Spark fundamentals

notes by Preeti Mahto

linkedin.com/in/preeti-mahto/

Spark Overview (Theory)

1) What is Apache Spark ?

2) Why Apache spark ? what problem does it solve ?

Apache spark is a unified computing engine and set of libraries for parallel data processing on computer cluster.

- What is unified ?

→ Spark is designed to support wide range of task over the same computing engine.

→ Data scientists, data analysts and data engineers, all can use the same platform for their analysis transformation or modelling.

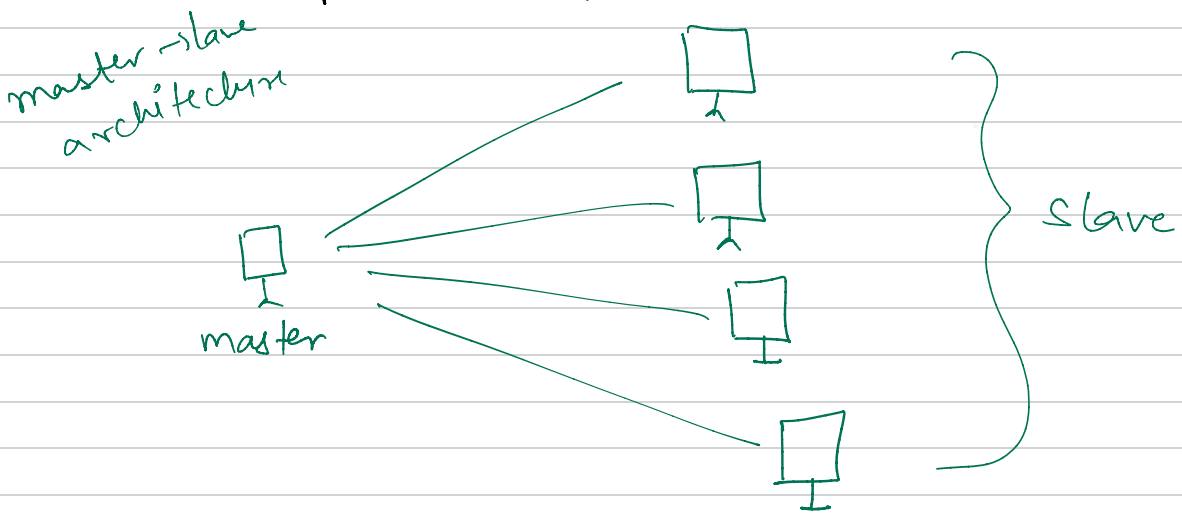
- What is computing engine ?

→ Spark is limited to a computing engine. It doesn't store the data.
→ spark can connect with different data sources like S3, HDFS, JDBC/ODBC, Azure storage etc.
→ spark works with almost all data storage system.

- What is parallel data processing ?

→ data is getting processed parallelly.

- What is computer cluster ?



Why Apache spark ?

Initially we had databases like Oracle, Teradata, exadata, mySQL which could only store structured data.

But now, mostly unstructured and semi-structured data is generated, text, csv, image, video, json YAML.

So, the problem was the above databases could handle only tabular format data. and not other forms of data.

3 Vs of Big Data

1) Velocity → speed of data processing. 1GB/sec, 5TB/hr.

2) Variety → number of types of data. {structured, unstructured, semi}

3) Volume → amount of data, 1GB, 4GB, 5TB.

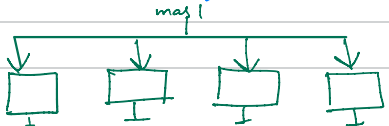
Before, there was data warehousing, where we used to do ETL
But now, we have data lake, where we do ELT.

Issues

1) Storage
2) Processing → RAM, CPU

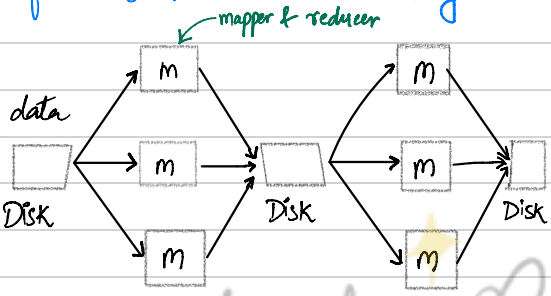
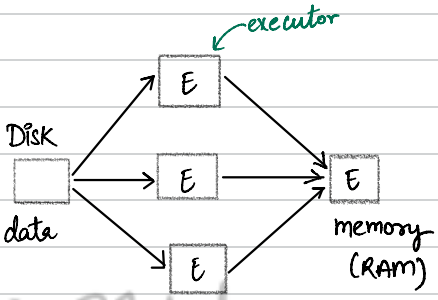
To solve these issues, we had two options:
- ↳ monolithic approach
- ↳ distributed approach.

Monolithic	Distributed
Vertical scaling	horizontal scaling
expensive	Economical
low availability	high availability.

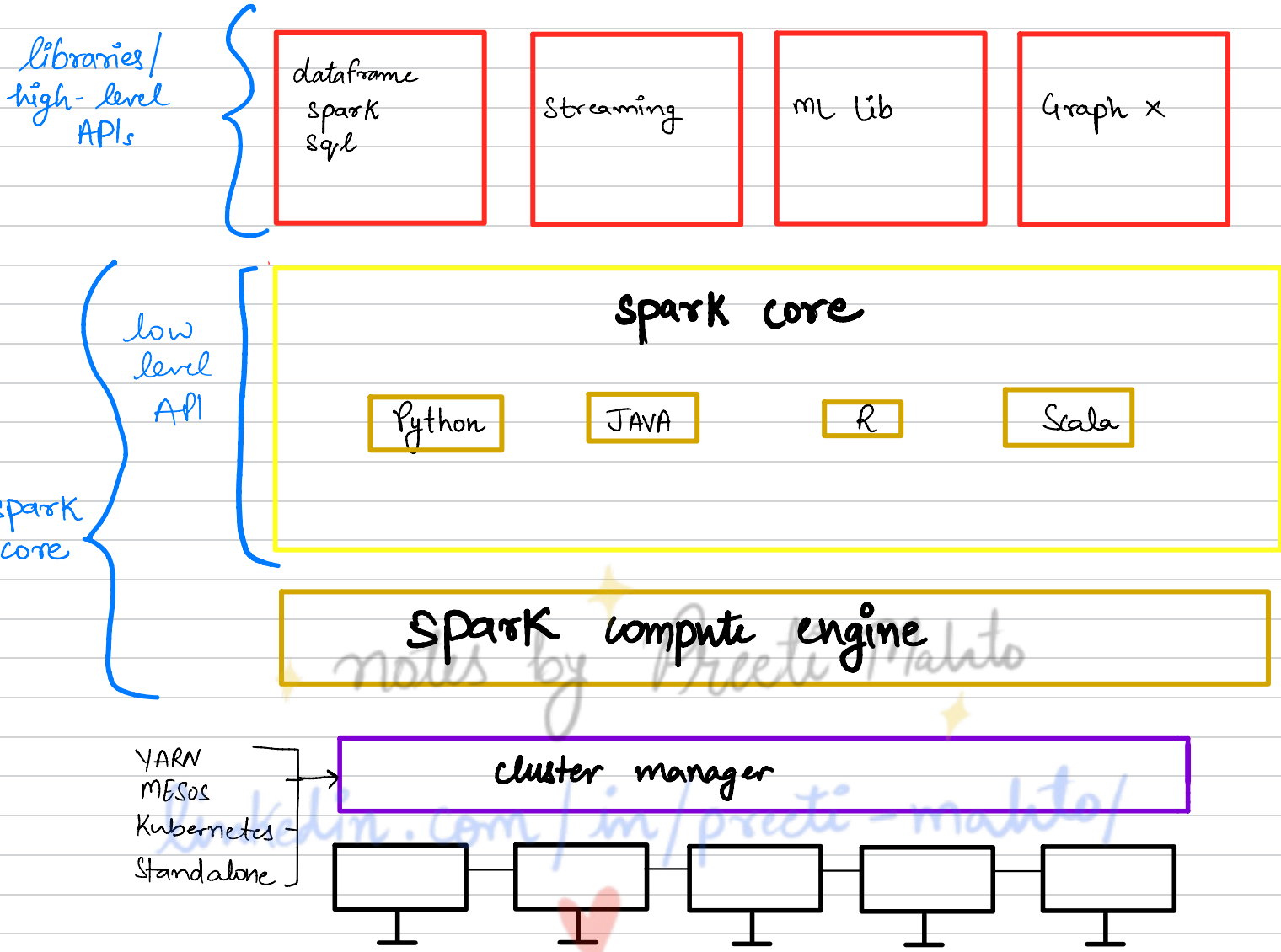


Why we moved from Hadoop to Spark?

Misconceptions → Hadoop is a database
→ spark is 100 times faster than hadoop.
→ spark processes data in RAM but hadoop don't

Parameter	Hadoop	Spark
Performance	Hadoop is slower than spark. Because it writes the data back to disk and read again from disk to in-memory.  There will no latency if the data volume is less.	Spark is faster than hadoop because spark do all the computation in-memory 
Batch / Streaming	Build for batch data processing.	Build for batch as well as streaming data processing.
Ease of use	Difficult to write code in hadoop	Easy to write and debug code. we have interactive shell to develop and test. Spark provides high level and low level API
Security	uses Kerberos authentification and ACL authorization.	Doesn't have security feature. uses HDFS storage, from there it gets ACL level control uses YARN for negotiating resource it gets Kerberos authentication.
Fault Tolerance	It is having block of data and replication factor to handle the failure.	uses DAG to provide fault tolerance.

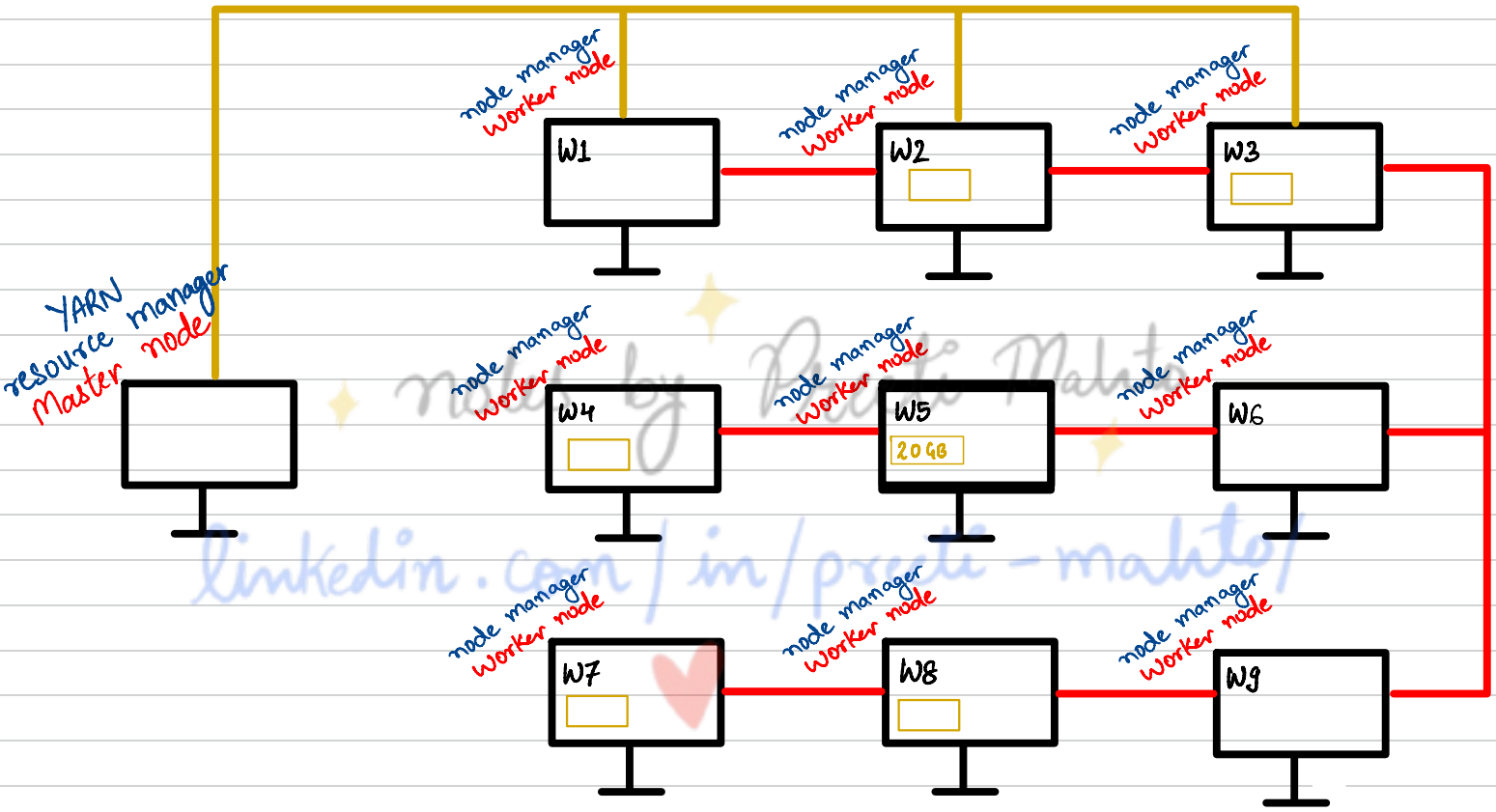
Spark Ecosystem



Spark architecture

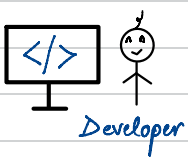
configuration of each machine:
CPU → 20 core
RAM → 100 GB

configuration of the cluster:
CPU → 20 x 10 = 200 core
RAM → 100 x 10 = ~1TB



Step 1:

→ Runs spark submit command asking for:
Driver - 20 GB
Executor - 25 GB
no. of executor - 5
CPU core - 5
to YARN.



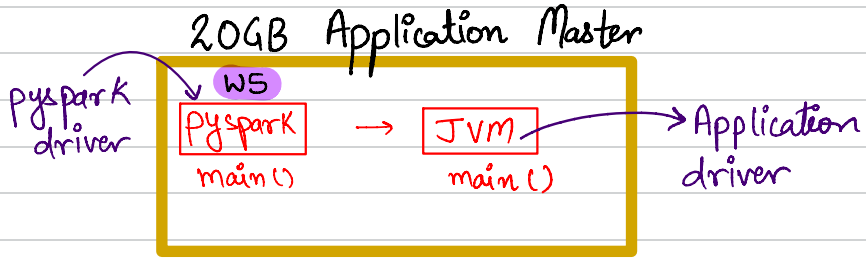
Step 2:

→ Randomly choose any one worker node and asks to create a container of 20 GB RAM.

YARN

suppose it chooses W5.

INSIDE CONTAINER

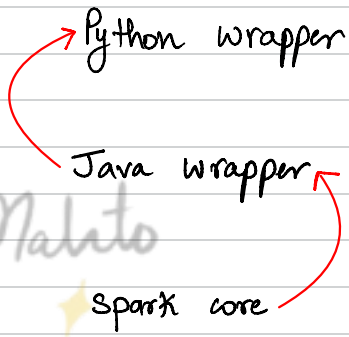


If we write code in python then main driver is Pyspark.

If we write code in JAVA/Scala then main driver is JVM main()

But Pyspark is written in scala so pyspark will indirectly run on JVM.

So, Application driver will always be there no matter what the language is but when we'll write in python, then along with application driver pyspark driver will also be created.



Step 3 :

→ creation of main method in the container.

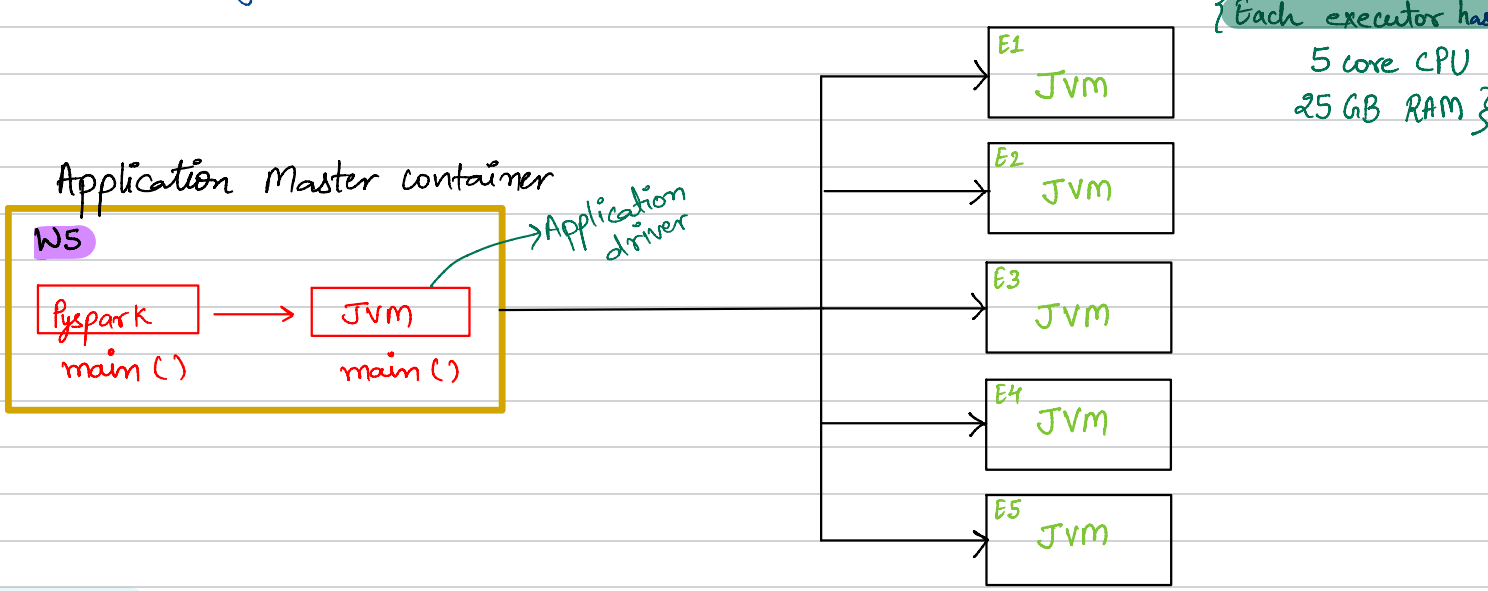
Step 4 :

→ main() goes back to YARN and asks to complete rest of the request.

Executor - 25 GB
no. of executor - 5
CPU core - 5

Step 5 :

→ It gets 5 more containers (executors) based on the request. suppose it got it in W2, W3, W4, W7, W8.

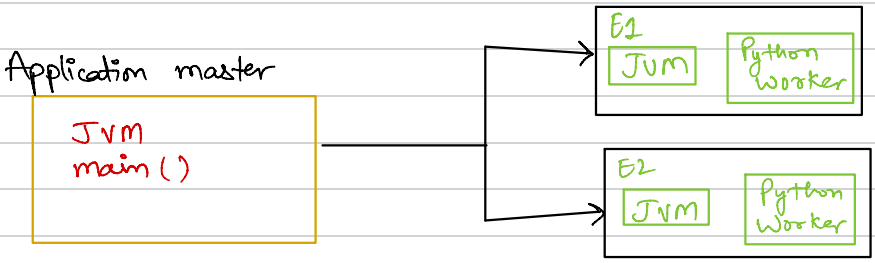


{Each executor has:
5 core CPU
25 GB RAM}

(udf)

Suppose we've have written user defined functions in our application. when it will go in runtime, it will throw errors because of the absence of python worker.

So, to rectify this we'll have python worker too in our executors.



Transformation and action

Potential interview questions :

1. What is transformation and how many types of transformation do we have ?
2. What happens when we use group by or join in transformation ?
3. How jobs are created in spark?

Types of Transformations

1) Narrow dependency : Transformation that doesn't require data movement between partitions , e.g. filter, select, union, map()

2) Wide dependency : Transformation that require data movement between partitions , e.g. groupby , reduce by key, join , distinct , etc.

example of Narrow dependency :

Que 1. Display employees whose age is less than 18 ?

Que 2. find out the total income of each employee.

Id	Name	Age	Income	Source
1	Manish	26	7500	Job
2	Ramshan	16	35000	Job
3	Mukesh	35	12000	Teaching
7	Nikita	55	9000	Youtube
15	Vikash	15	25000	freelancing
3	Mukesh	35	29000	Job
15	Vikash	15	40000	Job
1	Manish	26	25000	Youtube
8	Roshini	42	62000	Job
2	Ramshan	16	16000	Teaching

Data :
200 MB / 128 = 2 partitions

executor 1

Id	Name	Age	Income	Source
1	Manish	26	7500	Job
2	Roshan	16	35000	Job
3	Mukesh	35	12000	Teaching
7	Nikita	55	9000	Yo
15	Vikash	15	25000	Youtube

executor 2

Id	Name	Age	Income	Source
3	Mukesh	35	29000	Job
15	Vikash	15	40000	Job
1	Manish	26	25000	Youtube
8	Roshini	42	62000	Job
2	Ramshan	16	16000	Teaching

Ans 1.

output

Id	Name	Age	Income	Source
2	Ramshan	16	35000	Job
5	Vikash	15	40000	Job
15	Vikash	15	25000	freelancing
2	Ramshan	16	16000	Teaching

executor 1

Id	Name	Age	Income	Source
1	Manish	26	7500	Job
2	Roshan	16	35000	Job
3	Mukesh	35	12000	Teaching
7	Nikita	55	9000	Yo -
15	Vikash	15	28000	Youtube

executor 2

Id	Name	Age	Income	Source
3	Mukesh	35	29000	Job
15	Vikash	15	40000	Job
1	Manish	26	25000	Youtube
8	Roshini	42	62000	Job
2	Ranshan	16	16000	Teaching

Ans 2.

Id	Name	Age	Income
1	Manish	26	32500
2	Roshan	16	51000
3	Mukesh	35	41000
7	Nikita	55	9000
15	Vikash	15	65000
8	Roshini	42	62000

DAG Creation and Lazy Evaluation

```
1  flight_data=spark.read.format("csv")\
2              .option("header","true")\
3              .option("inferSchema","true")\
4              .load("dbfs:/FileStore/tables/flight_data.csv")
5
6  flight_data_repartition= flight_data.repartition(3)
7
8  us_flight_data=flight_data.filter("DEST_COUNTRY_NAME=='United States'")
9
10 us_india_data=us_flight_data.filter((col("ORIGIN_COUNTRY_NAME")=='India') |
   (col("ORIGIN_COUNTRY_NAME")=='Singapore'))
11
12 total_flight_ind_sing= us_india_data.groupby("DEST_COUNTRY_NAME").sum("count")
13
14 total_flight_ind_sing.show()
```

Actions

Read

wide dependency

Narrow dep.

transformations

wide dep.

Action

Spark Application

{consists of actions and transformations}

DAG → Directed Acyclic Graph.

↳ created for each job

↳ job is created when action is being hit.

Spark UI → Displays Application status.

Lazy Evaluation :

→ Application won't create job until action is being hit.
→ Application won't do anything at all until it gets action. Otherwise nothing is going to happen.

→ Optimizes code in the end.

Spark SQL Engine

Potential interview questions :

1) What is catalyst optimizer | spark SQL Engine ?

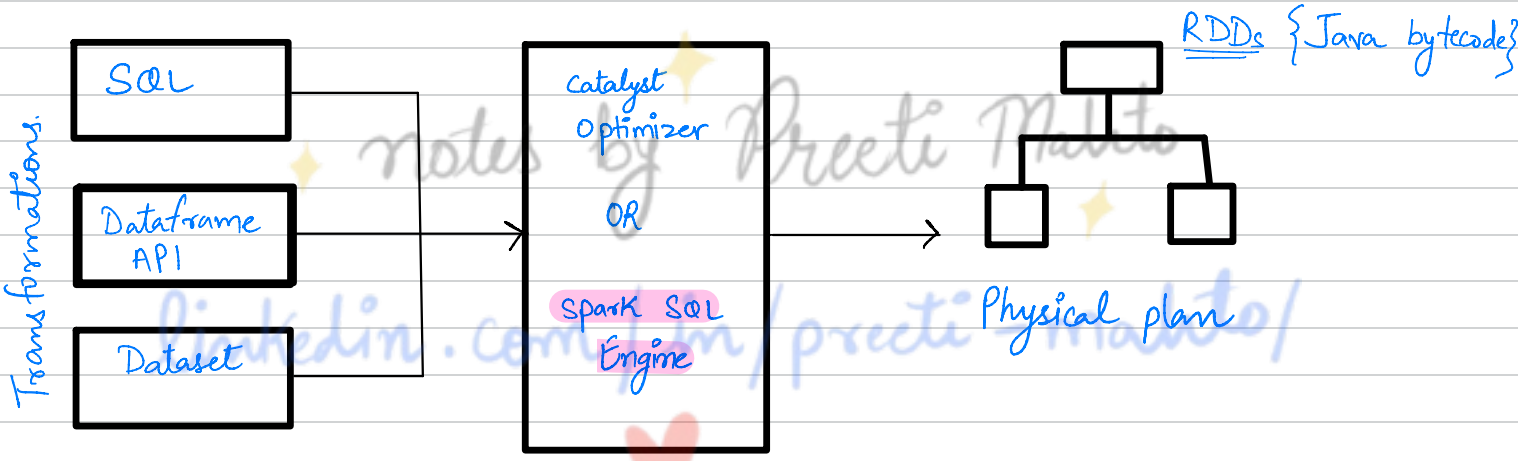
2) Why do we get analysis exception error ?

3) What is catalog ?

4) What is physical planning / spark plan ?

5) Is spark SQL engine a compiler ? Yes

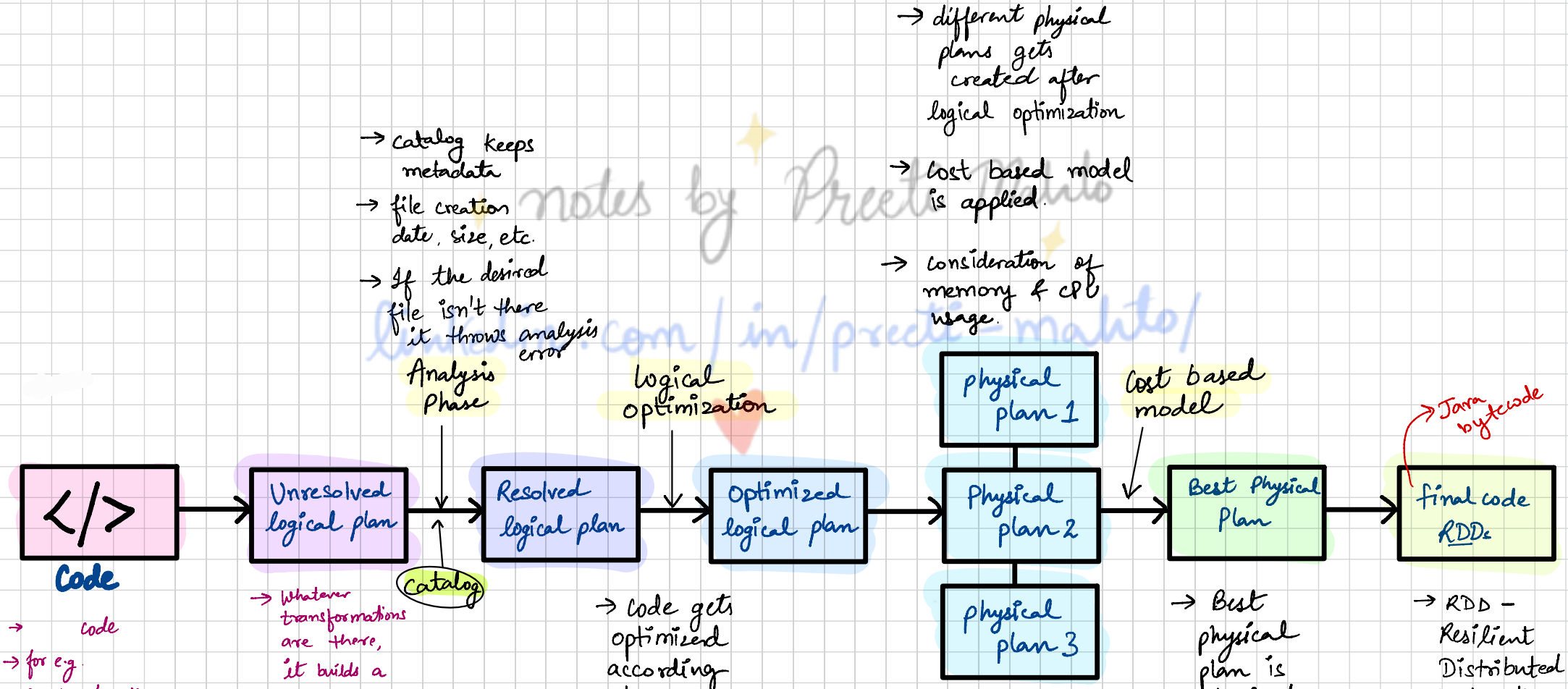
6) How many phases are involved in spark SQL engine to convert a code into Java bytecode ?



four phases of spark SQL Engine :

1) Analysis
2) logical planning
3) Physical planning
4) Code Generation.

Spark SQL Engine



RDDs

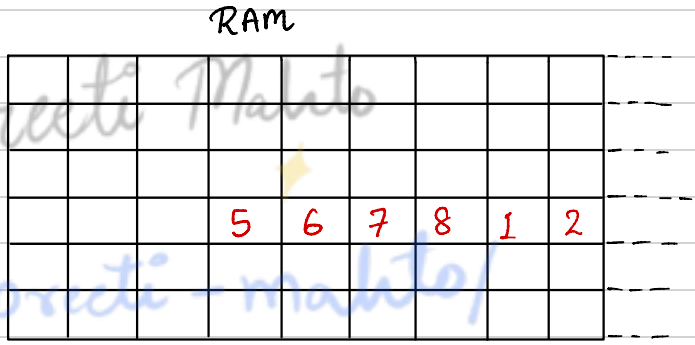
Potential interview Questions :

1) What is RDD ?
2) When do we need an RDD ?
3) features of an RDD ? Immutable, lazy
4) What is Dataframe / Dataset ?
5) Why we should not use an RDD ?

How list is stored in RAM.

Example :

list = [5, 6, 7, 8, 1, 2]
0 1 2 3 4 5

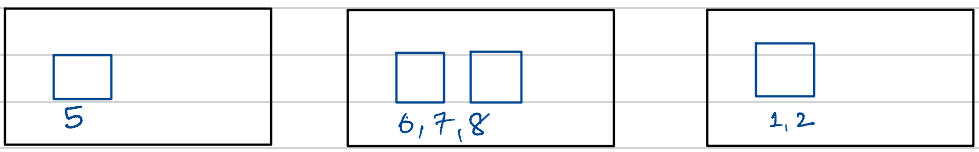


So, RDD is also a data structure.

Imagine we have 500 MB of data, than $\frac{500}{128}$ = 4 partitions {128 MB = 1 block size}

So, these partitions will be stored on HDD of cluster.

If we want to store same data on RDD, then it will be stored in the following way :



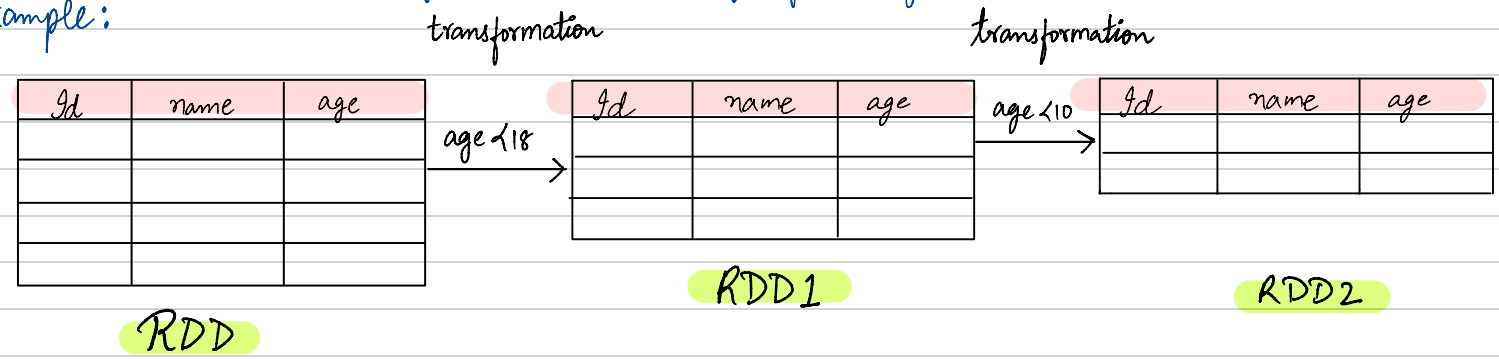
Now this is RDD, because it is distributed on the cluster.

Resilient → In case of failure, it knows how to recover, fault tolerant.
Distributed → Data is distributed over the cluster.
Dataset → Actual data.

Why RDD is Resilient ?

So, RDDs are immutable i.e it can't be changed but transformed. Every time there's a transformation, a new copy of RDD gets created.

Example:



Disadvantages of using RDD.

→ Spark doesn't do any optimization for RDD we have to manually do the optimizations.

→ we have to tell "how to" (RDD asks — कैसे करना है?) and "what to" (Dataframe tells — क्या करना है?)

Advantages of RDD

→ Works well with unstructured data
→ It is type safe. {gives error at compile time} {SQL give error at runtime}
→ flexibility and full control over data.

When do we need RDD ?

→ When we want full control on our data.
→ When our data is unstructured.

Why we shouldn't use RDD ?

```
DataFrame
data.groupBy("dept").avg("age")

SQL
select dept, avg(age) from data group by 1

RDD
data.map { case (dept, age) => dept -> (age, 1) }
    .reduceByKey { case ((a1, c1), (a2, c2)) => (a1 + a2, c1 + c2)}
    .map { case (dept, (age, c)) => dept -> age / c }
```

} optimization, Readability, anyone can use including data analysts, data scientists etc.

→ high complexity, few data engineers know how to code in RDD.

Spark session vs spark context

Feature	SparkContext	Spark Session
Introduction	Introduced in earlier versions of spark (before 2.0)	Introduced in Spark 2.0 as a unified entry point.
Purpose	Main entry point for spark functionality.	Unified entry point to spark functionality including SQL, streaming and more.
APIs Accessed	Primarily used to access core Spark APIs (RDDs)	Can access all APIs : RDDs, Dataframes, Datasets, SQL & Hive.
Use Case	low-level transformations and actions using RDD.	High level APIs like Dataframe, Dataset, and SQL queries
Session Management	Doesn't manage hive contexts or SQL sessions directly.	Manages Hive, SQL, Streaming, and more under a single object.
Example	`sc = SparkContext(appName = "App")`	`spark = SparkSession.builder.appName("App").getOrCreate()`
Preferred in New Projects	No, its older and lower-level.	Yes, recommended since Spark 2.0

Summary :

SparkSession is a wrapper that internally creates a SparkContext, SQLContext, and HiveContext.

If you're using Spark 2.0 or later, always use SparkSession, it's more powerful, flexible, and future-proof.

Jobs, Stages and Tasks

Potential Interview questions :

Q1. What is application, job, stage and task in spark ?

Q2. How many jobs will be created in the given question ? Two

Q3. How many stages will be created ? Job 1 → 1 stage , Job 2 → 3 stages

Q4. How many tasks will be created ? Job 1 → 1 task
Job 2 → 203 tasks

Ans 1.

Application

- top-level program that we run.
- contains spark code : main driver program + transformations / actions.
- example : If we run a PySpark script that does ETL, that script is one application.

Job (s)

- A job is triggered whenever an action is called on a spark RDD or a Dataframe.
- example : collect(), count(), save()
- each action starts a new job.
- A job is divided into one or more stages.

Stage (s)

- Spark divides jobs into stages based on shuffle boundaries (i.e. where data needs to be redistributed.)
- Two main types :
  1) ShuffleMapStage – outputs data to be shuffled.
  2) ResultStage – returns the final result to the driver.
- example : If a job involves a .groupBy() (which causes a shuffle), spark will break it into two stages.

NOTE: For every job created there will be min. one Stage and inside stage there will be min. one task created.

Task (s)

- Smallest unit of work in Spark.
- Single operation on a partition of data (like applying a function)
- A stage contains multiple tasks, one per data partition.
- 10 partitions → 10 tasks

Ans 2.

```
from pyspark.sql.functions import *
spark = SparkSession.builder.master("local[5]")\
        .appName("testing")\
        .getOrCreate()

employee_df=spark.read.format("csv")\
        .option("header","true")\
        .load("C:\\Users\\nikita\\Documents\\data_engineering\\spark_data\\employee_file.csv")
print(employee_df.rdd.getNumPartitions())
employee_df = employee_df.repartition(2)
print(employee_df.rdd.getNumPartitions())
employee_df=employee_df.filter(col("salary")>90000)\
        .select("id","name","age","salary")\
        .groupby("age").count()
employee_df.collect()
input("Press enter to terminate")
```

action → Job 1

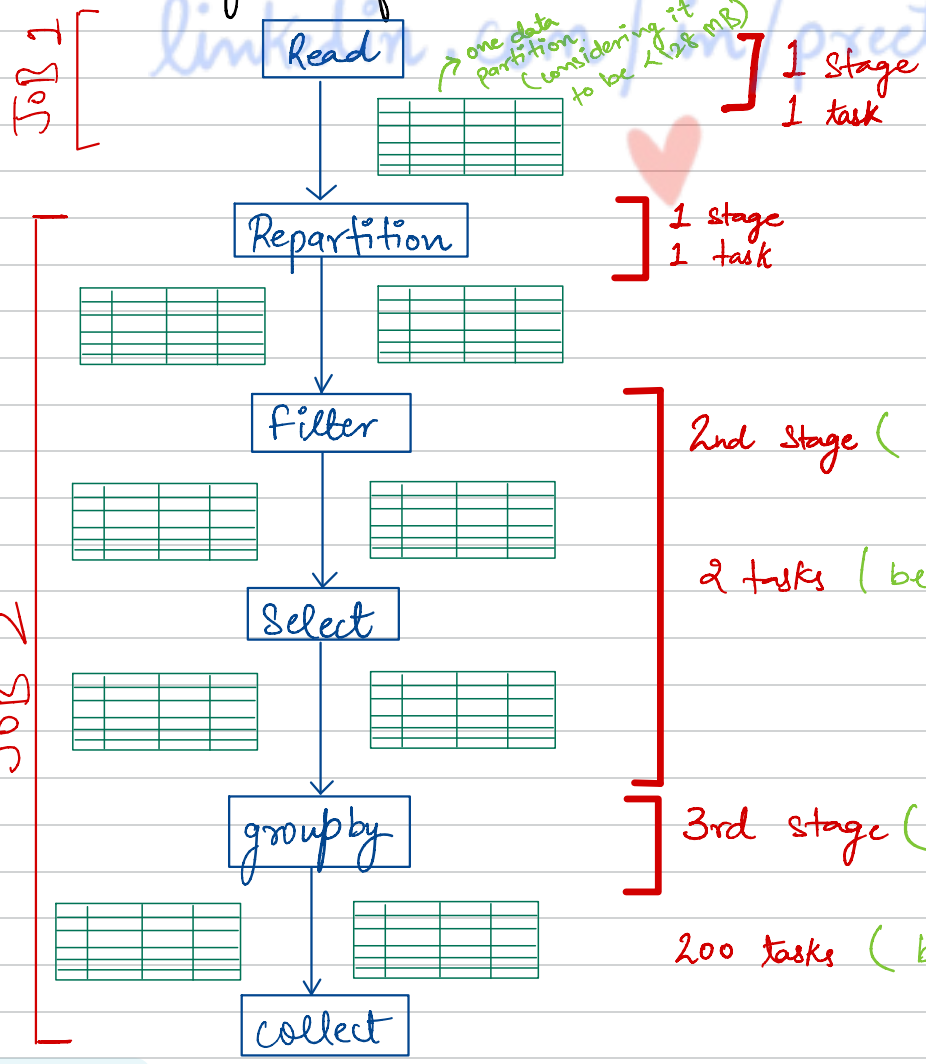
action → Job 2

Two jobs will be created.

NOTE:
Whenever there will be wide dependency transformation, (join, groupby, etc) there will be minimum 200 partitions which is created.
Me

Ans 3.

Workflow of the above code:

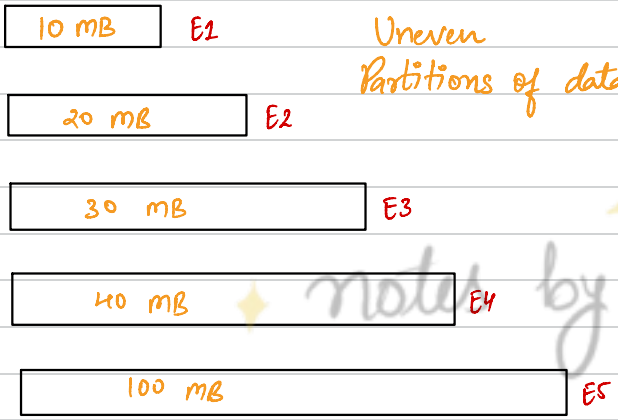


Repartition vs Coalesce

Potential interview questions :

1) What is repartitioning in spark ?
2) What is coalesce in spark ?
3) Which one will you choose and why ?
4) Repartitioning vs coalesce.

What was the need of repartition and coalesce ?



Uneven Partitions of data.

When there is a lot of data, then data gets divided into uneven partitions.

While processing this data, E1 will finish the processing first and stay idle till E5 finishes its processing, So there's delay in data processing, also resources wastage.

→ why this problem even arises ?
Because of skewness in data.

Imagine there's a company which sells different products. So they will have records of all the products but their best selling product's data will be large. So, while processing products data there might be delay in processing the best selling product's data which will lead to overall data processing delay.

Repartition and coalesce with an example :

Name	Country
A	India
B	Us
C	UK
D	India
E	Singapore
f	Us

Partition 1 : A, B, C

Partition 2 : D, E, f

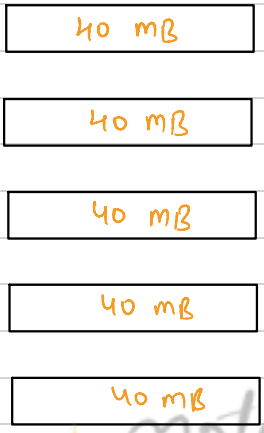
df. repartition (3)
partition 1 : A, D
partition 2 : B, f
partition 3 : C, E

df. coalesce (1)
partition 1 : A, B, C, D, E, f

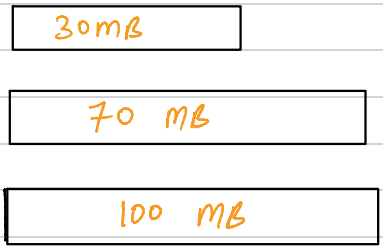
Ans 1. Repartitioning is basically when Spark reshuffles the data to create a new number of partitions. It actually moves the data around across the cluster, which can be a bit heavy on performance but its useful when we want to increase parallelism or balance out uneven data, like before doing a join or groupby.

for example:

df.repartition(5) → Spark will redistribute the data into 5 partitions evenly



Ans 2. Coalesce is used in Spark to reduce the number of partitions without causing a full shuffle. It simply tries to merge adjacent partitions, which is much faster and more efficient than repartitioning. It is commonly used when writing final results to a single file, like a csv.



Ans 3. It depends on different use cases:

If we need to increase partitions for better parallelism, like before a heavy join, groupby, or shuffle-heavy transformation, then we'll go for repartition, because it evenly redistributes data across partitions.

But if we just want to reduce the number of partitions, for example write fewer output files or optimize a write operation, we'll go for coalesce, because, it's more efficient, it avoids a full shuffle & is faster.

Ans 4.

Feature	Repartition	Coalesce
Purpose	Increases or decreases no. of partitions.	Decreases number of partitions only.
Shuffling	full shuffle of data across all nodes.	minimizes shuffle, avoids full shuffle.
Use Case	When increasing partitions or evenly redistributing data.	When reducing no. of partitions for optimized output.
Performance Cost	Expensive due to full shuffle.	Inexpensive due to limited movement.
Typical Scenario	Before wide transformations like join, groupby	Before writing data to disk.
Data redistribution	Even redistribution across partitions.	Narrows existing partitions. (merging adjacent ones)

Join in Spark

How data shuffling happens in join (wide dependency transformation)

Dataframe 1

$\frac{500\ MB}{128\ MB}$ = 4 partitions

P1 P2 P3 P4

Dataframe 2

$\frac{500\ MB}{128\ MB}$ = 4 partitions

P1 P2 P3 P4

P1

id	name
1	manish
7	Ramesh
10	Rahul

P2

id	name
2	Ram
6	Meera

P3

id	name
201	Rohan
205	Vikash

P4

id	name
12	x
202	y
206	z

P1

id	Salary
2	1000
205	2000

P2

id	Salary
6	3000
206	4000
10	5000

P3

id	Salary
1	6000
7	7000
11	8000

P4

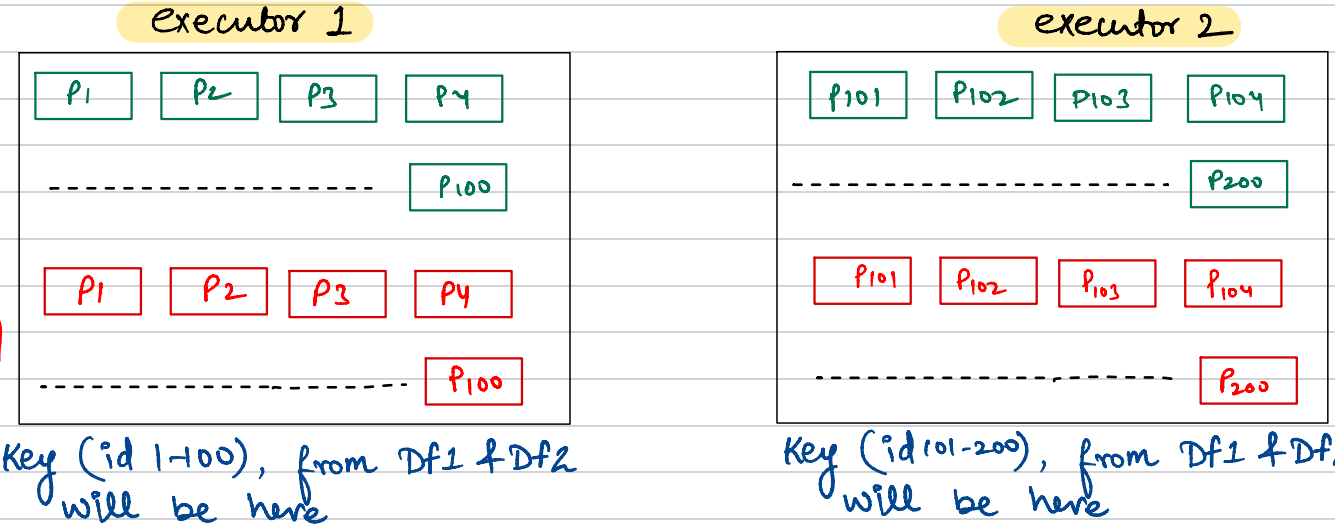
id	Salary
201	9000
202	1000
12	1100

We want to create a dataframe 3 with id, name & salary, in order to do that we will use join b/w Df1 and Df2. Either we have to put data of E1 into E2 or vice versa.

But how is that gonna happen?

So, Whenever we join, automatically 200 partitions is created

∴



Key (id 1-100), from Df1 & Df2 will be here

Key (id 101-200), from Df1 & Df2 will be here

So, what's gonna happen with id >200 ?

now, we'll divide Key (id) by the no. of partitions and get the remainder.

So, $\frac{201}{200}$ = 1st partition

$\frac{405}{200}$ = 5th partition

$\frac{909}{200}$ = 9th partition

$\frac{1103}{200}$ = 103rd partition

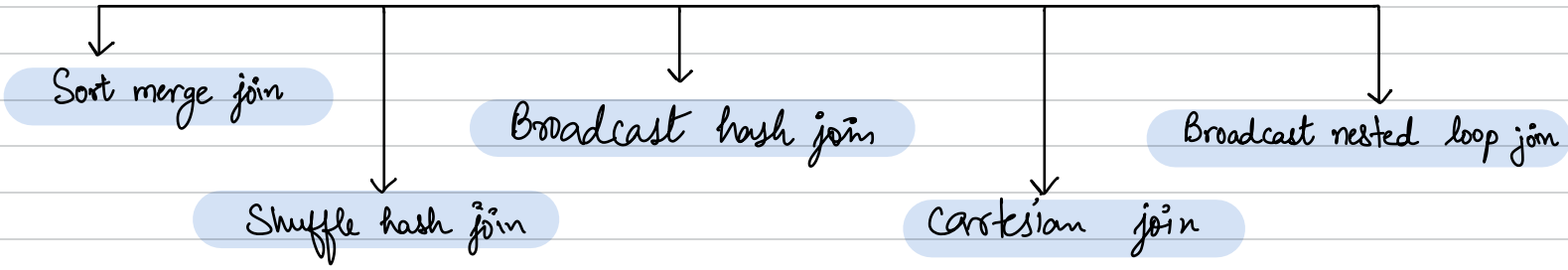
$\frac{9392}{200}$ = 192nd partition

Note :

When there's a lot of data, the system might get chocked and performance decreases

Me

Join Strategies in spark



1) Sort merge join

Default join implementation since spark 2.3.

Steps :

→ Sort both datasets based on join key
→ Merge step by step
→ Results.

Example :

Dataframe 1 Order

orderId	customerId	amount
1	101	50
2	102	100
3	101	75
4	103	60
5	102	120

Dataframe 2 Customer

customerId	Customer Name
101	John
102	Emma
103	Michael

Sort both datasets based on join key (here customerId)

Dataframe 1 Order

orderId	customerId	amount
1	101	50
2	101	75
3	102	100
4	102	120
5	103	60

Dataframe 2 Customer

customerId	Customer Name
101	John
102	Emma
103	Michael

MERGE Step 1 :

Dataframe 1 Order

OrderId	customerId	amount
1	101	50
2	101	75
3	102	100
4	102	120
5	103	60

Dataframe 2 Customer

customerId	Customer Name
101	John
102	Emma
103	Michael

Results

OrderId	CustomerId	amount	customerName
1	101	50	John

MERGE Step 2 :

Dataframe 1 Order

OrderId	customerId	amount
1	101	50
2	101	75
3	102	100
4	102	120
5	103	60

Dataframe 2 Customer

customerId	Customer Name
101	John
102	Emma
103	Michael

Results

OrderId	CustomerId	amount	customerName
1	101	50	John
2	101	75	John

MERGE Step 3 :

Dataframe 1 Order

OrderId	customerId	amount
1	101	50
2	101	75
3	102	100
4	102	120
5	103	60

Dataframe 2 Customer

customerId	Customer Name
101	John
102	Emma
103	Michael

Results

OrderId	CustomerId	amount	customerName
1	101	50	John
2	101	75	John
3	102	100	Emma

MERGE Step 4 :

Dataframe 1 Order

OrderId	customerId	amount
1	101	50
2	101	75
3	102	100
4	102	120
5	103	60

Dataframe 2 Customer

customerId	Customer Name
101	John
102	Emma
103	Michael

Results

OrderId	CustomerId	amount	customerName
1	101	50	John
2	101	75	John
3	102	100	Emma
4	102	120	Emma

MERGE Step 5 :

Dataframe 1 Order

OrderId	customerId	amount
1	101	50
2	101	75
3	102	100
4	102	120
5	103	60

Dataframe 2 Customer

customerId	Customer Name
101	John
102	Emma
103	Michael

Results

OrderId	CustomerId	amount	customerName
1	101	50	John
2	101	75	John
3	102	100	Emma
4	102	120	Emma
5	103	60	Michael

2) Shuffle Hash Join

Default join Implementation

Steps :

i) Partition
ii) Shuffling
iii) Hashing and In-memory join
iv) Result Aggregation

Example :

Dataframe 1 Order

OrderId	customerId	amount
1	101	50
2	102	100
3	101	75
4	103	60
5	102	120

Dataframe 2 Customer

customerId	Customer Name
101	John
102	Emma
103	Michael

I. Partition

Dataframe 1 Order

OrderId	customerId	amount
1	101	50
2	102	100
3	101	75
4	103	60
5	102	120

OrderId	customerId	amount
1	101	50
3	101	75

P1

OrderId	customerId	amount
2	102	100
5	102	120

P2

OrderId	customerId	amount
4	103	60

P3

Dataframe 2 Customer

customerId	Customer Name
101	John
102	Emma
103	Michael

customerId	Customer Name
101	John

P1

customerId	Customer Name
102	Emma

P2

customerId	Customer Name
103	Michael

P3

III. Hashing

Smaller dataset will get converted to hash tables.

Node 1 101 : [(101, John)]

Node 2 102 : [(102, Emma)]

Node 3 103 : [(103, Michael)]

Joining

node 1

customerId	Customer Name
101	John

OrderId	customerId	amount
1	101	50
3	101	75

node 2

customerId	Customer Name
102	Emma

OrderId	customerId	amount
2	102	100
5	102	120

node 3

customerId	Customer Name
103	Michael

OrderId	customerId	amount
4	103	60

IV. Results

node 1

OrderId	customerId	amount	Customer Name
1	101	50	John
3	101	75	John

node 2

OrderId	customerId	amount	Customer Name
2	102	100	Emma
5	102	120	Emma

node 3

OrderId	customerId	amount	Customer Name
4	103	60	Michael

Results

OrderId	CustomerId	amount	customerName
1	101	50	John
2	101	75	John
3	102	100	Emma
4	102	120	Emma
5	103	60	Michael

3) Broadcast hash join

Potential interview questions :

1) Why do we need broadcast hash join ?
2) How does broadcast join works ?
3) Difference b/w broadcast hash join and shuffle hash join ?
4) How can we change broadcast size of table ?
5) What are the possibilities of failure of Broadcast hash join ?

Ans 1. When joining two datasets, spark normally shuffles data across the cluster to align keys, this is expensive and slow, especially with big data.

But when one dataset is small enough to fit into memory, we can :
Broadcast it to all worker nodes,
- Avoid shuffling the larger dataset, and
- Perform fast, local joins on each node.

Ans 2.

Step 1 : Identifies small table
- Spark detects that one df is small and the other is large.
- It broadcasts this small table to all worker nod, a full copy goes to each node.

example :

Dataframe 1 Order

OrderId	customerId	amount
1	101	50
2	102	100
3	101	75
4	103	60
5	102	120

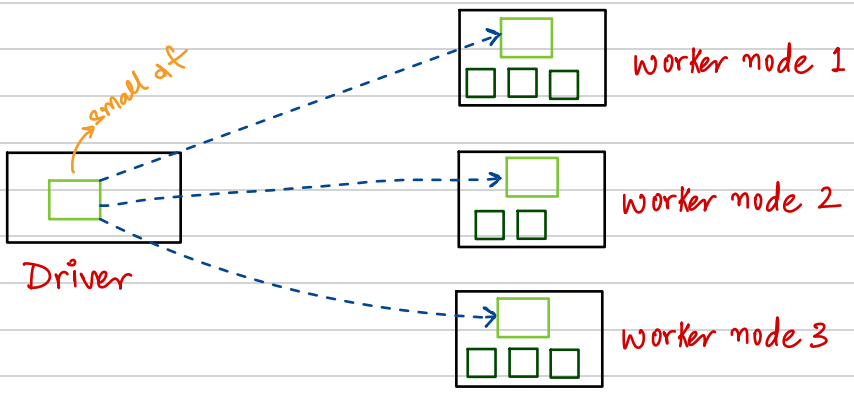
Big table

Dataframe 2 Customer

customerId	Customer Name
101	John
102	Emma
103	Michael

Small table

Step 2 : Broadcast small table
- The large df is already partitioned across nodes.
- Since every node now has a copy of small table, each node can do the join locally on its portion of big table.



Step 3: Build Hash Map

- On each executor, Spark builds an in-memory hash map from the broadcasted table.
- The hash map is built using the join key.
- Hashing and Joining happens same as in shuffle hash join.

worker node 1 101 : [(101, John)]

worker node 2 102 : [(102, Emma)]

worker node 3 103 : [(103, Michael)]

Step 4: Join locally

- Each executor now joins the large table's partitions with the broadcasted small table locally.
- for every row in large table, it looks up the corresponding value in the hash map.

worker node 1

customerId	Customer Name
101	John

orderId	customerId	amount
1	101	50
2	102	100

worker node 2

customerId	Customer Name
102	Emma

orderId	customerId	amount
3	101	75
4	103	60

worker node 3

customerId	Customer Name
103	Michael

orderId	customerId	amount
5	102	120

step 5: Combine Results

- Since all join processing happens locally (no data shuffle) the join results are fast.
- Spark merges results from all nodes to give the final output.

worker node 1

OrderId	customerId	amount	CustomerName
1	101	50	John
2	102	100	Emma

worker node 2

OrderId	customerId	amount	CustomerName
3	101	75	John
4	103	60	Michael

worker node 3

OrderId	customerId	amount	CustomerName
5	102	120	Emma

Results

OrderId	CustomerId	amount	customerName
1	101	50	John
2	102	100	Emma
3	101	75	John
4	103	60	Michael
5	102	120	Emma

Ans 3. Difference between shuffle hash join and Broadcast hash join.

Feature	Shuffle hash join	Broadcast hash join
Data Movement	Both tables are shuffled across the cluster.	Only the small table is broadcasted to all nodes
When Used	When no table is small enough to broadcast.	When one table is small. (usually <10 MB)
Performance	Slower due to network shuffle	faster due to local joins without shuffling
Memory Requirement	Needs enough memory to build hash table during join.	Needs memory to store broadcasted table on each executor.
Join Strategy	Spark moves data around and matches rows across all partitions.	Spark sends the small table to all nodes so each node can join locally.
Common use case	Joining two large datasets	Joining large + small dataset.
Manual Trigger	Can't be forced directly	Can be forced using broadcast() in PySpark.

Ans 4. We can change the broadcast threshold size in spark using the configuration setting :

```
spark.conf.set("spark.sql.autoBroadcastJoinThreshold", value_in_bytes)
```

example :

If we want to increase the threshold to 20 MB, then :

```
spark.conf.set("spark.sql.autoBroadcastJoinThreshold", 20 * 1024 * 1024)
```

By default, this value is 10 MB (i.e. 10485760 bytes). Increasing it allows larger tables to be considered for broadcast joins.

Note: Setting the value to -1 disables broadcast joins entirely.

```
spark.conf.set("spark.sql.autoBroadcastJoinThreshold", -1)
```

Ans. Possibilities of failure of Broadcast hash join :

1) Out of memory (OOM) Errors

- The most common failure.
- If the broadcasted table is too large to fit in memory on the executor, it can cause the job to crash.
- This can happen if :
  - → A table is manually broadcasted without checking its size
  - → Multiple joins are being broadcasted in a single job.

2) Network overhead

- Spark sends a copy of the small table to every executor.
- If you have a large number of nodes or the table is just under the threshold but still big, it can lead to :
  - → High network I/O
  - → Increased job latency.

3) Skewed Data in Large Table

- Even though the small table is broadcasted, if the large table has skewed keys, some partitions will still take longer to process.
- This can cause uneven workload, slowing down the job or causing task timeouts.

4) Multiple Broadcasts in one job

- Spark may try to broadcast multiple small tables.
- If the cumulative memory usage exceeds available executor memory, the join can fail.

5) Improper Configuration

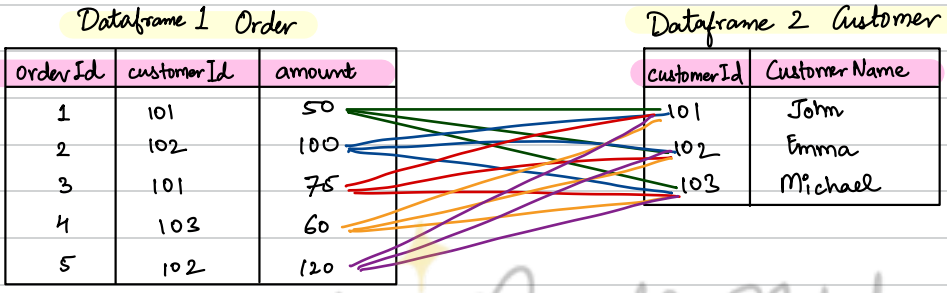
- If the spark.sql.autoBroadcastJoinThreshold is set too high, Spark might broadcast a table that's technically under the limit but still too big for executor memory → crash.

4) Cartesian Join

- A cartesian join returns the cross product of two datasets, it joins every row of the first dataset with every row of the second.

example:
- df1 has 3 rows
- df2 has 4 rows

Then a cartesian join will produce 3x4 = 12 rows



- Highly expensive: for large datasets, this can generate billions of rows.
- Heavy on memory and computation.
- Can cause Spark jobs to crash or run extremely slow if not handeled properly.
- Use only when we intentionally want all combinations.

5) Broadcast nested loop join (BNLJ)

- Broadcast nested loop join is a fallback join strategy used by spark when:

→ There's no join condition (non-equi join), OR
→ Join condition is too complex for hash-based joins (e.g. involving non-equality comparisons like <, !=, etc.)

- Steps to perform BNLJ
  1. Broadcast the small table.
  2. Iterate through larger table.
  3. Perform nested loop
     It checks join condition for every pair.
  4. Collect matching pairs.
- Very expensive

Spark Memory Management

Driver out of memory

Potential interview questions:

1) What is OOM in spark?
2) Why do we get driver OOM?
3) What is driver overhead memory?
4) Common reasons to get a driver OOM?
5) How to handle OOM?

Ans 1. An out of Memory (OOM) error in spark happens when a task, executor, or driver runs out of available memory to process data, causing the job to fail.

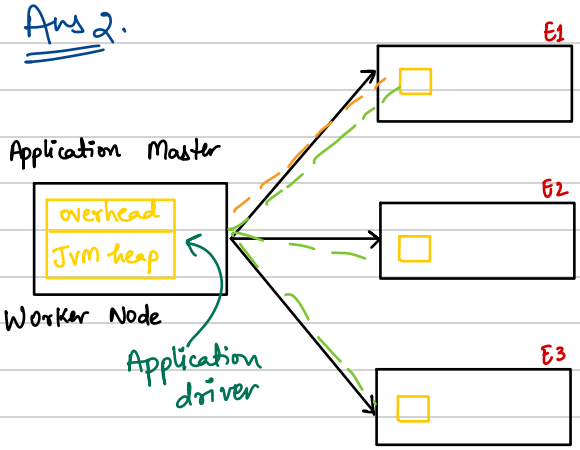
This usually means spark couldn't allocate enough memory to:

- Store intermediate data,
- Cache datasets,
- Shuffle data across the cluster, or
- Run user code inside transformations.

! Error messages you might see:

- java.lang.OutOfMemoryError: Java heap space
- GC overload limit exceeded
- ExecutorLostFailure

Ans 2.



- collect(): Entire data gets collected in the driver.
- show(): → displays 20 records by default.
  → we can get any no. of records we want by giving it the value.
  → Entire partition moves to the driver.

NOTE: Sometimes show() doesn't throw error but collect does for the same data because, show() gets the partition & collect() gets the entire data.

Application driver memory:
spark.driver.memory (JVM heap)
spark.driver.memoryOverhead (overhead)

So, we get driver OOM when the spark driver tries to handle more data or metadata it can hold in memory.

Ans 3.

Application driver memory :

spark.driver.memory (JVM heap) JVM process (1GB)

spark.driver.memoryOverhead (overhead) Non JVM process (10% of driver memory {min. 384 MB required}

example:

1 GB → 100 MB but takes 384 MB

4 GB → 400 MB , 400 MB

20 GB → 2GB , 2 GB

- Driver memory = for running your code & spark logic
- Driver overhead memory = → for JVM internals and operations
  → Important for stability and avoiding unexpected driver crashes.

So, Driver overhead memory is not part of the heap memory used to store Dataframes or execute jobs, it's like buffer zone for everything else that runs around the core logic.

Ans 4. Common reasons to get a driver OOM

1) Calling .collect() or toPandas() on large Dataframes
   - These actions pull all data to the driver.
   - If the dataset is too large, it overflows the driver's heap memory.

2) Improper memory allocation :
   - Default spark.driver.memory (like 1-2 GB) might be too low for large jobs.
   - Not enough driver memory = early OOM errors

3) Not enough Overhead memory :
   - Spark needs extra memory for JVM tasks
   - If overhead memory is too low, the driver crashes even when driver memory is fine.

4) Large Broadcast Variables:

- Broadcasting a big object that doesn't fit into the driver memory.

5) Too many shuffles or Metadata Accumulation

- When spark stages generate alot of metadata, the driver has to manage it all.
- In long-running jobs, this metadata can fill up memory.

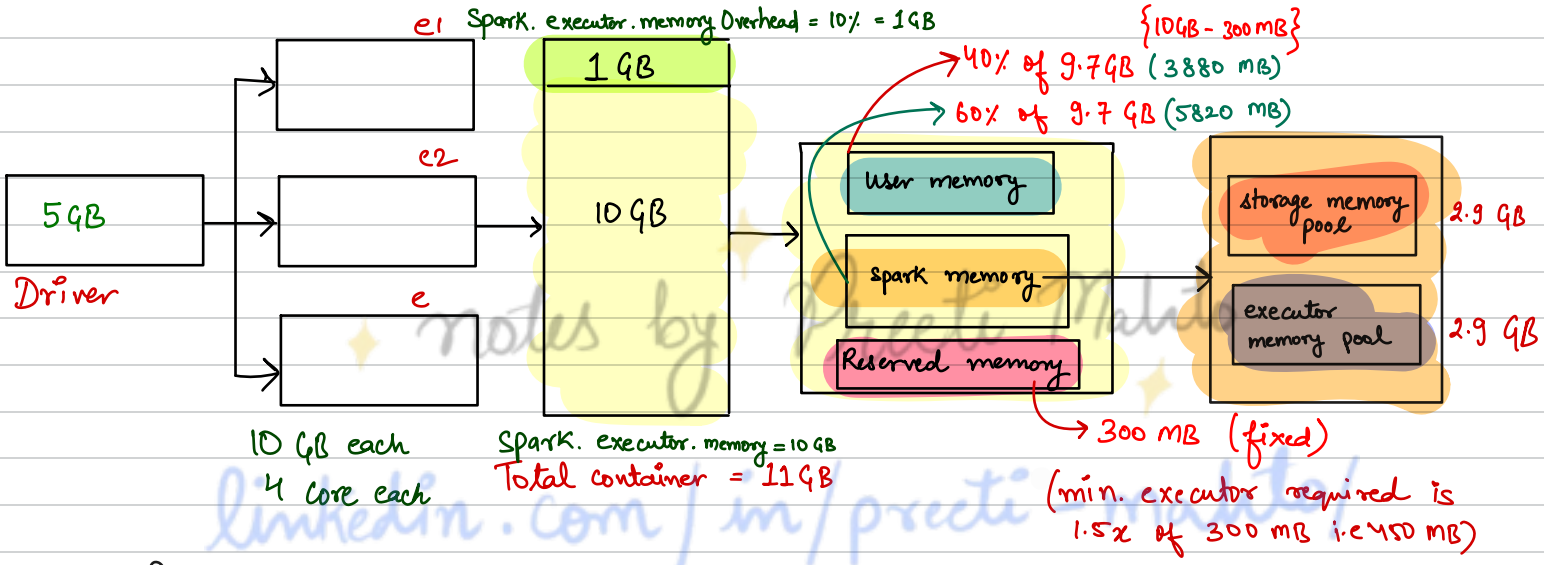
Ans: How to fix driver OOM errors:

1) Avoid .collect() unless you're sure the data is small. Use .show(), .limit(), or write data to storage.
2) Process data using spark transformations instead of pure python.
3) Increase driver memory, if needed
4) Use broadcast only on small data, or avoid if unecessary.
5) Break jobs into smaller steps.
6) Increase overhead memory, if needed

Executor out of memory

Potential interview questions :

1) When do we get executor OOM ?
2) How spark manages storage memory inside executor internally ?
3) How task is splitted in executor ?
4) Why do we need overhead memory?
5) Why do we get OOM when data can be spilled to the disk ?
6) Types of memory manager in spark ?



→ Spark. executor. memory. Overhead : {used for non JVM processes.
300-400 MB → used by container
600-700 MB → pyspark application

→ Reserved memory : used to store spark internal objects.
used by spark engine.

→ User Memory : → used to store user defined data structure, spark internal metadata and any udf created.
→ used by RDD operation
→ example: Aggregation → map partition transformation.

→ Spark memory : → used for storing intermediate state of tasks like joining.
→ used to store cached data.
→ Memory eviction is done in LRU fashion.

→ Executor memory usage : → used for storing object that is required during execution of spark tasks.
→ stores hash table for hash aggregation.
→ short lived, cleared after each operation.
→ spilling to disk.

Ans 1. An Executor OOM happens when a spark executor process uses more memory than it's allowed.

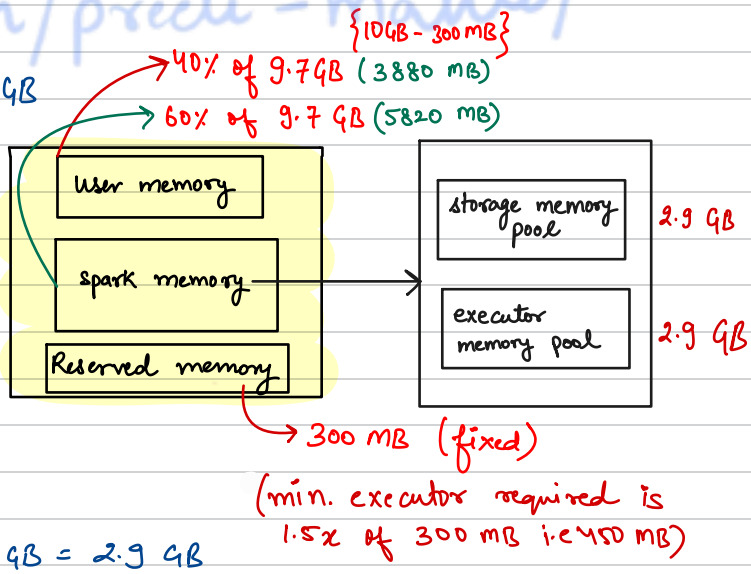
Situations when we get executor OOM errors:

1) large shuffles
2) Skewed data
3) Insufficient executor memory
4) Uncontrolled caching
5) Too many Tasks on a single executor
6) Large Broadcast variables: While broadcasting is helpful, broadcasting something too big will get loaded into executor memory and can crash it.
7) Memory misconfiguration

Ans 2.

Executor size = 10 GB
Executor overhead memory = 10% of 10GB = 1GB
Total container size = 11GB

Inside executor memory (10GB):
User memory = 40% of 9.7 GB (3880 MB)
spark memory = 60% of 9.7 GB (5820 MB)
Reserved memory = 300 MB (fixed)



Storage memory = 50% of 5.8 GB = 2.9 GB
Reserved memory = 50% of 5.8 GB = 2.9 GB

Executor memory

Ans 3. If the executor has 4 cores it can run 4 tasks in parallel.

Ans 4.

→ Spark.executor.memory.Overhead: {used for non JVM processes.
300-400 MB → used by container
600-700 MB → pyspark application

Ans 5.

Large of skewed partitions : If one partition is too large or skewed (due to uneven data distribution) a single task may try to process more data than fits into memory.

→ Spark cannot spill just part of a record, the entire record must be held before it can be spilled

Ans 6. 1) Static Memory Manager :

- Default before spark 1.6
- statically divided b/w execution & storage
- fixed size memory regions were allocated

Cons :

- If execution memory was idle but storage needed more space, spark couldn't share memory between them.
- often led to inefficient memory usage and OOMs

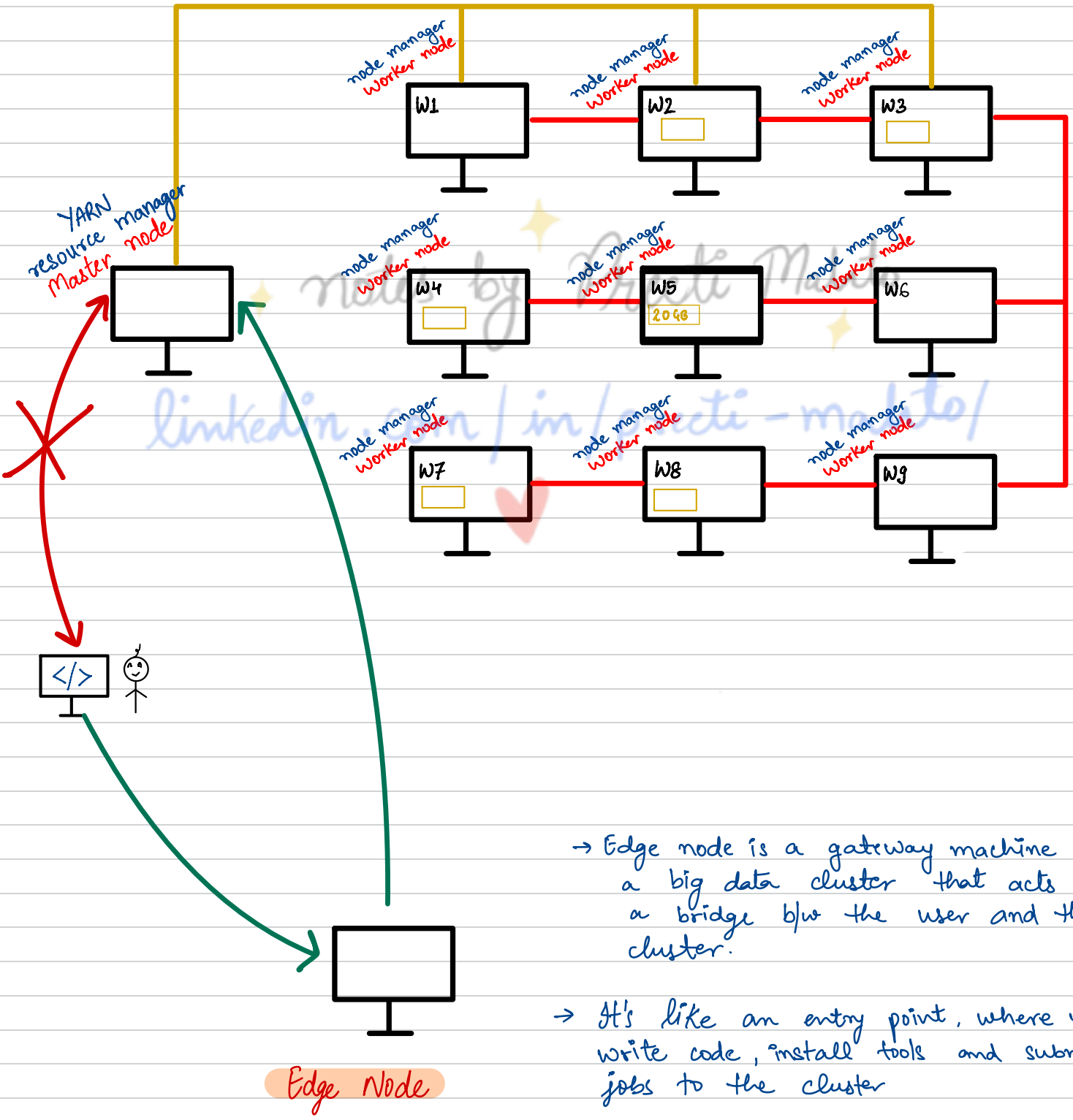
2) Unified Memory Manager :

- Introduced in spark 1.6, and is the default memory manager now.
- Unified memory pool shared dynamically b/w :
  - → Execution memory (50% of spark memory)
  - → Storage memory (50% of spark memory)

Deployment mode in Spark

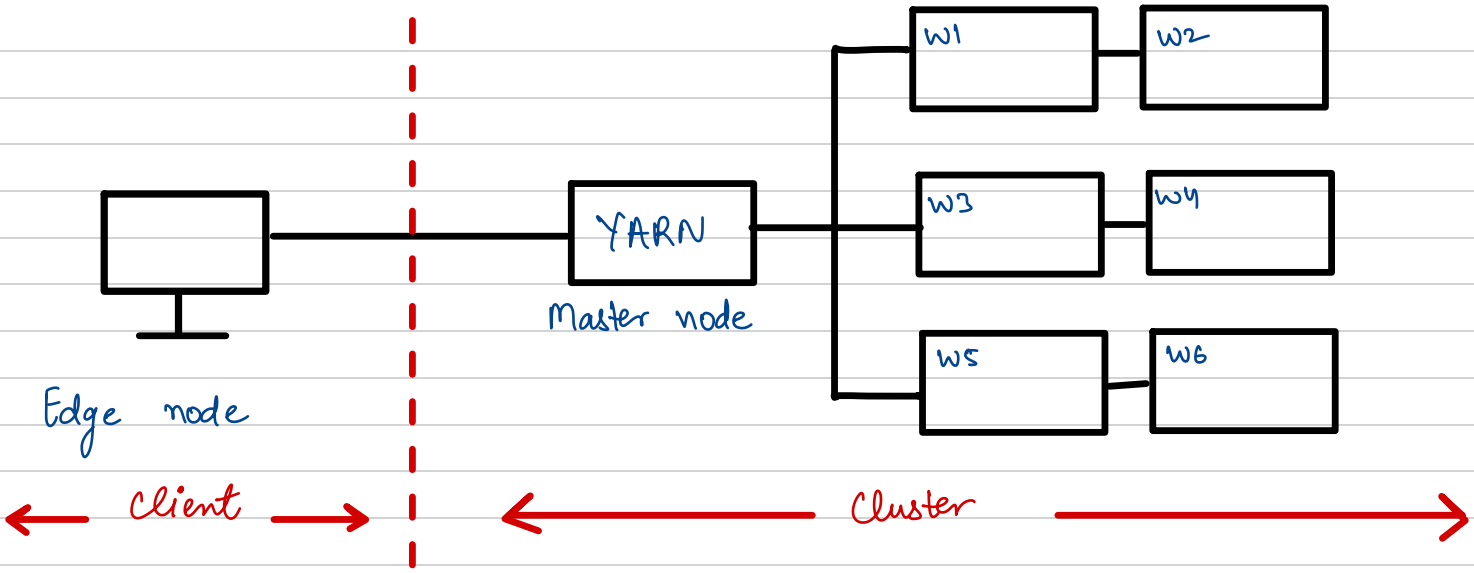
Potential interview questions :

1) What all deployment modes are there in spark ?
2) What is edge node ?
3) Why do we need client and cluster modes ?
4) What will happen if edge Node is closed ?



→ Edge node is a gateway machine in a big data cluster that acts as a bridge b/w the user and the cluster.

→ It's like an entry point, where you write code, install tools and submit jobs to the cluster



Feature	Client mode	Cluster mode
Where driver runs	On the machine that submits the application (e.g edge node)	On one of the cluster's worker nodes.
Latency & Reliability	less reliable, if the client goes down, the job fails.	More reliable, job continues even if the submitting machine shuts down.
Use case	Good for development, debugging, small jobs.	Ideal for production, long running jobs.
Network dependency	Needs constant connection b/w client and cluster	No need to stay connected after job submission.
Example command	--deploy-mode client	--deploy-mode cluster
How to set?	spark-submit \ --master yarn \ --deploy-mode client \	spark-submit \ --master yarn \ --deploy-mode cluster \

AQE (Adaptive Query Execution)

AQE is a spark optimization technique introduced in Spark 3.0 that allows Spark to dynamically optimize and adjust query plans at runtime, based on actual data statistics.

Before AQE, Spark used a static query plan, it guessed the best execution strategy before the job started.

With AQE, Spark can make smarter decisions after seeing real data during execution.

What's the need of AQE?

Because data can be skewed, unpredictable, or different than expected. AQE helps handle such scenarios in real time to improve performance and avoid failures.

Features of AQE

1) Dynamically coalescing shuffle partition.
2) Dynamically switching join strategy
3) Dynamically optimizing skew join.

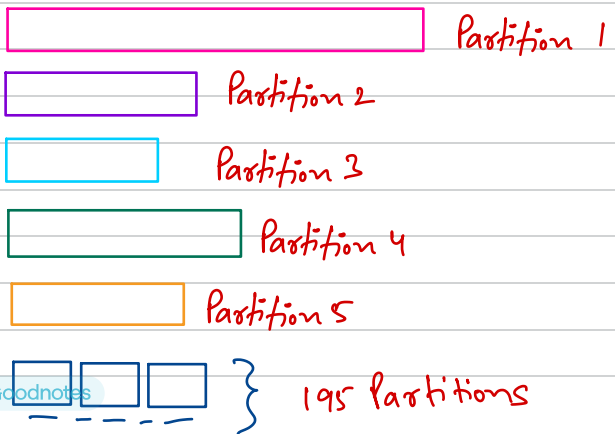
Dynamically coalescing shuffle partition.

Imagine we have 250 MB of skewed data, so these will be 2 partitions

sales data, Sugar is best selling product.

We run groupBy, so data gets shuffled.

After shuffling:



Now, Partition 1 will take a lot of time get processed.

What AQE does with coalescing now is:

Now, there's only 3 tasks which will take almost same time to get processed.

Also we freed 2 CPU cores.

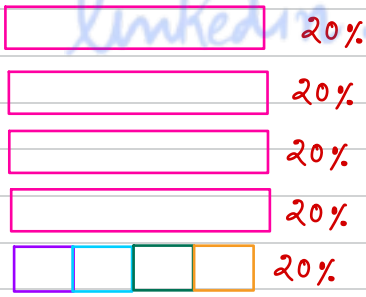
Sometimes, the data is too skewed, even coalescing doesn't work

80% of data.

20% of data.

In this situation we'll split the data into equal partitions but there's a condition:

skewed data should be 5 times of the median
& size of skewed data should be > 256 MB.



	Without AQE	with AQE
Shuffle Partitions	200	3
Tasks in Next stage	200	3
Job Execution Time	~15 seconds	~5 seconds

Dynamically switching join strategy.

Initially,

Table 1	Table 2
10 GB	20GB

After transformations,

Table 1	Table 2
8GB	5MB

Without AQE,

Spark doesn't know Table 2 now is only 5MB {<10MB}
It uses default joins (e.g. sort merge join)

That involves shuffling of Table 1, which is expensive

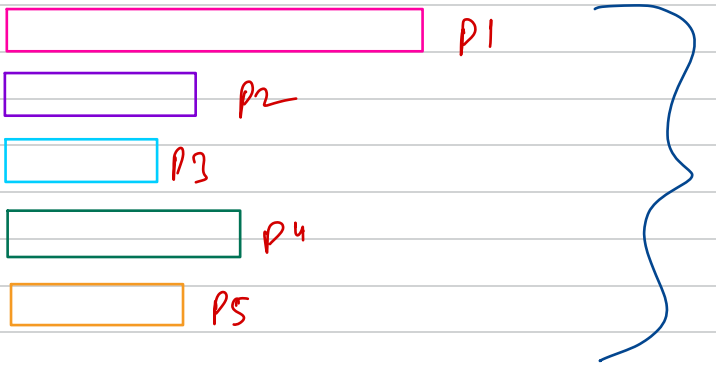
With AQE

Now, Spark will dynamically change the join type at runtime to the most efficient one based on actual data size.
In this case, broadcast join.

So, Only the small table is sent to all nodes, no shuffle on Table 1.

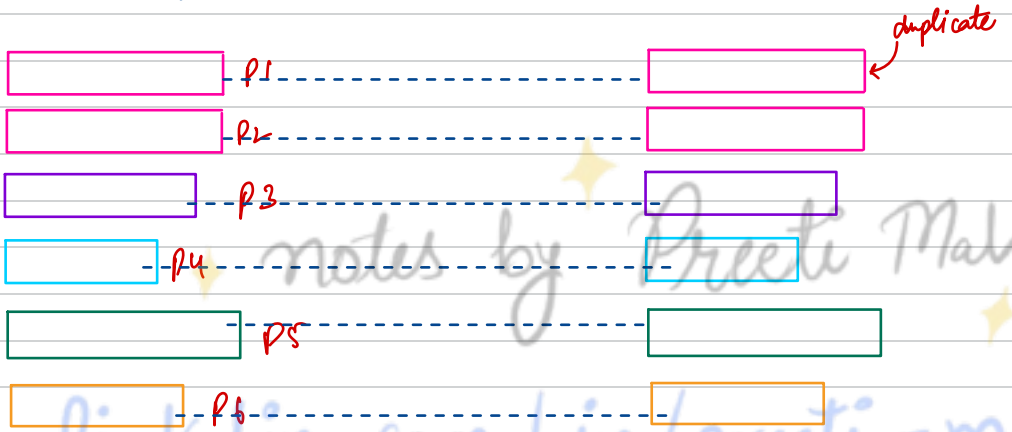
	Without AQE	With AQE
Shuffle	Both tables	none (only broadcast)
Execution Time	High	low
Memory usage	high	low
CPU usage	high	low

Dynamically optimizing skew join



P1 takes a lot of time to get processed. while P2, P3, P4, P5 gets processed and sit idle.

Now P1 gets splitted into smaller partitions.



At runtime, spark does the following:

1. Detects skew : It sees that partition 1 is larger than average.
2. Splits that partition: Instead of one task for sugar, it creates multiple smaller tasks.
3. Joins them independently: Each split chunk of the skewed data is joined with Table 2.
4. No task is overloaded. All finishes around the same time.

Cache and persist

Potential interview questions :

1) What is caching ?
2) Why do we need caching or persistance ?
3) When should we avoid caching ?
4) How to uncache the data ?
5) Difference between cache and persist ?
6) What are different storage levels in spark ?
7) Which storage level to choose ?

In spark, cache is a method used to store the results of a dataframe or RDD in memory for reuse across multiple actions. When you cache something, spark keeps it in memory so it doesn't have to compute it again.

How does it work ?

→ When you call .cache() on a Dataframe or RDD, spark marks it to be cached.

→ However it doesn't cache it immediately, caching happens only when an action (like .count(), .collect(), .show()) is trigerred.

→ Spark will then :

1) compute the result.
2) Store it in memory for the first time.
3) for subsequent actions, it uses the cached version, avoiding recomputation.

What storage level does it use ?

By default, .cache() is just a shortcut for :

```
.persist(StorageLevel.MEMORY_AND_DISK)
```

This means,

→ spark will try to store the data in RAM.
→ If the data doesn't fit, the remaining part is spilled to disk.
→ This helps prevent out-of-memory errors.

When should you use cache() ?

Use .cache() when () :

→ reusing the same Dataframe or RDD multiple times.
→ faster performance by avoiding recomputation.

What happens if cache is not used ?

→ Every time when an action is performed on a dataframe, Spark recomputes it from scratch.

→ This is slow for large datasets and repetitive queries.

What is persist() ?

In Spark, persist() is a method that lets you store intermediate results of a Dataframe or RDD across operations.

Unlike cache(), which uses only one storage level, persist gives you multiple options for how and where to store the data.

Why use persist() ?

By default, Spark is lazy, meaning it doesn't compute anything until an action is called.

Without persistence, spark recalculates your transformations every time you run a new action.

When you persist, Spark stores the result of an expensive computation so it doesn't have to re-run it.

Storage levels in spark()

1) MEMORY_ONLY : Stores RDD in RAM. Throws error if not enough memory.

2) MEMORY_AND_DISK : Tries memory first, then spills remainder to disk. (default for .cache())

3) DISK_ONLY : Skips RAM and stores directly to disk

4) MEMORY_ONLY_SER : Stores a serialized version in memory (more space-efficient, but CPU heavy).

5) MEMORY_AND_DISK_SER : Serialized version, spills to disk if needed.

6) OFF_HEAP : Uses off-heap memory. Rarely used manually.

How to choose storage level ?

If ...	Use ..
Data fits in memory	MEMORY_ONLY
Data doesn't fit in memory	MEMORY_AND_DISK
Memory is tight, okay with extra CPU	MEMORY_ONLY_SER
Memory tight + need fallback	MEMORY_AND_DISK_SER
Disk is okay, memory isn't available	DISK_ONLY

When we shouldn't use cache() and persist() ?

1) Small datasets: If the dataset is small and recomputation is cheap, caching adds unnecessary memory overhead.

2) Used only once : Don't cache if the Dataframe/RDD is used only once in your workflow.

3) Memory Constraints : If cluster doesn't have enough memory, caching can cause OOM errors or force frequent garbage collection.

4) Heavy Transformations before cache() : If transformations are computationally expensive and the result isn't reused, better to skip caching

How to uncache the data ?

df.unpersist()

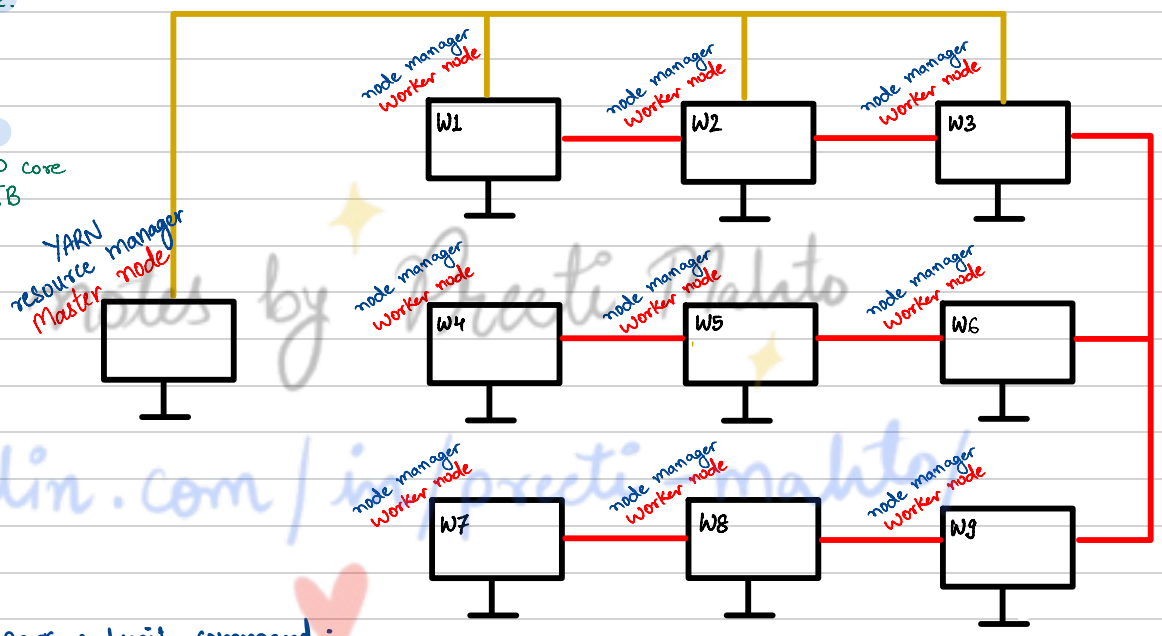
Dynamic Resource Allocation

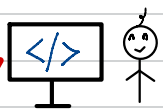
Potential interview questions :

1) What is dynamic resource allocation in spark ?
2) How resource manager provide the resources for dynamic resource allocation ?
3) What are the resource allocation techniques in spark ?
4) What are the challenges involved with dynamic resource allocation?
5) When to avoid dynamic resource allocation ?
6) How spark will remove or add resources ?
7) Configuration needed for dynamic resource allocation ?

Configuration of each machine:
CPU → 20 core
RAM → 100 GB

Configuration of the cluster :
CPU → 20 x 10 = 200 core
RAM → 100 x 10 = ~1TB





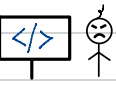
Developer1 submits spark-submit command :

```
spark-submit \
  --name YourAppName \
  --master yarn \
  --deploy-mode cluster \
  --conf spark.dynamicAllocation.enabled=true \
  --conf spark.dynamicAllocation.minExecutors=5 \
  --conf spark.dynamicAllocation.maxExecutors=49 \
  --conf spark.shuffleTracking.enabled=true \
  --conf spark.dynamicAllocation.executorIdleTimeout=45s \
  --conf spark.scheduler.backlogTimeout=2s \
  --conf spark.executor.memory=20g \
  --conf spark.executor.cores=4 \
  --conf spark.driver.memory=20g \
  --py-files python_dependencies.zip \
  main.py
```

Executors need :
49 x 20 = 980 GB
49 x 4 = 196 core

Driver need :
20 GB

→ When this spark-submit will run, it will fill up memory because the configurations are same.



→ developer2 comes, and asks for 20 GB executor & 5GB driver
→ But the cluster is fully occupied by developer 1.

To tackle this problem we need to understand Resource Allocation.

1) Static Resource Allocation : Static Resource Allocation in Spark means you define and fix the number of executors and their resources before job starts, and they remain constant throughout the job's execution. There's no automatic scaling based on workload, unlike DRA.

Limitations :

→ can waste resources if too many executors are allocated for a small workload.

→ May lead to oom errors or slow performance if too few resources are allocated.

→ No flexibility, can't respond to changes in workload or data volume.

2) Dynamic Resource Allocation : Dynamic resource allocation (DRA) in spark is a feature that allows your Spark application to scale the number of executors up and down automatically, based on workload, instead of setting a fixed number of executors manually.

Why DRA should be used?

→ Avoid Idle resources : Executors are removed when not in use.
→ Scale when needed : More executors are added if the workload increases.
→ Save cost and improve cluster efficiency.

How it works ?

1) Job starts
Spark starts with a minimum no. of executors (default 0 or 1)

2) Monitoring task backlog
Spark keeps checking if there are pending tasks or stages not yet running

3) Add executors
If there are backlogged tasks, Spark requests more executors (upto a max. limit) every 1 sec in two folds. (i.e asks for 1 ex, then 2 ex, then 4 ex, then 8 ex .....

4) Executor idle timeout
If an executor stays idle for a certain time (default 60 secs), spark removes it.

5) This cycle continues as the job runs, scaling up & down dynamically.

Example :

User wanted 1000 GB.

After filters, user needs 500 GB only.

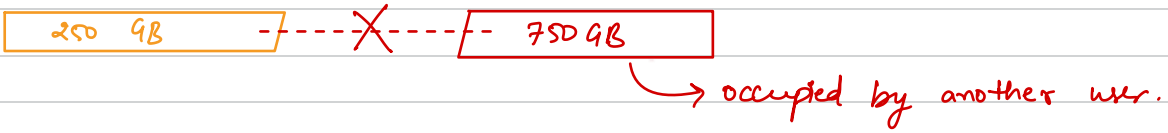
Again after filters, user needs 250 GB only.

Now, user needs 750 GB, so it will demand 500 GB.

Case 1: If 500 GB is available.

So,

Case 2. If Memory is taken by another user.



Process might fail in this case.

Ques. What happens to the data which is needed later is stored in an executor which is going to be released while DRA ?

External Shuffle service handles shuffle data, if spark.shuffle.service.enabled is set to 'true' (which is required for DRA), then shuffle data is stored outside the executor, in the external shuffle service.

So, even if the executor is removed, the shuffle files are still accessible to other running executors.

Ques. When should we avoid DRA?

→ In Production, when process is very critical

→ External shuffle service not enabled: Shuffle data may be lost if executors are removed.

Dynamic partition pruning

In large datasets stored in partitioned tables, Dynamic partition pruning is a runtime optimization in Spark that reduces the amount of data scanned when performing joins involving partitioned columns.

Instead of scanning all partitions of a table, Spark prunes unnecessary partitions at runtime, based on the join keys from the other table

Sale_id	product_id	amount	country	date
1	101	500	India	2023-01-01
2	102	800	USA	2023-01-02
3	103	200	Canada	2023-01-03
4	101	1000	Germany	2023-01-04
5	104	300	India	2023-01-05

fact table : Sales

Country	region
India	Asia
China	Asia
Germany	Europe
USA	North America
Canada	North America

Dimension Table : country_dim

If we write a query like :

```
SELECT f.*
FROM sales f
JOIN country_dim d
 ON f.country = d.country
WHERE d.region = 'Asia'
```

Without DPP, Spark doesn't know which partitions of sales to scan at compile time, so it scans all of them.

Spark scans all country partitions in sales even though only India & China are relevant. ∴ Huge waste of compute & time)

With DPP, Spark waits until it has read country_dim (at runtime), collect the relevant partition values, and prunes only those partitions in sales.

Spark first reads country_dim and extracts India and China.
Then only Sales partitions for India and China are scanned.

Salting in Spark

Potential Interview questions:

1) What is data skewness problem?
2) What are the ways to remove skewness?
3) What is Salting?
4) How can we implement salting?

TABLE 1

prod_id	sales
1	200
1	100
1	20
1	50
1	60
2	10
2	5
3	8

TABLE 2

id	date
1	12-05-23
1	13-08-23
2	17-02-23
2	11-01-22
3	30-08-22

>10 MB
Can't broadcast

Joining these tables on prod_id = id will cause skew for Key = 1.

Step 1: Apply salt (Add random suffixes to skewed key)

Modify Table 1, add a salt column for prod_id 1

We'll apply three salt values 0, 1, 2 (randomly distributed)

TABLE 1

prod_id	sales	salt	prod_id_salted
1	200	0	1_0
1	100	1	1_1
1	20	2	1_2
1	50	0	1_0
1	60	1	1_1
2	10	0	1_0
2	5	1	1_1
3	8	0	1_0

Step 2: Expand Table 2 for all salt values (duplicate keys for salted join)

TABLE 2

id	date	salt	id_salted
1	12-05-23	0	1_0
1	12-05-23	1	1_1
1	12-05-23	2	1_2
1	13-08-23	0	1_0
1	13-08-23	1	1_1
1	13-08-23	2	1_2
2	17-02-23	0	2_0
2	17-02-23	1	2_1
2	11-01-22	0	2_0
2	11-01-22	1	2_1
3	30-08-22	0	3_0

Step 3: Join on prod_id_salted = id_salted

This ensures even partitioning preventing all Key=1 records from going into the same partition.