Getting Started with Spark SQL

Contents

1. Overview of Spark SQL

- Introducing Spark SQL
- Spark SQL query languages
- Spark SQL performance benefits
- Usage scenarios

Introducing Spark SQL

- Spark SQL is a Spark library that runs on top of Spark
  - Provides a higher-level abstraction than Spark Core, for processing structured data
- Spark SQL supports a wide range of data sources
  - Relational databases, e.g. Oracle etc.
  - NoSQL data sources, e.g. Cassandra
  - Text formats, e.g. JSON, CSV
  - Binary formats, e.g. Avro
  - Data warehousing systems, e.g. Hive
  - Columnar storage systems, e.g. Parquet
- Spark SQL provides a uniform API over all these data sources
  - This is a huge simplification for developers!

Spark SQL Query Languages

- Spark SQL supports three query languages…
  - Spark SQL translates queries written in these languages into Spark Core API calls
- SQL
  - Ubiquitous, plentiful development skills available
- HiveQL
  - Similar to SQL, but more expressive
  - HiveQL runs much faster on Spark SQL than on Hive
- Language integrated queries
  - Embed your queries directly in your Python code
  - No need to know SQL (or HiveQL)

Spark SQL Performance Benefits

- Predicate pushdown
  - Where possible, Spark SQL pushes predicates (e.g. WHERE clauses) down to the data source layer
- Columnar storage
  - Spark SQL supports columnar storage formats, e.g. Parquet
- In-memory columnar storage
  - Spark SQL allows an application to cache data in an in-memory columnar format from any data source
- Skip rows
  - Where available, Spark SQL can utilize statistical info about a dataset (e.g. Parquet stores min/max values for each column)

Usage Scenarios (1 of 2)

- Extract Transform Load (ETL)
  - E = Extract data from various operational data sources
  - T = Transform data, e.g. cleanse and modify the data
  - L = Load data, e.g. into a destination database or file
  - Spark SQL is well suited to ETL apps, because of its support for multiple data sources and its high-performance compute engine
- Data virtualization
  - Collect data from heterogeneous data sources, and provide a uniform view via SQL, HiveQL, or language integrated queries
- Distributed JDBC/ODBC SQL query engine
  - Spark SQL has a pre-packaged distributed SQL query server engine
  - Client apps can submit queries to be executed on this engine

Usage Scenarios (2 of 2)

- Spark SQL can be used to build an open source data warehousing solution
  - You can mix-and-match different components in the data warehousing stack
  - Provides a scalable, economical, and flexible alternative to proprietary data warehousing solutions

- For example:
  - Data can originate from various data sources
  - The Spark SQL distributed query engine can execute queries
  - Data can be stored in HDFS

2. Getting Started with the Spark SQL API

- Context classes in Spark SQL
- Creating a SqlContext or HiveContext
- Representing data in Spark SQL

Context Classes in Spark SQL

- `SQLContext`
  - This is the main entry-point into the Spark SQL library
  - Allows you to create all the other objects you'll need

- `HiveContext`
  - Extends `SQLContext`, provides various additional features
  - Provides a HiveQL parser (more powerful than SQL)
  - You must use `HiveContext` for processing data in Hive
  - You can also use `HiveContext` for processing non-Hive data

- For full documentation on Spark SQL in Python, see:
  - https://spark.apache.org/docs/2.2.0/api/python/pyspark.sql.html

Creating a SqlContext or HiveContext

- Your application must either create a `SQLContext` or `SQLContext` object
  - … based on an existing `SparkContext` object

```
from pyspark.sql import SQLContext
…

sqlContext = SQLContext(sc)
```

```
from pyspark.sql import HiveContext
…

hiveContext = HiveContext(sc)
```

- Note:
  - `SqlContext` has a `sql()` method, supports SQL
  - `HiveContext` has a `sql()` method, supports HiveQL and SQL

Representing Data in Spark SQL

- `DataFrame`
  - Represents a distributed collection of rows in named columns
  - Conceptually similar to a table in a relational database
  - Supports common operations, e.g. selecting columns, filtering rows, aggregating columns, joining tables, etc.

- `DataFrame` is schema-aware (unlike RDD)
  - An RDD is a partitioned collection of opaque elements…
  - Whereas a `DataFrame` knows the names and types of the columns in a dataset

- The `DataFrame` is easier to understand than the RDD API
  - You can always dip down to the RDD API if necessary
  - You can also create a `DataFrame` from an existing RDD

3. Creating DataFrames from a Data Source

- Overview
- Creating a DataFrame from a JSON file
- Creating a DataFrame from an RDBMS
- Creating a DataFrame from a Parquet file
- Creating a DataFrame from a Hive table

Overview (1 of 2)

- There are two ways to create a `DataFrame`...
- You can create a `DataFrame` from a data source
  - Spark SQL provides builder methods for creating a `DataFrame` from a variety of data sources
  - Some of the data sources have built-in support, and others require external libraries
  - We'll discuss this approach in this section
- You can create a `DataFrame` can be created from an RDD
  - This allows you to use higher-order `DataFrame` APIs
  - Simplifies code, utilizes schema knowledge
  - We'll discuss this approach in the next section

Overview (2 of 2)

- Spark SQL has built-in support for many of the commonly used data sources
  - E.g. JSON, Hive, JDBC-compliant databases, Parquet, Cassandra
  - External packages are available for other data sources

- To start reading data from a data source, Spark SQL provides a class named `DataFrameReader`
  - This class has builder methods that allow you to specify different options for reading data (format, partitioning, etc.)
  - You obtain a `DataFrameReader` instance via the `read` method in `SQLContext` and `HiveContext`

```
dataFrameReader = sqlContext.read()
```

Creating a DataFrame from a JSON File (1 of 3)

- You can read data directly from a JSON file
  - The JSON file must have one JSON object per line (otherwise Spark SQL will fail to load it)

- Here's a sample JSON file for our examples

```
{"id" : 101, "name": "Simon Putz", "age" : 41, "salary": 1000}
{"id" : 102, "name": "Julie Evan", "age" : 42, "salary": 2000}
{"id" : 103, "name": "Micky Bale", "age" : 43, "salary": 3000}
{"id" : 104, "name": "Sarah Hill", "age" : 44, "salary": 4000}
```

`json/employees.json`

Creating a DataFrame from a JSON File (2 of 2)

- Here's how to create a `DataFrame` from a JSON file
  - The `DataFrameReader` class has a `json()` method

```
from pyspark import SparkContext
from pyspark.sql import SQLContext

# Create a SQLContext object.
sc = SparkContext()
sqlContext = SQLContext(sc)

# Create a DataFrame containing data read in from a json file.
employeeDF = sqlContext.read.json("employees.json")

# Register a temporary table.
employeeDF.registerTempTable("employee")

# Select fields from the table.
result = sqlContext.sql("FROM employee SELECT id, name, age, salary")

# Display the result.
result.show()
```

json/jsondemo.py

- Note the `registerTempTable()` method call
  - Creates a temporary table in Hive MetaStore
  - Allows us to execute SQL statements on the data - see later

Creating a DataFrame from a JSON File (3 of 3)

- Spark SQL automatically infers the schema of a JSON dataset
  - It scans the entire dataset once, to determine the schema

- If you know the schema, you can tell the `DataFrame`
  - Define the schema using `StructType()` and `StructField()`
  - Pass it into the `schema()` method on `DataFrameReader`

```
from pyspark.sql.types import StructType, StructField, IntegerType, StringType
…

employeeSchema = StructType([
     StructField("id", IntegerType(), False),
     StructField("name", StringType(), False),
     StructField("age", IntegerType(), False),
     StructField("salary", IntegerType(), False)
])

employeeDF = sqlContext.read.schema(employeeSchema).json("employees.json")
```

`json/jsondemoWithSchema.py`

Creating a DataFrame from an RDMBS (1 of 3)

- You can read data directly from a relational database
  - Any relational db - you just need the JDBC driver on the classpath
- We're using the Derby database engine
  - An open-source, free, pure Java Database Management System
  - Downloadable from `http://db.apache.org/derby`
  - We've downloaded/unzipped to `\BigData\Libraries\derby`
- To start the Derby engine:
  - Run `\BigData\Demos\05-SparkSQL\startDerby.bat`

Creating a DataFrame from an RDMBS (2 of 3)

- We've provided a simple Derby database
  - See `\BigData\MyDatabase`
  - Contains simple tables, e.g. `MySchema.employees`

- To access a relational database in Spark...
  - You must add the JDBC driver to the classpath
  - You can do this as follows when you start the Spark Shell

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

Creating a DataFrame from an RDMBS (3 of 3)

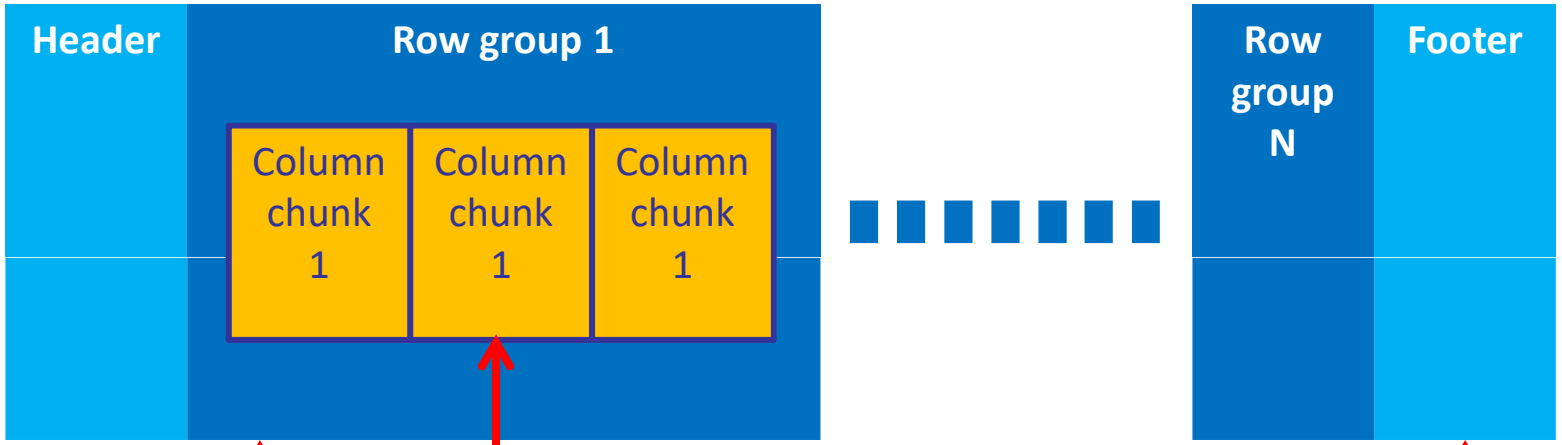
- Here's how to create a `DataFrame` from a n RDBMS table
  - The `DataFrameReader` class has a `jdbc()` method
  - Returns a `DataFrame` object that can read data from a table

```
from pyspark.sql import SQLContext
…

jdbcUrl    = "jdbc:derby://localhost:1527/C:/BigData/MyDatabase"
tableName  = "MySchema.Employees"
properties = {
    "driver":   "org.apache.derby.jdbc.ClientDriver"
#   "user":     "someUser",
#   "password": "somePassword"
}

# Create a DataFrame containing data read in from a table in a relational database.
jdbcDF = sqlContext.read.jdbc(jdbcUrl, tableName, properties)

# Register a temporary table.
jdbcDF.registerTempTable("employee")

# Select fields from the table.
result = sqlContext.sql("FROM employee SELECT EMPLOYEEID, NAME, SALARY, REGION")

# Display the result.
result.show()
```

rdbms/rdbmsdemo.py

Creating a DataFrame from a Parquet File (1 of 3)

- Parquet is a binary columnar format, supported by many data processing systems



A row group stores all the column values for a range of rows in a columnar layout

A column chunk contains all the values for a particular column in the row group

The footer contains schema details, plus metadata about row groups and columns

Creating a DataFrame from a Parquet File (2 of 3)

- Spark SQL provides support for both reading and writing parquet files
  - Automatically captures the schema of the original data, from the footer in the Parquet file

- Processing Parquet files is similar to dealing with JSON datasets

Creating a DataFrame from a Parquet File (3 of 3)

- Here's how to create a `DataFrame` from a Parquet file
  - The `DataFrameReader` class has a `parquet()` method
  - Returns a `DataFrame` object that can read data from Parquet file
  - (Note that we create the Parquet file from a JSON file initially)

```
from pyspark.sql import SQLContext
…

# Create a DataFrame from a JSON file, then write out in Parquet format.
jsonDF = sqlContext.read.json("employees.json")
jsonDF.write.parquet("employees.parquet")

# Create a DataFrame containing data read in from Parquet.
parquetDF = sqlContext.read.parquet("employees.parquet")

# Register a temporary table.
parquetDF.registerTempTable("employee")

# Select fields from the table.
result = sqlContext.sql("SELECT * FROM employee")

# Display the result.
result.show()
```

parquet/parquetdemo.py

Creating a DataFrame from a Hive Table (1 of 3)

- Hive comes bundled with the Spark SQL library, via the `HiveContext` class
  - Inherits from `SQLContext`
- Using `HiveContext`, you can create and find tables in Hive and write queries using HiveQL
  - HiveQL is based on SQL…
  - But it doesn't strictly follow the full SQL-92 standard
- For full details about HiveQL, see:
  - https://cwiki.apache.org/confluence/display/Hive/LanguageManual

Creating a DataFrame from a Hive Table (2 of 3)

- If you want to process Hive tables:
  - Define a `hive-site.xml` file, and put it in Spark's `conf` folder
  - `HiveContext` reads Hive configuration from this file

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<configuration>
    <property>
        <name>hive.metastore.warehouse.dir</name>
        <value>/usr/local/hiveMetastore</value>
        <description>Local or HDFS directory for storing tables.</description>
     </property>
     <property>
        <name>javax.jdo.option.ConnectionURL</name>
        <value>jdbc:derby:;databaseName=/usr/local/hiveMetastore;create=true</value>
        <description>JDBC connection URL.</description>
     </property>
</configuration>
```

hive-site.xml

- If file is missing, Hive creates a `metastore_db` directory in the current directory

Creating a DataFrame from a Hive Table (3 of 3)

- Here's how to create a `DataFrame` from a Hive table
  - First we use HiveQL to create a table in the Hive warehouse
  - Then we use HiveQL to load data into the table (from a text file)
  - Then we execute any kind of SQL queries on the table

```
from pyspark import SparkContext
from pyspark.sql import HiveContext

# Create a SQLContext object.
sc = SparkContext()
sqlContext = HiveContext(sc)

# Create Table using HiveQL.
sqlContext.sql(
  "CREATE TABLE IF NOT EXISTS employee(id INT, name STRING, age INT, salary INT) " +
  "ROW FORMAT DELIMITED FIELDS TERMINATED BY ',' " +
  "LINES TERMINATED BY '\n'")

# Load data into table using HiveQL.
sqlContext.sql("LOAD DATA LOCAL INPATH 'employees.txt' INTO TABLE employee")

# Select fields from the table.
result = sqlContext.sql("FROM employee SELECT id, name, age, salary")

# Display the result.
result.show()
```

hive/hivedemo.py

4. Creating DataFrames from an RDD

- Overview
- Using the `toDF()` method
- Using the `createDataFrame()` method

Overview

- The previous section showed how to create a `DataFrame` directly from a data source
  - E.g. JSON, RDBMS, Parquet, Hive, etc.
- It's also possible to create a `DataFrame` from an RDD
  - This is a very realistic scenario
  - E.g. you've already read RDD data elsewhere in your app…
  - And you want to use the higher-level API provided by Spark SQL
- Spark SQL has 2 methods for creating a `DataFrame` from an RDD
  - `toDF()`
  - `createDateFrame()`

Using the toDF() Method (1 of 3)

- The `toDF()` method creates a `DataFrame` from an RDD of objects represented by a case class
  - Spark SQL infers the schema of the dataset from the case class

- The `toDF()` method is not defined in the RDD class
  - It's a "monkey patch" executed inside the `SparkSession` ctor
  - So you must create a `SparkSession` object in your code

```
from pyspark.sql import SparkSession
…

sparksession = SparkSession(sc)
```

Using the toDF() Method (2 of 3)

- This example uses `toDF()` to create a `DataFrame` from an in-memory collection

```
from pyspark import SparkContext
from pyspark.sql import SQLContext, SparkSession
from employee import Employee

# Create a collection of data.
employees = [
    Employee(101, "Simon Putz", 41, 1000),
    Employee(102, "Julie Evan", 42, 2000),
    Employee(103, "Micky Bale", 43, 3000),
    Employee(104, "Sarah Hill", 44, 4000)
]

# Create an RDD on the data.
employeeRDD = sc.parallelize(employees)

# Create a SparkSession, which adds toDF() method to RDD.
sparksession = SparkSession(sc)

# Create a Spark SQL DataFrame from the RDD.
employeeDF = employeeRDD.toDF()
```

rdd/toDF1.py

Using the toDF() Method (3 of 3)

- This example uses `toDF()` to create a `DataFrame` from a CSV file - we create an `Employee` for each row in the file

```
from pyspark import SparkContext
from pyspark.sql import SQLContext, SparkSession
from employee import Employee

# Create Spark objects.
sc = SparkContext()
sparksession = SparkSession(sc)
sqlContext = SQLContext(sc)

# Read data from a CSV file, and map to an RDD.
employeeRDD = sc.textFile("employees.csv") \
                .map(lambda row: row.split(",")) \
                .map(lambda cols: Employee(int(cols[0]),
                                           cols[1],
                                           int(cols[2]),
                                           int(cols[3])))

# Create a Spark SQL DataFrame from the RDD.
employeeDF = employeeRDD.toDF()
```

rdd/toDF2.py

Using the createDataFrame() Method (1 of 3)

- The `createDataFrame()` method creates a `DataFrame` from an RDD of `Row` objects

- `Row` is a Spark SQL class, it behaves like a tuple
  - You can create `Row` objects directly in your code, if you like

```
from pyspark.sql import Row
…

row1 = Row(1,  "Łukasz Fabianski", "Goalkeeper", False)
row2 = Row(6,  "Ashley Williams",  "Defender",   True)
row3 = Row(23, "Gylfi Sigurdsson", "Midfielder", False)
```

Using the createDataFrame() Method (2 of 3)

- The `createDataFrame()` method takes two params
  - An RDD of `Row` objects
  - The row schema, specified via `StructType` and `StructField` from the package `org.apache.spark.sql.types`

```
from pyspark.sql.types import StructType, StructField, IntegerType, StringType
…

someRddOfRows = …     # Some RDD of Row objects.
someRowSchema = …     # Some StructType defining the schema of the Row objects.
…

rowsDF = sqlContext.createDataFrame(someRddOfRows, someRowSchema)
```

Using the createDataFrame() Method (3 of 3)

- Example:

```
from pyspark import SparkContext
from pyspark.sql import SQLContext, Row
from pyspark.sql.types import StructType, StructField, IntegerType, StringType

sc = SparkContext()
sqlContext = SQLContext(sc)

# Read data from a CSV file.
linesRDD = sc.textFile("employees.csv")

# Map to an RDD of Spark SQL rows.
rowsRDD = linesRDD.map(lambda row: row.split(",")) \
                  .map(lambda cols: Row(int(cols[0]),
                                        cols[1],
                                        int(cols[2]),
                                        int(cols[3])))

# Define the schema for employee data.
schema = StructType([
     StructField("id", IntegerType(), False),
     StructField("name", StringType(), False),
     StructField("age", IntegerType(), False),
     StructField("salary", IntegerType(), False)
])

# Read the data into a Spark SQL DataFrame.
rowsDF = sqlContext.createDataFrame(rowsRDD, schema)
```

rdd/createDataFrame.py

Any Questions?