

# MapReduce & Spark

**CS594: Big Data Visualization & Analytics**

**Fabio Miranda**

**<https://fmiranda.me>**

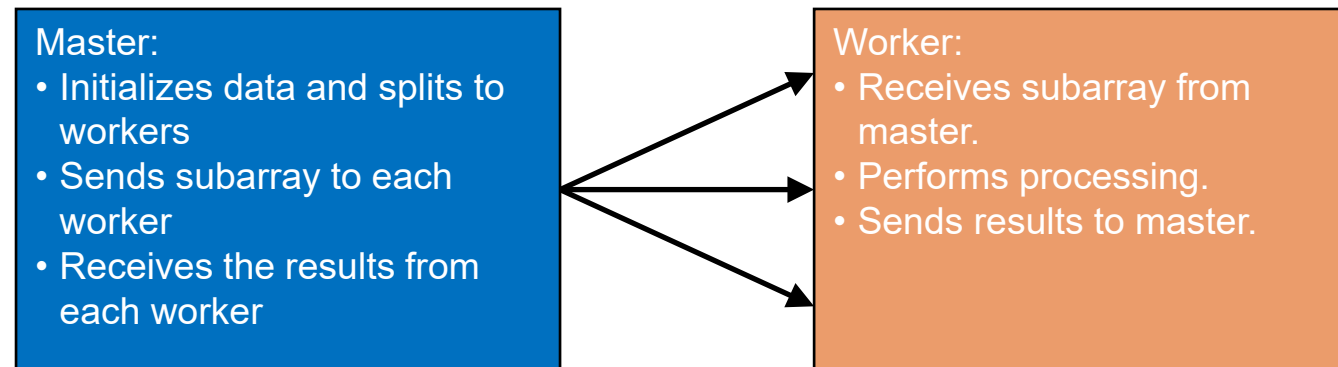
# Background

---

- Traditional programming is serial.
- Parallel programming: breaks processing into parts that can be executed concurrently on multiple processors.
- Challenge: identify tasks or groups of data that can be processed concurrently.

# Background

- Simple environment for parallel processing?
  - No data dependency.
  - Data can be split into smaller chunks.
  - Each process can work on a different chunk.
- Master / worker approach:



# Motivation: large-scale data processing

- Several tasks process lots of data to produce other data.
  - Easily parallelized to hundreds or thousands of CPUs.
  - Needs to be easy!
- MapReduce: programming model for processing big data.
- Main ideas:
  1. Abstraction
  2. Scale out vs. scale up
  3. Fault tolerance
  4. Move processing to data
  5. Avoid random access

# Idea 1: abstraction

- Finding the right level of abstraction:
  - Hyde system-level details from the developers.
  - No more race conditions, lock contention, etc.
- MapReduce separates the *what* from *how*:
  - Developer specifies the computation that needs to be performed.
  - Execution framework handles actual execution.
  - Inspired by LISP – functional programming.

# Idea 2: scale out vs. scale up

- Scale up: small number of high-end servers.
  - Large shared memory.
  - Not cost effective: cost of machine does not scale linearly; no single machine is big enough.
- Scale out: large number of commodity low-end servers.
  - 8 128-core machines vs. 128 8-core machines.

*“Low-end server platform is about 4 times more cost efficient than a high-end shared memory platform from the same vendor”*

*Barroso and Hölzle, 2009*

# Idea 3: fault tolerance

---

- Suppose a cluster is built using machines with a mean-time between failures (MTBF) of 1,000 days.
- For a 10,000-server cluster, there are on average 10 failures per day.
- MapReduce copes with failures:
  - Automatic task restarts.
  - Store files multiple times for reliability.

# Idea 4: move processing to data

- HPC: often have processing nodes and storage nodes.
  - Computationally expensive tasks.
  - High-capacity connection to move data around.
- Many data-intensive applications are not processor demanding.
  - Data movement leads to a bottleneck in the network.
  - Idea: move processing to where the data reside.
- MapReduce: processors and storage are co-located, leveraging locality.



# Idea 5: avoid random access

- Disk seek times are determined by mechanical factors.
- Example:
  - 1 TB database containing  $10^{10}$  100 byte records.
  - Random access: each update takes ~30 ms (seek, read, write).
  - Updating 1% of the records takes ~35 days.
  - Sequential access: 100 MB/s throughput
  - Reading the whole database and rewriting all records takes 5.6 hours.
- MapReduce was designed for batch processing: organize computations into long streaming operations.

# Large-scale problem

- Data: large number of records.
- Naïve solution:
  - Iterate over a large number of records.
  - Extract something of interest from each.
  - Sort and shuffle intermediate results.
  - Aggregate intermediate results.
  - Generate final output.

MapReduce key idea: provide a functional abstraction for these operations

# Large-scale problem in MapReduce

- Data: large number of records.
- MapReduce solution:
  - Iterate over a large number of records.
  - **Map**: Extract something of interest from each.  
$$\text{map}(k, v) \rightarrow \langle k', v' \rangle^*$$
  - Group by key: sort and shuffle intermediate results.
  - **Reduce**: Aggregate intermediate results.  
$$\text{reduce}(k', \langle v' \rangle^*) \rightarrow \langle k', v'' \rangle^*$$
  - Generate final output.

Structure remains the same, map and reduce functions change to fit the problem.

# MapReduce

- Map:
  - Grab the relevant data from the source.
  - User function gets called for each chunk of input.
  - Spits out (key, value) pairs.
- Reduce:
  - Aggregate the results.
  - User function gets called for each unique key with all values corresponding to that key.

## Simple example: map

```
var a = [1, 2, 3];

for(var i=0; i<a.length; i++){
    a[i] = a[i] * 2;
}

for(var i=0; i<a.length; i++){
    console.log(a[i]);
}
```

```
var a = [1, 2, 3];

function map(f, a){
    for(var i=0; i<a.length; i++){
        a[i] = f(a[i]);
    }
}

map(function(x){return x * 2;}, a);
map(alert, a);
```

## Simple example: reduce

```
var nums = [1, 2, 3, 4];

function sum(a){
  var sum = 0;
  for(var i=0; i<a.length; i++){
    sum += a[i];
  }
  return sum;
}

function mult(a){
  var mult=1;
  for(var i=0; i<a.length; i++){
    mult *= a[i];
  }
  return mult;
}

console.log(sum(nums));
console.log(mult(nums));
```

```
var nums = [1, 2, 3, 4];

function reduce(f, a, init){
  var s = init;
  for(var i=0; i<a.length; i++){
    s = f(s, a[i]);
  }
  return s;
}

function add(a, b){
  return a+b;
}

function mult(a, b){
  return a*b;
}

console.log(reduce(add, nums, 0));
console.log(reduce(mult, nums, 1));
```

# Moving towards MapReduce

- Adapt to a restricted model of computation.
- Goals:
  - Scalability: more machines, algorithm run faster.
  - Efficiency: resources will not be wasted.
- Translating some algorithms into MapReduce isn't obvious.
  - Think in terms of map & reduce.
  - Decompose complex algorithms into a sequence of jobs.

# Moving towards MapReduce

- Ideal scenario:
  - Twice the data, twice the running time.
  - Twice the resources, half the running time.
- Why can't we achieve this?
  - Synchronization requires communication.
- Big idea: avoid communication.
  - Reduce intermediate data via local aggregation.



# MapReduce: complete picture

- Map -  $map(k, v) \rightarrow \langle k', v' \rangle^*$ 
  - (input shard)  $\rightarrow$  intermediate(key / value pairs)
  - Automatically partition input data into M shards.
  - Generate (key, value) sets.
  - Groups together intermediate values with the same intermediate key & pass them to the reduce function.
- Reduce -  $reduce(k', \langle v' \rangle^*) \rightarrow \langle k', v'' \rangle^*$ 
  - Intermediate(key / value pairs)  $\rightarrow$  result files
  - Input: key \* set of values.
  - Merge these values together to form a smaller set of values.

# MapReduce: step by step

- Step 1: split input into chunks (shards)
- Step 2: fork processes
- Step 3: run map tasks
- Step 4: intermediate files
- Step 5: sorting
- Step 6: reduce
- Step 7: result

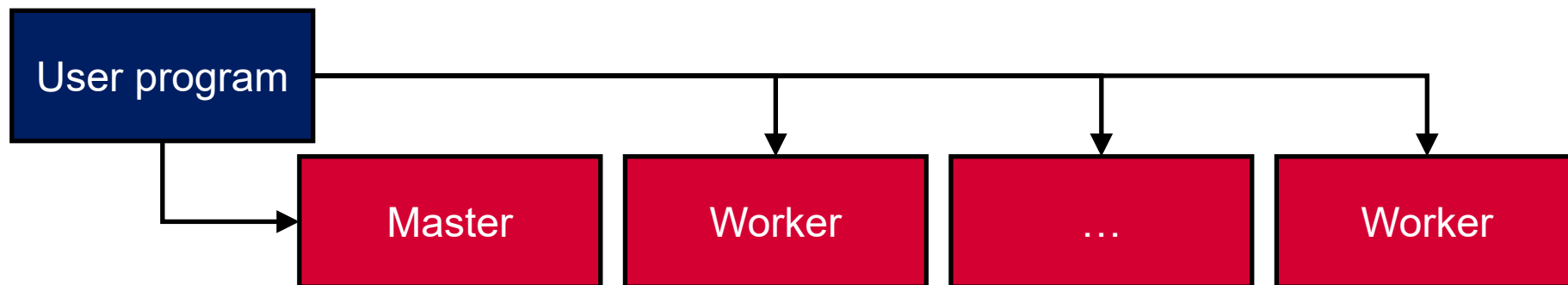
# Step 1: split input into chunks (shards)

- Break data into  $M$  smaller pieces.



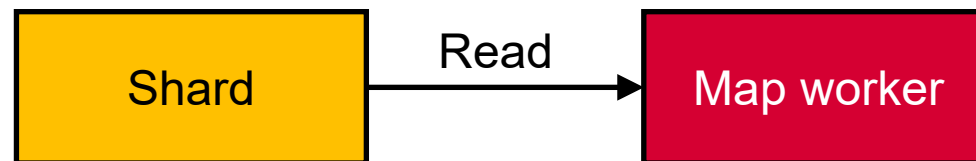
## Step 2: fork processes

- Start up many copies of the program on a cluster of machines.
  - One master to schedule & coordinate.
  - Lots of workers.
- Idle workers are assigned to:
  - **Map tasks**, each working on a shard: M map tasks
  - **Reduce tasks**, each working on intermediate files: R reduce tasks



## Step 3: run map tasks

- Reads input shard assigned to it.
- Parses key / value pairs of the input data.
- Passes *each* pair to a user-defined map function.
  - Produces intermediate key / value pairs.
  - Buffered in memory.



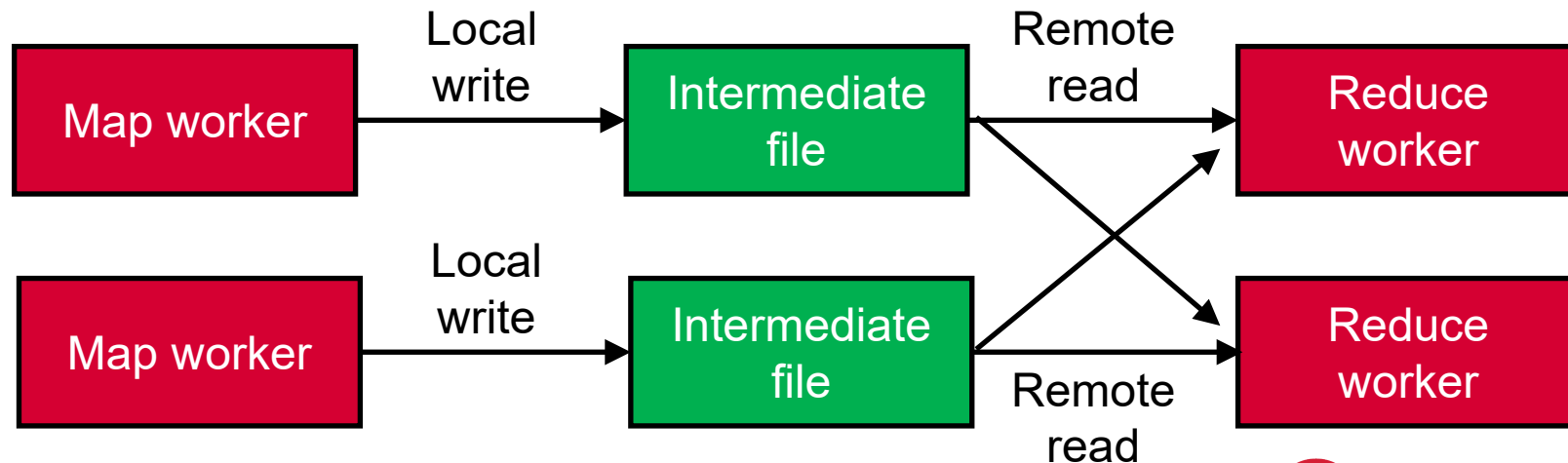
## Step 4: intermediate files

- Intermediate key / value pairs produced by user's map function are buffered in memory and written to local disk.



# Step 5: sorting

- Reduce workers access intermediate files.
- Sort: reduce workers read intermediate data.
  - Sorts data by intermediate keys.
  - All occurrence of the same key are grouped together



## Step 6: reduce

- Step 5 (sort) groups data with a unique intermediate key.
- User's reduce function is given the key and the set of intermediate values for that key.

$$\text{reduce}(k', \langle v' \rangle^*) \rightarrow \langle k', v'' \rangle^*$$

- Output of the reduce function is appended to an output file.

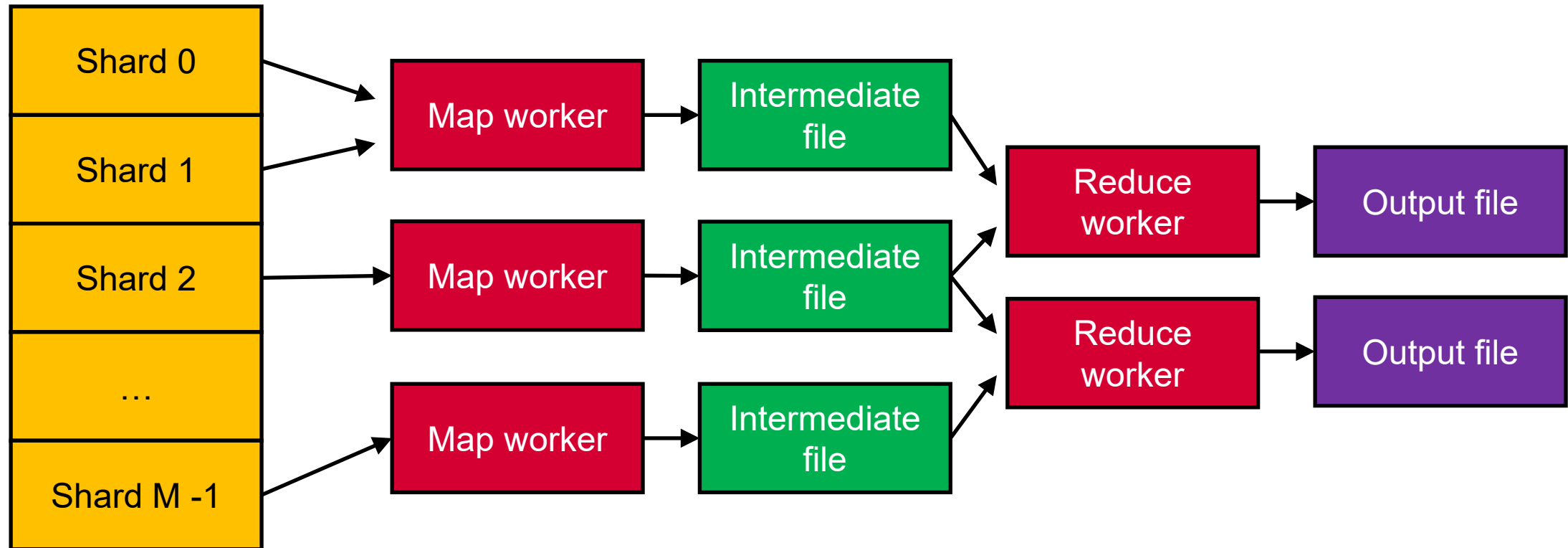




## Step 7: result

- When all map and reduce tasks have completed, master starts user program.
- Output of MapReduce is available in output files.

# MapReduce: complete picture

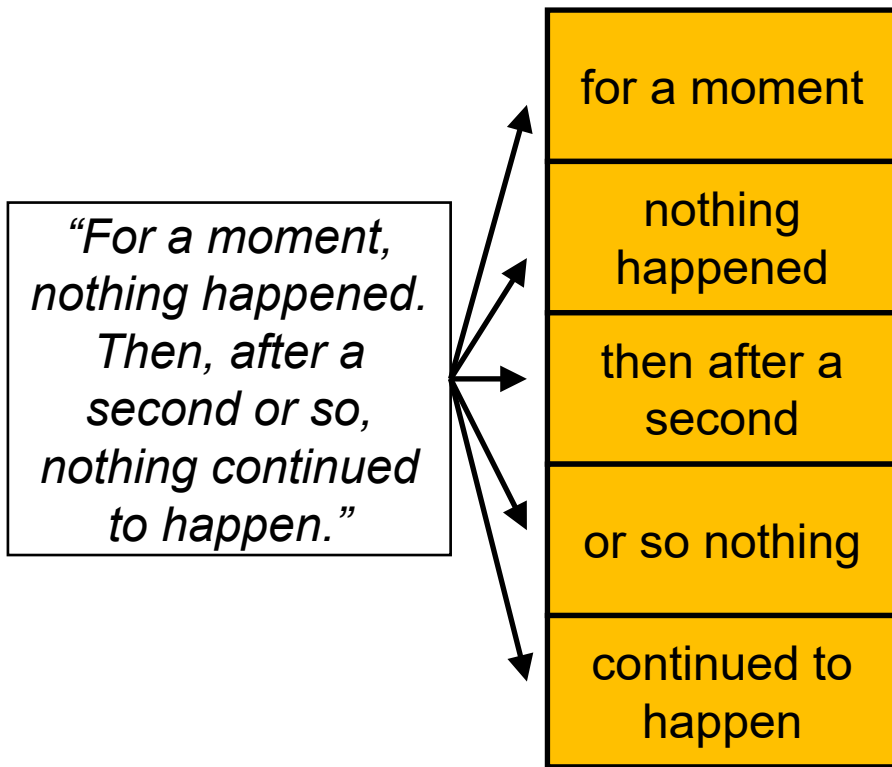


# MapReduce word count example

- Simple example: word count.
- Two programs:
  - mapper.py
  - reducer.py

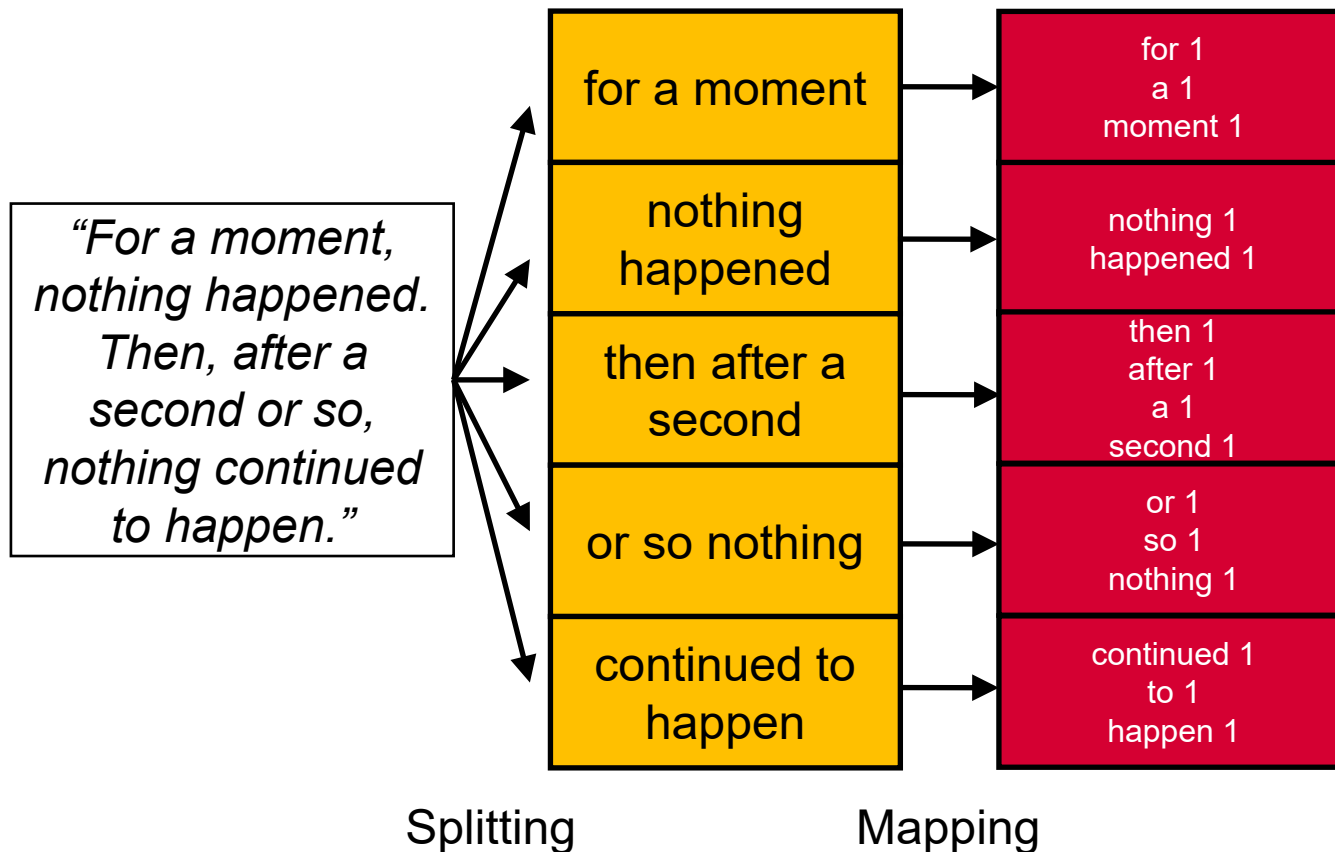
*“For a moment, nothing happened. Then, after a second or so, nothing continued to happen.”*

# MapReduce word count example

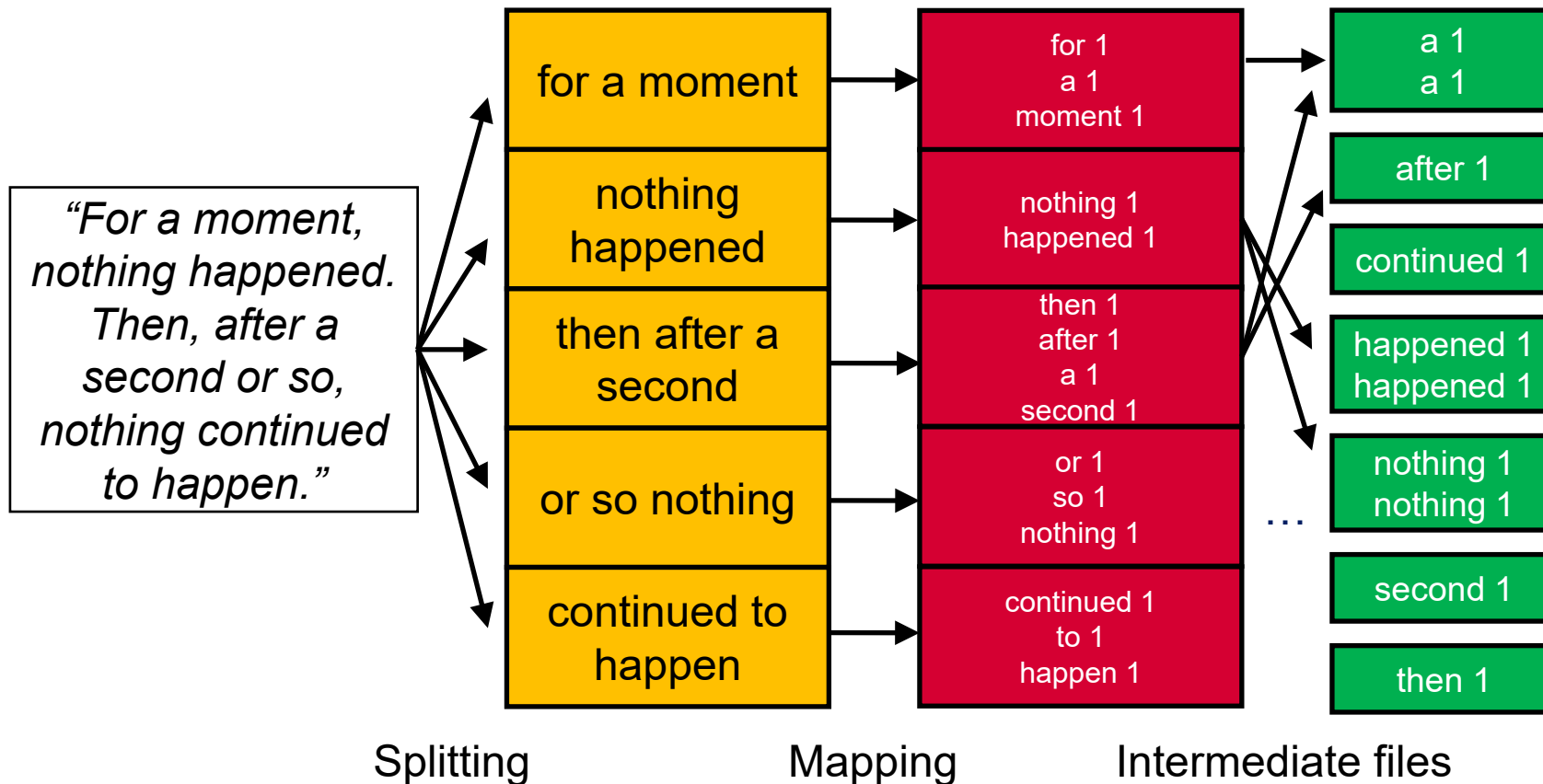


Splitting

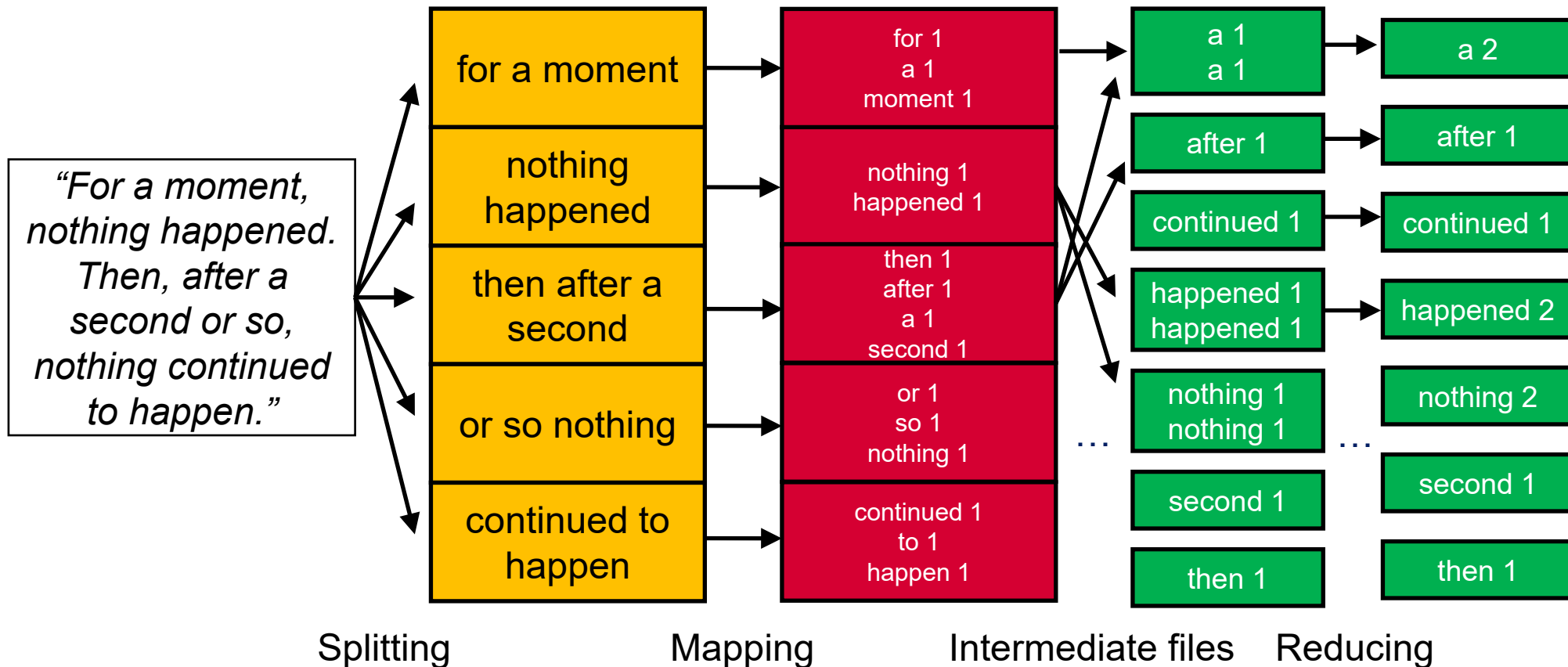
# MapReduce word count example



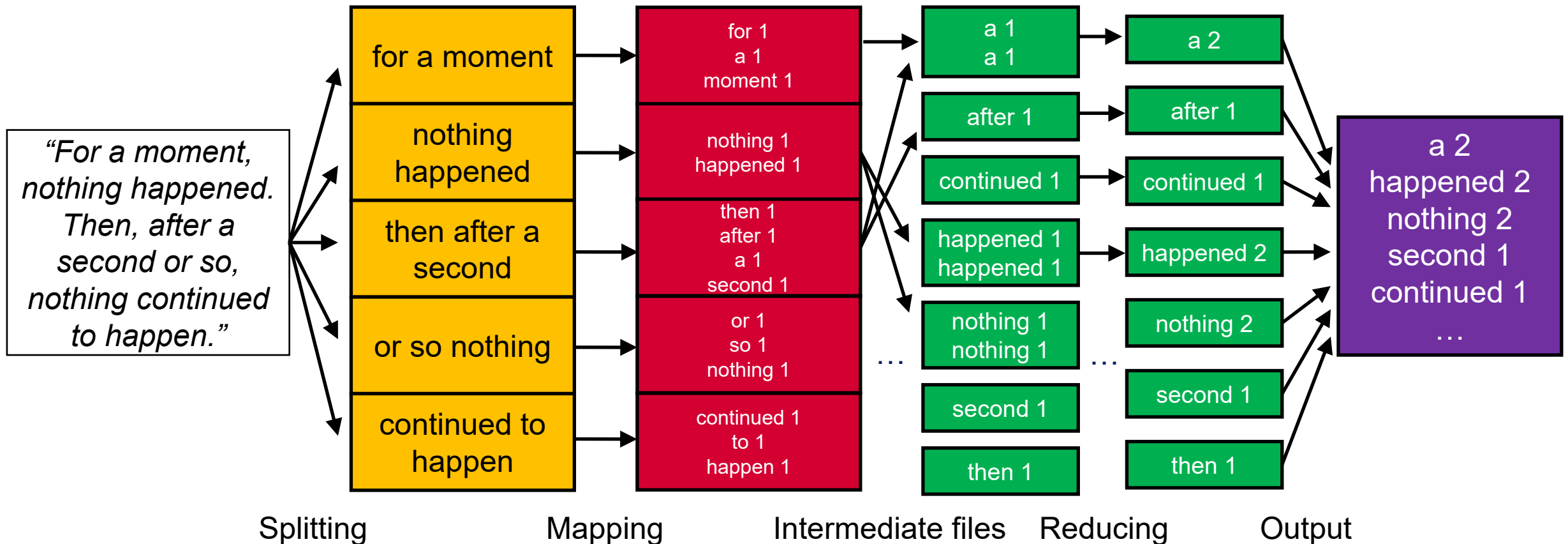
# MapReduce word count example



# MapReduce word count example



# MapReduce word count example





# MapReduce word count example

mapper.py

```
import sys
for line in sys.stdin:
    words = line.strip().split()
    for word in words:
        print('%s\t%s'%(word, 1))
```

reducer.py

```
import sys
cur_word = None
cur_count = 0
word = None
for line in sys.stdin:
    word, count = line.strip().split('\t',1)
    count = int(count)
    if cur_word == word:
        cur_count += count
    else:
        if cur_word:
            print('%s\t%s'%(cur_word, cur_count))
            cur_count = count
            cur_word = word
        if cur_word == word:
            print('%s\t%s'%(cur_word, cur_count))
```

# MapReduce word count example

mapper.py

```
import sys
for line in sys.stdin:
    words = line.strip().split()
    for word in words:
        print('%s\t%s'%(word, 1))
```

reducer.py

```
import sys
cur_word = None
cur_count = 0
word = None
for line in sys.stdin:
    word, count = line.strip().split('\t',1)
    count = int(count)
    if cur_word == word:
        cur_count += count
    else:
        if cur_word:
            print('%s\t%s'%(cur_word, cur_count))
            cur_count = count
            cur_word = word
        if cur_word == word:
            print('%s\t%s'%(cur_word, cur_count))
```

```
$ cat example.txt | python.exe map.py
for      1
a        1
moment   1
nothing  1
happened 1
then     1
after    1
a        1
second   1
or       1
so       1
nothing  1
continued 1
to       1
happen   1
```

# MapReduce word count example

mapper.py

```
import sys
for line in sys.stdin:
    words = line.strip().split()
    for word in words:
        print('%s\t%s'%(word, 1))
```

reducer.py

```
import sys
cur_word = None
cur_count = 0
word = None
for line in sys.stdin:
    word, count = line.strip().split('\t',1)
    count = int(count)
    if cur_word == word:
        cur_count += count
    else:
        if cur_word:
            print('%s\t%s'%(cur_word, cur_count))
            cur_count = count
            cur_word = word
        if cur_word == word:
            print('%s\t%s'%(cur_word, cur_count))
```

```
$ cat example.txt | python.exe map.py
for          1
a            1
moment      1
nothing     1
happened    1
then        1
after       1
a           1
second      1
or          1
so          1
nothing     1
continued   1
to          1
happen      1
```

```
$ cat example.txt | python.exe map.py |
sort -k1,1 | python.exe reduce.py
a          2
after      1
continued  1
for        1
happen     1
happened   1
moment     1
nothing    2
or         1
second     1
so         1
then       1
to         1
```

# Using Hadoop

- Hadoop is an Apache project that solves the problem of long data processing time.
- MapReduce is one part of Hadoop.

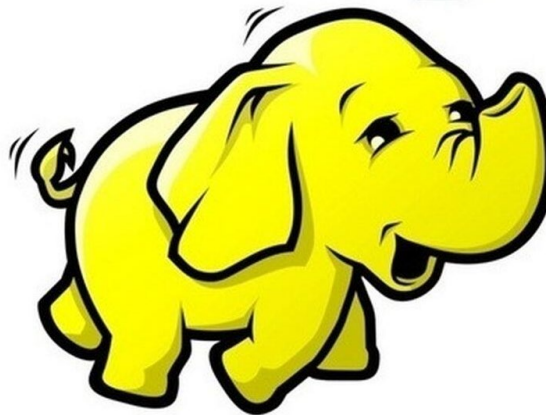
```
user@DESKTOP MINGW64 ~/example
$ hadoop jar contrib/hadoop-streaming-2.7.7.jar \
  -mapper mapper.py \
  -reducer reducer.py \
  -input ./*.txt \
  -output ./output/
```

# MapReduce summary

- MapReduce to handle large-scale problems.
- Map: parse & extract items of interest.
- Sort & partition.
- Reduce: aggregate results.
- Write to output files.

# MapReduce implementations

- Google: proprietary implementation in C++ (bindings in Java and Python).
- Hadoop: open-source implementation in Java.
  - Development led by Yahoo, now an Apache project.

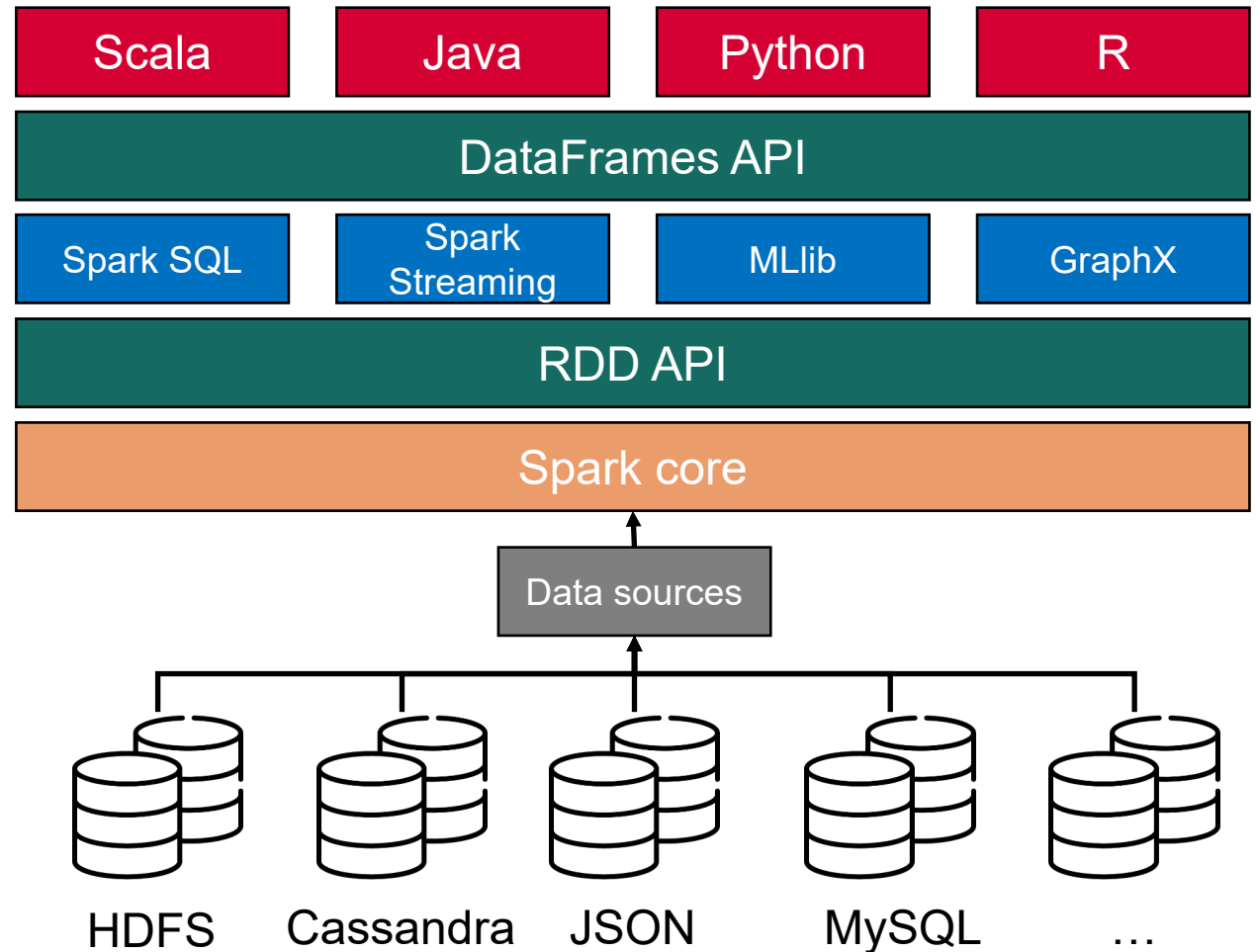


# Spark

- Hadoop relies on data distributed across nodes using on HDFS.
  - Master / worker architecture.
  - Out of memory approach.
- Spark to the rescue:
  - Processes data in RAM using RDD (Resilient Distributed Dataset).
  - Requires a lot of RAM to run in-memory (increased cluster cost in general).
- Hadoop: processes data in batches, reading and writing intermediate data to disk. Spark: in-memory analytics.

# Spark

- Spark unifies:
  - Batch processing
  - Real-time processing
  - Analytics
  - Machine learning
  - SQL



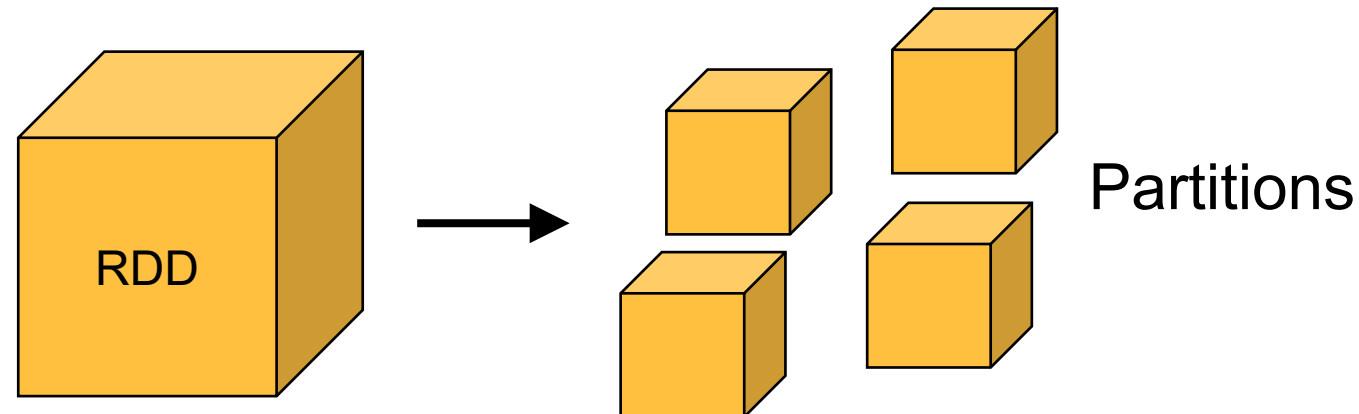


# Spark

- Framework designed for in-memory computation and fast performance.
- Performs different types of big data workloads:
  - Stream processing
  - Machine learning
  - Graph processing
- Easy to use high-level API:
  - Easily integrate with many libraries, including PyTorch and TensorFlow.

# Resilient distributed datasets

- Represents data or transformations on data.
- Distributed collection of items.
- Read only: immutable.
- Enables operations to be performed in parallel.



# Programming with RDDs

- Work is expressed either by:
  - Creating new RDDs.
  - Transforming existing RDDs.
  - Calling operations on RDDs to compute a result.
- Spark: distributes the data contained in RDDs across the nodes (executors) in the cluster, parallelizing the operations.
- Each RDD is split into multiple partitions, computed on different nodes of the cluster.

# RDD operations

- RDDs offer two types of operations:
  - **Transformations**
    - Map, filter, join, ...
    - Lazy operation to build RDDs from other RDDs
  - **Actions**
    - Count, collect, save, ...
    - Return a result or write it to storage

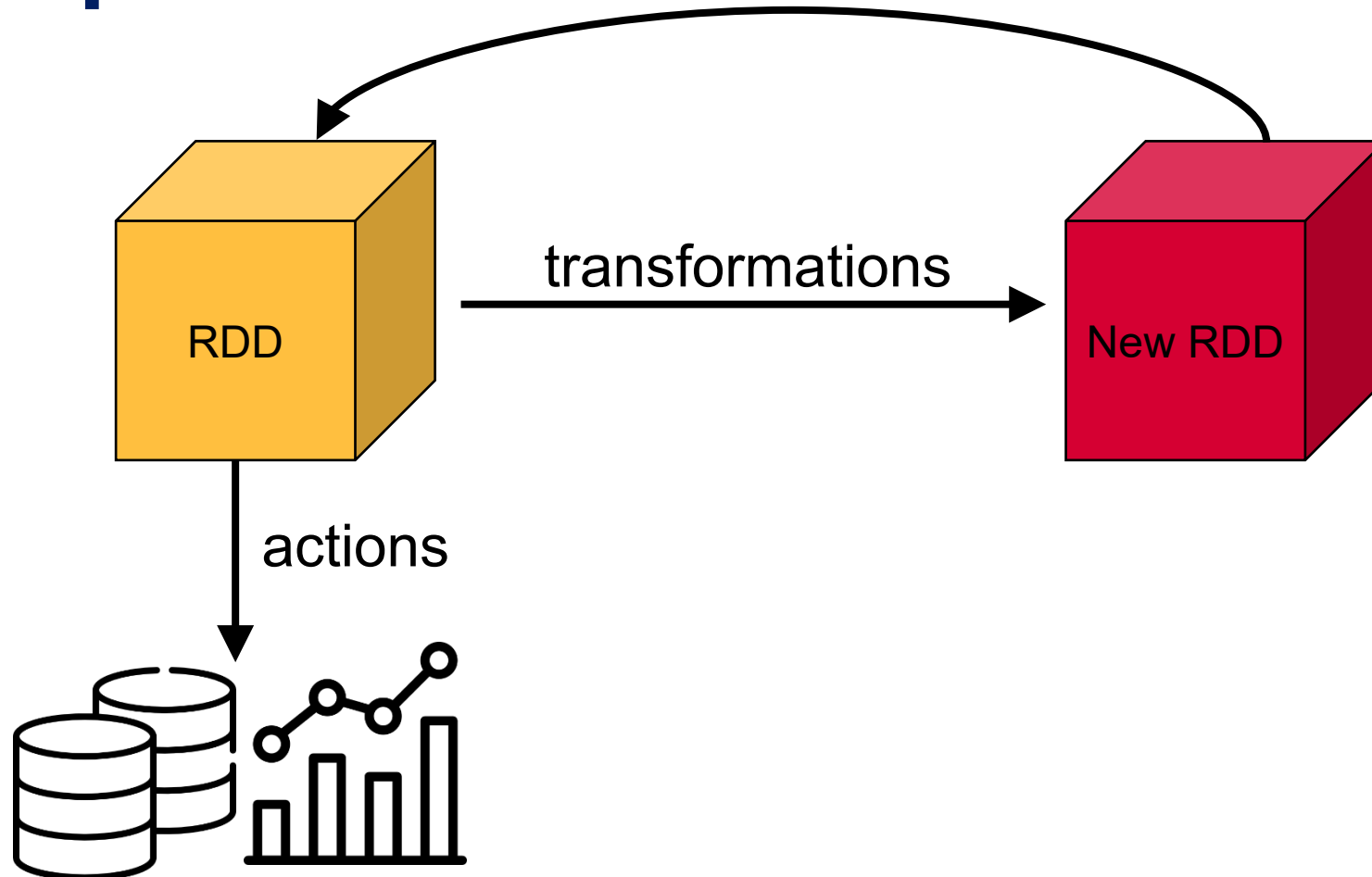
## Transformations

```
map(func)
flatMap(func)
filter(func)
groupByKey()
reduceByKey(func)
mapValues(func)
sample(...)
union(other)
distinct()
sortByKey()
...
```

## Actions

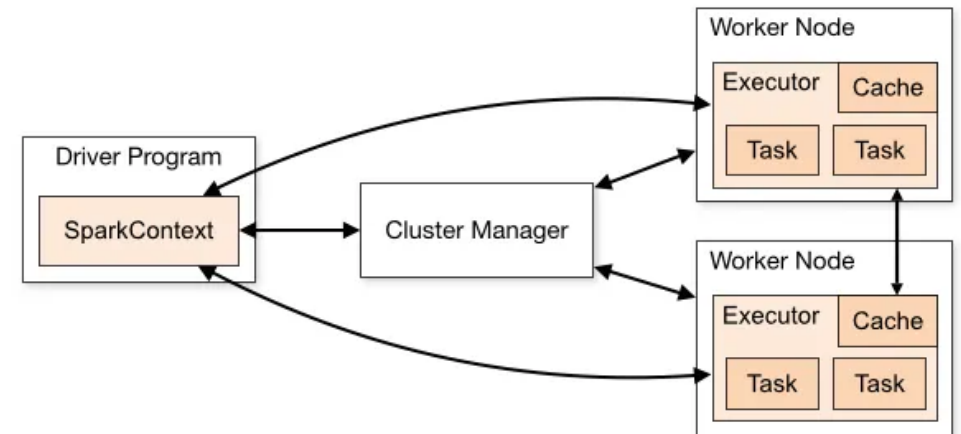
```
reduce(func)
collect()
count()
first()
take(n)
saveAsTextFile(path)
countByKey()
foreach(func)
...
```

# RDD operations



# Overview of a Spark program

1. **Create** input RDDs from external data or parallelize a collection.
2. Lazily **transform** them to define new RDDs using transformations.
3. Request Spark to **cache** any intermediate RDDs that will need to be reused.
4. Launch **actions** to start parallel computation, then executed by Spark.



# Spark initialization

```
from pyspark import SparkContext, SparkConf

conf = SparkConf().setAppName(appName).setMaster(master)
sc = SparkContext(conf=conf)
```

```
data = [1, 2, 3, 4, 5]
distData = sc.parallelize(data)
rdd.saveAsSequenceFile("path/to/file")
print(sorted(sc.sequenceFile("path/to/file").collect()))

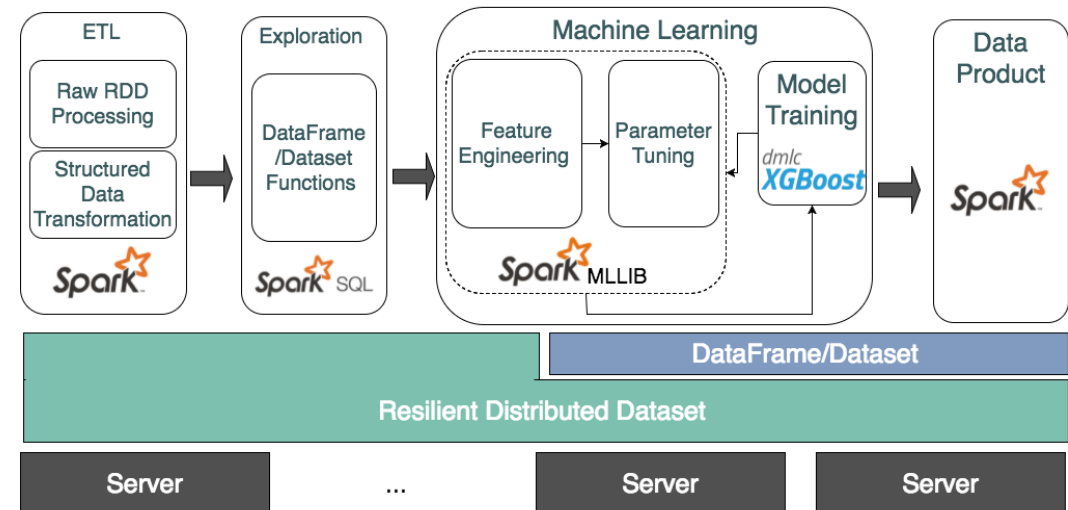
distFile = sc.textFile("data.txt")
```

Creates RDDs  
(lazily)

Action

# Spark and machine learning

- Spark's APIs interoperate with NumPy, and Hadoop data sources.
- Several algorithms and utilities:
  - Classification: logistic regression, ...
  - Linear regression
  - Random forest
  - K-means
- Machine learning workflows:
  - Feature transformation
  - Hashing





# Spark and Pandas

- Spark can easily access Panda's DataFrames.
- Pandas: data exploration with limited sample.
- Spark: large-scale analytics.

Pandas DataFrame	Spark DataFrame
Single Node	Multiple Nodes
Eager Execution – Computation is Executed Immediately	Lazy Execution – Computation isn't Executed Until Necessary which Allows Optimization of Tasks Across Cluster
Constraint by Computer Hardware	Scales Horizontally by Adding More Nodes
Computation Done In-Memory	Computation Done In-Memory
Data is Mutable	Data is Immutable
API Offers More Operations and Methods	Methods Require Additional Programming to Enable Parallel Computing

# Spark and DataFrames

Computing summaries with Spark and DataFrames:

```
from pyspark.sql.functions import sum, avg, count, first

ls = [['x', 'y', 3], ['x', 'y', 4], ['x', 'z', 3], ['y', 'y', 5]]
df = spark.createDataFrame(ls, schema=['A', 'B', 'C'])

group_df = df.groupby(['A', 'B'])
df_grouped = group_df.agg(sum("C").alias("sumC"),
                           avg("C").alias("avgC"),
                           count("C").alias("countC"),
                           first("C").alias("firstC"))

df_grouped.show()
```

# Spark and UDFs

Spark can evaluate user-defined functions (UDFs).

```
@udf
def to_upper(s):
    if s is not None:
        return s.upper()

@udf(returnType=IntegerType())
def mult(x):
    if x is not None:
        return x * 2

df = spark.createDataFrame([(1, "John Doe", 21)], ("id", "name", "age"))
df.select("age", mult("age")).show()
df.select("name", to_upper("name")).show()
```

# Taxi data using Spark

Loading the data and storing as parquet files:

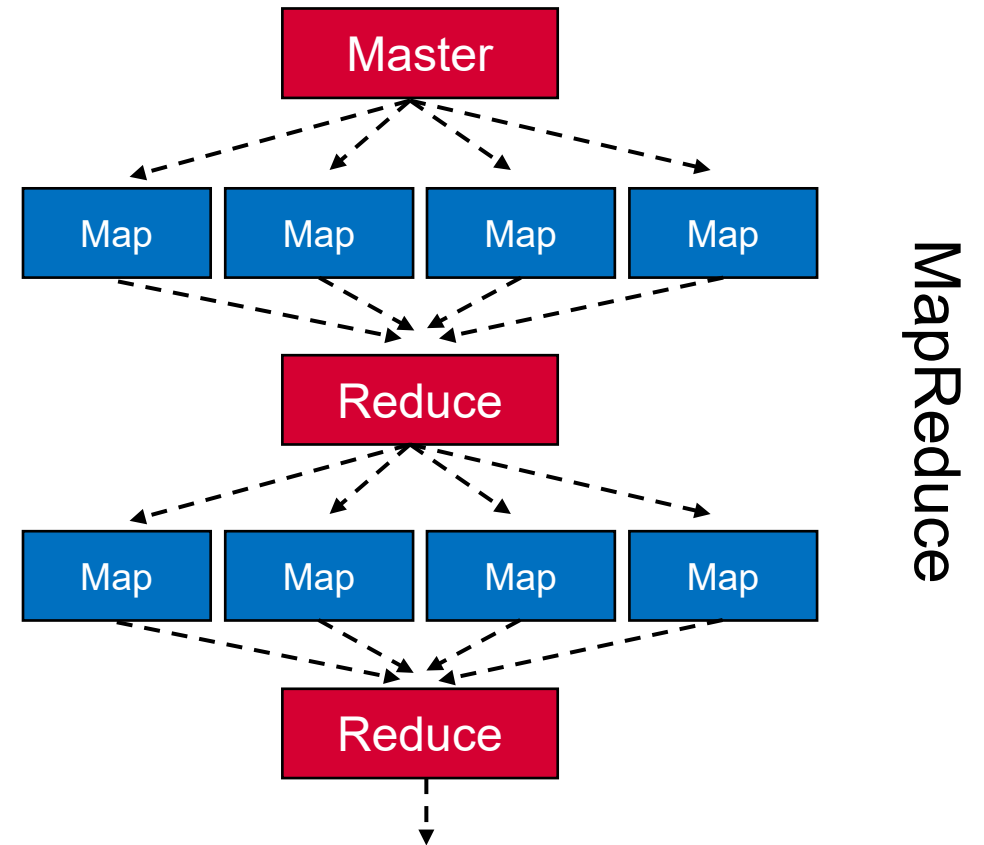
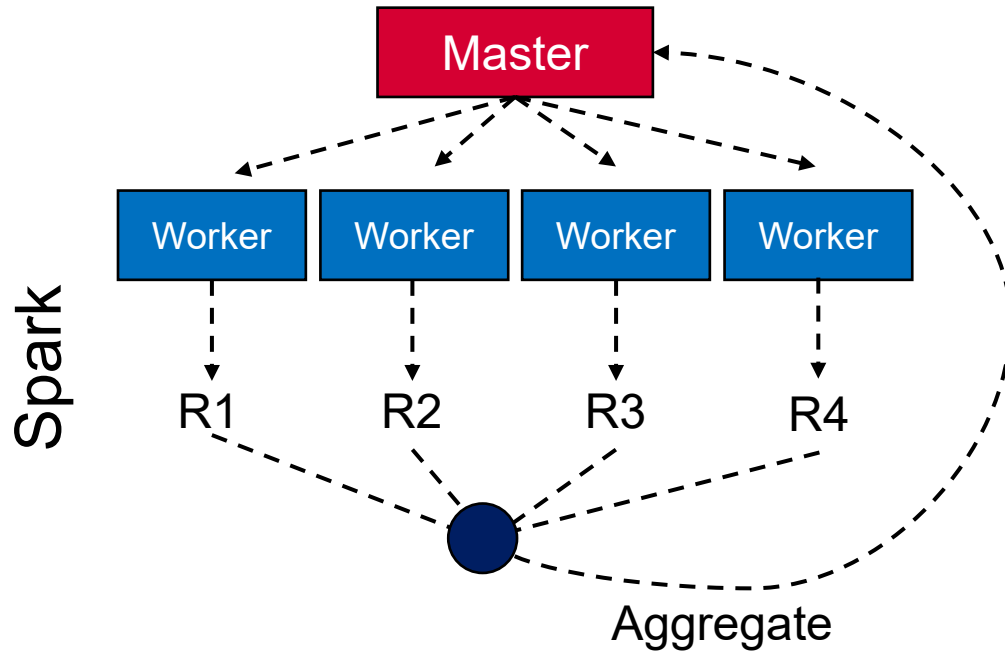
```
from pyspark.sql import SparkSession

conf = SparkConf().setAppName(appName).setMaster(master)
sc = SparkContext(conf=conf)

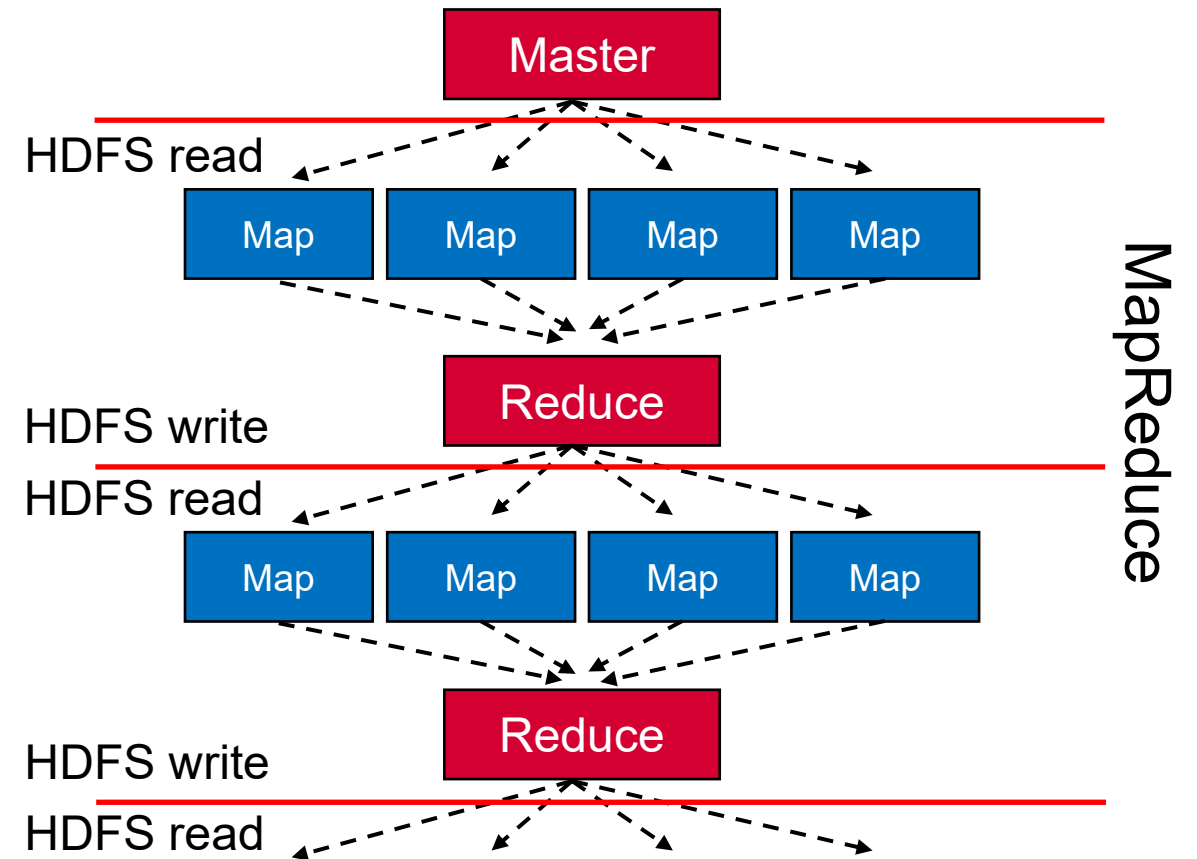
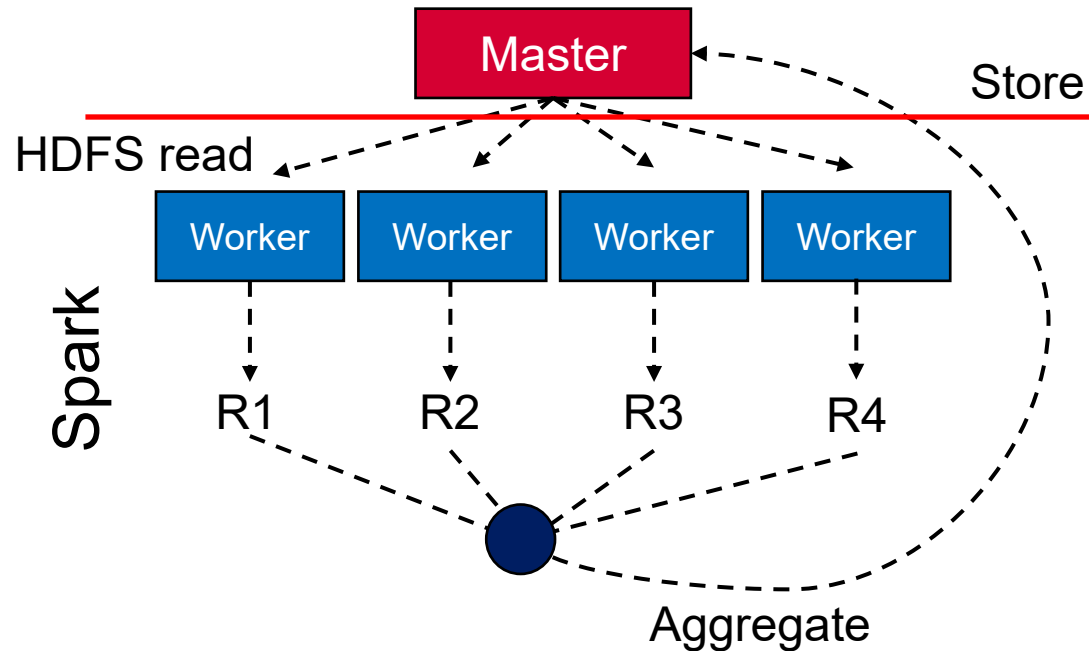
df = spark.read.csv("yellow*.csv", header="true", inferSchema="true")
df.write.parquet("2009-yellow.parquet")
spark.stop()
```

```
columns = ['passenger_count']
df = spark.read.parquet('2009-yellow.parquet').select(*columns)
df.agg({'passenger_count': 'sum'}).collect()
df.agg({'passenger_count': 'avg'}).collect()
df.groupby('passenger_count').agg({'*': 'count'}).collect()
```

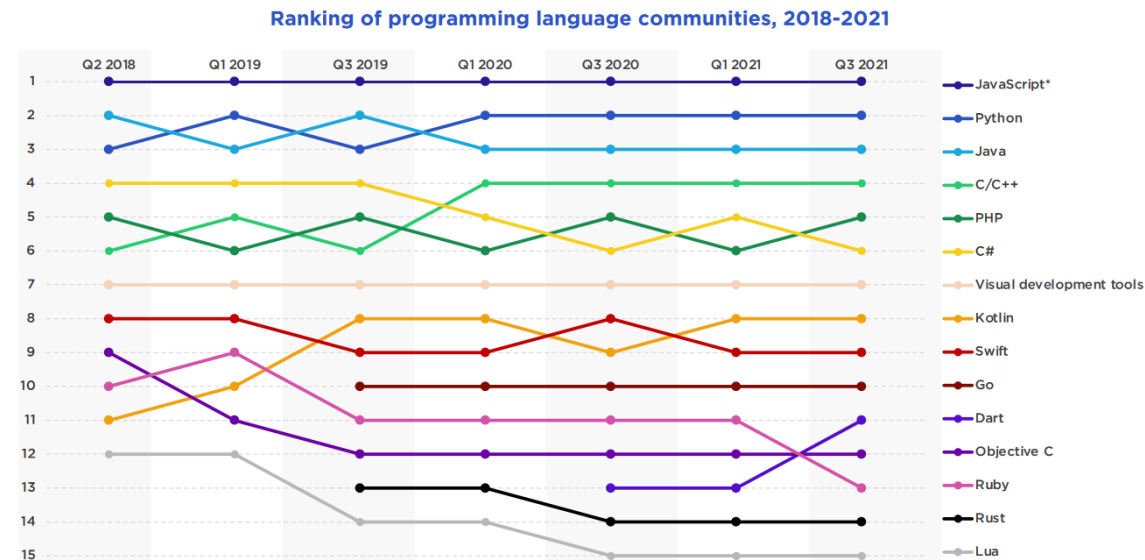
# Spark vs. MapReduce



# Spark vs. MapReduce



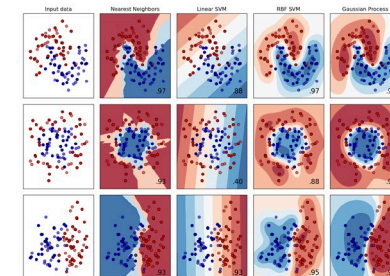
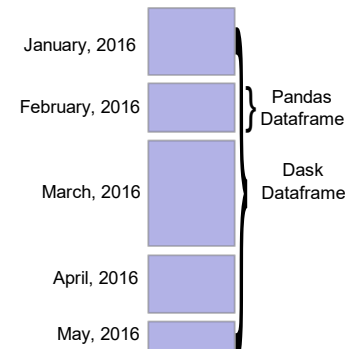
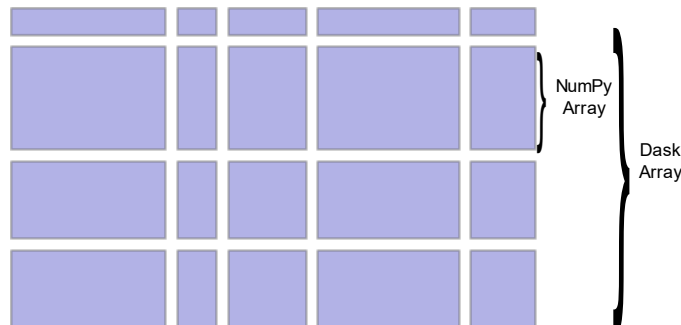
# Dask: parallel Pandas dataframe



- Python: leading platform for analytics and data science.
- Major libraries fail for big data or scalable computing.
- Python library for parallel computing:
  - Scales Numpy, Pandas, and Scikit-Learn
- Dask accelerates existing Python ecosystems.

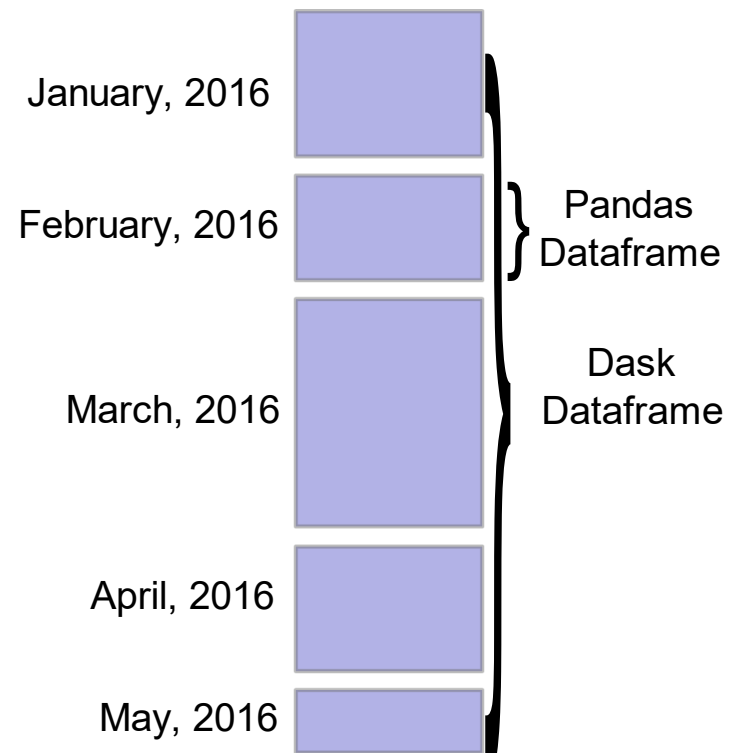
# Dask

- “Hadoop / Spark for Python”.
- Platform to build distributed applications.
- Dask can scale scikit-learn, Pandas and NumPy workflows:
  - Extends existing Python libraries
  - Enables easy transition for users
  - Leverages existing work, rather than reinvests the wheels.





# Dask



```
import pandas as pd

df = pd.read_csv("file.csv")
df.groupby("x").y.mean()
```

```
import dask.dataframe as dd

df = dd.read_csv("*.csv")
df.groupby("x").y.mean()
```