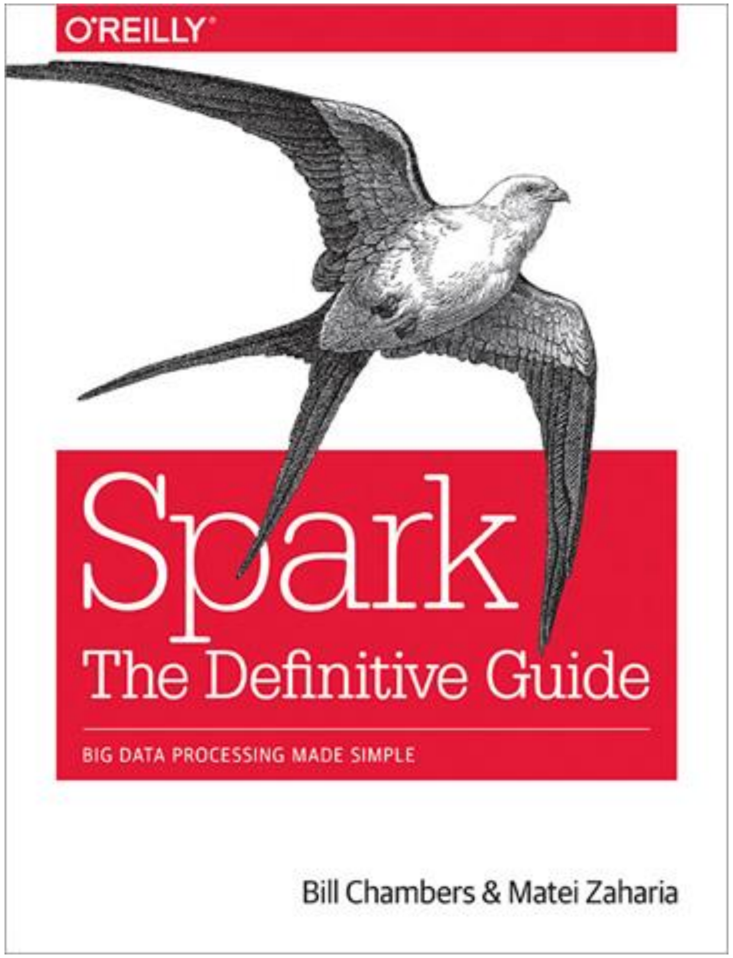
Cloud Computing

Apache Spark

Dell Zhang
Birkbeck, University of London
2018/19

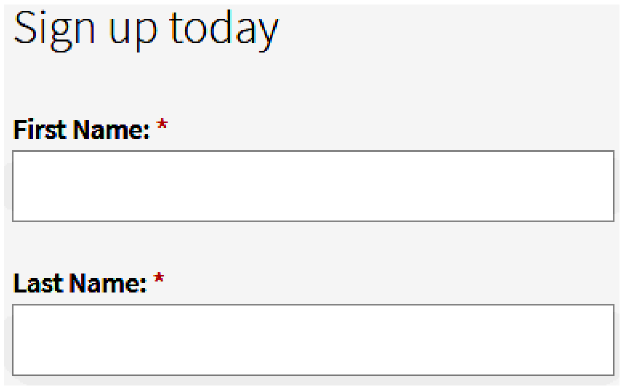
Spark: The Definitive Guide


O'REILLY
Spark
The Definitive Guide
BIG DATA PROCESSING MADE SIMPLE
Bill Chambers & Matei Zaharia

https://github.com/databricks/Spark-The-Definitive-Guide

The Apache Spark™ Collection

A Comprehensive Preview of the Definitive Guide to Spark


Sign up today

First Name: *

Last Name: *

https://pages.databricks.com/the-apache-spark-collection.html

What is Spark?

- Apache Spark is a *unified computing engine* and a set of *libraries* for parallel data processing on computer clusters.
  - The most actively developed *open source* engine for this task.
  - The de facto tool for any developer or data scientist interested in *big data*.

One popular answer to "What's beyond MapReduce?"

Spark’s Philosophy

- **Unified**: Spark is designed to support a wide range of data analytics tasks over the same computing engine and with a consistent set of APIs
  - Provides consistent, composable APIs.
  - Enables high performance by optimizing across the different libraries and functions composed together in a user program.

Spark’s Philosophy

- **Computing Engine**: Spark only handles loading data from storage systems and performing computation on it, not permanent storage as the end itself.
  - Focuses on performing computations over the data, no matter where it resides.
  - Makes it different from earlier big data software platforms such as Apache Hadoop.

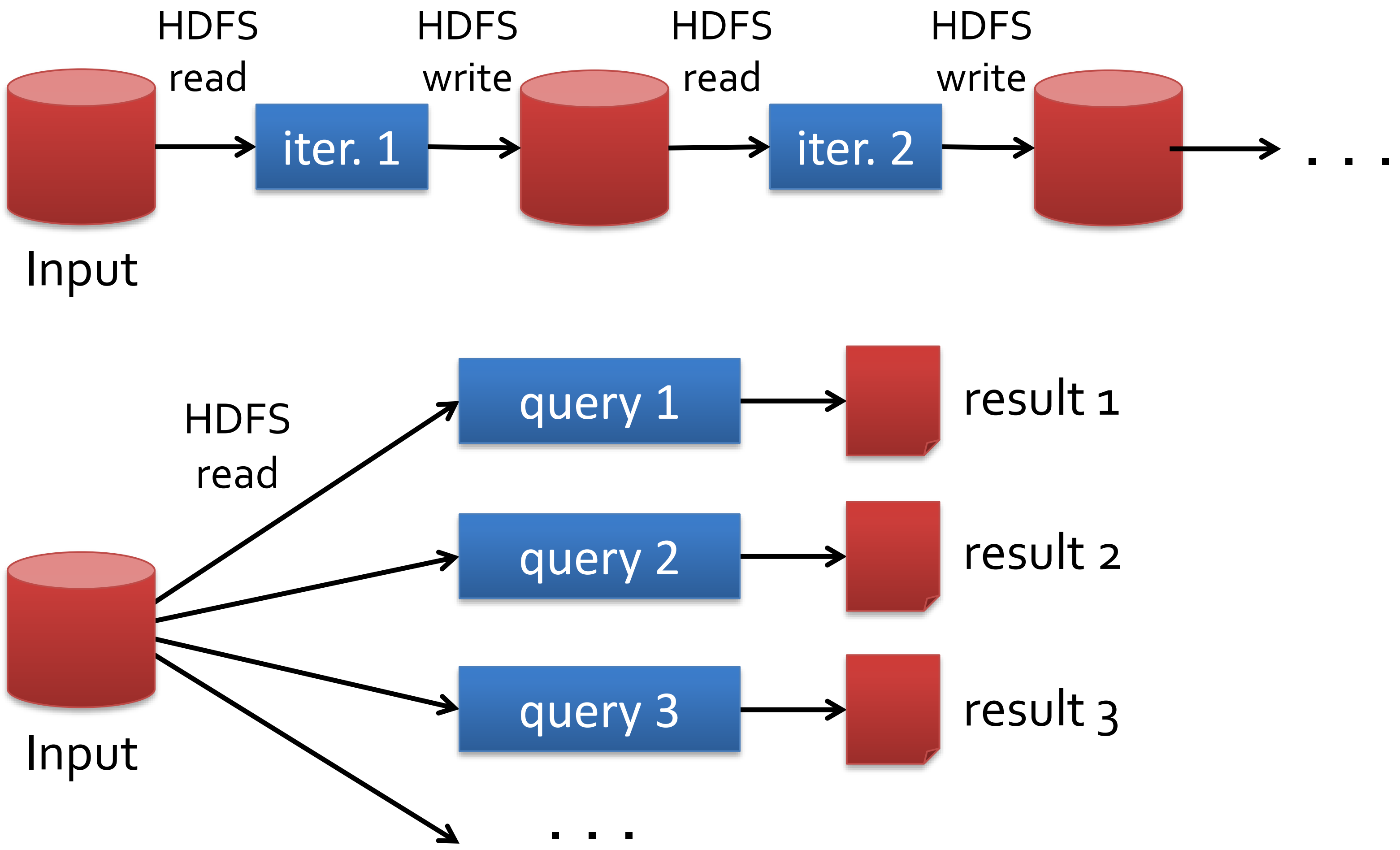
Spark's Philosophy

- **Libraries**: Spark supports both *standard* libraries that ship with the engine, and a wide array of *external* libraries published as third-party packages by the open source communities.
  - Spark SQL, Structured Streaming, GraphFrames
  - MLlib, Deep Learning Pipeline etc.

Spark's Motivation

- MapReduce greatly simplified "big data" analysis on large, unreliable clusters, but as soon as it got popular, users wanted more:
  - More **complex**, multi-stage applications (<u>iterative</u> machine learning & graph processing etc.)
  - More **interactive** ad-hoc queries
- Such complex apps and interactive queries both need one thing that MapReduce lacks: *in-memory data sharing*.

Spark's Motivation

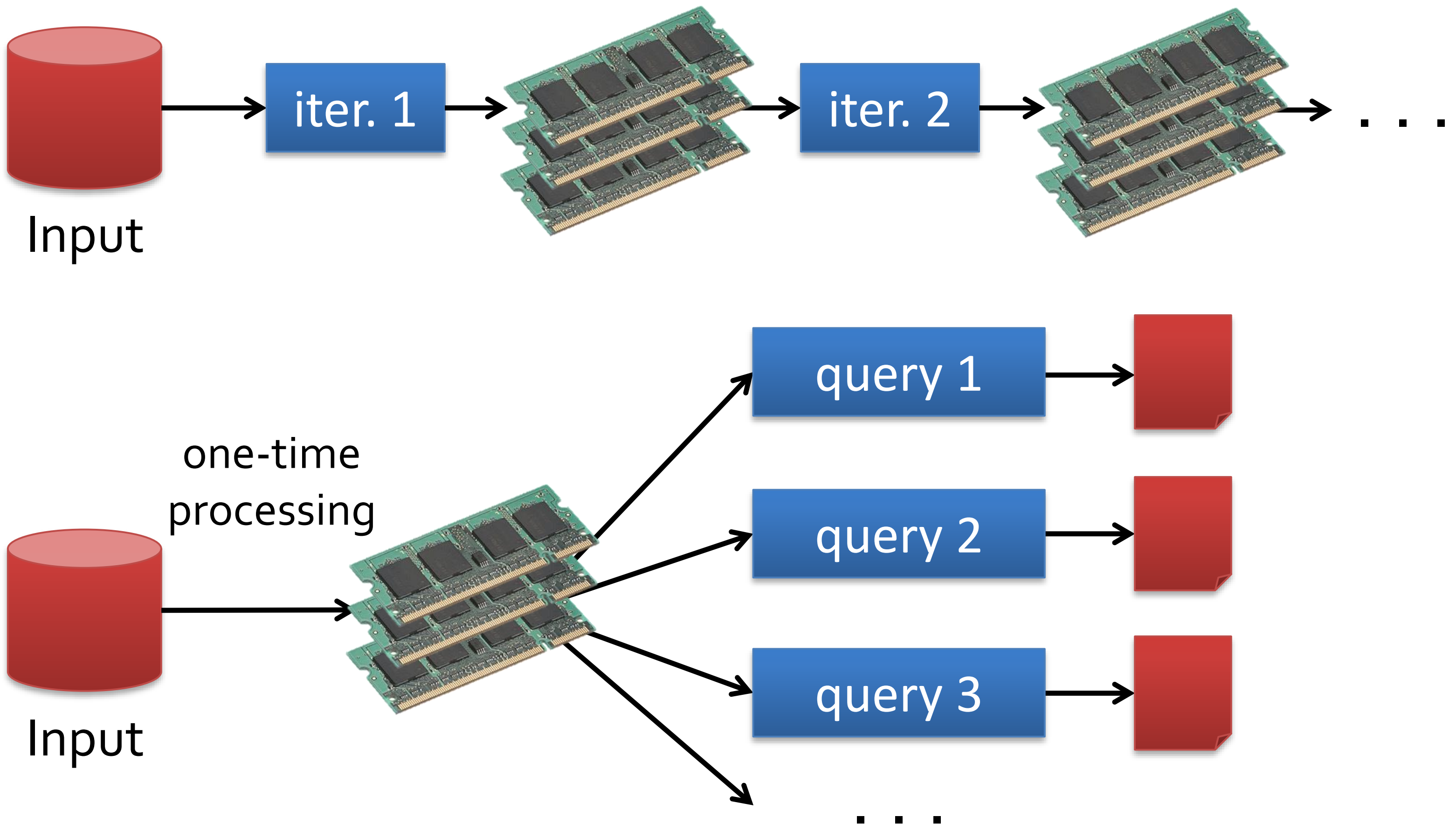


Slow due to replication and disk I/O, but necessary for fault tolerance

Spark’s Motivation

- In MapReduce, the only way to share data across jobs is using stable storage, which is ***slow***.
  - For example, the typical machine learning algorithm might need to make 10 or 20 passes over the data, and in MapReduce, each pass had to be written as a separate MapReduce job, which had to be launched separately on the cluster and load the data from scratch.

Spark’s Motivation



10-100× faster than network/disk, but how to get fault tolerance?

Spark's History

- Developed at UC Berkeley AMPLab in 2009
- Open-sourced in 2010
- Became top-level Apache project in 2014
- Commercial support provided by DataBricks
- Spark 1.0 (low-level APIs) ➔ **Spark 2.0 (structured APIs)**

amplab UC BERKELEY

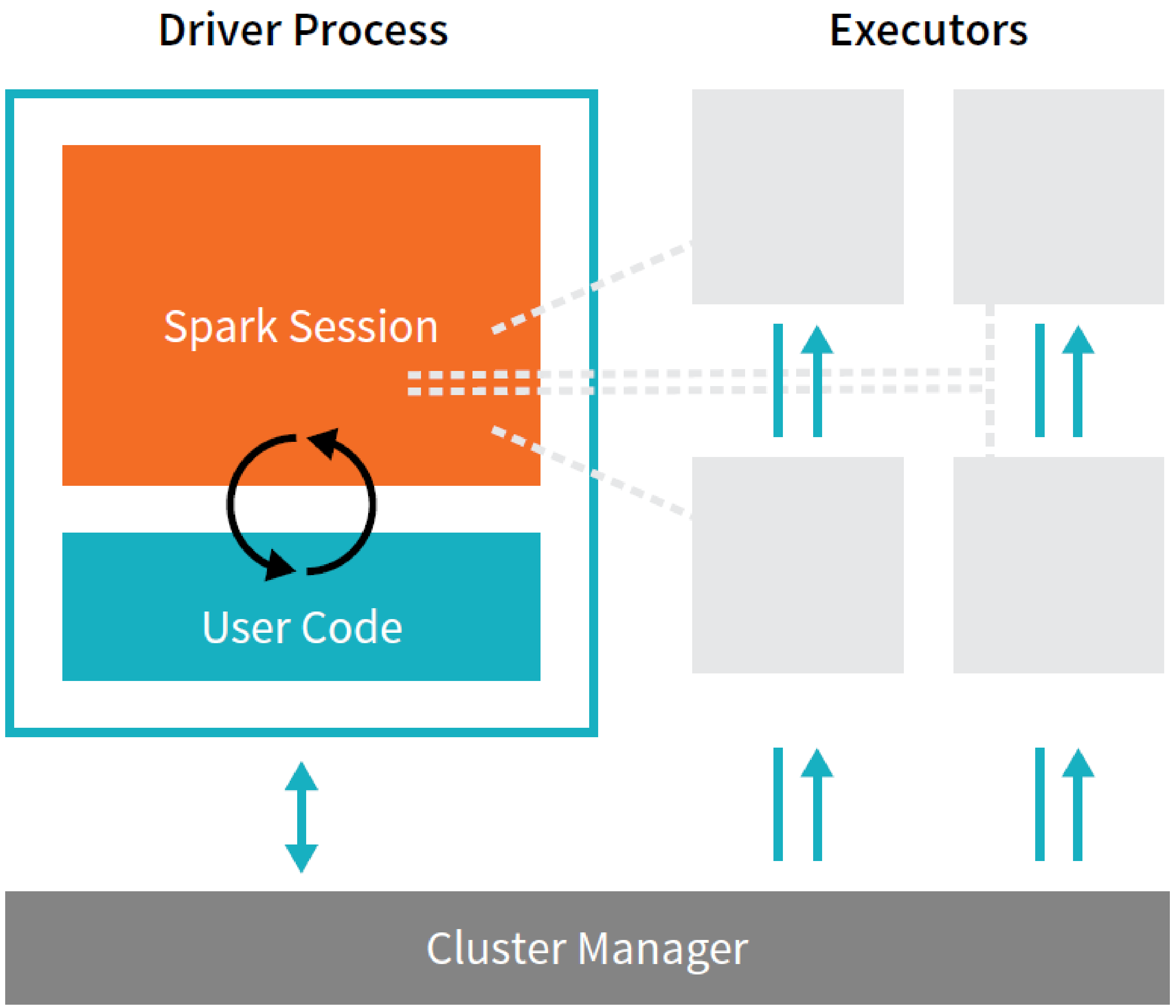
The Apache Software Foundation
Community-led development since 1999.

databricks™
making big data simple

Running Spark

- In the Cloud
  - Databricks Community Edition
    - Free for anyone to run Spark without the overhead and management of managing your own Spark cluster
    - https://community.cloud.databricks.com/
  - Amazon EMR
    - **mrjob** helps you to run Spark jobs on EMR (or your own Hadoop cluster)
    - https://mrjob.readthedocs.io/en/latest/guides/spark.html

Spark's Basic Architecture



Spark's Basic Architecture

- Spark Applications consist of a driver process and a set of executor processes.
  - The **driver** process runs your `main()` function, sits on a node in the cluster, and is responsible for three things:
    - maintaining information about the Spark Application;
    - responding to a user's program or input; and
    - analyzing, distributing, and scheduling work across the executors (defined momentarily).

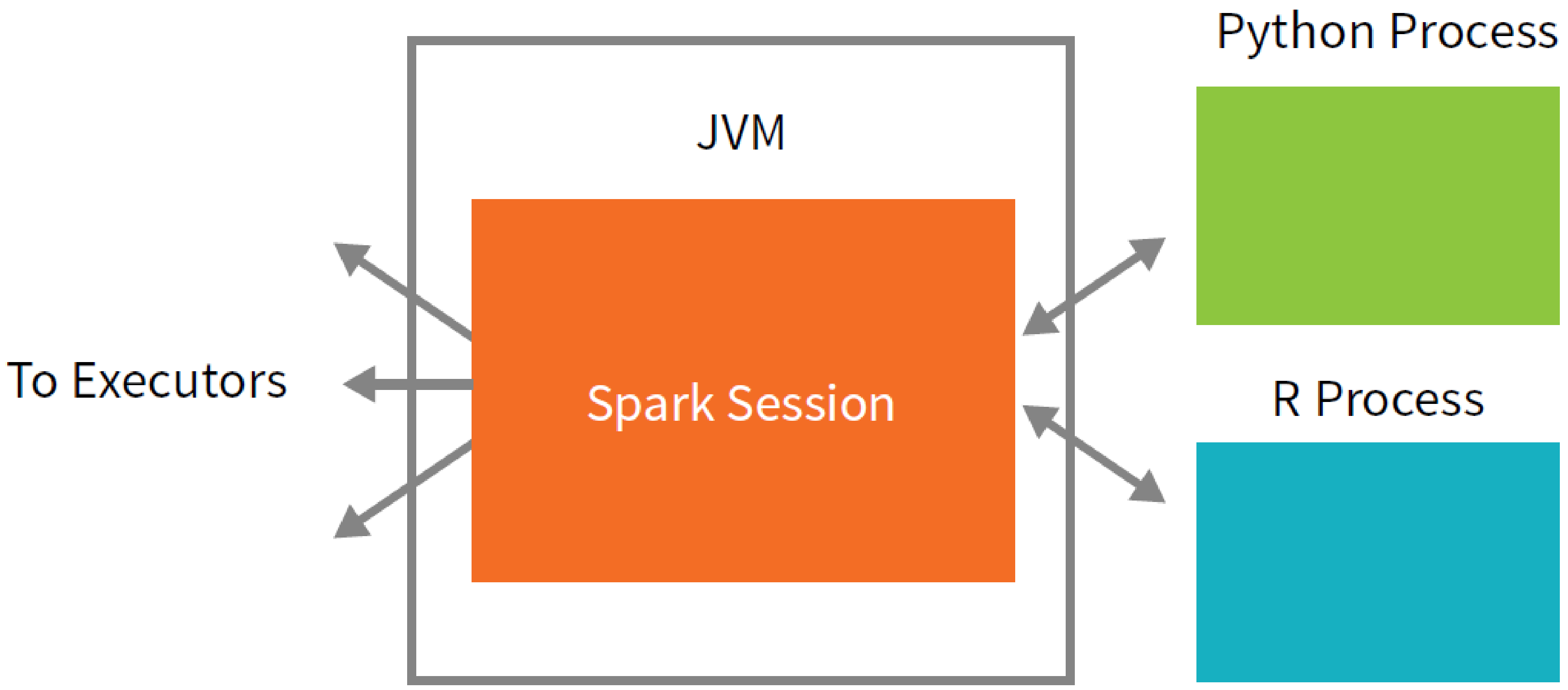
Spark’s Basic Architecture

- Spark Applications consist of a driver process and a set of executor processes.
  - Each **executor** is responsible for only two things:
    - executing code assigned to it by the driver and
    - reporting the state of the computation, on that executor, back to the driver node.

Spark’s Basic Architecture

- The **cluster manager** controls physical machines and allocates resources to Spark Applications.
  - This can be one of several core cluster managers: Spark’s standalone cluster manager, YARN, or Mesos.
  - There can be multiple Spark Applications running on a cluster at the same time.
  - The user can specify how many executors should fall on each node through configurations.

Spark's APIs



Spark’s APIs

- Spark itself is written in Scala, and runs on the Java Virtual Machine (JVM).
- Spark can be used from **Python**, Java, or Scala, R, or SQL.
  - When using Spark from a Python or R, the user never writes explicit JVM instructions, but instead writes Python and R code that Spark will translate into code that Spark can then run on the executor JVMs.

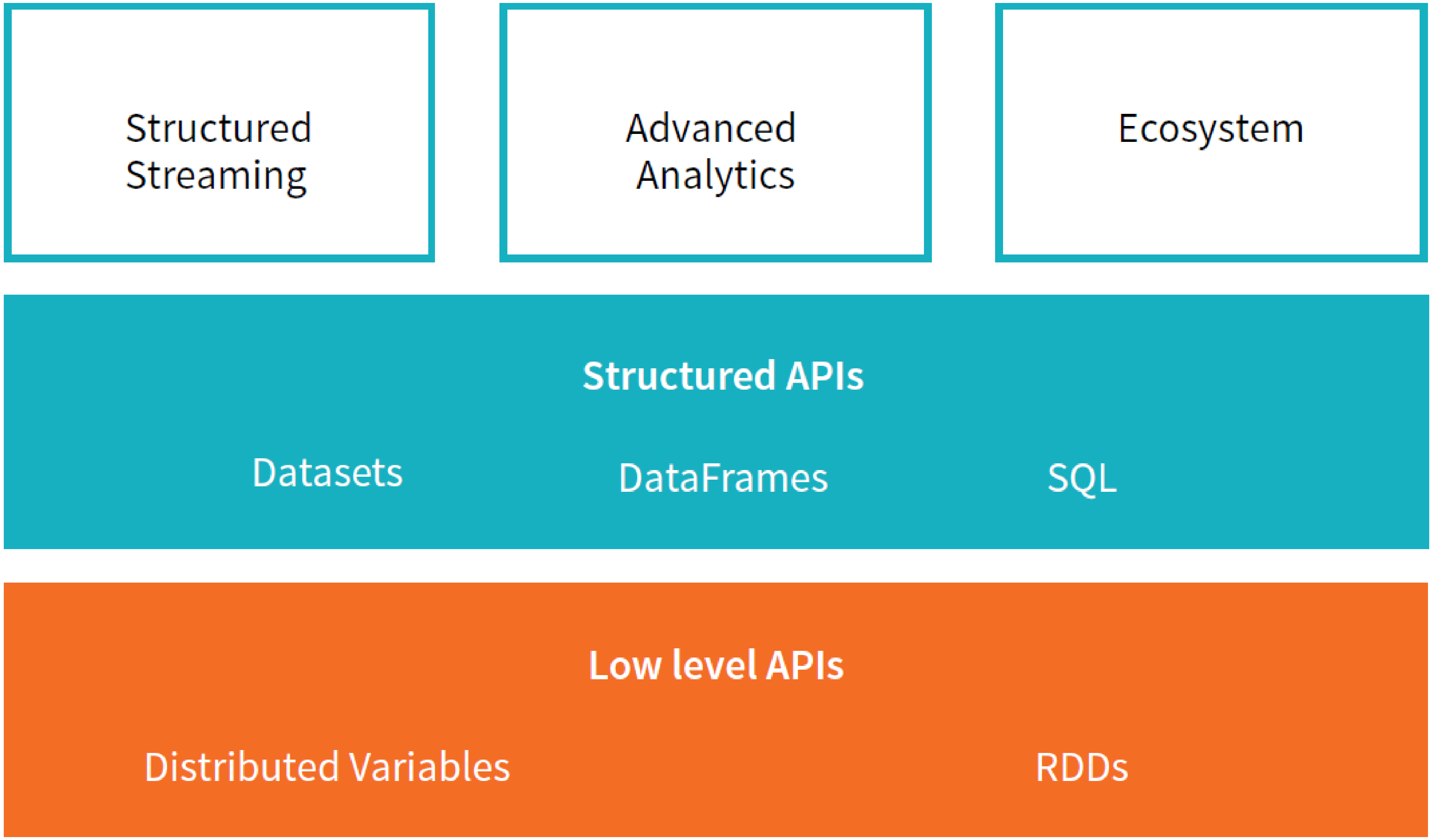
Spark's APIs

- The driver process manifests itself to the user as an object called the **SparkSession**.
  - The SparkSession instance is the way Spark executes user-defined manipulations across the cluster. There is a one to one correspondence between a SparkSession and a Spark Application.
  - In Scala and Python the variable is available as spark when you start up the console.

```
./bin/pyspark
```

```
spark
<pyspark.sql.session.SparkSession at 0x7efda4c1ccd0>
```

Spark's APIs


Structured Streaming
Advanced Analytics
Ecosystem
Structured APIs
Datasets
DataFrames
SQL
Low level APIs
Distributed Variables
RDDs

Spark’s APIs

- Spark has two fundamental sets of APIs
  - the low-level “Unstructured” APIs
    - Resilient Distributed Datasets (RDDs)
    - Distributed Variables
  - the high-level Structured APIs
    - DataFrames
    - Datasets
    - SQL

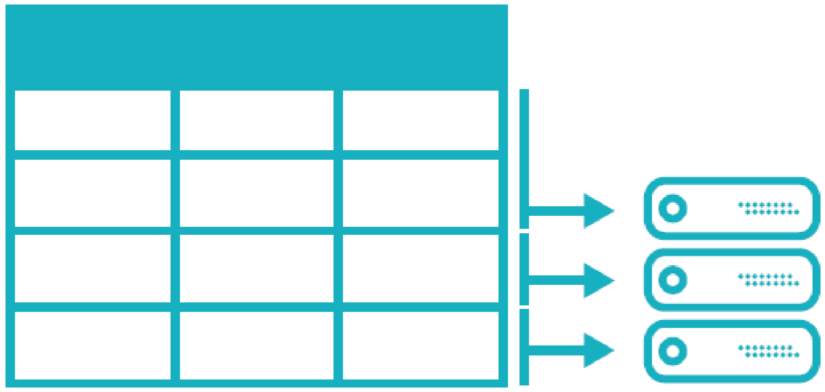
Spark’s APIs

- Virtually everything in Spark (DataFrame etc.) is built on top of RDDs.
- You should try to stick to the Structured APIs.
  - There are basically no instances in modern Spark where you should be using RDDs instead of the structured APIs (beyond manipulating some very raw unprocessed and unstructured data).

DataFrames

Spreadsheet on a single machine

Table or DataFrame partitioned across servers in a data center



DataFrames

- A DataFrame is the most common Structured API and simply represents *a **table** of data with rows and columns*.
  - The list of columns and the types in those columns are specified by the *schema*.
  - The fundamental difference from a spreadsheet with named columns is that while a spreadsheet sits on one computer in one specific location, a Spark DataFrame *can span many computers*.

DataFrames

- Python and R both have similar concepts.
  - However, Python or R DataFrames (with some exceptions) exist on one machine rather than multiple machines.
    - Python Dask
  - it’s quite easy to convert to Pandas (Python) or R DataFrames to Spark DataFrames.

DataFrames

- In order to allow every executor to perform work in parallel, Spark breaks up the data into chunks, called partitions.
  - A **partition** is a collection of *rows* that sit on one physical machine in our cluster.
  - With DataFrames, we do not (for the most part) manipulate partitions manually (on an individual basis).

Transformations & Actions

- Transformations
  - In Spark, the core data structures are *immutable* meaning they cannot be changed once created.
  - In order to “change” a DataFrame you will have to instruct Spark how you would like to modify the DataFrame you have into the one that you want. These instructions are called *transformations*.
  - Transformations are the core of how you will be expressing your business logic using Spark.

Transformations & Actions

- Actions
  - To trigger the computation, we run an *action* which instructs Spark to compute a result from a series of transformations.
    - (1) View data in the console;
    - (2) Collect data to native objects in the respective language;
    - (3) Write to output data sources.

Transformations & Actions

Transformations	map	flatMap
	filter	union
	sample	join
	groupByKey	cogroup
	reduceByKey	cross
	sortByKey	mapValues
Actions	collect	
	reduce	
	count	
	save	
	lookupKey	

Transformations & Actions

```
myRange = spark.range(1000).toDF("number")

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

divisBy2.count()
```

Lazy Evaluation

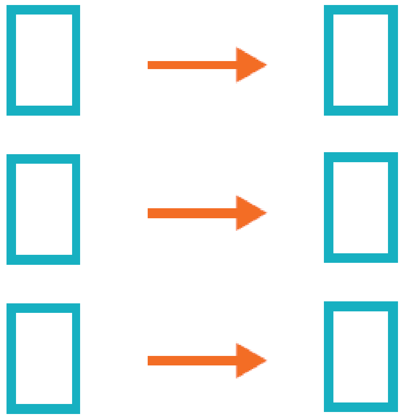
- Spark will wait until the very last moment to execute the graph of computation instructions.
  - In Spark, instead of modifying the data immediately when we express some operation, we build up a *plan* of transformations that we would like to apply to our data.
  - Spark, by waiting until the last minute to execute the code, will *compile* this plan from your raw, DataFrame transformations, to an efficient physical plan that will run as efficiently as possible across the cluster.

Lazy Evaluation

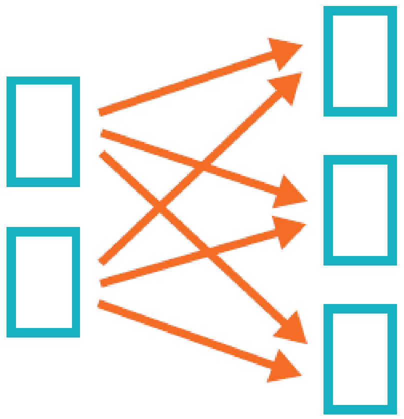
- This provides immense benefits to the end user because Spark can *optimize* the entire data flow *from end to end*.
  - An example of this is the so-called "predicate pushdown" on DataFrames.
    If we build a large Spark job but specify a filter at the end that only requires us to fetch one row from our source data, the most efficient way to execute this is to access the single record that we need.

Dependencies

Narrow Transformations
1 to 1



Wide Transformations (shuffles)
1 to N



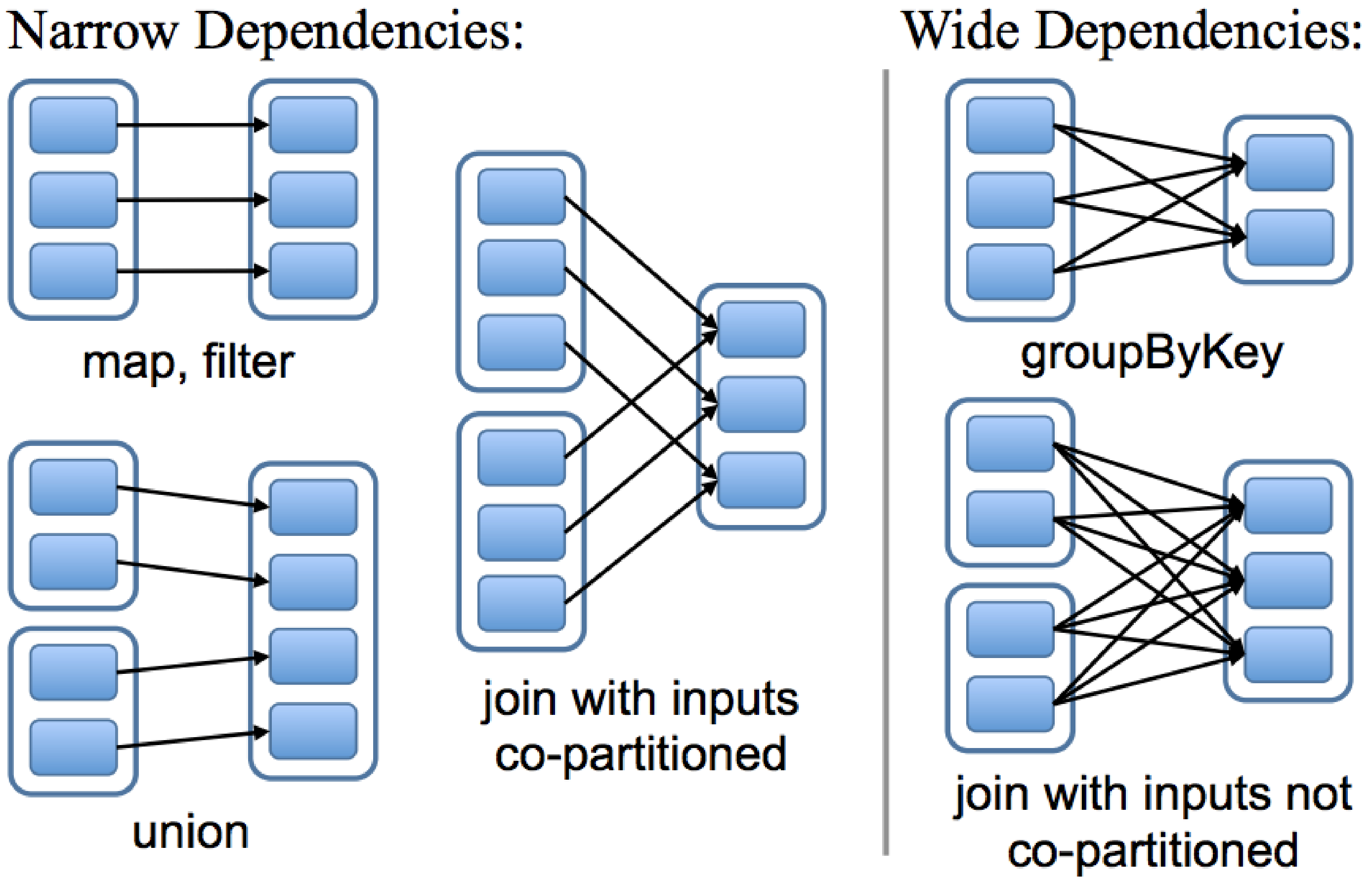
Dependencies

- Transformations consisting of *narrow dependencies* (or narrow transformations) are those where each input partition will contribute to only one output partition.
  - For example, in the preceding code snippet, our `where` statement specifies a narrow dependency.

Dependencies

- Transformations consisting of *wide dependencies* (or wide transformations) will have input partitions contributing to many output partitions.
  - You will often hear this referred to as a *shuffle* where Spark will exchange partitions across the cluster.

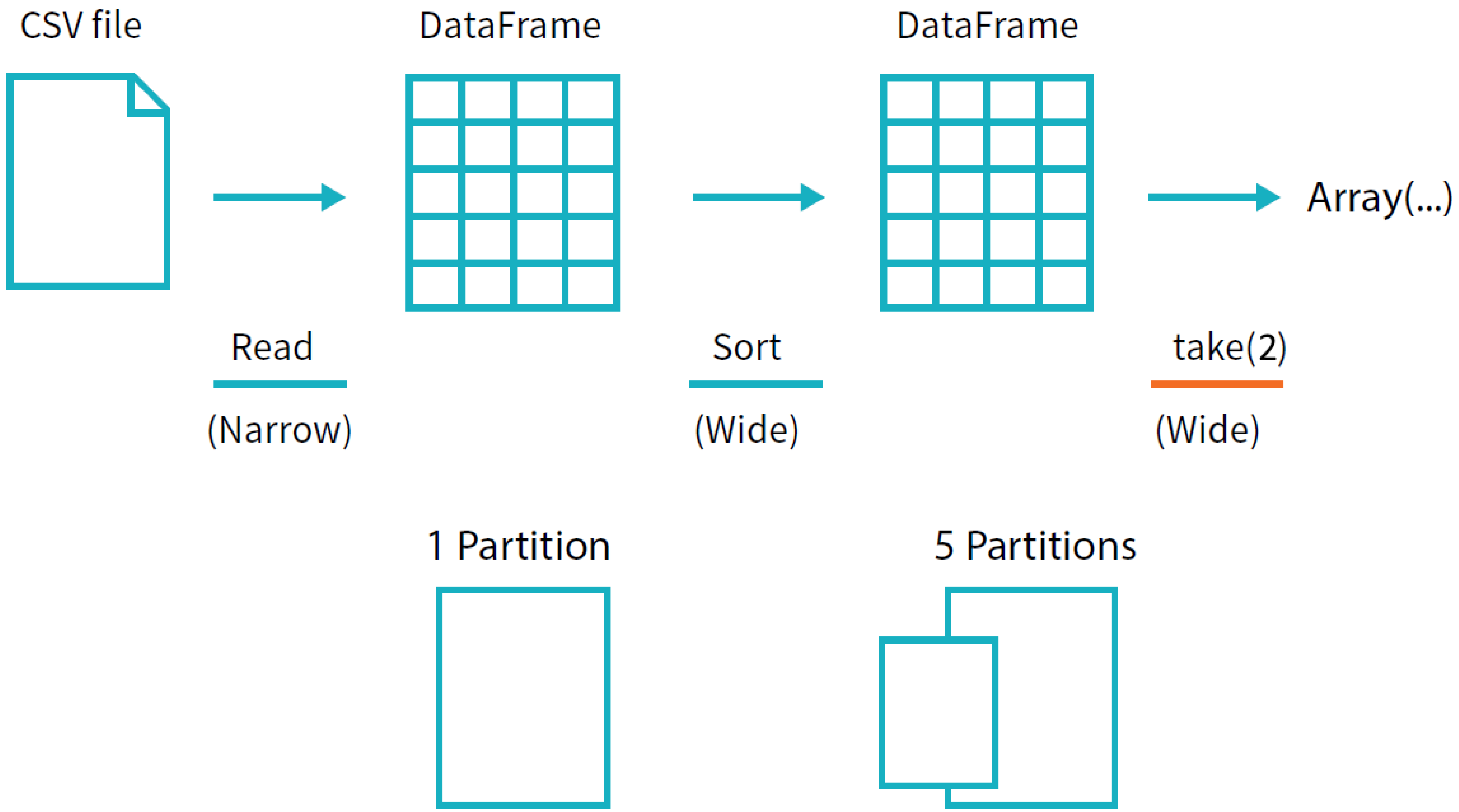
Dependencies


Narrow Dependencies:
Wide Dependencies:
map, filter
union
join with inputs co-partitioned
groupByKey
join with inputs not co-partitioned

Dependencies

- Spark will automatically perform an operation called ***pipelining*** on narrow dependencies.
  - If we specify multiple filters on DataFrames they'll all be performed *in-memory*.
- The same cannot be said for wide dependencies (shuffles).
  - When we perform a shuffle, Spark will write the results to disk.

An End-to-End Example



An End-to-End Example

```
$ head /mnt/defg/flight-data/csv/2015-summary.csv
DEST_COUNTRY_NAME,ORIGIN_COUNTRY_NAME,count
United States,Romania,15
United States,Croatia,1
United States,Ireland,344

flightData2015 = spark\
    .read\
    .option("inferSchema", "true")\
    .option("header", "true")\
    .csv("/mnt/defg/flight-data/csv/2015-summary.csv")

spark.conf.set("spark.sql.shuffle.partitions", "5")

flightData2015.sort("count").take(2)
... Array([United States,Singapore,1],
          [Moldova,United States,1])
```

MLlib

`org.apache.spark.ml`

- MLlib is a built-in library of Spark for machine learning that provides interfaces for:
    1. gathering and cleaning data, feature engineering and feature selection,
    2. training and tuning large scale supervised and unsupervised machine learning models,
    3. using those models in production.

MLlib helps with all three steps of the process but it really shines in steps one and two.

MLlib

- There are numerous tools for performing machine learning on a single machine.
  - scikit.learn, TensorFlow, etc.
- These single machine tools do reach limits in terms of
  - either the size of data you would like to train on
  - or the processing time.

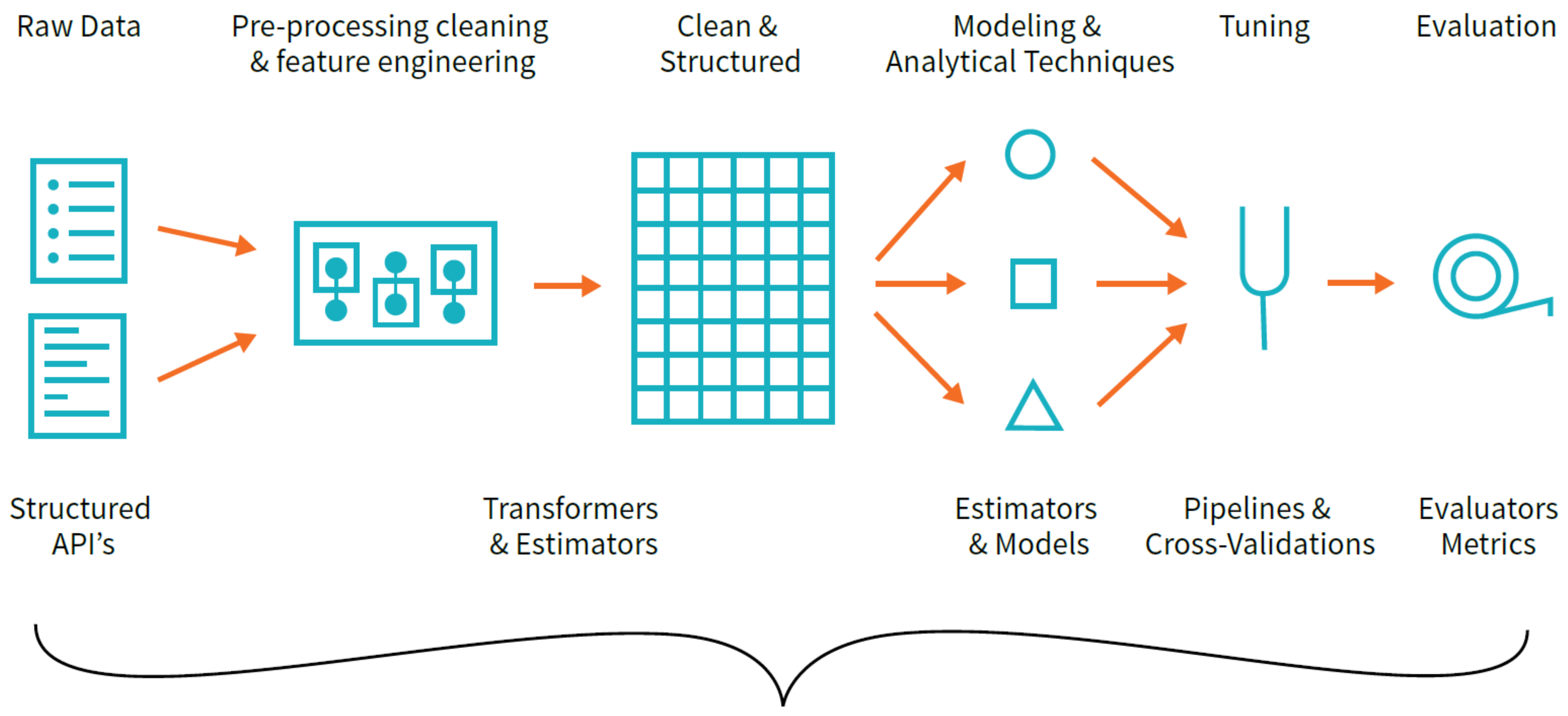
MLlib

- When and why should we use MLlib?
  1. To leverage Spark for *pre-processing and feature generation* to reduce the amount of time it might take to produce training and test sets from a large amount of data.
  2. To use Spark to do the heavy lifting when your input data or model size become too difficult or inconvenient to put on one machine.

MLlib

- Caveats
  - For example, if you train a recommender system on a Spark cluster, the resulting model will end up being way too large for use on a single machine for prediction.
  - For another example, Spark’s execution engine for logistic regression is not a low-latency one.
    - Therefore making single predictions quickly (< 500ms) is still challenging because of the costs of starting up and executing a Spark jobs, even on a single machine.

MLlib

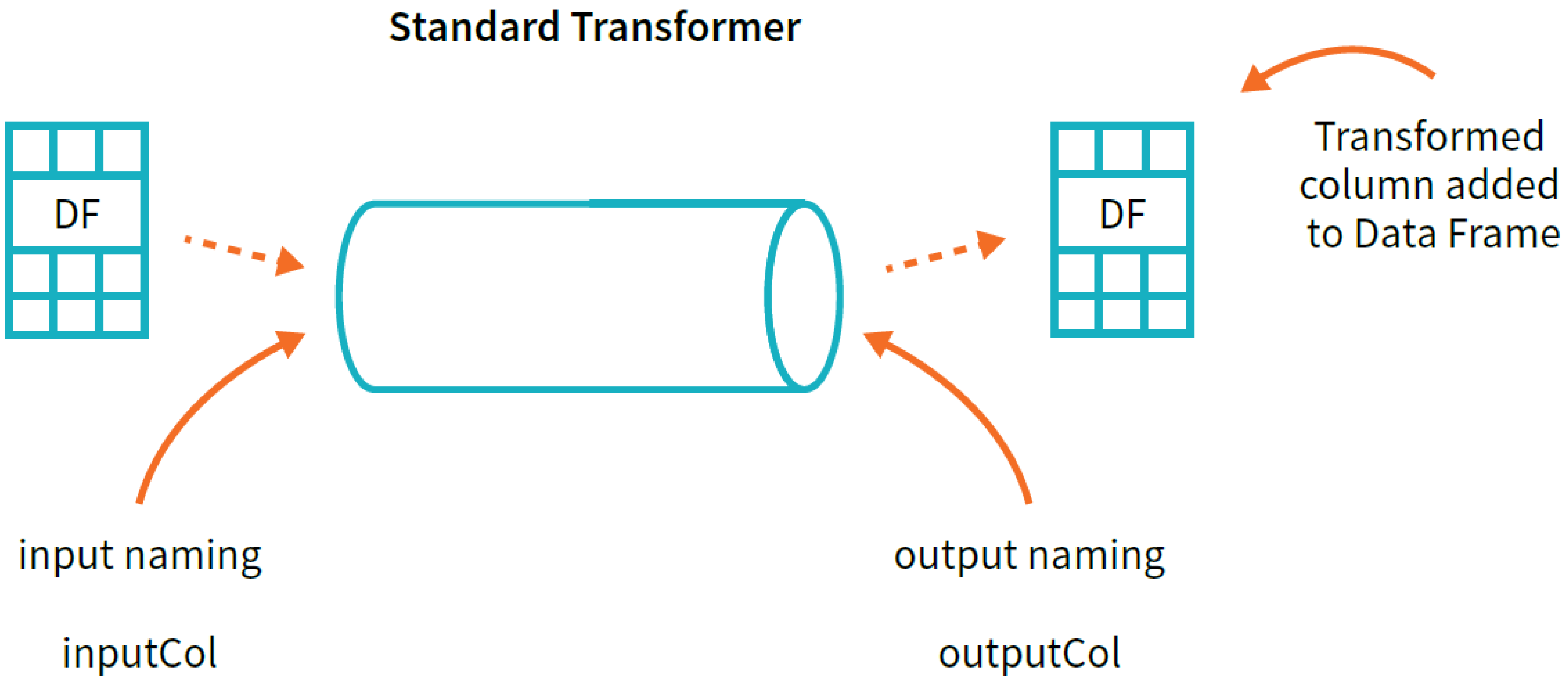

Raw Data
Pre-processing cleaning & feature engineering
Clean & Structured
Modeling & Analytical Techniques
Tuning
Evaluation
Structured API's
Transformers & Estimators
Estimators & Models
Pipelines & Cross-Validations
Evaluators Metrics

All in one pipeline

MLlib

- **Transformers** are functions that convert raw data in some way.
  - This might be to create a new interaction variable (from two other variables), normalize a column, or simply turn it into a `Double` to be input into a model.
  - An example of a transformer is one that converts string categorical variables into numerical values that can be used in MLlib.

MLlib



In general, transformers add new columns to DataFrames.

MLlib

- **Estimators** are one of two kinds of things.
  1. Firstly, estimators can be a kind of transformer that is initialized with data.
     - For example, in order to convert a column into a percentile representation we will need to initialize it based on the values in that column.
  2. Lastly, estimators are Spark’s name for the actual models that we will be training and turning into models so that we can use them to make predictions.

MLlib

- **Evaluators** allow us to see how a given estimator performs according to some criteria that we specify.
  - e.g., Area under the ROC Curve etc.
  - Once we select the best model from the ones that we tested, we can then use it to make predictions.

MLlib

- All inputs to machine learning algorithms in MLlib must consist of type `Double` (for labels) and `Vector[Double]` for features
  - For *dense vectors*, we specify the exact values.
  - For *sparse vectors*, we specify the total size and which values are nonzero.

```
from pyspark.ml.linalg import Vectors

denseVec = Vectors.dense(1.0, 2.0, 3.0)
size = 3
idx = [1, 2]
values = [2.0, 3.0]
sparseVec = Vectors.sparse(size, idx, values)
```

MLlib

```
df = spark.read.json("/mnt/defg/simple-ml")
df.orderBy("value2").show()
```

```
+-----+----+------+------------------+
|color| lab|value1|            value2|
+-----+----+------+------------------+
|green|good|     1|14.386294994851129|
|green| bad|    16|14.386294994851129|
| blue| bad|     8|14.386294994851129|
...
|  red| bad|    16|14.386294994851129|
|green|good|    12|14.386294994851129|
+-----+----+------+------------------+
```

MLlib

- RFormula is a declarative language for specifying machine learning models.
  1. ~ separate target and terms;
  2. + concat a term ("+ 0" means no intercept);
  3. - remove a term ("- 1" means no intercept too);
  4. : interaction (multiplication for numeric values, or binarized categorical values);
  5. . all columns except the target/dependant variable.

MLlib

```
from pyspark.ml.feature import Rformula

supervised = RFormula(formula="lab ~ . + color:value1
                                       + color:value2")
fittedRF = supervised.fit(df)
preparedDF = fittedRF.transform(df)
```

```
+-----+----+------+------------------+--------------------+-----+
|color| lab|value1|            value2|            features|label|
+-----+----+------+------------------+--------------------+-----+
|green|good|     1|14.386294994851129|(10,[1,2,3,5,8],[...|  1.0|
| blue| bad|     8|14.386294994851129|(10,[2,3,6,9],[8....|  0.0|
...
|  red| bad|     1| 38.97187133755819|(10,[0,2,3,4,7],[...|  0.0|
|  red| bad|     2|14.386294994851129|(10,[0,2,3,4,7],[...|  0.0|
+-----+----+------+------------------+--------------------+-----+
```

MLlib

- A **pipeline** allows you to set up a *dataflow* of the relevant transformations, ending with an estimator that is automatically *tuned* according to your specifications.
  – The result is a tuned model ready for a production use case.
  – Instances of transformers or models are not reused across pipelines or different models.

MLlib

```
train, test = df.randomSplit([0.7, 0.3])

rForm = RFormula()

from pyspark.ml.classification import LogisticRegression

lr = LogisticRegression()\
     .setLabelCol("label")\
     .setFeaturesCol("features")

from pyspark.ml import Pipeline

stages = [rForm, lr]
pipeline = Pipeline().setStages(stages)
```

MLlib

```
from pyspark.ml.tuning import ParamGridBuilder

params = ParamGridBuilder()\
         .addGrid(rForm.formula, [
             "lab ~ . + color:value1",
             "lab ~ . + color:value1 + color:value2"])\
         .addGrid(lr.elasticNetParam, [0.0, 0.5, 1.0])\
         .addGrid(lr.regParam, [0.1, 2.0])\
         .build()

from pyspark.ml.evaluation import
                           BinaryClassificationEvaluator

evaluator = BinaryClassificationEvaluator()\
            .setMetricName("areaUnderROC")\
            .setRawPredictionCol("prediction")\
            .setLabelCol("label")
```

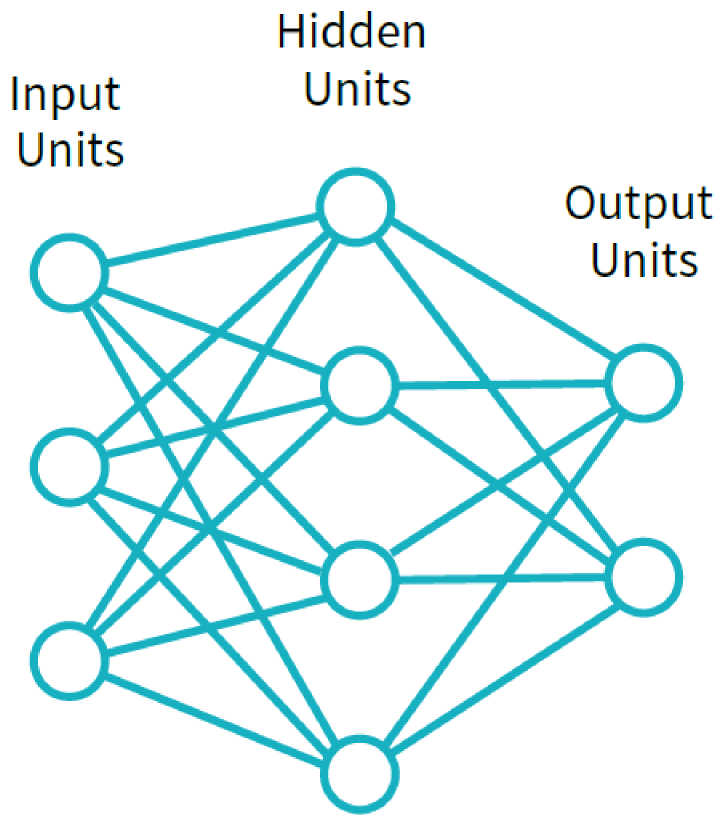
MLlib

```
from pyspark.ml.tuning import TrainValidationSplit

tvs = TrainValidationSplit()\
      .setTrainRatio(0.75)\
      .setEstimatorParamMaps(params)\
      .setEstimator(pipeline)\
      .setEvaluator(evaluator)

tvsFitted = tvs.fit(train)

evaluator.evaluate(tvsFitted.transform(test))
```

```
0.9166666666666667
```

Deep Learning

- Deep learning is rapidly growing into one of the most powerful techniques for solving machine learning problems, especially those involving unstructured data such as images, audio and text.



Deep Learning

- Major ways to use deep learning in Spark
  - **1. Inference**
    - You can often take a model from your favorite deep learning framework and apply it in parallel using a Spark function (e.g., `map`).

Deep Learning

- Major ways to use deep learning in Spark
  - **2. Featurization and Transfer Learning**
    - Many deep learning models learn useful feature representations in their lower layers as they train the network for an end-to-end task.
    - We can then use these features to learn models for a new problem not covered by the original dataset.
    - This method is called *transfer learning*, and generally involves cutting off the last few layers of a pre-trained model and retraining them with the data of interest.

Deep Learning

- Major ways to use deep learning in Spark
  - **3. Model Training**
    - You can use a Spark cluster to parallelize the training of a *single* model over multiple servers, communicating updates between them.
    - Alternatively, some libraries let the user train *multiple* instances of similar models in parallel to try various model architectures and hyperparameters, accelerating the model search and tuning process.

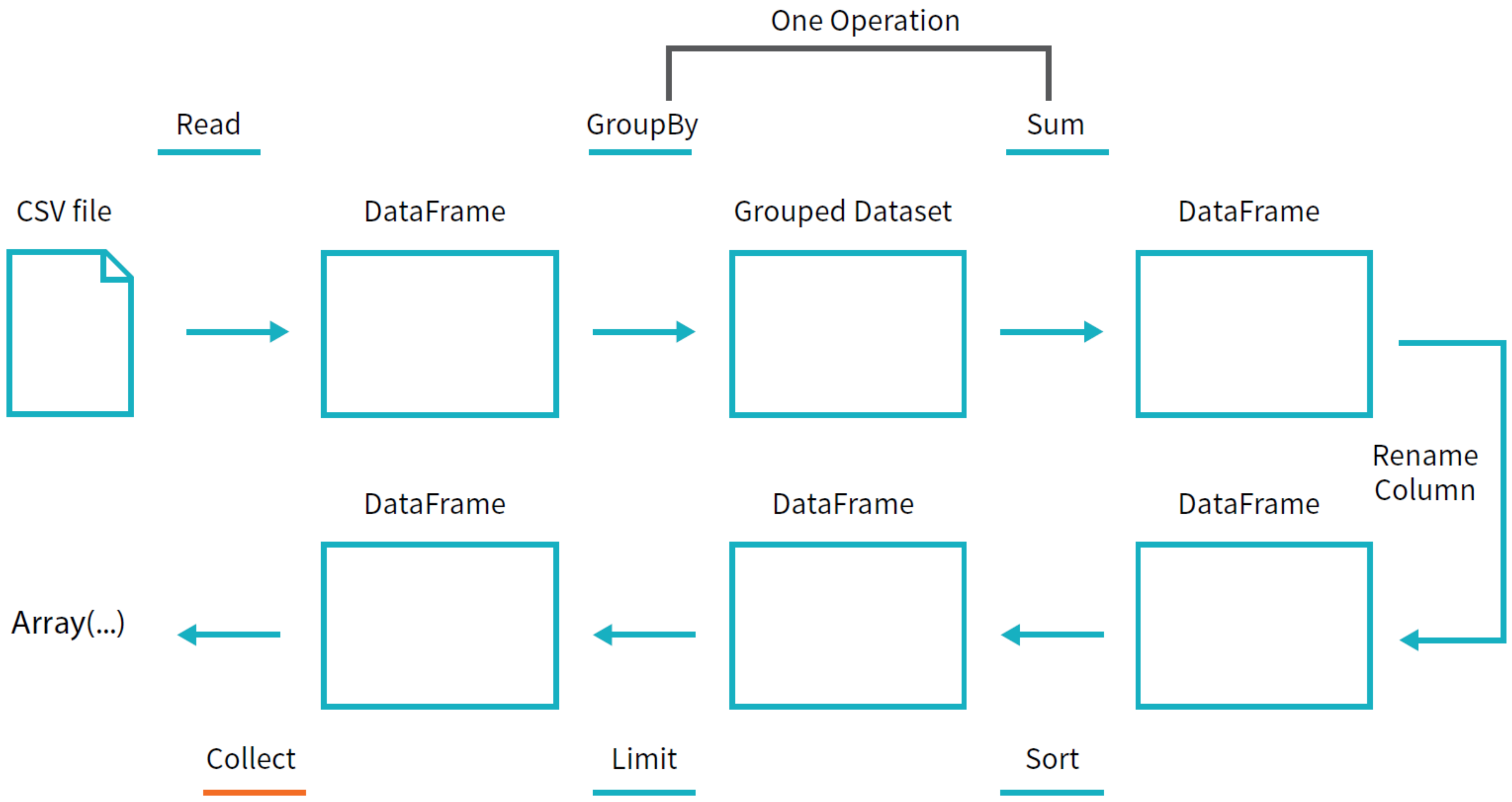
Deep Learning

- Deep Learning Libraries in Spark
  - MLlib Neural Network Support
    - `ml.classification.MultilayerPerceptronClassifier`
    - Most useful for training the last few layers of a classification model when using *transfer learning* on top of an existing deep learning based *featurizer*.
  - Deep Learning Pipelines
    - An open source package from Databricks that integrates deep learning functions into Spark's ML Pipelines API

Spark SQL

- Spark SQL allows you as a user to register any DataFrame as a table or view (a temporary table) and query it using pure SQL.
  - You can express your business logic in SQL or DataFrames (either in Python, R, Scala, or Java)
  - There is no performance difference between writing SQL queries or writing DataFrame code: they both "compile" to the same underlying plan (that we see in `explain`).

Spark SQL


One Operation
Read
GroupBy
Sum
CSV file
DataFrame
Grouped Dataset
DataFrame
Rename Column
DataFrame
DataFrame
DataFrame
Array(...)
Collect
Limit
Sort

Spark SQL

```
flightData2015.createOrReplaceTempView("flight_data_2015")

maxSql = spark.sql("""
    SELECT DEST_COUNTRY_NAME, sum(count) as destination_total
    FROM flight_data_2015
    GROUP BY DEST_COUNTRY_NAME
    ORDER BY sum(count) DESC
    LIMIT 5
""")

maxSql.collect()
```

Spark SQL

```
flightData2015.createOrReplaceTempView("flight_data_2015")

from pyspark.sql.functions import desc

flightData2015\
    .groupBy("DEST_COUNTRY_NAME")\
    .sum("count")\
    .withColumnRenamed("sum(count)", "destination_total")\
    .sort(desc("destination_total"))\
    .limit(5)\
    .collect()
```

Spark SQL

```
maxSql.explain()
```

```
== Physical Plan ==
TakeOrderedAndProject(limit=5, orderBy=[destination_total#16194L DESC],
output=[DEST_COUNTRY_NAME#7323,...
+- *HashAggregate(keys=[DEST_COUNTRY_NAME#7323], functions=[sum(count#7325L)])
+- Exchange hashpartitioning(DEST_COUNTRY_NAME#7323, 5)
+- *HashAggregate(keys=[DEST_COUNTRY_NAME#7323], functions=[partial sum(count#7325L)])
+- InMemoryTableScan [DEST_COUNTRY_NAME#7323, count#7325L]
+- InMemoryRelation [DEST_COUNTRY_NAME#7323, ORIGIN_COUNTRY_NAME#7324, count#7325L]...
+- *Scan csv [DEST_COUNTRY_NAME#7578,ORIGIN_COUNTRY_NAME#7579,count#7580L]...
```

Structured Streaming

- Structured Streaming allows you to take the same operations that you perform in batch mode using Spark's structured APIs, and run them in a streaming fashion.
  - This can reduce latency and allow for incremental processing.
  - The biggest change is that we would use `readStream` instead of `read`.

Structured Streaming

```
InvoiceNo,StockCode,Description,Quantity,InvoiceDate,UnitPrice,CustomerID,Country
536365,85123A,WHITE HANGING HEART T-LIGHT HOLDER,6,2010-12-01
08:26:00,2.55,17850.0,United Kingdom
536365,71053,WHITE METAL LANTERN,6,2010-12-01 08:26:00,3.39,17850.0,United Kingdom
536365,84406B,CREAM CUPID HEARTS COAT HANGER,8,2010-12-01
08:26:00,2.75,17850.0,United Kingdom
```

```
staticDataFrame = spark.read.format("csv")\
    .option("header", "true")\
    .option("inferSchema", "true")\
    .load("/mnt/defg/retail-data/by-day/*.csv")

staticDataFrame.createOrReplaceTempView("retail_data")

staticSchema = staticDataFrame.schema
```

Structured Streaming

```
streamingDataFrame = spark.readStream\
    .schema(staticSchema)\
    .option("maxFilesPerTrigger", 1)\
    .format("csv")\
    .option("header", "true")\
    .load("/mnt/defg/retail-data/by-day/*.csv")
```

Structured Streaming

```
from pyspark.sql.functions import window, column, desc, col

purchaseByCustomerPerHour = streamingDataFrame\
    .selectExpr(
    "CustomerId",
    "(UnitPrice * Quantity) as total_cost",
    "InvoiceDate")\
    .groupBy(
     col("CustomerId"), window(col("InvoiceDate"), "1 day"))\
    .sum("total_cost")

purchaseByCustomerPerHour.writeStream\
    .format("memory")\
    .queryName("customer_purchases")\ // in-memory table name
    .outputMode("complete")\
    .start()
```

Structured Streaming

Once we start the stream, we can run queries against the stream to debug what our result will look like.

```
spark.sql("""
    SELECT *
    FROM customer_purchases
    ORDER BY `sum(total_cost)` DESC
    """)\
    .show(5)
```

```
+----------+--------------------+------------------+
|CustomerId|              window|   sum(total_cost)|
+----------+--------------------+------------------+
|   17450.0|[2011-09-20 00:00...|          71601.44|
|      null|[2011-11-14 00:00...|          55316.08|
|      null|[2011-11-07 00:00...|          42939.17|
|      null|[2011-03-29 00:00...| 33521.39999999998|
|      null|[2011-12-08 00:00...|31975.590000000007|
+----------+--------------------+------------------+
```

GraphFrames

- GraphFrames vs Graph Databases
  - GraphFrames can scale to much larger workloads than many graph databases.
  - GraphFrames perform well in the context of analytics but not transaction data processing and serving.

GraphFrames

```
bikeStations = spark.read\
    .option("header","true")\
    .csv("/mnt/defg/bike-data/201508_station_data.csv")
tripData = spark.read\
    .option("header","true")\
    .csv("/mnt/defg/bike-data/201508_trip_data.csv")

stationVertices = bikeStations\
    .withColumnRenamed("name", "id")\
    .distinct()
tripEdges = tripData\
    .withColumnRenamed("Start Station", "src")\
    .withColumnRenamed("End Station", "dst")
```

GraphFrames

```
from graphframes import GraphFrame

stationGraph = GraphFrame(stationVertices, tripEdges)
stationGraph.cache()

print("Total Number of Stations: " +
      str(stationGraph.vertices.count()))
print("Total Number of Trips in Graph: " +
      str(stationGraph.edges.count()))
```

```
Total Number of Stations: 70
Total Number of Trips in Graph: 354152
```

GraphFrames

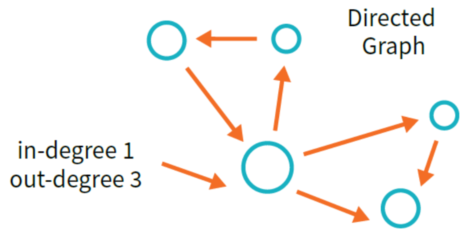
```
from pyspark.sql.functions import desc

stationGraph\
    .edges\
    .where("src = ‘Townsend at 7th’ OR
           dst = ‘Townsend at 7th’")\
    .groupBy("src", "dst")\
    .count()\
    .orderBy(desc("count"))\
    .show(10)
+--------------------+--------------------+-----+
|                 src|                 dst|count|
+--------------------+--------------------+-----+
|San Francisco Cal...|     Townsend at 7th| 3748|
|     Townsend at 7th|San Francisco Cal...| 2734|
...
|   Steuart at Market|     Townsend at 7th|  746|
|     Townsend at 7th|Temporary Transba...|  740|
+--------------------+--------------------+-----+
```

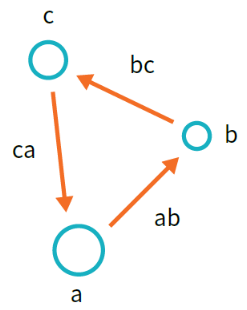
GraphFrames

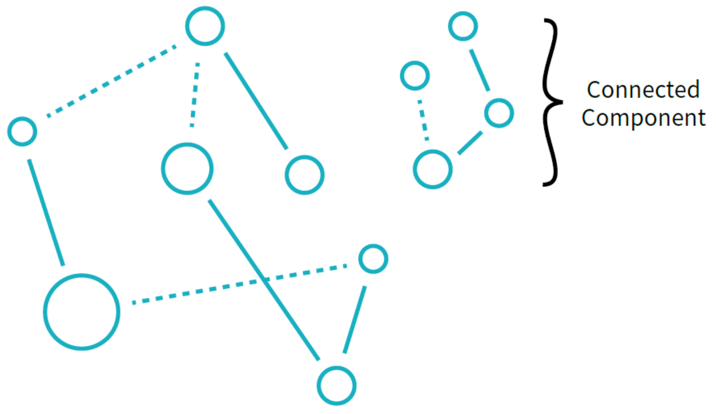
```
townAnd7thEdges = stationGraph\
    .edges\
    .where("src = ‘Townsend at 7th’ OR
            dst = ‘Townsend at 7th’")

subgraph = GraphFrame(stationGraph.vertices,
                      townAnd7thEdges)
```

GraphFrames

- Graph Algorithms
  - PageRank
  - In and Out Degrees
  - Breadth-first Search
  - (Strongly) Connected Components
  - Motif Finding
  - ……







Take Home Messages

- A Gentle Introduction to Spark
  - DataFrames vs RDDs
  - Transformations vs Actions
  - Lazy Evaluation
  - Narrow vs Wide Dependencies
- The Data Scientist’s Guide to Spark
  - MLlib