Apache Spark Course: Introduction to Spark SQL

Luca Canali
CERN IT, Data Analytics and Spark Service

Spark SQL and Spark DataFrames

- Two APIs for the same functionality
  - DataFrame declarative API
  - Spark SQL

```
$ bin/pyspark
# Spark DataFrame API
>>> df = spark.range(10)
>>> df.count()
10
```

```
# Spark SQL
>>> df = spark.sql("select id from range(10)")
>>> df.count()
10
```

```
>>> df.show()

+---+
| id|
+---+
|  0|
|  1|
|  2|
|  3|
|  4|
|  5|
|  6|
|  7|
|  8|
|  9|
+---+
```

Why SQL?

- If you already have a SQL application
- If you are already familiar with SQL
- Spark implements ANSI SQL
- Certain computations can be more naturally expressed in SQL
  - At the end it’s a matter of taste

Why DataFrame API

- Modern API, similar to Python Pandas
- If you are porting an application written with Pandas and/or DataFrames
- Sometimes the best choice
  - Example: DF.count()
  - When building queries dynamically at runtime:

```
if (a > 0):
   df2 = df.filter(col1 > 1)
else:
   df2 = df.filter(col2 > 1)
```

DataFrames -> Tables and Views

- SQL operates on tables and views
  - DataFrames can be mapped into views using createOrReplaceTempView

```
>>> df = spark.createDataFrame([(1, "event1"), (2,"event2"), (3, "event3")],
("id","name"))
>>> df.createOrReplaceTempView("myEvents")

>>> spark.sql("select count(*) from myEvents").show()
+--------+
|count(1)|
+--------+
|       3|
+--------+
```

Tables, Views -> DataFrames

- We can map tables and views into DataFrames
  - Key point: no data is being copied

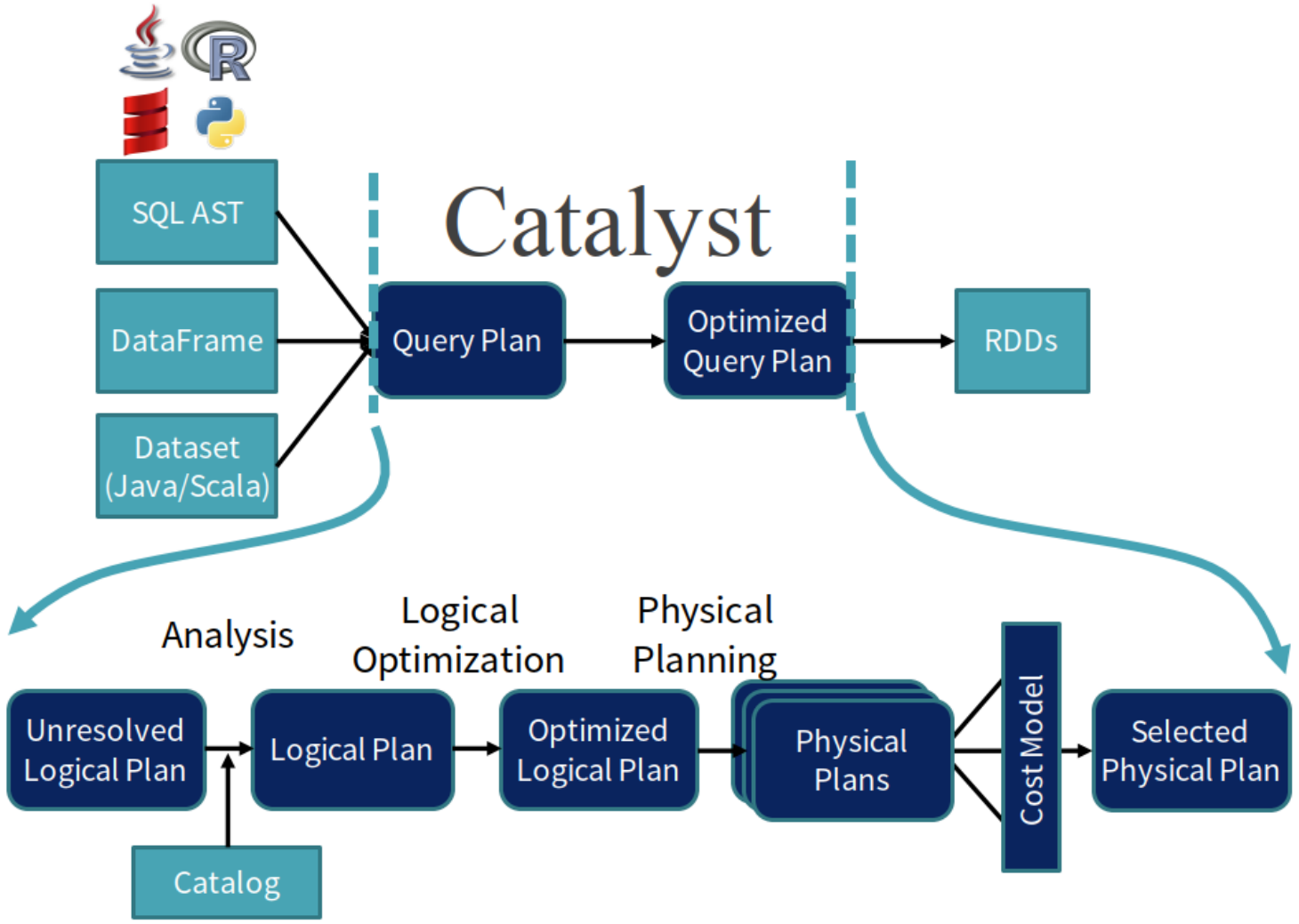
```
>>> df = spark.table("myEvents")
>>> df
DataFrame[id: bigint, name: string]
>>> df.show()
+---+------+
| id|  name|
+---+------+
|  1|event1|
|  2|event2|
|  3|event3|
+---+------+
```

Underlying Concepts Behind DataFrame API and SQL

- Applies to structured data -> that is data with a schema
- Main categories of operations
  - Selection
  - Projection
  - Join
  - Union
- You describe the computation
  - No need to tell the system how to perform it
  - Spark will find an (optimized) way to execute it

Spark Execution Plans

Spark (Catalyst) generates the optimized physical plan



From: https://databricks.com/session/deep-dive-into-catalyst-apache-spark-2-0s-optimizer

Example of an Execution Plan

- This is how Spark runs the query
- DF.explain or sql("explain select …")

```
spark.sql("select movieId, title from movies where movieId=1").explain()

== Physical Plan ==
*(1) Filter (isnotnull(movieId#3580) AND (cast(movieId#3580 as int) = 1))
  +- FileScan csv [movieId#3580,title#3581] .. Format: CSV, …
```

Spark Uses Lazy Execution

- Spark SQL and DataFrame API
  - API to describe computation
- You need an action to trigger execution
  - Example of actions: DF.count(), DF.show(), DF.collect()

Spark SQL – Basic Examples

```
df = sql("select * from values (1, 'event1'), (2,'event2'), (3, 'event3') as (id, name)")
# this registers a temporary view called t1
df.createOrReplaceTempView("t1")
```

```
sql("select id, name, id % 2 even_or_odd from t1").show()

sql("select id, name, id % 2 even_or_odd from t1 where id < 3").show()

sql("""select id % 2 as even_or_odd, count(*) N_samples, sum(id) Sum_Ids
       from t1
       group by id % 2""").show()
```

Joins

Joins are key data operations

- optimizer and execution engine can make a difference

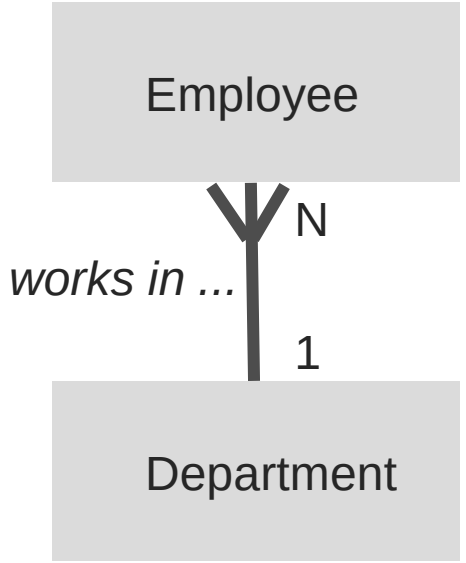
Refresher: parent – child relationship (department - employee)

- Inner join

```
select employees.id, employees.name, departments.name
from employees join departments
on employees.dep_id = departments.id
```

- Outer join

```
select departments.id, departments.name, employees.name
from departments left outer join employees
on employees.dep_id = departments.id
```



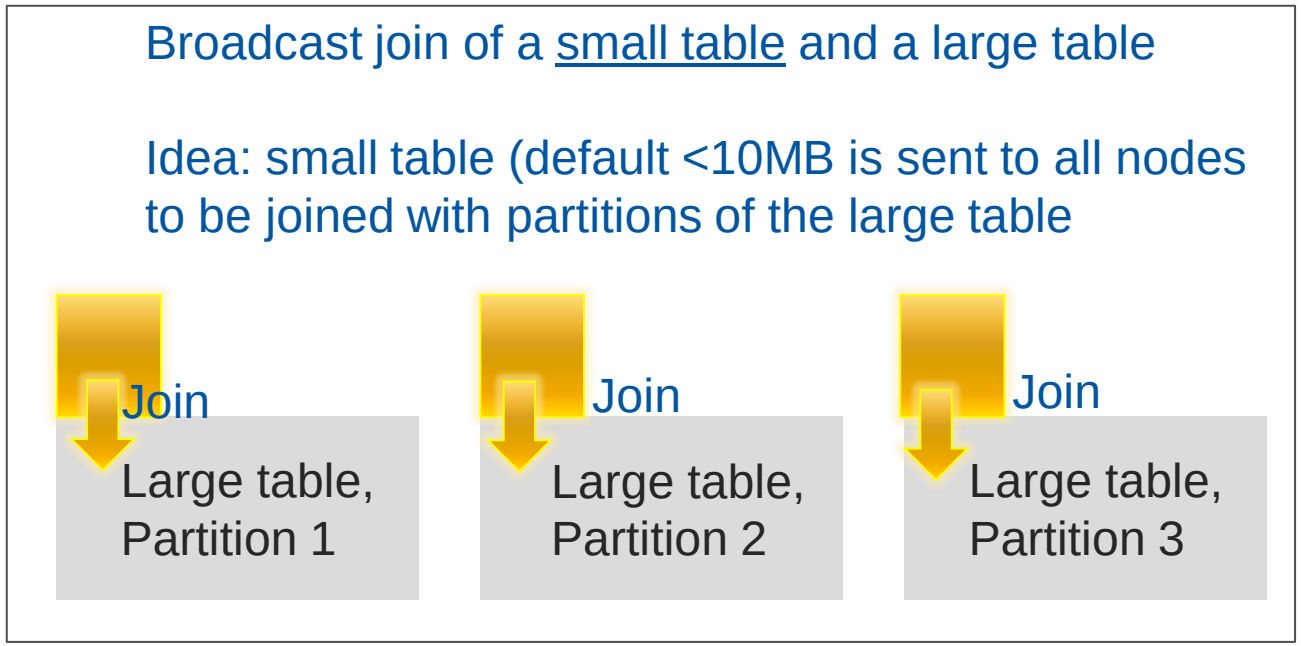
Joins Execution in Spark

Spark optimizer (catalyst): finds a suitable execution plan: join order and type

Spark engine: scalable and fault-resilient parallel engine to run queries

Main join methods in Spark:

- Sort merge joins
- Shuffle hash join
- Broadcast join



Spark Reading From Databases

- An example, reading from Oracle
  - Read from a table or feed a query to run

```
df = spark.read.format("jdbc")
          .option("url", "jdbc:oracle:thin:@dbserver:port/service_name")
          .option("driver", "oracle.jdbc.driver.OracleDriver")
          .option("dbtable", "(select col1 from MYSCHEMA.MYTABLE)")
          .option("user", "MYORAUSER")
          .option("password", "XXX")
          .option("fetchsize",10000).load()

# test
df.printSchema()
df.show(5)
```

SQL Can be Complex

- Fizzbuzz in SQL, for demo purpose
  - using the SQL case statement
  - https://en.wikipedia.org/wiki/Fizzbuzz

```
sql("""
select case
    when id % 15 = 0 then 'FizzBuzz'
    when id % 3 = 0 then 'Fizz'
    when id % 5 = 0 then 'Buzz'
    else cast(id as string)
    end as FizzBuzz
from range(1,20)
order by id""").show()
```

```
+--------+
|FizzBuzz|
+--------+
|       1|
|       2|
|    Fizz|
|       4|
|    Buzz|
|    Fizz|
|       7|
|       8|
|    Fizz|
|    Buzz|
|      11|
|    Fizz|
|      13|
|      14|
|FizzBuzz|
|      16|
|      17|
|    Fizz|
|      19|
+--------+
```

SQL Window Functions

- SQL standard, used in complex analytics
  - https://en.wikipedia.org/wiki/SQL_window_function

```
spark.sql("""
select id, name, dep_id,
       max(id) over (partition by dep_id) as greatest_id_same_department,
       lag(name) over (order by id) as previous_employee_name
from employees
order by id
""").show()
```

Spark SQL and Arrays

- When data is not more complex
  - Example with numeric arrays

```
# Prepare test data with arrays
schema = "id INT, val ARRAY<INT>"

t_list = [1,[35, 36, 32, 30, 40, 42, 38]], [2,[31, 32, 34, 55, 56]]
spark.createDataFrame(t_list, schema).createOrReplaceTempView("data1")
```

```
spark.sql("select * from data1").show(10,False)
+---+---------------------------+
|id |temp_celsius               |
+---+---------------------------+
|1  |[35, 36, 32, 30, 40, 42, 38]|
|2  |[31, 32, 34, 55, 56]       |
+---+---------------------------+
```

Processing Arrays in SQL is Slow

Overcome the limitation using Spark Higher Order Functions

```
# Prepare test data with arrays
schema = "id INT, val ARRAY<INT>"

t_list = [1,[35, 36, 32, 30, 40, 42, 38]], [2,[31, 32, 34, 55, 56]]
spark.createDataFrame(t_list, schema).createOrReplaceTempView("data1")
```

```
# Array processing with Spark higher order functions in SQL
# Multiply by 2 the values in the array

spark.sql("""
SELECT id, val,
transform(val, t -> t * 2 as twice_the_value)
FROM data1""").show()
```

Spark Higher Order Functions in SQL

- Push filters into arrays with higher order functions

```
# Array processing with Spark higher order functions in SQL
# Filter values > 38 from an array of temperatures

spark.sql("""
SELECT id, val, filter(val, t -> t > 38) as high
FROM temp_data""").show()

+---+-----------------------------+--------+
|id |temp_celsius                 |high    |
+---+-----------------------------+--------+
|1  |[35, 36, 32, 30, 40, 42, 38]|[40, 42]|
|2  |[31, 32, 34, 55, 56]         |[55, 56]|
+---+-----------------------------+--------+
```

Schemas with Arrays and Structs

- Example inspired from physics datasets

```
schema = "event LONG, HLT struct<flag1:boolean, flag2:boolean>, muons
ARRAY<STRUCT<pt:FLOAT, eta:FLOAT, mass:FLOAT>>"

df.printSchema()

 |-- event: long (nullable = true)
 |-- HLT: struct (nullable = true)
 |    |-- flag1: boolean (nullable = true)
 |    |-- flag2: boolean (nullable = true)
 |-- muons: array (nullable = true)
 |    |-- element: struct (containsNull = true)
 |    |    |-- pt: float (nullable = true)
 |    |    |-- eta: float (nullable = true)
 |    |    |-- phi: float (nullable = true)
 |    |    |-- mass: float (nullable = true)
```

Python User Defined Functions (UDF)

- Write your Python code and apply it to data

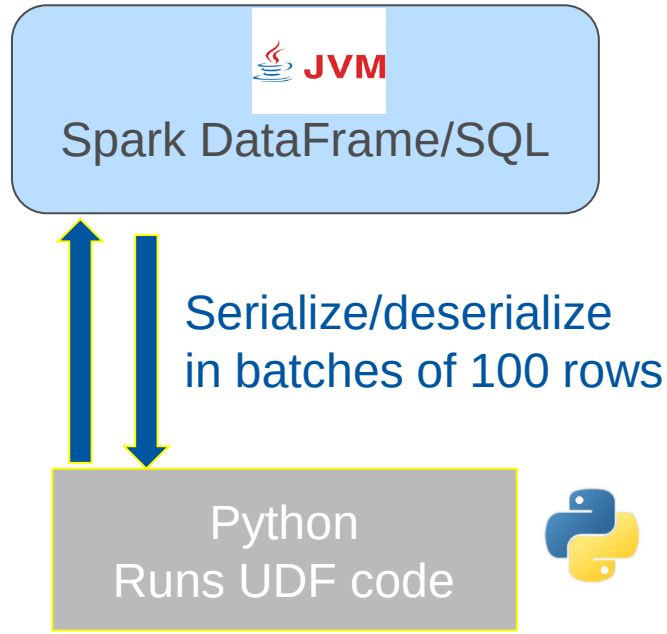
```
from pyspark.sql.functions import udf
import time

@udf("int")
def slowf(s):
  time.sleep(1)
  return 2*s

spark.udf.register("slowf", slowf)

sql("select slowf(1)").show()

sql("select slowf(id) from range(10)").show()
```



Pandas UDF – Improved Performance

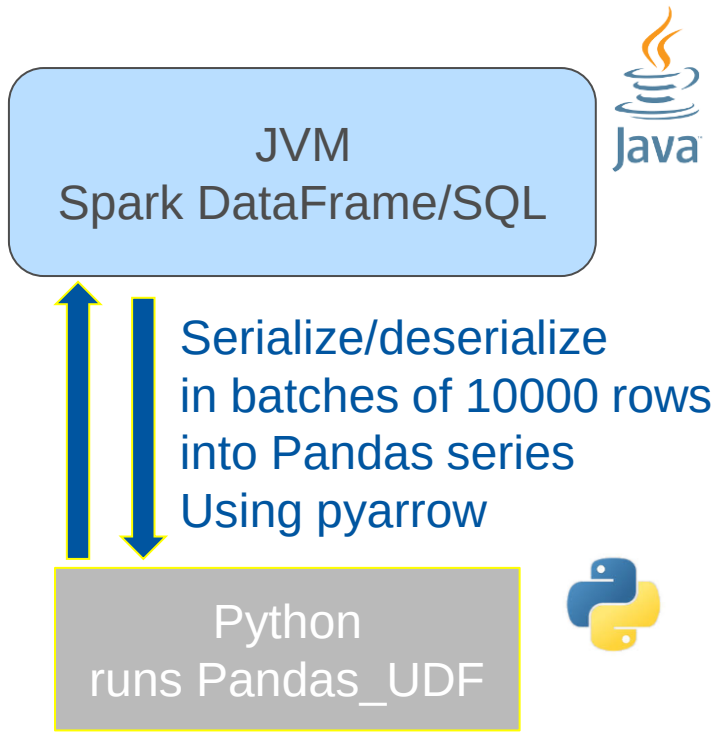
- Faster serialization (data movement Python - JVM)
- Send Pandas series to Python UDF for “bulk processing”

```
import time
from pyspark.sql.functions import pandas_udf

@pandas_udf("long")
def multiply_func(a, b):
return a * b

spark.udf.register("multiply_func", multiply_func)

sql("select multiply_func(1,1)").show()
sql("select multiply_func(id,2) from range(10)").show()
sql("select multiply_func(id,2) from range(10000)").collect()
```



Note: in the examples notebook you can see this in action.

Pandas UDF and Type Annotations

```
import pandas as pd

@pandas_udf("long")
def multiply_func(a: pd.Series, b: pd.Series) -> pd.Series:
    return a * b
```

Recommended to use Python type annotations.

```
@pandas_udf("long")
def calculate(iterator: Iterator[pd.Series]) -> Iterator[pd.Series]:
    # Do some expensive initialization with a state
    state = very_expensive_initialization()
    for x in iterator:
        # Use that state for whole iterator.
        yield calculate_with_state(x, state)

df.select(calculate("value")).show()
```

Pandas UDF with iterators.
Example: ML serving UDF

Key Learning Points

- DataFrame API and Spark SQL
  - Are two interfaces for the same execution engine
  - Powerful abstractions for processing structured data at scale
  - Extensible with user defined functions