Data Frames and Spark SQL

MSBA 6330 Prof Liu

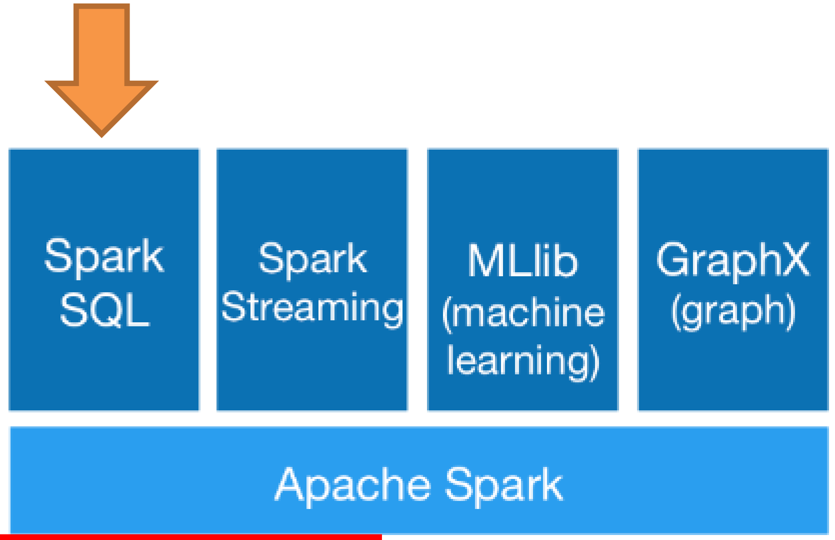
DataFrames and SparkSQL

- In this module you will learn
  - What Spark SQL is
  - How to create a DataFrame
  - How to query data in a DataFrame
  - How to manipulate data with DataFrame
  - Comparison between Spark SQL, Hive, and Impala

Data Frames and Spark SQL

WHAT IS SPARK SQL

What is Spark SQL?



- What is Spark SQL?
  - Spark module for structured data processing
  - Built on top of core Spark
- What does Spark SQL provide?
  - The DataFrame API – a library for working with data as tables
  - Defines DataFrames containing Rows and Columns
  - Catalyst Optimizer – an extensible optimization framework
  - A SQL Engine and command line interface

Why Spark SQL

- Spark SQL can be used for
  - Complex data manipulation and analytics
  - Integration with other data systems and APIs
  - Machine learning (data preparation)
  - Streaming and other long-running applications
- Spark SQL APIs tries to mimic Pandas APIs
  - Make it easy for python data scientists to use SparkSQL
  - though differences exist

Starting Point for Spark SQL: SparkSession

- The entry point into all functionality in Spark is a SparkSession
  - Spark 2.0+ provides built-in support for Hive features including the ability to write queries using HiveQL, access to Hive UDFs, and the ability to read data from Hive tables.
  - To use these features, you do not need to have an existing Hive setup.
- With a SparkSession, applications can create DataFrames from an existing RDD, from a Hive table, or from Spark data sources.
- A SparkSession spark is automatically created in a spark shell.
  - In a standalone spark application, you must create it yourself.

Creating a Spark Session

- Create a SparkSession programmatically

python

```python
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName("MyApp") \
    .config("SomeOption", "SomeValue").getOrCreate()
```

scala

```scala
import org.apache.spark.sql.SparkSession
val spark = SparkSession.builder().appName("MyApp")
    .config("SomeOption", "SomeValue").getOrCreate()
// For implicit conversions like converting RDDs to DataFrames
import spark.implicits._
```

In our Spark Shell environment, a SparkSession is automatically created and save in the variable *spark*.

Data Frames and Spark SQL

HOW TO CREATE A DATAFRAME

DataFrame

There is also a concept of Spark **DataSet** (available in Scala and Java but not in PySpark or SparkR). DataSet is more strongly typed DataFrame.

- DataFrame is the main abstraction in Spark SQL
  - Is a distributed collection of named columns.
  - It is conceptually equivalent to a (columnar) table in a relational database
- DataFrame & RDD resilient distributed dataset
  - DataFrame corresponds to an RDD of Row objects

```
from pyspark.sql import Row

data = [('Ankit',25),('Jalfaizy',22),('saurabh',20),('Bala',26)]
rdd = sc.parallelize(data)
rowRdd = rdd.map(lambda x: Row(name=x[0], age=int(x[1])))

df = spark.createDataFrame(rowRdd)
```

Create a list of tuples

Create a RDD from the the list above

Convert each tuple to a Row

Create a DataFrame by applying CreateDataFrame on RDD[Row]

python

RDD[tuple]

('Ankit',25)
('Jalfaizy',22)
('saurabh',20)
('Bala',26)

DataFrame

name:**string**	age:**int**
'Ankit'	25
'Jalfaizy'	22
'saurabh'	20
'Bala'	26

Create DataFrame from RDDs

- We can create DataFrames from an existing RDD, from a Hive table, or from Spark data sources
- From an existing RDD using

```
spark.CreateDataFrame(rdd, schema=None)
```

- The main issue is how RDD will gain schema (column names & types)
  - if rdd is RDD[Row] type, then no need to specify schema (previous example)

    ```
    df = spark.createDataFrame(rowRdd)
    ```

  - if rdd is RDD[tuple], schema will be inferred, unless specified.

    ```
    df = spark.createDataFrame(rdd) #infer column types, default col names _0,_1, ...
    ```

https://spark.apache.org/docs/latest/api/python/pyspark.sql.html#pyspark.sql.SparkSession.createDataFrame

Create DataFrame from RDDs

- Schema can be a list of column names or StructType
  - specify column names only

    ```
    df = spark.createDataFrame(rdd,['name','age'])
    ```

    - Data types will be inferred
  - Specify both column names and data types (and whether they are nullable).

```
from pyspark.sql.types import *
schema = StructType([
    StructField("name", StringType(), False),
    StructField("age", IntegerType(), True)
])

df = spark.createDataFrame(rdd,schema)
```

python

```
import org.apache.spark.sql.types._
val schema = StructType(Array(
  StructField("name", StringType, false),
  StructField("age", IntegerType, true)
))

val df = spark.createDataFrame(rdd,schema)
```

scala

Create DataFrame from Spark Data Sources

- Spark SQL supports a wide range of data source types and formats for DataFrames
  - Text files
    - CSV, JSON, Plain text
  - Binary format files
    - Apache Parquet, Apache ORC
  - Tables
    - Hive metastore, JDBC
  - You can also use custom or third-party data source types

Read Data using `.read`

```
spark.read.format().option('key','value').load('/path/to/file')
```

- `spark.read` returns a **DataFrameReader** object

- Use **DataFrameReader** settings to specify how to load data from the data source
  - **`.format(source)`**: e.g. `json`, `parquet`, `csv`, `jdbc`, etc.
  - **`.options`**: add options such as `header`, `inferSchema, delimiter, url`
  - **`.option(key,value)`**: add options one by one
  - **`.schema(schema):`** specify input schema, can be either `StructType` or string (in the form of "`col0 INT, col1 DOUBLE`")

- Create the DataFrame based on the data source
  - `load()` loads data from a file or files
  - `table()` loads data from a Hive table

https://spark.apache.org/docs/latest/api/python/pyspark.sql.html#pyspark.sql.DataFrameReader

`DataFrameReader` Convenience Functions

- You can call a format-specific load function
  - A shortcut instead of setting the format and using load
  - `csv`, `json`, `orc`, `parquet`, `text`, `table`, `jdbc`.
- The following two code examples are equivalent

```
spark.read.option("header","true").format("csv").load("/loudacre/myFile.csv")
spark.read.csv("/loudacre/myFile.csv",header=True)
```

python

```
val sfpd = spark.read.format("json").load("iris.json")
val sfpd = spark.read.json("iris.json")
```

scala

Specifying Data Source File Locations

- You must specify a location when reading from a file data source
  - The location can be a single file, a list of files, a directory, or a wildcard
    - `spark.read.json("myfile.json")`
    - `spark.read.json("mydata/")`
    - `spark.read.json("mydata/*.json")`
    - `spark.read.json("myfile1.json","myfile2.json")`
- Files and directories are referenced by absolute or relative URI
  - Relative URI (uses the default file system)
    - `myfile.json  (in the HDFS's home directory on our VM)`
  - Absolute URI
    - `hdfs://host/loudacre/myfile.json  (on the HDFS)`
    - `file:/home/cloudera/myfile.json  (on local host)`

Examples: Read CSV file and Hive Table

- Read a CSV text file, treating the first line in the file as a header instead of data

```python
myDF = spark.read. \
    format("csv"). \
    option("header","true"). \
    load("/loudacre/myFile.csv")
```

python

```scala
val myDF = spark.read.
    format("csv").
    option("header","true").
    load("/loudacre/myFile.csv")
```

scala

- Other options include `inferSchema`, `sep`, `quote`, `escape`, etc

```
option("header","true").option("sep","\t").option("inferSchema","true")
options(header="true",sep="\t",inferSchema="true")
```

References
https://spark.apache.org/docs/latest/api/python/pyspark.sql.html#pyspark.sql.DataFrameReader

Creating a DataFrame from a JSON File

- A JSON source by default is newline-delimited JSON where each line is a row.

```
peopleDF = spark.read.json("people.json")
```

python

```
val peopleDF = spark.read.json("people.json")
```

scala

File: people.json

```
{"name":"Alice", "pcode":"94304"}
{"name":"Brayden", "age":30, "pcode":"94304"}
{"name":"Carla", "age":19, "pcode":"10036"}
{"name":"Diana", "age":46}
{"name":"Étienne", "pcode":"94104"}
```

age	name	pcode
null	Alice	94304
30	Brayden	94304
19	Carla	10036
46	Diana	null
null	Étienne	94104

Create DataFrame JDBC Data Sources

Example: Loading from a MySQL database

```
accountsDF = spark.read.format('jdbc').options( \
  url="jdbc:mysql://loalhost/mydb?user=…&password=…", dbtable="accounts").load()
```

python

```
val accountsDF = spark.read.format('jdbc').options( Map("url"->
"jdbc:mysql://localhost/mydb?user=…&password=…",  "dbtable" -> "accounts")).load()
```

scala

Warning: Avoid direct access to databases in production environments, which may overload the DB or be interpreted as service attacks

- Use Sqoop to import instead

Creating DataFrames: Summary

- DataFrames can be created
  - From an existing structured data source
    - Parquet file, JSON file, etc. (schema info is embedded in the source data)
    - csv, text, etc. (schema is inferred)
  - From transforming an existing RDD with inferred or specified schema
  - By performing an operation or query on another DataFrame

Data Frames and Spark SQL

SAVE A DATAFRAME

DataFrameWriter Functions

- The DataFrame's write function returns a **DataFrameWriter**
  - Saves data to a data source such as a table or set of files
  - Works similarly to DataFrameReader
- DataFrameWriter methods
  - **`format`** specifies a data source type
  - **`mode`** determines the behavior if the directory or table already exists: `error`, `overwrite`, `append`, or `ignore` (default is error)
  - **`partitionBy`** stores data in partitioned directories in the form of column=value (as with Hive/Impala partitioning)
  - **`option`** specifies properties for the target data source
  - **`save`** saves the data as files in the specified directory
    - Or use `json`, `csv`, `parquet`, and so on
  - **`saveAsTable`** saves the data to a Hive metastore table
    - Uses default table location (/user/hive/warehouse)
    - Set path option to override location

Examples: Saving a DataFrame to a Data Source

- Example: Write data to a Hive metastore table called `my_table`
  - Append the data if the table already exists
  - Use an alternate location

```
myDF.write. \
    mode("append"). \
    option("path","/loudacre/mydata"). \
    saveAsTable("default.my_table")
```

python

- Example: Write data as Parquet files in the mydata directory

```
myDF.write.save("mydata")
```

python

Note: saveAsTable seems not able to create Hive compatible table.
An alternate solution (using TempView) is suggested here https://goo.gl/3Pp1gj

Register the DataFrame as a “table”

- First, register the DataFrame as a temporary table using the given name
- Then, you can use the table in subsequent SQL queries.

```
peopleDF.createOrReplaceTempView("people")
spark.sql("SELECT * FROM people WHERE name LIKE \"A%\" ").show()
```

python/scala

age	name	pcode
null	Alice	94304
30	Brayden	94304
19	Carla	10036
46	Diana	null
null	Étienne	94104

age	name	pcode
null	Alice	94304

- The lifetime of the temporary table is tied to the SparkSession
  - Use createGlobalTempView to create references that can be used across spark sessions.

Data Frames and Spark SQL

DATAFRAME OPERATIONS

Working with Data in a DataFrame

- Meta operations – operates on meta data rather than data itself.
  - E.g. printSchema
- Queries – create a new DataFrame
  - DataFrames are immutable
  - Queries are analogous to RDD transformations
  - Queries are lazily evaluated
  - Queries can be chained like transformations
- Actions – return data to the Driver
  - Actions trigger “lazy” execution of queries
  - E.g. show()

DataFrame Meta Operations

- Meta Operations deal with DataFrame metadata (rather than its data)

Operation	Description
printSchema()	displays the schema as a visual tree
columns	returns an array containing the names of the columns
dtypes	returns an array of (column-name,type) pairs
explain()	prints debug information about the DataFrame to the console
createTempView()	Registers this DataFrame as a temporary view using the given name.
toDF(*cols)	Returns a new Data Frame with new specified column names
cache()	persists the DataFrame to disk or memory

DataFrame Meta Operation Examples

- show the schema of a DataFrame (col names and data types)

```
peopleDF.printSchema()
root
 |-- age: long (nullable = true)
 |-- gender: string (nullable = true)
 |-- name: string (nullable = true)
```

python / scala

- Obtain a list of column names & # of columns

```
peopleDF.columns
['age', 'name', 'pcode']
len(peopleDF.columns)
3
```

python

DataFrame Meta Operation Examples

- Displaying column data types as a list of tuples using dtypes

```
for item in peopleDF.dtypes:
    print item
('age', 'bigint')
('name', 'string')
('pcode', 'string')
```

python

```
people.dtypes.foreach(println)
(age,LongType)
(name,StringType)
(pcode,StringType)
```

scala

Commonly Used Actions

- DataFrame actions return value/data to the driver program

collect()	return all rows as an array of Row objects
count()	Return the number of rows
first(); head()	Returns the first row, same as take(1)
show(n)	Print the first n (default 20) rows in tabular form
take(n)	Returns the first n rows as an array of Row objects

DataFrame Action Examples

- Show the first n rows, using show().

```
peopleDf = spark.read.json("people.json")
peopleDF.show(4)
+----+-------+-----+
| age|   name|pcode|
+----+-------+-----+
|null|  Alice|94304|
|  30|Brayden|94304|
|  19|  Carla|10036|
|  46|  Diana| null|
+----+-------+-----+
```

python/scala

DataFrame's `head` works differently from Pandas. It returns a list or `Row` objects

```
> peopleDF.head(4)
[Row(age=None, name='Alice', pcode='94304'),
 Row(age=30, name='Brayden', pcode='94304'),
 Row(age=19, name='Carla', pcode='10036'),
 Row(age=46, name='Diana', pcode=None)]
```

python/scala

- count # of rows in data frame

```
> peopleDF.count()
5
```

python/scala

Commonly used Queries

- Queries (like transformation) return another DataFrame

describe(cols)	calculate summary statistics of columns
select(cols)	Selects a set of columns based on expressions
groupBy(col1, col2,..)	Groups DataFrame using the specified columns so we can run aggregation on them
filter(conditionExpr)	Filters based on given SQL expression
distinct()	Returns a new DataFrame that contains only unique rows
limit(n)	a new DF with the first **n** rows of this DataFrame
sort(cols); orderBy(cols)	Returns a new DataFrame sorted by the specified column(s)
join(other, joinExpr, joinType)	joins this DataFrame with a second DataFrame using the join expression (types include inner, outer, left_outer, etc)

https://spark.apache.org/docs/latest/api/python/pyspark.sql.html#pyspark.sql.DataFrame (python)
http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.Dataset (scala)

DataFrame Query – `Limit` & `Describe`

- Example: A basic query with `limit`

```
peopleDF.limit(3).show()
```

age	name	pcode
null	Alice	94304
30	Brayden	94304
19	Carla	10036
46	Diana	null
null	Étienne	94104

age	name	pcode
null	Alice	94304
30	Brayden	94304
19	Carla	10036

Output of `show()`

- Example: `Describe`

```
peopleDF.describe().show()
+-------+------------------+-------+------------------+
|summary|               age|   name|             pcode|
+-------+------------------+-------+------------------+
|  count|                 3|      5|                 4|
|   mean|31.666666666666668|   null|           73187.0|
| stddev|13.576941236277534|   null|42100.772233614276|
|    min|                19|  Alice|             10036|
|    max|                46|Étienne|             94304|
+-------+------------------+-------+------------------+
```

Specify a column or columns

- Most DataFrame transformations require you to specify a column or columns
  - `select(column1, column2, …)`
  - `orderBy(column1, column2, …)`
- For many simple queries, you can just specify the column name as a string
  - `peopleDF.select("firstName","lastName")`
- Some types of transformations use column references or column expressions instead of column name strings

```python
# two ways of referencing a col.
peopleDF['age']
peopleDF.age
# a col expression
peopleDF.age*10
```

python

```scala
# two ways of referencing a col
peopleDF("age")
$"age"
# a col expression
peopleDF("age")*10
```

scala

Column Expressions

- Using column references to create column expressions
  - Arithmetic operators such as `+`, `-`, `%`, `/`, and `*`
  - Comparative and logical operators such as `>`, `<`, `& (and)` and `| (or)`
    - The equality comparator is === in Scala, and == in Python
- DataFrame's column methods (use dot notation)
  - String methods such as `contains`, `like`, `isin`, `substr`, `startwith`, `rlike`
    - `df.name.contains('smith')` : if the name contains sub string "smith"
    - `df.name.like("A%")`: sql style like operator
    - `df.name.substr(1,3).alias("short_name")`: first three letters of names.
    - `df.name.isin("Bob", "Mike")`

For the full list of operators and methods, see the API documentation for Column
https://spark.apache.org/docs/latest/api/python/pyspark.sql.html#pyspark.sql.Column

Column Expressions (continue)

- `alias` and `cast(datatype)`:
  - `df.age.cast("string").alias("age2")`
- SQL style methods such as `isNull`, `isNotNull`, and `NaN` (not a number)
  - `df.height.isNull()`
- Sorting methods such as `asc()` and `desc()`
  - Work only when used in `sort/orderBy`
  - E.g. `df.orderBy(df.name.desc())`

- Column operations via built-in SQL functions
  - `pyspark.sql.functions` module has a host of SQL functions that can be used with the column expressions. Those correspond to SQL functions you can use in your SQL queries.
  - E.g. `avg, datediff, lower, rand, explode`

```
import pyspark.sql.functions as f
df.select(f.explode(f.split(df.field,",")))
```

We import SQL functions as f to avoid conflicts with python's math functions

DataFrame Query - Select

- `select(*cols)`
  - `Cols` can be strings or column expressions
- `selectExpr()` accepts sql expressions

age	name	pcode
null	Alice	94304
30	Brayden	94304
19	Carla	10036
46	Diana	null
null	Étienne	94104

peopleDF.select("age")

age
null
30
19
46
null

peopleDF.select("name","age")

name	age
Alice	null
Brayden	30
Carla	19
Diana	46
Étienne	null

More Examples

```
.select('*')
.select('name', 'age')
.select(df.name, (df.age + 10).alias('age'))
.selectExpr("age * 2", "abs(age)")
```

```
.select("colA", "colB")
.select($"colA", $"colB")
.select($"colA", $"colB" + 1)
.selectExpr("colA", "colB as newName", "abs(colC)")
```

DataFrame Query - Filter

- Filtering rows using the given condition, which could be Column expression of Boolean type or a string of SQL expression.

age	name	pcode
null	Alice	94304
30	Brayden	94304
19	Carla	10036
46	Diana	null
null	Étienne	94104

```
peopleDF.
filter("age >21")
```

age	name	pcode
30	Brayden	94304
46	Diana	null

More Examples

```
.filter(peopleDF.age > 20)
.filter("age > 20")
.filter((peopleDF.age > 20) & (peopleDF.age < 35))
.where(peopleDF.age == 30)
.where("age > 20 and age < 35")
```

- `where()` is an alias for `filter()`

```
.filter($"age" > 20)
.where($"age" > 20)
.filter("age > 20")
```

Querying DataFrames - Sort

- sort(*cols): orderBy is an alias of sort

```
peopleDF.sort(peopleDF.age.desc())
```

```
peopleDF.sort(peopleDF("age").desc)
```

.asc and **.desc** are column expression methods used with **sort**

age	name	pcode
null	Alice	94304
30	Brayden	94304
19	Carla	10036
46	Diana	null
null	Étienne	94104

age	name	pcode
46	Diana	null
30	Brayden	94304
19	Carla	10036
null	Alice	94304
null	Étienne	94104

More Examples

```
.orderBy(df.age.desc())
.sort("age",ascending=False)
.sort(['age','name'],ascending=[0,1])
.sort(df.age.desc(), "name")
```

```
.sort($"col1", $"col2".desc)
.sort("sortcol")
```

"Manipulating" DataFrame

- Queries return another DataFrame

withColumn(colName,col)	Return a new DF by adding or replacing a column
withColumnRenamed(existing, new)	Returns a new DF by renaming an existing column
sample()	Take a sample from the DF
sampleBy()	Return a stratified sample
replace(from,to, *subset*)	Return a DF by doing a search and replace
describe()	Returns a new DataFrame sorted by the specified column(s)
fillna(value)	Replace null values with new value
drop(col)	Remove a column
dropDuplicates(*subset*)	Remove duplicate rows, optional within certain columns

https://spark.apache.org/docs/latest/api/python/pyspark.sql.html#pyspark.sql.DataFrame (python)
http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.Dataset (scala)

"Manipulating" DataFrame Examples

- Add columns: the new DF has a new column 'age2' added.

```
peopleDF.withColumn('age2', peopleDF.age + 2)
```

- Rename a column

```
peopleDF.withColumnRenamed('age2','age_new')
```

- Drop duplicate rows

```
peopleDF.select('age','gender').dropDuplicates().show()
```

- Fill the NA values with a new value: `fillna` or `na.fill`

```
peopleDF.fillna(0,"age").show()
peopleDF.na.fill(0,"age").show()
```

"Manipulating" DataFrame Examples

- `Drop` Example: the new DF drops the age column

```
peopleDF.drop('age')
```

- `Replace` example: replace Alice -> A, Bob → B in the `name` column.

```
peopleDF.na.replace(['Alice', 'Bob'], ['A', 'B'], 'name').show()
+----+------+----+
| age|height|name|
+----+------+----+
|  10|    80|   A|
|   5|  null|   B|
|null|  null| Tom|
|null|  null|null|
+----+------+----+
```

Data Frames and Spark SQL

AGGREGATION AND WINDOWING

Aggregation

- To execute an aggregation on a set of grouped values, use `groupBy` combined with an aggregation function
- `groupBy` takes one or more column names or references
  - In Scala, returns a **RelationalGroupedDataset** object
  - In Python, returns a **GroupedData** object
- Returned objects provide aggregation functions, including
  - `count`
  - `max` and `min`
  - `mean` (and its alias `avg`)
  - `sum`
  - `pivot`
  - `agg` (aggregates using additional aggregation functions)

```
peopleDF.groupBy("pcode").count().show()
+-----+-----+
|pcode|count|
+-----+-----+
|94020| 2   |
|87501| 1   |
|02134| 2   |
+-----+-----+
```

Aggregate Examples

- `groupBy(*col)`: Group the DataFrame using the specified column(s).
  - It is usually followed by an aggregate function, e.g. `count, avg, min, max, sum`
  - An `agg()` function accepts a map of fields to type of aggregate functions.
- Group all rows together:

```
iris = spark.read.json('iris.json')
# group all, average all [numerical cols]
iris.groupBy().avg().show()
# group all, average petalLength
iris.groupBy().avg('petalLength').show()
# group all, average petalLength and petalWidth
iris.groupBy().mean('petalLength', 'petalWidth').show()
```

- Group by values of some fields

```
# group by petalLength & petalWidth, max sepalLength
iris.groupBy(['petalLength','petalWidth']).max('sepalLength').show()
# group by species, average sepalLength, max sepalWidth
iris.groupBy("species").agg({'sepalLength': 'mean','sepalWidth':'max'}).show()
from pyspark.sql import functions as f
iris.groupBy("species").agg(f.mean(iris.sepalLength).alias('avg_sepal'),f.max(iris.sepalWidth)).show()
```

Aggregate Examples

- Scala examples

```
val iris = spark.read.json('iris.json')
#group by species, avg all
iris.groupBy("species").avg().show()
#group by petalLength & petalWidth, average sepalLength, max sepalWidth
df.groupBy($"petalLength", $"petalWidth").agg(Map(
  "sepalLength" -> "avg",
  "sepalWidth" -> "max"
)).show()
```

- Other aggregate functions

  `countDistinct` returns the number of unique items in a group
  `approx_count_distinct` returns an approximate counts of unique items (Much faster than a full count)
  `stddev` calculates the standard deviation for a group of values
  `var_sample/var_pop` calculates the variance for a group of values
  `covar_samp/covar_pop` calculates the sample and population covariance of a group of values
  `corr` returns the correlation of a group of values

Window functions

- To answer questions such as "What is the difference between the revenue of each product and the revenue of the best selling product in the same category as that product?"
- To use windows functions, on need to
  - A. Define the window specification.
  - B. Mark an function to use the given window specification
    - Special window functions include: `rank, dense_rank, lag, lead, first_value, last_value, percent_rank, row_number, etc.`

```
from pyspark.sql.window import Window

wind = Window.partitionBy(iris.species).orderBy(iris.sepalLength.desc())

iris.select(iris.species,f.max(iris.sepalLength).over(wind).alias("max_sep"),iris.sepalLength, \
            (f.max(iris.sepalLength).over(wind)-iris.sepalLength).alias("diff_sep")).show()
+---------+-------+-----------+-------------------+
|  species|max_sep|sepalLength|           diff_sep|
+---------+-------+-----------+-------------------+
|virginica|    7.9|        7.9|                0.0|
|virginica|    7.9|        7.7|0.20000000000000018|
|virginica|    7.9|        7.7|0.20000000000000018|
|virginica|    7.9|        7.7|0.20000000000000018|
|virginica|    7.9|        7.7|0.20000000000000018|
|virginica|    7.9|        7.6| 0.3000000000000007|
|virginica|    7.9|        7.4|                0.5|
...
```

https://databricks.com/blog/2015/07/15/introducing-window-functions-in-spark-sql.html

Data Frames and Spark SQL

JOIN DATAFRAMES

Join DataFrames

- **`df.join(df2, joinExpr, joinType)`**
  - joins this DataFrame with a second DataFrame using the join expression
  - `joinExpr` can be:
    - a string for the join column name (col on both sides), e.g. `"ssn"`
    - A list of column names, e.g. `["firstname","lastname"]`
    - A join expression: e.g.
      - `df.id ==df2.id`
      - `[df.fname==df2.fname,df.lname==df2.lname]`
  - `joinType` includes *`inner`*, *`outer`*`(or` *`full/full_outer`*`)`, *`left_outer`* `(or` *`left`*`)`, *`right_outer`* `(or` *`right`*`)`, *`cross`*, *`left_semi`*, *`left_anti`*

Join Example: inner join

people-no-pcode.csv

```
pcode,lastName,firstName,age
02134,Hopper,Grace,52
,Turing,Alan,32
94020,Lovelace,Ada,28
87501,Babbage,Charles,49
02134,Wirth,Niklaus,48
```

pcodes.csv

```
pcode,city,state
02134,Boston,MA
94020,Palo Alto,NM
87501,Santa Fe,CA
60645,Chicago,IL
```

```
# load left table into a dataFrame nopcode
# load right table into a dataFrame pcodes
nopcode.join(pcodes, "pcode").show()
+-----+--------+---------+---+---------+-----+
|pcode|lastName|firstName|age|     city|state|
+-----+--------+---------+---+---------+-----+
|02134|  Hopper|    Grace| 52|   Boston|   MA|
|94020|Lovelace|      Ada| 28|Palo Alto|   CA|
|87501| Babbage|  Charles| 49| Santa Fe|   NM|
|02134|   Wirth|  Niklaus| 48|   Boston|   MA|
+-----+--------+---------+---+---------+-----+
```

Join Example: outer join

people-no-pcode.csv

```
pcode,lastName,firstName,age
02134,Hopper,Grace,52
,Turing,Alan,32
94020,Lovelace,Ada,28
87501,Babbage,Charles,49
02134,Wirth,Niklaus,48
```

pcodes.csv

```
pcode,city,state
02134,Boston,MA
94020,Palo Alto,NM
87501,Santa Fe,CA
60645,Chicago,IL
```

```
nopcode.join(pcodes, "pcode", "left_outer").show()  # or
nopcode.join(pcodes, nopcode.pcode == pcodes.pcode, "left_outer").show()
```

```
nopcode.join(pcodes,nopcode("pcode") === pcodes("pcode"), "left_outer").show()
```

```
+-----+--------+---------+---+---------+-----+
|pcode|lastName|firstName|age|     city|state|
+-----+--------+---------+---+---------+-----+
|02134|  Hopper|    Grace| 52|   Boston|   MA|
| null|  Turing|     Alan| 32|     null| null|
|94020|Lovelace|      Ada| 28|Palo Alto|   CA|
|87501| Babbage|  Charles| 49| Santa Fe|   NM|
|02134|   Wirth|  Niklaus| 48|   Boston|   MA|
+-----+--------+---------+---+---------+-----+
```

Data Frames and Spark SQL

INTERACT WITH TABLES AND VIEWS

Run SQL Queries

- SQL queries and DataFrame transformations provide equivalent functionality
- The following Python examples are equivalent

```
myDF = spark.sql("SELECT * FROM people WHERE pcode = 94020")
myDF = spark.read.table("people").where("pcode=94020")
```

- Both are executed as series of transformations - Optimized by the Catalyst optimizer

Query Files

- You can query directly from Parquet or JSON files that are not Hive tables

```
# save myDF as parquet format
spark.\
 sql("SELECT * FROM parquet.`/path/to/my.parquet` WHERE firstName LIKE 'A%' ").show()

+-----+--------+---------+---+
|pcode|lastName|firstName|age|
+-----+--------+---------+---+
|94020|  Turing|     Alan| 32|
|94020|Lovelace|      Ada| 28|
+-----+--------+---------+---+
```

Create Views

- You can also query a view
  - Views provide the ability to perform SQL queries on a DataFrame or Dataset
- Views are temporary
  - **Regular views** can only be used within a single Spark session
  - **Global views** can be shared between multiple Spark sessions within a single spark application
- Creating a view
  - `DataFrame.createTempView(view-name)`
  - `DataFrame.createOrReplaceTempView(view-name)`
  - `DataFrame.createGlobalTempView(view-name)`

Query a View

- After defining a DataFrame view, you can query with SQL just as with a table

```
spark.read.load("/path/my.parquet"). \
  select("firstName", "lastName"). \
  createTempView("user_names")

spark.sql( \
  "SELECT * FROM user_names WHERE firstName LIKE 'A%'" ). \
  show()
+---------+--------+
|firstName|lastName|
+---------+--------+
| Alan    | Turing |
| Ada     |Lovelace|
+---------+--------+
```

Catalog APIs

- Use the Catalog API to explore tables and manage views
  - The entry point for the Catalog API is `spark.catalog`
- Functions include
  - `listDatabases` returns a Dataset (Scala) or list (Python) of existing databases
  - `setCurrentDatabase(dbname)` sets the current default database for the session
    - Equivalent to the USE statement in SQL
  - `listTables` returns a Dataset (Scala) or list (Python) of tables and views in the current database
  - `listColumns(tablename)` returns a Dataset (Scala) or list (Python) of the columns in the specified table or view
  - `dropTempView(viewname)` removes a temporary view

```
spark.catalog.listTables.show
+------------+--------+-----------+---------+-----------+
|        name|database|description|tableType|isTemporary|
+------------+--------+-----------+---------+-----------+
|      people| default|       null| EXTERNAL|      false|
|  user_names| default|       null|TEMPORARY|       true|
+------------+--------+-----------+---------+-----------+
```

Data Frames and Spark SQL

DATAFRAME AND RDD

DataFrames and RDDs (1)

- DataFrames are built on RDDs
  - Base RDDs contain Row objects
  - Use rdd to get the underlying RDD

```
peopleRDD = peopleDF.rdd
```

peopleDF

age	name	pcode
null	Alice	94304
30	Brayden	94304
19	Carla	10036
46	Diana	null
null	Étienne	94104

peopleRDD

Row[null,Alice,94304]
Row[30,Brayden,94304]
Row[19,Carla,10036]
Row[46,Diana,null]
Row[null,Étienne,94104]

DataFrames and RDDs (2)

- Row RDDs have all the standard Spark actions and transformations
  - Actions – collect, take, count, etc.
  - Transformations – map, flatMap, filter, etc.

Working with Row Objects

- **The syntax for extracting data from Rows depends on language**
- **Python**
  - Column names are object attributes
    - `row.age` – return age column value from row
- **Scala**
  - Use Array-like syntax
    - `row(0)` – returns element in the first column
    - `row(1)` – return element in the second column
    - etc.
  - Use type-specific `get` methods to return typed values
    - `row.getString(n)` – returns n^{th} column as a String
    - `row.getInt(n)` – returns n^{th} column as an Integer, etc.

Example: Extracting Data from Rows

- Extract data from Rows

```
peopleRDD = peopleDF.rdd
peopleByPCode = peopleRDD \
   .map(lambda row:(row.pcode,row.name)) \
   .groupByKey()
```

```
val peopleRDD = peopleDF.rdd
peopleByPCode = peopleRDD.
   map(row => (row(2),row(1))).
   groupByKey())
```

```
Row[null,Alice,94304]
Row[30,Brayden,94304]
Row[19,Carla,10036]
Row[46,Diana,null]
Row[null,Étienne,94104]
```

```
(94304,Alice)
(94304,Brayden)
(10036,Carla)
(null,Diana)
(94104,Étienne)
```

```
(null,[Diana])
(94304,[Alice,Brayden])
(10036,[Carla])
(94104,[Étienne])
```

Data Frames and Spark SQL

COMPARING SPARK SQL, IMPALA AND HIVE-ON-SPARK

Query Tables with SQL

- Data analysts often need to query Hive metastore tables
- There are several ways to use SQL with tables in Hive
  - Apache Impala
  - Apache Hive
    - Running on Hadoop MapReduce, Tez or Spark
  - Spark SQL API (SparkSession.sql)

Apache Hive

- Apache Hive
  - Runs using either Spark or MapReduce
  - In most cases, Hive on Spark has much better performance
  - Very mature
  - High stability and resilience
- Best for
  - Batch ETL processing
  - Typical job: minutes to hours

Impala

- **Impala** is a specialized SQL engine
  - Better performance than Spark SQL
  - More mature
  - Robust security using Apache Sentry
  - Highly optimized
  - Low latency
- **Best for**
  - Interactive and ad hoc queries
  - Data analysis
  - Integration with third-party visual analytics and business intelligence tools such as Tableau, Zoomdata, or Microstrategy
- Typical job: seconds or less

Spark SQL

- Spark SQL API
  - Mixed procedural and SQL applications
  - Supports a rich ecosystem of related APIs for machine learning, streaming, statistical computations
  - Catalyst optimizer for good performance
  - Supports Python, a common language for data scientists
- Best for
  - Complex data manipulation and analytics
  - Integration with other data systems and APIs
  - Machine learning
  - Streaming and other long-running applications

Essential Points

- Spark SQL is a Spark API for handling structured and semi-structured data
- Entry point is a Spark Session object: `spark`
- DataFrames are the key unit of data
  - DataFrames are based on an underlying RDD of Row objects
  - DataFrames query methods return new DataFrames; similar to RDD transformations
  - The full Spark API can be used with Spark SQL Data by accessing the underlying RDD
  - Spark SQL is not a replacement for a database, or a specialized SQL engine like Impala
- Spark SQL is most useful for ETL or incorporating structured data into other applications