SPARK

- most important topic
  - Interview
  - Work / Actual Job
    - → Theory
    - → Practicals

→ foundamentals of
1. distributed processing
2. distributed storage

HDFS – majorly getting replaced by cloud based platform

YARN – still preferred as a resource manager (mesos, docker, k8s) but in DE system.

MR – writing & performance both issues are here

In hadoop ecosystem, spark is replacing MR

plug in → spark

Pre-requisite.
1. Python
2. SQL
3. Basics of big data & hadoop

we can seperately learn spark but its better to learn with hadoop.

Some common Questions about Spark

Q: Do I need to Know Hadoop to learn SparK?

No, we can learn spark independently.

Having idea about distributed storage & distributed processing that is very helpful. YARN (Resource manager) is helpful.

Q. Do I need to install Hadoop to run SparK?

No. Spark can run in standalone mode without say YARN.

If we want HDFS then hadoop is required.

Q. What are the prerequisite to learn SparK?

Python, SQL + Knowledge of distributed system is a plus.
or (Java, Scala)

Q. Can I use Spark on my laptop? Where all can we run it?

Yes, it is known as a local mode & good for learning or small testing

1. Google dataproc cluster
2. Google colab
3. Local machine
4. Databricks

+ ....

Q. What programming Language can I use with SparK?

Python Java Scala R

Q. How is spark different from tradional databases?

wrong Question

Databases are for storing & Querying data.

Spark is a data processing language.

Q. What storage system can Spark with?

HDFS, S3, ADLS Gen2, Cassandra, Kafka etc.

Q. Is spark free to use?

Yes, free to use. Apache Foundation

Android
↓
Samsung, Motorola

Q. How is spark different from data warehousing tools like Hive?

Hive is a data warehousing tool

Spark is a general purpose processing engine.

Q. Can I use spark on the cloud?

Yes, GCP, AWS, Azure, Databricks

Limitations of Map Reduce

1. Performance : heavily dependent on disk ⟶ high latency
   many I/O dependency/ operation on disk
2. Complexity & hard to write code: boilerplate code
   verbose
3. Only batch processing supported: Real time processing not supported
   so we cannot work on streaming data
4. Rigid Structure : filter
   ↳ map → shuffle → reduce
5. No interactive mode/way to monitor : JAR packaged & sent

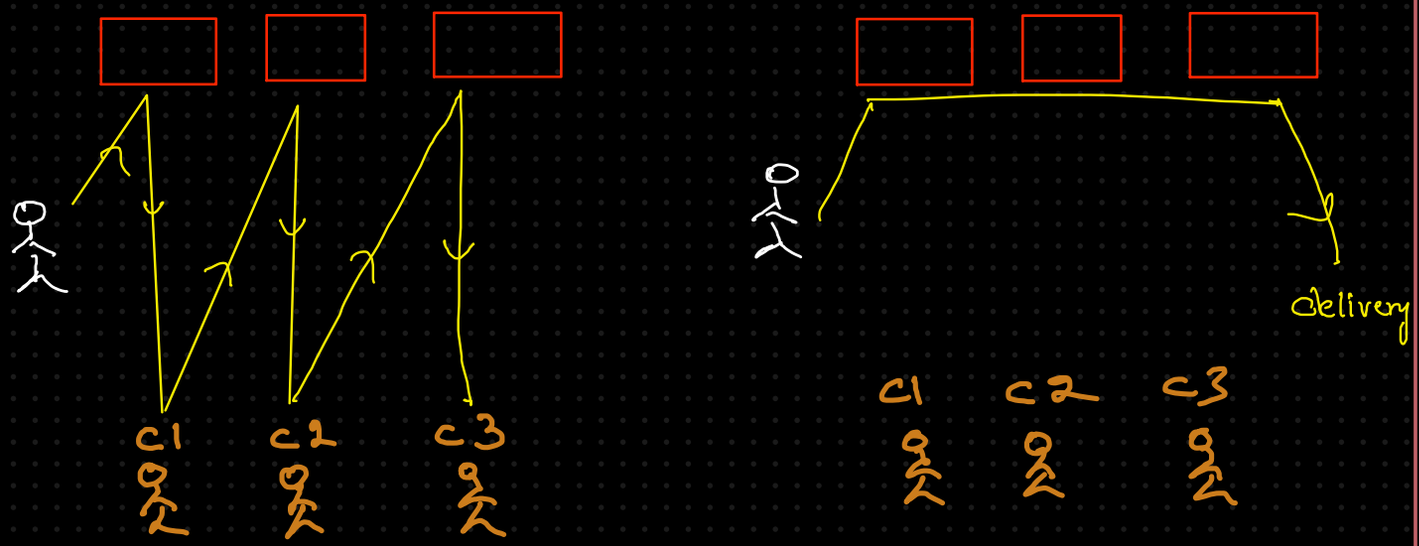
Apache Spark

Apache spark is an open source, distributed computing system designed to process large scale datasets quickly.

→ fast
→ general purpose → any storage and say resource
→ batch & real time processing

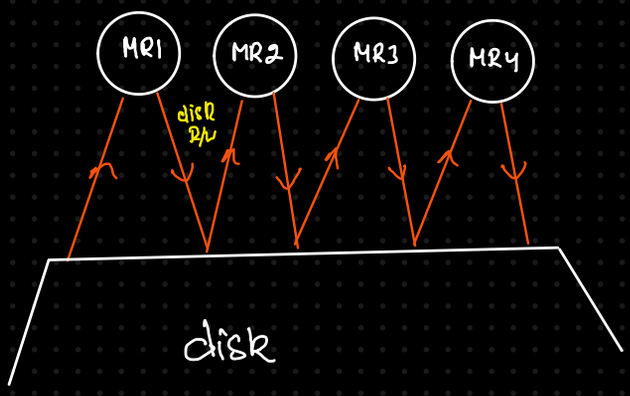
Characteristic

1. In-memory processing : Spark processes data in memory as compared to traditional disk-based systems like MR.



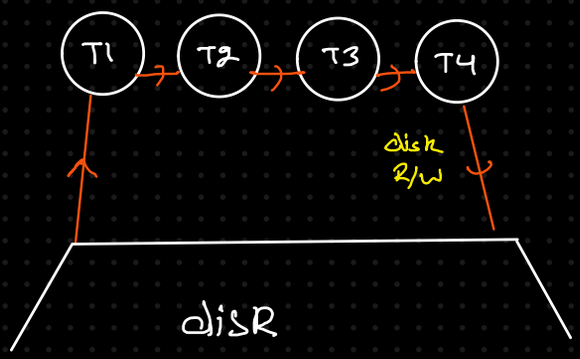
2. Ease of use : Spark provides API for major lang. & easy to write code.

3. Unified Framework: Batch processing
Realtime processing
ML
Graph Processing



Map Reduce

5 MR → 10 Read & write



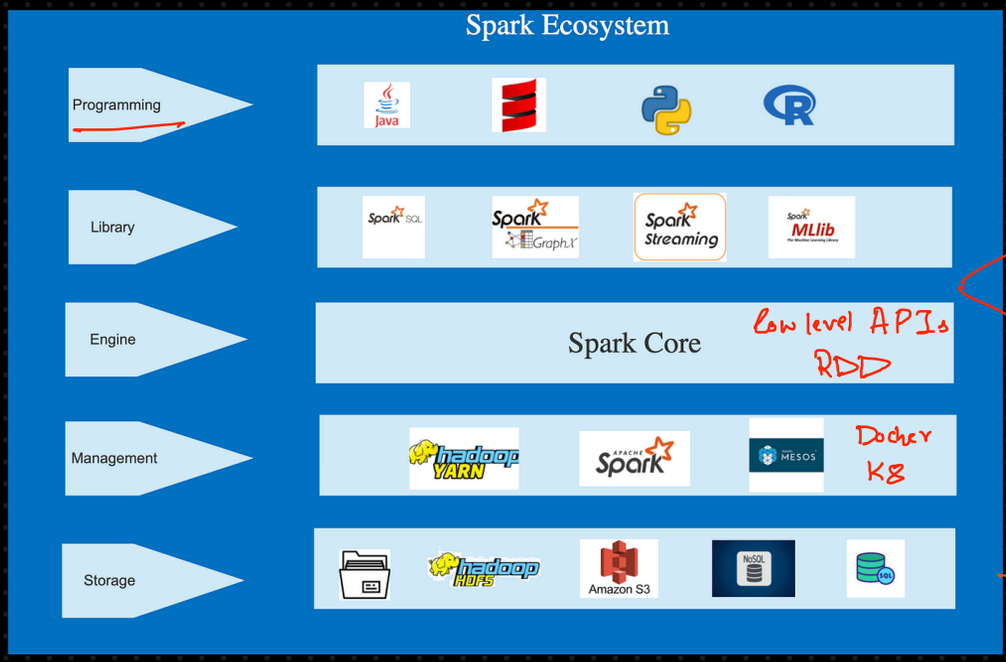
Spark

5 Spark → 2 read/write

Features:

1. Speed
2. Scalability ⇒ 1 m/c to 100 of m/c
3. Fault tolerance: DAG (direct acyclic Graph)
4. Polyglot: Multi lang. support
5. Unified Performance.
6. Integration with Existing System

Spark Ecosystem



to make life easier

easier but less flexibility

Layer 2 (High level API)

Layer 1 (Low level API)

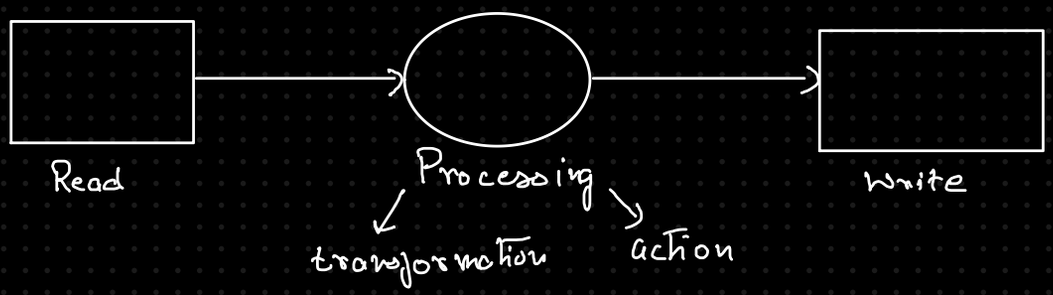
Low level APIs RDD

Core level

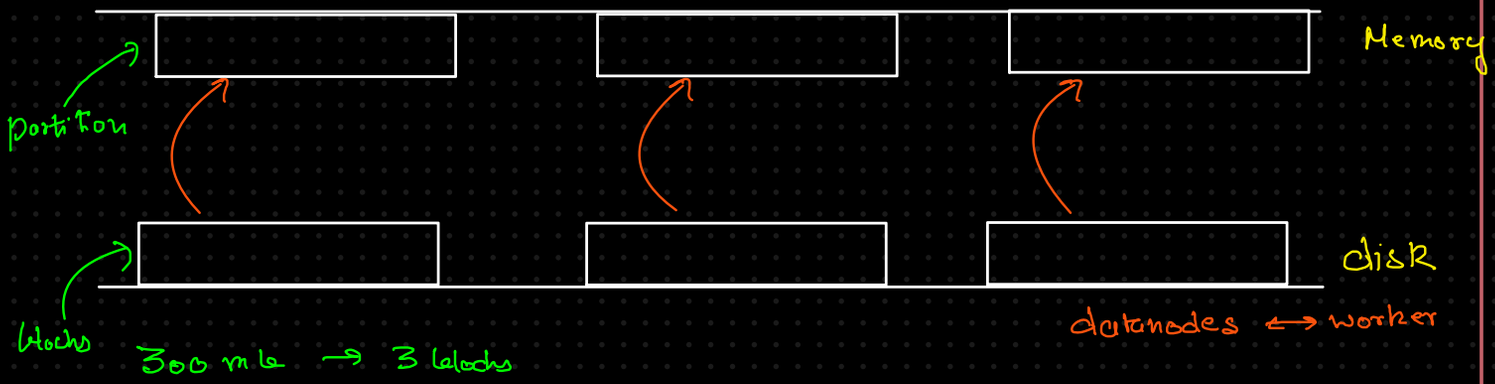
Docker K8

difficult but flexible

→ data is stored



Execution Code in Spark



Blocks on disk ⟶ partitions in memory

In partitions, the data is still distributed. & RDD is most basic unit

hdfs file ⟶ read in spark as a rdd ⟶ do some processing

↓

Save each to some place

Word Count Program in Spark

Spark Code, Spark UI and History Server

1. Google Data proc ⟶ Jupyter lab
   Multicluster setup
   Job Submit

   YARN

   History server
   Spark Live UI

   3 months + $300 credits

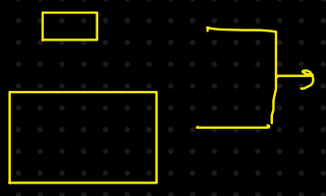
→ use a new Id where you have never created your gcloud Id
→ use a VISA/mastercard credit card

2. Databricks — easy to create free account
   one machine 15gb & 2cores
   We can create notebooks

3. Google Colab — We can access spark UI
   run it online
   free to use

Transformation vs Action

map filter flatmap

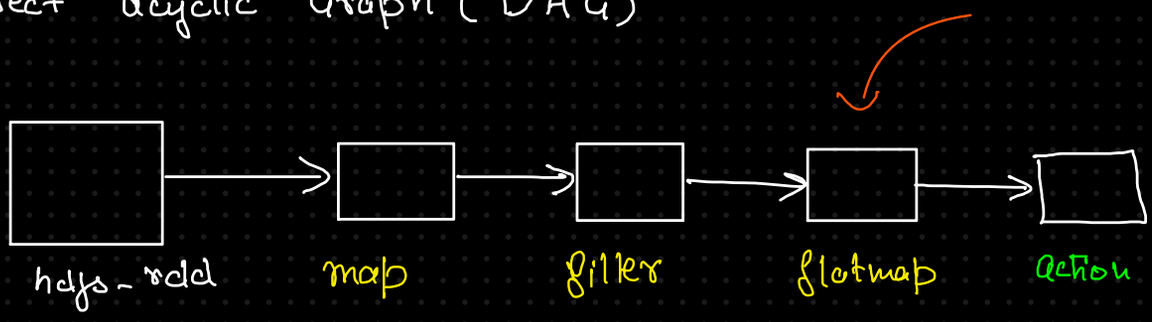


	Transformation	Action
Execution	lazily evaluated	eagerly evaluated
Definition	operation which creates a new RDD from existing one	an operation that triggers execution of transform. & return results.
Output	new RDD	final value / writes output to storage

Transformations				
map	join	union	distinct	repartition
mapPartitions	flatMap	intersection	pipe	coalesce
cartesian	cogroup	filter	sample	
sortByKey	groupByKey	reduceByKey	aggregateByKey	
mapPartitionswithIndex		repartitionAndSortWithinPartitions		
Actions				
reduce	take	collect	takeSample	count
takeOrdered	countByKey	first	foreach	saveAsTextFile
saveAsSequenceFile		saveAsObjectFile		

Occurs when an action is called	execution is immediate
Represented as an intermediate node in our DAG	Represented as a terminal node.

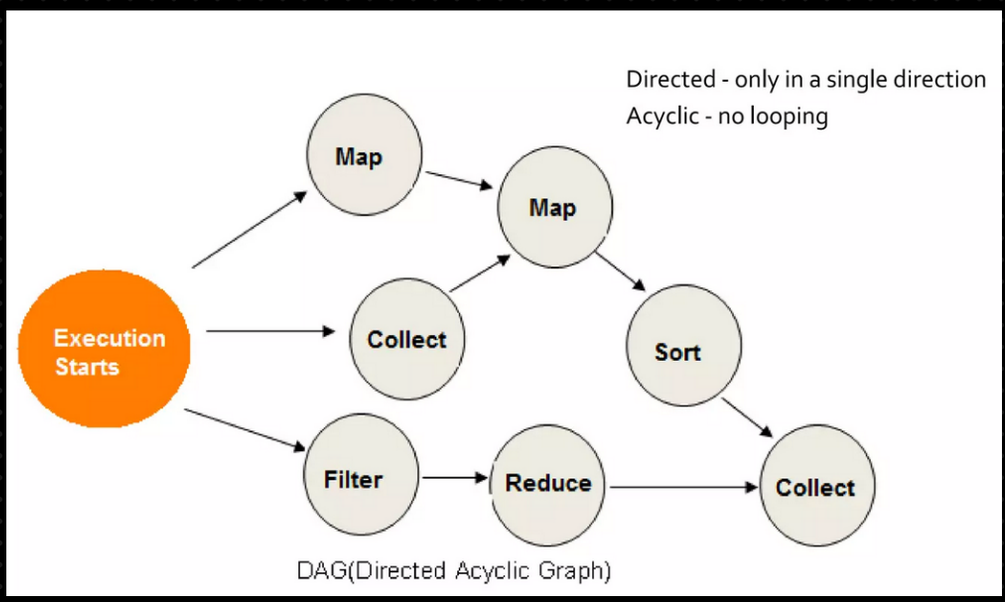
Direct acyclic Graph (DAG)



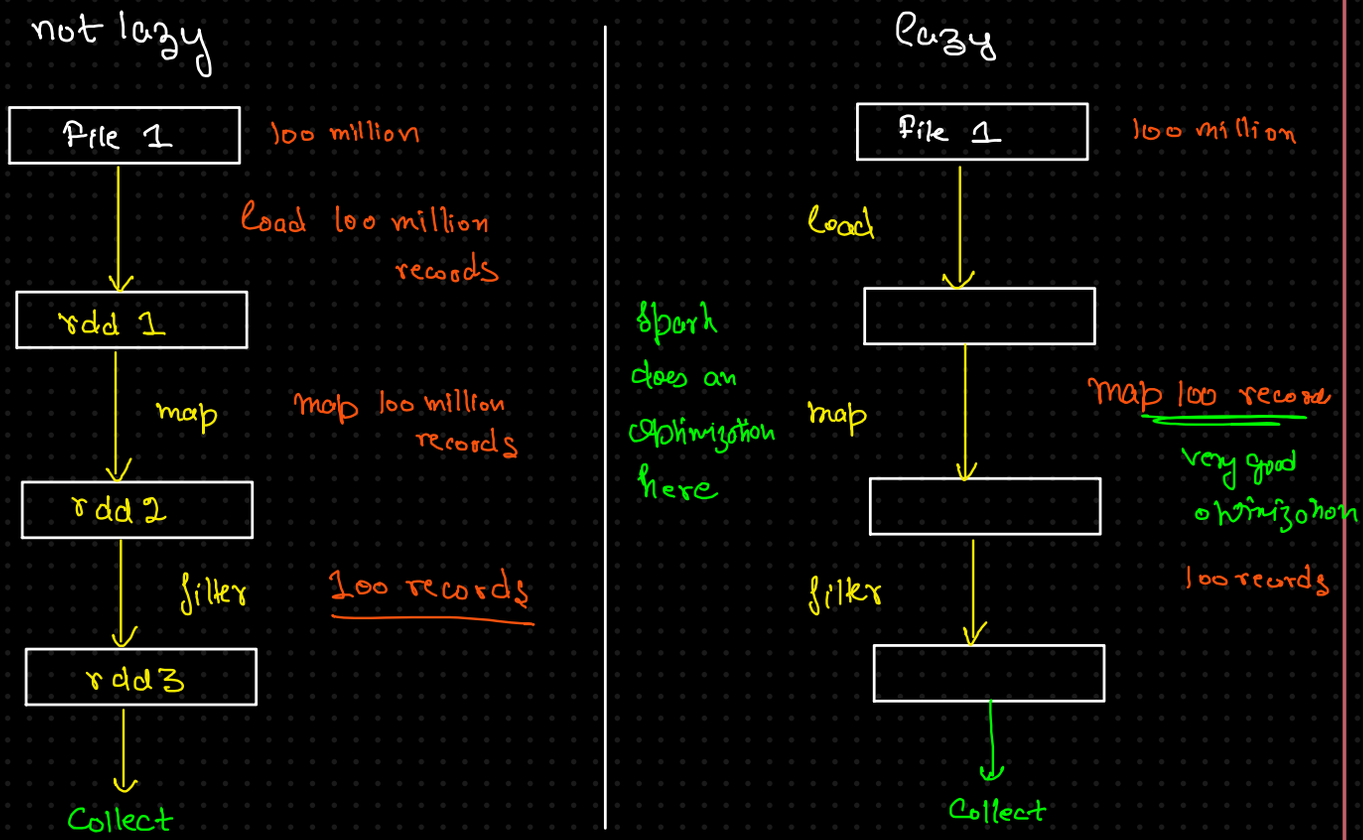
rdd rdd

fault Tolerant

At each step a new RDD is getting created



Why transformations are lazy ?

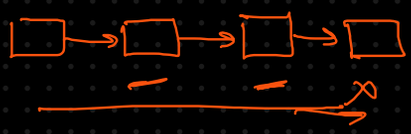


Shopping ⇒ discount

Coke ⇒ 10kg flour + 1kg chocolate + 1kg sugar ⟶ mix ↓ add some water ↓ bake ↓ 5 people

eat / serve

Spark



low level APIs

R - resilient → resistant to failure

D - distributed →

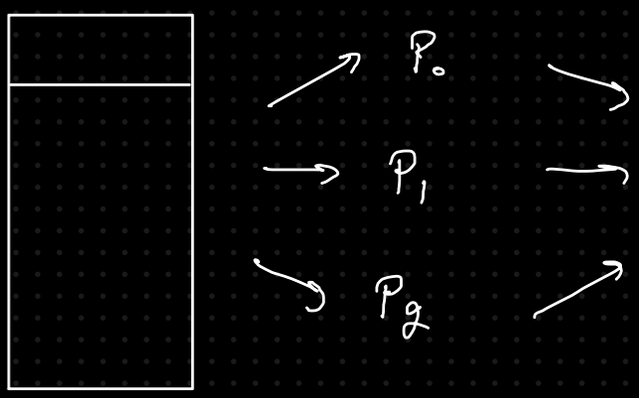
D - dataset

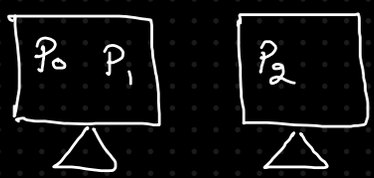
DAG

we need spark context to work with RDD

Properties

1. Immutability : Cannot be modified
2. Lazy evaluation : Computation not performed until an action is triggered
3. Partitioned :
4. Fundamental data structure.



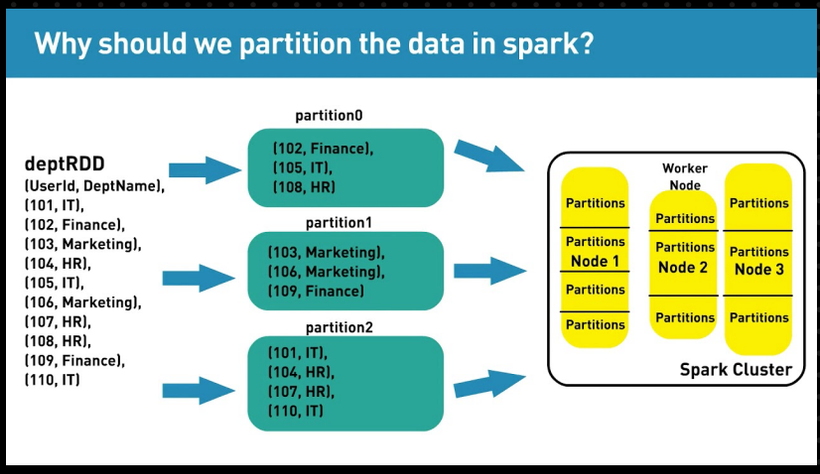


How Spark does data partitioning?

reading the data / defining data will of course be there

Data → rdd

Partitioning in spark refers to dividing large datasets into smaller, manageable chunks called partitions.



Aspect	Input Split (Hadoop)	Partition (Spark)
Nature	Logical division of input data for mappers	Logical division of RDDs/DataFrames for tasks
Creation	Defined by `InputFormat`, often based on HDFS blocks	Typically one per HDFS block but configurable
Control	Less dynamic control over split size	Dynamic control via transformations
Execution Model	Disk-based processing	In-memory processing
Performance Impact	Affects map task efficiency	Directly influences task scheduling and parallelism

How spark reads the data

1. Source of data → S3, hdfs, adls gen2, dB

Types of data sources

- already distributed like s3, hdfs
- read data & create partition based on configuration

2. Logical partitioning →

alec.txt (hdfs) → 1gb

⇓

8 blocks → 8 partitions

3. Distributed reading / processing

Each partition is processed independently by a executor in the cluster.

⇓

fetch data from storage → in-memory

⇓

transform & action

8 blocks → 8 partitions

\+ −

Spark parallelize and partitioning

sc.parallelize( )

a function to create an RDD from an in-memory location

sc.textfile( )

a function to read a hdfs file.

540 mb file ⇒ 5 blocks ⇒ 5 partition

1 mb file ⇒ 1 block ⇒ 2 partition

default minPartitions = 2

for local file ⇒ n cores ⇒ 4 partition

↳ default Parallelism = 4

Properties are dependent on platform.

↓

4 cores

Spark want to use all the cores in the machine for better parallelization.

```
# get num of Partition from rdd => getNumPartition()

# 544mb => 5 blocks => 5 partition
big_rdd_from_hdfs.getNumPartitions()
```

```
5
```

```
#1 mb => 1 block --> 1 partition ( This is wrong)
small_rdd_from_hdfs.getNumPartitions()
```

```
2
```

```
print(f"Max Partition Bytes: {spark.conf.get('spark.sql.files.maxPartitionBytes')}")
```

```
Max Partition Bytes: 134217728b
```

```
spark.sparkContext.defaultMinPartitions
```

```
2
```

```
local_rdd.getNumPartitions()
```

```
4
```

```
spark.sparkContext.defaultParallelism
```

```
4
```

Understanding the test data & its Generation

Table	Column Name	Description
Customers	`customer_id`	Unique identifier for each customer.
	`name`	Name of the customer.
	`city`	City where the customer resides.
	`state`	State of the customer.
	`country`	Country of the customer (set to India).
	`registration_date`	Date when the customer registered.
	`is_active`	Whether the customer is currently active or not.
Orders	`order_id`	Unique identifier for each order.
	`customer_id`	Foreign key linking to the `customers` table (which customer placed the order).
	`order_date`	Date when the order was placed.
	`total_amount`	Total amount of the order.
	`status`	Current status of the order (e.g., "Pending", "Shipped", "Delivered", "Cancelled").
Items	`item_id`	Unique identifier for each item.
	`order_id`	Foreign key linking to the `orders` table (which order the item belongs to).
	`item_name`	Name of the item.
	`category`	Category of the item (e.g., "Electronics", "Clothing").
	`price`	Price of the item.
Payments	`payment_id`	Unique identifier for each payment.
	`order_id`	Foreign key linking to the `orders` table (which order the payment is for).
	`payment_date`	Date when the payment was made.
	`amount`	Amount paid.
	`payment_method`	Method of payment (e.g., "Credit Card", "Debit Card", "UPI").
Shippings	`shipping_id`	Unique identifier for each shipping.
	`order_id`	Foreign key linking to the `orders` table (which order the shipping is for).
	`shipping_date`	Date when the shipping was made.
	`shipping_address`	Shipping address where the order was delivered.
	`shipping_method`	Method used for shipping (e.g., "Standard", "Express").

dbs ⇒ { ' ' , ' ' }

Customers - 1mb
- 500 mb

E commerce - related → 50+ tbs

1. We try to understand the data
2. We are going to understand problem statement.
3. We use/solve the problem on smaller data & then we scale it.

1 sec ↓ 0.5 sec

For projects, we will use this data and some real world so we get full confidence.

Spark RDD operations

Refer the notebook

```
# we can chain the functions together
customers_per_city = parsed_rdd.map(lambda row :(row[2],1)).reduceByKey(lambda x,y:x+y)
```

```
customers_per_city
```

```
PythonRDD[40] at RDD at PythonRDD.scala:53
```

```
customers_per_city.collect()
```

```
[('Bangalore', 3), ('Hyderabad', 1), ('Ahmedabad', 1), ('Delhi', 1)]
```

```
# CountByValue
```

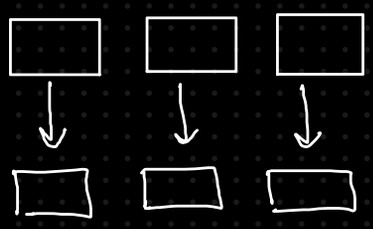
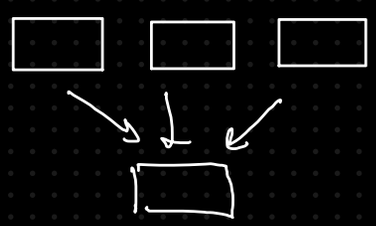
```
cust_per_city = parsed_rdd.map(lambda row:row[2]).countByValue()
```

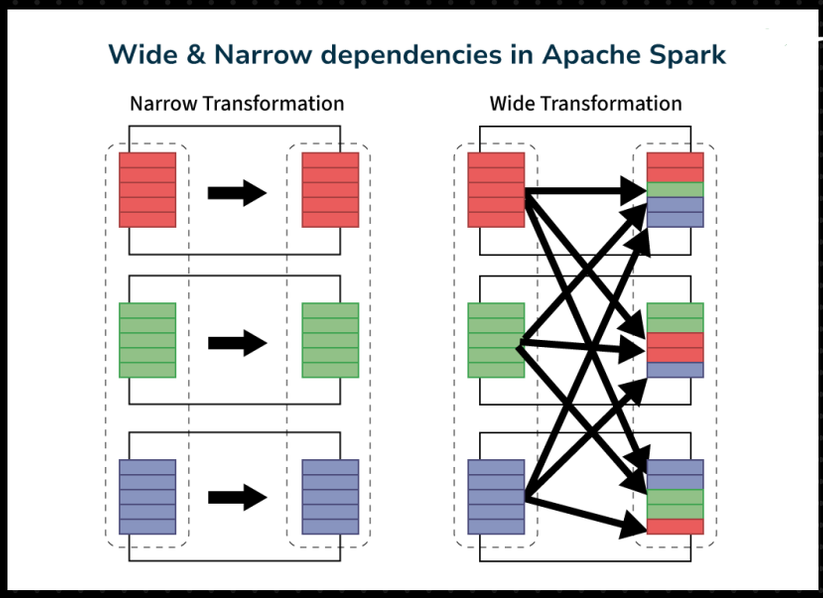
```
cust_per_city
```

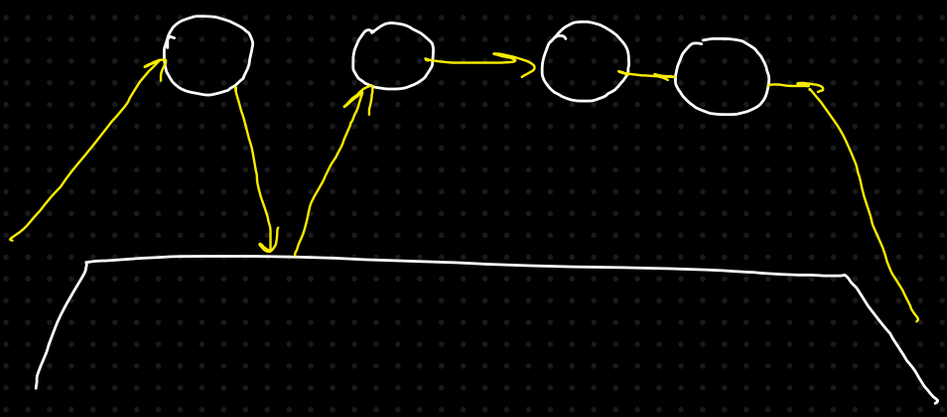
```
defaultdict(int, {'Bangalore': 3, 'Hyderabad': 1, 'Ahmedabad': 1, 'Delhi': 1})
```

Transformation: Narrow vs Wide

Transformations in Spark are operations that create new RDD.

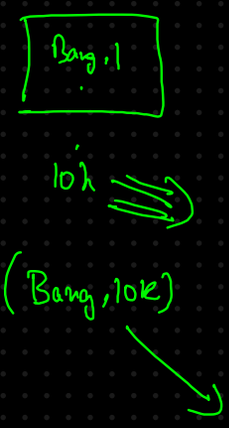
	Narrow	Wide
	filter	reduceByKey
Data movement	Data is not shuffled across partitions	Data is shuffled across partitions, requiring network I/O
Dependency	Each child partition depend on single parent partition	Each child partition depends on multiple parent partition
		
Performance	Faster as it avoids data shuffling	Slowers due to network & disk I/O
	map, filter, flatmap	reduceByKey, GroupByKey





final aim

1. try to have very less wide transformation
2. narrow down the data so that data getting shuffled is less.



Job, tasks and stages in Spark UI

Job → no. of actions

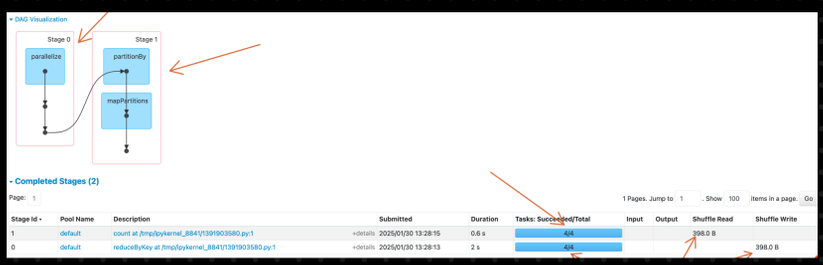
↓

Stages → no. of wide transformation + 1

↓

Tasks → no. of partition used

The same can be optimized / diff based on platform



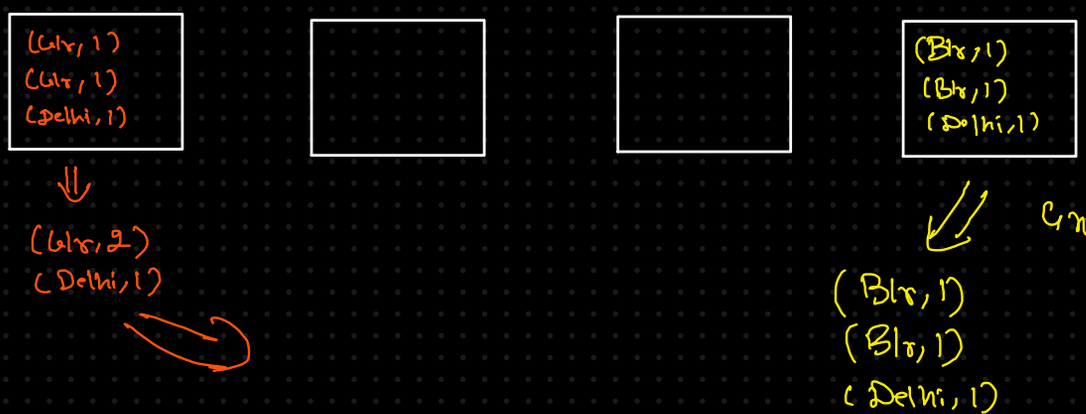
☆ Interview Question

ReduceByKey and GroupByKey

479 bytes → 2 kb → for 1mb file

679 bytes → 98 kb → for 500mb file

	ReduceByKey	GroupByKey
Output	Returns a key-value pair with aggregated values	Returns a key value pairs where values are collections
	Local aggregation is there.	no local aggregation

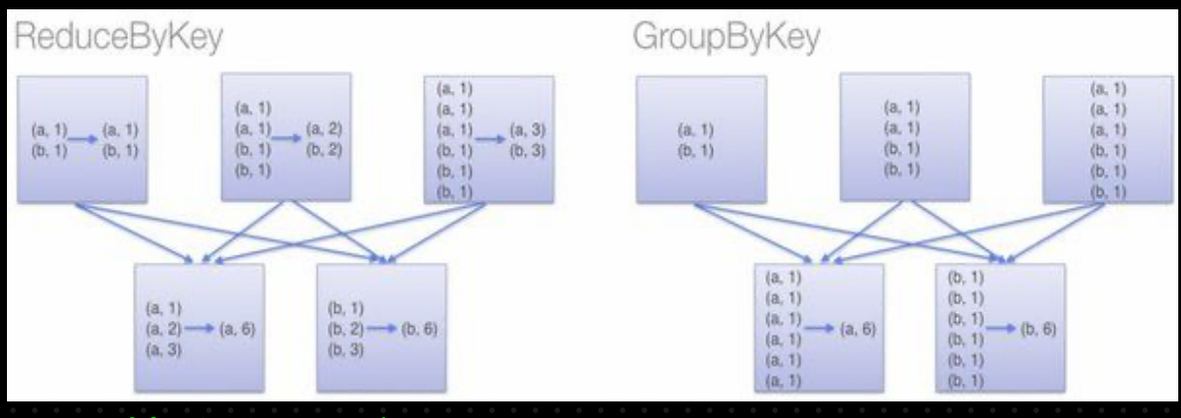


minimize data transfer by doing local aggregation

Less data for shuffling + I/O for disk/network

Group all the values based on key & aggregation happens after shuffle.

more data for shuffling



max sum min

Note: For non-commutative and non-associative property like say median, avg etc. we have to use groupbykey

☹ of groupbykey

1. chances of OOM error
2. Shuffling lots of data
3. not doing / restricting parallelism.

Reduce by Key

	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
el_32253/266034041.py:1 +details	2025/01/30 14:01:57	2 s	5/5			2.1 KiB	
y at el_32253/266034041.py:1 +details	2025/01/30 14:01:44	12 s	5/5	544.6 MiB			2.1 KiB

Group by Key

	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
el_32253/1858560816.py:1 +details	2025/01/30 14:03:33	0.3 s	5/5			96.1 KiB	
at el_32253/1858560816.py:1 +details	2025/01/30 14:03:23	10 s	5/5	544.6 MiB			96.1 KiB

~ 49x

Increasing or Decreasing Number of Partition

no. of partitions affect the parallelism
≈ no. of tasks

Increase the number the partitions

2 gb file ⟶ 16 partitions

you have a great 30 node cluster

its better to use all the nodes

⇒ can be helpful to increase llism

Cluster should be able to handle it.

Decreasing the number of Partition

1 tb file ⟶ 8000 partition

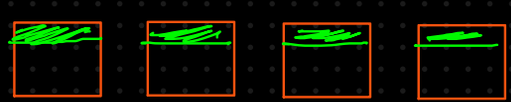
& say a 50 node cluster

⇒ decreasing is better

Say we have 100 partitions

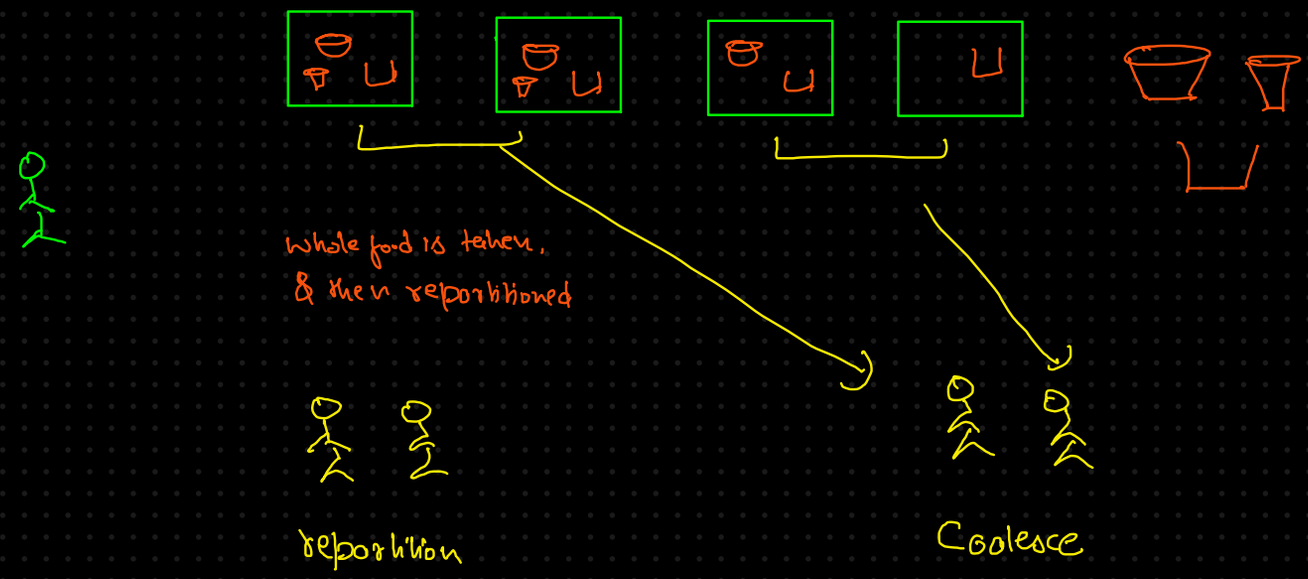
8 node cluster

100 partitions ⇒ filter



Avoid Sparse partition but should aim for dense partition

Repartition vs Coalesce



Aspect	Repartition	Coalesce
Definition	Shuffles data across nodes to **increase or decrease** partitions.	Combines partitions **without shuffle**, typically reducing their number.
Use Case	To balance load or increase parallelism.	To optimize performance by reducing partitions.
Shuffling	Yes, full shuffle across the cluster.	No shuffle; only adjacent partitions are merged.
Performance	Slower due to shuffling overhead.	Faster as it avoids shuffling.
Scalability	Used to increase partitions or even redistribute them.	Used only for decreasing partitions.
When to Use	After reading unbalanced data (e.g., uneven partitions).	When shrinking partitions after a filter operation.

we prefer Coalesce when we have to reduce no. of partition

Optimization in Spark is a incremental process