INTRODUCTION TO APACHE SPARK

3. SPARK SQL AND DATAFRAMES

REMINDER: HOMEWORKS

- Expected by email before the following class
- Submit the exported notebook (.ipynb extension)
- File name should include both student name and no space (FirstName1LastName1_FirstName2LastName2.py) for instance

OVERVIEW

- Spark SQL is a library included in Apache Spark since version 1.3
- Its main goal is to provide an easier interface to process tabular data
- Instead of RDDs, we deal with DataFrames
- Starting from Spark 1.6, there is also the concept of Datasets, but only for Scala and Java

CONTEXTS AND SPARKSESSION

- Before Spark 2, there was only SparkContext and SQLContext
- All core functionality was accessed with SparkContext
- All SQL functionality needed the SQLContext, which can be created from an SparkContext
- Starting from Spark 2.0, there is now the SparkSession class
- SparkSession is now the global entry-point for everything Spark-related

CREATING A SPARK SESSION

```
from pyspark import SparkConf, SparkContext
from pyspark.sql import SQLContext, SparkSession

# Before Spark 2
conf = SparkConf().setAppName(appName).setMaster(master)
sc = SparkContext(conf = conf)
sql_context = new SQLContext(sc)

# Since Spark 2.0
spark = SparkSession \
    .builder \
    .appName(appName) \
    .master(master)
    .config("spark.some.config.option", "some-value") \
    .getOrCreate()
```

DATAFRAME

- The main entity of Spark SQL is the DataFrame
- A DataFrame is actually an RDD of `Rows` with a Schema definition
- The schema defines the names of the columns and their data types
- `Row` is a class representing a row of the DataFrame.
- It can be used almost as a List, with its size equal to the number of columns in the schema.

EXAMPLES

```
>>> from pyspark.sql import Row
>>> row1 = Row(name="John", age=21)
>>> row2 = Row(name="James", age=32)
>>> row3 = Row(name="Jane", age=18)
>>> row1['name']
'John'
```

EXAMPLES

```
>>> df = spark.createDataFrame([row1, row2, row3])
>>> df.printSchema()
>>> df.show()
```

```
|-- name: string (nullable = true)
|-- age: long (nullable = true)

+-----+---+
| name|age|
+-----+---+
| John| 21|
|James| 32|
| Jane| 18|
+-----+---+
```

EXAMPLES

```
>>> print(df.rdd.toDebugString())
```

```
(8) MapPartitionsRDD[209] at javaToPython at NativeMethodAcces
 |  MapPartitionsRDD[208] at javaToPython at NativeMethodAccess
 |  MapPartitionsRDD[207] at javaToPython at NativeMethodAccess
 |  MapPartitionsRDD[204] at applySchemaToPythonRDD at NativeMe
 |  MapPartitionsRDD[203] at map at SerDeUtil.scala:123 []
 |  MapPartitionsRDD[202] at mapPartitions at SerDeUtil.scala:1
 |  PythonRDD[201] at RDD at PythonRDD.scala:48 []
 |  ParallelCollectionRDD[200] at parallelize at PythonRDD.scal
```

CREATING DATAFRAMES

- We can use the method `createDataFrame` present in the SparkSession instance.
- DataFrames can be created from:
  - a `pandas.DataFrame` object
  - a local python list
  - an RDD
- The full documentation and parameters can be found in the API docs

CREATING DATAFRAMES

```
>>> rows = [Row(name="John", age=21, gender="male"),
            Row(name="James", age=25, gender="female"),
            Row(name="Albert", age=46, gender="male")]
>>> df = spark.createDataFrame(rows)
>>> df.show()
+---+------+------+
|age|gender|  name|
+---+------+------+
| 21|  male|  John|
| 25|female| James|
| 46|  male|Albert|
+---+------+------+
```

CREATING DATAFRAMES

```
>>> column_names = ["name", "age", "gender"]
>>> rows = [["John", 21, "male"],
            ["James", 25, "female"],
            ["Albert", 46, "male"]]
>>> df = spark.createDataFrame(rows, column_names)
>>> df.show()
+------+---+------+
|  name|age|gender|
+------+---+------+
|  John| 21|  male|
| James| 25|female|
|Albert| 46|  male|
+------+---+------+
```

CREATING DATAFRAMES

```
>>> column_names = ["name", "age", "gender"]
>>> rdd = sc.parallelize([
    ("John", 21, "male"),
    ("James", 25, "female"),
    ("Albert", 46, "male")
])
>>> df = spark.createDataFrame(rdd, column_names)
>>> df.show()
```

```
+------+---+------+
|  name|age|gender|
+------+---+------+
|  John| 21|  male|
| James| 25|female|
|Albert| 46|  male|
+------+---+------+
```

SCHEMA AND TYPES

- A DataFrame always contains a schema
- The schema defines the column names and types
- The schema of a DataFrame is represented by the class `types.StructType` (docs)
- When creating a DataFrame, the schema can be either inferred or defined by the user

CREATING A CUSTOM SCHEMA

```
from pyspark.sql.types import *
schema = StructType([StructField("name", StringType(), True),
                     StructField("age", IntegerType(), True),
                     StructField("gender", StringType(), True)
])
rows = [("John", 21, "male")]
df = spark.createDataFrame(rows, schema)
df.printSchema()
df.show()
```

```
root
|-- name: string (nullable = true)
|-- age: integer (nullable = true)
|-- gender: string (nullable = true)

+----+---+------+
|name|age|gender|
+----+---+------+
|John| 21|  male|
+----+---+------+
```

TYPES SUPPORTED BY SPARK SQL

READING DATA FROM SOURCES

- In real use-cases, data is usually read from external sources
- Spark SQL provides many connectors to read from:
  - Text files (csv, json)
  - Distributed tabular files (Parquet, ORC)
  - General relational Databases (via JDBC)
- Specific connectors to many other databases can be found in third-party libraries…
- …or you can implement your own connector for Spark (Scala API).

READING DATA FROM SOURCES

- In all cases, the syntax is similar:

  ```
  spark.read.{source}(path)
  ```

- Spark supports different file systems to look for the data:
  - Local files: `file://`
  - HDFS (hadoop filesystem): `hdfs://`
  - Amazon S3: `s3://`

READING FROM CSV

```
df = spark.read.csv("/path/to/file.csv")
```

```
path = "/path/to/file.csv"
df = spark.read.option("header", "true").csv(path)
```

```
df = spark.read\
          .format("csv")\
          .option("header", "true")\
          .option("sep", ";")\
          .load("/path/to/file.csv")
```

```
df = spark.read.csv(path, sep=";", header=True)
```

READING FROM CSV - MAIN OPTIONS

option	description
`sep`	The separator character
`header`	If "true", the first line contains the column names
`inferSchema`	If "true", the column types will be guessed from the contents
`dateFormat`	A string representing the format of the date columns

The full list of options can be found in the API Docs

READING FROM OTHER FILE TYPES

```
# JSON file
df = spark.read.json("/path/to/file.json")
df = spark.read.format("json").load("/path/to/file.json")
```

```
# Parquet file (distributed tabular data)
df = spark.read.parquet("hdfs:///path/to/file.parquet")
df = spark.read.format("parquet").load("hdfs:///path/to/file.p
```

```
# ORC file (distributed tabular data)
df = spark.read.orc("hdfs:///path/to/file.orc")
df = spark.read.format("orc").load("hdfs:///path/to/file.orc")
```

READING FROM EXTERNAL DATABASES

- JDBC drivers (Java) can be used to read from relational Databases (Oracle, PostgreSQL, MySQL, etc.)
- The java driver file must be uploaded to the cluster before trying to access
- This operation can be very heavy. When available, database-specific connectors provided by third-party libraries should be used.

READING FROM EXTERNAL DATABASES

```
df = spark.read.format("jdbc") \
    .option("url", "jdbc:postgresql:dbserver") \
    .option("dbtable", "schema.tablename") \
    .option("user", "username") \
    .option("password", "p4ssw0rd") \
    .load()
# or
df = spark.read.jdbc(
    url="jdbc:postgresql:dbserver",
    table="schema.tablename"
    properties={
        "user": "username",
        "password": "p4ssw0rd"
    }
)
```

PERFORMING QUERIES

- Spark SQL was created to be compatible with SQL queries
- So it supports actual SQL queries to be performed on DataFrames
- First, the DataFrame must be tagged as a temporary view
- Then, the queries can be applied using `spark.sql`

PERFORMING QUERIES

```
column_names = ["name", "age", "gender"]
rows = [
    ["John", 21, "male"],
    ["Jane", 25, "female"]
]
df = spark.createDataFrame(rows, column_names)

# Create a temporary view from the DataFrame
df.createOrReplaceTempView("new_view")

# Apply the query
query = "SELECT name, age FROM new_view WHERE gender='male'"
men_df = spark.sql(query)
men_df.show()
```

```
+----+---+
|name|age|
+----+---+
|John| 21|
+----+---+
```

USING THE API

- Although allowing SQL queries is a very powerful feature, it's not the best way to code a complex logic
- Errors are harder to find in Strings
- Using queries makes the code less modularizable
- So Spark SQL provides a full API with SQL-like operations
- It's the best way to code complex logic when using Spark SQL

BASIC OPERATIONS

operation	description
`select`	Chooses columns from the table
`where`	Filters rows based on a boolean rule
`limit`	Limits the number of rows
`orderBy`	Sorts the DataFrame based on one or more columns
`alias`	Changes the name of a column
`cast`	Changes the type of a column
`withColumn`	Adds a new column

SELECT

```
# In a SQL query:
df.createOrReplaceTempView("some_table")
query = "SELECT name, age FROM some_table"
spark.sql(query).show()

# Using Spark SQL API:
df.select("name", "age").show()
```

```
+-----+---+
| name|age|
+-----+---+
| John| 21|
| Jane| 25|
+-----+---+
```

WHERE

```
# In a SQL query:
query = "SELECT * FROM table WHERE age > 21"

# Using Spark SQL API:
df.where("age > 21").show()

# alternatively:
df.where(df['age'] > 21).show()
df.where(df.age > 21).show()
df.select("*").where("age > 21").show()
```

```
+----+---+------+
|name|age|gender|
+----+---+------+
|Jane| 25|female|
+----+---+------+
```

LIMIT

```
# In a SQL query:
query = "SELECT * FROM table LIMIT 1"

# Using Spark SQL API:
df.limit(1).show()
# or
df.select("*").limit(1).show()

# Note: The result is not deterministic!
```

```
+----+---+------+
|name|age|gender|
+----+---+------+
|Jane| 25|female|
+----+---+------+
```

ORDER BY

```
# In a SQL query:
query = "SELECT * FROM table ORDER BY name ASC"

# Using Spark SQL API:
df.orderBy(df.name.asc()).show()
```

```
+----+---+------+
|name|age|gender|
+----+---+------+
|Jane| 25|female|
|John| 21|  male|
+----+---+------+
```

ALIAS (NAME CHANGE)

```
# In a SQL query:
query = "SELECT name, age, gender AS sex FROM table"

# Using Spark SQL API:
df.select(df.name, df.age, df.gender.alias('sex')).show()
```

```
+----+---+------+
|name|age|   sex|
+----+---+------+
|John| 21|  male|
|Jane| 25|female|
+----+---+------+
```

CAST (TYPE CHANGE)

```
# In a SQL query:
query = "SELECT name, cast(age AS float) AS age_f FROM table"

# Using Spark SQL API:
df.select(df.name, df.age.cast("float").alias("age_f")).show()

# or
new_age_col = df.age.cast("float").alias("age_f")
df.select(df.name, new_age_col).show()
```

```
+----+-----+
|name|age_f|
+----+-----+
|John| 21.0|
|Jane| 25.0|
+----+-----+
```

ADDING NEW COLUMNS

```
# In a SQL query:
query = "SELECT *, 12*age AS age_months FROM table"

# Using Spark SQL API:
df.withColumn("age_months", df.age * 12).show()
# or
df.select("*", (df.age * 12).alias("age_months")).show()

# Note: Using withColumn is preferable
```

```
+----+---+------+----------+
|name|age|gender|(age * 12)|
+----+---+------+----------+
|John| 21|  male|       252|
|Jane| 25|female|       300|
+----+---+------+----------+
```

BASIC OPERATIONS

- The full list of operations that can be applied to a `DataFrame` can be found in the DataFrame docs
- The list of operations on `Columns` can be found in the Column docs

COLUMN FUNCTIONS

- Most of the time we need chain many transformations using one or more functions
- Spark SQL has a package called `functions` with many functions available for that
- Some of those functions are only for aggregations
  - For example: `avg`, `sum`, etc
  - We will cover them later
- Some others are for column transformation or operations
  - examples: `substr`, `concat`, `datediff`, `floor`, etc.
- The full list with descriptions is, as usual, in the API docs

COLUMN FUNCTIONS

- To use these functions, we first need to import them:

```
from pyspark.sql import functions as fn
```

NUMERIC FUNCTIONS EXAMPLES

```
from pyspark.sql import functions as fn

df = spark.createDataFrame([
    ("garnier", 3.49),
    ("elseve", 2.71)
], ["brand", "cost"])

round_cost = fn.round(df.cost, 1)
floor_cost = fn.floor(df.cost)
ceil_cost = fn.ceil(df.cost)
df.withColumn('round', round_cost)\
  .withColumn('floor', floor_cost)\
  .withColumn('ceil', ceil_cost).show()
```

```
+-------+----+-----+-----+----+
|  brand|cost|round|floor|ceil|
+-------+----+-----+-----+----+
|garnier|3.49|  3.5|    3|   4|
| elseve|2.71|  2.7|    2|   3|
+-------+----+-----+-----+----+
```

STRING FUNCTIONS EXAMPLES

```
from pyspark.sql import functions as fn

df = spark.createDataFrame([
    ("John", "Doe"),
    ("Mary", "Jane")
], ["first_name", "last_name"])

last_name_initial = fn.substring(df.last_name, 0, 1)
name = fn.concat_ws(" ", df.first_name, last_name_initial)

df.withColumn("name", name).show()
```

```
+----------+---------+------+
|first_name|last_name|  name|
+----------+---------+------+
|      John|      Doe|John D|
|      Mary|     Jane|Mary J|
+----------+---------+------+
```

DATE FUNCTIONS EXAMPLES

```
from datetime import date
from pyspark.sql import functions as fn

df = spark.createDataFrame([
    (date(2015, 1, 1), date(2015, 1, 15)),
    (date(2015, 2, 21), date(2015, 3, 8)),
], ["start_date", "end_date"])

days_between = fn.datediff(df.end_date, df.start_date)
start_month = fn.month(df.start_date)
df.withColumn('days_between', days_between)\
  .withColumn('start_month', start_month)\
  .show()
```

```
+----------+----------+------------+-----------+
|start_date|  end_date|days_between|start_month|
+----------+----------+------------+-----------+
|2015-01-01|2015-01-15|          14|          1|
|2015-02-21|2015-03-08|          15|          2|
+----------+----------+------------+-----------+
```

CONDITIONAL TRANSFORMATIONS

- In the functions package there is a special function called `when`
- This function is used to create a new column which value depends on the value of other columns
- `otherwise` is used to match "the remainder"
- Combination between conditions can be done using "&" for "and" and "|" for "or"

EXAMPLES

```
df = spark.createDataFrame([
    ("John", 21, "male"),
    ("Jane", 25, "female"),
    ("Albert", 46, "male"),
    ("Brad", 49, "super-hero")
], ["name", "age", "gender"])

supervisor = fn.when((df.gender == 'male', 'Mr. Smith') & \
               (df.gender == 'female', 'Miss Jones'))\
               .otherwise('NA')
df.withColumn("supervisor", supervisor).show()
```

```
+------+---+----------+----------+
|  name|age|    gender|supervisor|
+------+---+----------+----------+
|  John| 21|      male| Mr. Smith|
|  Jane| 25|    female|Miss Jones|
|Albert| 46|      male| Mr. Smith|
|  Brad| 49|super-hero|        NA|
+------+---+----------+----------+
```

EXAMPLES

```
df = spark.createDataFrame([
    ("John", 21, "male"),
    ("Jane", 25, "female"),
    ("Albert", 46, "male"),
    ("Brad", 49, "super-hero")
], ["name", "age", "gender"])

supervisor = fn.when(df.gender == 'male', 'Mr. Smith')\
               .when(df.gender == 'female', 'Miss Jones')\
               .otherwise('NA')
df.withColumn("supervisor", supervisor).show()
```

```
+------+---+----------+----------+
|  name|age|    gender|supervisor|
+------+---+----------+----------+
|  John| 21|      male| Mr. Smith|
|  Jane| 25|    female|Miss Jones|
|Albert| 46|      male| Mr. Smith|
|  Brad| 49|super-hero|        NA|
+------+---+----------+----------+
```

USER DEFINED FUNCTIONS

- When you need a transformation that is not available in the functions package, you can create an User Defined Function (UDF)
- Beware, UDFs' can be very slow.
- So, it should be used only when you are sure the operation cannot be done with by combining existing functions
- To create an UDF, you use `functions.udf`, passing a lambda or a named functions
- It is similar to the `map` operation of RDDs

EXAMPLE

```
from pyspark.sql import functions as fn
from pyspark.sql.types import StringType

df = spark.createDataFrame([(1, 3), (4, 2)], ["first", "second

def my_func(col_1, col_2):
    if (col_1 > col_2):
        return "{} is bigger than {}".format(col_1, col_2)
    else:
        return "{} is bigger than {}".format(col_2, col_1)

my_udf = fn.udf(my_func, StringType())
df.withColumn("udf", my_udf(df['first'], df['second'])).show()
```

```
+-----+------+-----------------+
|first|second|              udf|
+-----+------+-----------------+
|    1|     3|3 is bigger than 1|
|    4|     2|4 is bigger than 2|
+-----+------+-----------------+
```

PANDASUDFS

- Spark 3 introduced a faster, vectorized variant
- It leverages some of Pandas functions (API docs)

```
import pandas as pd
from pyspark.sql.functions import pandas_udf

@pandas_udf('long')
def pandas_plus_one(s: pd.Series) -> pd.Series:
    return s + 2

spark.range(3).select(pandas_plus_one("id")).collect()
```

```
[Row(id=2), Row(id=3), Row(id=4)]
```

PERFORMING JOINS

- Spark SQL supports joins between 2 DataFrames
- As in normal SQL, a join rule must be defined
  - The rule can either be a set of join keys, or a conditional rule
  - Join with conditional rules in Spark can be very heavy
- Many types of joins are available:

EXAMPLES

```
from datetime import date
products = spark.createDataFrame([
    ('1', 'mouse', 'microsoft', 39.99),
    ('2', 'keyboard', 'logitech', 59.99),
], ['prod_id', 'prod_cat', 'prod_brand', 'prod_value'])

purchases = spark.createDataFrame([
    (date(2017, 11, 1), 2, '1'),
    (date(2017, 11, 2), 1, '1'),
    (date(2017, 11, 5), 1, '2'),
], ['date', 'quantity', 'prod_id'])
# The default join type is the "INNER" join
purchases.join(products, 'prod_id').show()
```

```
+-------+----------+--------+--------+----------+----------+
|prod_id|      date|quantity|prod_cat|prod_brand|prod_value|
+-------+----------+--------+--------+----------+----------+
|      1|2017-11-01|       2|   mouse| microsoft|     39.99|
|      1|2017-11-02|       1|   mouse| microsoft|     39.99|
|      2|2017-11-05|       1|keyboard|  logitech|     59.99|
+-------+----------+--------+--------+----------+----------+
```

EXAMPLES

```
# We can also use a query string (not usually recommended)
products.createOrReplaceTempView("products")
purchases.createOrReplaceTempView("purchases")
query = """SELECT * FROM
    (purchases AS prc INNER JOIN
     products AS prd
     ON prc.prod_id = prd.prod_id)"""
spark.sql(query).show()
```

```
+----------+--------+-------+-------+--------+----------+-----
|      date|quantity|prod_id|prod_id|prod_cat|prod_brand|prod_
+----------+--------+-------+-------+--------+----------+-----
|2017-11-01|       2|      1|      1|   mouse| microsoft|
|2017-11-02|       1|      1|      1|   mouse| microsoft|
|2017-11-05|       1|      2|      2|keyboard|  logitech|
+----------+--------+-------+-------+--------+----------+-----
```

EXAMPLES

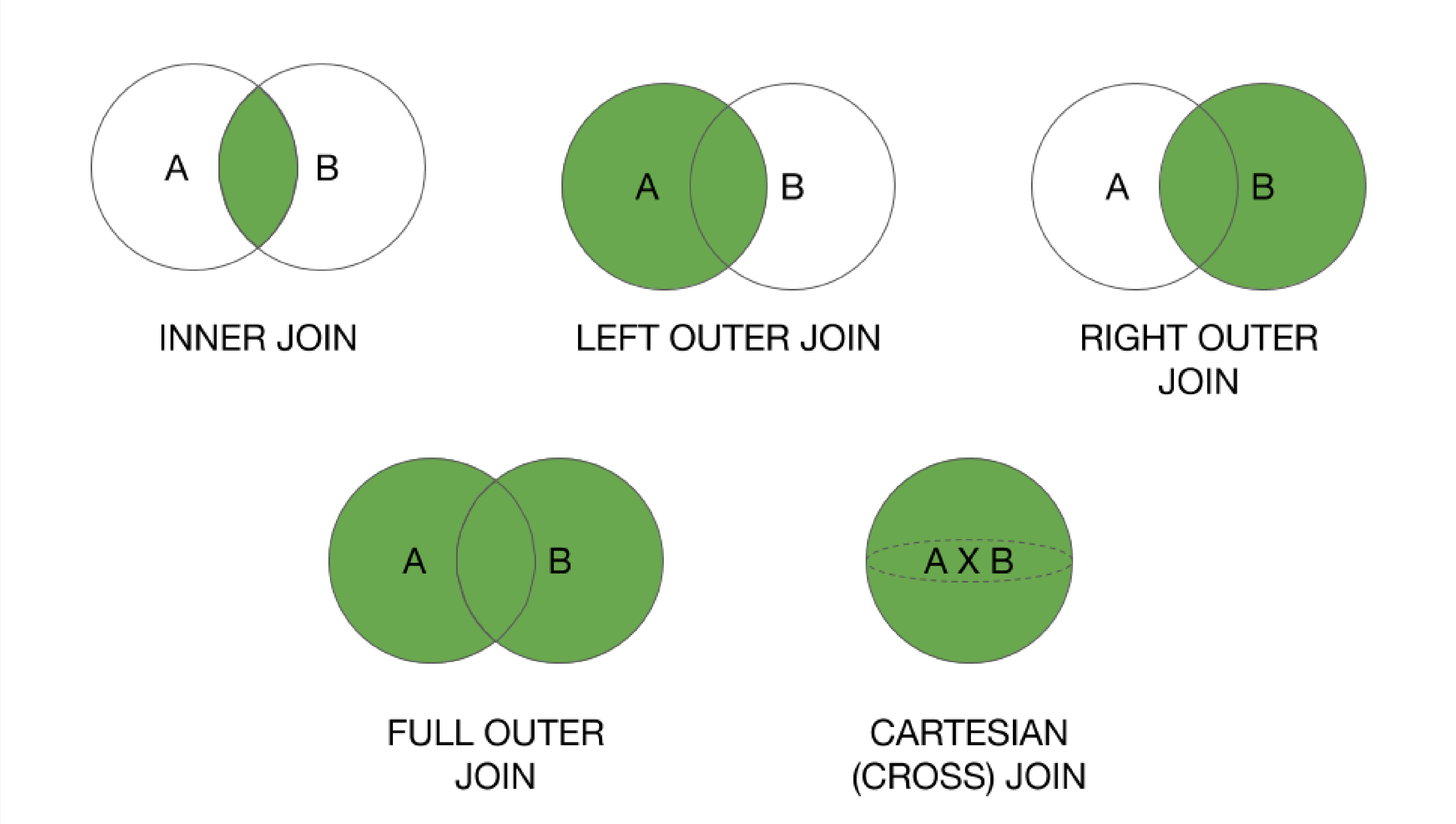
```
new_purchases = spark.createDataFrame([
    (date(2017, 11, 1), 2, '1'),
    (date(2017, 11, 2), 1, '3'),
], ['date', 'quantity', 'prod_id_x'])

# The default join type is the "INNER" join
join_rule = new_purchases.prod_id_x == products.prod_id
new_purchases.join(products, join_rule, 'left').show()
```

```
+----------+--------+---------+-------+--------+----------+---
|      date|quantity|prod_id_x|prod_id|prod_cat|prod_brand|pro
+----------+--------+---------+-------+--------+----------+---
|2017-11-02|       1|        3|   null|    null|      null|
|2017-11-01|       2|        1|      1|   mouse| microsoft|
+----------+--------+---------+-------+--------+----------+---
```

JOIN TYPES

SQL Join Type	In Spark
CROSS	cross
INNER	inner
FULL OUTER	outer, full, fullouter+
LEFT ANTI	leftanti
LEFT OUTER	leftouter, left
LEFT SEMI	leftsemi
RIGHT OUTER	rightouter, right

JOIN TYPES



A
B
INNER JOIN
A
B
LEFT OUTER JOIN
A
B
RIGHT OUTER JOIN
A
B
FULL OUTER JOIN
A X B
CARTESIAN (CROSS) JOIN

PERFORMING AGGREGATIONS

- Maybe the most used operations in SQL and Spark SQL
- Similar to SQL, we use "group by" to perform aggregations
- For simple aggregations, we can call the function just after `groupBy`
- Usually, we use `groupBy().agg()`
- There are many aggregation functions in `pyspark.sql.functions`
- Some examples:
  - Numeric: `fn.avg`, `fn.sum`, `fn.min`, `fn.max`, etc.
  - General: `fn.first`, `fn.last`, `fn.count`, `fn.countDistinct`, etc.

EXAMPLES

```
products = spark.createDataFrame([
    ('1', 'mouse', 'microsoft', 39.99),
    ('2', 'mouse', 'microsoft', 59.99),
    ('3', 'keyboard', 'microsoft', 59.99),
    ('4', 'keyboard', 'logitech', 59.99),
    ('5', 'mouse', 'logitech', 29.99),
], ['prod_id', 'prod_cat', 'prod_brand', 'prod_value'])

products.groupBy('prod_cat').avg('prod_value').show()
#or
from pyspark.sql import functions as fn
products.groupBy('prod_cat').agg(fn.avg('prod_value')).show()
```

```
+--------+-----------------+
|prod_cat|  avg(prod_value)|
+--------+-----------------+
|keyboard|            54.99|
|   mouse|43.32333333333333|
+--------+-----------------+
```

EXAMPLES

```
from pyspark.sql import functions as fn
products.groupBy('prod_brand', 'prod_cat')
        .agg(fn.avg('prod_value')).show()
```

```
+----------+--------+---------------+
|prod_brand|prod_cat|avg(prod_value)|
+----------+--------+---------------+
| microsoft|   mouse|          49.99|
|  logitech|keyboard|          49.99|
| microsoft|keyboard|          59.99|
|  logitech|   mouse|          29.99|
+----------+--------+---------------+
```

EXAMPLES

```
from pyspark.sql import functions as fn
products.groupBy('prod_brand').agg(
    fn.round(fn.avg('prod_value'), 1).alias('average'),
    fn.ceil(fn.sum('prod_value')).alias('sum'),
    fn.min('prod_value').alias('min')
).show()
```

```
+----------+-------+---+-----+
|prod_brand|average|sum|  min|
+----------+-------+---+-----+
|  logitech|   40.0| 80|29.99|
| microsoft|   53.3|160|39.99|
+----------+-------+---+-----+
```

EXAMPLES

```
# Using SQL query
products.createOrReplaceTempView("products")
query = """
    SELECT
        prod_brand,
        round(avg(prod_value), 1) AS average,
        min(prod_value) AS min
    FROM products
    GROUP BY prod_brand
"""
spark.sql(query).show()
```

```
+----------+-------+-----+
|prod_brand|average|  min|
+----------+-------+-----+
|  logitech|   40.0|29.99|
| microsoft|   53.3|39.99|
+----------+-------+-----+
```

WINDOW FUNCTIONS

- A very, very **powerful** feature
- They allow to solve very complex problems
- They exist in some relational databases
- There's a very good article about this feature here

WINDOW FUNCTIONS

- It's similar to aggregations, but the number of rows doesn't change
- Instead, new columns are created, and the aggregated values are duplicated for values of the same "group"
- There are "traditional" aggregations, such as min, max, avg, sum
- and "special" types, such as "lag", "lead", "rank"
- Examples are worth more than 1000 words :)

NUMERIC WINDOW FUNCTIONS

```
from pyspark.sql import Window
from pyspark.sql import functions as fn

# First, we create the Window definition
window = Window.partitionBy('prod_brand')
# Then, we can use "over" to aggregate on this window
avg = fn.avg('prod_value').over(window)
# Finally, we can use this as usualusual
products.withColumn('avg_brand_value', fn.round(avg, 2)).show(
```

```
+-------+--------+----------+----------+---------------+
|prod_id|prod_cat|prod_brand|prod_value|avg_brand_value|
+-------+--------+----------+----------+---------------+
|      4|keyboard|  logitech|     49.99|          39.99|
|      5|   mouse|  logitech|     29.99|          39.99|
|      1|   mouse| microsoft|     39.99|          53.32|
|      2|   mouse| microsoft|     59.99|          53.32|
|      3|keyboard| microsoft|     59.99|          53.32|
+-------+--------+----------+----------+---------------+
```

NUMERIC WINDOW FUNCTIONS

```
from pyspark.sql import Window
from pyspark.sql import functions as fn

# The window can be defined on multiple columns
window = Window.partitionBy('prod_brand', 'prod_cat')

avg = fn.avg('prod_value').over(window)
products.withColumn('avg_value', fn.round(avg, 2)).show()
```

```
+-------+--------+----------+----------+---------+
|prod_id|prod_cat|prod_brand|prod_value|avg_value|
+-------+--------+----------+----------+---------+
|      1|   mouse| microsoft|     39.99|    49.99|
|      2|   mouse| microsoft|     59.99|    49.99|
|      4|keyboard|  logitech|     49.99|    49.99|
|      3|keyboard| microsoft|     59.99|    59.99|
|      5|   mouse|  logitech|     29.99|    29.99|
+-------+--------+----------+----------+---------+
```

NUMERIC WINDOW FUNCTIONS

```
from pyspark.sql import Window
from pyspark.sql import functions as fn

# Multiple windows can be defined
window1 = Window.partitionBy('prod_brand')
window2 = Window.partitionBy('prod_cat')

avg_brand = fn.avg('prod_value').over(window1)
avg_cat = fn.avg('prod_value').over(window2)

products.withColumn('avg_by_brand', fn.round(avg_brand, 2))\
        .withColumn('avg_by_cat', fn.round(avg_cat, 2))\
        .show()
```

```
+-------+--------+----------+----------+------------+----------+
|prod_id|prod_cat|prod_brand|prod_value|avg_by_brand|avg_by_cat|
+-------+--------+----------+----------+------------+----------+
|      4|keyboard|  logitech|     49.99|       39.99|     54.99|
|      3|keyboard| microsoft|     59.99|       53.32|     54.99|
|      5|   mouse|  logitech|     29.99|       39.99|     43.32|
|      1|   mouse| microsoft|     39.99|       53.32|     43.32|
|      2|   mouse| microsoft|     59.99|       53.32|     43.32|
+-------+--------+----------+----------+------------+----------+
```

LAG AND LEAD

- `lag` and `lead` are special functions used over an ordered window
- They are used to take the "previous" and "next" value within the window
- Very useful in datasets with a date column

LAG AND LEAD

```
purchases = spark.createDataFrame([
    (date(2017, 11, 1), 'mouse'),
    (date(2017, 11, 2), 'mouse'),
    (date(2017, 11, 4), 'keyboard'),
    (date(2017, 11, 6), 'keyboard'),
    (date(2017, 11, 9), 'keyboard'),
    (date(2017, 11, 12), 'mouse'),
    (date(2017, 11, 18), 'keyboard')
], ['date', 'prod_cat'])

purchases.show()
```

```
+----------+--------+
|      date|prod_cat|
+----------+--------+
|2017-11-01|   mouse|
|2017-11-02|   mouse|
|2017-11-04|keyboard|
|2017-11-06|keyboard|
|2017-11-09|keyboard|
|2017-11-12|   mouse|
|2017-11-18|keyboard|
+----------+--------+
```

LAG AND LEAD

```
window = Window.partitionBy('prod_cat').orderBy('date')
prev_purch = fn.lag('date', 1).over(window)
next_purch = fn.lead('date', 1).over(window)

purchases.withColumn('prev', prev_purch)\
    .withColumn('next', next_purch)\
    .orderBy('prod_cat', 'date')\
    .show()
```

```
+--------+----------+----------+----------+
|prod_cat|      date|      prev|      next|
+--------+----------+----------+----------+
|keyboard|2017-11-04|      null|2017-11-06|
|keyboard|2017-11-06|2017-11-04|2017-11-09|
|keyboard|2017-11-09|2017-11-06|2017-11-18|
|keyboard|2017-11-18|2017-11-09|      null|
|   mouse|2017-11-01|      null|2017-11-02|
|   mouse|2017-11-02|2017-11-01|2017-11-12|
|   mouse|2017-11-12|2017-11-02|      null|
+--------+----------+----------+----------+
```

RANK, DENSERANK AND ROWNUMBER

- Another set of useful "special" functions
- Also used on ordered windows
- They create a rank, or an order of the items within the window

RANK AND ROWNUMBER

```
contestants = spark.createDataFrame([
    ('veterans', 'John', 3000),
    ('veterans', 'Bob', 3200),
    ('veterans', 'Mary', 4000),
    ('young', 'Jane', 4000),
    ('young', 'April', 3100),
    ('young', 'Alice', 3700),
    ('young', 'Micheal', 4000),
], ['category', 'name', 'points'])

contestants.show()
```

```
+--------+-------+------+
|category|   name|points|
+--------+-------+------+
|veterans|   John|  3000|
|veterans|    Bob|  3200|
|veterans|   Mary|  4000|
|   young|   Jane|  4000|
|   young|  April|  3100|
|   young|  Alice|  3700|
|   young|Micheal|  4000|
+--------+-------+------+
```

RANK AND ROWNUMBER

```
window = Window.partitionBy('category').orderBy(contestants.points.desc())

rank = fn.rank().over(window)
dense_rank = fn.dense_rank().over(window)
row_number = fn.row_number().over(window)

contestants\
     .withColumn('rank', rank)\
     .withColumn('dense_rank', dense_rank)\
     .withColumn('row_number', row_number)\
     .orderBy('category', fn.col('points').desc())\
     .show()
```

```
+--------+-------+------+----+----------+----------+
|category|   name|points|rank|dense_rank|row_number|
+--------+-------+------+----+----------+----------+
|veterans|   Mary|  4000|   1|         1|         1|
|veterans|    Bob|  3200|   2|         2|         2|
|veterans|   John|  3000|   3|         3|         3|
|   young|   Jane|  4000|   1|         1|         1|
|   young|Micheal|  4000|   1|         1|         2|
|   young|  Alice|  3700|   3|         2|         3|
|   young|  April|  3100|   4|         3|         4|
+--------+-------+------+----+----------+----------+
```

WRITING DATAFRAMES

- Very similar to Reading
- Output targets are the same (csv, json, parquet, jdbc, etc.)
- Instead of `df.read.{source}`, just use `df.write.{target}`

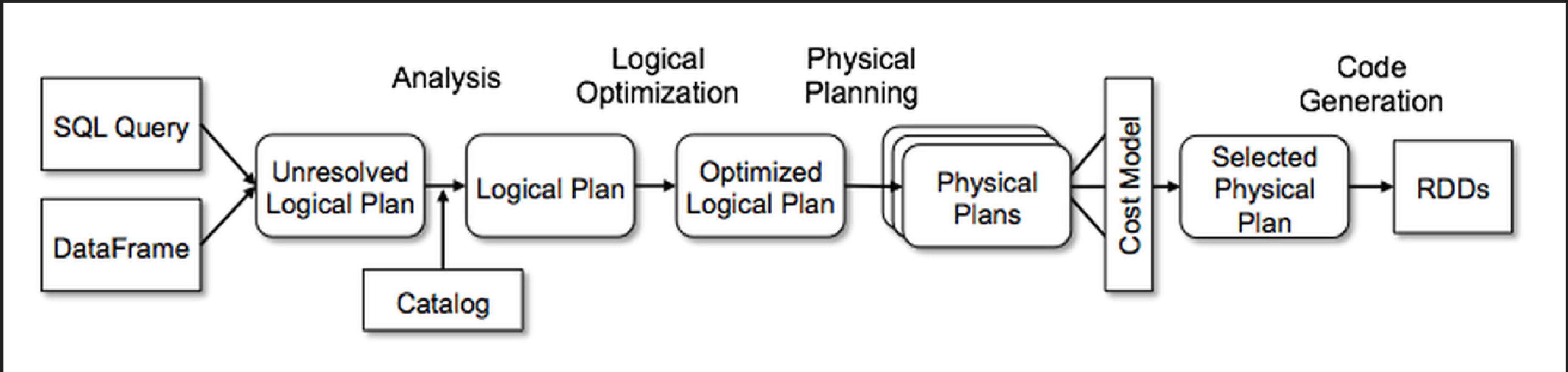
WRITING DATAFRAMES

- Main option: `mode`
- Possible values:
  - "append": Append contents of this DataFrame to existing data.
  - "overwrite": Overwrite existing data.
  - "error": Throw an exception if data already exists.
  - "ignore": Silently ignore this operation if data already exists.

EXAMPLES

```
products.write.csv('/products.csv')
products.write.mode('overwrite').parquet('/file.parquet')
products.write.format('parquet').save('/file.parquet')
```

QUERY PLANNING AND OPTIMIZATION



A good post if you want more details on this.

(: !DNE EHT

ANY QUESTIONS?