



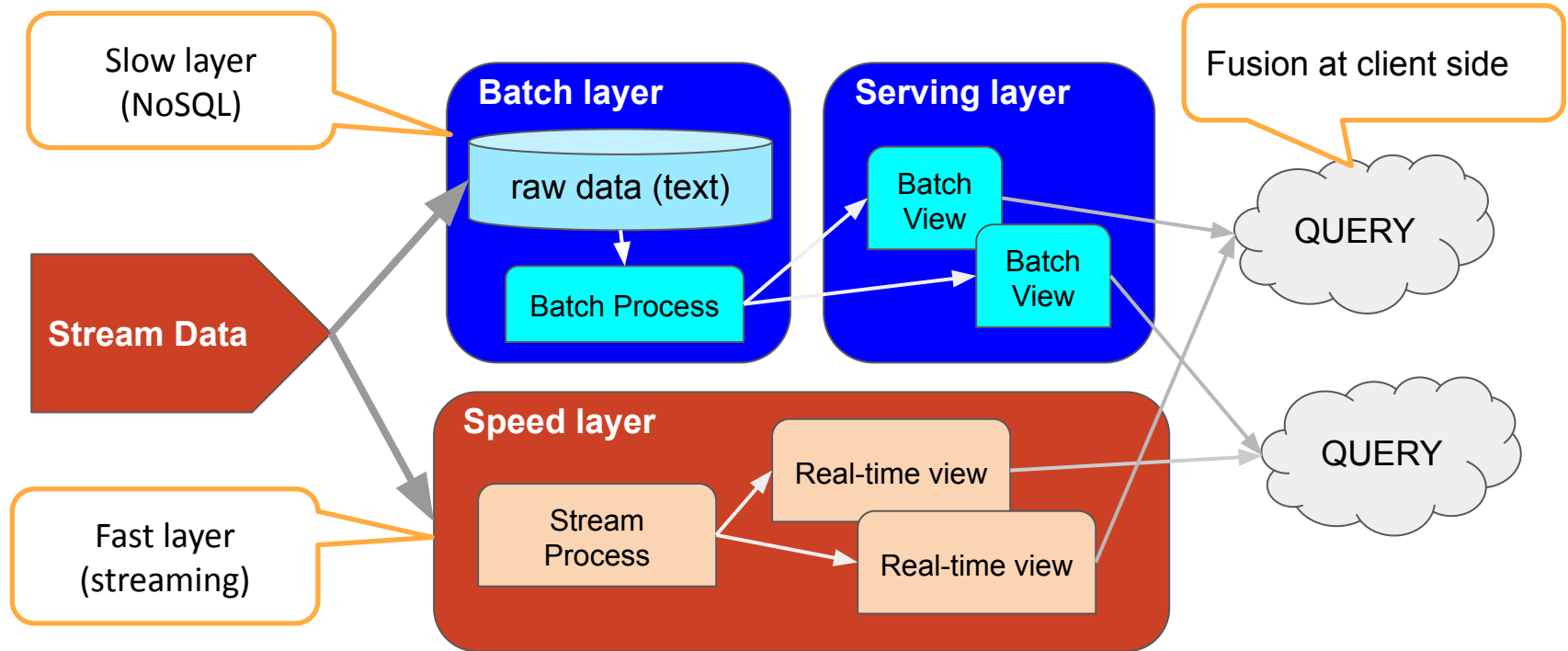
# Scaling up Data Analytics

Different sizes different solutions

# Learning goals

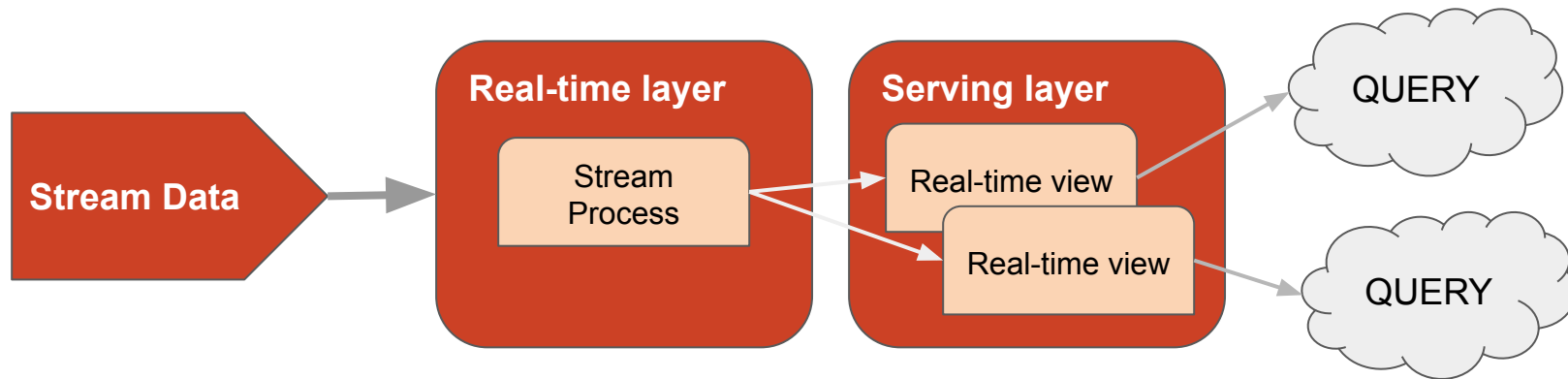
- To understand the different data structures for efficient storage
- To manage large datasets for analysis in Python
- To test different Python libraries for large-scale datasets
- To distribute and parallelize very large datasets with Spark
- To process streams of data

# Recap. Lambda Architecture (Streaming data)

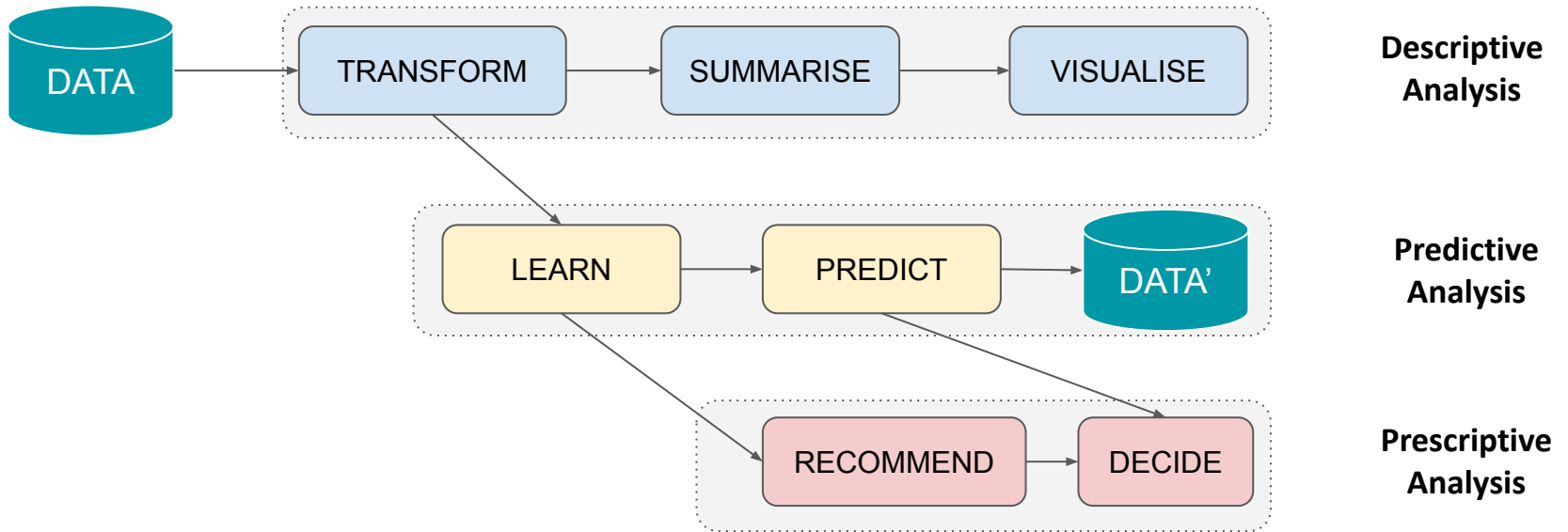


# Recap. Kappa Architecture

- Use of “logs” (append-only files) to store data.
- The “batch” layer of the Lambda architecture is removed.
- Greatly simplifies code maintenance.
- Great processing speed.



# Recap. Data Value Chains



# Optimising Data Storage & Transfer

Processing large datasets

or how to organise data for efficient  
retrieval and processing

# OLTP vs. OLAP

OnLine Transactional Processing

## **OLTP**

Optimised for quick read and write transactions (normal forms)

The rows are the atomic piece of data because of frequent updates

Most of the SQL databases are OLTP

OnLine Analytical Processing

## **OLAP**

Multidimensional nature of data

Read-only large databases (historical data)

Complex aggregations of large datasets for analysis

# Row vs. Columnar layouts

Data is stored as a *sequence of bytes*, not as a bi-dimensional structure.

Data of interest must be stored closed to each other for efficient retrieval (blocks/pages):

- In a row-oriented layout, records are arranged sequentially
- In a columnar layout, columns are arranged sequentially (e.g. [Pandas](#))
- In a hybrid layout, rows are split in blocks and each block is arranged with columnar layout

Logical table

|      | col1 | col2 | col3 |
|------|------|------|------|
| row1 | 1    | 2    | 3    |
| row2 | 4    | 5    | 6    |
| row3 | 7    | 8    | 9    |
| row4 | 10   | 11   | 12   |

Row-oriented layout (SequenceFile)

| row1  | row2  | row3  | row4     |
|-------|-------|-------|----------|
| 1 2 3 | 4 5 6 | 7 8 9 | 10 11 12 |

OLTP ✓

Column-oriented layout (RCFile)

| row split 1 |   |      |   |      |   | row split 2 |    |      |    |      |    |
|-------------|---|------|---|------|---|-------------|----|------|----|------|----|
| col1        |   | col2 |   | col3 |   | col1        |    | col2 |    | col3 |    |
| 1           | 4 | 2    | 5 | 3    | 6 | 7           | 10 | 8    | 11 | 9    | 12 |

OLAP ✓



# The two pillars

## ARROW MEMORY

Fast in-memory processing

- vectorized execution
- zero copy transfer
- cross language (**pyarrow** for Python)
- integration with Pandas > 2.0

```
pd.read_csv(fname, engine='pyarrow', type_backend='pyarrow')
```

<https://arrow.apache.org/docs/python/pandas.html>  
<https://datapythonista.me/blog/pandas-20-and-the-arrow-revolution-part-i>

## PARQUET

Fast Disk Retrieval (I/O)

- projection/predicate push down
- at rest compression
- cross language (**parquet** for Python)
- included in Pandas > 2.0

```
df.to_parquet('df.parquet.gzip', compression='gzip')  
pd.read_parquet('df.parquet.gzip')
```

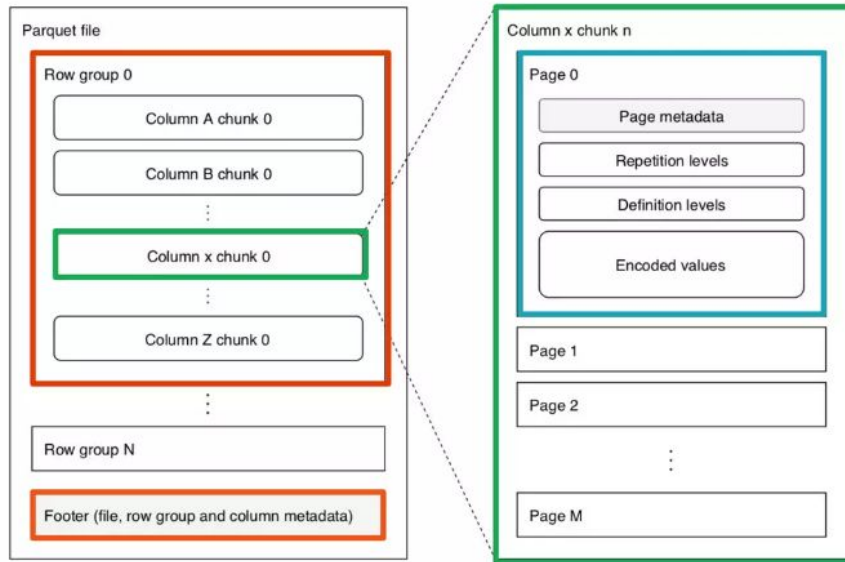
<https://towardsdatascience.com/demystifying-the-parquet-file-format-13adb0206705>

# Parquet

## Organisation:

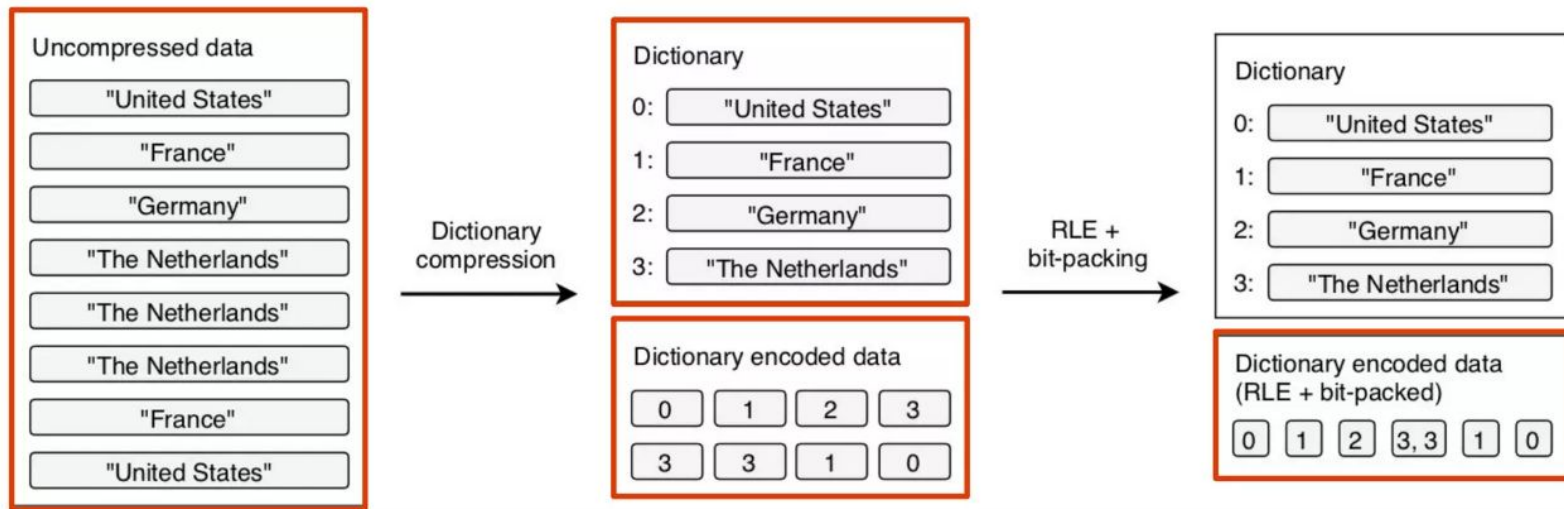
- Usually one dataset = multiple files
- Row group (128 MB)
- Column chunks
- Pages (default 1MB)
  - Metadata (min, max, count)
  - Rep/Def levels
  - Encoded values
- Encoding:
  - Plain: bytes [data] ...
  - RLE: Run-length encoding + bit packing + dictionary compression

It allows *predicates* to be pushed down!



<https://www.slideshare.net/databricks/the-parquet-format-and-performance-optimization-opportunities>

# Parquet RLE\_Dictionary Encoding



<https://www.slideshare.net/databricks/the-parquet-format-and-performance-optimization-opportunities>

# Alternatives/Complements to Pandas

## POLA-RS (<https://www.pola.rs/>)

- Implemented in RUST
- Arrow Memory in RUST
- Lazy/Eager modes
- Parquet import/export

## VAEX (<https://vaex.io/>)

- Lazy mode
- Arrow Memory
- Parquet import/export
- Focused on temporal-spatial data

## DuckDB ([link](#))

- SQL-like interaction with datasets
- Parquet import/export
- Arrow Memory
- Integrations with Pandas/Polars/Vaex

## Feather ([link](#))

- File format of Apache Arrow

```
import pyarrow.feather as feather
feather.write_feather(df, '/path/to/file')
read_df = feather.read_feather('/path/to/file')
```

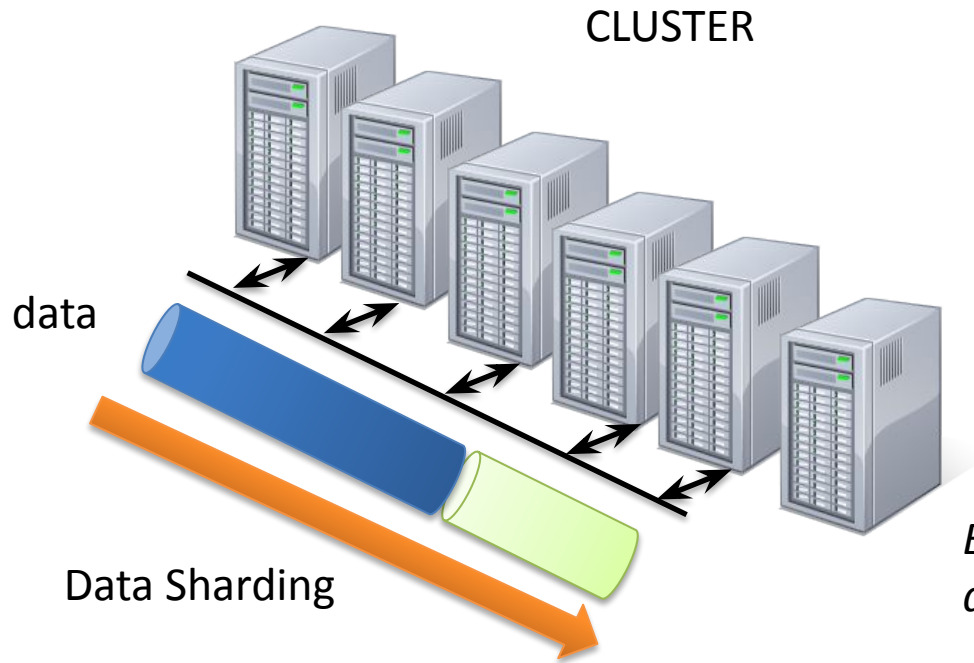
# Hadoop

Distributed File System

Dealing with large data in a cluster of servers:

- HDFS
- Map-Reduce

# Horizontal data scaling



Data is partitioned and distributed across the cluster.

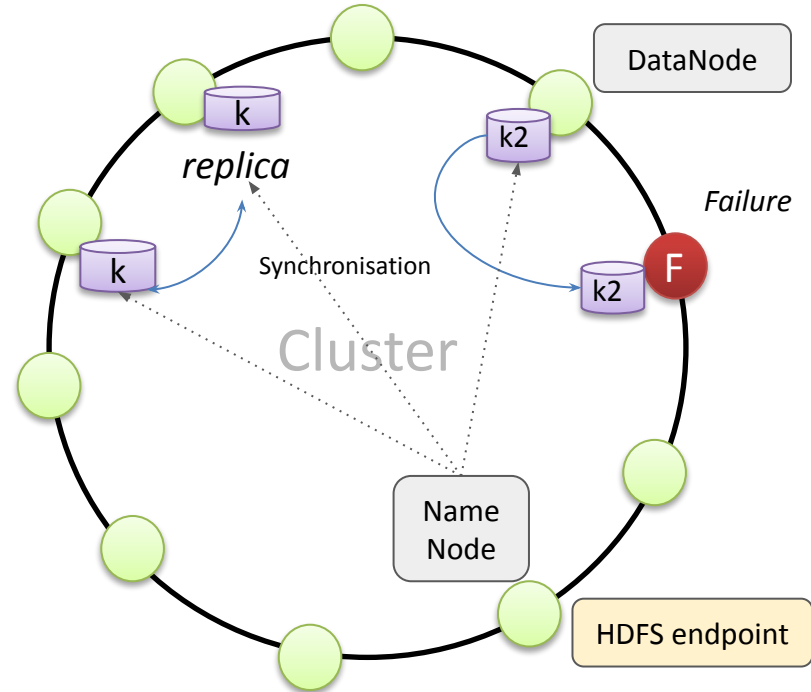
Replication is necessary for fault tolerance.

*Extend it with new nodes as needed by the data size*

# HDFS: Hadoop Distributed File System

## File system:

- Files and folders
- Segmentation
  - Files divided into blocks
  - Replication of blocks
- Commands inspired in Linux:
  - `hdfs dfs -ls`, `-cat`, `-mkdir`, etc.
- Data transfer between nodes:
  - `hdfs dfs -put`
- Data processing with MAP-REDUCE



# MAP-REDUCE

MR = Message-passing + data parallel + pipeline work + high level specification

The framework (Hadoop) is responsible for partitioning the input files, distributing the workload, mixing the outputs and exchanging messages.

Relies on the principles of functional programming:

The user only specifies the *mappers* and *reducers* (high level specification)

**Mappers** process parts of data and emit the results as (key, value) pairs  
**Reducers** gather all the values for a same key and emit the aggregation of values also as (key, value) pairs

All the emitted data is captured and processed by the framework.



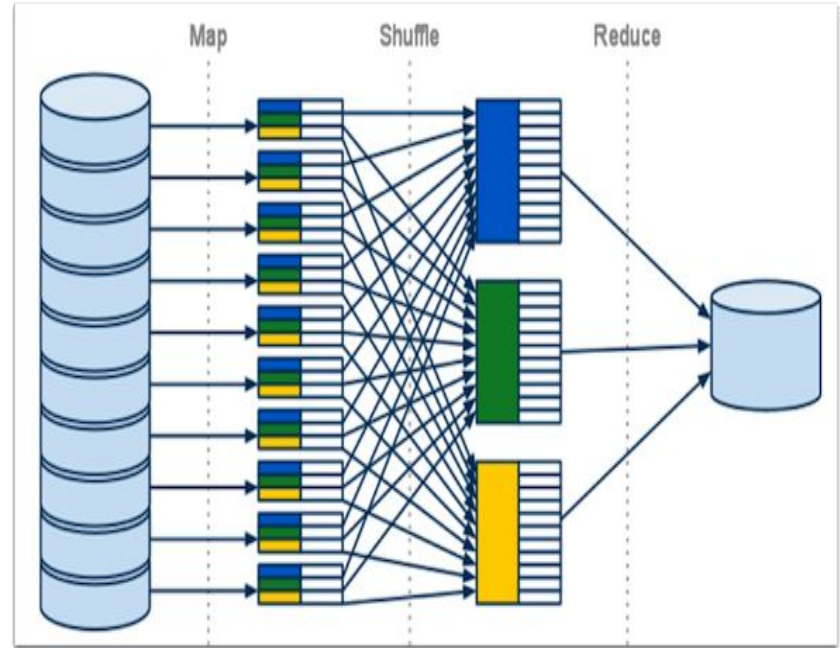
# MR Architecture

MR consists of three sequential phases:

1. Map: Data is partitioned and distributed to the *mappers* (jobs)
2. Shuffle: Once all parts are processed by the mappers, all emitted pairs (*key, value*) are sorted by keys
3. Reduce: Pairs with the same key are grouped and sent to the *reducers* (jobs) as iterators.

All the outputs of mappers/reducers are sent to temporary distributed files.

The number of mappers and reducers is determined before launching the process.



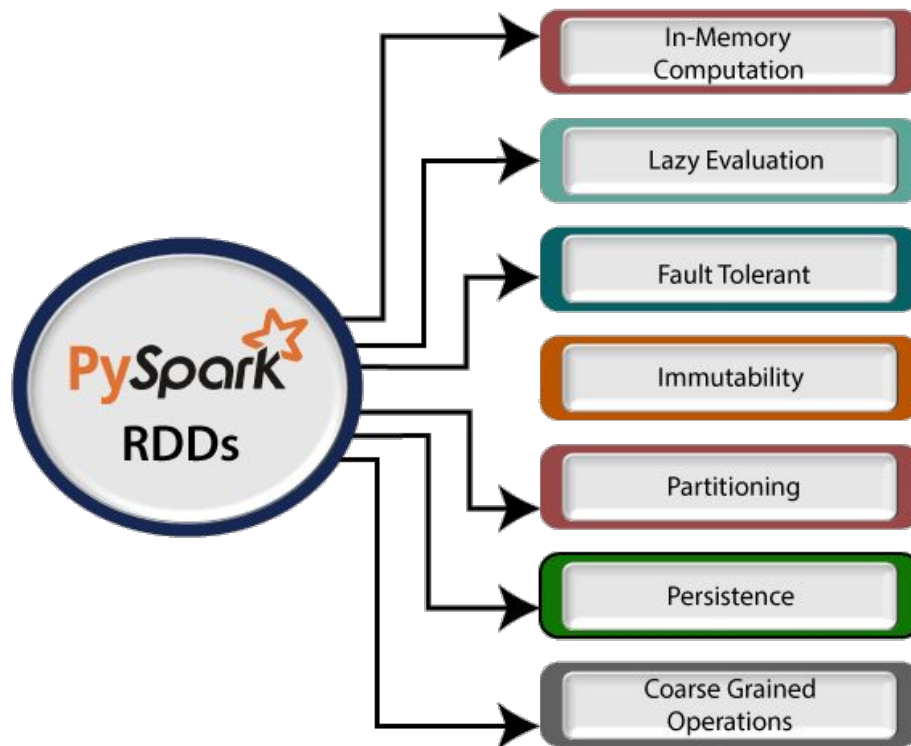
# SPARK

Resilient Distributed Datasets  
(RDD)

RDD is a solution to the low performance of Hadoop due to the extensive use of disk

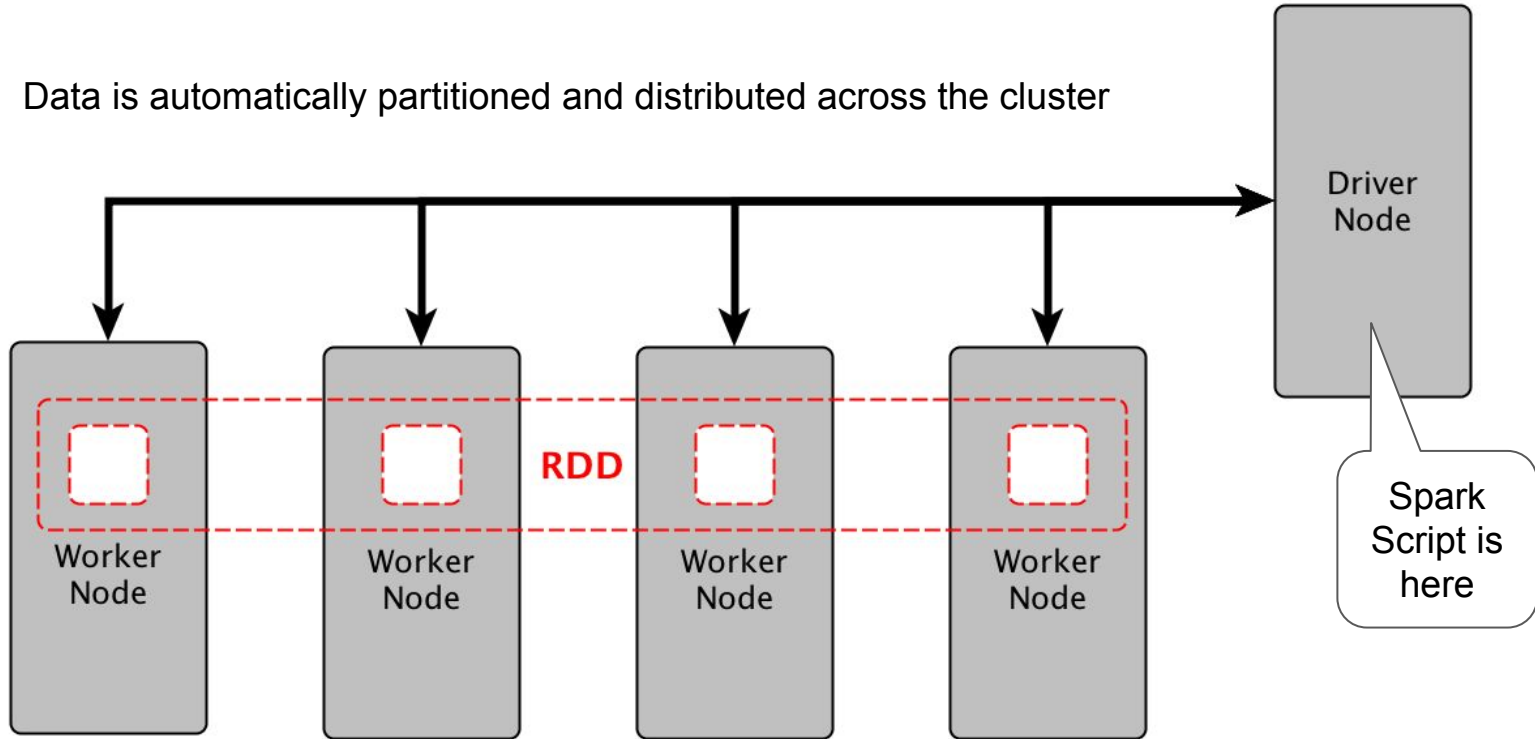
Memory is the primary resource

# Resilient Distributed Datasets



# An RDD distributes across the available nodes

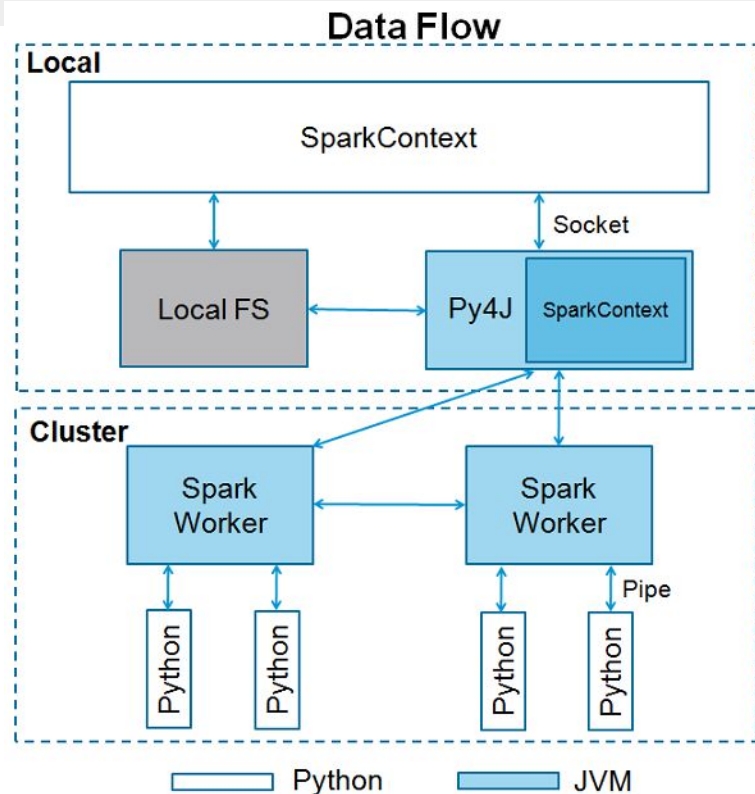
Data is automatically partitioned and distributed across the cluster



# SparkContext

The SparkContext is created in the main program (e.g., using pyspark)

- It connects to the cluster (or local)
- It allows the creation of RDDs and other diffusion variables & accumulators
- It broadcasts the diffusion variables
- It manages the RDDs created in the script **as transformations** (DAG)
- It activates the RDDs **actions**



DAG: stands for directed acyclic graph

# Transformations versus actions

## TRANSFORMATIONS

sc is the SparkContext

```
rdd = sc.parellelize(data)
rdd = sc.textFile(path/to/files)
```

RDDs transformations (methods):

- filter(*f*), map(*f*), flatMap(*f*), sortByKey(*f*), groupByKey(*f*), reduceByKey(*f*)
- join(*rdd*), substractByKey(*rdd*)

Transformations can be chained to get the intended RDDs.

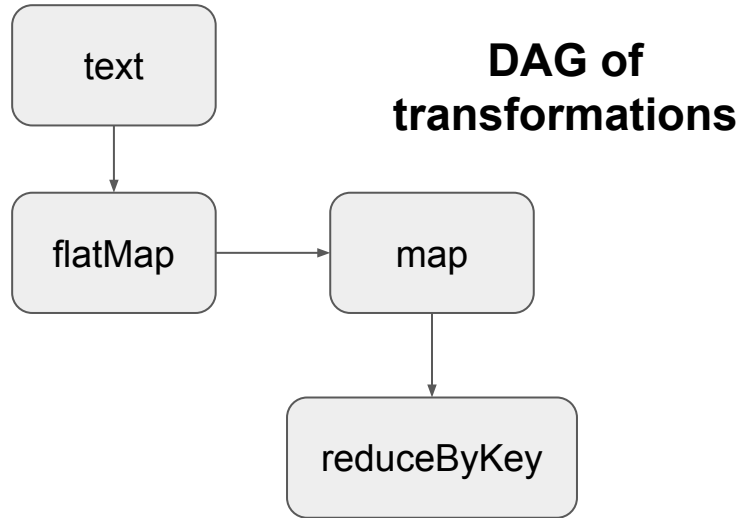
## ACTIONS

These methods execute the DAG of transformations to materialise the RDD:

- count(), first(), max(), min()
- take(*n*), collect(), collectAsMap()
- reduce()
- countByKey(), lookup(*key*)
- saveAsFile(path/to/files)

# Example

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()
sc = spark.sparkContext
text = sc.textFile("path/to/file")
f = lambda line: line.split(" ")
counts = text.flatMap(f) \
               .map(lambda word: (word, 1)) \
               .reduceByKey(lambda a, b: a + b)
counts.take(5)
```



The action `.take(5)` triggers the execution of the jobs associated to the DAG, finishing with the consolidation and transfer of the required results.

# Data frames as wrapper of RDDs

Any RDD can be transformed into a Spark *data frame* as follows:

```
df = rdd.toDF()  
df = df.toDF('col1', 'col2', ...)
```

A data frame is indeed an RDD (df.rdd) where elements are structured as data records.

Spark has a special session for managing data frames:

```
from pyspark.sql import SparkSession  
  
spark = SparkSession.builder.getOrCreate()
```

We can read/write any kind of data file (e.g., CSV, parquet, json, etc.) as a data frame or table:

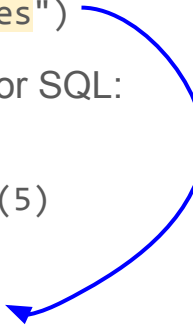
```
df = spark.read.options(header=True).csv(fname)  
df = spark.read.parquet(fname)  
df = spark.write.options(...).csv(fname)
```

```
spark.read.csv(fname)  
    .createOrReplaceTempView("Zipcodes")
```

And then working with them with functions or SQL:

```
df.select("country", "city", "zipcode",  
         "state").where("state == 'AZ']").show(5)
```

```
spark.sql(""" SELECT  country, city,  
               zipcode, state FROM ZIPCODES WHERE  
               state = 'AZ' """).show(5)
```





# Spark ML

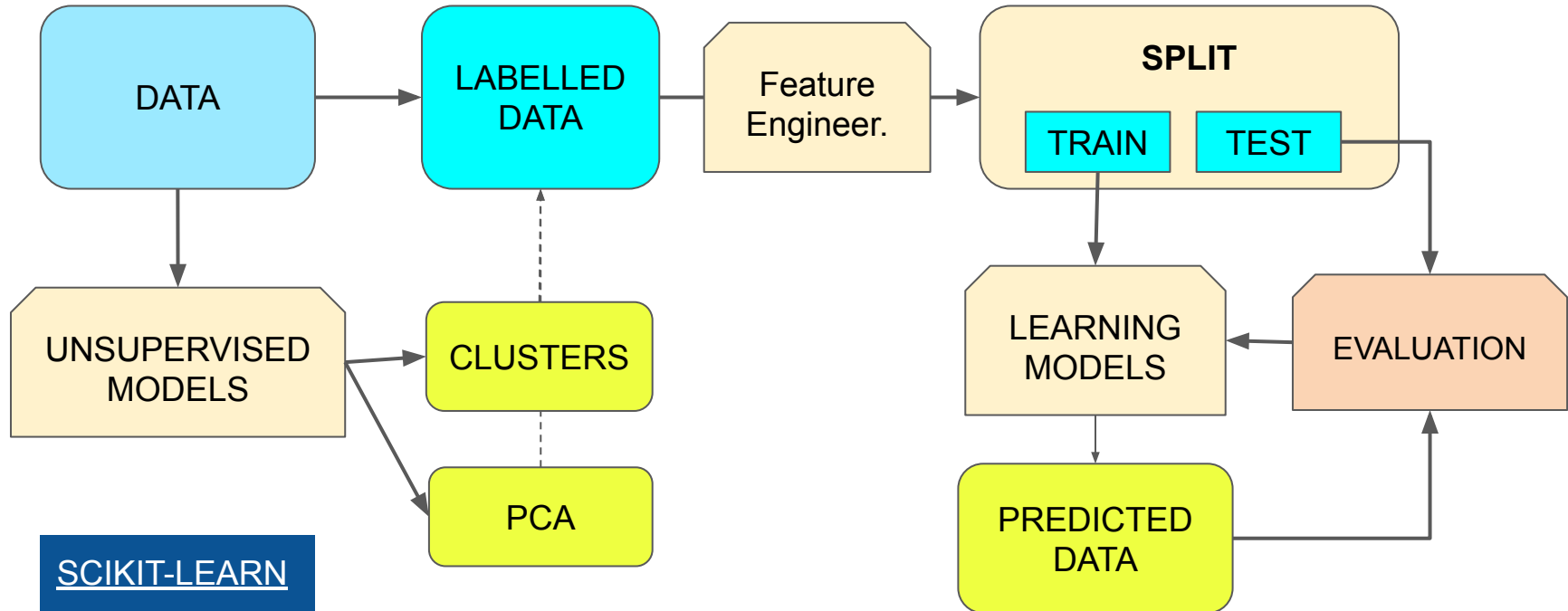
Distributing and scaling up  
machine learning algorithms

Scalable Machine Learning with  
Spark:

- Building estimators from RDDs
- Building estimators from DFs

Taking profit of massive parallelism  
of Spark to cluster, train and predict  
with machine learning algorithms.

# Machine Learning at a glance



# What is Spark ML

Library of Spark that gives support to Machine Learning methods:

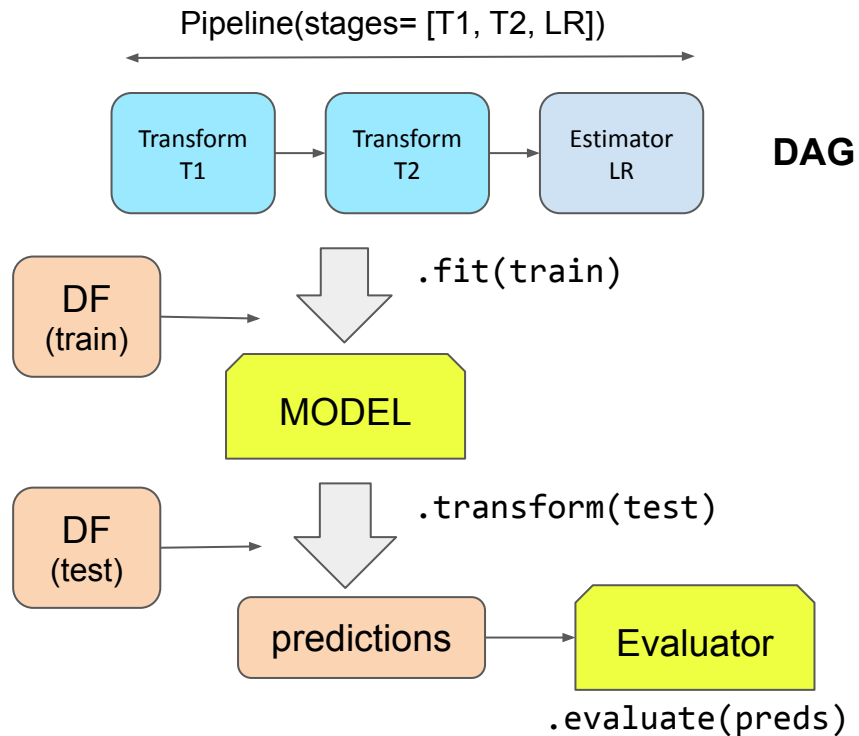
- **Vectors** and matrices ([spark.ml.linalg](http://spark.ml.linalg)) [see [Vectors](#)]
- **Statistics** and correlations ([spark.ml.stat](http://spark.ml.stat))
- **Feature** transformations ([spark.ml.feature](http://spark.ml.feature)) [from DFs to Features]
  - PCA, Binarizer, OneHotEncoder, VectorIndexer, Normalizer, \*Scaler
  - Tokenizer, TF, IDF, etc.
- **Classification** models ([spark.ml.classification](http://spark.ml.classification))
- **Clustering** models ([spark.ml.clustering](http://spark.ml.clustering))
- Cross validation & parameter **tuning** ([spark.ml.tuning](http://spark.ml.tuning))
- **Evaluation** ([spark.ml.evaluation](http://spark.ml.evaluation))

# Spark ML Pipelines

Originally for RDDs (MLlib), now built on Data Frames (DF) -Spark 3.0-

DF-based [ML components](#):

- Transformers (features)
- Estimators (fit)
- Pipelines (Pipeline)
- Parameters (Param)
- [Evaluators](#)



# How Spark ML works

