Spark SQL DataFrame Operations

Contents

1. Basic Operations

- Introduction
- Sample DataFrames
- Obtaining metadata
- Caching and persistence
- Actions
- Additional operations

- Demo code for this section:
  - `basic.py`

Introduction

- The previous chapter introduced the `sql()` method in the `SQLContext` and `HiveContext` classes
  - Allows you to process a dataset using SQL or HiveQL statements
- The `DataFrame` API provides an alternative set of operations for processing a dataset…
  - Basic operations
  - Language-integrated query operations
  - RDD operations
  - Actions
  - Output operations
- We'll explore this API in detail in this chapter

Sample DataFrames

- The demos in this section are in `basic.py`
  - The first part of the code creates Spark objects and sets up some sample `DataFrame` objects

```
from pyspark import SparkContext, StorageLevel
from pyspark.sql import SQLContext, SparkSession
from entities import Customer, Product, Home

sc = SparkContext()
sparkSession = SparkSession(sc)
sqlContext = SQLContext(sc)

customers = [ … … … ]
customerDF = sc.parallelize(customers).toDF()

products = [ … … … ]
productDF = sc.parallelize(products).toDF()

homes = [ … … … ]
homeDF = sc.parallelize(homes).toDF()

# The rest of the code illustrates common DataFrame methods
… … …
```

basic.py

Obtaining Metadata

- `columns`
  - Returns the names of all the columns, as an array of strings
- `dtypes`
  - Returns the data types of the columns, as an array of tuples (each tuple contains the column name and data-type)
- `explain()`
  - Prints the physical plan on the console (useful for debugging)
  - If you pass the parameter `True`, it also prints the logical plan
- `printSchema()`
  - Prints the schema on the console, in a tree format

Caching and Persistence

- `cache()`
  - Stores the `DataFrame` in memory, using a columnar format
  - It scans only the required columns and stores them in compressed in-memory columnar format
  - You can tune the caching functionality via the `setConf()` method in `SQLContext` or `HiveContext`

- `persist()`
  - Persists the DataFrame in memory, or on disk
  - You can specify the storage level for a persisted `DataFrame`, just like you can for a persisted RDD

Actions (1 of 2)

- `count()`
  - Returns the number of rows in the `DataFrame`
- `first()`
  - Returns the first row in the `DataFrame`, as a `Row`
- `collect()`
  - Returns all rows in the `DataFrame`, as an array of `Row`
- `take(numberOfRows)`
  - Returns the first N rows in the `DataFrame`, as an array of `Row`

Actions (2 of 2)

- `show(optionalNumberOfRows)`
  - Displays the rows in the `DataFrame` (top 20 by default)

- `describe(columnNames)`
  - Returns summary stats for numeric columns in the `DataFrame`

Additional Operations

- `registerTempTable(tableName)`
  - Creates a temporary table in Hive MetaStore
  - You can then query the table using the `sql()` method in `SQLContext` and `HiveContext`
  - Note that the temporary table is only during the lifespan of the application that creates it

- `toDF()`
  - Allows you to rename the columns in the `DataFrame`
  - Takes the new names of the columns as arguments and returns a new `DataFrame`

2. Language-Integrated Query Operations

- Introduction
- SQL-like operations
- Selection operations
- Analysis operations
- Set-based operations

- Demo code for this section:
  - `languageIntegratedQueries.py`

Introduction

- The `DataFrame` class has properties and methods for accessing data in a `DataFrame` as `Column` objects

- To get a `Column` object, you can just apply the column name to the `DataFrame` object
  - See the "Accessing columns" part of the demo

```
productDF.price
```

SQL-Like Operations (1 of 2)

- `filter(filterExpression)`
  - Returns a `DataFrame` containing just the filtered rows
- `distinct()`
  - Returns a `DataFrame` without duplicate tuples
- `limit(number)`
  - Returns a `DataFrame` containing the specified number of rows
- `orderBy(columns)`
  - Returns a `DataFrame` ordered by the specified column(s)

SQL-Like Operations (2 of 2)

- `agg(aggregateExpressions)`
  - Returns a new `DataFrame`, containing the results of the specified aggregation(s)

- `groupBy(columns)`
  - Returns a `GroupedData` object, containing rows grouped by the specified column(s)
  - You can perform aggregations on the grouped data - see the `GroupedData` class for a list of available aggregations

Selection Operations

- `select(columns)`
  - Returns a `DataFrame` containing only the specified column(s), making use of built-in functions if you like
- `selectExpr(sql expressions)`
  - Returns a `DataFrame` containing the result of executing the specified SQL expression(s)
- `sample(sampleWithReplacement,fraction)`
  - Returns a `DataFrame` containing the specified fraction of rows
- `randomSplit(weightingFactorArray)`
  - Returns a `DataFrame` array, created by splitting from the source `DataFrame` according to the specified weighting factors

Analysis Operations

- `cube(columnNames)`
  - Takes the names of one or more columns and returns a cube containing aggregates of all possible combinations of dimensions
  - Useful for generating cross-tabular reports

- `rollup(columnNames)`
  - Takes the names of one or more columns and returns a multi-dimensional rollup
  - Useful for sub-aggregation along a hierarchical dimension, e.g. location or time

Set-Based Operations

- `intersect(otherDataFrame)`
  - Joins another `DataFrame` and performs a intersect operation
  - Returns a new `DataFrame` containing the intersect results

- `join(otherDataFrame,joinExpresssions,type)`
  - Joins another `DataFrame` and performs a join operation
  - You can optionally specify join expressions
  - You can optionally specify the join type (e.g. "inner", "outer", etc.)
  - Returns a new `DataFrame` containing the join results

3. RDD Operations

- Converting a DataFrame to an RDD
- Converting a DataFrame to an RDD of JSON

- Demo code for this section:
  - `rdd.py`

Converting a DataFrame to an RDD

- The `DataFrame` class supports commonly used RDD operations
  - E.g. `map()`, `flatMap()`, `foreach()`, `foreachPartition()`
  - These methods work similar to the corresponding RDD operations

- If you need RDD functionality not present in `DataFrame`, you can get an RDD from a `DataFrame`
  - Via the `rdd` property
  - Returns an RDD of `Row` objects
  - Each `Row` represents a tuple in the source `DataFrame`

Converting a DataFrame to an RDD of JSON

- You can also get an RDD of JSON strings from a `DataFrame`
  - Via the `toJSON()` method
  - Returns an RDD of `JSON` objects

4. Output Operations

- Overview
- Saving a DataFrame in a JSON file
- Saving a DataFrame in an RDBMS
- Saving a DataFrame in a Parquet file
- Saving a DataFrame in a Hive table

- Demo code for this section:
  - `output.py`

Overview

- You can write a `DataFrame` to a data source
  - The first step is to get a `DataFrameWriter`
  - Via the `write()` method in `SQLContext` and `HiveContext`

```
dataFrameWriter = sqlContext.write()
```

- `DataFrameWriter` has methods for writing the contents of a DataFrame to a data source, such as:
  - JSON, relational database, Parquet, Hive

- You can write to any Hadoop-supported storage system
  - E.g. HDFS, local file system, Amazon S3

Saving a DataFrame in a JSON File

- Here's how to save a `DataFrame` to a JSON file
  - The `DataFrameWriter` class has a `json()` method
  - Creates a directory with the specified name
  - The directory contains partitioned data

```
customerDF.write.json("customerJson")
```

```
{"cId":1,"name":"David","age":21,"gender":"M"}
{"cId":2,"name":"Lydia","age":22,"gender":"F"}
{"cId":3,"name":"Peter","age":23,"gender":"M"}
{"cId":4,"name":"Frank","age":24,"gender":"F"}
{"cId":5,"name":"Benny","age":25,"gender":"M"}
```

Saving a DataFrame in an RDBMS

- Here's how to save a `DataFrame` to a table in an RDBMS
  - The `DataFrameWriter` class has a `jdbc()` method
  - Saves the `DataFrame` to the specified table

```
# Specify JDBC connection info.
jdbcUrl    = "jdbc:derby://localhost:1527/C:/BigData/MyDatabase"
tableName  = "MySchema.Employees"
properties = { "driver": "org.apache.derby.jdbc.ClientDriver"}

# Write DataFrame to the table.
customerDF.write.jdbc(jdbcUrl, tableName, properties)
```

- How to run the demo:
  - First start the Derby database engine, via `StartDerby.bat`
  - Then run the demo, with the JDBC driver on the classpath

```
spark-submit --jars \BigData\Libraries\derby\lib\derbyclient.jar output.py
```

Saving a DataFrame in a Parquet File

- Here's how to save a `DataFrame` to a Parquet file
  - The `DataFrameWriter` class has a `parquet()` method
  - Creates a directory with the specified name
  - The directory contains the Parquet file (it's binary!)

```
customerDF.write.parquet("customerParquet")
```

Saving a DataFrame in a Hive Table

- Here's how to save a `DataFrame` to a Hive table
  - The `DataFrameWriter` class has a `saveAsTable()` method
  - Writes the data to the specified table name in the Hive warehouse
  - The data is in Parquet format

```
customerDF.write.saveAsTable("customer")
```

Any Questions?