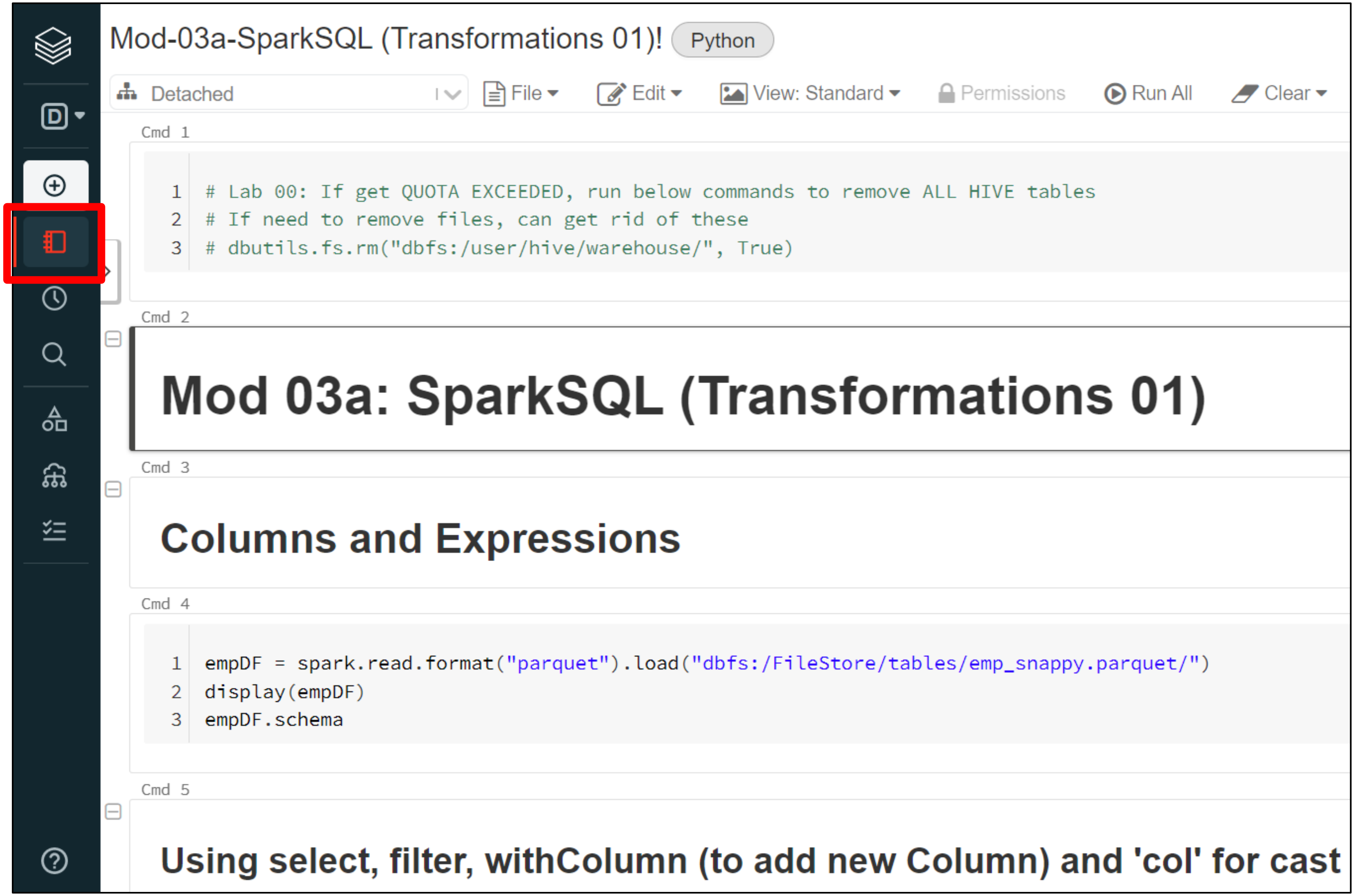
Module 03 – Spark SQL – Transformations

Table of Contents

Don't forget to start WebEx recording
Do Spark Jeopardy review

Go to: community.cloud.databricks.com and Logon
In Left-pane, Click on 'Clusters' or 'Compute' and Terminate old Cluster
Then click 'Create Cluster' button to create New one

Before we Begin: Open Notebook Mod-03a



Module 03 – Spark SQL

After completing this module, you'll be able to work with Spark SQL including:

Transformations 01 Notebook

- Columns and Expressions
- Operators and Methods
- Transformations and Actions

Transformation 02

- Lab 01: Dates and Timestamps
- Lab 02: Aggregations
- Lab 03: Widgets
- Non-Aggregate functions
- Other built-in functions
- Lab 04: User Defined Functions

Spark SQL – Columns and Expressions

Columns

- Column is an object we will be transforming in a DataFrame/Table using an expression

- In a DataFrame, you refer to a Column using various syntax including

```
df["columnName"]

df.columnName

col("columnName")

col("columnName.field")
```

- You can manufacture new values from Column values via Operators/Methods

```
col("x") * col("b")

col("x").asc()

col("x").cast("float") * 100
```

Column Operators and Methods

Operator/Method	Description
&, |	Boolean AND, OR in Python
***, + , <, >=**	Math and comparison operators
==, !=	Equality and inequality tests in Python
alias, as	Gives the column an alias, **as** only in Scala
cast, astype	Casts the column to different data type, **astype** only in Python
isNull, isNotNull, isNan	Is null, is not null, is NaN (Not A Number)
isin	Boolean evaluated to true if value contained by values of the arguments
asc, desc	Returns a sort expression based on ascending/descending order of the column

DataFrame Operations

DataFrames support two types of Operations:

1. **Actions**, which computes a result based on an DataFrame
   - Actions return a result either to the <u>console</u> or a <u>written file</u>
   - For example, in earlier labs, we used the display**()** function which is an Action to display the result set to the console
2. **Transformations**, which create a new DataFrame from an existing DataFrame
   - For example, **isin** is a Transformation that passes each column value through a Boolean evaluation. If true, column values are returned
   - All **Transformations** in Spark are lazy, in that they do not compute their results right away. Instead, they just remember the Transformations applied to some base dataset (e.g. a file). The Transformations are only computed when an **Action** requires a result to be returned to the Driver program

DataFrame Transformations

Transformation	Description
select	Returns a new DataFrame by computing given expression
selectExpr	Variant of select() that accepts SQL expression and returns new DataFrame
drop	Returns a new DataFrame with a column dropped
withColumnRenamed	Returns a new DataFrame with a column renamed
withColumn	Returns a new DataFrame by adding a column or replacing the existing column that has the same name
filter, where	Filters rows using the given condition
sort, orderBy	Returns a new DataFrame sorted by the given expressions
dropDuplicates, distinct	Returns a new DataFrame with duplicate rows removed
limit	Returns a new DataFrame by taking the first n rows
groupBy	Groups DataFrame using specified columns, so we can run aggregation on them

DataFrame Actions

Operator/Method	Description
display	Displays the DataFrame in tabular form via http format
show	Displays the top n rows of DataFrame in a tabular form
count	Returns the number of rows in the DataFrame
describe, summary	Computes basic statistics for numeric and string columns
first, head	Returns the the first row
collect	Returns an array that contains all rows in this DataFrame
take	Returns an array of the first n rows in the DataFrame

Actions start the Execution of a Job

Spark SQL – Date and Timestamps (Start of Transformations 02)

Copy

What is Unix Time?

- Unix time is a system for describing a point in time. It is the number of micro-seconds that have elapsed since the Unix epoch, excluding leap seconds. The Unix epoch is 00:00:00 UTC on 1 January 1970

- There are several ways to convert Unix time to Spark Timestamp format

```
df2 = df1.withColumn("ts", (col("unixtime") / 1e6).cast("timestamp"))
display(df2)
```

▸ (1) Spark Jobs

▸ df2: pyspark.sql.dataframe.DataFrame = [user_id: string, unixtime: long ... 1 more field]

	user_id	unixtime	ts
1	UA000000107379500	1593878946592107	2020-07-04T16:09:06.592+0000
2	UA000000107359357	1593877011756535	2020-07-04T15:36:51.756+0000
3	UA000000107375547	1593878815459100	2020-07-04T16:06:55.459+0000

https://www.unixtimestamp.com/

Built-in functions

```
df = spark.createDataFrame([('1997-02-28 10:30:00',)], ['col1'])
display(df.select(to_date(df.col1).alias('dt')))
```

	dt
1	1997-02-28

Function	Description
date_format	Converts a date/timestamp/string to a value of string in the format specified by the date format given by the second argument.
add_months	Returns the date that is numMonths after startDate
to_date	Converts Column into pyspark.sql.types.DateType via optionally specified format
dayofweek	Extracts the day of the month as an integer from a given date/timestamp/string
date_add	Returns the date that is N days after start
from_unixtime	Converts the number of seconds from unix epoch (1970-01-01 00:00:00 UTC) to a string representing timestamp of that moment in current system time zone in given format
year | month | minute | second	Extracts the time frame as an integer from a given date/timestamp/string.
unix_timestamp	Converts time string with given pattern to Unix timestamp (in seconds)

Date/Time Patterns

Symbol	Meaning	Presentation	Examples
G	era	text	AD; Anno Domini
y	year	year	2020; 20
D	day-of-year	number(3)	189
M/L	month-of-year	month	7; 07; Jul; July
d	day-of-month	number(3)	28
Q/q	quarter-of-year	number/text	3; 03; Q3; 3rd quarter
E	day-of-week	text	Tue; Tuesday
F	week-of-month	number(1)	3
a	am-pm-of-day	am-pm	PM
h	clock-hour-of-am-pm (1-12)	number(2)	12

```
SELECT date_format(current_date, "MMM")
AS CurrentMonth
```

(1) Spark Jobs

	CurrentMonth
1	Oct

Spark SQL – Aggregations and Joins

Aggregate functions

Function	Description
agg	Compute aggregates by specifying a series of aggregate columns
avg	Compute the average value for each numeric columns for each group
count	Count the number of rows for each group
max	Compute the max value for each numeric columns for each group
mean	Compute the average value for each numeric columns for each group
min	Compute the min value for each numeric column for each group
pivot	Pivots a column of the current DataFrame and performs the specified aggregation
sum	Compute the sum for each numeric columns for each group

'groupBy' and 'count'

<u>Query</u>: Count how many rows per Dept where Dept > 400. Sort in Descending order by Dept

```
1  empDF.select("last_name", "dept", "salary").filter("dept > 400").groupBy("dept").count().orderBy("dept", ascending=False).show()
```

▸ (2) Spark Jobs

```
+----+-----+
|dept|count|
+----+-----+
| 501|    4|
| 403|    6|
| 402|    2|
| 401|    7|
+----+-----+
```

groupBy() with agg()

Query: For each 'Dept', do multiple aggregate.
Want Count # of row per 'Dept', Max 'salary' per 'Dept'

```
empDF.select("emp", "l_name", "dept", "salary").show()
```

```
#// DataFrame query
empDF.select("dept", "salary").groupBy("dept").agg({"*": "count", "salary": "max"}).show()

#// SQL equivalent
spark.sql("select dept, count(*) as ct_dept, max(salary) as max_sal
                    from emp_tbl group by dept").show()
```

```
+----+---------+----+--------+
| emp|   l_name|dept|  salary|
+----+---------+----+--------+
|1018| Ratzlaff| 501| 54000.0|
|1016|   Rogers| 302| 56500.0|
|1014|    Crane| 402| 24500.0|
|1004|  Johnson| 401| 36300.0|
|1002|    Brown| 401| 43100.0|
|1021|Morrissey| 201| 38750.0|
|1019|    Kubic| 301| 57700.0|
|1017|   Runyon| 501| 66000.0|
|1015|   Wilson| 501| 53625.0|
| 801|  Trainer| 100|100000.0|
|1003|   Trader| 401| 37850.0|
|1022|  Machado| 401| 32300.0|
|1001|   Hoover| 401| 25525.0|
|1020|  Charles| 403| 39500.0|
|1012|  Hopkins| 403| 37900.0|
|1010|   Rogers| 401| 46000.0|
|1008| Kanieski| 301| 29250.0|
|1006|    Stein| 301| 29450.0|
|1025|    Short| 201| 34700.0|
|1023|   Rabbit| 501| 26500.0|
+----+---------+----+--------+
```

```
+----+-----------+-----------+
|dept|count(dept)|max(salary)|
+----+-----------+-----------+
| 100|          1|   100000.0|
| 301|          3|    57700.0|
| 501|          4|    66000.0|
| 302|          1|    56500.0|
| 401|          7|    46000.0|
| 201|          2|    38750.0|
| 402|          2|    52500.0|
| 403|          6|    49700.0|
+----+-----------+-----------+
```

Join functions

Function	Description
join	Joins with another DataFrame. Supports INNER, LEFT OUTER, RIGHT OUTER, LEFTSEMI, LEFTANTI CROSS

empDF

emp_id	name	mgr	year_joined	emp_dept_id	gender	salary
1	Smith	-1	2018	10	M	3000
2	Rose	1	2010	20	M	4000
3	Williams	1	2010	10	M	1000
4	Jones	2	2005	10	F	2000
5	Brown	2	2010	40		-1
6	Brown	2	2010	50		-1

deptDF

dept_name	dept_id
Finance	10
Marketing	20
Sales	30
IT	40

```
display(empDF.join(deptDF,empDF.emp_dept_id == deptDF.dept_id, "inner"))
```

emp_id	name	mgr	year_joined	emp_dept_id	gender	salary	dept_name	dept_id
1	Smith	-1	2018	10	M	3000	Finance	10
3	Williams	1	2010	10	M	1000	Finance	10

Inner join() using DataFrames

```
empDF.join(deptDF, "dept").limit(3).show()

empDF.join(deptDF, "dept").select("last_name", "dept", "dept_name").limit(4).show()
```

▸ (2) Spark Jobs

Join column

```
+----+----+----+------+---------+----------+----------+----------+--------+----------------+---------+----+
|dept| emp| mgr|   job|last_name|first_name|      hire|     birth|  salary|       dept_name|   budget| mgr|
+----+----+----+------+---------+----------+----------+----------+--------+----------------+---------+----+
| 501|1018|1017|512101| Ratzlaff|     Larry|1978-07-15|1954-05-31|54000.00| marketing sales|308000.00|1017|
| 302|1016| 801|321100|   Rogers|      Nora|1978-03-01|1959-09-04|56500.00|product planning|226000.00|1016|
| 402|1014|1011|422101|    Crane|    Robert|1978-01-15|1960-07-04|24500.00|software support|308000.00|1011|
+----+----+----+------+---------+----------+----------+----------+--------+----------------+---------+----+

+---------+----+----------------+
|last_name|dept|       dept_name|
+---------+----+----------------+
| Ratzlaff| 501| marketing sales|
|   Rogers| 302|product planning|
|    Crane| 402|software support|
|  Johnson| 401|customer support|
+---------+----+----------------+
```

Left join() using DataFrames

Put invalid Dept 999 in empDF, then do Left Outer Join

```python
# Lab 12b: Change dept 100 to 999 for next Lab (Left JOIN,)
# Dept 999 exists in empDF2, but not deptDF now
from pyspark.sql.functions import when

empDF2 = empDF.withColumn("dept", when(empDF["dept"] == 100, 999).otherwise(empDF["dept"]))
empDF3 = empDF2.distinct().orderBy(["dept"], ascending=False)
```

```python
# Lab 12c: Left-outer Join
empDF3.join(deptDF, "dept", "left_outer").orderBy(["dept"], ascending=False).show()

# Lab 12d: Join on 2 columns:
empDF3.join(deptDF, (empDF2.dept == deptDF.dept) & (empDF2.mgr == deptDF.mgr)).limit(3).show()
```

dept	emp	mgr	job	last_name	first_name	hire	birth	salary	dept_name	budget	mgr
999	801	[illegible]	111100	Trainer	I.B.	1973-03-01	1945-08-11	100000.00	null	null	null
501	1017	801	511100	Runyon	Irene	1978-05-01	1951-11-10	66000.00	marketing sales	308000.00	1017
501	1015	1017	512101	Wilson	Edward	1978-03-01	1957-03-04	53625.00	marketing sales	308000.00	1017
501	1023	1017	512101	Rabbit	Peter	1979-03-01	1962-10-29	26500.00	marketing sales	308000.00	1017

Non-match

Matching

emp	mgr	dept	job	last_name	first_name	hire	birth	salary	dept	dept_name	budget	mgr
1008	1019	301	312102	Kanieski	Carol	1977-02-01	1958-05-17	29250.00	301	research and deve...	465600.00	1019
1020	1005	403	432101	Charles	John	1978-10-01	1949-06-21	39500.00	403	education	932000.00	1005
1015	1017	501	512101	Wilson	Edward	1978-03-01	1957-03-04	53625.00	501	marketing sales	308000.00	1017

Spark SQL – Miscellaneous Functions
(No labs on these)

Non-Aggregate Built-in functions

Function	Description
col	Returns a Column based on the given column name
lit	Creates a Column of literal value
IsNull | isNotNull	Return true if the column is null | Returns true if column is Not Null
concat	Concatenates multiple input columns together into a single column
rand	Generate a random column with independent and identically distributed (i.i.d.) samples uniformly distributed in [0.0, 1.0)

```
from pyspark.sql.functions import *
df = spark.createDataFrame([([123],), ([666],), ([456],), ([789],)], ['id'])
display(df.select('id').orderBy(rand()))
```

1st

id
[123]
[789]
[456]
[666]

2nd

id
[789]
[123]
[456]
[666]

3rd

id
[666]
[456]
[123]
[789]

String Built-in functions

Function	Description
translate	Translate any character in the src by a character in replaceString
regexp_replace	Replace all substrings of the specified string value that match regexp with rep
regexp_extract	Extract specific group matched by Java regex, from specified string column
ltrim	Removes the leading space characters from the specified string column
lower	Converts a string column to lowercase
split	Splits str around matches of the given pattern

Na Built-in functions

Function	Description
na.drop	Returns a new DataFrame omitting rows with any, all, or a specified number of null values, considering an optional subset of columns
na.fill (value)	Replace NULL values with the specified value for an optional subset of columns
replace (str, search [,replace])	Returns a new DataFrame replacing a value with another value, considering an optional subset of columns (Replaces all occurrences of 'search' with 'replace'

```
>>> df4.na.fill({'age': 50, 'name': 'unknown'}).show()
+---+------+-------+
|age|height|   name|
+---+------+-------+
| 10|    80|  Alice|
|  5|  null|    Bob|
| 50|  null|    Tom|
| 50|  null|unknown|
+---+------+-------+
```

Fill NULL for 'age' with 50
Fill NULL for 'name' with 'unknown'

```
%sql
SELECT replace('ABCabc', 'abc', 'DEF');
```

Replace 'abc' with 'DEF'

	replace(ABCabc, abc, DEF)
1	ABCDEF

Spark SQL – Widgets

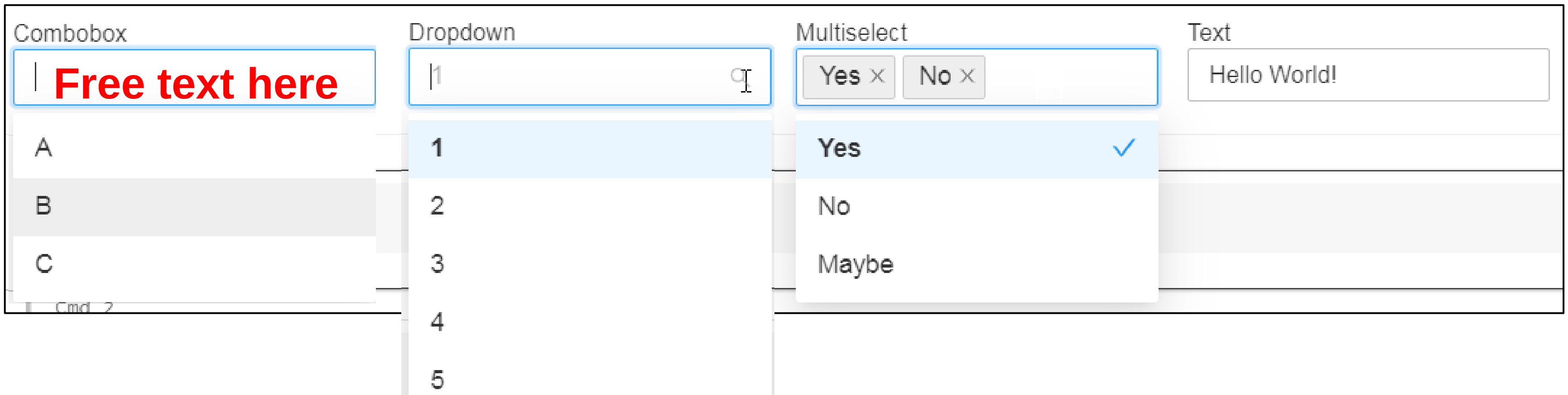
Widgets: Parameterize your Notebook

There are 4 types of Widgets

- text: Input a value in a text box
- dropdown: Select a value from a list of provided values
- combobox: Combination of text and dropdown. Select a value from a provided list or input one in the text box.
- multiselect: Select one or more values from a list of provided values

```
dbutils.widgets.help()
```

Widgets: Parameterize your Notebook




```
dbutils.widgets.text("Text", "Hello World!")
dbutils.widgets.dropdown("Dropdown", "1", [str(x) for x in range(1,10)])
dbutils.widgets.combobox("Combobox", "A", ["A", "B", "C"])
dbutils.widgets.multiselect("Multiselect", "Yes", ["Yes", "No", "Maybe"])
```

```
dbutils.widgets.help()
```

Widgets: Parameterize your Notebook

last_name

Brown

Cmd 28

```
%sql
-- Notice in upper left-hand corner, parameter appears
CREATE widget TEXT last_name DEFAULT "Brown"
```

```
SELECT * FROM emp_tbl
WHERE last_name = getArgument("last_name")
```

(1) Spark Jobs

	emp	mgr	dept	job	last_name	first_name
1	1002	1003	401	413201	Brown	Alan
2	1024	1005	403	432101	Brown	Allen

last_name

Johnson

```
-- Run 'as is', then change Parameter in upper left-hand corner = Johnson
SELECT * FROM emp_tbl
WHERE last_name = getArgument("last_name")
```

(1) Spark Jobs

	emp	mgr	dept	job	last_name	first_name
1	1004	1003	401	412101	Johnson	Darlene

```
dbutils.widgets.help()
```

Spark SQL – User Defined Functions (UDFs)

Copy

UDF Performance compared to built-in Functions

- The Spark DataFrame functions are natively a JVM structure and standard access methods are implemented by simple calls to Java API. UDF from the other hand are implemented in Python and require moving data back and forth

- In other words, Spark SQL functions operate directly on JVM and typically are well integrated with both Catalyst and Tungsten. It means these can be optimized in the execution plan and most of the time can benefit from Whole Stage Code Gen and other Tungsten optimizations. Moreover, these can operate on data in its "native" representation

- In order of Preference when Performance is important:

1. Higher-Order functions
2. Pandas UDFs
3. Python UDFs

Type	Time(s)
Python UDF	43.0632779598
Python Vectorized UDF	13.9144539833
Scala UDF	0.257154205

Using Scala UDFs in PySpark. TL;DR | by WB Advanced Analytics | wbaa | Medium

UDFs for DataFrames and Tables

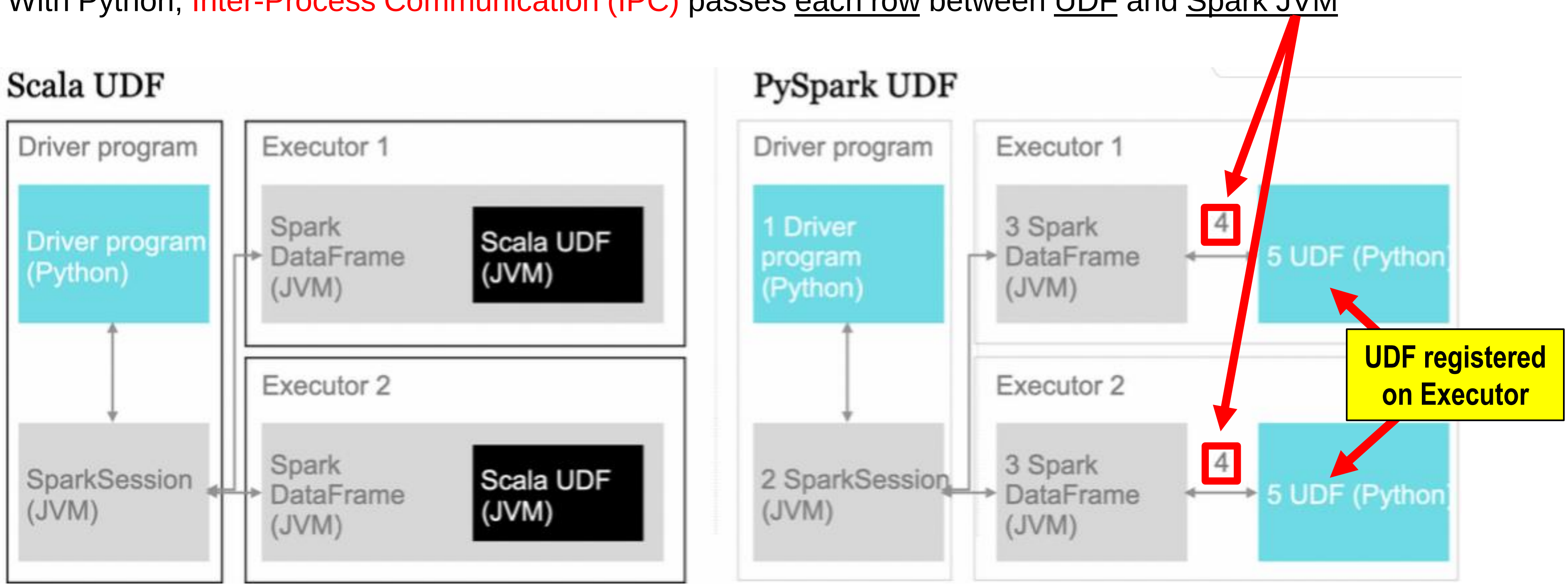
- UDF are when you wish to write custom Functions
- UDFs are a 'black box' in the sense that the Catalyst optimizer cannot tune the query. Hence, performance goes down when compared to using a 'built-in' function
- There are Performance differences between Scala and Python UDFs
  - Python incurs additional cost of Python interpreter
- UDF's are Serialized and sent to the Executors for processing

This registers the UDF on the Executors

```
%python
# Our input/output is a string
@udf("string")
def decoratorUDF(email: str) -> str:
  return email[0]
```

Python UDFs incur more cost compared to Scala UDFs

With Python, Inter-Process Communication (IPC) passes each row between UDF and Spark JVM



https://medium.com/quantumblack/spark-udf-deep-insights-in-performance-f0a95a4d8c62

UDFs for DataFrames and Tables (Scala)

- **Python UDF (slow)**
  - Serialize/Deserialize data with Pickle
  - Fetch data block, but invoke UDF row by row

- **Pandas UDF (faster)**

```
import pyarrow as pa
import pandas as pd
```

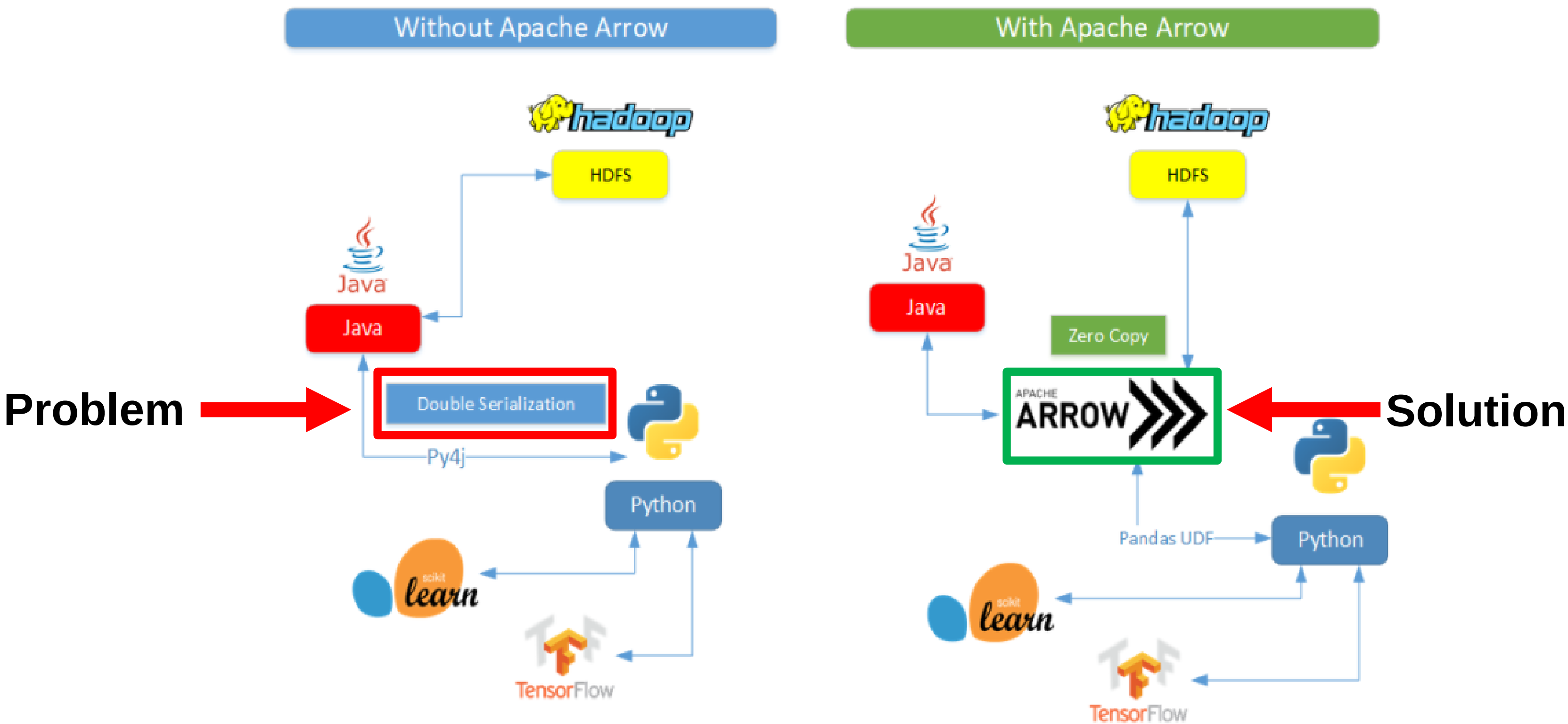
  - Data transfer without the cost of Serialization using Arrow
  - Fetch data block and invoke UDF block by block

Arrow format is an in-memory data format that was specifically designed to exchange the data between different systems efficiently

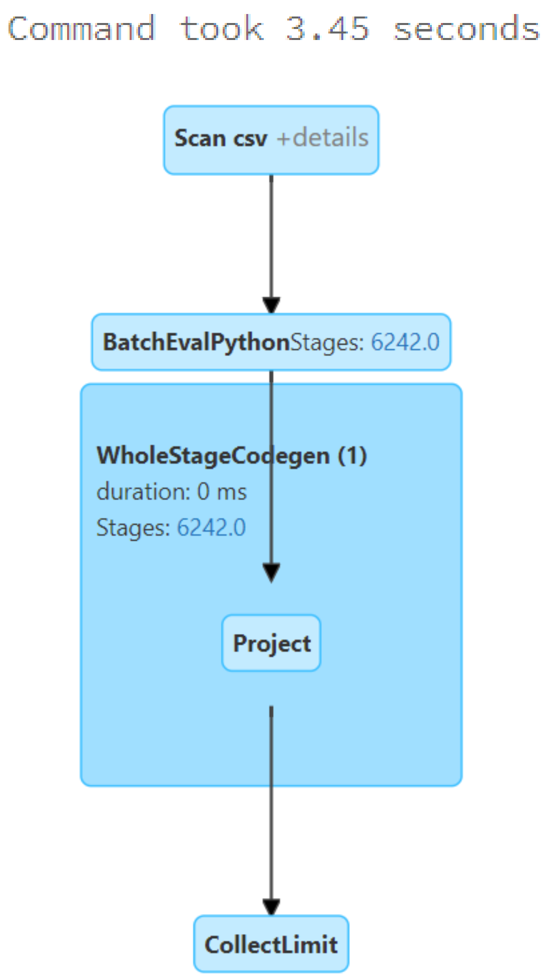
https://databricks.com/blog/2020/05/20/new-pandas-udfs-and-python-type-hints-in-the-upcoming-release-of-apache-spark-3-0.html

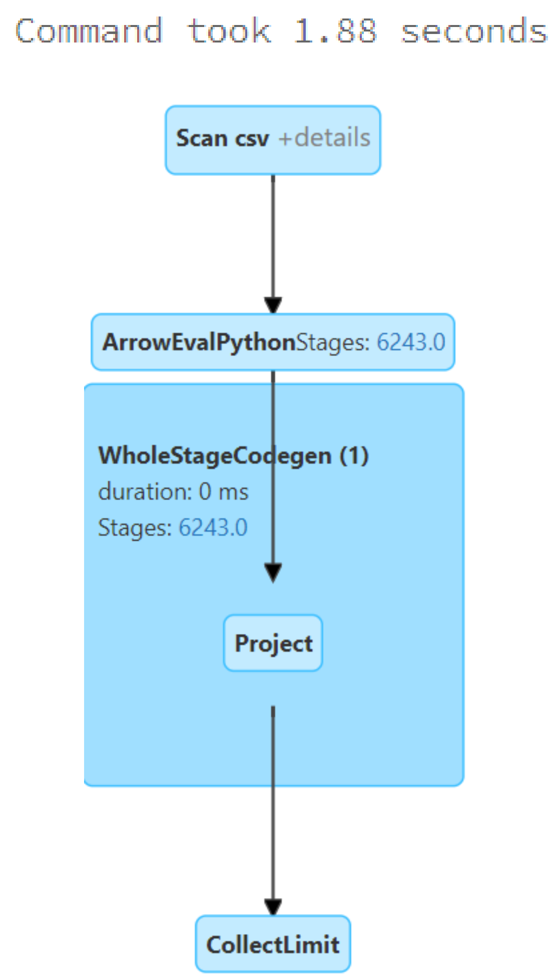
UDFs with/without Apache Arrow on Pandas

Apache Arrow is in-memory columnar format used in Spark to efficiently transfer data between JVM and Python. Beneficial to Python users that work with Pandas/NumPy data



UDFs vs Panda UDFs





UDFs for DataFrames and Tables (Scala)

You can create your own User Defined Functions (UDF). You must:

1. Install correct UDF library for language you are working
2. Create named UDF for either DataFrames or Tables as needed

Need this UDF library

I create my UDF, then call it in DF

To use UDF in SQL table, must register it first

```
import org.apache.spark.sql.functions.udf

case class Purchase(cust_id:Int, purch_id:Int, date:String, time:String, tz:String, amt:Int)

val x = sc.parallelize(Array(
  Purchase(123, 234, "2007-12-12", "20:50", "UTC", 4),
  Purchase(123, 247, "2007-12-12", "15:30", "PST", 8),
  Purchase(189, 254, "2007-12-13", "00:50", "EST", 10),
  Purchase(187, 299, "2007-12-12", "07:30", "UTC", 12)))

val df = sqlContext.createDataFrame(x)

val squareUDF = udf((s:Int) => {s*s})

df.select($"amt",squareUDF($"amt")).show()

// Now make it available for sqlContext.sql

df.registerTempTable("table_df")

sqlContext.udf.register("sqr", (s:Int) => {s*s})

sqlContext.sql("select amt, sqr(amt) from table_df").show()
```

```
+---+--------+
|amt|UDF(amt)|
+---+--------+
|  4|      16|
|  8|      64|
| 10|     100|
| 12|     144|
+---+--------+
```

```
+---+---+
|amt|_c1|
+---+---+
|  4| 16|
|  8| 64|
| 10|100|
| 12|144|
+---+---+
```

Creating UDFs for DataFrames

```python
from pyspark.sql.functions import pandas_udf, PandasUDFType

@pandas_udf('long', PandasUDFType.SCALAR)
def pandas_plus_one(v):
    # `v` is a pandas Series
    return v + 1  # outputs a pandas Series

# Output prior to UDF
spark.range(10).show()

# Output using the UDF to increment all values by 1
spark.range(10).select(pandas_plus_one("id")).show()
```

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

```
+------------------+
|pandas_plus_one(id)|
+------------------+
|                 1|
|                 2|
|                 3|
|                 4|
|                 5|
|                 6|
|                 7|
|                 8|
|                 9|
|                10|
+------------------+
```

Creating **UDFs** for <u>Table</u>

Note you create the UDF in Python, which is then used in SQL API

```
%py

# Lab 21b: Create UDF to square all values
def squared(s):
  return s * s
spark.udf.register("squaredWithPython", squared)

#  Below creates TempView with values 0 - 19
spark.range(1, 10).createOrReplaceTempView("test")
```

```
%sql

-- Lab 21c:  Querying SQL Table using UDF

SELECT id, squaredWithPython(id) as id_squared FROM test;
```

(3) Spark Jobs

	id	id_squared
1	1	1
2	2	4
3	3	9
4	4	16
5	5	25
6	6	36
7	7	49

In Review: Spark SQL

After completing this module, you'll be able to work with Spark SQL including:

Transformations 01 Notebook

- Columns and Expressions
- Operators and Methods
- Transformations and Actions

Transformation 02

- Lab 01: Dates and Timestamps
- Lab 02: Aggregations
- Lab 03: Widgets
- Non-Aggregate functions
- Other built-in functions
- Lab 04: User Defined Functions