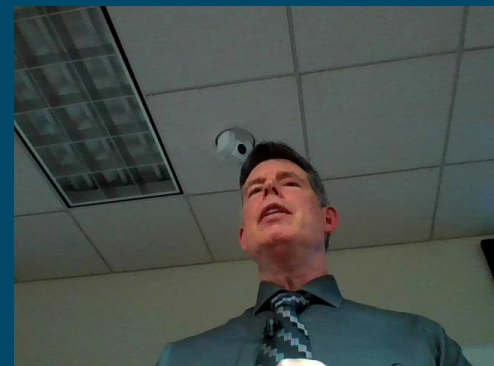
Spark Programming Architecture

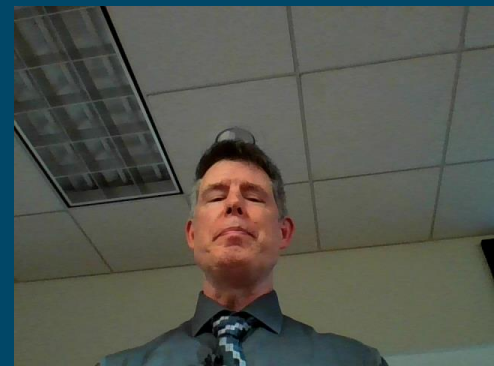
Conceptual model for spark programming



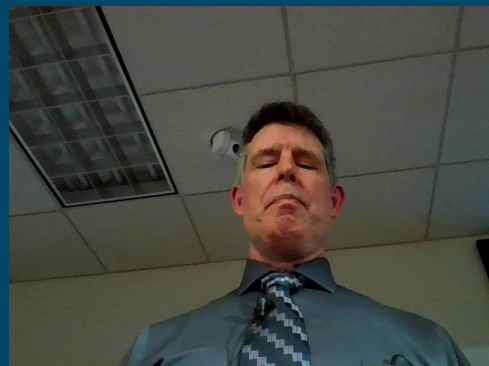
Apache Spark

- is a unified computing engine
  and a set of libraries
  for parallel data processing
  on computer clusters

- Let’s look at each of these elements



What’s a cluster?



Cluster

Cluster

- A computer cluster is a set of loosely or tightly connected computers that work together so that, in many respects, they can be viewed as a single system
- The Beowulf cluster is a cluster implemented on multiple identical commercial off-the-shelf con connected with a TCP/IP Ethernet local area ne

Wikipedia

Cluster usage

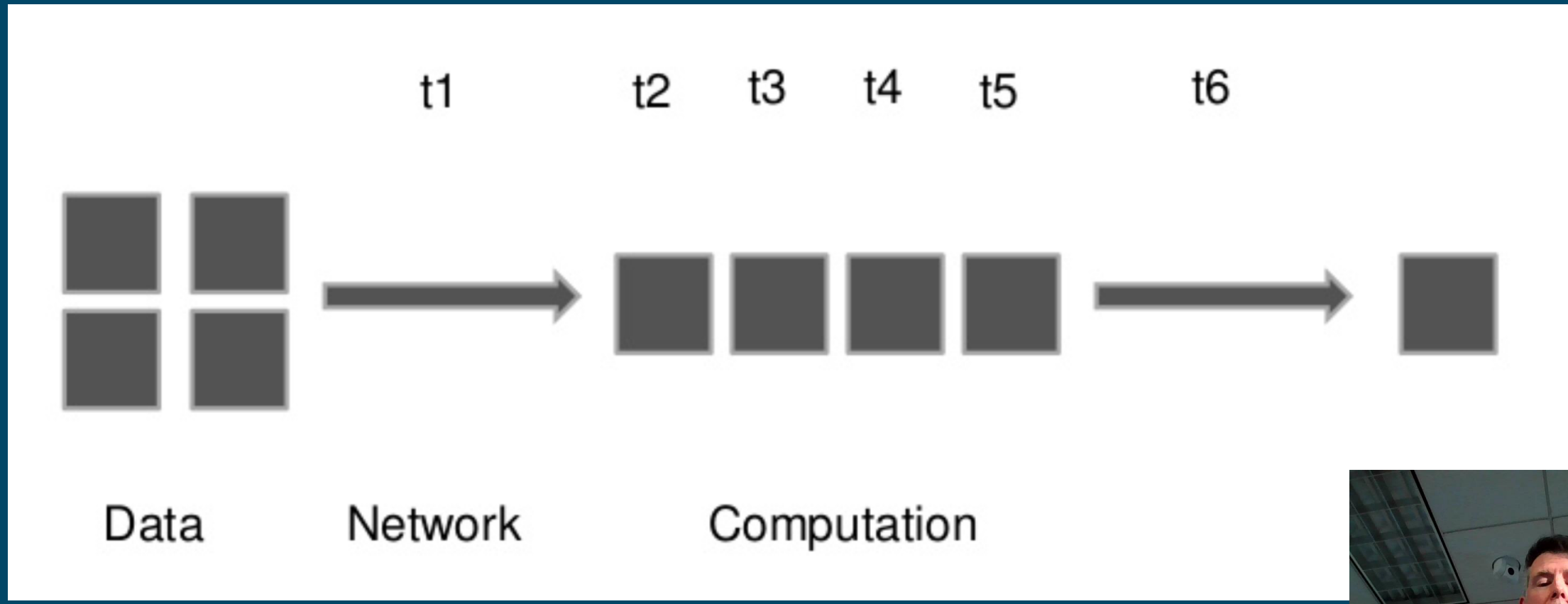
- Clusters provide
  - Many nodes (computers)
  - Each node has
    - local disk storage
    - multiple CPUs or cores on each node
- Cluster processing typically…
  - Runs a task on each CPU
    - Task is a copy of a program
  - Ideally, accesses data that is on the node's (local) disk

What’s parallel processing?

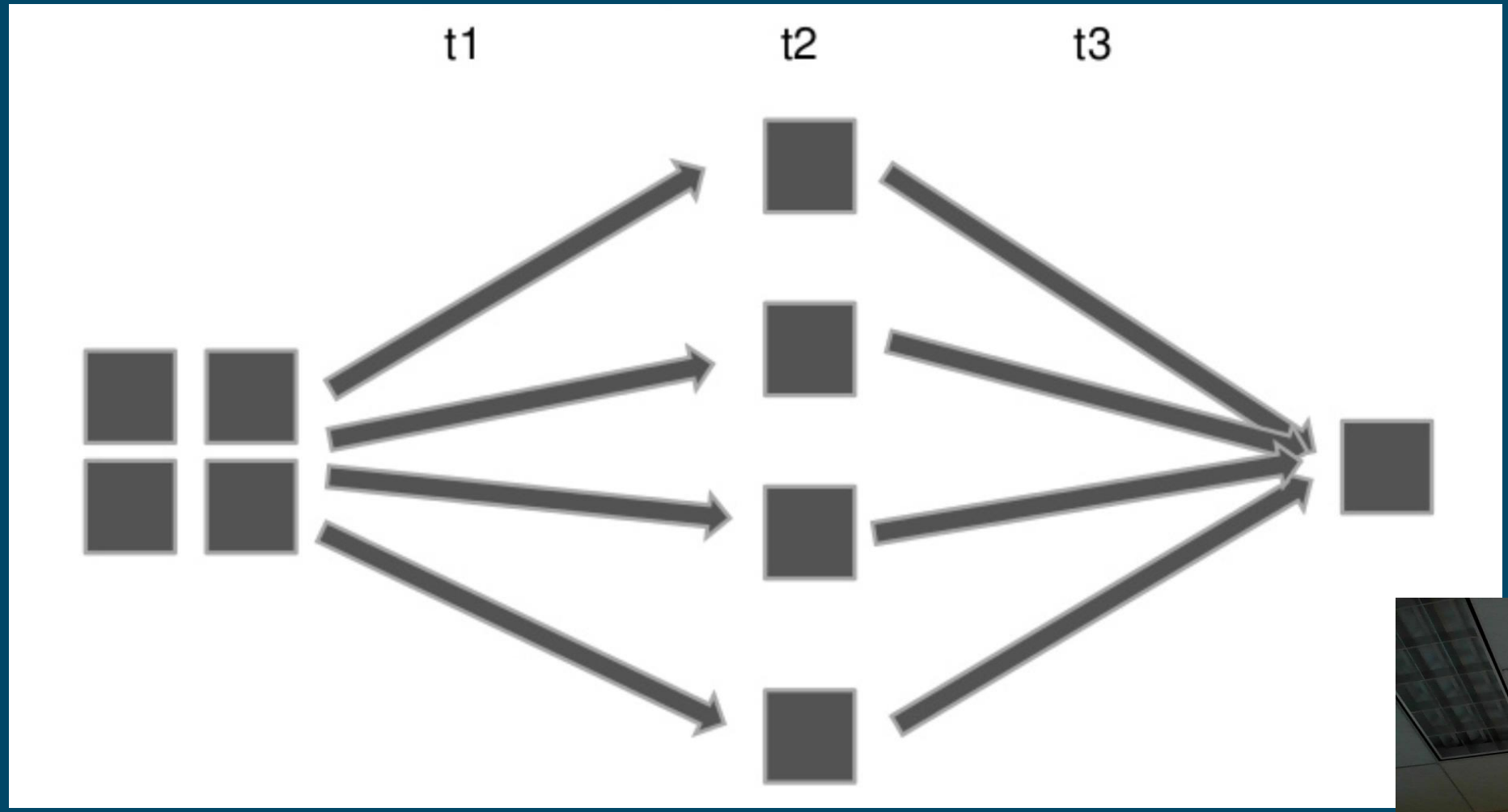
Parallel data processing

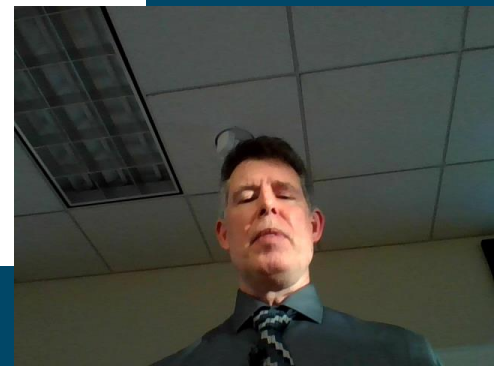
- many calculations or the execution of processes are carried out simultaneously.
- Large problems can often be divided into smaller ones
  - which can then be solved at the same time.
- In Spark,
  - a large dataset is divided into smaller datasets,
  - each of which is processed by task,
  - where tasks are distributed over the cluster.

Data sequentially processed (time: t1 … t6)

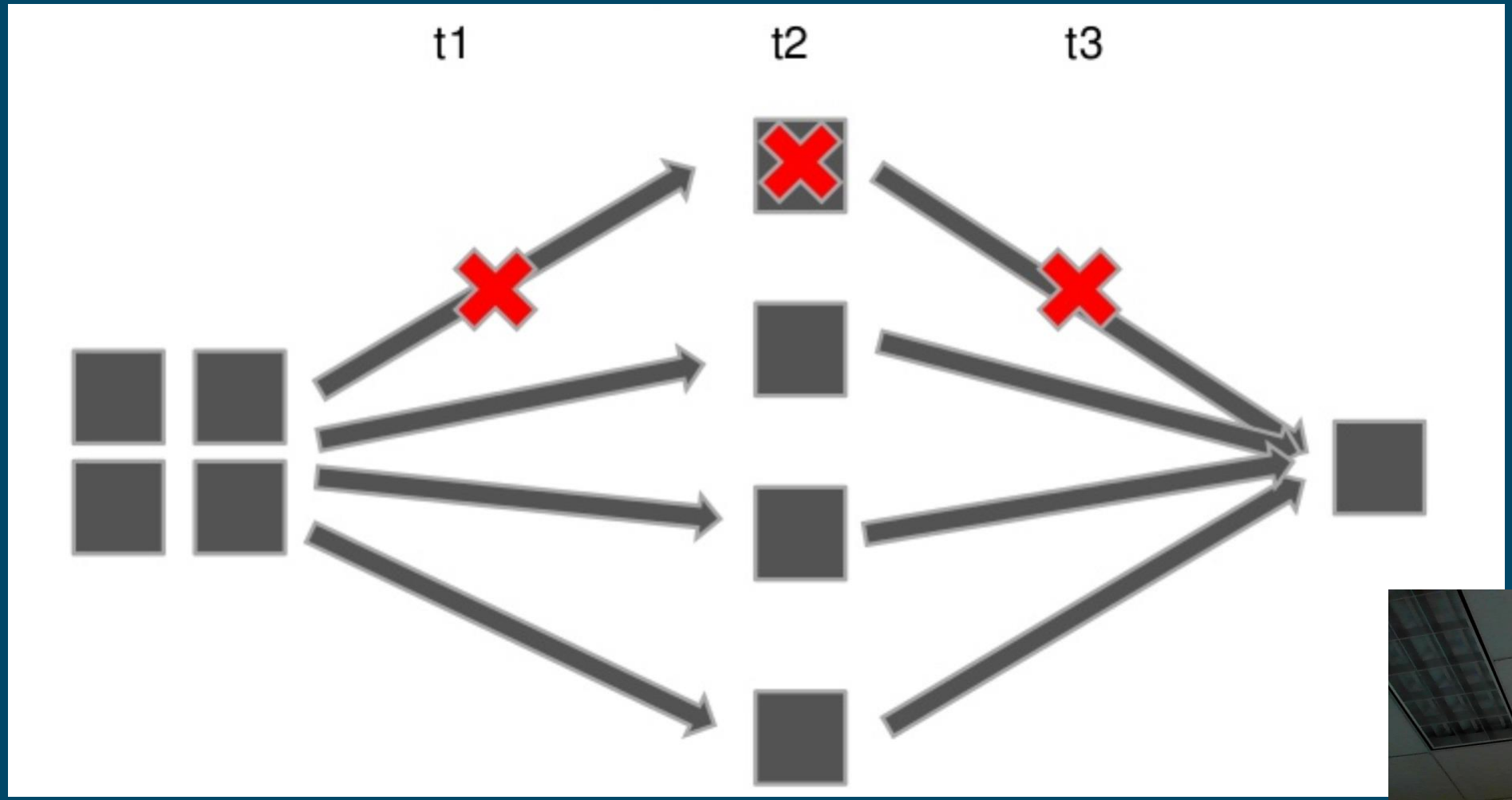


Data processed in parallel



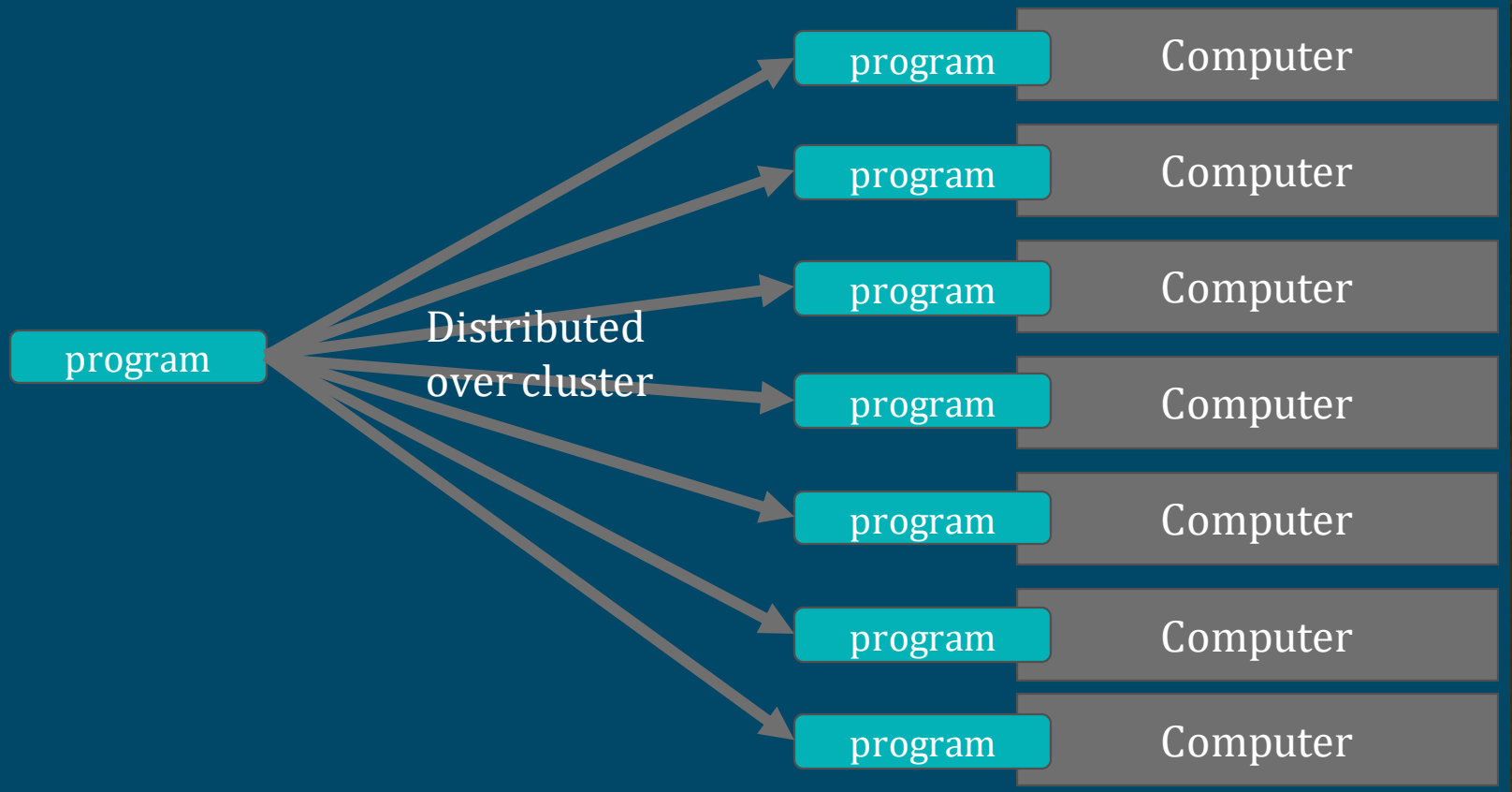


Data processing can fail!



Parallel processing over a cluster

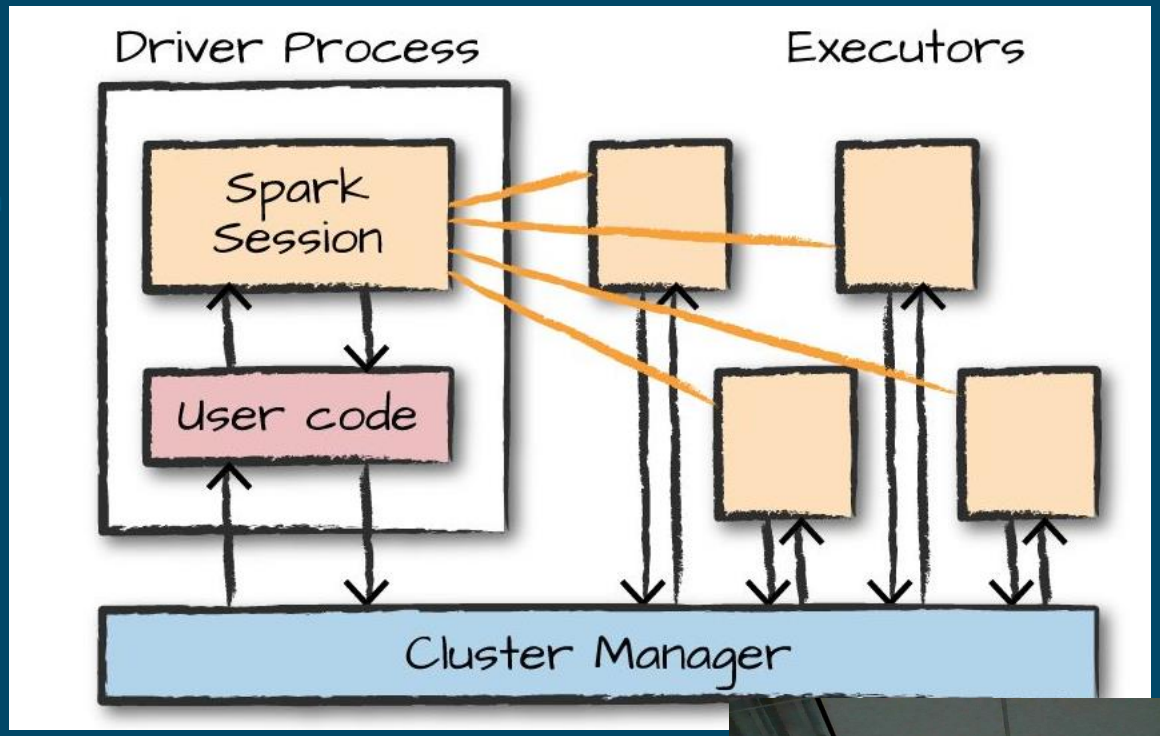
Spark copies a program to nodes in the cluster, for parallel execution



Task execution in Spark

Driver sends code, via manager, to cluster which contains the data

- Cluster manager tracks resources
- The driver process is responsible for executing the driver program's tasks on executors to complete a given job
  - Your program (job) is executed by the driver process
- Executor is a (Spark) process in a computer that executes Spark tasks



Read a file, filter by ‘data”

```
1  lines = sc.textFile("dbfs:/databricks-datasets/README.md")
2  lines.collect()
```

▸ (1) Spark Jobs

```
['Databricks Hosted Datasets',
 '==========================',
 '',
 'The data contained within this directory is hosted for users to build ',
 'data pipelines using Apache Spark and Databricks.'
```

```
1  data = lines.filter(lambda s: s.startswith('data'))
2  messages = data.map(lambda s: s.split(' '))
3  messages.cache()
4  messages.collect()
```

▸ (1) Spark Jobs

```
[['data', 'pipelines', 'using', 'Apache', 'Spark', 'and', 'Databricks.'],
 ['data',
  'allows',
```

Example: Mining Console Logs

- Load error messages from a log into memory, then interactively search for patterns

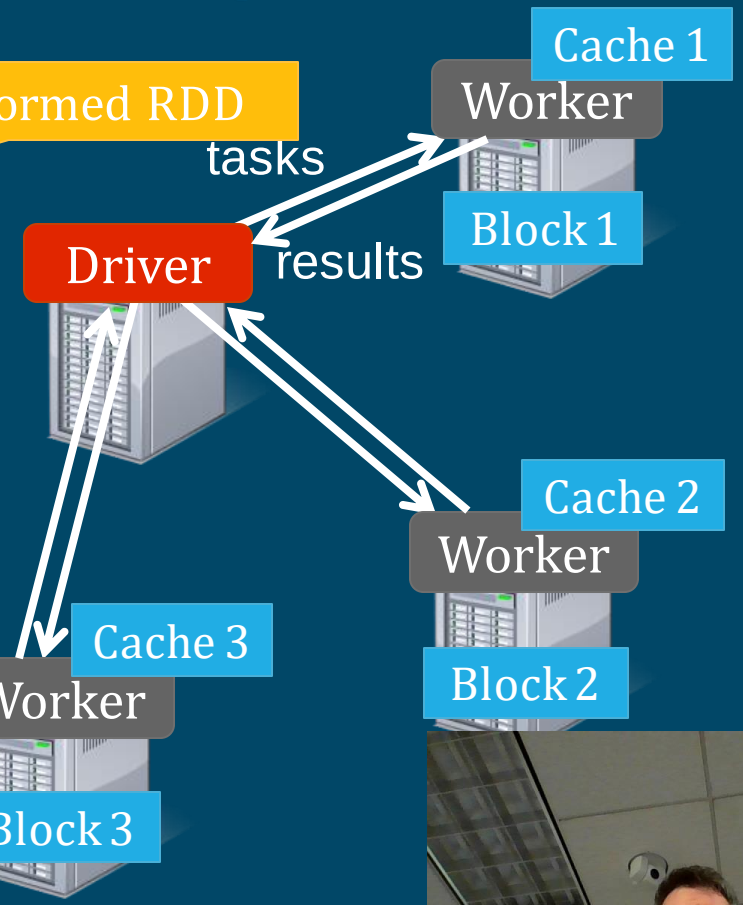
```
lines = spark.textFile(“hdfs://...”)
errors = lines.filter(lambda s: s.startswith(“ERROR”))
messages = errors.map(lambda s: s.split(‘\t’)[2])
messages.cache()

messages.filter(lambda s: “foo” in s).count()
messages.filter(lambda s: “bar” in s).count()
. . .
```

Base RDD

Transformed RDD

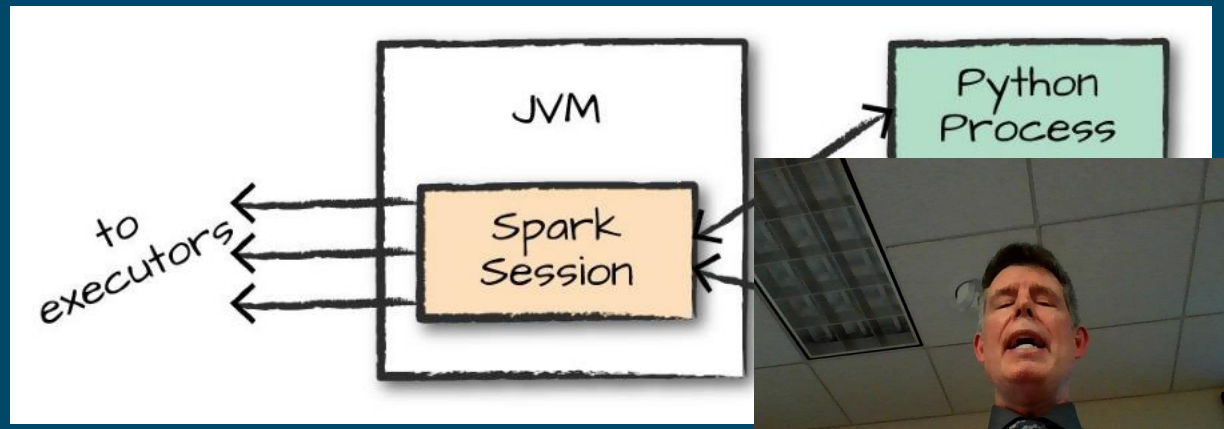
Action



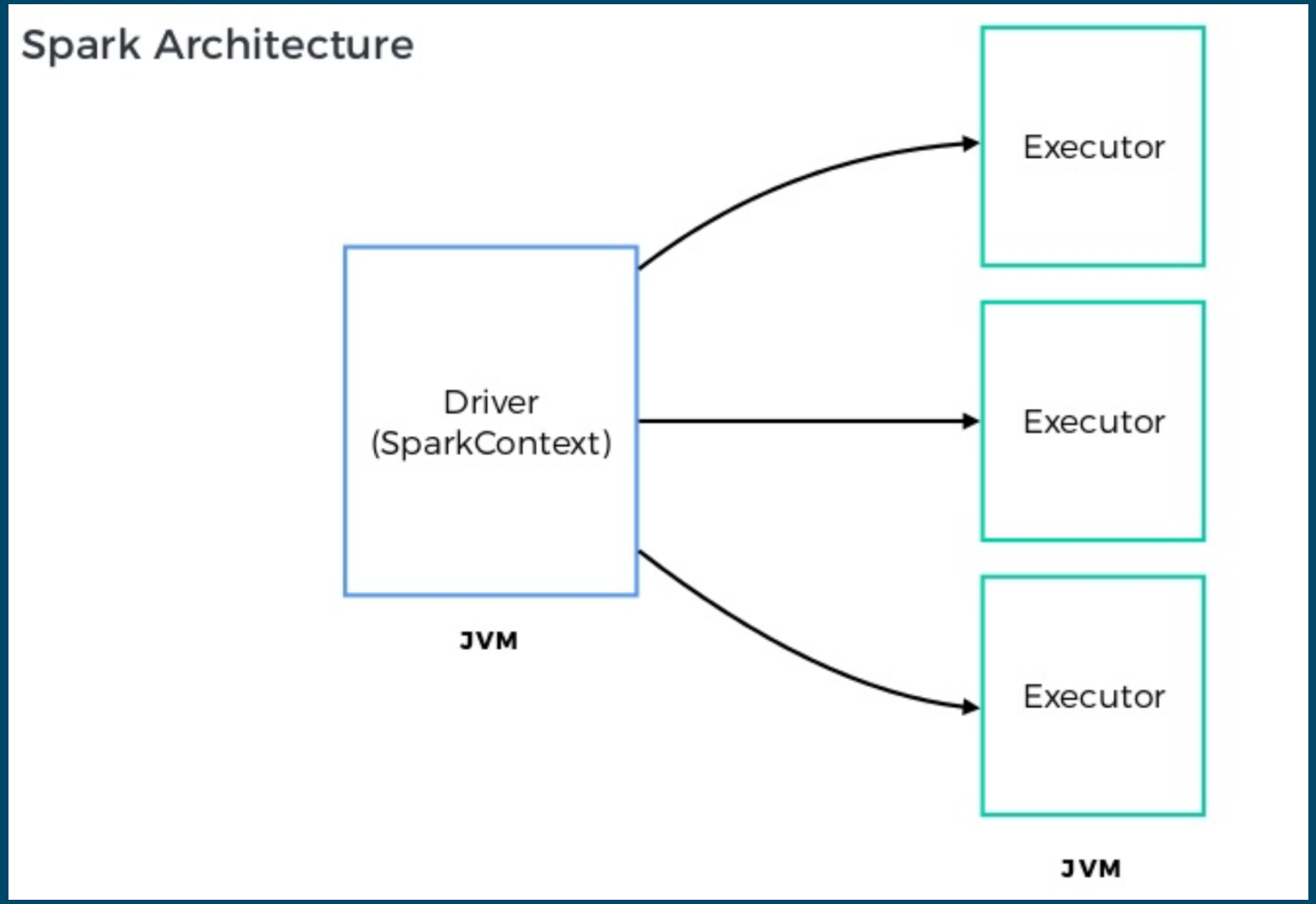
Result: scaled to 1 TB data in 5-7 sec
(vs 170 sec for on-disk data)

Task executors run in JVMs

- Spark runs as JVM
  - Languages translate to JVM or interface with it
- Native Spark code runs faster
  - Scala and Java run native in JVM
  - PySpark can impose extra overhead



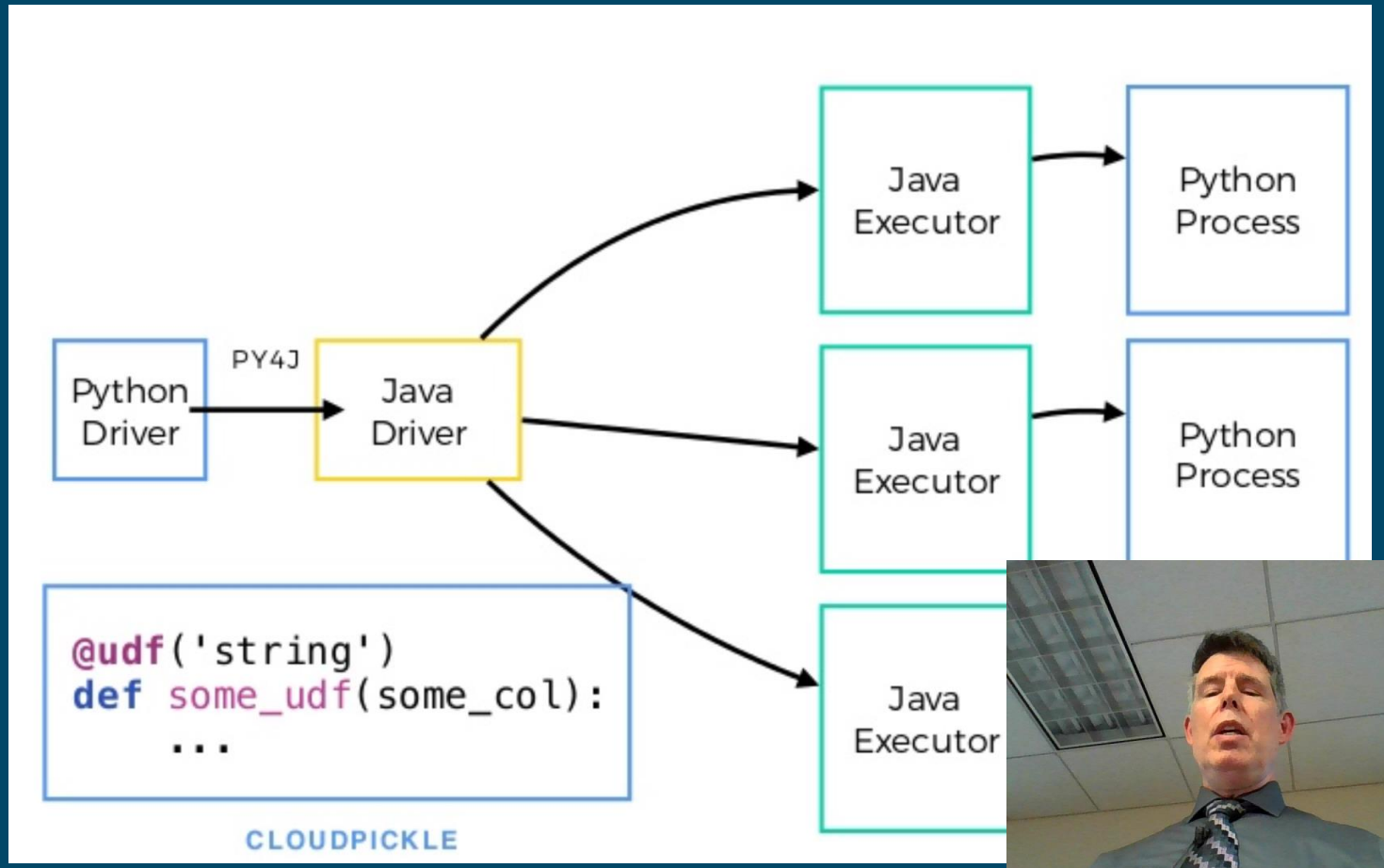
Standard Spark



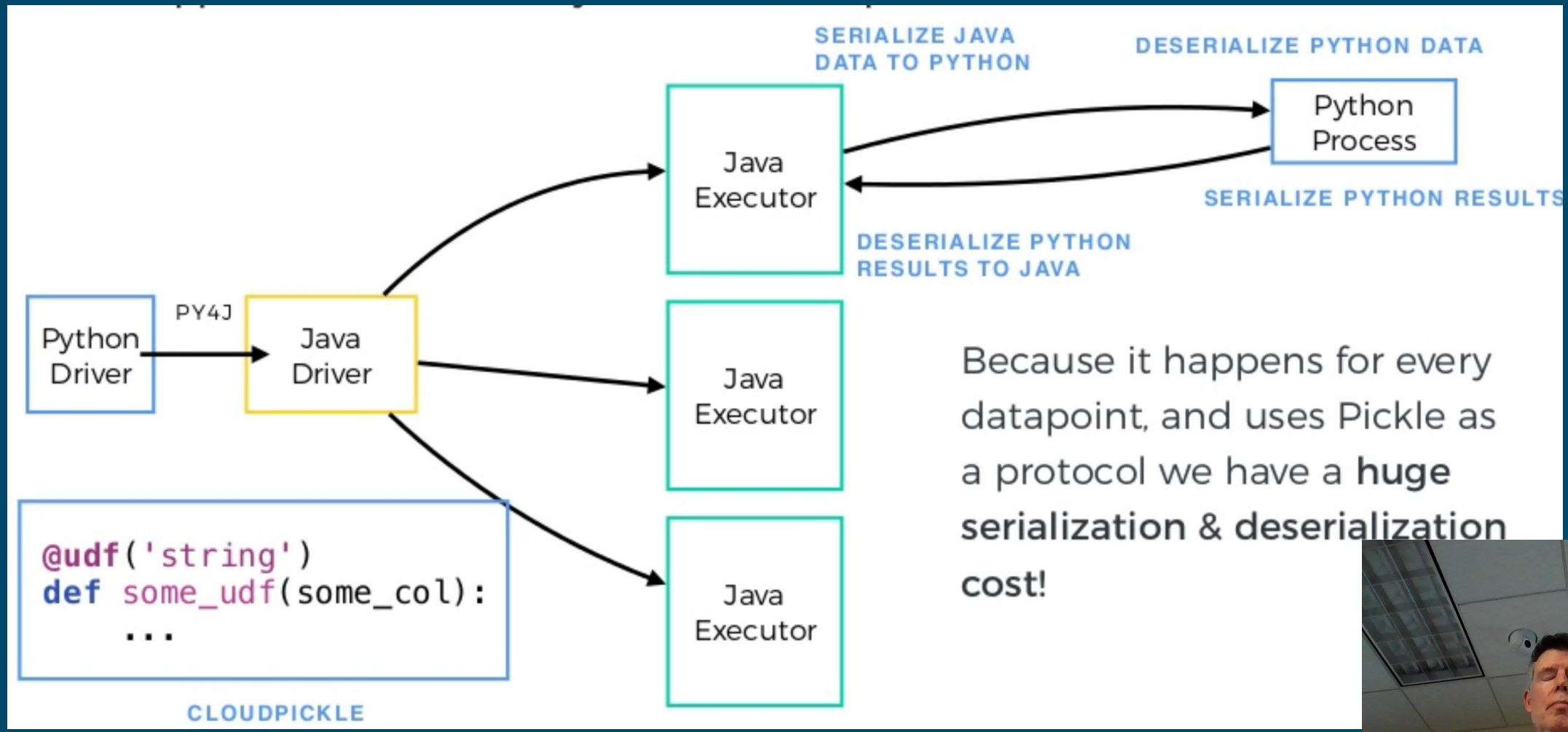
https://www.slideshare.net/mateuszbuskiewicz/dive-into-pyspark

PySpark is a thin wrapper over JVM

- PySpark has associated JVM context (with Java objects)
  - UDFs have overhead, but Spark API accessed from Python is efficient



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

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

▸ (1) Spark Jobs

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

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

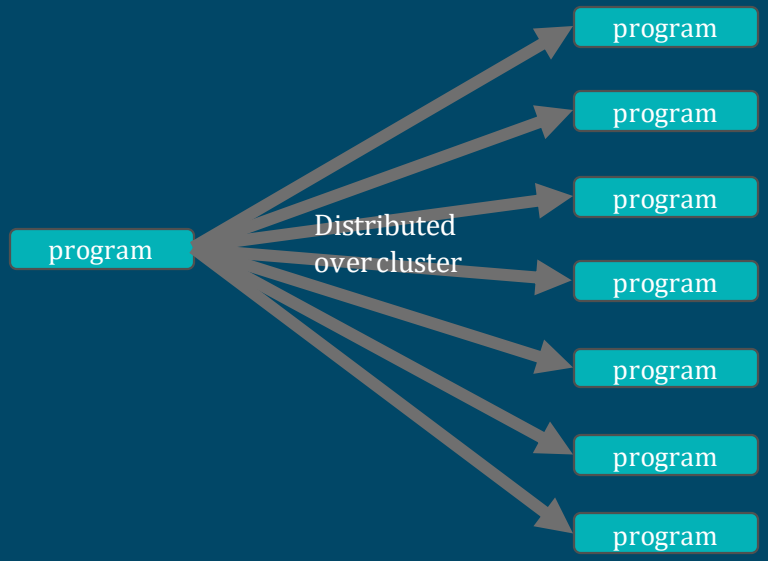
UDF split (mySplitUDF) is slow

```
1  from pyspark.sql.functions import udf,col
2
3  def splitUDF(col):
4    if col is not None:
5      return col.split(' ')
6
7  mySplitUDF = udf(splitUDF)
8
9  df2 = df.select(mySplitUDF(col('Description')))
10 display(df2)
```

▸ (1) Spark Jobs

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

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



Spark creates multiple tasks from your PySpark program

By default, Spark creates t

Programmer can specify how many t

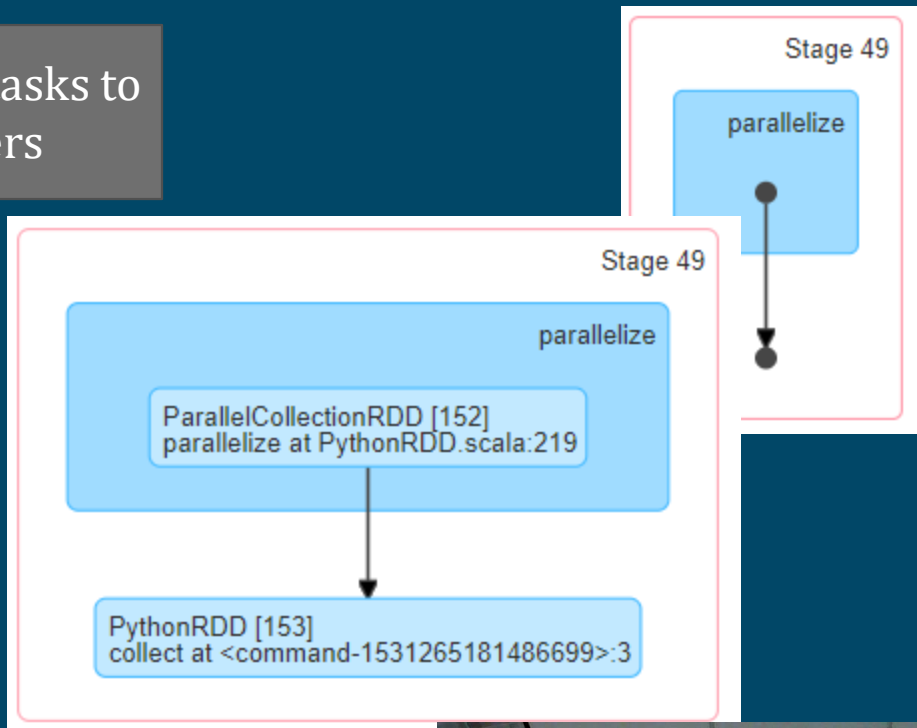
2 partitions -> 2 tasks

```
1  data = range(1,100)
2  rdd = sc.parallelize(data,2)
3  rdd.map(lambda x: x * x).collect()
4
```

▾ (1) Spark Jobs

▸ Job 36 View (Stages: 1/1)

Spark sends tasks to computers



Tasks (2)

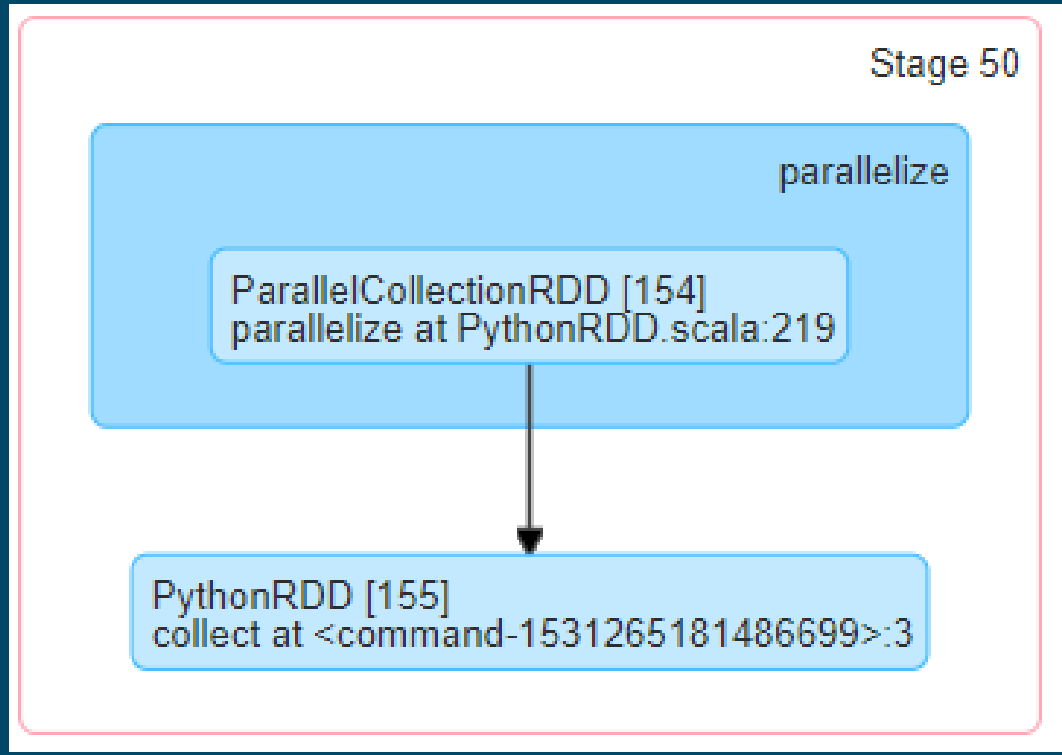
Index ▲	ID	Attempt	Status	Locality Level	Executor ID	Host	Laun
0	206	0	SUCCESS	PROCESS_LOCAL	driver	localhost	2018
1	207	0	SUCCESS	PROCESS_LOCAL	driver	localhost	2018

5 partitions -> 5 tasks

```
1  data = range(1,100)
2  rdd = sc.parallelize(data,5)
3  rdd.map(lambda x: x * x).collect()
4
```

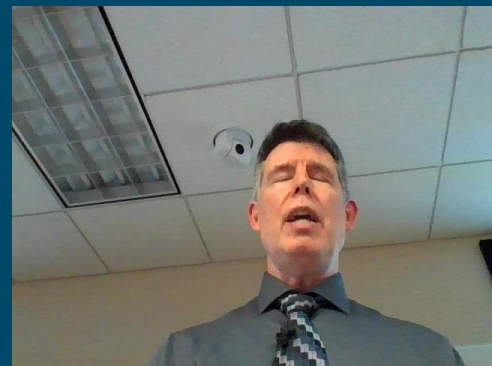
▼ (1) Spark Jobs

▶ Job 37 View (Stages: 1/1)



Tasks (5)

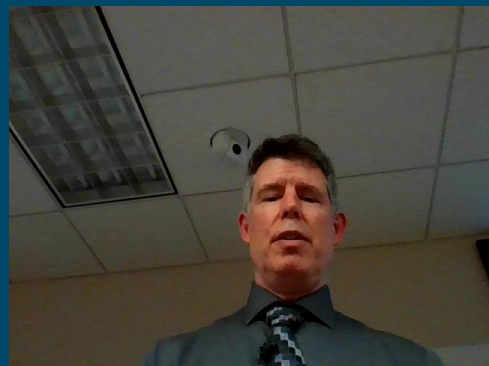
Index ▲	ID	Attempt	Status	Locality Level	Executor ID	Host	Launch Time	Duration
0	208	0	SUCCESS	PROCESS_LOCAL	driver	localhost	2018/11/02 16:58:40	0.2 s
1	209	0	SUCCESS	PROCESS_LOCAL	driver	localhost	2018/11/02 16:58:40	0.2 s
2	210	0	SUCCESS	PROCESS_LOCAL	driver	localhost	2018/11/02 16:58:40	0.3 s
3	211	0	SUCCESS	PROCESS_LOCAL	driver	localhost	2018/11/02 16:58:40	0.2 s
4	212	0	SUCCESS	PROCESS_LOCAL	driver	localhost	2018/11/02 16:58:40	0.2 s



PySpark can run fast…

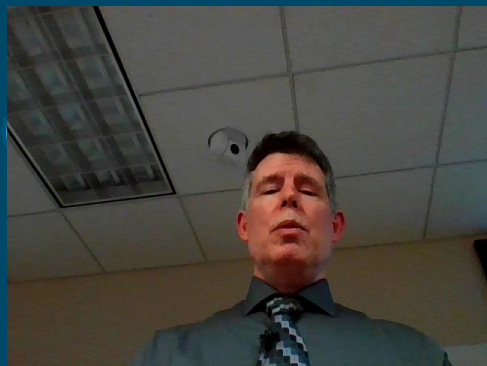
- Avoid Python UDFs
- Use PySpark as scripting access to built-in Spark functions (APIs)
- Some improvements coming with newer version of Spark, e.g., Arrow API

Processing data with Spark



Spark processing

1. Data is copied onto the cluster
   - A table is split into partitions (e.g., by rows)
   - The partitions are stored on disks of clustered computers
   - Partitions are duplicated over multiple computers
2. After the data has been stored
3. A program is run
   - Copies are sent to the clustered computers
   - Each copy executes tasks on a computer, which has a local data

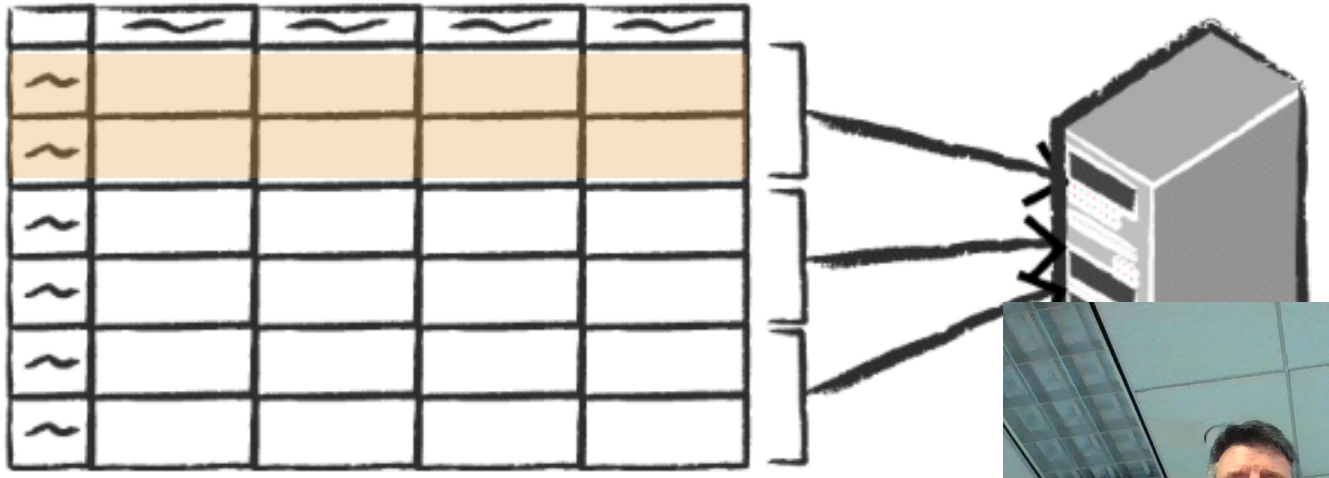


Tabular data is distributed over cluster

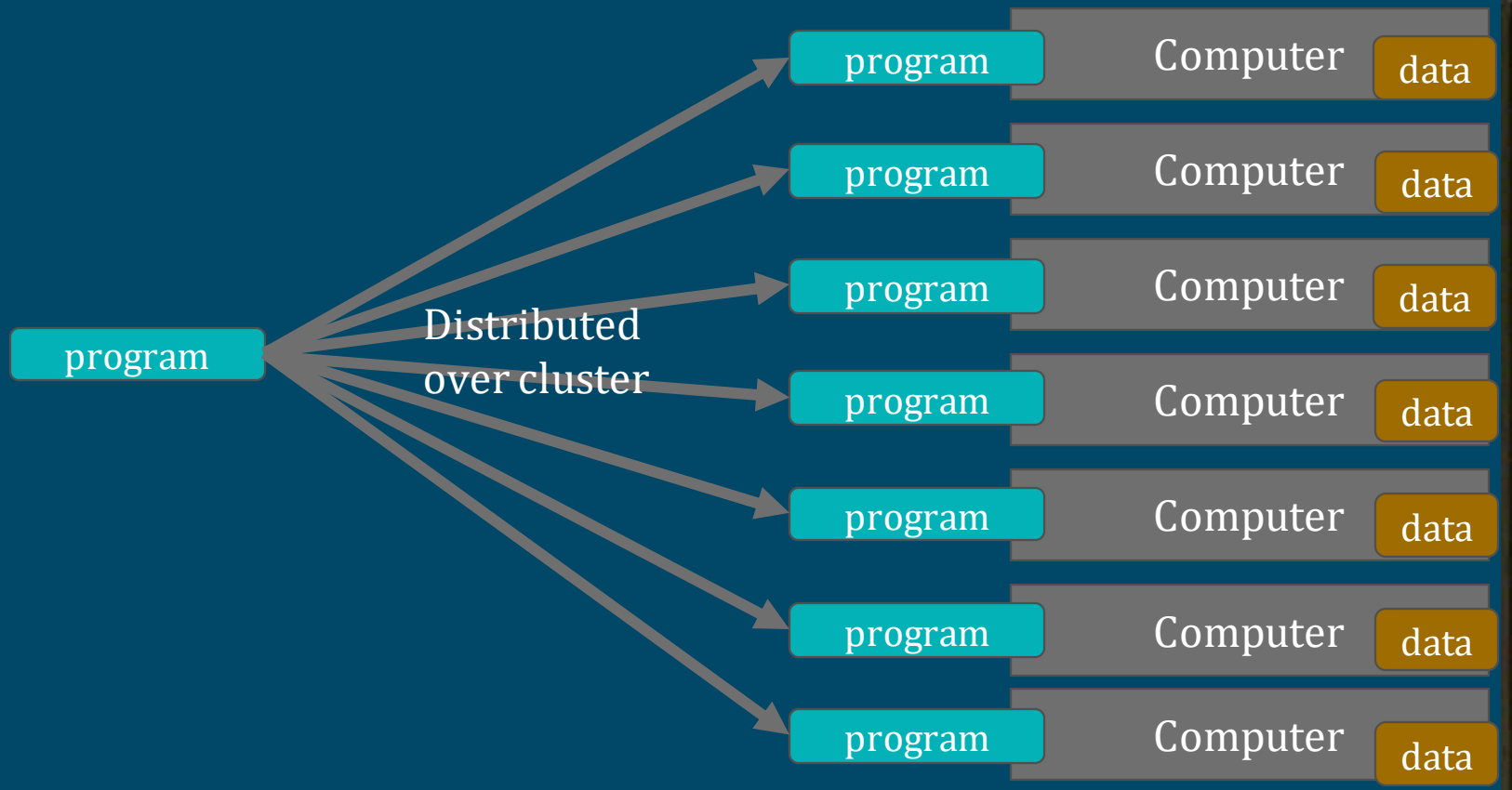
row subsets are partitions

Spreadsheet on a single machine

Table or Data Frame partitioned across servers in a data center



Spark copies a program to nodes in the cluster, for parallel execution



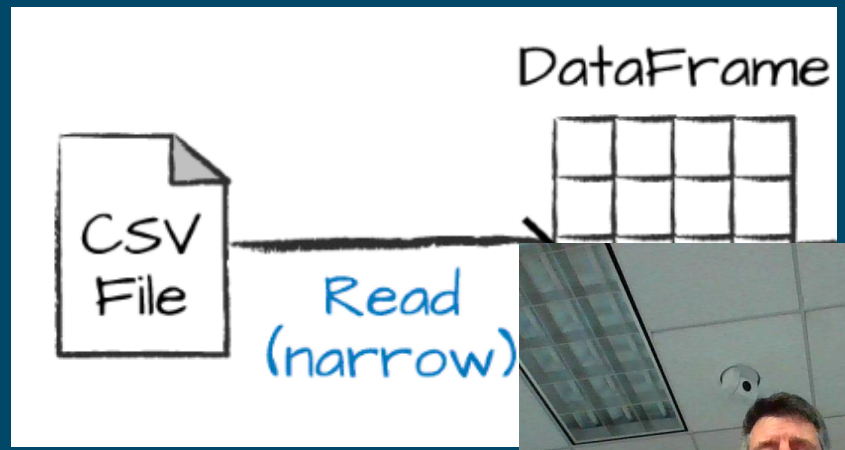
Example: read a file into a DataFrame (table)

- PySpark

```
flightData2015 = spark.read.csv("2015-summary.csv")
```

- Explanation

  - Program (e.g., Spark notebook) copied to a computer that has the data
  - Run program on data

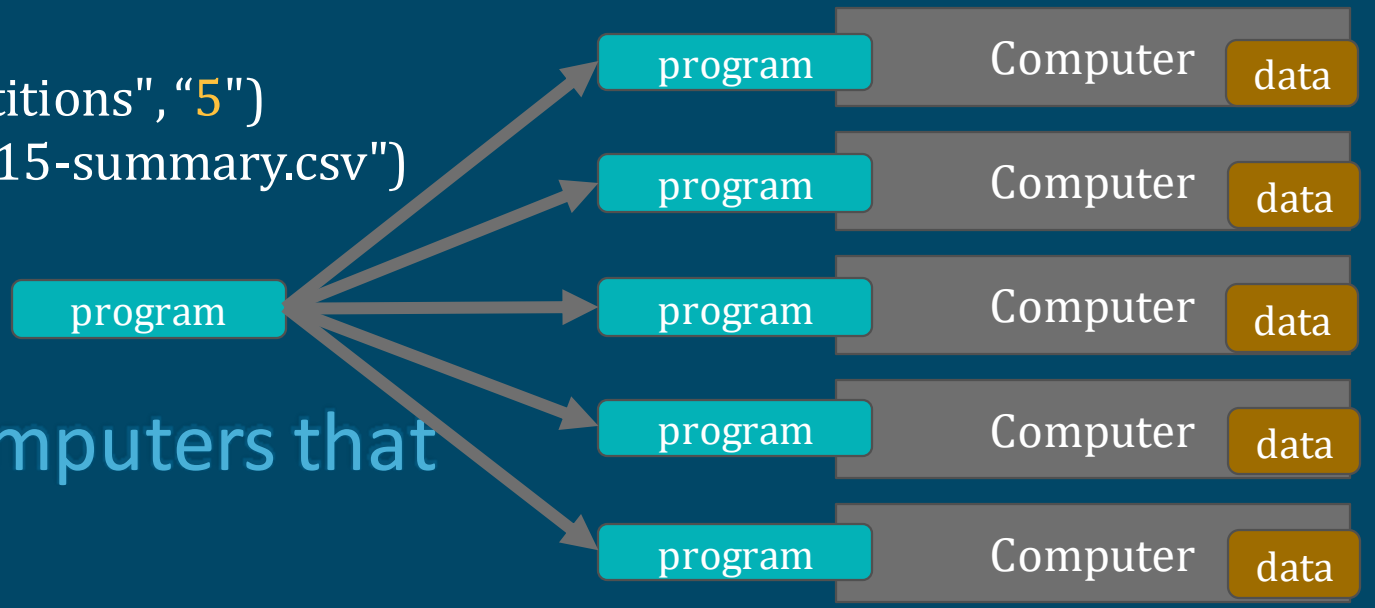


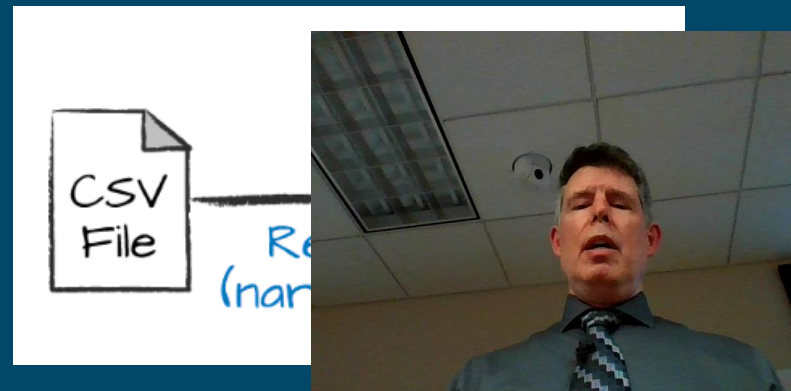
Example: read a file into a DataFrame (table), which is partitioned over cluster

- PySpark

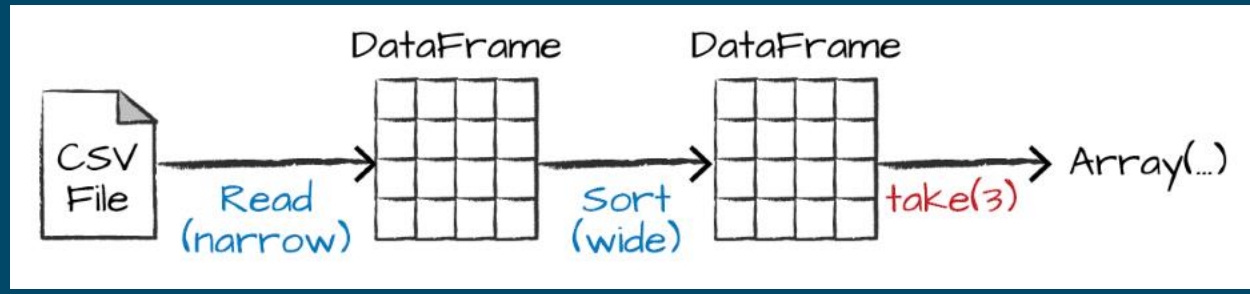
```
spark.conf.set("spark.sql.shuffle.partitions", "5")
flightData2015 = spark.read.csv("2015-summary.csv")
```

- Explanation
  - Program copied to computers that have the data
  - Run program on computers each accessing local data partitions





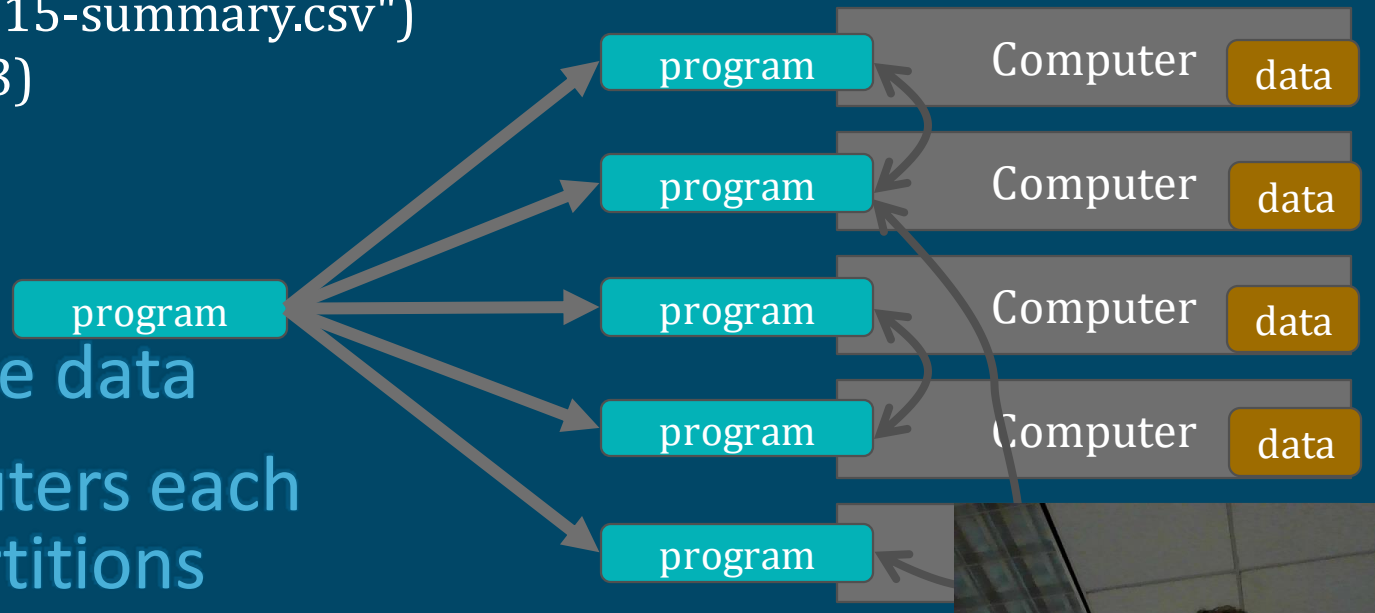
Example: read & sort data



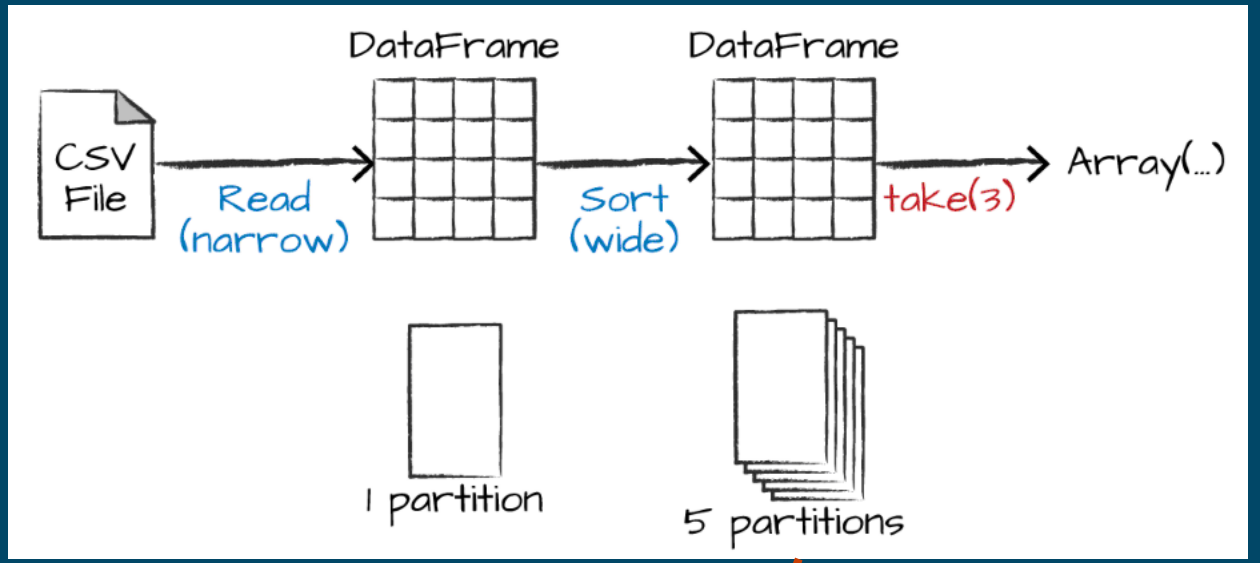
- PySpark

```
spark.conf.set("spark.sql.shuffle.partitions", "5")
flightData2015 = spark.read.csv("2015-summary.csv")
flightData2015.sort("count").take(3)
```

- Explanation
  - Program copied to computers that have the data
  - Run program on computers each accessing local data partitions
  - Programs exchange results to sort them



Program evaluation



- PySpark

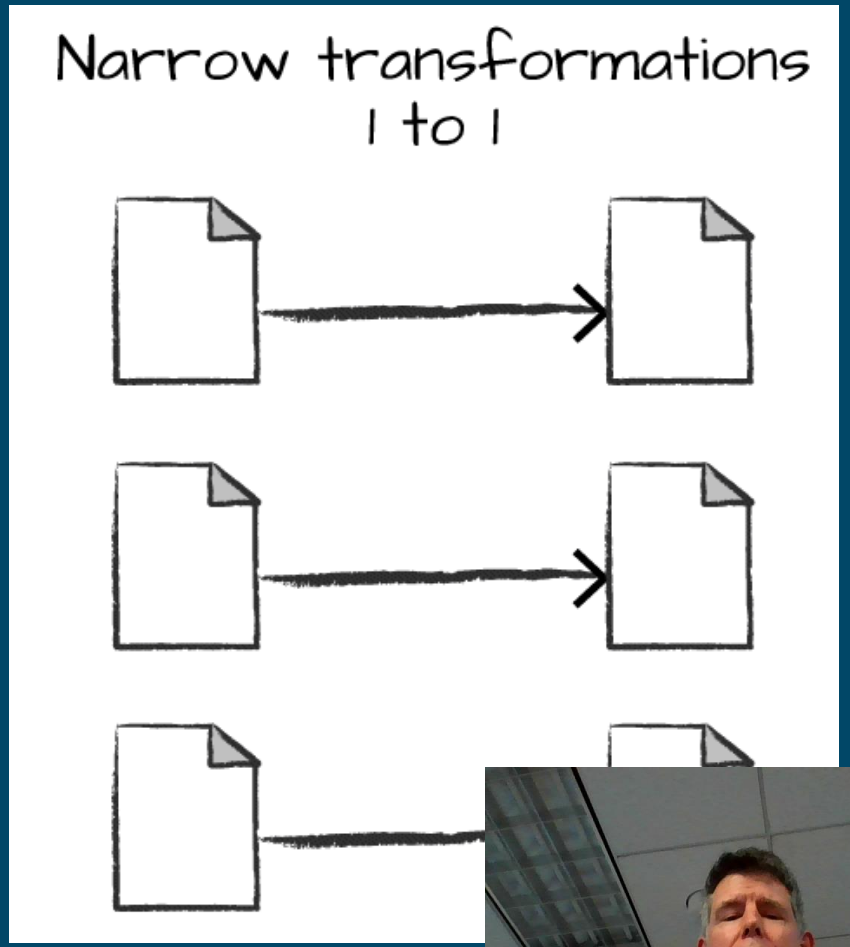
```
spark.conf.set("spark.sql.shuffle.partitions", "5")
flightData2015 = spark.read.csv("2015-summary.csv")
flightData2015. sort("count"). take(3)
```

- Explained

```
flightData2015. sort(" count"). explain()
= = Physical Plan = =
*Sort [count# 195 ASC NULLS FIRST], true, 0
 +- Exchange rangepartitioning( count# 195 ASC NULLS FIRST, 200)
  +- *FileScan csv [DEST_COUNTRY_NAME# 193, ORIGIN_COUNTRY_NAME# 194, c
```

Partitioned data is narrow transformed (e.g., filter)

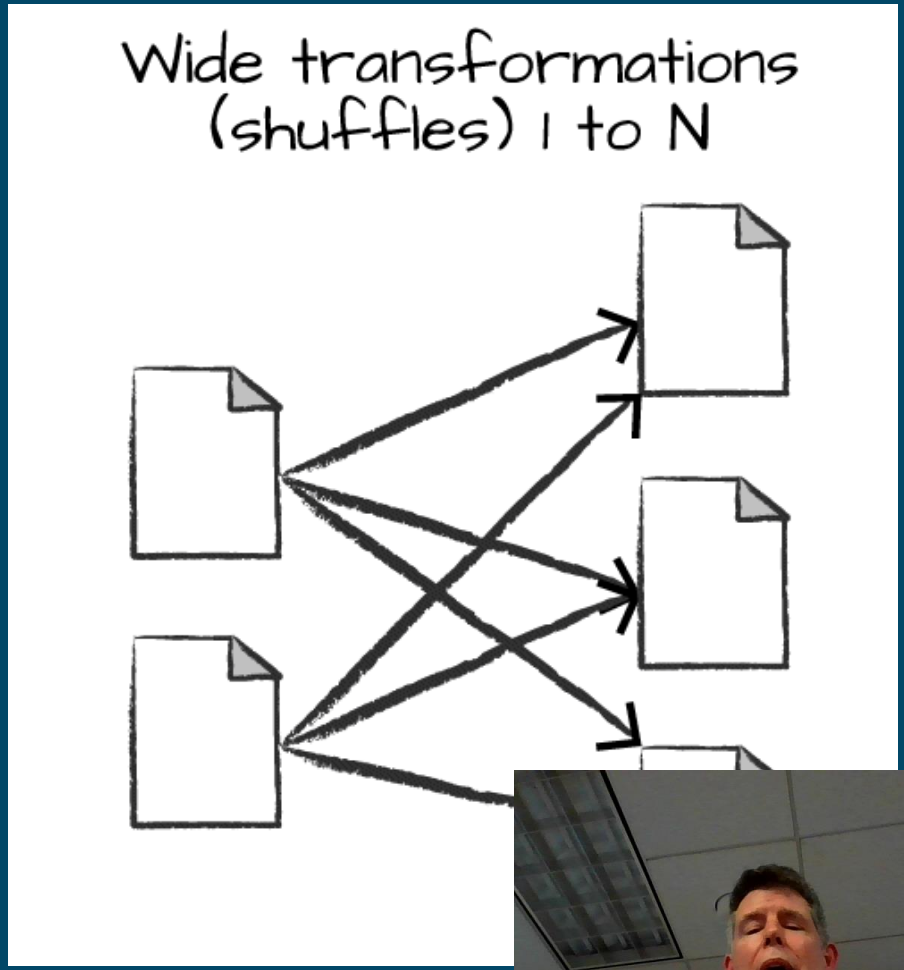
- Transformations consisting of narrow dependencies (narrow transformations) are those for which each input partition will contribute to only one output partition.




```
divisBy2 = myRange.where("number % 2 = 0")
```

Partitioned data is wide transformed (e.g., sort)

- A wide dependency (or wide transformation) transformation will have input partitions contributing to many output partitions.
  - This is a shuffle where Spark will exchange partitions across the cluster.
  - Data partitions are combined (often through sorting) to form a new combined partition.




```
data. sort("count")
```

Why do transformations matter?

- Narrow outputs to one executor
  - Efficient transfer of data
- Wide outputs to multiple executors
  - Less efficient, as more data must travel over network
- Programming advice
  - Reduce the use of wide transformations
  - Reduce data (filter) before applying wide transfor

Spark does lazy evaluation

Waiting until it must produce output allows the interpreter to optimize the execu

Example of lazy evaluation

- PySpark

```
spark.conf.set("spark.sql.shuffle.partitions", "5")
flightData2015 = spark.read.csv("2015-summary.csv")
flightData2015.sort("count").take(3)
```

Spark interpreter

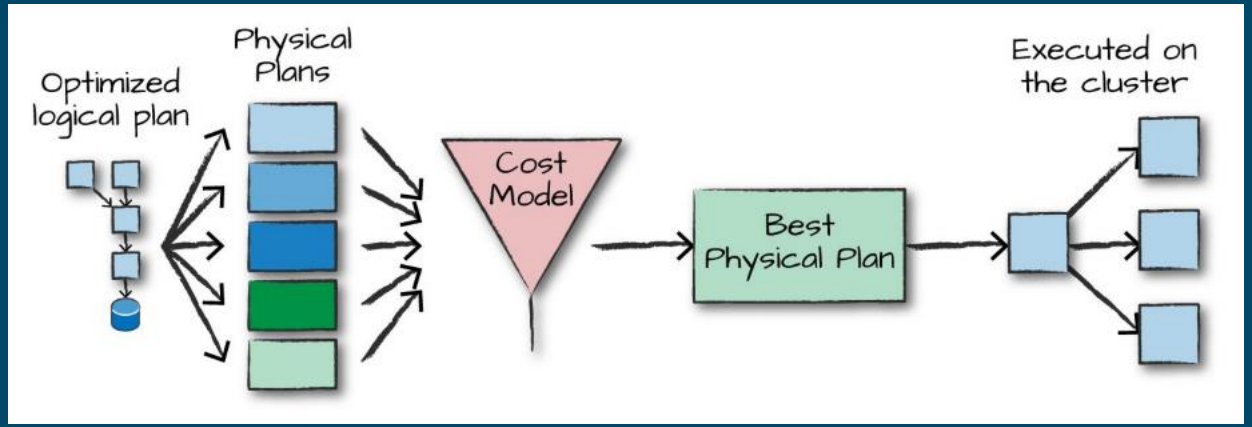
→ record step
→ record step
→ execute with take

- No execution occurs until an action is called
  - Produces output
  - Demands data from cluster to calculate

Actions

reduce()
collect()
count()
first()
take(num)
takeSample()
takeOrdered()
saveAsTextFile()
saveAsSequenceFile()
saveAsObjectFile()
countByValue()
countByKey()
foreach()
…

Lazy evaluation



- Spark waits until the last moment to execute the graph of instructions
- Transformations create a logical transformation plan
- Different languages (Scala, PySpark) can have the same plan, which compiles to the same code (mostly)
- An action triggers the computation.
  - Spark then compute a result from the series of transformations

Spark is a collection of APIs

Written in Scala, compiled to JVM

Multiple languages supported, each translated to

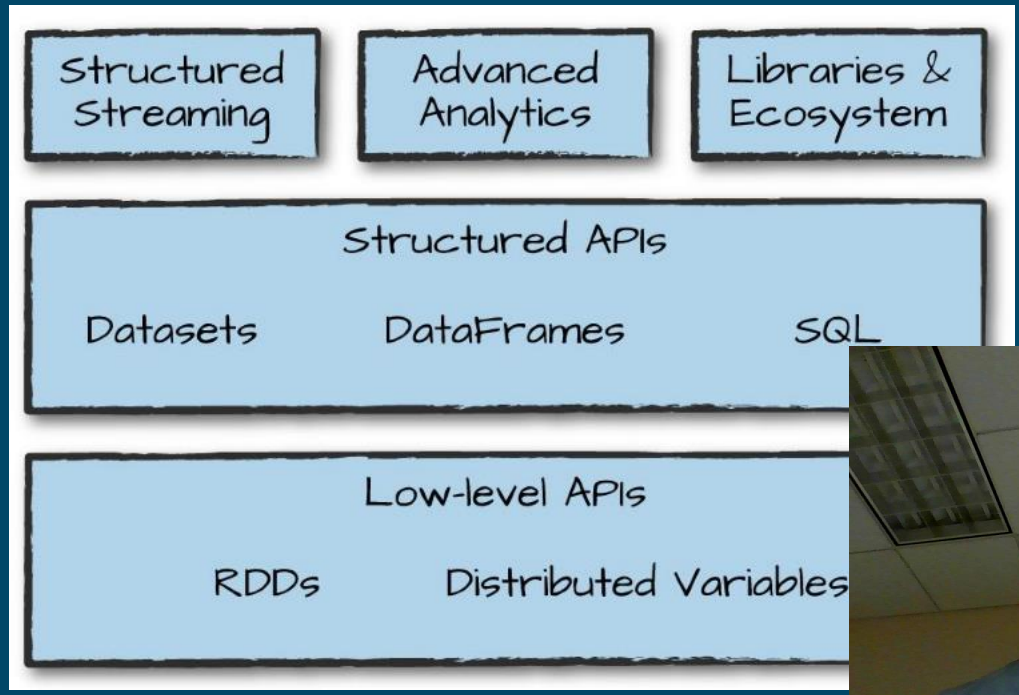
Spark as a set of libraries (APIs)

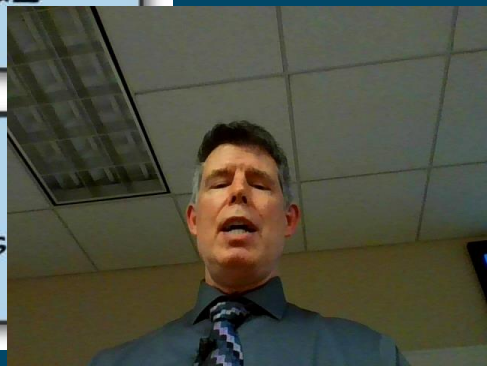
- An **application programming interface** (**API**) is an interface or communication protocol between different parts of a program that simplify the implementation and maintenance of software

Ingestion and analysis of data

Structured data over cluster

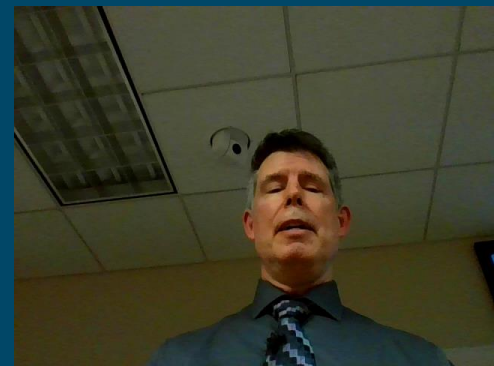
Data over cluster





Unified computing engine

- Spark is designed to support a wide range of data analytics tasks,
  - ranging from simple data loading and SQL queries to machine learning and streaming computation,
  - over the same computing engine and with a consistent set of APIs.
- Example
  - If you load data using a SQL query and then evaluate a machine learning model over it using Spark's ML library, the engine can combine these steps into one scan over the data.
  - Most everything is processed as a table (Data Frame)



History of Spark APIs

RDD (2011)

Distribute collection of JVM objects

Functional Operators (map, filter, etc.)

DataFrame (2013)

Distribute collection of Row objects

Expression-based operations and UDFs

Logical plans and optimizer

Fast/efficient internal representations

DataSet (2015)

Internally rows, externally JVM objects

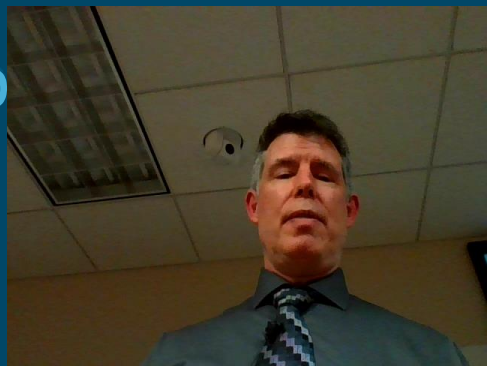
Almost the "Best of both worlds": **type safe + fast**

But
Not as
analysi

databricks

DataFrames

- DataFrame
  - represents a table of data with rows and columns
  - is backed by an RDD
  - Use the Structured (DataFrame) APIs when possible
- Partition
  - is a collection of rows that are on one physical machine in a cluster
  - partitions represent how the data is physically distrib across the cluster of machines during execution



Resilient Distributed Datasets (RDDs)

- DataFrame operations are built on top of RDDs
- RDD allow precise control of data
  - E.g., parallelize raw data stored in memory on the driver machine.

```
# in Python
from pyspark.sql import Row

spark.sparkContext.parallelize([Row(1), Row(2), Row(3)]).toD
```

Streaming & Analytics

- Structured Streaming
  - Run the same operations that you perform in batch mode in a streaming fashion (i.e., as data arrives)
    - Reduced latency and allow for incremental processing
- MLlib
  - allows for preprocessing, munging, training of models and making predictions at scale on data

Apache Spark

- is a unified computing engine
  and a set of libraries
  for parallel data processing
  on computer clusters

Important to remember

- Spark supports parallel data processing on clusters
  - It manages the resources and routing of code & data
- Data is stored on the cluster prior to processing
- A program is copied to cluster computers
- Partitioned data is processed by tasks
  - Tasks run in JVM
  - PySpark code runs in a separate Python process
- Two kinds of transformations
  - Narrow outputs to one executor
  - Wide outputs to multiple executors
- Spark does lazy evaluation