PySpark

PySpark SQL and Data Frames

Agenda

01 Introduction to Spark SQL

02 Spark SQL Architecture

03 Components of SQL

04 User-defined Functions

05 Spark DataFrame Features

06 Need for Spark DataFrames

07 DataFrame Operations

08 SQL Context in Spark SQL

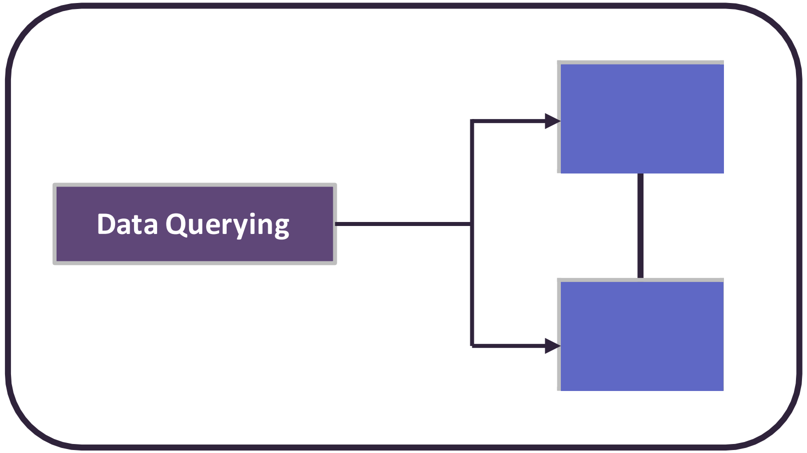
09 Performance Tuning

10 Integrating Spark and Hive

Introduction to **Spark SQL**

Introduction to Spark SQL

Spark SQL is used to integrate relational processing with the functional programming API of Spark



Spark SQL works with **semi-structured** and **structured** data

Introduction to Spark SQL

- Spark SQL is a Spark module for **structured data processing**
- It lets you query structured data inside Spark programs, using either SQL or DataFrame API
- Coding for Spark SQL can be done in Java, Scala, Python, or R
- Spark SQL can also be used to query data from heterogeneous sources like CSV, JSON, Hive, Cassandra, etc.

Need for Spark SQL

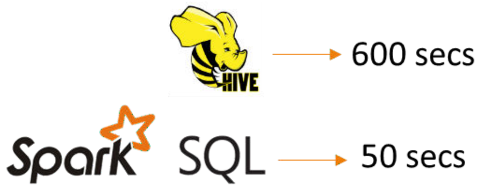
Spark SQL was built to overcome the limitations of Apache Hive running on top of Spark

Limitations of Apache Hive

- Hive uses MapReduce that **lags in performance** with medium- and small-sized datasets (< 200 GB)
- Hive has **no resume** capability
- Hive **cannot drop** encrypted databases

Advantages of Spark SQL

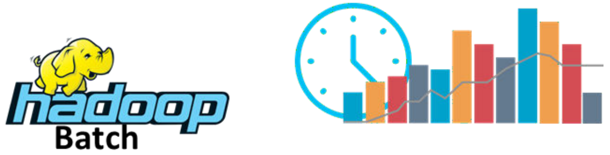
Faster execution



No migration hurdles

HIVE
Spark SQL

Real-time querying



Spark SQL uses the metastore services of Hive to query the data stored and managed by Hive

Comparing **Hive**, **Impala**, and **Spark SQL**

Comparing Hive, Impala, and Spark SQL

	Apache Hive	Apache Impala	Apache Spark SQL
Users	ETL Developers	Business Analysts	Data Engineers and Data Scientists
Strengths	▪ Built for long-running ETL, data processing, or batch processing ▪ Supports custom file formats ▪ Handles massive ETL sorts with joins	▪ Scales to high concurrency ▪ Supports a high-performance interactive SQL ▪ Compatible with BI tools and skills ▪ Supports Hadoop integration and usability	▪ Easy to embed SQL into Java, Scala, or Python applications ▪ Simple language for common operations ▪ Seamlessly mix SQL and Spark codes within a single application ▪ Automatic performance optimizations

Features of **Spark SQL**

Features of Spark SQL

Integrated

- One can mix SQL queries with Spark programs easily
- Structured data can be queried inside Spark programs using either Spark SQL or a DataFrame API
- Running SQL queries, alongside analytic algorithms, is easy because of this tight integration

Hive Compatibility

- Hive queries can be run as they are as Spark SQL supports HiveQL, along with UDFs (user-defined functions) and Hive SerDes
- This allows one to access the existing Hive warehouses

Features of Spark SQL

Unified Data Access

- Loading and querying data from variety of sources is possible
- Needs only a single interface to work with structured data that the schemaRDDs provide

Performance & Scalability

- To make queries agile using the Spark engine, Spark SQL incorporates a code generator, a cost-based optimizer, and a columnar storage
- Spark has ample information regarding the structure of data, as well as the type of computation being performed which is provided by the interfaces of Spark SQL. This leads to extra optimization from Spark SQL, internally

Architecture of **Spark SQL**

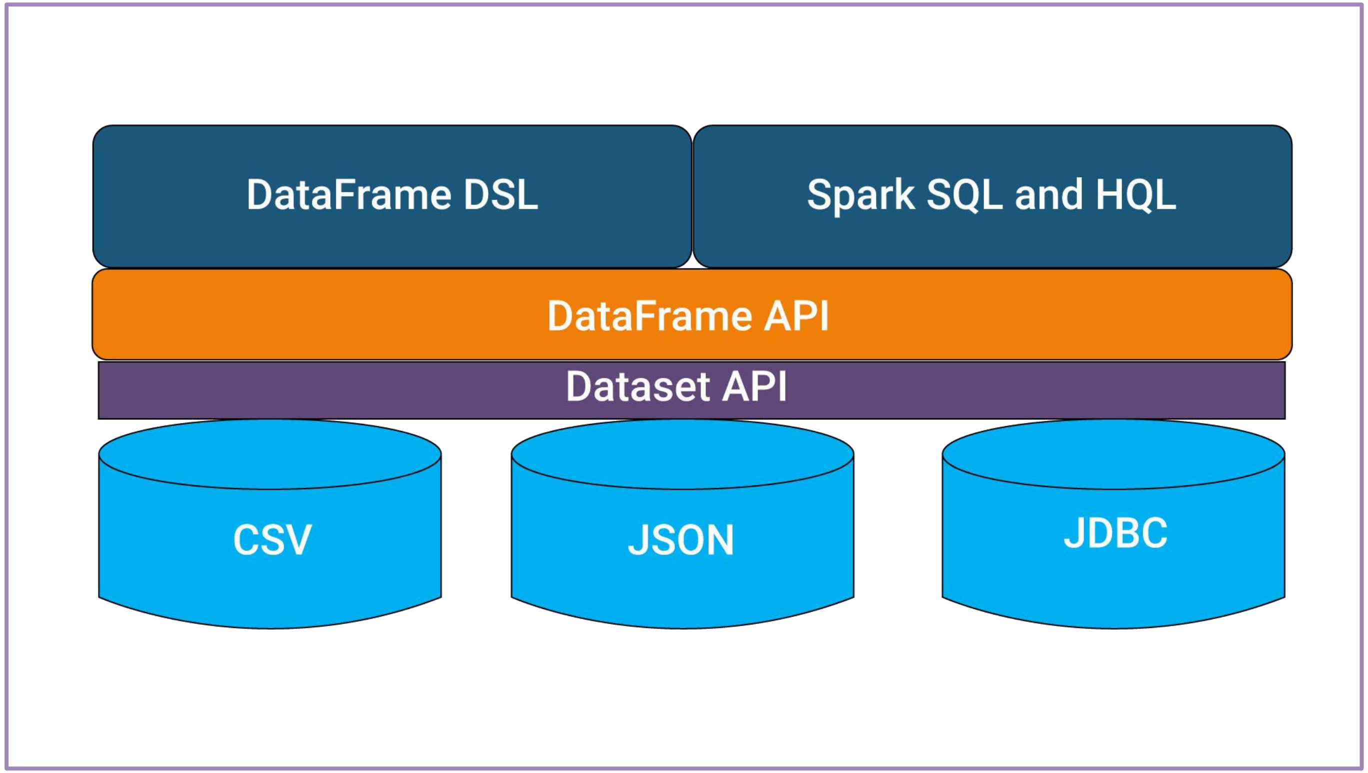
Architecture of Spark SQL

Spark SQL consists of three main layers such as:

IMPORTANT

Language APIs	Spark is compatible and even supported by the languages like Python, HiveQL, Scala, and Java
SchemaRDDs	An RDD of Row objects that has an associated schema. The underlying JVM object is a SchemaRDD, not a PythonRDD, so we can utilize the relational query API exposed by SparkSQL.
Data Sources	For Spark core, the data source is usually a text file, Avro file, etc. Data sources for Spark SQL are different like JSON document, Parquet file, Hive tables, and Cassandra database.

Architecture of Spark SQL



Spark SQL Libraries

Spark SQL Libraries

Spark SQL has four libraries that are used to interact with relational and procedural processing:

- **Data Source API (Application Programming Interface)**
- **DataFrame API**
- **SQL Interpreter and Optimizer**
- **SQL Service**

Data Source API

Data Source API is a universal API for loading and storing structured data

- It has built-in support for Hive, Avro, JSON, JDBC, Parquet, and so on
- It can handle structured/semi-structured data
- It supports third-party integration through Spark packages

DataFrame API

DataFrame API converts the data that is read through the Data Source API into tabular columns to help perform SQL operations

- Processes data in the size ranging from kilobytes to petabytes on single-node or multi-node clusters
- Supports different data formats (Avro, CSV, Elasticsearch, and Cassandra) and storage systems (HDFS, Hive tables, MySQL, etc.)
- A distributed collection of data organized into named columns
- Equivalent to a relational table in SQL
- Lazily evaluated

SQL Interpreter and Optimizer

- It handles the **functional programming** part of Spark SQL
- It transforms DataFrames and RDDs to get the required results in the required formats

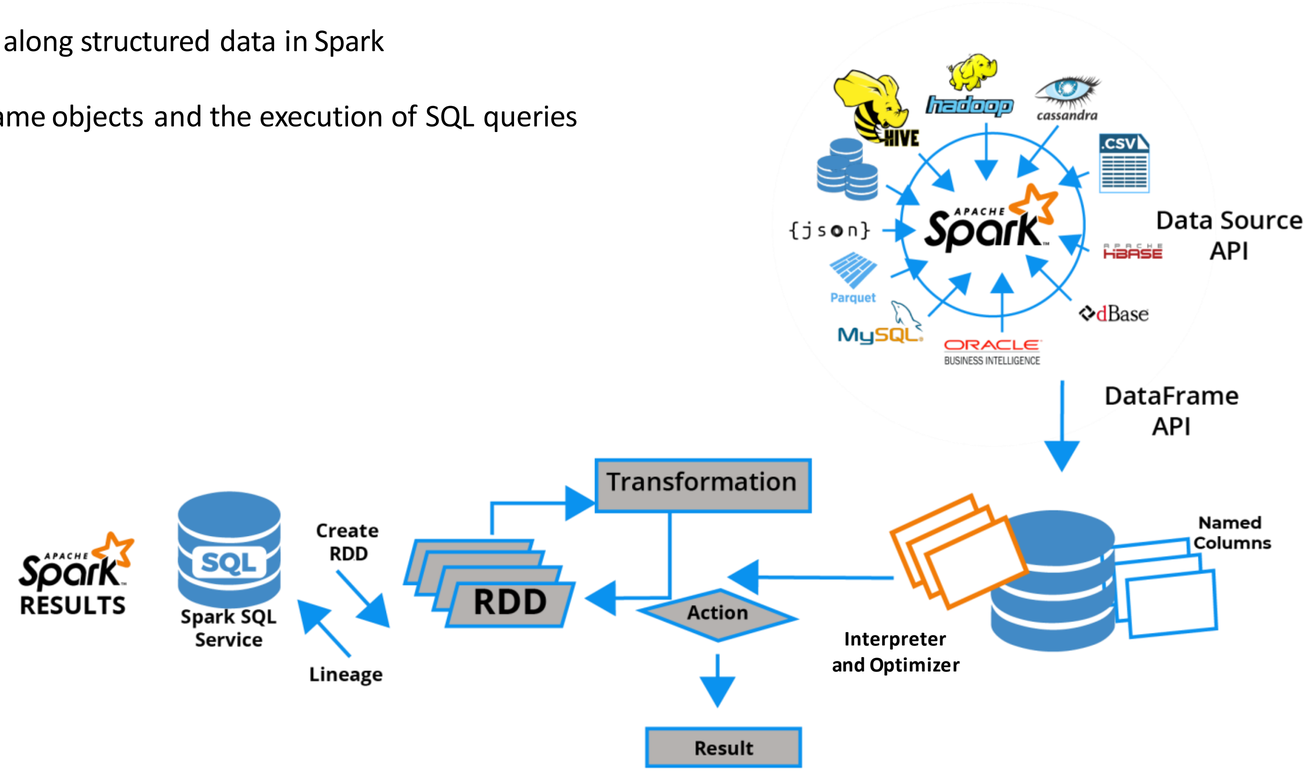
- It provides a general framework for transforming trees, which is used to perform analysis/evaluation, optimization and planning, and runtime code spawning
- It supports cost-based optimization (runtime utilization and resource utilization are termed as cost) and rule-based optimization, making queries run much faster than their RDD (Resilient Distributed Dataset) counterparts

Example:

Catalyst: A modular library for distinct optimization

SQL Interpreter and Optimizer

- It is the **entry point** for working along structured data in Spark
- It allows the **creation** of DataFrame objects and the execution of SQL queries



Use Cases of **Spark SQL**

Use Cases of Spark SQL

Twitter Sentiment Analysis

- Initially, data regarding a topic, say Narendra Modi, used to get gathered from Spark Streaming
- Later, Spark SQL came into the picture to analyze everything about the topic
- Thus, every tweet regarding Modi is gathered, and then Spark SQL does its magic by classifying tweets as neutral tweets, positive tweets, negative tweets, very positive tweets, and very negative tweets
- This is just one of the ways how sentiment analysis is done
- It is useful in target marketing, crisis management, and service adjusting

Use Cases of Spark SQL

Stock Market Analysis

- As you are streaming data in the real time, you can also do the processing in the real time
- Stock movements and market movements generate so much data and traders need an edge, an analytics framework, which will calculate all the data in the real time
- This will provide the most rewarding stock or contract, all within the nick of time
- If there is a need for a real-time analytics framework, then Spark, along with its components, is the technology to be considered

Use Cases of Spark SQL

Banking

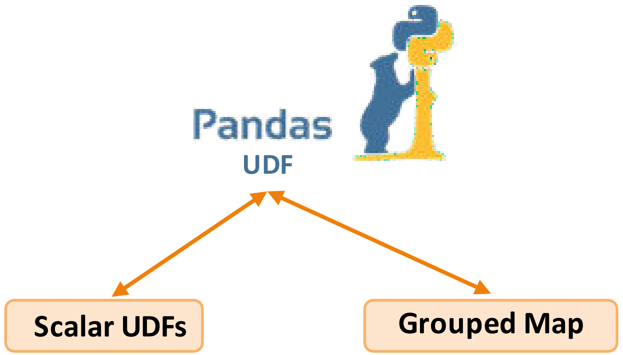
- Real-time processing is required in credit card fraud detection
- Assume that a transaction happens in Bangalore where there is a purchase worth ₹4,000 has been done swiping a credit card
- Within 5 minutes, there is another purchase of ₹10,000 in Kolkata swiping the same credit card
- Banks can make use of real-time analytics provided by Spark SQL for detecting fraud in such cases

User-defined Functions

User-defined Functions

- Pandas UDFs are user-defined functions that are executed by Spark to transfer data and to work with it in the later stages.

- A Pandas UDF is defined using the keyword **pandas_udf** as a decorator or to wrap the function. No additional configuration is required



User-defined Functions

Scalar UDFs

- Scalar Pandas UDFs are used for vectorizing scalar operations
- They can be used with functions such as select and **withColumn**
- The Python function should take **pandas.Series** as inputs and return a **pandas.Series** of the same length
- Internally, Spark will execute a Pandas UDF by splitting columns into batches and calling the function for each batch as a subset of the data, then concatenating the results together

User-defined Functions

Let’s see how to create a scalar Pandas UDF that computes the product of two columns:

```
import pandas as pd

from pyspark.sql.functions import col, pandas_udf
from pyspark.sql.types import LongType

# Declare the function and create the UDF
def multiply_func(a, b):
    return a * b

multiply = pandas_udf(multiply_func, returnType=LongType())
```

User-defined Functions

Let's see how to create a scalar Pandas UDF that computes the product of two columns:

```
# The function for a pandas_udf should be able to execute with local Pandas data
x = pd.Series([1, 2, 3])
print(multiply_func(x, x))
# 0    1
# 1    4
# 2    9
# dtype: int64

# Create a Spark DataFrame, 'spark' is an existing SparkSession
df = spark.createDataFrame(pd.DataFrame(x, columns=["x"]))
```

User-defined Functions

Let’s see how to create a scalar Pandas UDF that computes the product of two columns:

```
# Execute function as a Spark vectorized UDF
df.select(multiply(col("x"), col("x"))).show()
# +-------------------+
# |multiply_func(x, x)|
# +-------------------+
# |                  1|
# |                  4|
# |                  9|
# +-------------------+
```

User-defined Functions

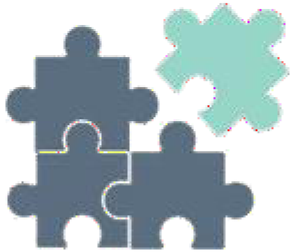
Grouped Map

Grouped Map Pandas UDFs are used with **groupBy().apply()** that implements the **split-apply-combine** pattern. Split-apply-combine consists of three steps:

1. Split the data into groups using **DataFrame.groupBy**

2. Apply a function on each group. The input and output of the function are both **pandas.DataFrame**. The input data contains all rows and columns for each group

3. Combine the results into a **new DataFrame**

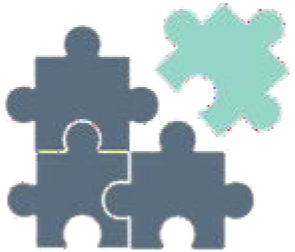


User-defined Functions

Grouped Map

To use **groupBy().apply()**, the user needs to define the following:

- A Python function that defines the computation for each group
- A StructType object or a string that defines the schema of the output DataFrame



Note: All data for a group will be loaded into memory before the function is applied. This can lead to out-of-memory exceptions, especially if the group sizes are skewed. The configuration for **maxRecordsPerBatch** is not applied on groups, and it is up to the user to ensure that the grouped data fits into the available memory

User-defined Functions

Let us see how to use **groupby().apply()** to subtract the mean from each value in the group:

```
from pyspark.sql.functions import pandas_udf, PandasUDFType

df = spark.createDataFrame(
    [(1, 1.0), (1, 2.0), (2, 3.0), (2, 5.0), (2, 10.0)],
    ("id", "v"))

@pandas_udf("id long, v double", PandasUDFType.GROUPED_MAP)
def substract_mean(pdf):
    # pdf is a pandas.DataFrame
    v = pdf.v
    return pdf.assign(v=v - v.mean())

df.groupby("id").apply(substract_mean).show()
# +---+----+
# | id|   v|
# +---+----+
# |  1|-0.5|
# |  1| 0.5|
# |  2|-3.0|
# |  2|-1.0|
# |  2| 4.0|
# +---+----+
```

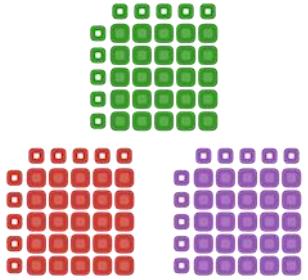
DataFrames

DataFrames

What is a DataFrame?

A DataFrame is an **ordered collection** of information, which is organized into named columns

- It can be compared to **relational tables** with decent optimization techniques
- A DataFrame can be constructed from an array of different sources such as Hive tables, structured data files, external databases, or existing RDDs
- The API was designed for modern Big Data and Data Science applications taking inspiration from DataFrames in R programming and Pandas in Python



DataFrames

It also shares some common characteristics with RDDs:

- **Immutable in nature:** We can create a DataFrame/RDD once but can’t change it. However, we can transform a DataFrame/RDD after applying transformations

- **Lazy evaluation:** In DataFrames, a task is not executed until an action is performed

- **Distributed:** RDDs and DataFrames, both are distributed in nature

Features of DataFrames

- Ability to process data in the size ranging from kilobytes to petabytes on a single-node cluster to a large cluster as well

- Supports different data formats
  - Avro
  - CSV
  - Elasticsearch
  - Cassandra

- Supports different storage systems
  - HDFS
  - Hive tables
  - MySQL

How to create a DataFrame?

- There are many ways to read data into Spark as a DataFrame
- Here, let us load the first few rows of Titanic data on Kaggle into a Pandas DataFrame and then convert it into a Spark DataFrame

TITANIC

```
import findspark
findspark.init()
import pyspark # only run after findspark.init()
from pyspark.sql  import  SparkSession
spark = SparkSession.builder.getOrCreate()
import pandas as pd
sc = spark.sparkContext
```

How to create a DataFrame?

- **Make a sample DataFrame from the Titanic data:**

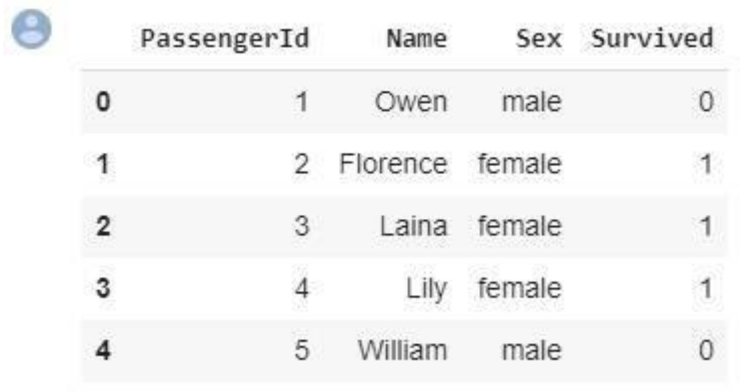
```
data1 = {'PassengerId': {0: 1, 1: 2, 2: 3, 3: 4, 4: 5},
         'Name': {0: 'Owen', 1: 'Florence', 2: 'Laina', 3: 'Lily', 4: 'William'},
         'Sex': {0: 'male', 1: 'female', 2: 'female', 3: 'female', 4: 'male'},
         'Survived': {0: 0, 1: 1, 2: 1, 3: 1, 4: 0}}

data2 = {'PassengerId': {0: 1, 1: 2, 2: 3, 3: 4, 4: 5},
         'Age': {0: 22, 1: 38, 2: 26, 3: 35, 4: 35},
         'Fare': {0: 7.3, 1: 71.3, 2: 7.9, 3: 53.1, 4: 8.0},
         'Pclass': {0: 3, 1: 1, 2: 3, 3: 1, 4: 3}}

df1_pd = pd.DataFrame(data1, columns=data1.keys())
df2_pd = pd.DataFrame(data2, columns=data2.keys())
```

How to create a DataFrame?

- **Print Pandas DataFrame:**

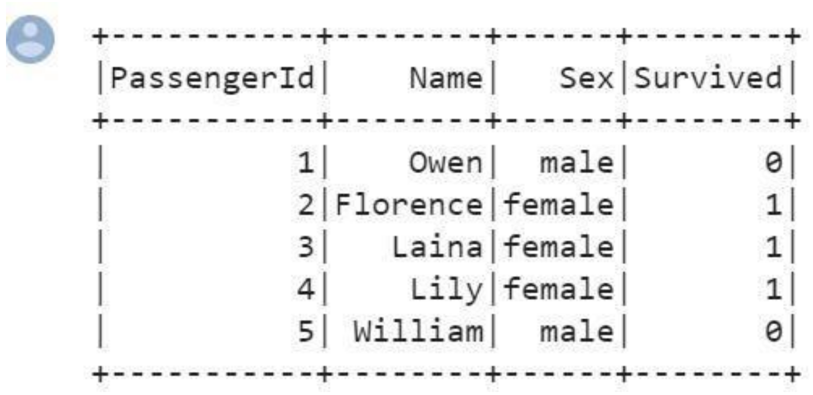


	PassengerId	Name	Sex	Survived
0	1	Owen	male	0
1	2	Florence	female	1
2	3	Laina	female	1
3	4	Lily	female	1
4	5	William	male	0

How to create a DataFrame?

- **Convert Pandas DataFrame to Spark DataFrame:**

```
df1 = spark.createDataFrame(df1_pd)
df2 = spark.createDataFrame(df2_pd)
df1.show()
```



```
+-----------+--------+------+--------+
|PassengerId|    Name|   Sex|Survived|
+-----------+--------+------+--------+
|          1|    Owen|  male|       0|
|          2|Florence|female|       1|
|          3|   Laina|female|       1|
|          4|    Lily|female|       1|
|          5| William|  male|       0|
+-----------+--------+------+--------+
```

Need for **Spark DataFrames**

Why do we need DataFrames?

Overview

- DataFrames are designed for processing a **large collection** of structured or semi-structured data
- Observations in the Spark DataFrame are **organized** under named columns, which helps Apache Spark understand the schema of a DataFrame. This helps Spark optimize the execution plan on these queries
- DataFrame in Apache Spark can handle **petabytes** of data
- DataFrame has a support for a **wide range** of data formats and sources
- It has the API support for different **languages** like Python, R, Scala, and Java

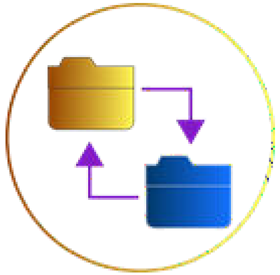
Spark DataFrame Transformations

Spark DataFrame Transformations

Select

- Used to select a subset of columns
- The method **select()** takes either a list of column names or an unpacked list of names

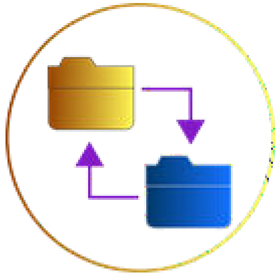
```
cols1 = ['PassengerId', 'Name']
df1.select(cols1).show()
```



Spark DataFrame Transformations

Filter

- Used to filter a subset of rows
- The method **filter()** takes column expressions or SQL expressions
- Similar to the WHERE clause in SQL queries

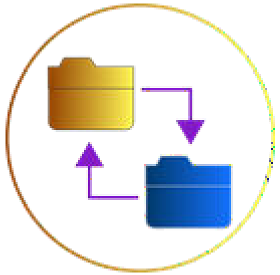


```
df1.filter(df1.Sex == 'female').show()
```

Spark DataFrame Transformations

Mutate

- It is possible to create new columns in Spark using **.withColumn()**
- There is no convenient way to create multiple columns at once without chaining multiple .withColumn() methods



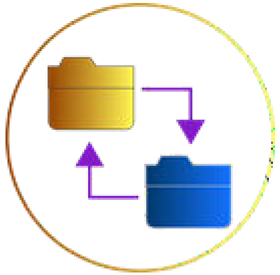
```
df2.withColumn('AgeTimesFare', df2.Age*df2.Fare).show()
```

Spark DataFrame Transformations

Summarize

- To summarize or aggregate a DataFrame, it has to be converted into a GroupedData object with **groupby()**
- Then, aggregate functions should be called

```
gdf2 = df2.groupby('Pclass')
gdf2
```



Spark DataFrame Transformations

Summarize

- The average of columns can be taken by passing an unpacked list of column names

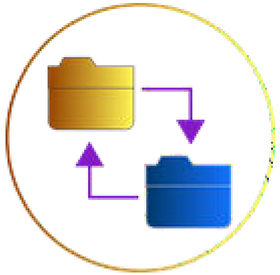
```
avg_cols = ['Age', 'Fare']
gdf2.avg(*avg_cols).show()
```

Spark DataFrame Transformations

Summarize

- To call multiple aggregation functions at once, pass a dictionary

```
gdf2.agg({'*': 'count', 'Age': 'avg', 'Fare':'sum'}).show()
```



Spark DataFrame Transformations

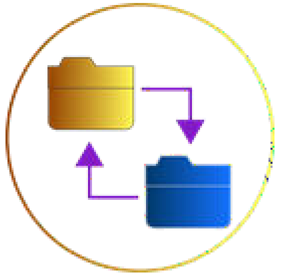
Summarize

- To rename the columns count(1), avg(Age), etc., use **toDF()**

```
(
    gdf2
    .agg({'*': 'count', 'Age': 'avg', 'Fare':'sum'})
    .toDF('Pclass', 'counts', 'average_age', 'total_fare')
    .show()
)
```

Spark DataFrame Transformations

Arrange



- Make use of the **.sort()** method to sort DataFrames
- Not many applications of this in Spark yet. Can expect more with updates.

```
df2.sort('Fare', ascending=False).show()
```

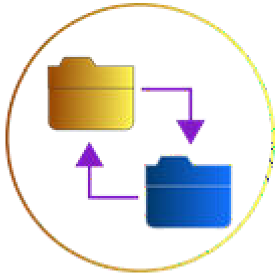
Spark DataFrame Transformations

Joins

- Joins and Unions are used to combine DataFrames
- The concept is very similar to the operation in SQL

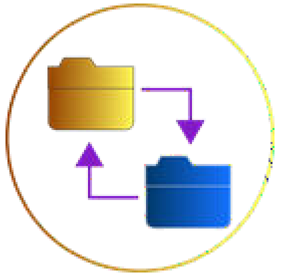
```
df1.join(df2, ['PassengerId']).show()
```

Example: Join the two Titanic DataFrames by the column PassengerId



Spark DataFrame Transformations

Joins

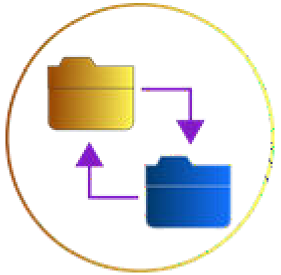


- Supports join by **conditions**
- Creates **duplicate** column names if the keys have the same name
- To **avoid** this, pass a list of join keys as in the previous code snippet
- If there is a requirement for non-equi joins, then keys need to be **renamed** before join

Spark DataFrame Transformations

Non-equi joins

- They are slow due to skewed data
- This is one thing that Spark can do and Hive cannot



Example:

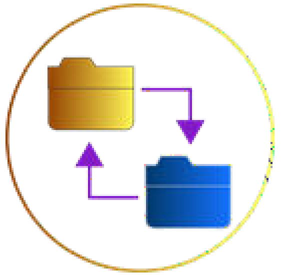
```
df1.join(df2, df1.PassengerId <= df2.PassengerId).show()
# Note the duplicate col names
```

Spark DataFrame Transformations

Unions

- Union() returns a DataFrame from the union of two DataFrames

```
df1.union(df1).show()
```



SQLContext

SQLContext

- SQLContext is the entry point for all SparkSQL functionalities
- HiveContext provides the superset of functionalities over SQLContext
- To create a SQLContext object, all you need is a SparkContext
- For initializing the SparkContext through Spark Shell, use Spark-shell
- By default, the SparkContext object is initialized with the name sc when the Spark Shell starts

SQLContext

- The entry point into all SQL functionality in Spark is the SQLContext class
- To create a basic instance, all we need is a SparkContext reference

```
from pyspark.sql import SQLContext
sqlContext = SQLContext(sc)
```

SQLContext: Inferring Schema

- With a SQLContext, we can create a DataFrame from the existing RDD
- But first, we need to tell Spark SQL the schema in our data
- Spark SQL can convert an RDD of row objects to a DataFrame
- Rows are constructed by passing a list of key–value pairs as kwargs to the Row class
- The keys define the column names, and the types are inferred by looking at the first row
- Hence, it is important that there is no missing data in the first row of the RDD in order to properly infer the schema

We will use the reduced dataset (10 percent) provided for the KDD Cup 1999, containing nearly half million network interaction

SQLContext: Inferring Schema

- We will first need to split the comma separated data, and then use the information in KDD's 1999 task description to obtain the column names

```
from pyspark.sql import Row

csv_data = raw_data.map(lambda l: l.split(","))
row_data = csv_data.map(lambda p: Row(
    duration=int(p[0]),
    protocol_type=p[1],
    service=p[2],
    flag=p[3],
    src_bytes=int(p[4]),
    dst_bytes=int(p[5])
    )
)
```

SQLContext: Inferring Schema

Once we have our, we can infer and register the schema

```
interactions_df = sqlContext.createDataFrame(row_data)
interactions_df.registerTempTable("interactions")
```

Now, we can run SQL queries over our DataFrame that has been registered as a table

```
# Select tcp network interactions with more than 1 second duration and
no transfer from destination
tcp_interactions = sqlContext.sql("""
    SELECT duration, dst_bytes FROM interactions WHERE protocol_type =
'tcp' AND duration > 1000 AND dst_bytes = 0
""")
tcp_interactions.show()
```

Performance Tuning

Performance Tuning

- For some workloads, it is possible to improve performance by either **caching data in memory** or **turning** on some experimental options
- Spark SQL can cache tables using an in-memory columnar format by calling **Spark.cacheTable("tableName")** or **Dataframe.cache()**
- Then, Spark SQL will scan only required columns and will automatically tune compression to minimize memory usage and GC pressure
- You can call **Spark.uncacheTable("tableName")** to remove the table from memory

Caching Data in Memory

Configuration of **in-memory caching** can be done using the **setConf** method on **SparkSession** or by running **SET key=value** commands using SQL

Property Name	Default	Meaning
Spark.SQL.inMemoryColumnarStorage.compressed	true	When set to true, Spark SQL will automatically select a compression codec for each column based on the statistics of the data
Spark.SQL.inMemoryColumnarStorage.batchSize	10000	It controls the size of batches for columnar caching. Larger batch sizes can improve memory utilization and compression but risk OOMs when caching data

Configuration Options

Following options can also be used to tune the performance of query execution:

Property Name	Default	Meaning
Spark.SQL.files.maxPartitionBytes	134217728 (128 MB)	This is the maximum number of bytes to pack into a single partition when reading files
Spark.SQL.files.openCostInBytes	4194304 (4 MB)	The estimated cost to open a file, measured by the number of bytes, can be scanned at the same time. This is used when putting multiple files into a partition
Spark.SQL.broadcastTimeout	300	This is the timeout in seconds for the broadcast wait time in broadcast joins
Spark.SQL.autoBroadcastJoinThreshold	10485760 (10 MB)	This configures the maximum size in bytes for a table that will be broadcast to all worker nodes when performing a join. By setting this value to −1, broadcasting can be disabled. Currently, statistics only supports Hive metastore tables where the command ANALYZE TABLE <tableName> COMPUTE STATISTICS noscan has been run
Spark.SQL.shuffle.partitions	200	This configures the number of partitions to use when shuffling data for joins or aggregations

Loading Data Through Different Sources

Parquet Files

- **Parquet** is a columnar format that is supported by many other data processing systems
- Spark SQL provides support for both **reading** and **writing** Parquet files that automatically preserve the schema of the **original** data
- When writing Parquet files, all columns are **automatically converted** to be nullable for compatibility reasons

Parquet Files

- Let us load the data programmatically:

```
peopleDF = spark.read.json("examples/src/main/resources/people.json")

# DataFrames can be saved as Parquet files, maintaining the schema information.
peopleDF.write.parquet("people.parquet")

# Read in the Parquet file created above.
# Parquet files are self-describing so the schema is preserved.
# The result of loading a parquet file is also a DataFrame.
parquetFile = spark.read.parquet("people.parquet")

# Parquet files can also be used to create a temporary view and then used in SQL statements.
parquetFile.createOrReplaceTempView("parquetFile")
teenagers = spark.sql("SELECT name FROM parquetFile WHERE age >= 13 AND age <= 19")
teenagers.show()
# +------+
# |  name|
# +------+
# |Justin|
# +------+
```

JSON Datasets

- Spark SQL can automatically infer the schema of a JSON dataset and load it as a DataFrame. This conversion can be done using SparkSession.read.json on a JSON file

- Note that the file that is offered as a json file is not a typical JSON file. Each line must contain a separate, self-contained valid JSON object

```
sc = spark.sparkContext

# A JSON dataset is pointed to by path.
# The path can be either a single text file or a directory storing text
files path = "examples/src/main/resources/people.json"
peopleDF = spark.read.json(path)

# The inferred schema can be visualized using the printSchema() method
peopleDF.printSchema()
# root
#  |-- age: long (nullable = true)
#  |-- name: string (nullable = true)
```

JSON Datasets

```
# Creates a temporary view using the DataFrame
peopleDF.createOrReplaceTempView("people")

# SQL statements can be run by using the SQL methods provided by spark
teenagerNamesDF = spark.sql("SELECT name FROM people WHERE age BETWEEN 13 AND
19")
teenagerNamesDF.show()
# +------+
# |  name|
# +------+
# |Justin|
# +------+

# Alternatively, a DataFrame can be created for a JSON dataset represented
by # an RDD[String] storing one JSON object per string
jsonStrings = ['{"name":"Yin","address":{"city":"Columbus","state":"Ohio"}}']
otherPeopleRDD = sc.parallelize(jsonStrings)
otherPeople = spark.read.json(otherPeopleRDD)
otherPeople.show()
# +---------------+----+
# |        address|name|
# +---------------+----+
# |[Columbus,Ohio]| Yin|
# +---------------+----+
```

Other Files

- To load a **CSV** file, you can use:

```
df = spark.read.load("examples/src/main/resources/people.csv",
                     format="csv", sep=":", inferSchema="true", header="true")
```

- You can run **SQL** on files **directly**:

```
df = spark.sql("SELECT * FROM parquet.`examples/src/main/resources/users.parquet`")
```

Spark SQL with Hive

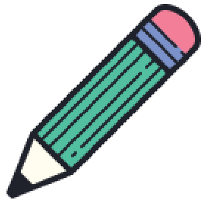
Spark–Hive Integration

- Spark SQL also supports reading and writing data stored in **Apache Hive**
- Hive has many dependencies; these dependencies are not included in the default Spark distribution
- If Hive dependencies can be found on the **classpath**, Spark will load them **automatically**
- These Hive dependencies must also be present on all worker nodes, as they will need access to the Hive serialization and deserialization libraries (SerDes) in order to access data stored in Hive

Spark–Hive Integration

- Configuration of Hive is done by placing your hive-site.xml, core-site.xml (for security configuration), and hdfs-site.xml (for HDFS configuration) file in **conf/**
- When working with Hive, one must instantiate SparkSession with Hive support, including connectivity to a persistent Hive metastore and support for Hive SerDes and Hive user-defined functions
- Users who do not have an existing Hive deployment can still enable Hive support
- When not configured by hive-site.xml, the context automatically creates metastore_db in the current directory and creates a directory configured by spark.sql.warehouse.dir, which defaults to the directory spark-warehouse in the current directory that the Spark application has started
- The hive.metastore.warehouse.dir property in hive-site.xml is deprecated since Spark 2.0.0. Instead, now, you can use spark.sql.warehouse.dir to specify the default location of the database in the warehouse

(You may need to grant write privilege to the user who starts the Spark application)



Spark–Hive Integration

Import Statements

```
from os.path import expanduser, join, abspath

from pyspark.sql import SparkSession
from pyspark.sql import Row
```

Warehouse Location

```
# warehouse_location points to the default location for managed databases and tables
warehouse_location = abspath('spark-warehouse')

spark = SparkSession \
    .builder \
    .appName("Python Spark SQL Hive integration example") \
    .config("spark.sql.warehouse.dir", warehouse_location) \
    .enableHiveSupport() \
    .getOrCreate()
```

Spark–Hive Integration

Spark is an existing SparkSession

```
spark.sql("CREATE TABLE IF NOT EXISTS src (key INT, value STRING) USING hive")
spark.sql("LOAD DATA LOCAL INPATH 'examples/src/main/resources/kv1.txt' INTO TABLE src")
```

Queries are expressed in HiveQL

```
spark.sql("SELECT * FROM src").show()
# +---+-------+
# |key|  value|
# +---+-------+
# |238|val_238|
# | 86| val_86|
# |311|val_311|
# ...
```

Aggregation queries are also supported

```
spark.sql("SELECT COUNT(*) FROM src").show()
# +--------+
# |count(1)|
# +--------+
# |    500 |
# +--------+
```

Spark–Hive Integration

The results of SQL queries are themselves DataFrames and support all normal functions

```
sqlDF = spark.sql("SELECT key, value FROM src WHERE key < 10 ORDER BY key")
```

The items in DataFrames are of type Row, which allows you to access each column by ordinal

```
stringsDS = sqlDF.rdd.map(lambda row: "Key: %d, Value: %s" % (row.key, row.value))
for record in stringsDS.collect():
    print(record)
# Key: 0, Value: val_0
# Key: 0, Value: val_0
# Key: 0, Value: val_0
# ...
```

Spark–Hive Integration

You can also use DataFrames to create temporary views within a SparkSession

```
Record = Row("key", "value")
recordsDF = spark.createDataFrame([Record(i, "val_" + str(i)) for i in range(1, 101)])
recordsDF.createOrReplaceTempView("records")
```

Queries can then join the DataFrame data with the data stored in Hive

```
spark.sql("SELECT * FROM records r JOIN src s ON r.key = s.key").show()
# +---+------+---+------+
# |key| value|key| value|
# +---+------+---+------+
# |  2| val_2|  2| val_2|
# |  4| val_4|  4| val_4|
# |  5| val_5|  5| val_5|
# ...
```

Spark–Hive Integration

Specifying storage format for Hive tables:

- For the creation of a Hive table, the table should be defined for read/write data from/to a file system, i.e., the **input format** and **output format**
- More definitions are needed for the table to deserialize the data to rows, or serialize rows to data, i.e., the **SerDe**
- **Hive storage handler** is not supported, yet when creating table you can create a table using storage handler at Hive side and use Spark SQL to **read** it later

Quiz!

Question #1

What does RDD stand for in Spark?

A	Redundant Datatype Deduction
B	Resilient Distributed Datasets
C	Resuable Datatype Dataset
D	Repeated Distributed Datasets

Answer #1

What does RDD stand for in Spark?

A	Redundant Datatype Deduction
B	**Resilient Distributed Datasets** ✓
C	Resuable Datatype Dataset
D	Repeated Distributed Datasets

Question #2

Joining is a valid operation using Spark SQL.

A True

B False

Answer #2

Joining is a valid operation using Spark SQL.

A	**True** ✓
B	False

Question #3

DataFrames are supported by Python, Java, Scala, and R programming languages.

A True

B False

Answer #3

DataFrames are supported by Python, Java, Scala, and R programming languages.

A	**True** ✓
B	False

+91-9030485102

rganesh0203@gmail.com

https://topmate.io/rganesh_0203