

MapReduce for Big Data

Examples by PySpark

- Compiled by: [Mahmoud Parsian](#)
- Last updated: 2025-09-18

Table of Contents

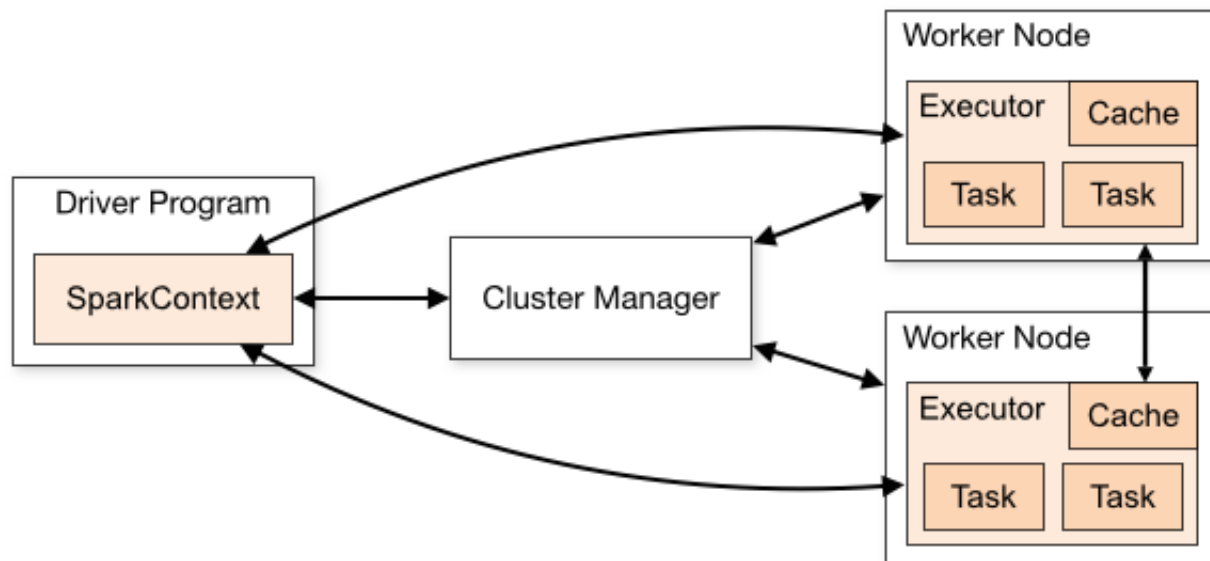
1. What is MapReduce?
2. How Does MapReduce Work?
3. Key Characteristics of MapReduce
4. MapReduce by an Example
5. MapReduce in Action
6. What is an Apache Spark
7. What is PySpark?
8. Teach MapReduce with using PySpark
9. 5 Simple MapReduce Examples with PySpark
10. 5 Intermediate MapReduce Examples with PySpark
11. 2 Complex MapReduce Examples with PySpark
12. References

1. What is MapReduce?

1. MapReduce is a programming model for expressing distributed computations on massive amounts of data and an execution framework for large-scale data processing on clusters of commodity servers.

source : [Data-Intensive Text Processing with MapReduce, Jimmy Lin]

Cluster of commodity servers



1. What is MapReduce? continued...

2. MapReduce was originally developed by Google and built on well-known principles in parallel and distributed processing dating back several decades.

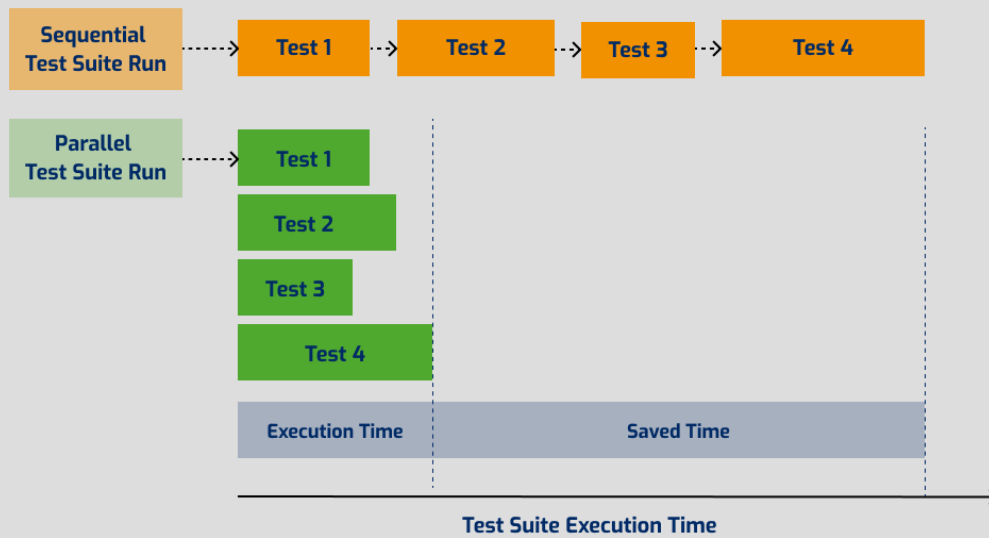
Sequential-processing-vs-parallel-processing

Sequential Computing vs. Parallel Computing

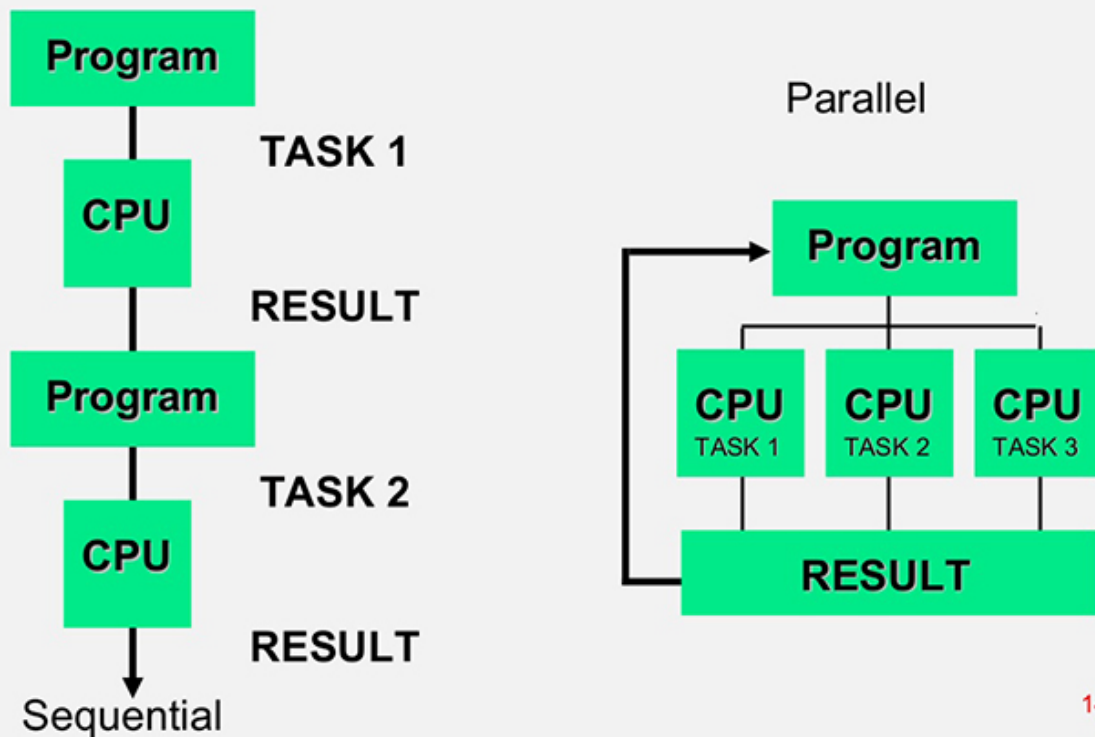
In sequential computing, operations are performed in order one at a time.

In parallel computing, the program is broken into smaller steps, some of which are performed at the same time. Modern computers have multiple processors (4, 8, 16, 32, ...) in a single computer, so you can do small-scale parallel processing on the machine on your desk.

Effect of Parallel Testing on Test Suite Execution Time



Sequential and parallel processing

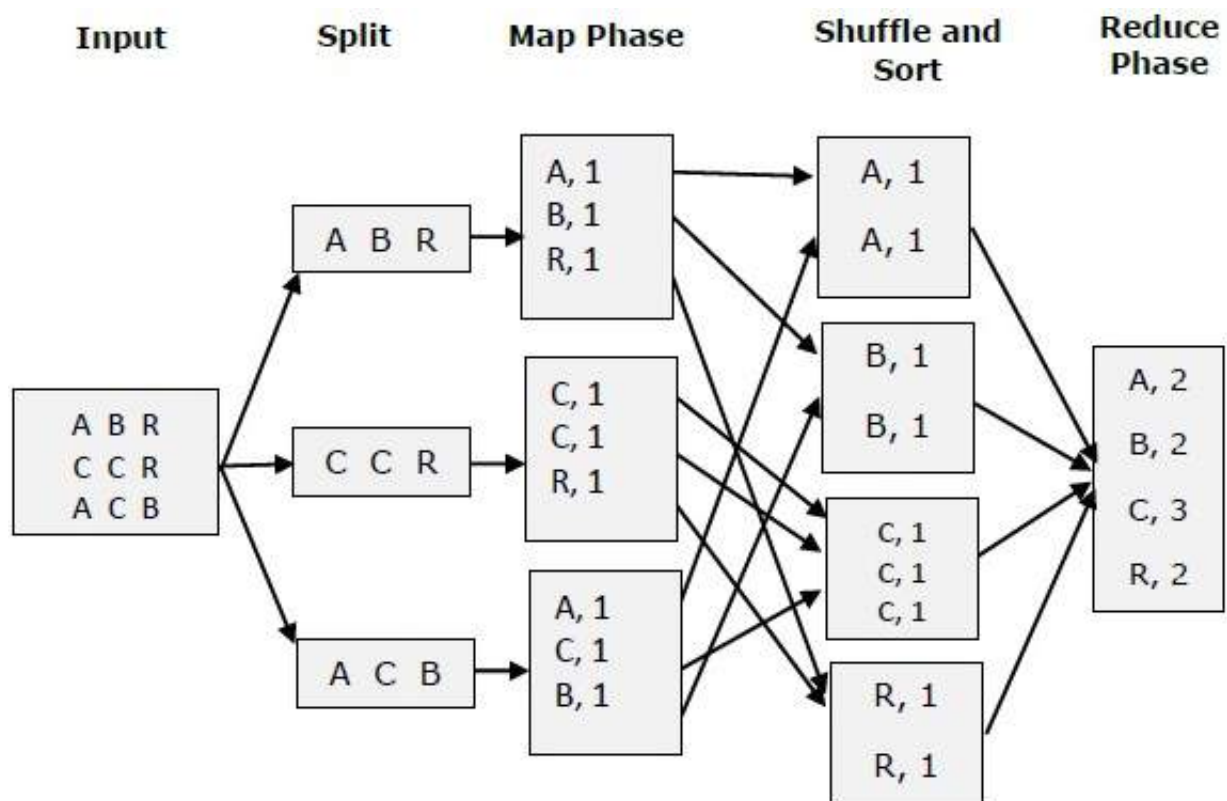


1. What is MapReduce? continued...

3. MapReduce is a programming model and execution framework, notably used in Apache Hadoop, for processing vast amounts of data in parallel across a large cluster of machines. It works by dividing data into smaller pieces for independent, parallel processing by "map" functions, then combining those results using "reduce" functions to produce the final output.

4. This approach simplifies distributed computing, allowing programmers to handle Big Data without deep expertise in parallel systems, making it suitable for tasks like data mining, search engine operations, and enterprise analytics.

MapReduce Architecture by Example



✓ 2. How Does MapReduce Work?

MapReduce follows a three-phase process to handle large datasets:

1. Map phase:

The input data is divided into fixed-size chunks and distributed across a cluster of machines.

A user-defined "map" function processes each chunk of data independently, transforming it into intermediate (key, value) pairs.

For example, if your input has N records, Then N (key, value)'s will be created: to be distributed/passed to mappers:

```
(key-1, record-1)
(key-2, record-2)
...
(key-N, record-N)
```

Mapper I/O:

```
input: (K, V)
output: [(k1, v1), (k2, v2), (k3, v3), ...]
```

Therefore a single (K, V) can be mapped/converted into 0, 1, 2, 3, ... (key, value) pairs.

If a given (K, V) to a mapper does not produce any new (key, value) pairs, then we say that a given input of (K, V) is filtered out.

Mapper I/O Note

```
if a mapper does not emit any
(key, value) output pairs, it means
that input is filtered out (ignored)
```

Mapper Example: Word Count

```
mapper input: a (K, V) pair as
               K = key for a pair such as a records number
               V = a single record as a string:
               "word1 word2 word3 ..."
```

```
mapper output: [(word1, 1), (word2, 1), (word3, 1), ...]
```

Consider a scenario where the goal is to count the occurrences of each word in a large text file.

Input to Map Phase: The input to the map phase is typically a (key, value) pair where the key is the byte offset of a line in the input file, and the value is the content of that line.

Input to mappers as (K, V) pairs:

```
(0, "The quick brown fox")
(20, "jumps over the lazy dog")
```

Map Function Output:

The map function processes each line,
tokenizes it into individual words, and
for each word, it emits a (key, value) pair
where:

Key: The word itself (e.g., "The", "quick", "brown", "fox").

Value: A count of 1, indicating a single occurrence of that word.

A Mapper input: (0, "The quick brown fox")

A Mapper output:

```
("The", 1)
("quick", 1)
("brown", 1)
("fox", 1)
```

A Mapper input: (20, "jumps over the lazy dog")

A Mapper output:

```
("jumps", 1)
("over", 1)
("the", 1)
("lazy", 1)
("dog", 1)
```

In this word count example, the word itself acts as the key during the map phase. This allows the MapReduce framework to group all occurrences of the same word together (e.g., all "the" keys will be sent to the same reducer) for aggregation in the subsequent reduce phase.

Note:

For the word count mapper, the keys
(such as 0 and 25) are not used at all
and are ignored.

2. Shuffle phase:

In MapReduce, the shuffle function refers to the phase where the intermediate (key, value) pairs generated by the Map tasks are prepared and transferred to the Reduce tasks. It is a crucial step that bridges the gap between the mapping and reducing phases.

Shuffle phase I/O:

```
input: [(K, v1), (K, v2), (K, v3), ...]  
output: (K, [v1, v2, v3, ...])
```

This is similar to the "GROUP BY" function in SQL.

3. Reduce phase:

All intermediate key/value pairs from the map phase are grouped by their shared keys.

A user-defined "reduce" function merges these grouped values for each key, performing operations such as aggregation or summation to produce the final output.

Reducer I/O

```
input: (k, [v1, v2, v3, ...])  
output: [(k1, u1), (k2, u2), (k3, u3), ...]
```

Reducer example-1: add values of a given key

```
input: (K, [1, 1, 1, 1])  
output: (K, 4)
```

Reducer example-2: find average of values of a given key

```
input: (K, [3, 6, 2, 8, 2])  
output: (K, 4.2)
```

where average = $(3 + 6 + 2 + 8 + 2) / 5 = 21 / 5 = 4.2$

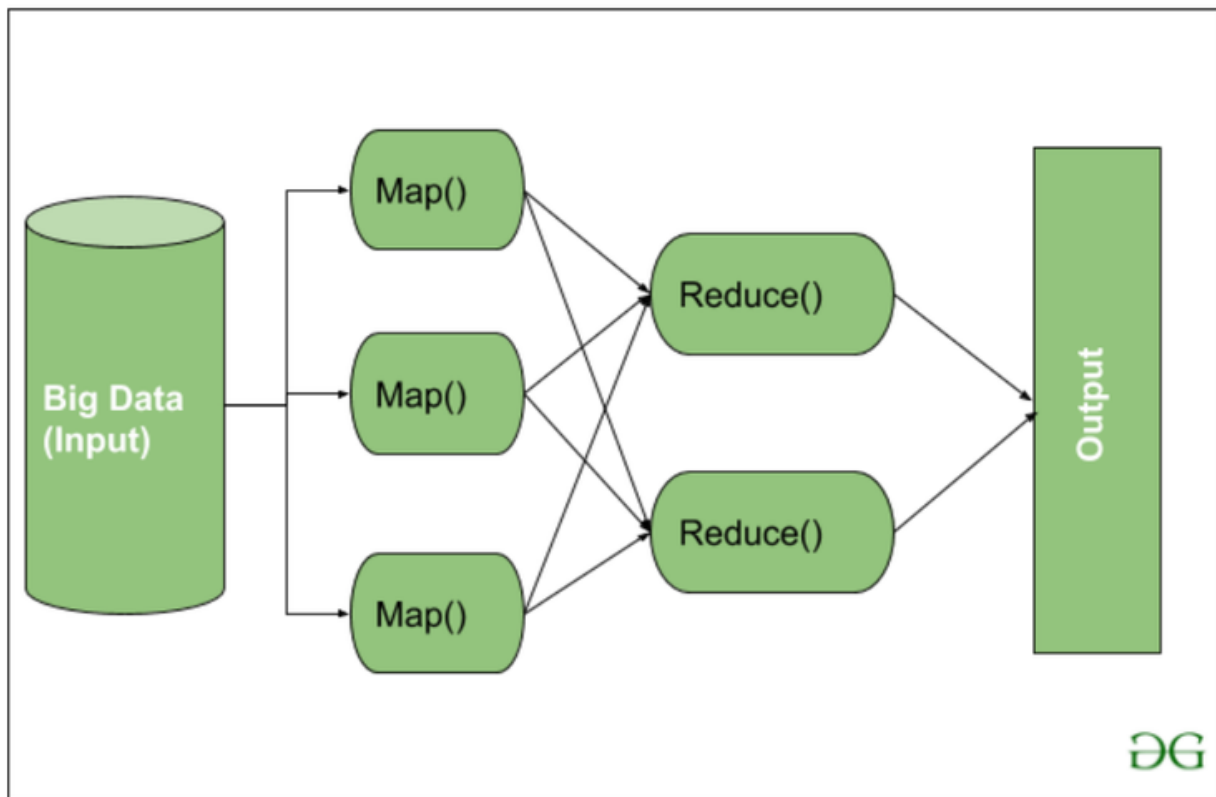
Reducer example-3: find median of values of a given key

```
input: (K, [3, 6, 2, 8, 2])  
output: (K, 3)
```

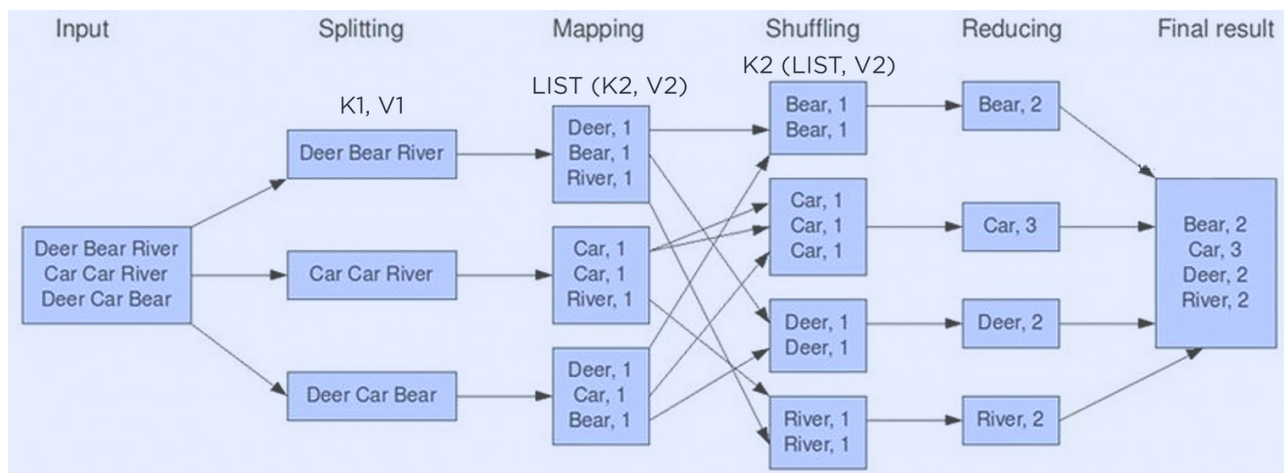
Sort values: $[3, 6, 2, 8, 2] \rightarrow [2, 2, 3, 6, 8]$

Find median: 3

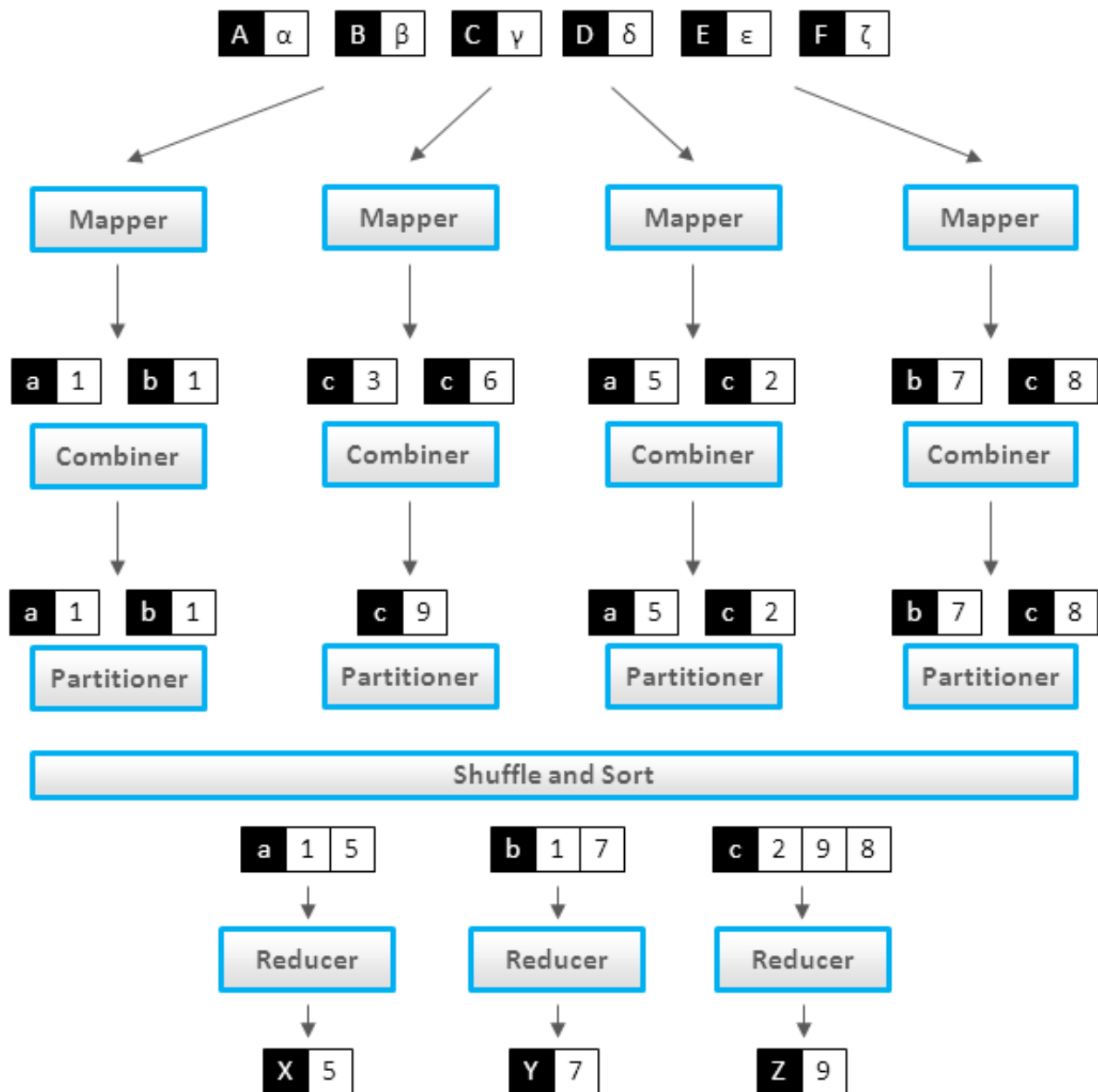
MapReduce Architecture



MapReduce: mappers, shuffling, reducers



MapReduce with Combiners



✓ 3. Key Characteristics of MapReduce

1. Parallel Processing:

Data is processed simultaneously across many machines, speeding up computation.

2. Fault Tolerance:

The framework automatically handles machine failures by distributing data and rerouting processes, ensuring job completion.

3. Scalability:

MapReduce can scale to thousands of machines to process petabytes of data.

4. Simplicity for Developers:

It hides the complexities of parallel and distributed systems, enabling developers to write programs for large-scale data without prior experience.

5. Applications

MapReduce is used across various industries for large-scale data processing, including:

- * **Search Engines:**
Creating and updating search indexes.
- * **Data Mining:**
Extracting insights and patterns from large datasets.
- * **Analytics:**
Processing data for enterprise-wide analysis.
- * **Finance:**
Validating data accuracy and preventing fraud.
- * **Healthcare:**
Storing and structuring large medical and genomic datasets.



4. MapReduce by an Example

MapReduce is a programming model designed for processing large datasets in a distributed computing environment. It involves two main phases: Map and Reduce, with an intermediate Shuffle phase.

Example: Word Count

Consider the task of counting the occurrences of each word in a large collection of documents distributed across multiple machines.

Input Data:

A set of text files, each containing words.

Map Phase:

The input files are split into smaller chunks, and each chunk is processed by a "mapper" function on a separate machine.

The mapper reads its assigned chunk of text. For each word encountered, the mapper emits a (key, value) pair: (word, 1). This signifies that the word has been found once.

Example: If a mapper processes the text "this is an apple", it would output:

```
(this, 1),  
(is, 1),  
(an, 1),  
(apple, 1)
```

Shuffle Phase:

The outputs from all mappers are collected. These (key, value) pairs are then grouped by key (the word). All values associated with the **same key** are sent to the same "reducer" function. This often involves transferring data between different machines.

Example:

All (apple, 1) pairs from all mappers are sent to the same reducer.

Reduce Phase:

Each reducer receives a key and a list of values associated with that key. The reducer aggregates these values to produce a final result for that key.

In the word count example, the reducer sums all the '1's in the list to get the total count for that word.

Example: A reducer receiving (apple, [1, 1, 1, 1]) would output (apple, 4).

Output:

The final output is a set of (key, value) pairs representing the total count of each unique word across all input documents.

Summary of Phases:

Map: Transforms input data into intermediate (key, value) pairs.

Shuffle: Groups intermediate (key, value) pairs by key.

Reduce: Aggregates values for each key to produce the final output.

This distributed approach allows for efficient processing of massive datasets by leveraging the power of multiple machines in parallel.

5. MapReduce in Action

Introduce MapReduce concepts to students
in my big data modeling class:

1. Assume we have a cluster of 101 nodes, where one node is a master, and other 100 are worker nodes. Master node is a manager and does not perform any computation of mappers and reducers.

Master node: cluster manager
Worker nodes: 100 nodes
2. given a cluster of 101 nodes (only 100 nodes are worker nodes) given an input of 800 TB, explain how mappers and reducers will be executed in this cluster,
3. assume that each node in the cluster can handle 10 mappers and 10 reducers.
4. Provide detailed explanation how this will happen and carried out.
5. I'll walk through exactly how MapReduce will run on a 100-node cluster with 800 TB input when each node can run
 - * 10 mappers and
 - * 10 reducers.
6. I'll show concrete arithmetic (step-by-step), explain scheduling, the shuffle, and practical tuning points you can use in class.

Executive summary (key numbers)

- Cluster concurrency (slots):
 $100 \text{ nodes} \times 10 \text{ mapper slots} = 1,000 \text{ mappers concurrently};$
 $100 \text{ nodes} \times 10 \text{ reducer slots} = 1,000 \text{ reducers concurrently}.$
- If HDFS/MapReduce uses 128 MB splits,
the job creates 6,553,600 map tasks,
which must run in $\approx 6,554$ waves of 1,000 concurrent mappers.
- With 1,000 reducers, each reducer would receive about
0.8 TB of intermediate data (assuming map output \approx input)
and must fetch $\sim 6,554$ map output files (for the 128 MB split case).

Phases, bottlenecks, and tuning.

1) How many mappers? (step-by-step arithmetic)

I'll use the common binary units:

1. $1 \text{ TB} = 1024 \text{ GB}.$
2. $1 \text{ GB} = 1024 \text{ MB} \rightarrow 1 \text{ TB} = 1024 \times 1024 \text{ MB} = 1,048,576 \text{ MB}.$
3. $800 \text{ TB} = 800 \times 1,048,576 \text{ MB} = 838,860,800 \text{ MB}.$

Now mapper counts for several common split sizes:

a) Split = 128 MB (Data Block Size = 128 MB)

- $\text{Mappers} = 838,860,800 \text{ MB} \div 128 \text{ MB}$
- $128 = 2^7,$
 $1,048,576 = 2^{20} \rightarrow 800 \times 2^{20} \div 2^7 =$
 $800 \times 2^{13} =$
 $800 \times 8192 = 6,553,600 \text{ mappers}.$

b) Split = 256 MB

- $\text{Mappers} = 838,860,800 \div 256 = 3,276,800 \text{ mappers}.$

c) Split = 512 MB

- $\text{Mappers} = 838,860,800 \div 512 = 1,638,400 \text{ mappers}.$

d) Split = 1 GB (1024 MB)

- $\text{Mappers} = 838,860,800 \div 1024 = 819,200$ mappers.

2) How they are scheduled (mapper waves & data locality)

- Concurrent mapper capacity =
 $100 \text{ nodes} \times 10 = 1,000$ mapper slots.
- If you have M total mappers,
the number of waves (sequential batches) = $\text{ceil}(M / 1,000)$.

Using 128 MB splits ($M = 6,553,600$):

- $\text{Waves} = 6,553,600 \div 1,000 = 6,553.6 \rightarrow \text{ceil} \rightarrow 6,554$ waves.
- That means the cluster runs ~1,000 mappers at a time,
repeats ~6,554 times until all map tasks have completed.

Data locality:

HDFS (Hadoop Distributed File System) places each block with (typical) 3 replicas. The scheduler will try to launch a mapper on a node that holds the block (node-local). However, because total map tasks vastly outnumber slots, some waves will lose locality (you'll see more rack-local or remote reads), which increases network IO and slows the job.

Teaching point:

high task count + limited slots \rightarrow locality suffers.

Stragglers and speculative execution:

with millions of tasks, some will be slow or fail;
speculative execution re-runs the slow tasks on other nodes and picks the winner – helpful but can increase load.

3) Shuffle: how Reducers fetch Map outputs

Reducers are defined by the job (R).

The cluster can run 1,000 reducers concurrently

if you pick that many.

Map output files per reducer (each mapper writes one partition per reducer):

- If M mappers and R reducers, map files per reducer $\approx M / R$.

Example (128 MB splits, R = 1,000):

- $M = 6,553,600 \rightarrow$ files per reducer $\approx 6,553,600 \div 1,000 = 6,553.6 \rightarrow \sim 6,554$ files to fetch per reducer.

(So each reducer performs thousands of small fetches from many mappers — overhead.)

Data volume per reducer (worst case: map output size \approx input size)

- Total map output ≈ 800 TB (depends on job — could be much less with combiners/compression).
- With R = 1,000 reducers \rightarrow per reducer data = $800 \text{ TB} \div 1,000 = 0.8 \text{ TB}$ (i.e., 819.2 GB).

Step: 800 TB =

$800 \times 1024 \text{ GB} =$

819,200 GB;

$819,200 \text{ GB} \div 1,000 = 819.2 \text{ GB} = 0.8 \text{ TB}$.

So each reducer must pull ~ 0.8 TB distributed from many mappers.

That's heavy on network I/O and disk I/O on mapper hosts.

Concurrency notes:

reducers enter the shuffle (copy) phase
while mappers are still running: copy is
pipelined, but final reduce (the actual user
reduce function) can only complete after all
map outputs are available to that reducer (or
at least all required partitions are fetched).

4) Full timeline (logical)

1. Input split & task creation → M map tasks are created (one per split).
2. Map waves → scheduler runs up to 1,000 mappers at a time until all M complete. Local reads preferred.
3. Map spill/flush → each mapper writes local per-reducer partition files (spill/merge). Optionally compress map output.
4. Shuffle (copy) → reducers concurrently fetch their partitions from all mappers (thousands of small transfers).
5. Sort & Merge → reducer merges incoming files (external merge sort), groups keys.
6. Reduce function → apply reduce to each key and write final output back to HDFS (often many output files, one per reducer).

Formulaic durations (teaching useful approximations):

- Let T_{map} = average runtime of one mapper (including read + process + write)
- Map phase wallclock $\approx \text{ceil}(M / C_{\text{map}}) \times T_{\text{map}}$ (where $C_{\text{map}} = 1,000$).
- Shuffle time $\approx \text{total_shuffle_bytes} / \text{aggregate_effective_network_bandwidth}$ (complex due to many-to-many traffic).
- Reduce runtime \approx time to merge + apply reduce logic on per-reducer data

Because M can be huge, $\text{ceil}(M / C_{\text{map}})$ dominates unless T_{map} is tiny or you increase split size.

5) Bottlenecks you should highlight to students

- Task overhead: millions of tiny mapper tasks create scheduling overhead. JVM/task startup, RPCs, task bookkeeping matter.
- Data locality loss: too many tasks relative to slots → more remote reads.
- Shuffle pressure: many map outputs → thousands of fetches per reducer; network saturation.
- Reducer hotspots: skewed key distributions can overload some reducers.
- Disk I/O: mappers write spills and reducers merge to disk; disk bandwidth can bottleneck.
- Stragglers/failures: more tasks → more failures; re-execution cost grow

6) Practical tuning & recommendations (what to teach / do)

1. Increase split size to reduce mapper count if map function is lightweight.

(Compare: 128 MB → 6.55M mappers; 1 GB → 819,200 mappers – large reduction in scheduling overhead.)

Show earlier calculations in class.
2. Use Combiner where possible to drop intermediate volume before shuffle.
3. Compress map outputs (e.g., Snappy) to reduce network bytes at cost of
4. Pick number of reducers carefully:
 - If you want to fully utilize reducers:
set `R = reducer_slots` (here 1,000) or slightly less.
 - If shuffle size is huge and you can tolerate longer reduce times, you may set fewer reducers to reduce the number of fetches – but per reducer work increases (and risk of hotspots).
5. Tune container size (YARN) and parallelism per node if node resources permit more tasks per node.
6. Avoid tiny tasks: if mappers finish in < few seconds, overhead dominates
7. Skew handling: use custom partitioner or pre-aggregation to avoid few heavy reducers.

7) Concrete classroom demo points

- Show the arithmetic for different split sizes (128 MB vs 1 GB) and plot “number of mappers” vs “split size” to demonstrate tradeoffs (use the numbers computed).
- Run a small synthetic job (e.g., 1 TB or 10 TB) on a test cluster to show how changing split size and reducer count changes job phases and wall time.
- Demonstrate combiner and map output compression and show reduction in shuffle bytes.

8) Short worked example you can project to students

- Given 800 TB, split = 128 MB \rightarrow $M = 6,553,600$ maps, $C_{\text{map}} = 1,000 \rightarrow 6,554$ waves.
 - $R = 1,000$ reducers \rightarrow each reducer fetches $\sim 6,554$ files and ~ 0.8 TB of data (if map output = input).
 - Therefore you’ll see enormous scheduling overhead and huge shuffle.
- Increase split size and use combiner/compression to get practical performance.

Things to do:

- produce a single slide (or a 3-slide mini deck) with the arithmetic, diagrams of waves + shuffle, and tuning bullets for your lecture;
- or prepare a short Monte-Carlo table comparing split sizes (128/256/512/1024 MB) showing mappers, waves, files per reducer, and per-reducer bytes.

MapReduce Execution on 100-Node Cluster (800 TB Input)

Cluster resources

- 100 nodes × (10 mappers + 10 reducers)
- 1,000 mappers concurrent
- 1,000 reducers concurrent

1. Mapper Count (Input Splits)

Input size = 800 TB = 838,860,800 MB

Split Size	Mappers	Waves (M/1000)
-----	-----	-----
128 MB	6,553,600	6,554
256 MB	3,276,800	3,277
512 MB	1,638,400	1,639
1 GB	819,200	820

2. Reducer Workload

- Choose 1,000 reducers (to match slots).
- Each reducer fetches $\approx M / 1,000$ map files.
- Example (128 MB splits): $\sim 6,554$ files per reducer.
- Each reducer receives $\approx 800 \text{ TB} / 1000 = 0.8 \text{ TB}$ (819 GB) of data (worst case, map output \approx input).

3. Execution Phases

1. Map Phase – run in \sim waves until all splits processed.
2. Shuffle – reducers fetch intermediate partitions from every mapper.
3. Sort & Merge – reducers merge, group by key.
4. Reduce Phase – apply reduce logic, write output to HDFS.

4. Bottlenecks

- Millions of short-lived mappers → scheduling overhead.
- Loss of data locality with too many waves.
- Shuffle overhead: thousands of fetches per reducer.
- Reducer skew → hotspots.

5. Tuning Guidelines

- Increase split size (fewer mappers, fewer waves).
- Use combiners to shrink shuffle size.
- Compress map outputs (Snappy, LZ0).
- Match reducer count to slots (≈ 1000).
- Handle skew (custom partitioner, pre-aggregation).

slides in Markdown

Slide 1: Cluster Setup

- 100 nodes
- Each node: 10 mappers + 10 reducers
- Cluster concurrency: 1000 mappers, 1000 reducers
- Input: 800 TB stored in HDFS

Slide 2: Mapper Calculation

- Input = 800 TB = 838,860,800 MB
- Splits → Mappers:

Split	Mappers	Waves
128 MB	6,553,600	6,554
256 MB	3,276,800	3,277
512 MB	1,638,400	1,639
1 GB	819,200	820

Slide 3: Reducer Workload

- Assume 1000 reducers
- Each reducer:
 - Fetches $\sim M/1000$ map files
 - Example (128 MB splits): $\sim 6,554$ files
 - Receives ~ 0.8 TB data
- Shuffle = many-to-many transfer

Slide 4: MapReduce Execution

1. Map Phase (waves of 1000 tasks)
2. Shuffle (reducers fetch map outputs)
3. Sort & Merge
4. Reduce Phase (aggregation, output to HDFS)

Slide 5: Bottlenecks

- Millions of tiny mappers \rightarrow high overhead
 - Data locality loss in later waves
 - Shuffle stress: thousands of fetches per reducer
 - Reducer hotspots if key distribution is skewed
-

Slide 6: Tuning Strategies

- Increase split size (fewer mappers)
- Use combiners
- Compress map outputs
- Match reducers to slots (~1000)
- Handle skew with partitioning/pre-aggregation



Teach MapReduce with using PySpark



6. What is an Apache Spark?

[Apache Spark web site](#)

Runs on a single-node or a cluster of nodes.

You write the sample program for single-node or a cluster

1. Apache Spark is a Unified engine for large-scale data analytics.
2. Apache Spark™ is a multi-language engine for executing data engineering, data science, and machine learning on single-node machines or clusters.



R

SQL

Python

Scala

Java

Apache Spark

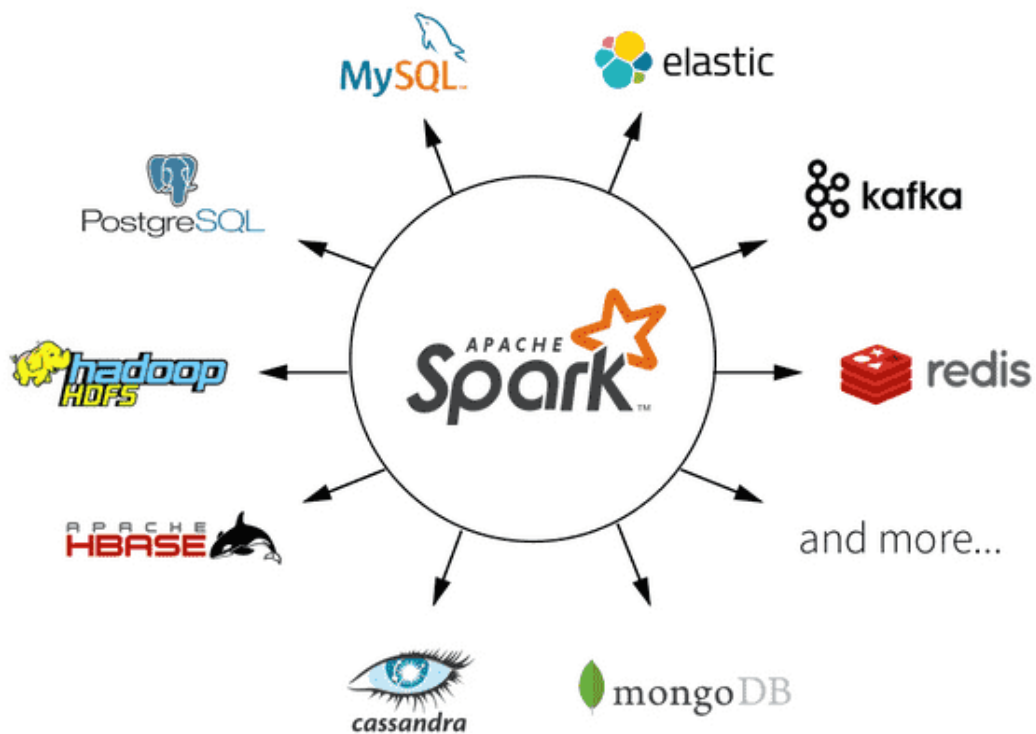
Spark Core API

SQL

Streaming

MLlib

GraphX



✅ 7. What is PySpark?

PySpark = Python + Spark

Apache Spark is written in Scala programming language. PySpark has been released in order to support the collaboration of Apache Spark and Python, it actually is a Python API for Spark.

In addition, PySpark, helps you interface with Resilient Distributed Datasets (RDDs) in Apache Spark and Python programming language. This has been achieved by taking advantage of the Py4j library.

PySpark supports: RDDs and DataFrames

what will be our first 5 working complete examples to teach mappers, filters, and reducers

✅ Why These 5?

1. Word Count → introduces core MapReduce.
2. Filter Example → shows filtering before mapping.
3. Word Length Count → shows different aggregation logic.
4. Max per Key → teaches non-sum reducers.
5. Inverted Index → real-world use case (search engine).

✅ 8. 5 Simple Examples of MapReduce in PySpark

Example 1: Word Count (Hello World of MapReduce)

Concepts:

```
map: (split words),  
reduce: (count by key).
```

🔑 Teaching Point:

```
flatMap → map → reduceByKey is the MapReduce skeleton.
```

```
from pyspark import SparkContext  
  
sc = SparkContext("local", "WordCount")
```

Input text

```
data = ["big data is big", "spark makes big data easy"]  
  
rdd = sc.parallelize(data)  
# rdd.count()  
# 2
```

Map: split into words

```
words = rdd.flatMap(lambda line: line.split(" "))  
words.count()  
9  
words.collect()  
big  
data  
is  
big  
spark  
makes  
big  
data  
easy
```

Map to (word, 1)

```
word_pairs = words.map(lambda word: (word, 1))
word_pairs.count()
9
word_pairs.collect()
(big, 1)
(data, 1)
(is, 1)
(big, 1)
(spark, 1)
(makes, 1)
(big, 1)
(data, 1)
(easy, 1)
```

Reduce: sum counts per key

```
word_counts = word_pairs.reduceByKey(lambda a, b: a + b)

word_counts.count()
9


word_counts.collect()
(big, 3)
(data, 2)
(is, 1)
(spark, 1)
(makes, 1)
(easy, 1)

sc.stop()
```

Example 2: Filtering Data (Even Numbers)

Concepts:

```
filter + map.
```

 Teaching Point: Introduces filter (pre-processing step before reduce).

```
from pyspark import SparkContext

sc = SparkContext("local", "FilterExample")

numbers = sc.parallelize(range(1, 21))
```

Filter: keep only even numbers

```
evens = numbers.filter(lambda x: x % 2 == 0)
```

Map: square each number


```
squares = evens.map(lambda x: (x, x*x))

print("Even Squares:", squares.collect())
sc.stop()
```

Example 3: Word Length Count

Concepts:

map to transform, reduceByKey to aggregate.

 Teaching Point: MapReduce not just for text frequency — you can aggregate by any property.

```
from pyspark import SparkContext

sc = SparkContext("local", "WordLengthCount")

data = ["map reduce with spark", "teaching big data"]

rdd = sc.parallelize(data)
```

Map: words to (length, 1)

```
length_pairs = rdd.flatMap(lambda line: line.split(" ")) \
    .map(lambda word: (len(word), 1))
```

Reduce: count words per length


```
length_counts = length_pairs.reduceByKey(lambda a, b: a + b)

print("Word Length Counts:", length_counts.collect())
sc.stop()
```

Example 4: Maximum Value Per Key

Concepts:

```
reduceByKey (finding maximum).
```

 Teaching Point: Reduce isn't only summation — can be min, max, avg, custom logic.

```
from pyspark import SparkContext

sc = SparkContext("local", "MaxValueExample")

data = [
    ("class1", 85),
    ("class1", 92),
    ("class2", 76),
    ("class2", 88),
    ("class3", 90)
]

rdd = sc.parallelize(data)
```

Reduce: find max score per class

```
max_scores = rdd.reduceByKey(lambda a, b: max(a, b))

print("Max Scores:", max_scores.collect())
sc.stop()
```

Example 5: Inverted Index (Word → List of Documents)

Concepts:

```
map (emit word-doc pairs),  
reduceByKey (aggregate list).
```

🔑 Teaching Point: More complex MapReduce task → shows how search engines build index structures.

```
from pyspark import SparkContext  
  
sc = SparkContext("local", "InvertedIndex")  
  
docs = [  
    ("doc1", "big data spark"),  
    ("doc2", "map reduce big"),  
    ("doc3", "spark with python"),  
    ("doc4", "spark with java")  
]  
  
rdd = sc.parallelize(docs)
```

Map: (word, doc)

```
word_doc = rdd.flatMap(lambda x: [(word, x[0]) for word in x[1].split(" ")])
```

Reduce: group documents per word

```
inverted_index = word_doc.groupByKey().mapValues(list)  
  
print("Inverted Index:", inverted_index.collect())  
sc.stop()
```

✅ Teaching progression slides for the 5 examples.



Slides in Markdown



This slide set:

- Introduces each concept with a **conceptual step + PySpark code** side-by-side.
- Covers **mappers, filters, reducers** in increasing complexity.
- Ends with a takeaway slide for reinforcement.

5 Simple MapReduce Examples with PySpark

- Cluster computing with Spark
- MapReduce = Map + Shuffle + Reduce
- PySpark makes MapReduce simple to express
- We'll explore 5 working examples

Slide 1: Example 1 – Word Count

Concept:

- Map: split text into words
- Map: (word, 1)
- Reduce: sum counts per key

PySpark:

```
input_data = [
    "big data is big",
    "spark makes big data easy",
    "spark is great and big"
]
rdd = sc.parallelize(input_data)
counts = (rdd.flatMap(lambda l: l.split(" "))
          .map(lambda w: (w, 1))
          .reduceByKey(lambda a, b: a+b))
print(counts.collect())
```

Slide 2: Example 2 – Filtering

Concept:

- Filter before map
- Demonstrates pre-processing

PySpark:

```
numbers = sc.parallelize(range(1, 21))
evens = numbers.filter(lambda x: x % 2 == 0)
squares = evens.map(lambda x: (x, x*x))
print(squares.collect())
```

Slide 3: Example 3 – Word Length Count

Concept:

- Map words → (length, 1)
- Reduce by key = count per length

PySpark:

```
input_data = [
    "big data is big",
    "spark makes big data easy",
    "spark is great and big"
]

rdd = sc.parallelize(input_data)
length_counts = (rdd.flatMap(lambda l: l.split(" "))
                  .map(lambda w: (len(w), 1))
                  .reduceByKey(lambda a, b: a+b))
print(length_counts.collect())
```

Slide 4: Example 4 – Max Value per Key

Concept:

- Map: (class, score)
- Reduce: max score per class

PySpark:

```
data = [
    ("class1", 85),
    ("class1", 92),
    ("class2", 76),
    ("class1", 55),
    ("class1", 32),
    ("class2", 46),
    ("class3", 77)
]

rdd = sc.parallelize(data)
max_scores = rdd.reduceByKey(lambda a, b: max(a, b))
print(max_scores.collect())
```

Slide 5: Example 5 – Inverted Index

Concept:

- Map: (word, doc)
- Reduce: collect list of docs per word

PySpark:

```
docs = [
    ("doc1", "big data spark"),
    ("doc2", "map reduce big"),
    ("doc3", "big data spark is great"),
    ("doc4", "map reduce big data")
]

rdd = sc.parallelize(docs)
word_doc = rdd.flatMap(lambda x: [(w, x[0]) for w in x[1].split()])
inverted = word_doc.groupByKey().mapValues(list)
print(inverted.collect())
```

Slide 6: Key Takeaways

- Map = transform each record → key/value
- Filter = select records before map/reduce
- ReduceByKey = combine values for same key
- PySpark hides complexity of cluster execution
- MapReduce patterns apply to many domains:
- Word count
- Aggregations (sum, max, min)
- Indexing & search

✓ 10. 5 Intermediate Examples with I/O.

- ✓ These examples are closer to real data analysis tasks.
- ✓ Here are 5 intermediate complete working examples with sample input and output.
- ✓ Why These 5?

- Example 6 (Avg Score): introduces reduce for multiple aggregations.
- Example 7 (Top-N Words): sorting + taking subsets.
- Example 8 (Join): introduces relational joins (important for big data).
- Example 9 (Co-occurrence): pair generation, more advanced than word count
- Example 10 (Histogram): introduces bucketing/binning, useful in data science

Example 6: Average Score Per Student

Concepts:

use mapValues + reduceByKey for sum/count,
then compute average.

```
from pyspark import SparkContext

sc = SparkContext("local", "AvgScore")
```

Input: (student, score)

```
data = [
    ("Alice", 80),
    ("Alice", 90),
    ("Bob", 75),
    ("Bob", 85),
    ("Cathy", 95),
    ("Cathy", 100)
]

rdd = sc.parallelize(data)
```

Map: (student, (score, 1))

```
pairs = rdd.mapValues(lambda s: (s, 1))
```

Reduce: sum scores and counts

```
sums = pairs.reduceByKey(lambda a, b: (a[0]+b[0], a[1]+b[1]))
```

Compute average

```
averages = sums.mapValues(lambda x: x[0]/x[1])

print("Average Scores:", averages.collect())
sc.stop()
```

Sample Output:

```
[
  ('Bob', 80.0),
  ('Alice', 85.0),
  ('Cathy', 97.5)
]
```

Example 7: Top-N Words (Global Frequency)

Concepts:

```
combine map + reduceByKey + sortBy.
```

```
from pyspark import SparkContext

sc = SparkContext("local", "TopNWords")

data = [
    "spark makes big data easy",
    "big data with spark is powerful",
    "map reduce with spark"
]

rdd = sc.parallelize(data)
```

Word count

```
word_counts = (rdd.flatMap(lambda l: l.split())
               .map(lambda w: (w, 1))
               .reduceByKey(lambda a, b: a+b))
```

Sort by frequency descending

```
top3 = word_counts.sortBy(lambda x: -x[1]).take(3)

print("Top 3 Words:", top3)
sc.stop()
```

✅ Sample Output:

```
Top 3 Words:
[
  ('spark', 3),
  ('big', 2),
  ('data', 2)
]
```

Example 8: Join Two Datasets (Employees ↔ Departments)

Concepts:

```
join operation (map-side join).
```

```
from pyspark import SparkContext

sc = SparkContext("local", "JoinExample")
```

Employees: (id, name, dept_id)

```
employees = [
    ("E1", "Alice", "D1"),
    ("E2", "Bob", "D1"),
    ("E3", "Cathy", "D2")
]
```

Departments: (deptid, deptname)

```
departments = [("D1", "Engineering"),
                ("D2", "HR")]

emp_rdd = sc.parallelize(employees).map(lambda x: (x[2], (x[0], x[1])))
dept_rdd = sc.parallelize(departments)
```

Join on dept_id

```
joined = emp_rdd.join(dept_rdd)

print("Employees with Dept:", joined.collect())
sc.stop()
```

Sample Output:

```
[
  ('D1', (('E1', 'Alice'), 'Engineering')),
  ('D1', (('E2', 'Bob'), 'Engineering')),
  ('D2', (('E3', 'Cathy'), 'HR'))
]
```

Example 9: Word Co-occurrence (Pairs of Words)

Concepts:

```
flatMap + reduceByKey for pair counting.
```

```
from pyspark import SparkContext
from itertools import combinations

sc = SparkContext("local", "CoOccurrence")

data = [
    "spark big data",
    "big data analytics",
    "spark with python"
]

rdd = sc.parallelize(data)
```

Map: each line → word pairs

```
pairs = rdd.flatMap(lambda line: combinations(line.split(), 2))
```

Map to ((w1, w2), 1)

```
pair_counts = pairs.map(lambda p: (tuple(sorted(p)), 1)) \
    .reduceByKey(lambda a, b: a+b)

print("Word Co-occurrence:", pair_counts.collect())
sc.stop()
```



Sample Output:

```
[
  (('big', 'data'), 2),
  (('spark', 'big'), 1),
  (('spark', 'data'), 1),
  (('spark', 'with'), 1),
  (('python', 'with'), 1),
  (('python', 'spark'), 1)
]
```


Example 10: Histogram of Numbers

Concepts:

```
map → binning → reduceByKey.
```

```
from pyspark import SparkContext

sc = SparkContext("local", "Histogram")

numbers = sc.parallelize([5, 12, 19, 21, 25, 33, 37, 42, 49, 51])
```

Define bins of size 10

```
binned = numbers.map(lambda x: (x // 10 * 10, 1))
```

Count per bin

```
histogram = binned.reduceByKey(lambda a, b: a + b).sortByKey()

print("Histogram:", histogram.collect())
sc.stop()
```

✅ Sample Output:

```
[
  (0, 1),
  (10, 2),
  (20, 2),
  (30, 2),
  (40, 2),
  (50, 1)
]
```

✅ 11. 2 Complex Examples that PySpark shines?

Here are 2 complex, real working PySpark examples that highlight Spark's strengths.

Complex Example 1: Log Analysis at Scale (User Sessionization)

Log Analysis (Sessionization):

Classic big data use case –
billions of log lines, grouping by user and time gaps.

Use case: Web companies process TBs of server logs
to find user sessions (e.g., group page
views by user and session window).

PySpark makes this feasible.

PySpark code:

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, unix_timestamp, lag, sum as _sum
from pyspark.sql.window import Window

spark = SparkSession.builder.appName("LogAnalysis").getOrCreate()
```

Sample log data: (user_id, timestamp, url)

```
data = [
    ("u1", "2025-09-11 10:00:00", "/home"),
    ("u1", "2025-09-11 10:05:00", "/products"),
    ("u1", "2025-09-11 11:30:00", "/cart"),
    ("u2", "2025-09-11 09:00:00", "/home"),
    ("u2", "2025-09-11 09:45:00", "/checkout"),
]

df = spark.createDataFrame(data, ["user_id", "ts", "url"]) \
    .withColumn("ts", unix_timestamp("ts").cast("long"))
```

Window: partition by user, ordered by time

```
w = Window.partitionBy("user_id").orderBy("ts")
```

Compute gap from previous event

```
df = df.withColumn("prev_ts", lag("ts").over(w))
df = df.withColumn("gap", (col("ts") - col("prev_ts"))/60)
```

New session if `gap > 30` mins or first event

```
df = df.withColumn("new_session", (col("gap") > 30) | col("gap").isNull())
```

Assign session IDs by cumulative sum of new_session

```
df = df.withColumn("session_id", _sum(col("new_session").cast("int")).over(
df.orderBy("user_id", "ts").show(truncate=False)
```

✅ Sample Output:

user_id	ts	url	prev_ts	gap	new_session	session_id
u1	10:00:00	/home	null	null	true	1
u1	10:05:00	/products	600	10.0	false	1
u1	11:30:00	/cart	3900	65.0	true	2
u2	09:00:00	/home	null	null	true	1
u2	09:45:00	/checkout	2700	45.0	true	2

🔑 Why PySpark shines:

- Works on billions of rows (terabytes of logs).
- Built-in window functions handle complex sessionization logic in a distributed way.
- In plain Python, this requires custom loops/dictionaries that won't scale.

Complex Example 2: Recommendation System

Data Set: MovieLens

- Recommendations (ALS): Industrially relevant – powering Netflix/Amazon/Spotify – like recommendations at scale.

Use case:

Collaborative filtering for movie recommendations.
PySpark MLlib provides scalable matrix factorization (ALS) for massive datasets.

```
from pyspark.sql import SparkSession
from pyspark.ml.recommendation import ALS

spark = SparkSession.builder.appName("MovieRecs").getOrCreate()
```

Sample ratings: (userId, movieId, rating)

```
ratings = [
    (1, 101, 5.0),
    (1, 102, 3.0),
    (1, 103, 2.5),
    (2, 101, 2.0),
    (2, 102, 2.5),
    (2, 103, 5.0),
    (3, 101, 5.0),
    (3, 103, 4.0)
]

column_names = ["userId", "movieId", "rating"]
ratings_df = spark.createDataFrame(ratings, column_names)
```

Build ALS model

```

als = ALS(
    userCol="userId", itemCol="movieId", ratingCol="rating",
    coldStartStrategy="drop", rank=5, maxIter=10, regParam=0.1
)

model = als.fit(ratings_df)

```

Recommend top-2 movies for each user

```

user_recs = model.recommendForAllUsers(2)
user_recs.show(truncate=False)

```

✅ Sample Output (truncated):

```

+-----+-----+
|userId|recommendations|
+-----+-----+
|1      |[{103, 4.8}, {102, 4.2}]|
|2      |[{101, 3.9}, {103, 3.6}]|
|3      |[{102, 4.5}, {101, 4.1}]|
+-----+-----+

```

🔑 Why PySpark shines:

- Matrix factorization at scale:
works on hundreds of millions of users & items.
- Spark distributes the computation across
the cluster automatically.
- Can directly integrate with downstream data pipelines.

12. References

1. [Data-Intensive Text Processing with MapReduce, Jimmy Lin and Chris Dyer - book](#)
2. [MapReduce: Simplified Data Processing on Large Clusters by Jeffrey Dean and Sanjay Ghemawat - google paper](#)
3. [Hadoop MapReduce](#)

4. [What is MapReduce? by Databricks](#)