MET CS 777

Big Data Analytics

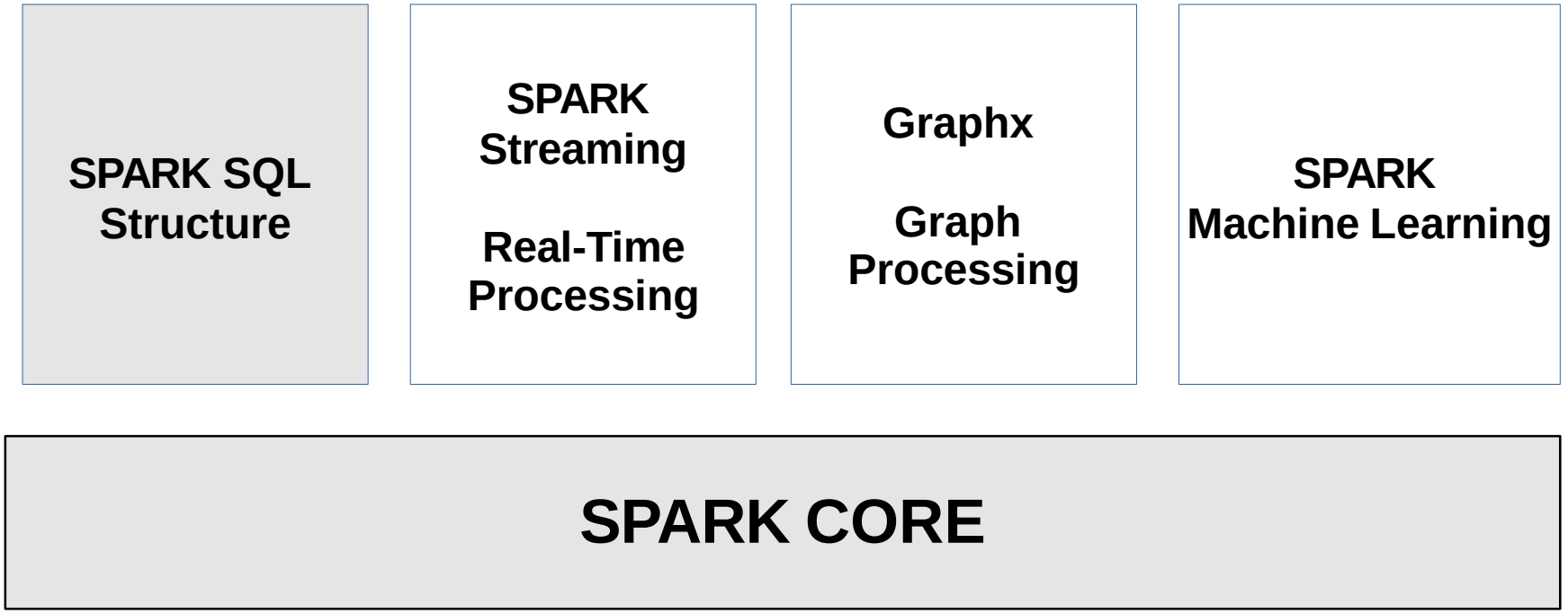
Spark Dataframes and Spark SQL

Lecture - 3

Dimitar Trajanov

SPARL Libraries

Libraries included in SPARK



Spark SQL Documentations

- Spark SQL Programming Guide

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

- API Documentations

  http://spark.apache.org/docs/latest/api/python/pyspark.sql.html

DataFrames

- A **DataFrame** is a distributed collection of data organized into named columns. (i.e. RDD with schema)
  - Equivalent to a table in a relational database or
  - a data frame in R/Python
- **DataFrames** can be constructed from different sources such as:
  1. structured data files, (CSV, JSON, …)
  2. external databases (MySql, PostgreSQL, S3, JDBS, …),
  3. existing RDDs
- **SPARK SQL**
  - Spark SQL is Spark's module for working with structured data.
  - Querying structured data inside Spark programs, using either SQL or **DataFrame API.**

Spark DataFrames

- The key difference between RDD and DataFrame is that DataFrame stores much more information about the data, such as the data types and names of the columns, than RDD.
- This allows the DataFrame to optimize the processing much more effectively than Spark transformations and Spark actions doing processing on RDD.

- **DataFrames enforce types !**

Spark data types vs Python Data Type

Data type	Value type in Python	API to access or create a data type
ByteType	int or long Note: Numbers will be converted to 1-byte signed integer numbers at runtime. Please make sure that numbers are within the range of -128 to 127.	ByteType()
ShortType	int or long Note: Numbers will be converted to 2-byte signed integer numbers at runtime. Please make sure that numbers are within the range of -32768 to 32767.	ShortType()
IntegerType	int or long	IntegerType()
LongType	long Note: Numbers will be converted to 8-byte signed integer numbers at runtime. Please make sure that numbers are within the range of -9223372036854775808 to 9223372036854775807.Otherwise, please convert data to decimal.Decimal and use DecimalType.	LongType()

Spark data types vs Python Data Type

Data type	Value type in Python	API to access or create a data type
FloatType	float Note: Numbers will be converted to 4-byte single-precision floating point numbers at runtime.	FloatType()
DoubleType	float	DoubleType()
DecimalType	decimal.Decimal	DecimalType()
StringType	string	StringType()
BinaryType	bytearray	BinaryType()
BooleanType	bool	BooleanType()
TimestampType	datetime.datetime	TimestampType()
DateType	datetime.date	DateType()

Spark data types vs Python Data Type

Data type	Value type in Python	API to access or create a data type
ArrayType	list, tuple, or array	ArrayType(elementType, [containsNull]) Note:The default value of containsNull is True.
MapType	dict	MapType(keyType, valueType, [valueContainsNull]) Note:The default value of valueContainsNull is True.
StructType	list or tuple	StructType(fields) Note: fields is a Seq of StructFields. Also, two fields with the same name are not allowed.
StructField	The value type in Python of the data type of this field (For example, Int for a StructField with the data type IntegerType)	StructField(name, dataType, [nullable]) Note: The default value of nullable is True.

Creating DataFrames

- **sqlContext.createDataFrame**(data, schema=None, samplingRatio=None)
  - Creates a DataFrame from an RDD of tuple/list, list or …

From List

```
>>> a = [('Chris', 'Berliner', 5)]
>>> sqlContext.createDataFrame(a, ['drinker', 'beer', 'score']).collect()
[Row(drinker=u'Chris', beer=u'Berliner', score=5)]
```

From RDD

```
>>> rdd = sc.parallelize(a)
>>> sqlContext.createDataFrame(rdd).collect()
[Row(_1=u'Chris', _2=u'Berliner', _3=5)]
>>> df = sqlContext.createDataFrame(rdd, ['drinker', 'beer', 'score'])
>>> df.collect()
[Row(drinker=u'Chris', beer=u'Berliner', score=5)]
```

From RDD and by add schema

```
>>> from pyspark.sql.types import *
>>> schema = StructType([
...    StructField("drinker", StringType(), True),
...    StructField("beer", StringType(), True),
...    StructField("score", IntegerType(), True)])
>>> df3 = sqlContext.createDataFrame(rdd, schema)
>>> df3.collect()
[Row(drinker=u'Chris', beer=u'Berliner', score=5)]
```

Reading Data from CSV files

```
customers = sqlContext.read.format('csv')\
    .options(header='true', inferSchema='true',  sep="|")\
    .load("file:///customer.tbl")
```

- **Show the table**

  customers.show()
- **Print out schema** of Dataframe

  customers.printSchema()

CSV Files - Spark 3.2.1 Documentation (apache.org)
pyspark.SparkContext.textFile — PySpark 3.1.1 documentation (apache.org)

Creating DataFrames

- **JDBC**
  - Imports Tables from remote database as a DataFrame

```
df = sqlContext.load( source="jdbc",
  url="jdbc:postgresql://<HOST>/<DATABASE>?user=<USERNAME>&password=<PAS
  SWORD>", dbtable="<SCHEMA>.<TABLENAME>")
```

- **jsonFile(PATH)**
  - Loads a text file storing one JSON object per line as a DataFrame.

```
>>> sqlContext.jsonFile('file:///home/rates.json').dtypes
[ ('drinker', 'string'), ('beer', 'string'), ('score', 'bigint'),]
```

Operations on DataFrames

- **registerDataFrameAsTable**(df, tableName)
  - Registers the given DataFrame as a temporary table in the catalog.

```
>>> sqlContext.registerDataFrameAsTable(df, "rates")
```

- **sql(sqlQuery)**
  - Returns a DataFrame representing the result of the given query.

```
>>> sqlContext.registerDataFrameAsTable(df, "rates")
>>> df2 = sqlContext.sql("SELECT drinker AS d, beer as b, score as s from rates")
>>> df2.collect()
[Row(d=u'Chris', b=u'Berliner', s=5), Row(d=u'Kia', b=u'Erdinger Kristal', s=2) ]
```

SQL optimizer is not efficient as the one used in Databases.
Use simple execution plans only

Operations on DataFrames

- **select(*cols)**
  - Projects a set of expressions and returns a new DataFrame.

```
>>> df.select('*').collect()
[Row(drinker=u'Chris', beer=u'Berliner', score=5), Row(drinker=u'Kia', beer=u'Erdinger Kristal',
score=2)]

>>> df.select('beer', 'score').collect()
[Row(beer=u'Berliner', score=5),  Row(beer=u'Erdinger Kristal', score=2)]
```

- **selectExpr(*expr)**
  - Projects a set of SQL expressions and returns a new DataFrame.

```
>>> df.selectExpr("(score / 10) * 100", "abs(score)").collect()

[Row(((score / 10) * 100)=50, abs(age)=5), Row(((score / 10) * 100)=20, abs(age)=2)]
```

Different way to access columns

- Four different way to access columns

```
# Using the string to column conversion
df.select("Country_code", "Country", "Total_Library_Size")
df.select(*["Country_code", "Country", "Total_Library_Size"])

# Passing the column object explicitly
from pyspark.sql import functions as F
df.select(
F.col("Country_code"), F.col("Country"), F.col("Total_Library_Size")
)
df.select(
*[F.col("Country_code"), F.col("Country"), F.col("Total_Library_Size")]
)
```

Operations on DataFrames

- **Projection operation : filter(condition)**
  - Filters rows using the given condition.

    Condition: a Column of types.BooleanType or a string of SQL expression.

```
>>> df.filter("df.score > 3").collect()
[Row(drinker=u'Chris',  beer=u'Berliner', score=5)]

>>> df.filter("df.score = 2").collect()
[Row(drinker=u'Kia', beer=u'Erdinger Kristal', score=2)]
```

- **drop(col)**
  - Returns a new DataFrame that drops the specified column.

```
>>> df.drop('drinker').collect()
[Row(beer=u'Berliner', score=5), Row(beer=u'Erdinger Kristal', score=2)]
```

Operations on DataFrames

- **groupBy(*cols)**
  - Groups the DataFrame using the specified columns

```
>>> df.drop('drinker').groupBy('beer').agg({'score': 'mean'}).collect()
[Row(beer=u'Berliner', score=5), Row(beer=u'Erdinger Kristal', score=2)]
```

- **join(other, on=None, how=None)**
  - Joins with another DataFrame, using the given join expression.
  - how: how – str, default '*inner*'. One of *inner, outer, left_outer, right_outer, semijoin.*

```
>>> df.join(df2, df.drinker == df2.drinker, 'outer').select(df.drinker, df2.bar).collect()
[Row(drinker=u'Chris', bar=u'Boheme'), Row(drinker=u'Kia', bar='Anvil')]
```

Operations on DataFrames

- df.show()
  - shows the content of the DataFrame
- df.printSchema() - Prints its Schema

```
## root
## |-- drinker: string (nullable = true)
## |-- beer: string (nullable = true)
## |-- score: bigint (nullable = true)
```

- df.count()
  - Counts the number of records
- df.distinct()
- df.rdd() - returns the content as an pyspark.RDD.

Important Classes of SparkSQL and DataFrames

- **pyspark.sql.SQLContext**
  - Main entry point for DataFrame and SQL functionality.
- **pyspark.sql.DataFrame**
  - A distributed collection of data grouped into named columns.
- **pyspark.sql.Column**
  - A column expression in a DataFrame.
- **pyspark.sql.Row**
  - A row of data in a DataFrame.
- **pyspark.sql.DataFrameStatFunctions**
  - Methods for statistics functionality.
- **pyspark.sql.functions**
  - List of built-in functions available for DataFrame.

Sql Types - pyspark.sql.types

- Numeric types

  ByteType: Represents 1-byte signed integer numbers. (-128 to 127)

  ShortType: Represents 2-byte signed integer numbers. (-32768 to 32767)

  IntegerType: Represents 4-byte signed integer numbers. (-2147483648 to 2147483647)

  LongType: Represents 8-byte signed integer numbers.

  (-9223372036854775808 to 9223372036854775807)

  FloatType: Represents 4-byte single-precision floating point numbers.

  DoubleType: Represents 8-byte double-precision floating point numbers.
- **StringType**: Represents character string values.
- **BinaryType**: Represents byte sequence values.
- **BooleanType**: Represents boolean values.
- **TimestampType**: Represents values comprising values of fields year, month, day, hour, minute, and second.
- **DateType**: Represents values comprising values of fields year, month, day.

- **CAST** – You can cast types

  variableName.cast(LongType())

Add a Column to Table

- DataFrame.withColumn(colName, col)
  - ColName: string, name of the new column.
  - col: a Column expression for the new column.

- Similar to RDD map() operation

```
from pyspark.sql.functions import lit
# lit(col) Creates a Column of a literal value.
NEW_DF=YOUR_DF.withColumn("COUNT", lit(1))
```

Aggregate function: agg(*exprs)

- Compute aggregates and returns the result as a **DataFrame**.

```
NEW_DF = df.groupBy("MYKEY").agg(func.sum("COUNT")).show()
```

- The available aggregate functions can be:
  - built-in aggregation functions, such as *avg*, *max*, *min*, *sum*, *count*
  - group aggregate pandas UDFs, created with **pandas_udf()**
- **Note**
  - There is no partial aggregation with group aggregate UDFs, i.e., **a full shuffle is required.**
  - Also, all the data of a group will be loaded into memory, so the user should be aware of the potential OOM risk if data is skewed and certain groups are too large to fit in memory.
- If `exprs` is a single **`dict`** mapping from string to string, then
  - the key is the column to perform aggregation on, and
  - the value is the aggregate function.
- Alternatively, exprs can also be a list of aggregate Column expressions.

Aggregate function: agg(*exprs)

- agg() Examples

```
from pyspark.sql import functions as F
df.agg({"age": "max"}).collect()
df.agg(F.min(df.age)).collect()
```

- Predefined aggregate functions

```
df.groupBy().avg('age').collect()
```

- Available predefined aggregate functions

avg(*cols)	Computes average values for each numeric columns for each group.
count()	Counts the number of records for each group.
max(*cols)	Computes the max value for each numeric columns for each group.
mean(*cols)	Computes average values for each numeric columns for each group.
min(*cols)	Computes the min value for each numeric column for each group.
sum(*cols)	Computes the sum for each numeric columns for each group.

UDF – User Defined Functions

```
from pyspark.sql.types import StringType
from pyspark.sql.functions import udf

maturity_udf = udf(lambda age: "adult" if age >=18 else "child", StringType())

df = sqlContext.createDataFrame([{'name': 'Alice', 'age': 1}])
df.withColumn("maturity", maturity_udf(df.age))
```

Top-K Queries

from pyspark.sql.types import *

from pyspark.sql import functions as func

from pyspark.sql.functions import lit

```
lines = lineitems.select("ORDERKEY", "PARTKEY")\
.withColumn("COUNT", lit(1))
.groupBy("PARTKEY").agg(func.sum("COUNT"))\
```

lines.orderBy("sum(COUNT)", ascending=False).limit(10).show()

pyspark.sql.DataFrameStatFunctions

How to calculate Quantiles on Big Data

- **df.approxQuantile(col, probabilities, relativeError)**

  Calculates the approximate quantiles of a numerical column of a DataFrame.

- **col** – the name of the numerical column
- **probabilities** – a list of quantile probabilities Each number must belong to [0, 1]. For example 0 is the minimum, 0.5 is the median, 1 is the maximum.
- **relativeError** – The relative target precision to achieve (>= 0). If set to zero, the exact quantiles are computed, which could be very expensive. Note that values greater than 1 are accepted but give the same result as 1.

  e.g.,

  df.approxQuantile("x", [0.5], 0.25)

- Implements Greenwald-Khanna algorithm

  http://infolab.stanford.edu/~datar/courses/cs361a/papers/quantiles.pdf

Related Projects in the Hadoop Ecosystem

- **Apache Parquet (http://parquet.apache.org/)**
  - Column Store database
  - Data is stored as columns
  - Data can be read parallel using multiple processes