

# **Lessons from the Field, Episode II:**

## **Applying Best Practices to Your Apache Spark™ Applications**

Silvio Fiorito  
@granturing

#SAIExp9

# About Me

Resident Solutions Architect @ Databricks

Developer, application security engineer,  
consultant, instructor

Using Spark since ~2012

Work with Databricks customers in banking,  
manufacturing, info-sec, new & experienced Spark users

SSE17: [Lessons from the Field](#)



To build more reliable,  
& better performing Spark  
applications we should  
understand...

File Formats & Data Types

Partitioning, Shuffling, & Parallelization

Using Query Plans to Debug & Tune

## CSV, JSON, Parquet, Avro, ORC,...

File formats have different  
benefits and tradeoffs that  
we need to be aware of...

# Reading CSV Files

Raw text

```
1000,user1,"2010-01-23",123.34,.....  
2002,user2,"2008-02-14",349.02,.....
```

Read each  
line as String

```
"1000,user1,2010-01-23,123.34,.....",  
"2002,user2,2008-02-14,349.02,.....",
```

Tokenize

```
Array("1000","user1","2010-01-23","123.34",.....),  
Array("2002","user2","2008-02-14","349.02",.....),
```

Convert to types  
& create rows

1000	user1	14632	123.34	
2002	user2	13923	349.02	

# Common Issues with CSV

- Slow startup on large datasets
- Poor parallelization due to non-splittable files
- Be mindful of “wide” files
  - GC issues
  - Spark 2.4 column pruning ([SPARK-24244](#))

GC Time	Min	25th percentile	Median	75th percentile	Max
Column Pruning <b>Off</b>	0.5 s	0.5 s	0.6 s	0.6 s	0.6 s
Column Pruning <b>On</b>	9 ms	31 ms	61 ms	64 ms	70 ms

Selecting 2 out of 200 columns  
from 3.6GB compressed dataset

# Schema Inference

- CSV
  - `inferSchema` causes full scan of data
  - Specify schema or use tables
  - Be precise, use the right types!
- JSON
  - Full scan of data
  - Specify schema or use tables

	Shuffle Write
strings	217.0 MB
doubles	159.0 MB
decimals	145.3 MB

Aggregate query on  
2 columns & 10 million rows



# Schema Inference

- Parquet
  - If no `_metadata`, use schema from first file
  - If `mergeSchema` read footers from all files
  - Specify schema or use tables
- ORC
  - Reads first file for schema
  - Specify schema or use tables

# Optimizing Parquet Storage

- Embedded column statistics
  - min, max, count
  - Useful for filtering
- Most effective on sorted data
  - Order by filter column when writing
  - Reduce data reads by skipping files

```
select * from mytable  
where id between 1 and 100
```

### UNSORTED

rows output: 1,108,655 est: N/A

**Filter**  
rows output: 100 est: N/A

Duration	Tasks: Succeeded/Total	Input
12 s	32/32	209.3 MB

### SORTED

rows output: 34,650 est: N/A

**Filter**  
rows output: 100 est: N/A

Duration	Tasks: Succeeded/Total	Input
2 s	32/32	12.8 MB

## Challenges with Parquet Data Skipping

- Really most effective on single sort key
- Need to be careful with data skew
- How to maintain performance as new data appended?
- Data Skipping and ZORDER clustering
  - [Processing Petabytes of Data in Seconds with Databricks Delta](#)

# Parallelization with File Formats

- Spark 2.0 default split 128MB
  - `spark.sql.files.maxPartitionBytes`
- Bin packing of smaller files
  - Auto-compaction, avoids coalesce
- Based on file-size
  - High compression ratio may result in smaller files
  - Data is decoded & uncompressed when reading

# Parallelization & File Sizes

- Controlling file sizes during ETL
- Generating more rows in queries (`explode`, `flatMap`, broadcast joins)
- Solution - `repartition`?
  - Adds shuffle boundary, memory & I/O constraints
- Consider instead
  - `maxPartitionBytes` (for splittable input)
  - `maxRecordsPerFile` (for controlling output)



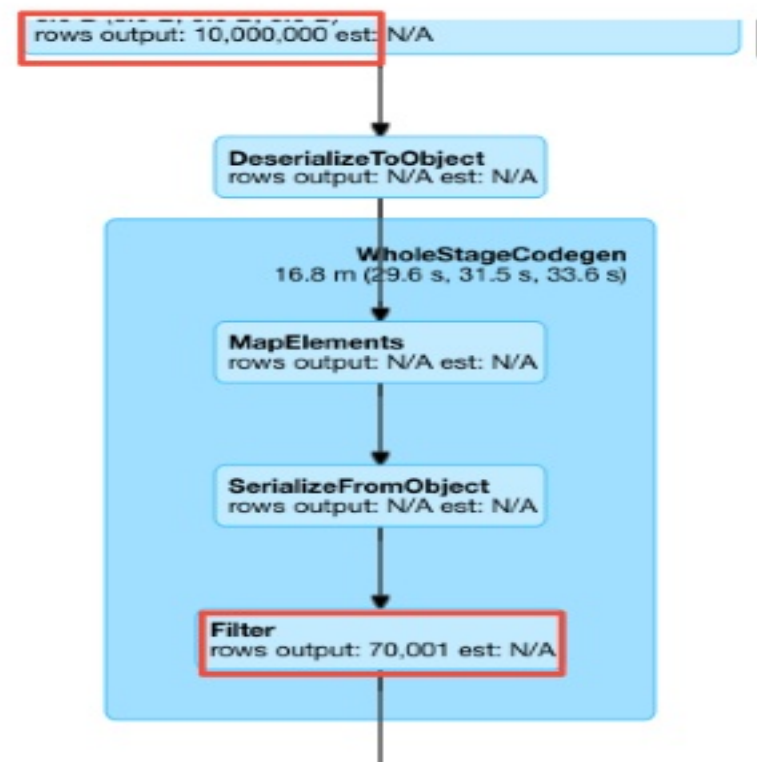
# Using Spark UI & Query Plans

- SQL UI is invaluable to understanding execution
- Identify stage boundaries (shuffles)
- Use metrics for diagnosing your query
  - Are filters and column pruning applied?
  - How large are your shuffles?
  - How much memory usage and spilling?

# Datasets & Order of Operations

```
case class Input(id: Long, col1: String)
case class Output(id: Long, col1: String, col2: String)
def transformToOutput(input: Input): Output = {
  ...
}

val query = basedata.as[Input]
  .map(i => transformToOutput(i))
  .filter('id <= 700000)
```



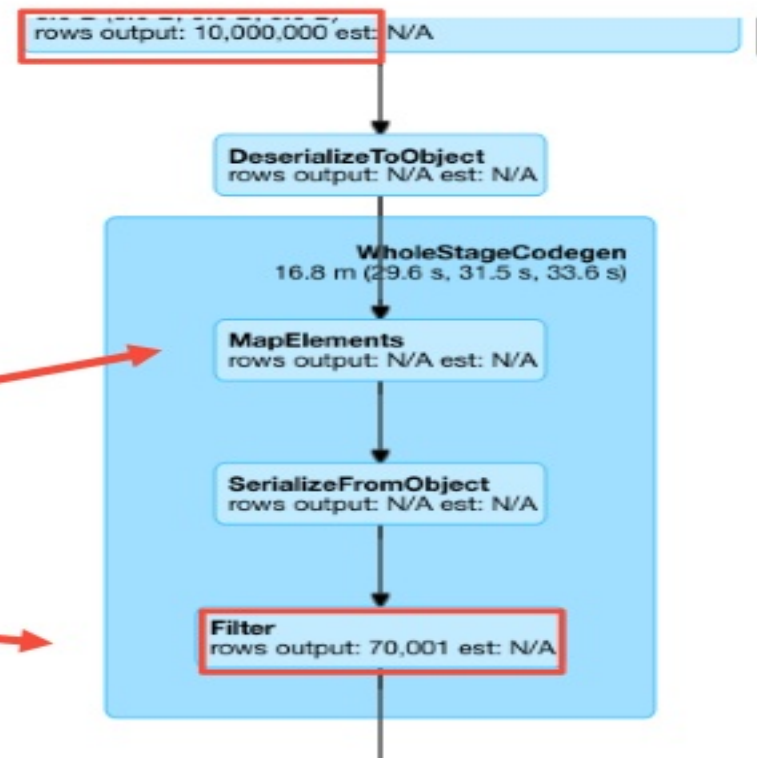


# Datasets & Order of Operations

```
case class Input(id: Long, col1: String, col2: String)
case class Output(id: Long, col1: String, col2: String)
def transformToOutput(input: Input): Output = {
  ...
}
```

```
val query = basedata.as[Input]
  .map(i => transformToOutput(i))
  .filter('id <= 700000)
```

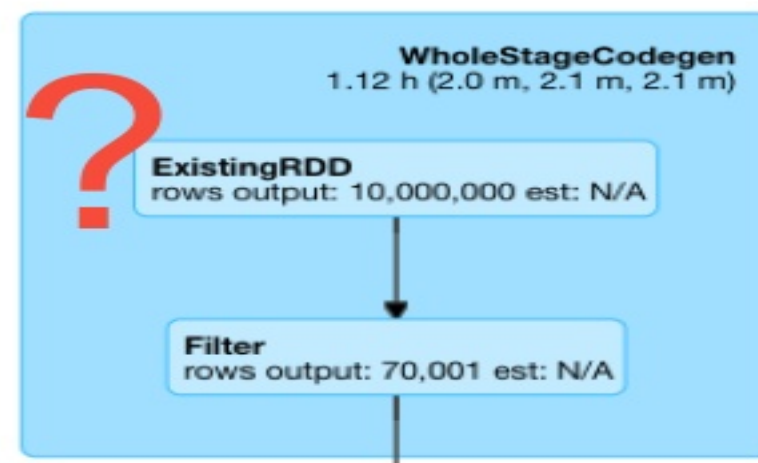
Dataset `map` breaks  
lineage of "id" column,  
no predicate pushdown!



# Moving Between DataFrames & RDDs

```
output = df.rdd.map(myPythonFunc)

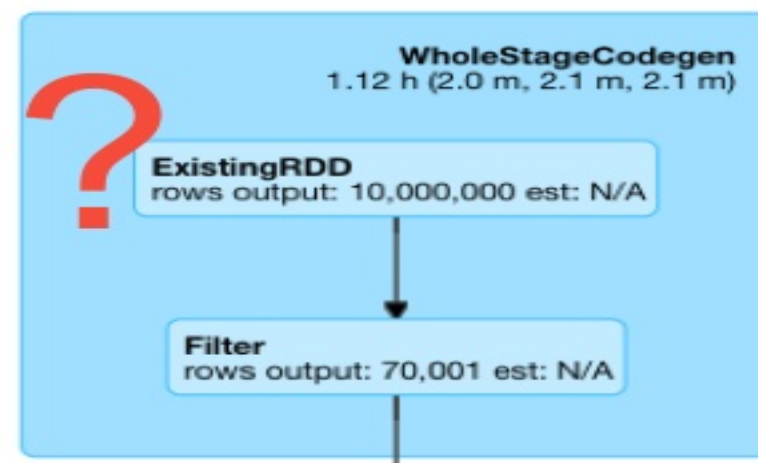
outputDF = spark.createDataFrame(
    output,
    ["id", "col1", "col2"]
).where("id <= 700000")
```



# Moving Between DataFrames & RDDs

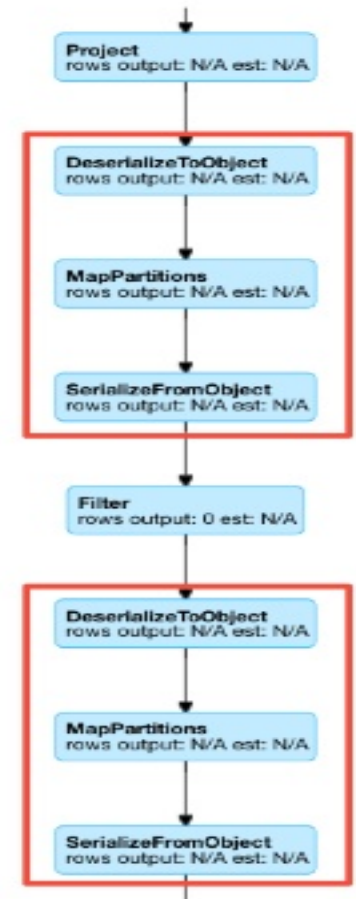
```
output = df.rdd.map(myPythonFunc)
outputDF = spark.createDataFrame(
    output,
    ["id", "col1", "col2"]
).where("id <= 700000")
```

Converting to RDD breaks  
DataFrame lineage, no predicate  
pushdown, no column pruning, no SQL plan,  
and less efficient PySpark RDD transformations...



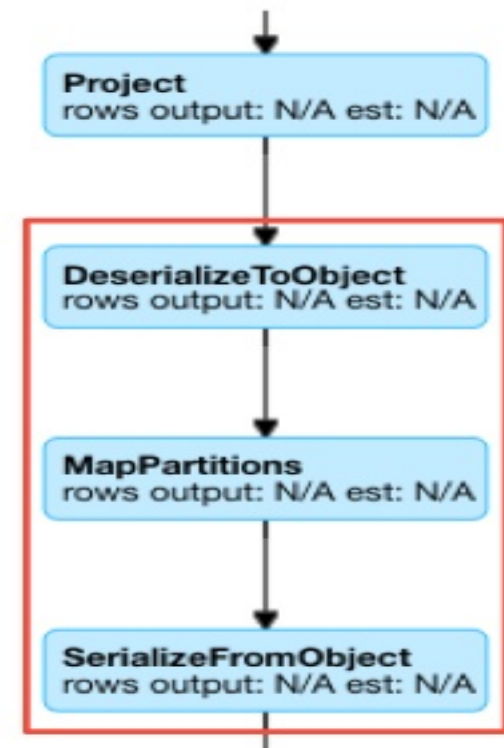
# Using SparkR UDFs

```
step1 = dapply(input, firstUDF, step1Schema) # UDF
step2 = filter(step1, step1$id2 >= 5000000) # Spark filter
step3 = dapply(step2, secondUDF, step3Schema) # UDF
```



# Using SparkR UDFs

```
bothUDFs <- function(input) {  
  step1 <- firstUDF(input)  
  step2 <- subset(step1, id2 >= 5000000)  
  step3 <- secondUDF(step2)  
  
  return(step3)  
}  
  
singleStep = dapply(input, bothUDFs, step3Schema)
```



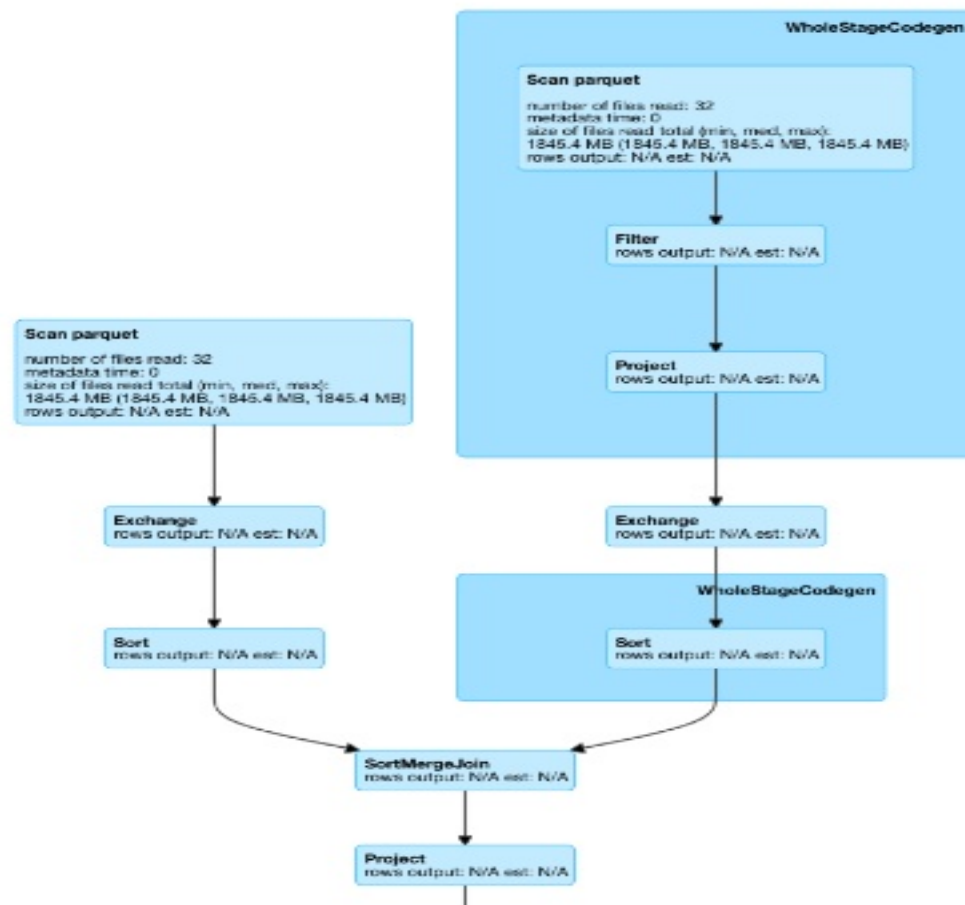


# Self-Joins

```
val basedata_ex = basedata  
  .withColumn("id", ...)   
  .withColumn("mycol", ...)
```

```
val joined = basedata  
  .join(basedata_ex, Seq("id"), "left")
```

- Two DataFrames referring to same data
- Typically, some transformation applied to original DF
- Join second DF to original DF (look ahead/behind, shifting rows, self reference, etc)

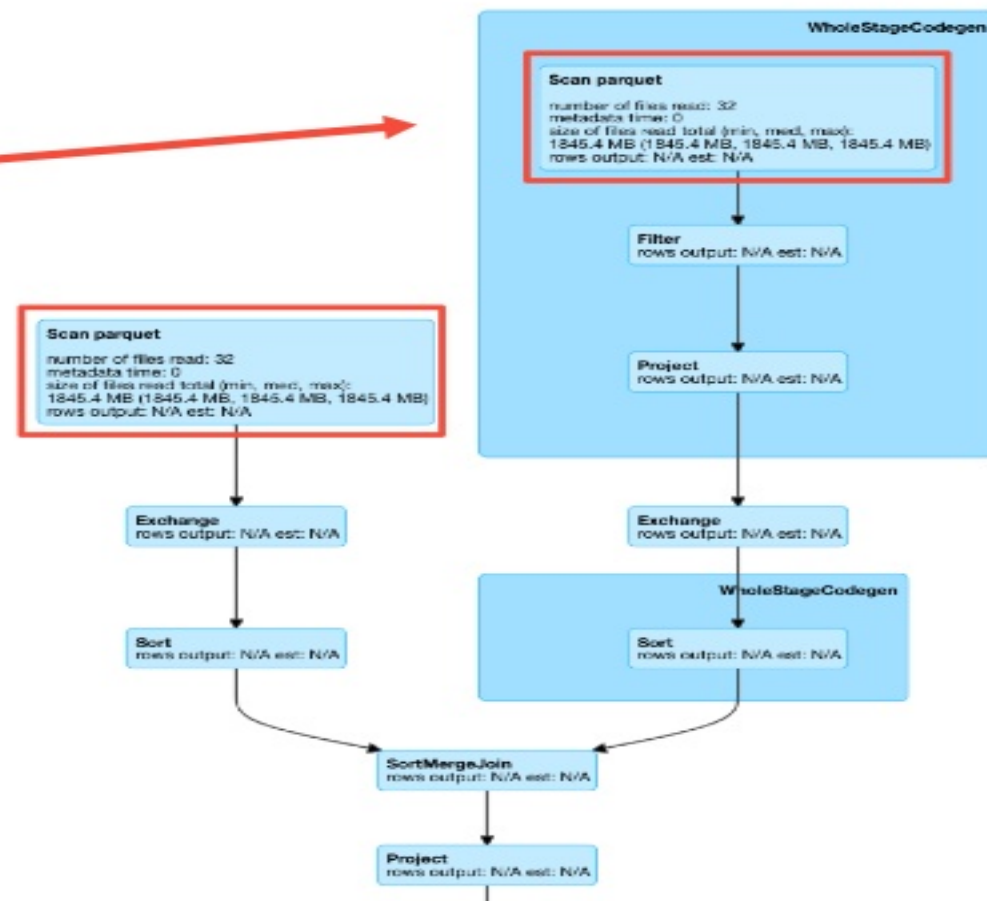


# Self-Joins

Reading the SAME data twice!

Instead consider window functions,  
`flatMap`, `explode`

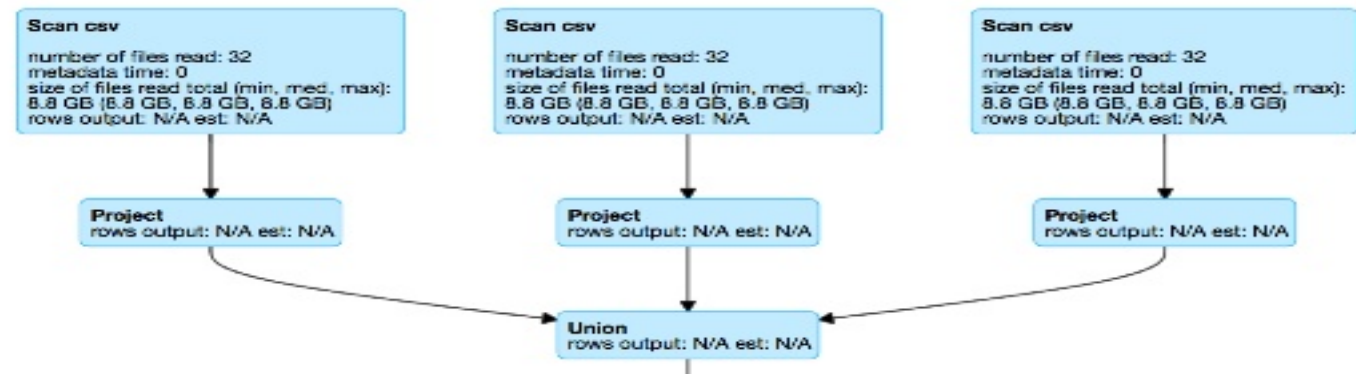
If absolutely necessary then  
try caching, checkpointing, or  
[local checkpointing](#)



# Repetitive Unions

```
for config in configs:  
    temp = computedDF(input, config)  
    unioned_dfs = unioned_dfs.union(temp)
```

- Typically for different transformations to same record
- Sometimes used for Python distributed Cross Validation





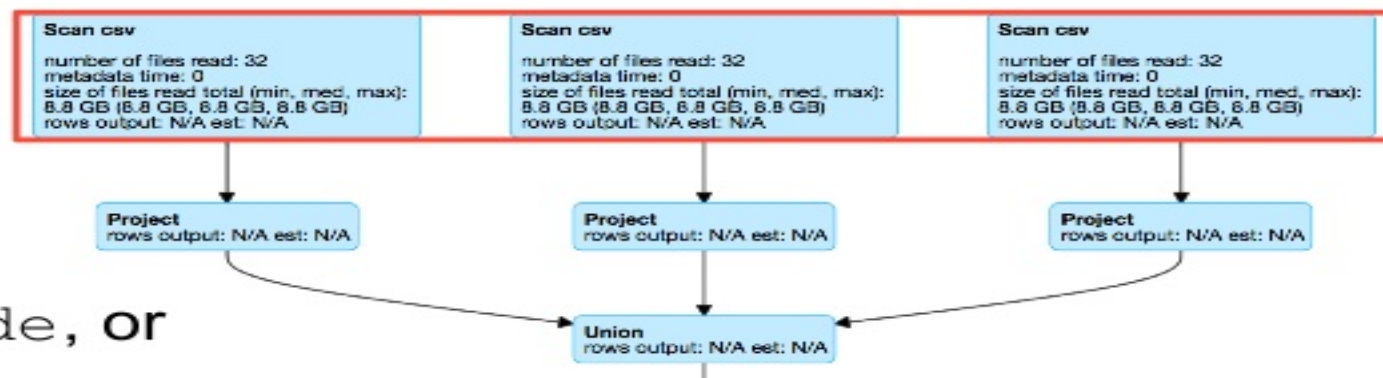
# Repetitive Unions

Reading the SAME data!

Increases number of tasks  
and Catalyst complexity  
with each union

Instead consider `flatMap`, `explode`, or  
broadcast join (dep. on data size)

If absolutely necessary then  
try caching, checkpointing, or  
[local checkpointing](#)



# UDFs and Datasets

- PLEASE – Learn the built-in [functions](#)!
  - Prefer [Pandas UDFs](#) to PySpark UDFs
- Filter and project early
  - UDFs, lambdas are opaque to Catalyst
- Prefer DataFrames & Datasets
- SparkR – [100x Faster UDFs](#)

## How Can You Learn More?

- Spend the time to understand your query plan
  - Is Spark doing what you expect?
- Understand how different transformations, UDFs, and configs impact execution
- If you're not sure...go to the source!
  - [Apache Spark GitHub](#)

# QUESTIONS?