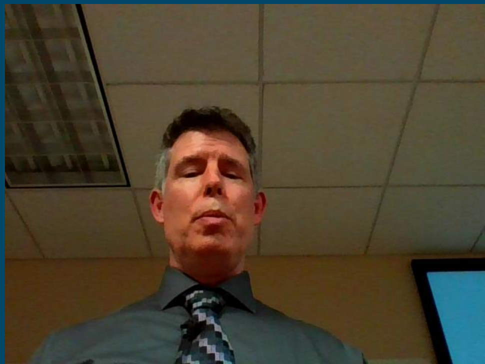
Advance DataFrame Considerations

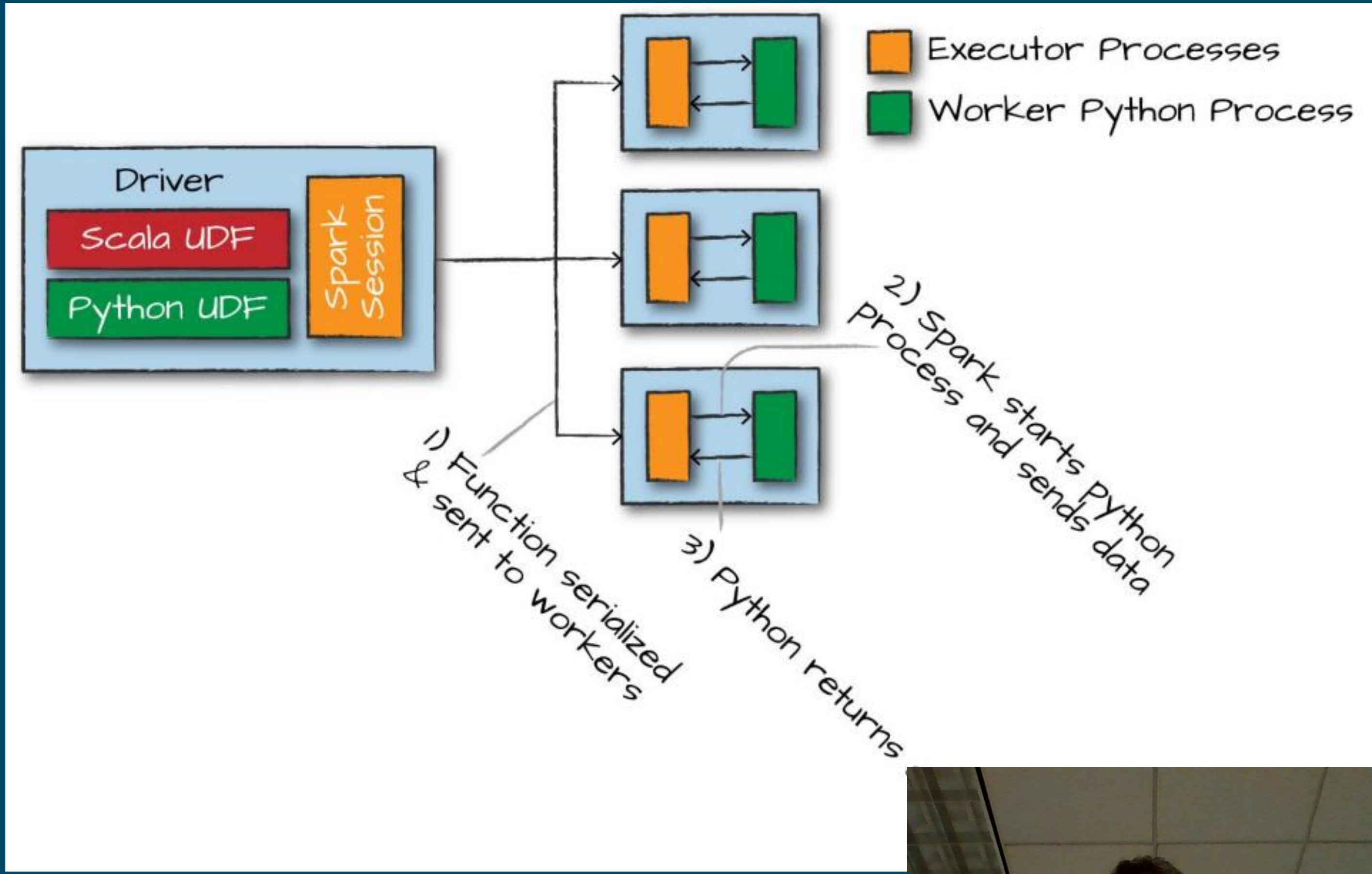
Operations and optimizations

User Defined Function (UDF)

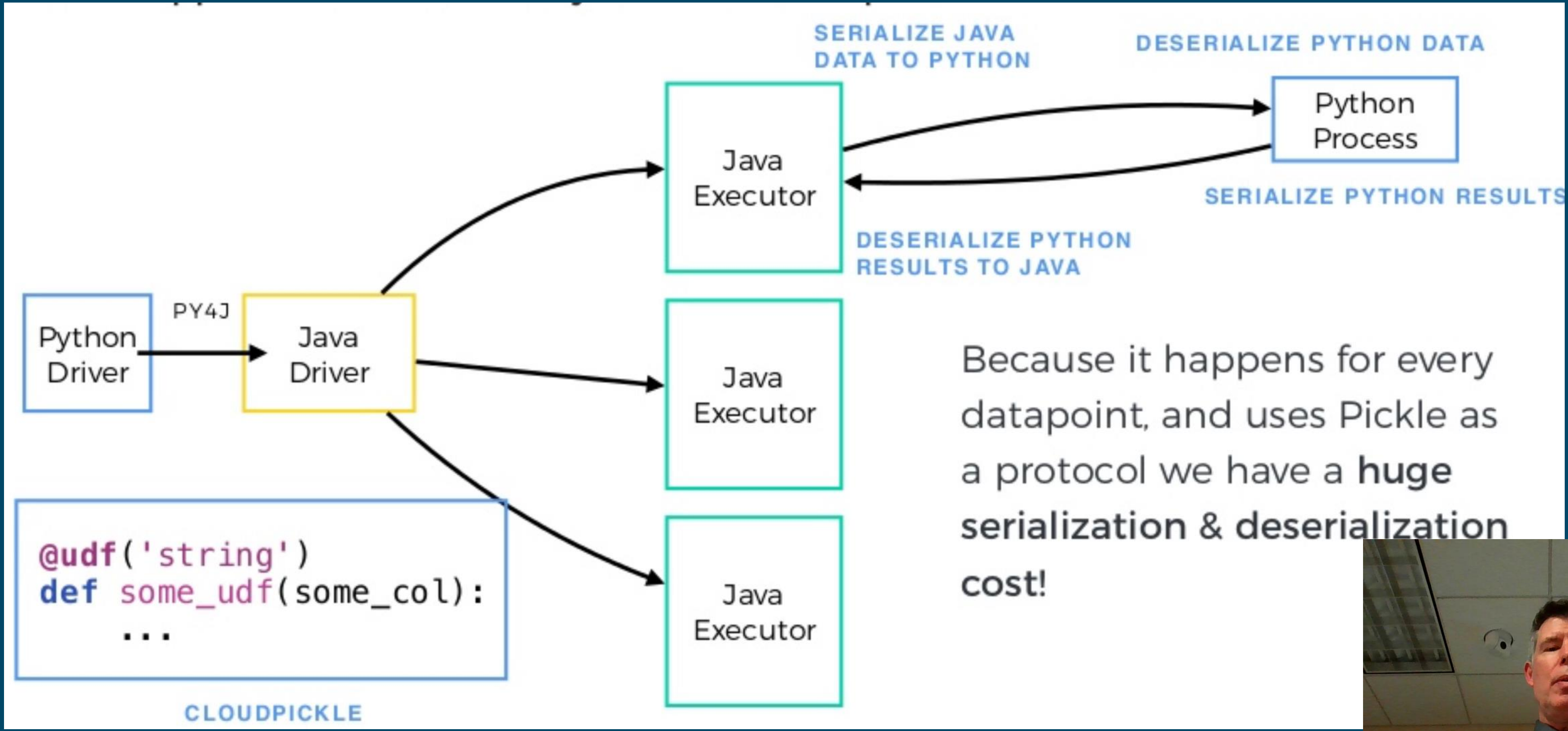


UDF

- UDF runs on each row of a DataFrame/SQL table
- Row is serialized & sent to worker (JVM) process
  - Added overhead of Python process for PySpark UDF
  - Process can fail because of memory, etc.
- For production systems, write UDF in Java or Scala



Python objects serialized and deserialized to Java (over pipe), which is slow (for UDFs)



UDF: User Defined Function is a procedure that runs over a DataFrame or SQL table. Calls to the UDF occur for **each row**. So many calls to the Py

Example PySpark native vs UDF

- Split text, by space, into words and put into vector

Native API split is fast

```
from pyspark.sql.functions import split,col
df1 = df.select(split(col("Description"), " ").alias("array_col"))
df1.show(1)
```

▸ (1) Spark Jobs

▸ df1: pyspark.sql.dataframe.DataFrame = [array_col: array]

```
+--------------------+
|           array_col|
+--------------------+
|[WHITE, HANGING, ...|
+--------------------+
only showing top 1 row
```

UDF split (mySplitUDF) is slow

```
from pyspark.sql.functions import udf,col

def splitUDF(col):
  if col is not None:
    return col.split(' ')

mySplitUDF = udf(splitUDF)

df2 = df.select(mySplitUDF(col('Description')))
display(df2)
```

▸ (1) Spark Jobs

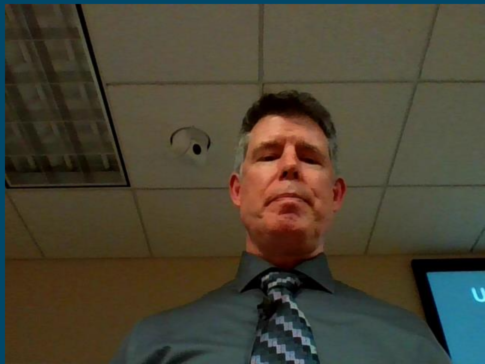
▸ df2: pyspark.sql.dataframe.DataFrame = [splitUDF(

	splitUDF(Description)
1	[WHITE, HANGING, HEART, T-LIGHT, HOLDER]

UDF try-catch

- Because PySpark UDF runs in its own process, good practice to catch error

```
def splitUDF(col):
  try:
    if col is not None:
      return(col.split(' '))
  except:
    return(None)
```



UDF for SQL

- UDF is for DataFrame
- Can be registered to work in SQL context

Register UDF in SQL context

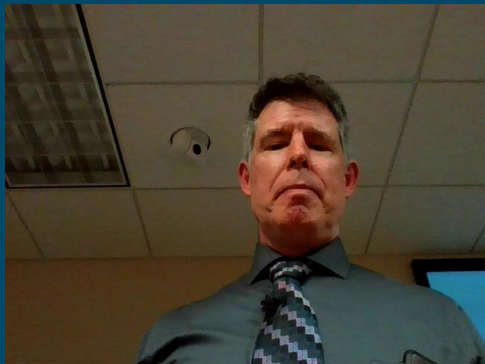
```python
from pyspark.sql.types import StringType
sqlContext.udf.register("splitUDF", splitUDF, StringType())
```

```sql
%sql
select *,splitUDF(Description)
from retail
```

▸ (1) Spark Jobs

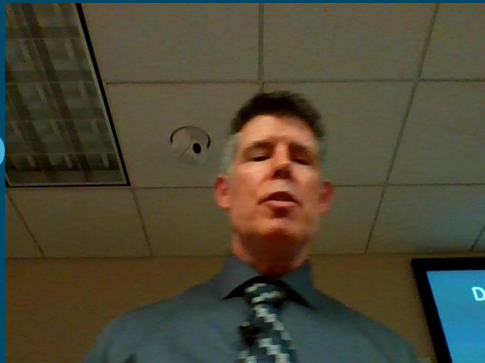
	Description	Quantity	InvoiceDate	UnitPrice	CustomerID	Country	splitUDF(Desc
1	WHITE HANGING HEART T-LIGHT HOLDER	6	2010-12-01 08:26:00	2.55	17850	United Kingdom	[WHITE, HANG
2	WHITE METAL LANTERN	6	2010-12-01 08:26:00	3.39	17850	United Kingdom	[WHITE, META

Complex types in columns



DataFrame supports complex column types

- Array
  - Functions for size, split, and find in arrays
- Structure
  - Defined
    - with Struct
    - Read from JSON
  - Functions to expand into new columns (col.*)
- Maps
  - Functions to create a map from columns, explode a map to



DataFrame supports array column types

- Array
  - Functions to split, find in arrays

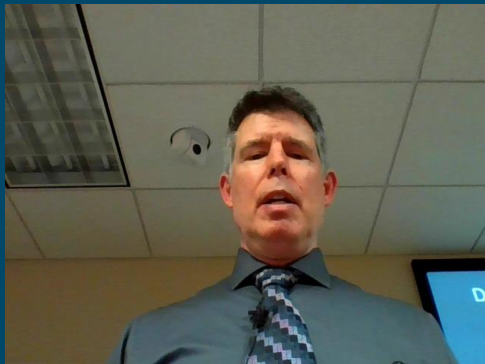
```
# in Python
from pyspark.sql.functions import array_contains
df.select(array_contains(split(col("Description"), " "), "WHITE")).show(2)

-- in SQL
SELECT array_contains(split(Description, ' '), 'WHITE') FROM dfTable
```

```
+--------------------------------------------+
|array_contains(split(Description, ), WHITE)|
+--------------------------------------------+
|                                        true|
|                                        true|
+--------------------------------------------+
```

  - Collect into a list

```
# in Python
from pyspark.sql.functions import collect_set, collect_list
df.agg(collect_set("Country"), collect_list("Country")).show()
```



DataFrame supports structure column types

- Structure
  - Defined
    - with Struct
    - Read from JSON
  - Functions to expand into new columns (col.*)

```
complexDF.select("complex.*")

-- in SQL
SELECT complex.* FROM complexDF
```

Expanding column structure with *

- Id and name attributes within Json in column

	department
1	{"id": "123456", "name": "Computer Science"}
2	{"id": "789012", "name": "Mechanical Engineering"}
3	{"id": "345678", "name": "Theater and Drama"}
4	{"id": "901234", "name": "Indoor Recreation"}

```
display(unionDF.select('department.*').limit(5))
```

(3) Spark Jobs

	id	name
1	123456	Computer Science
2	789012	Mechanical Engineering
3	345678	Theater and Drama
4	901234	Indoor Recreation

Expanding embedded rows with explode

- Multiple employees in column

employees
[{"firstName": "michael", "lastName": "armbrust", "email": "no-reply@berkeley.edu", "salary": 100000}, {"firstName": "xiangrui", "lastName": "meng", "email": "no-reply@stanford.edu", "salary": 120000}]
[{"firstName": "matei", "lastName": null, "email": "no-reply@waterloo.edu", "salary": 140000}, {"firstName": null, "lastName": "wendell", "email": "no-reply@berkeley.edu", "salary": 160000}]
[{"firstName": "michael", "lastName": "armbrust", "email": "no-reply@berke … "lastName": "wendell", "email": "no-reply@berkeley.edu", "salary": 160000}]
[{"firstName": "xiangrui", "lastName": "meng", "email": "no-reply@stanford. … "lastName": null, "email": "no-reply@waterloo.edu", "salary": 140000}]

```
from pyspark.sql.functions import explode
exploded_df = unionDF.select(explode("employees").alias("e"))
display(exploded_df)
```

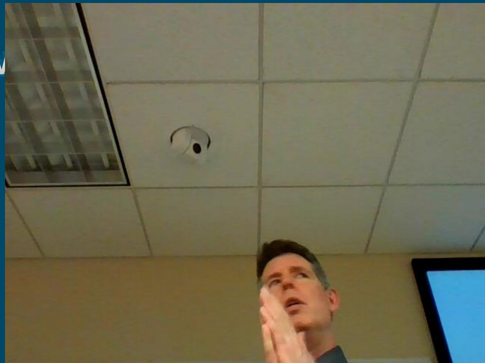
(3) Spark Jobs

exploded_df: pyspark.sql.dataframe.DataFrame = [e: struct]

	e
1	{"firstName": "michael", "lastName": "armbrust", "email": "no-reply … 100000}
2	{"firstName": "xiangrui", "lastName": "meng", "email": "no-reply@s … 120000}
3	{"firstName": "matei", "lastName": null, "email": "no-reply@waterlo …
4	{"firstName": null, "lastName": "wendell", "email": "no-reply@berk …
5	{"firstName": "michael", "lastName": "armbrust", "email": "no-reply … 100000}

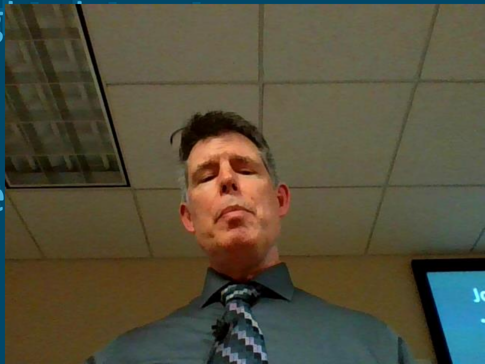
Joins

Try to reduce the amount of data transmitted over netv



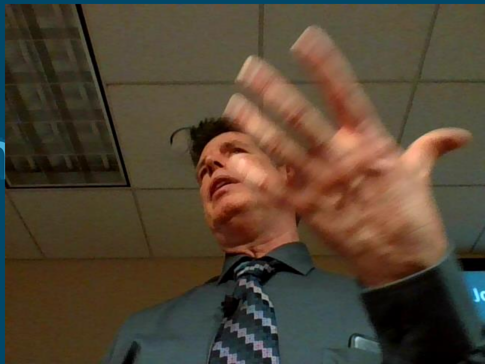
Join types

- Inner joins
  - keep rows with keys that exist in the left and right datasets
- Outer joins
  - keep rows with keys in either the left or right datasets
- Left outer joins
  - keep rows with keys in the left dataset
- Right outer joins
  - keep rows with keys in the right dataset
- Left semi joins
  - keep the rows in the left, and only the left, dataset where the key appears in the rig
- Left anti joins
  - keep the rows in the left, and only the left, dataset where they do not appear in the



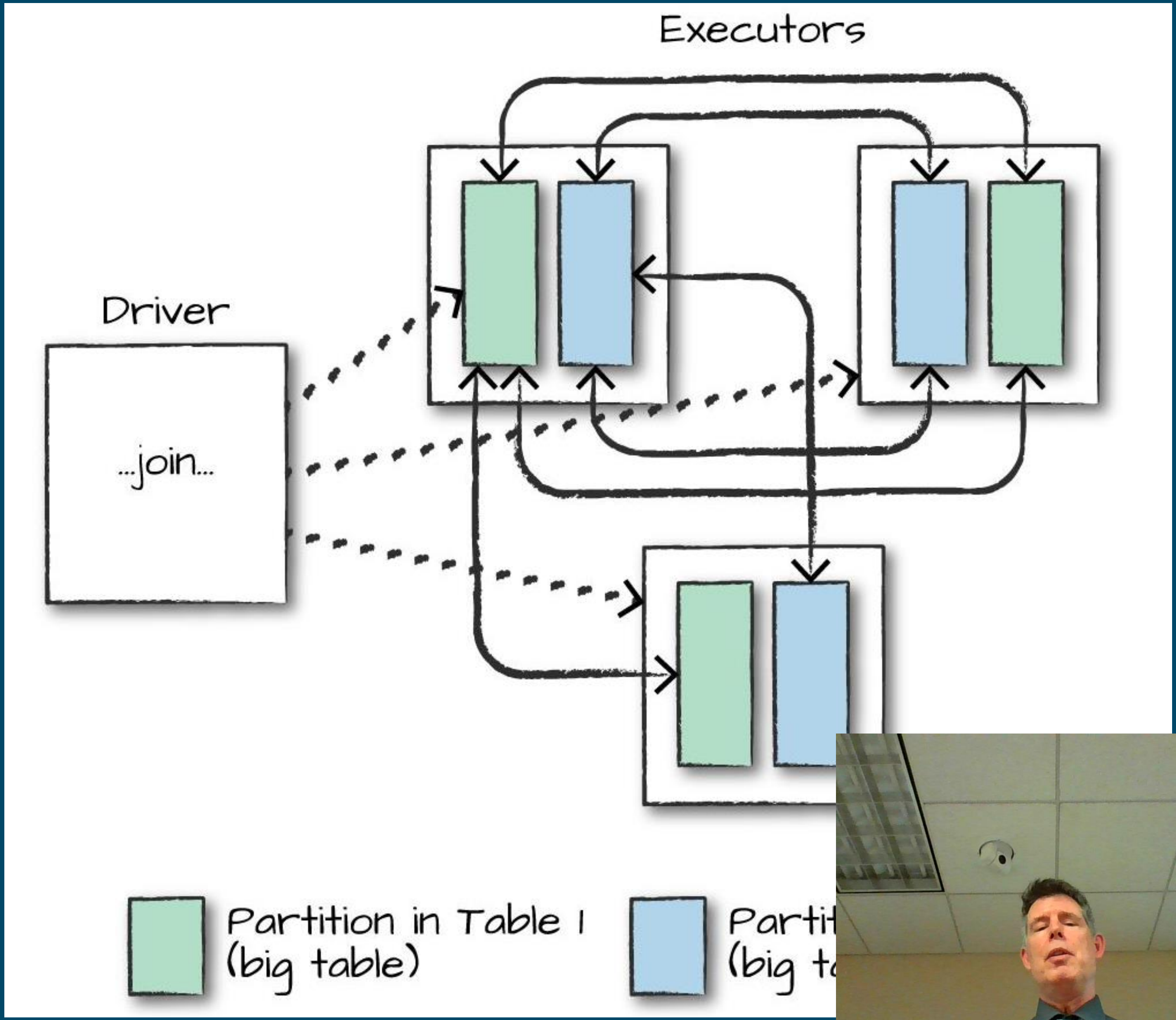
Joins are slow

- Data is distributed over cluster
- To join, each row from each table is compared
- So, each row has to be sent to each node
- Consequently, joins move lots of data over the network
- Network bandwidth becomes a bottleneck
- Improve join by transmitting less data
    - Send small tables to nodes with large tables
    - Filter data (push down predicated) before transmittin



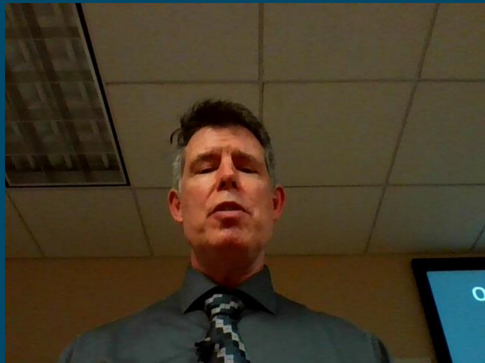
Joining large tables transmits all data

- Each node receives data from all other nodes
- Each partition is sent to all other nodes



Optimize when there is a smaller table

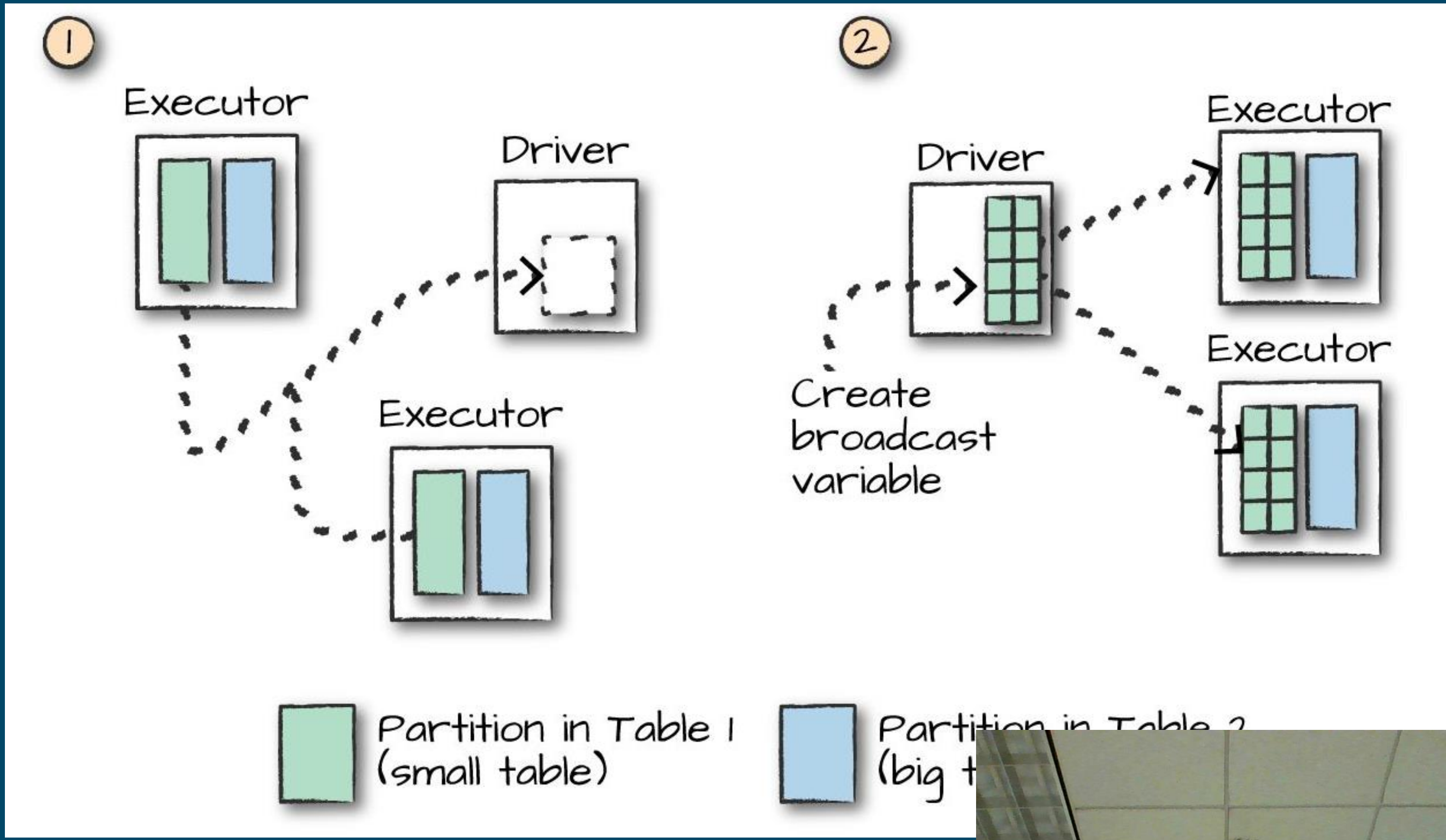
- Broadcast the small table to improve the join
- When one of the join tables is small enough to fit into memory, then broadcast that table to the nodes
  - Reduces the total data transmitted
  - Can offer a hint to the Catalyst optimizer, but generally it optimizes OK



Broadcasting a table to join

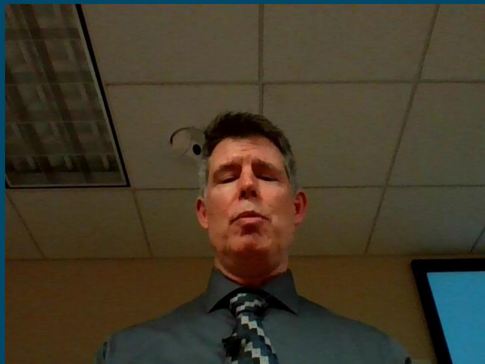
1. Table is sent to Driver
2. Special *broadcast variable* makes the data in the Driver accessible to all nodes

- Only transmitting the smaller table reduces data transmitted



df = blue.join(broadcast(green), blue.id =

Reading data and partitions



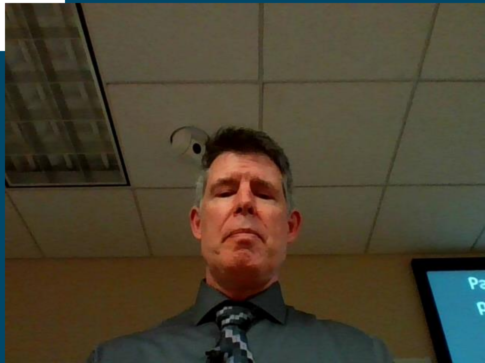
Partition the data during read for subsequent parallel processing

- Data may already be distributed over the nodes, where each node contains a partition
- Data that is not partitioned, can be partitioned during the read
  - From a file or SQL table

```
sqlContext.setConf("spark.sql.shuffle.partitions", "200")
```

```
# in Python
dbDataFrame = spark.read.format("jdbc")\
 .option("url", url).option("dbtable", tablename).option("driver", driver)\
 .option("numPartitions", 10).load()
```

- Such partitions distribute the data over the nodes, allows for subsequent parallel processing



Important to remember

- PySpark UDF runs slowly in a separate Python process
  - Catch errors
- DataFrame support complex types
  - Arrays or structures as a column data type
  - Explode pulls multipe rows in a cell into rows
- DataFrame supports a variety of joins
  - Joins are slow because the tables must be copied to a
    nodes