DDAM
SPARK SQL & DATAFRAME

Docente: Patrizio Dazzi

SPARK SQL

One use of Spark SQL is to execute SQL queries. When running SQL the results will be returned as a Dataset/DataFrame.

A Dataset is a distributed collection of data. Dataset is a new interface added since Spark 1.6 that provides the benefits of RDDs (strong typing, ability to use powerful lambda functions) with the benefits of Spark SQL's optimized execution engine.

https://spark.apache.org/docs/latest/sql-programming-guide.html

SPARK SESSION

The entry point into all functionality in Spark is the SparkSession class. To create a basic SparkSession, just use SparkSession.builder:

```
from pyspark.sql import SparkSession

spark = SparkSession \
    .builder \
    .appName("Python Spark SQL basic example") \
    .config("spark.some.config.option", "some-value") \
    .getOrCreate()
```

CREATING DATAFRAMES

With a SparkSession, applications can create DataFrames from an existing RDD, from a Hive table, or from Spark data sources.

As an example, the following creates a DataFrame based on the content of a JSON file:

```
# spark is an existing SparkSession
df = spark.read.json("examples/src/main/resources/people.json")
# Displays the content of the DataFrame to stdout
df.show()
# +----+-------+
# | age|   name|
# +----+-------+
# |null|Michael|
# |  30|   Andy|
# |  19| Justin|
# +----+-------+
```

INTERACTING WITH DATAFRAMES

```
# spark, df are from the previous example
# Print the schema in a tree format
df.printSchema()
# root
# |-- age: long (nullable = true)
# |-- name: string (nullable = true)

# Select only the "name" column
df.select("name").show()
# +-------+
# |   name|
# +-------+
# |Michael|
# |   Andy|
# | Justin|
# +-------+

# Select everybody, but increment the age by 1
df.select(df['name'], df['age'] + 1).show()
# +-------+---------+
# |   name|(age + 1)|
# +-------+---------+
# |Michael|     null|
# |   Andy|       31|
# | Justin|       20|
# +-------+---------+
```

```
# Select people older than 21
df.filter(df['age'] > 21).show()
# +---+----+
# |age|name|
# +---+----+
# | 30|Andy|
# +---+----+

# Count people by age
df.groupBy("age").count().show()
# +----+-----+
# | age|count|
# +----+-----+
# |  19|    1|
# |null|    1|
# |  30|    1|
# +----+-----+
```

JOIN

```
%python

llist = [('bob', '2015-01-13', 4), ('alice', '2015-04-23',10)]
left = sqlContext.createDataFrame(llist, ['name','date','duration'])
right = sqlContext.createDataFrame([('alice', 100),('bob', 23)],['name','upload'])

df = left.join(right, left.name == right.name)

display(df)
```

https://docs.databricks.com/spark/latest/dataframes-datasets/introduction-to-dataframes-python.html

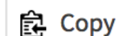
CREATING DATAFRAMES RELATIONS

```
# import pyspark class Row from module sql
from pyspark.sql import *

# Create Example Data - Departments and Employees

# Create the Departments
department1 = Row(id='123456', name='Computer Science')
department2 = Row(id='789012', name='Mechanical Engineering')
department3 = Row(id='345678', name='Theater and Drama')
department4 = Row(id='901234', name='Indoor Recreation')

# Create the Employees
Employee = Row("firstName", "lastName", "email", "salary")
employee1 = Employee('michael', 'armbrust', 'no-reply@berkeley.edu', 100000)
employee2 = Employee('xiangrui', 'meng', 'no-reply@stanford.edu', 120000)
employee3 = Employee('matei', None, 'no-reply@waterloo.edu', 140000)
employee4 = Employee(None, 'wendell', 'no-reply@berkeley.edu', 160000)

# Create the DepartmentWithEmployees instances from Departments and Employees
departmentWithEmployees1 = Row(department=department1, employees=[employee1, employee2])
departmentWithEmployees2 = Row(department=department2, employees=[employee3, employee4])
departmentWithEmployees3 = Row(department=department3, employees=[employee1, employee4])
departmentWithEmployees4 = Row(department=department4, employees=[employee2, employee3])

print department1
print employee2
print departmentWithEmployees1.employees[0].email
```

WORKING WITH DATAFRAMES

Copy

```
unionDF = df1.unionAll(df2)
display(unionDF)
```


Copy

```
filterDF = explodeDF.filter(explodeDF.firstName == "xiangrui").sort(explodeDF.lastName)
display(filterDF)
```

Copy

```
from pyspark.sql.functions import col, asc

# Use `|` instead of `or`
filterDF = explodeDF.filter((col("firstName") == "xiangrui") | (col("firstName") == "michael")).sort(asc("lastName"))
display(filterDF)
```

Copy

```
from pyspark.sql.functions import countDistinct

countDistinctDF = explodeDF.select("firstName", "lastName")\
  .groupBy("firstName", "lastName")\
  .agg(countDistinct("firstName"))

display(countDistinctDF)
```

RUNNING AN SQL QUERY

The sql function on a SparkSession enables applications to run SQL queries programmatically and returns the result as a DataFrame.

```
# Register the DataFrame as a SQL temporary view
df.createOrReplaceTempView("people")

sqlDF = spark.sql("SELECT * FROM people")
sqlDF.show()
# +----+-------+
# | age|   name|
# +----+-------+
# |null|Michael|
# |  30|   Andy|
# |  19| Justin|
# +----+-------+
```

GLOBAL TEMP VIEWS

Temporary views in Spark SQL are session-scoped and will disappear if the session that creates it terminates. If you want to have a temporary view that is shared among all sessions and keep alive until the Spark application terminates, you can create a global temporary view.

Global temporary view is tied to a system preserved database global_temp, and we must use the qualified name to refer it, e.g. SELECT * FROM global_temp.view1.

```
# Register the DataFrame as a global temporary view
df.createGlobalTempView("people")

# Global temporary view is tied to a system preserved database `global_temp`
spark.sql("SELECT * FROM global_temp.people").show()
# +----+-------+
# | age|   name|
# +----+-------+
# |null|Michael|
# |  30|   Andy|
# |  19| Justin|
# +----+-------+

# Global temporary view is cross-session
spark.newSession().sql("SELECT * FROM global_temp.people").show()
# +----+-------+
# | age|   name|
# +----+-------+
# |null|Michael|
# |  30|   Andy|
# |  19| Justin|
# +----+-------+
```