"/ścieżka/do/źródła/danych"`*	Ścieżka dostępu do źródła danych. Wartość ta może być pusta, o ile ścieżka dostępu została podana w metodzie `option("path", "...")`.

Wprawdzie nie zamierzamy tutaj dokładnie omawiać wszystkich możliwych wariantów argumentów i opcji, ale w dokumentacji dotyczącej języków Python, Scala, R i Java (*https://spark.apache.org/docs/latest/sql-data-sources-load-save-functions.html#manually-specifying-options*) znajdują się odpowiednie podpowiedzi i wskazówki. Mimo tego warto w tym miejscu przedstawić kilka przykładów.

```
// W kodzie Scali.
// Użycie pliku w formacie Parquet.
val file = """/databricks-datasets/learning-spark-v2/flights/summary-
  data/parquet/2010-summary.parquet"""
val df = spark.read.format("parquet").load(file)
// Użycie pliku w formacie Parquet. Można pominąć wywołanie format("parquet"), ponieważ jest ono używane domyślnie.
val df2 = spark.read.load(file)
// Użycie pliku w formacie CSV.
val df3 = spark.read.format("csv")
  .option("inferSchema", "true")
  .option("header", "true")
  .option("mode", "PERMISSIVE")
  .load("/databricks-datasets/learning-spark-v2/flights/summary-data/csv/*")
// Użycie pliku w formacie JSON.
val df4 = spark.read.format("json")
  .load("/databricks-datasets/learning-spark-v2/flights/summary-data/json/*")
```

Ogólnie rzecz biorąc, schemat nie jest potrzebny podczas wczytywania danych ze statycznego źródła danych typu Parquet — metadane Parquet zwykle zawierają schemat, więc będzie on użyty. Natomiast w przypadku strumieniowanych źródeł danych konieczne będzie dostarczenie schematu. (Odczytywanie danych ze strumieniowanych źródeł danych zostanie dokładnie omówione w rozdziale 8.).

Parquet to domyślne i preferowane źródło danych w Sparku, ponieważ jest ono efektywne, używa kolumnowego magazynu danych oraz stosuje szybkie algorytmy kompresji. Kolejne zalety poznasz później, gdy będziemy znacznie dokładniej omawiać optymalizator Catalyst.

DataFrameWriter

`DataFrameWriter` działa w zupełnie odwrotny sposób niż `DataFrameReader` — zapisuje dane do podanego wbudowanego źródła danych. Inaczej niż w przypadku `DataFrameReader` dostępu do egzemplarza nie uzyskuje się z poziomu obiektu `SparkSession`, tylko z poziomu zapisywanego egzemplarza DataFrame. Z `DataFrameWriter` wiąże się kilka zalecanych wzorców użycia:

```
DataFrameWriter.format(args)
  .option(args)
  .bucketBy(args)
  .partitionBy(args)
  .save(path)

DataFrameWriter.format(args).option(args).sortBy(args).saveAsTable(table)
```

W celu pobrania uchwytu do egzemplarza należy użyć następującego polecenia:

```
DataFrame.write
// Ewentualnie:
DataFrame.writeStream
```

Argumenty poszczególnych metod publicznych `DataFrameWriter` pobierają różne wartości. Zostały one wymienione w tabeli 4.2 razem z podzbiorem obsługiwanych argumentów.

Tabela 4.2. Metody, argumenty i opcje DataFrameWriter

Metoda	Argumenty	Opis
`format()`	`"parquet"`, `"csv"`, `"txt"`, `"json"`, `"jdbc"`, `"orc"`, `"avro"` itd.	Jeżeli metoda ta nie zostanie użyta, to domyślnie stosowany będzie format Parquet lub inny format zdefiniowany w `spark.sql.sources.default`.

`option()`	`("mode", {append | overwrite | ignore | error or error ifexists})`
	`("mode", {SaveMode.Overwrite | SaveMode.Append, SaveMode.Ignore, SaveMode.ErrorIfExists})`
	`("path", "`*`ścieżka_do_pliku_danych_źródłowych`*`")`
		Seria par klucz-wartość i opcji. Przykłady znajdują się w dokumentacji Sparka (*https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.sql.DataFrameWriter.mode.html*). Metoda ta jest przeciążona. Opcje domyślne to `error or error ifexists` i `SaveMode.ErrorIfExists`. Jeżeli dane już istnieją, opcje te spowodują zgłoszenie wyjątku.

`bucketBy()`	`(numBuckets, col, col..., coln)`	Liczba kubełków i nazwy kolumn mapowane na kubełki. Metoda ta stosuje schemat kubełków Hive w systemie plików.
`save()`	`"`*`/ścieżka/do/źródła/danych`*`"`	Ścieżka dostępu wskazująca miejsce zapisu danych. Może to być pusta wartość, o ile ścieżka dostępu została podana w metodzie `option("path", "...")`.
`saveAsTable()`	`"`*`nazwa_tabeli`*`"`	Tabela, w której będą zapisane dane.

Oto krótki fragment kodu pokazujący użycie metod i argumentów `DataFrameWriter`.

```
// W kodzie Scali.
// Użycie pliku w formacie JSON.
val location = ...
df.write.format("json").mode("overwrite").save(location)
```

Parquet

Omawianie źródeł danych rozpoczynamy od Parquet (*https://spark.apache.org/docs/latest/sql-data-sources-parquet.html*), ponieważ stanowi on domyślne źródło danych w Sparku. Obsługiwany i powszechnie używany przez wiele ogromnych frameworków i platform przetwarzania danych, Parquet to kolumnowy format pliku typu open source, oferujący wiele optymalizacji wejścia-wyjścia, takich jak kompresja, która pozwala zaoszczędzić miejsce i jednocześnie zapewnia szybki dostęp do kolumn danych.

Ze względu na efektywność działania i wprowadzone optymalizacje sugerujemy, że po transformacji i oczyszczeniu danych warto zapisać je w egzemplarzach DataFrame w formacie Parquet, by następnie z nich korzystać. (Parquet to również domyślny otwarty format tabeli dla Delta Lake, co omówimy dokładniej w rozdziale 9.).

Wczytywanie plików Parquet do egzemplarzy DataFrame

Pliki Parquet (*https://github.com/apache/parquet-format#file-format*) są przechowywane w strukturze katalogu zawierającego pliki danych, metadane, pewną liczbę plików skompresowanych i niektóre pliki stanu. Metadane zawierają informacje o wersji formatu pliku, schemacie i kolumnach danych, takie jak ścieżka dostępu itd.

Na przykład katalog w pliku Parquet może składać się z przedstawionego tutaj zbioru plików:

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

W katalogu może znajdować się wiele skompresowanych plików *part-XXXX* (pokazane tutaj nazwy zostały skrócone, aby zmieściły się na stronie drukowanej wersji tej książki).

W celu wczytania pliku Parquet do egzemplarza DataFrame należy po prostu podać format i ścieżkę dostępu.

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

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

O ile dane nie są odczytywane ze strumieniowanego źródła danych, nie ma konieczności dostarczania schematu, ponieważ Parquet zapisuje go jako część swoich metadanych.

Wczytywanie zawartości pliku typu Parquet do tabeli Spark SQL

Zawartość pliku typu Parquet można wczytywać nie tylko do egzemplarza DataFrame w Sparku, ale również do widoku lub tabeli niezarządzanej, używając w tym celu bezpośrednio kodu SQL.

```
-- W kodzie 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/" )
```

Po utworzeniu widoku lub tabeli dane można wczytywać do egzemplarza DataFrame za pomocą kodu SQL, jak pokazaliśmy we wcześniejszych przykładach.

```
// W kodzie Scali.
spark.sql("SELECT * FROM us_delay_flights_tbl").show()

# W kodzie Pythona.
spark.sql("SELECT * FROM us_delay_flights_tbl").show()
```

Wyniki wykonania obu tych operacji są dokładnie takie same.

```
+-----------------+-------------------+-----+
|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
```

Zapisywanie egzemplarza DataFrame do pliku typu Parquet

Zapisywanie egzemplarza DataFrame jako tabeli lub pliku to często przeprowadzana operacja w Sparku. W celu zapisania egzemplarza DataFrame można po prostu użyć metod i argumentów `DataFrameWriter` omówionych we wcześniejszej części tego rozdziału. Trzeba przy tym podać położenie zapisywanego pliku Parquet. Spójrz na kolejny przykład.

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

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

Pamiętaj, że Parquet to domyślny format pliku. Jeżeli nie użyjesz metody `format()`, egzemplarz DataFrame wciąż będzie zapisywany w postaci pliku typu Parquet.

Polecenia te powodują utworzenie zbioru zwięzłych i skompresowanych plików typu Parquet w podanej lokalizacji. Ponieważ w celu przeprowadzenia kompresji użyliśmy opcji `snappy`, pliki zostały skompresowane z wykorzystaniem tego algorytmu. W celu zachowania zwięzłości w tym przykładzie pokazaliśmy wygenerowanie tylko jednego pliku, podczas gdy w rzeczywistości mogą ich być dziesiątki.

```
-rw-r--r--  1 jules  wheel    0 May 19 10:58 _SUCCESS
-rw-r--r--  1 jules  wheel  966 May 19 10:58 part-00000-<...>-c000.snappy.parquet
```

Zapisywanie egzemplarza DataFrame do tabeli Spark SQL

Zapisywanie egzemplarza DataFrame do tabeli SQL jest równie łatwe jak zapisywanie danych w pliku. Wystarczy jedynie użyć metody `saveAsTable()` zamiast `save()`. W omawianym przykładzie została utworzona tabela zarządzana o nazwie `us_delay_flights_tbl`.

```
// W kodzie Scali.
df.write
  .mode("overwrite")
  .saveAsTable("us_delay_flights_tbl")

# W kodzie Pythona.
(df.write
  .mode("overwrite")
  .saveAsTable("us_delay_flights_tbl"))
```

Podsumowując, Parquet to preferowany i domyślny format pliku wbudowanego źródła danych w Sparku. Jest on wykorzystywany przez wiele innych frameworków. Zalecamy używanie tego formatu w trakcie analizy danych i przeprowadzania operacji typu ETL.

JSON

JSON (ang. *javascript object notation*) to inny popularny format danych. Jego twórcy twierdzą, że w porównaniu z formatem XML jest on łatwy do odczytania i przetwarzania. Istnieją dwa reprezentacyjne tryby formatu JSON: pojedynczego wiersza i wielu wierszy (*https://docs.databricks.com/data/data-sources/read-json.html*). Oba są obsługiwane przez Sparka.

W trybie pojedynczego wiersza każdy wiersz określa pojedynczy obiekt JSON (*https://jsonlines.org/*), podczas gdy w trybie wielowierszowym pojedynczy obiekt JSON jest opisywany przez wiele wierszy. Aby odczytywać dane w tym formacie, w metodzie `option()` trzeba przypisać wartość `true` opcji `multiLine`.

Wczytywanie zawartości pliku JSON do egzemplarza DataFrame

Zawartość pliku JSON można wczytać do egzemplarza DataFrame dokładnie w taki sam sposób, jak to miało miejsce podczas pracy z danymi Parquet — wystarczy w metodzie `format()` użyć opcji `"json"`.

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

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

Wczytywanie zawartości pliku JSON do tabeli Spark SQL

Istnieje możliwość utworzenia tabeli SQL na podstawie pliku JSON, podobnie jak w przypadku pracy z danymi Parquet.

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

Po utworzeniu tabeli można przystąpić do wczytywania danych do egzemplarza DataFrame za pomocą kodu SQL.

```
// W kodzie Scali i Pythona.
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   |
|United States    |India              |62   |
|United States    |Singapore          |1    |
|United States    |Grenada            |62   |
|Costa Rica       |United States      |588  |
|Senegal          |United States      |40   |
|Moldova          |United States      |1    |
+-----------------+-------------------+-----+
only showing top 10 rows
```

Zapisywanie egzemplarza DataFrame do pliku typu JSON

Zapisywanie egzemplarza DataFrame do pliku typu JSON stanowi prostą operację. Należy w tym celu skorzystać z odpowiednich metod i argumentów `DataFrameWriter` oraz podać miejsce, w którym ma być utworzony plik JSON.

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

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

Polecenia te spowodują utworzenie w podanym miejscu katalogu, w którym następnie zostanie umieszczony zbiór plików JSON.

```
-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
```

Opcje źródła danych JSON

W tabeli 4.3 zostały wymienione najczęściej używane opcje JSON dla `DataFrameReader` (*https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.sql.DataFrameReader.json.html*) i `DataFrameWriter` (*https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrameWriter.json.html?highlight=json*). Pełną listę opcji znajdziesz w dokumentacji.

Tabela 4.3. Opcje dla DataFrameReader i DataFrameWriter dotyczące formatu JSON

Nazwa właściwości	Wartości	Opis	Zasięg
`compression`	`none`, `uncompressed`, `bzip2`, `deflate`, `gzip`, `lz4` lub `snappy`	Użycie podanego kodeka kompresji w trakcie operacji zapisu. Warto dodać, że podczas odczytywania danych zastosowanie kompresji lub kodeka będzie możliwe tylko na podstawie rozszerzenia pliku.	Zapis
`dateFormat`	`yyyy-MM-dd` lub `DateTimeFormatter`	Użycie podanego formatu lub dowolnego formatu obsługiwanego przez `DateTimeFormatter` w Javie.	Odczyt i zapis
`multiLine`	`true`, `false`	Użycie trybu wielowierszowego. Wartością domyślną jest `false` (tryb pojedynczego wiersza).	Odczyt
`allowUnquoted ↪FieldNames`	`true`, `false`	Użycie niecytowanych nazw pól w JSON. Wartością domyślną jest `false`.	Odczyt

CSV

Używany równie często jak zwykłe pliki tekstowe, format ten przechwytuje poszczególne pola rozdzielone przecinkami. Każdy wiersz zawierający rozdzielone przecinkami pola przedstawia rekord. Nawet pomimo tego, że przecinek jest separatorem domyślnym, istnieje możliwość wskazania innego separatora, jeśli przecinek stanowi część danych. Popularne aplikacje do tworzenia arkuszy kalkulacyjnych potrafią generować pliki w formacie CSV. Format ten jest również popularny wśród analityków danych i analityków biznesowych.

Wczytywanie zawartości pliku CSV do egzemplarza DataFrame

Podobnie jak w przypadku innych wbudowanych źródeł danych metody i argumenty `DataFrameReader` można wykorzystać w celu wczytania zawartości pliku CSV do egzemplarza DataFrame.

```
// W kodzie Scali.
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")      // Zakończenie działania w przypadku wystąpienia błędów.
  .option("nullValue", "")         // Brakujące pola danych będą zastąpione cudzysłowem.
  .load(file)

# W kodzie Pythona.
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")  # Zakończenie działania w przypadku wystąpienia błędów.
  .option("nullValue", "")     # Brakujące pola danych będą zastąpione cudzysłowem.
  .load(file))
```

Wczytywanie zawartości pliku CSV do tabeli Spark SQL

Utworzenie tabeli SQL na podstawie źródła danych CSV niczym się nie różni od używania jako źródła pliku typu Parquet lub JSON.

```
-- W kodzie SQL.
CREATE OR REPLACE TEMPORARY VIEW us_delay_flights_tbl
    USING csv
    OPTIONS (
      path "/databricks-datasets/learning-spark-v2/flights/summary-data/csv/*",
      header "true",
      inferSchema "true",
      mode "FAILFAST"
    )
```

Po utworzeniu tabeli dane można wczytywać do egzemplarza DataFrame za pomocą kodu SQL — dokładnie w taki sam sposób jak we wcześniejszych przykładach.

```
// W kodzie Scali i Pythona.
spark.sql("SELECT * FROM us_delay_flights_tbl").show(10)

+-----------------+-------------------+-----+
|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
```

Zapisywanie egzemplarza DataFrame do pliku typu CSV

Zapisywanie egzemplarza DataFrame do pliku typu CSV stanowi prostą operację. Należy w tym celu skorzystać z odpowiednich metod i argumentów `DataFrameWriter` oraz podać miejsce, w którym ma być utworzony plik CSV.

```
// W kodzie Scali.
df.write.format("csv").mode("overwrite").save("/tmp/data/csv/df_csv")

# W kodzie Pythona.
df.write.format("csv").mode("overwrite").save("/tmp/data/csv/df_csv")
```

Polecenia te spowodują utworzenie w podanym miejscu katalogu, w którym następnie zostanie umieszczony zbiór plików CSV.

```
-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
```

Opcje źródła danych CSV

W tabeli 4.4 zostały wymienione najczęściej używane opcje CSV dla `DataFrameReader` (*https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.sql.DataFrameReader.csv.html*) i `DataFrameWriter` (*https://spark.apache.org/docs/3.2.0/api/python/reference/api/pyspark.sql.DataFrameWriter.csv.html*). Ponieważ pliki CSV mogą być skomplikowane, dostępnych jest wiele opcji służących do ich obsługi. Pełną listę opcji znajdziesz w dokumentacji.

Tabela 4.4. Opcje dla DataFrameReader i DataFrameWriter dotyczące formatu CSV

Nazwa właściwości	Wartości	Opis	Zasięg
`compression`	`none`, `bzip2`, `deflate`, `gzip`, `lz4` lub `snappy`	Użycie podanego kodeka kompresji w trakcie operacji zapisu.	Zapis
`dateFormat`	`yyyy-MM-dd` lub `DateTimeFormatter`	Użycie podanego formatu lub dowolnego formatu obsługiwanego przez `DateTimeFormatter` w Javie.	Odczyt i zapis
`multiLine`	`true`, `false`	Użycie trybu wielowierszowego. Wartością domyślną jest `false` (tryb pojedynczego wiersza).	Odczyt
`inferSchema`	`true`, `false`	Wartość `true` oznacza, że Spark ma ustalić typ danych kolumny. Wartością domyślną jest `false`.	Odczyt
`sep`	Dowolny znak	Użycie podanego znaku jako separatora kolumn wartości w rekordzie. Domyślnym separatorem jest przecinek.	Odczyt i zapis
`escape`	Dowolny znak	Użycie podanego znaku jako znaku sterującego. Domyślnie jest nim `\`.	Odczyt i zapis
`header`	`true`, `false`	Wskazuje, czy pierwszy wiersz danych to nagłówek określający nazwy poszczególnych kolumn. Wartością domyślną jest `false`.	Odczyt i zapis

Avro

Wprowadzony w Sparku 2.4 (*https://databricks.com/blog/2018/11/30/apache-avro-as-a-built-in-data-source-in-apache-spark-2-4.html*) format Avro (*https://docs.databricks.com/data/data-sources/read-avro.html*), jako wbudowane źródło danych, jest używany np. przez Apache Kafka (*https://www.confluent.io/blog/avro-kafka-data/*) do serializacji i deserializacji komunikatów. Format ten posiada wiele zalet, m.in. bezpośrednie mapowanie na JSON, dużą szybkość działania i efektywność. Jego obsługę zapewniono w wielu językach programowania.

Wczytywanie zawartości pliku Avro do egzemplarza DataFrame

Wczytanie zawartości pliku w formacie Avro do egzemplarza DataFrame za pomocą `DataFrameReader` odbywa się podobnie jak w przypadku innych źródeł danych omówionych we wcześniejszej części tego rozdziału.

```
// W kodzie Scali.
val df = spark.read.format("avro")
  .load("/databricks-datasets/learning-spark-v2/flights/summary-data/avro/*")
```

```
df.show(false)

# W kodzie Pythona.
df = (spark.read.format("avro")
  .load("/databricks-datasets/learning-spark-v2/flights/summary-data/avro/*"))
df.show(truncate=False)

+-----------------+-------------------+-----+
|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
```

Wczytywanie zawartości pliku Avro do tabeli Spark SQL

Utworzenie tabeli SQL na podstawie źródła danych Avro niczym się nie różni od używania jako źródła pliku typu Parquet, JSON lub CSV.

```
-- W kodzie SQL.
CREATE OR REPLACE TEMPORARY VIEW episode_tbl
    USING avro
    OPTIONS (
      path "/databricks-datasets/learning-spark-v2/flights/summary-data/avro/*"
    )
```

Po utworzeniu tabeli dane można wczytywać do egzemplarza DataFrame za pomocą kodu SQL.

```
// W kodzie Scali.
spark.sql("SELECT * FROM episode_tbl").show(false)

# W kodzie Pythona.
spark.sql("SELECT * FROM episode_tbl").show(truncate=False)

+-----------------+-------------------+-----+
|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
```

Zapisywanie egzemplarza DataFrame do pliku typu Avro

Zapisywanie egzemplarza DataFrame do pliku typu Avro stanowi prostą operację. Jak zwykle należy skorzystać z odpowiednich metod i argumentów `DataFrameWriter` oraz podać miejsce, w którym ma być utworzony plik Avro.

```
// W kodzie Scali.
df.write
  .format("avro")
  .mode("overwrite")
  .save("/tmp/data/avro/df_avro")

# W kodzie Pythona.
(df.write
  .format("avro")
  .mode("overwrite")
  .save("/tmp/data/avro/df_avro"))
```

Polecenia te spowodują utworzenie w podanym miejscu katalogu, w którym następnie zostanie umieszczony zbiór plików Avro.

```
-rw-r--r--  1 jules  wheel    0 May 17 11:54 _SUCCESS
-rw-r--r--  1 jules  wheel  526 May 17 11:54 part-00000-ffdf70f4-<...>-c000.avro
```

Opcje źródła danych Avro

W tabeli 4.5 zostały wymienione najczęściej używane opcje Avro dla `DataFrameReader` i `DataFrameWriter`. Ich pełną listę znajdziesz w dokumentacji (*https://spark.apache.org/docs/latest/sql-data-sources-avro.html*).

Tabela 4.5. Opcje dla DataFrameReader i DataFrameWriter dotyczące formatu Avro

Nazwa właściwości	Wartości	Opis	Zasięg
`avroSchema`	Brak	Opcjonalny schemat Avro dostarczony w formacie JSON przez użytkownika. Typy danych i nazwy pól rekordu powinny odpowiadać danym wejściowym Avro (wewnętrznego typu danych Sparka) lub danym dostarczonym przez optymalizator Catalyst. W przeciwnym razie operacja odczytu/zapisu zakończy się niepowodzeniem.	Odczyt i zapis
`recordName`	`topLevelRecord`	Nazwa rekordu najwyższego poziomu zapisana w wyniku, wymagana przez specyfikację Avro.	Zapis
`recordNamespace`	`""`	Przestrzeń nazw rekordu w wyniku zapisu.	Zapis
`ignoreExtension`	`true`	Jeżeli ta opcja jest włączona, zostaną wczytane wszystkie pliki (zarówno z rozszerzeniem *.avro*, jak i bez niego). W przeciwnym razie pliki bez rozszerzenia *.avro* zostaną zignorowane.	Odczyt
`compression`	`snappy`	Pozwala na podanie kodeka kompresji używanego podczas operacji zapisu. Obecnie obsługiwane są następujące kodeki: `uncompressed`, `snappy`, `deflate`, `bzip2` i `xz`. Jeżeli ta opcja nie jest zdefiniowana, uwzględniona zostanie wartość przypisana `spark.sql.avro.compression.codec`.	Zapis

ORC

W Sparku 2.x dodano obsługę dodatkowego zoptymalizowanego i kolumnowego formatu pliku — służy do tego wektorowy czytnik ORC (*https://spark.apache.org/docs/latest/sql-data-sources-orc.html*). To, która implementacja ORC zostanie użyta, określają dwa ustawienia konfiguracyjne Sparka. Gdy `spark.sql.orc.impl` ma przypisaną wartość `native`, a `spark.sql.orc.enableVectorizedReader` ma przypisaną wartość `true`, wówczas Spark używa czytnika wektorowego ORC. Czytnik (*https://cwiki.apache.org/confluence/display/Hive/Vectorized+Query+Execution*) ten nie odczytuje rekordów pojedynczo, tylko w postaci bloków (blok często zawiera 1024 rekordów), co zapewnia płynniejsze działanie oraz zmniejszenie poziomu użycia procesora podczas przeprowadzania obciążających operacji, takich jak skanowanie, filtrowanie, agregowanie i złączanie.

W przypadku tabel Hive ORC SerDe (serializacji i deserializacji) utworzonych za pomocą polecenia SQL `USING HIVE OPTIONS (fileFormat 'ORC')` czytnik wektorowy używany jest wtedy, gdy parametr konfiguracyjny Sparka `spark.sql.hive.convertMetastoreOrc` ma przypisaną wartość `true`.

Wczytywanie zawartości pliku ORC do egzemplarza DataFrame

Wczytanie danych do egzemplarza DataFrame za pomocą czytnika wektorowego ORC jest możliwe przy użyciu standardowych metod i opcji `DataFrameReader`.

```
// W kodzie Scali.
val file = "/databricks-datasets/learning-spark-v2/flights/summary-data/orc/*"
val df = spark.read.format("orc").load(file)
df.show(10, false)

# W kodzie Pythona.
file = "/databricks-datasets/learning-spark-v2/flights/summary-data/orc/*"
df = spark.read.format("orc").option("path", file).load()
df.show(10, False)
```

```
+-----------------+-------------------+-----+
|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
```

Wczytywanie zawartości pliku ORC do tabeli Spark SQL

Utworzenie tabeli SQL na podstawie źródła danych ORC niczym się nie różni od używania jako źródła pliku typu Parquet, JSON, CSV lub Avro.

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

Po utworzeniu tabeli dane można wczytywać do egzemplarza DataFrame za pomocą kodu SQL.

```
// W kodzie Scali i Pythona.
spark.sql("SELECT * FROM us_delay_flights_tbl").show()

+-----------------+-------------------+-----+
|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
```

Zapisywanie egzemplarza DataFrame do pliku typu ORC

Zapisywanie egzemplarza DataFrame do pliku typu ORC stanowi prostą operację. Jak zwykle należy skorzystać z odpowiednich metod i argumentów `DataFrameWriter`.

```
// W kodzie Scali.
df.write.format("orc")
  .mode("overwrite")
  .option("compression", "snappy")
  .save("/tmp/data/orc/df_orc")

# W kodzie Pythona.
(df.write.format("orc")
  .mode("overwrite")
  .option("compression", "snappy")
  .save("/tmp/data/orc/flights_orc"))
```

Polecenia te spowodują utworzenie w podanym miejscu katalogu, w którym następnie zostanie umieszczony zbiór plików ORC.

```
-rw-r--r--  1 jules  wheel    0 May 16 17:23 _SUCCESS
-rw-r--r--  1 jules  wheel  547 May 16 17:23 part-00000-<...>-c000.snappy.orc
```

Obrazy

W wydaniu Spark 2.4 społeczność wprowadziła nowe źródło danych, plik obrazu (*https://databricks.com/blog/2018/12/10/introducing-built-in-image-data-source-in-apache-spark-2-4.html*), obsługujący uczenie głębokie i frameworki uczenia maszynowego, takie jak TensorFlow i PyTorch. W przypadku

aplikacji służących do rozpoznawania obrazów za pomocą uczenia maszynowego wczytywanie i przetwarzanie zbiorów danych obrazów odgrywa ważną rolę.

Wczytywanie zawartości pliku obrazu do egzemplarza DataFrame

Wczytanie pliku obrazu do egzemplarza DataFrame, podobnie jak w przypadku wszystkich wcześniej omówionych formatów plików, jest możliwe przy użyciu standardowych metod i opcji `DataFrameReader`, jak możesz zobaczyć w kolejnym fragmencie kodu.

```
// W kodzie Scali.
import org.apache.spark.ml.source.image

val imageDir = "/databricks-datasets/learning-spark-v2/cctvVideos/train_images/"
val imagesDF = spark.read.format("image").load(imageDir)

imagesDF.printSchema

imagesDF.select("image.height", "image.width", "image.nChannels", "image.mode",
  "label").show(5, false)
```

```
# W kodzie Pythona.
from pyspark.ml import image

image_dir = "/databricks-datasets/learning-spark-v2/cctvVideos/train_images/"
images_df = spark.read.format("image").load(image_dir)
images_df.printSchema()
```

```
root
|-- image: struct (nullable = true)
|    |-- origin: string (nullable = true)
|    |-- height: integer (nullable = true)
|    |-- width: integer (nullable = true)
|    |-- nChannels: integer (nullable = true)
|    |-- mode: integer (nullable = true)
|    |-- data: binary (nullable = true)
|-- label: integer (nullable = true)
```

```
images_df.select("image.height", "image.width", "image.nChannels", "image.mode",
  "label").show(5, truncate=False)
```

```
+------+-----+---------+----+-----+
|height|width|nChannels|mode|label|
+------+-----+---------+----+-----+
|288   |384  |3        |16  |0    |
|288   |384  |3        |16  |1    |
|288   |384  |3        |16  |0    |
|288   |384  |3        |16  |0    |
|288   |384  |3        |16  |0    |
+------+-----+---------+----+-----+
only showing top 5 rows
```

Pliki binarne

W Sparku 3.0 dodano obsługę źródła danych w postaci plików binarnych (*https://spark.apache.org/docs/latest/sql-data-sources-binaryFile.html*). Egzemplarz `DataFrameReader` konwertuje każdy plik binarny na pojedynczy rekord DataFrame zawierający niezmodyfikowaną treść i metadane pliku.

Źródło danych w postaci pliku binarnego prowadzi do powstania egzemplarza DataFrame z następującymi kolumnami:

- `path`: `StringType`
- `modificationTime`: `TimestampType`
- `length`: `LongType`
- `content`: `BinaryType`

Wczytywanie zawartości pliku binarnego do egzemplarza DataFrame

W celu wczytania zawartości pliku binarnego należy podać `binaryFile` jako format źródła danych. Istnieje możliwość wczytywania plików ze ścieżkami dostępu dopasowanymi do danego wzorca globalnego przy jednoczesnym pozostawieniu niezmienionego zachowania wykrywania partycji dzięki opcji `pathGlobFilter` źródła danych. Na przykład w kolejnym fragmencie kodu odczytywane są wszystkie pliki JPG z katalogu źródłowego, ze wszelkimi partycjonowanymi katalogami.

```
// W kodzie Scali.
val path = "/databricks-datasets/learning-spark-v2/cctvVideos/train_images/"
val binaryFilesDF = spark.read.format("binaryFile")
  .option("pathGlobFilter", "*.jpg")
  .load(path)
binaryFilesDF.show(5)

# W kodzie Pythona.
path = "/databricks-datasets/learning-spark-v2/cctvVideos/train_images/"
binary_files_df = (spark.read.format("binaryFile")
  .option("pathGlobFilter", "*.jpg")
  .load(path))
binary_files_df.show(5)
```

```
+--------------------+------------------+------+--------------------+-----+
| path               |   modificationTime|length| content            |label|
+--------------------+------------------+------+--------------------+-----+
|file:/Users/jules...|2020-02-12 12:04:24| 55037|[FF D8 FF E0 00 1...|    0|
|file:/Users/jules...|2020-02-12 12:04:24| 54634|[FF D8 FF E0 00 1...|    1|
|file:/Users/jules...|2020-02-12 12:04:24| 54624|[FF D8 FF E0 00 1...|    0|
|file:/Users/jules...|2020-02-12 12:04:24| 54505|[FF D8 FF E0 00 1...|    0|
|file:/Users/jules...|2020-02-12 12:04:24| 54475|[FF D8 FF E0 00 1...|    0|
+--------------------+------------------+------+--------------------+-----+
only showing top 5 rows
```

Aby zignorować odkrywanie partycjonowania danych w katalogu, opcji `recursiveFileLookup` można przypisać wartość `"true"`.

```
// W kodzie Scali.
val binaryFilesDF = spark.read.format("binaryFile")
  .option("pathGlobFilter", "*.jpg")
  .option("recursiveFileLookup", "true")
  .load(path)
binaryFilesDF.show(5)

# W kodzie Pythona.
binary_files_df = (spark.read.format("binaryFile")
  .option("pathGlobFilter", "*.jpg")
```

```
  .option("recursiveFileLookup", "true")
  .load(path))
binary_files_df.show(5)

+--------------------+-------------------+------+--------------------+
| path               |   modificationTime|length| content            |
+--------------------+-------------------+------+--------------------+
|file:/Users/jules...|2020-02-12 12:04:24| 55037|[FF D8 FF E0 00 1...|
|file:/Users/jules...|2020-02-12 12:04:24| 54634|[FF D8 FF E0 00 1...|
|file:/Users/jules...|2020-02-12 12:04:24| 54624|[FF D8 FF E0 00 1...|
|file:/Users/jules...|2020-02-12 12:04:24| 54505|[FF D8 FF E0 00 1...|
|file:/Users/jules...|2020-02-12 12:04:24| 54475|[FF D8 FF E0 00 1...|
+--------------------+-------------------+------+--------------------+
only showing top 5 rows
```

Zwróć uwagę na brak kolumny `label`, w przypadku gdy opcji `recursiveFileLookup` została przypisana wartość `"true"`.

Obecnie źródło danych w postaci pliku binarnego nie obsługuje możliwości zapisu egzemplarza DataFrame z powrotem do pliku binarnego.

W tym podrozdziale wyjaśniliśmy, jak można wczytywać dane do egzemplarza DataFrame i zapisywać ten egzemplarz w wielu różnych obsługiwanych formatach plików. Omówiliśmy również tworzenie widoków tymczasowych i tabel na podstawie istniejących wbudowanych źródeł danych. Niezależnie od tego, czy używasz API DataFrame, czy kodu SQL, przygotowane zapytania prowadzą do wygenerowania dokładnie takich samych wyników. Niektóre z tych zapytań możesz przeanalizować w notatniku, który znajdziesz w materiałach do tej książki (*ftp://ftp.helion.pl/przyklady/sparb2.zip*).

Podsumowanie

W tym rozdziale omówiliśmy temat współpracy między API DataFrame i silnikiem Spark SQL. Przede wszystkim pokazaliśmy, jak można używać silnika Spark SQL do:

- tworzenia tabel zarządzanych i niezarządzanych za pomocą Spark SQL i API DataFrame;
- odczytywania danych z różnych wbudowanych źródeł danych i formatów plików i zapisywania ich;
- korzystania z interfejsu programistycznego `spark.sql` w celu wykonywania zapytań SQL dotyczących danych strukturalnych, przechowywanych w postaci widoków i tabel SQL;
- posługiwania sią egzemplarzem `Catalog` w Sparku w celu analizowania metadanych powiązanych z tabelami i widokami;
- stosowania API `DataFrameWriter` i `DataFrameReader`.

Na podstawie przykładowych fragmentów kodu zaprezentowanych w tym rozdziale i notatników zamieszczonych w materiałach do tej książki (*ftp://ftp.helion.pl/przyklady/sparb2.zip*) możesz zobaczyć, jak używać DataFrame i Spark SQL. Będziemy kontynuować ten wątek. W następnym rozdziale wyjaśnimy, jak Spark współdziała z zewnętrznymi źródłami danych, które zostały pokazane na rysunku 4.1 umieszczonym na początku rozdziału. Zapoznasz się z bardziej zaawansowanymi przykładami transformacji i współpracy między API DataFrame i silnikiem Spark SQL.

Spark: twórz skalowalne i niezawodne aplikacje big data!

Apache Spark jest oprogramowaniem open source, przeznaczonym do klastrowego przetwarzania danych dostarczanych w różnych formatach. Pozwala na uzyskanie niespotykanej wydajności, umożliwia też pracę w trybie wsadowym i strumieniowym. Framework ten jest również świetnie przygotowany do uruchamiania złożonych aplikacji, włączając w to algorytmy uczenia maszynowego czy analizy predykcyjnej. To wszystko sprawia, że Apache Spark stanowi znakomity wybór dla programistów zajmujących się big data, a także eksploracją i analizą danych.

To książka przeznaczona dla inżynierów danych i programistów, którzy chcą za pomocą Sparka przeprowadzać skomplikowane analizy danych i korzystać z algorytmów uczenia maszynowego, nawet jeśli te dane pochodzą z różnych źródeł. Wyjaśniono tu, jak dzięki Apache Spark można odczytywać i ujednolicać duże zbiory informacji, aby powstawały niezawodne jeziora danych, w jaki sposób wykonuje się interaktywne zapytania SQL, a także jak tworzy się potoki przy użyciu MLlib i wdraża modele za pomocą biblioteki MLflow. Omówiono również współdziałanie aplikacji Sparka z jego rozproszonymi komponentami i tryby jej wdrażania w poszczególnych środowiskach.

W książce:

- API strukturalne dla Pythona, SQL, Scali i Javy
- operacje Sparka i silnika SQL
- konfiguracje Sparka i interfejs Spark UI
- nawiązywanie połączeń ze źródłami danych: JSON, Parquet, CSV, Avro, ORC, Hive, S3 i Kafka
- operacje analityczne na danych wsadowych i strumieniowanych
- niezawodne potoki danych i potoki uczenia maszynowego

Jules S. Damji jest inżynierem oprogramowania dla wielu wiodących firm, takich jak Netscape, Sun Microsystems, Verisign i ProQuest. Zajmuje się systemami rozproszonymi.
Brooke Wenig kieruje zespołem, który opracowuje potoki uczenia maszynowego. Prowadzi też szkolenia z zakresu rozproszonego uczenia maszynowego.
Tathagata Das jest członkiem Apache Spark Project Management Committee. Pracuje nad strumieniowaniem strukturalnym i Delta Lake.
Denny Lee zajmuje się systemami rozproszonymi i inżynierią danych, zwłaszcza dla branży ochrony zdrowia.

Helion
helion.pl
HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl
KOD KORZYŚCI
Sięgnij po więcej!
ISBN 978-83-283-9914-3
9 788328 399143
Cena: 89,00 zł