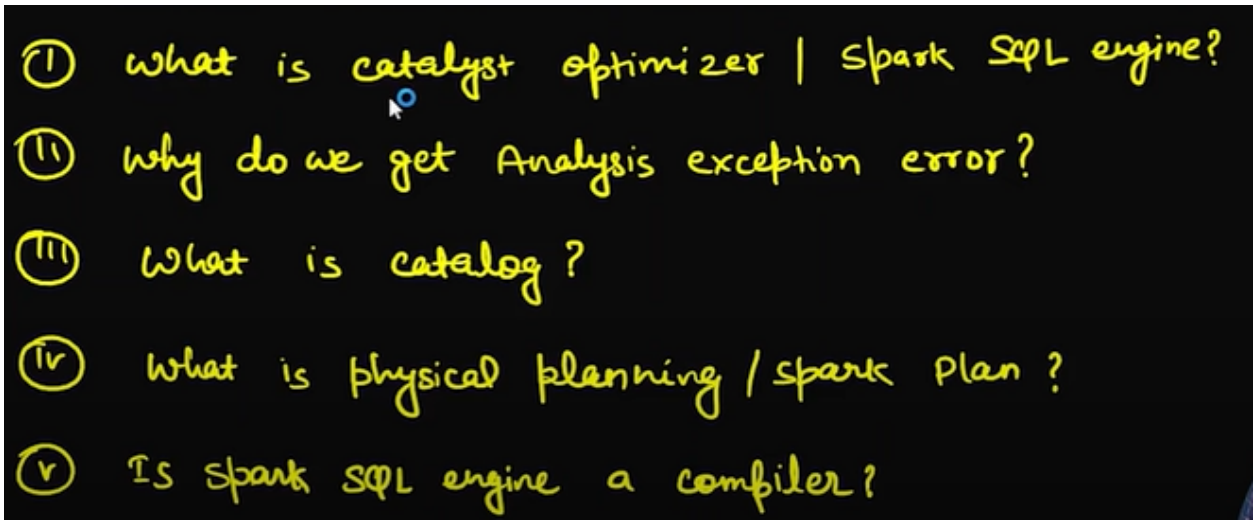
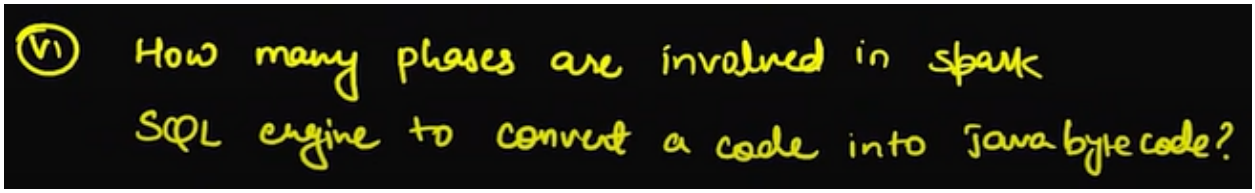
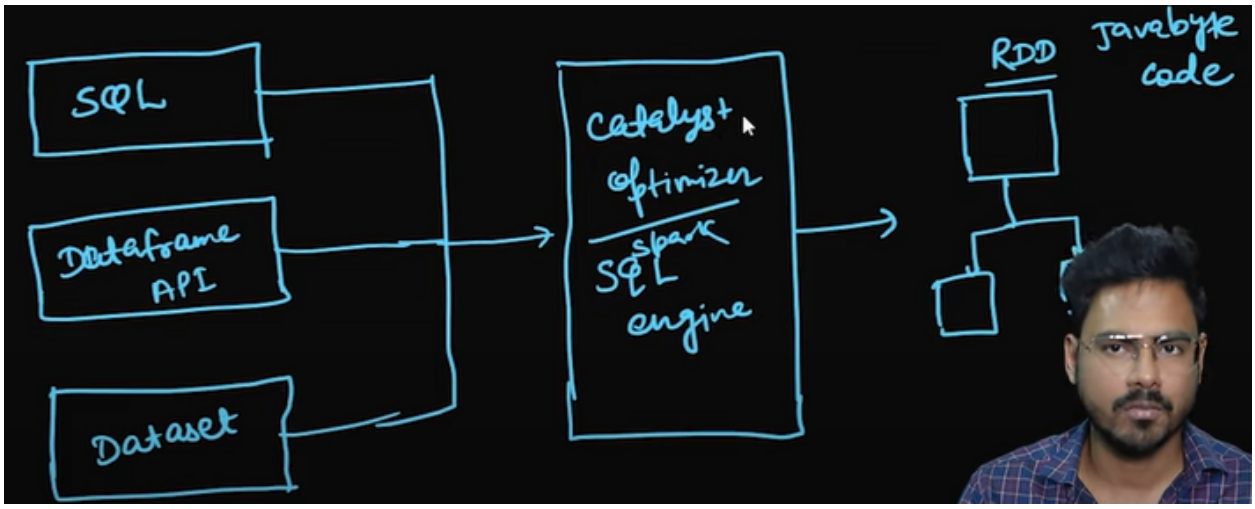
Spark SQL Engine

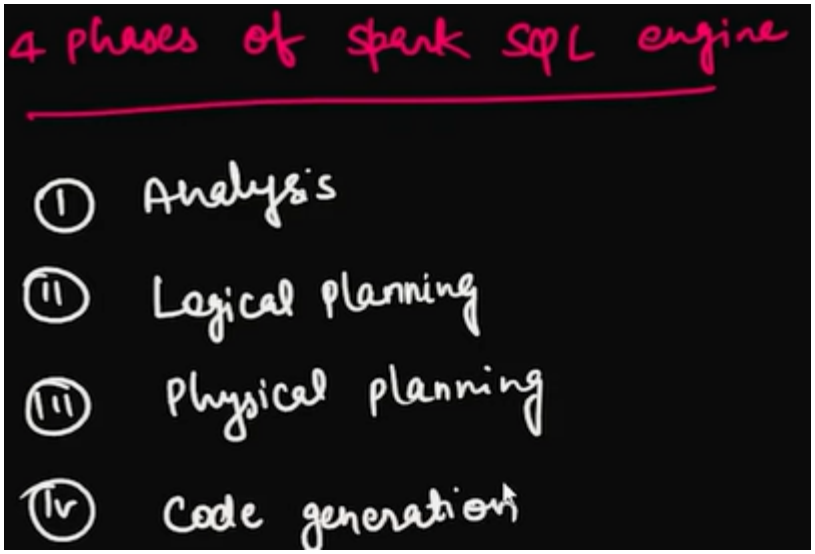
1. Now we'll learn how our code converts into java byte code
2. Potential interview questions



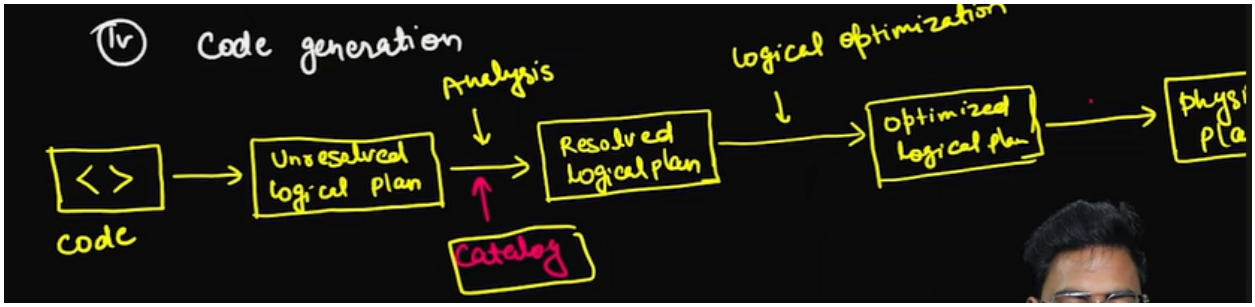


3. Lets look at high level picture of spark sql engine
4. 

5. Here lets suppose we wrote code in 3 diff types…now every type of code..must go into catalyst optimizer(SQL engine) ..it gives the output in RDD's(java byte code)
6. Now we will learn Spark SQL Engine

7. It has 4 phases 

8. Here we have the flow of spark SQL engine



9. First out code will move to unresolved logical plan(it will create a logical plan for our transformations)
10. Now we'll move to analysis stage..and it has a catolog
11. Catalog is nothing but the metadata of our data

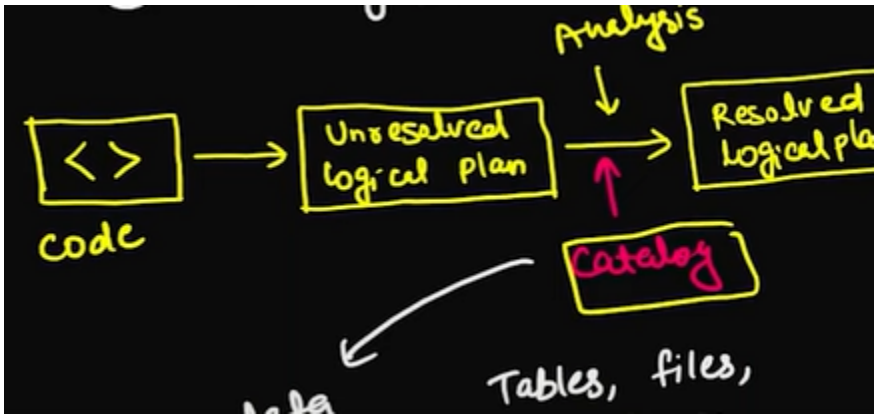
12. If the file or table or column is not present in our catalog..then it will give analysis error

13. Practical example

```
df=spark.read.format("json")\
        .option("inferSchema","true")\
        .option("mode","PERMISSIVE")\
        .option("multiline","true")\
        .load("/FileStore/tables/multiline_correct.json")
df.select("name1").show()
```

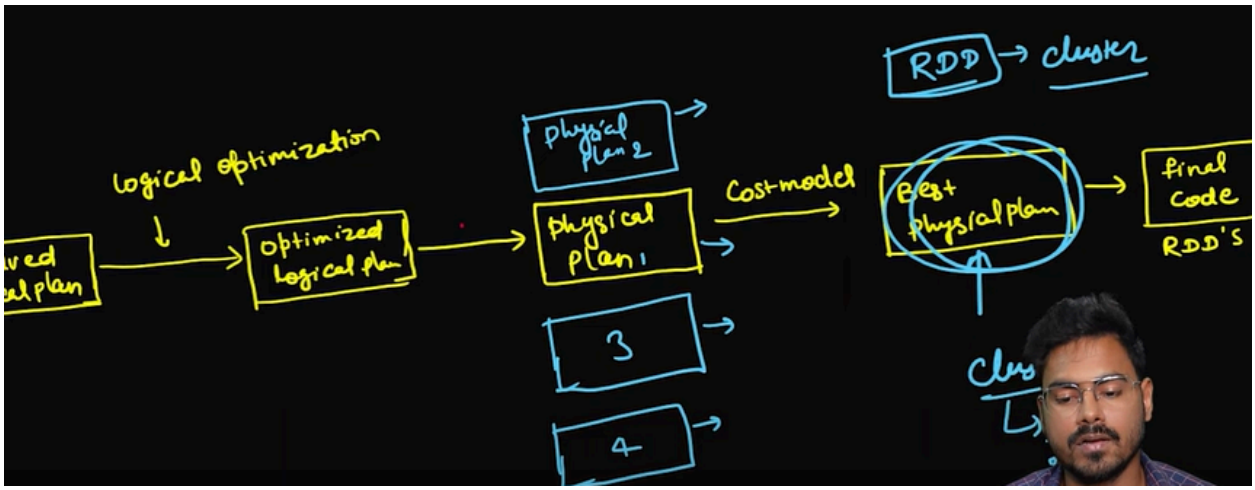
```
▸ (1) Spark Jobs
AnalysisException: [UNRESOLVED_COLUMN.WITH_SUGGESTION] A column or function parameter with na
be resolved. Did you mean one of the following? [`name`, `age`, `salary`];
'Project ['name1]
+- Relation [age#28L,name#29,salary#30L] json
Command took 24.48 seconds -- by manisnitt@gmail.com at 4/21/2023, 9:54:07 AM on My Cluster
```

14. Here we have asked dataframe to show “name1” col..which is not present in our df…and it gave an analysis exception error.
15. See here..it went to unresolved logical plan and created a logical plan for transformations too
16. But as the col “name1” was not present in our catalog..it gave an error
17. Now instead of “name1” if we have “name” then it goes to resolved logical plan



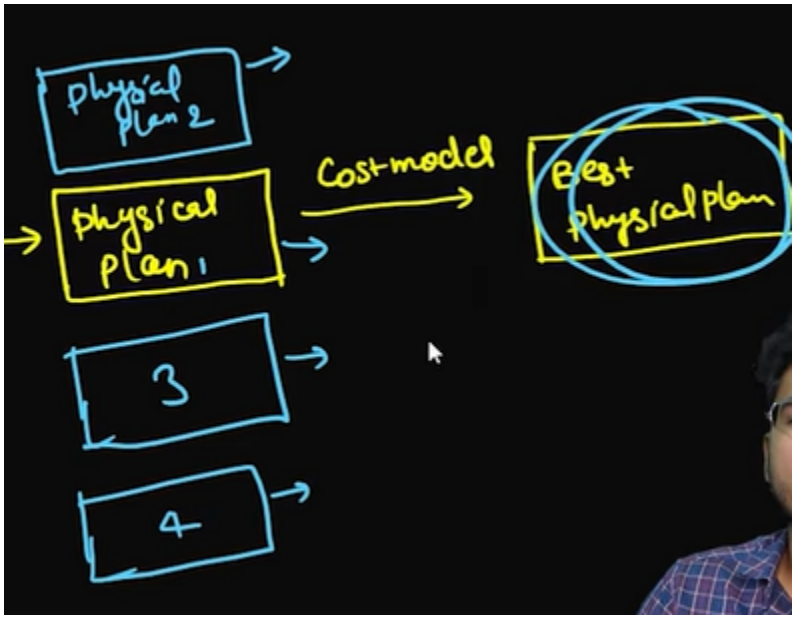
18. 2. **Logical Optimization:**
    - **What it Does:** Here, Spark rewrites your query to create a logical plan for processing the data. This plan represents the operations to be performed on the data, independent of the underlying storage or execution engine.
    - **Example:** Continuing with the previous query, Spark might rewrite it to optimize filtering and aggregation steps. It could potentially push down filtering logic closer to the data source for efficiency.

19. Suppose if you only need 2 col's…but you have used "*" to get all the columns…then our logic optimizer will optimize the query and give 2 col's



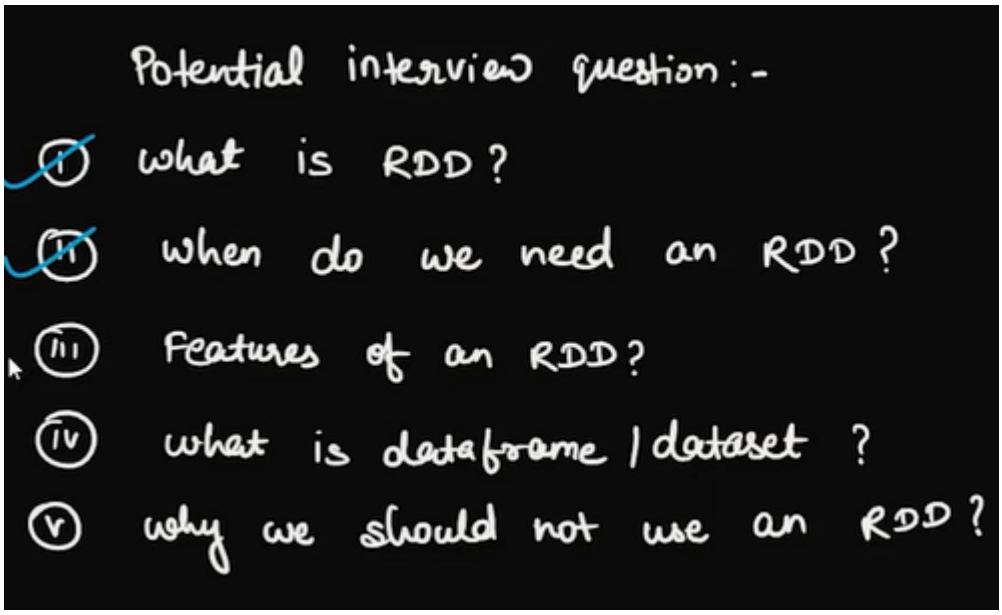
20. Physical plan

3. **Physical Planning:**
   - **What it Does:** This phase translates the logical plan into a physical plan. It considers factors like data partitioning, shuffle operations (data movement across machines), and available execution engines (like MapReduce or specialized engines for certain data formats). Spark might generate multiple physical plans and choose the one with the estimated lowest execution cost.
   - **Example:** Based on your data and cluster configuration, Spark might decide to use a MapReduce execution engine for shuffles or a specialized engine for processing data stored in Parquet format.

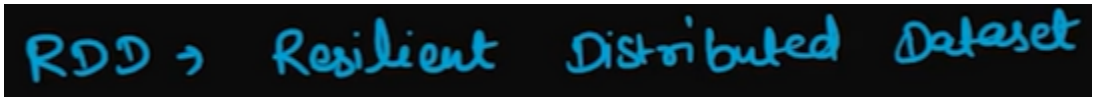


RDD

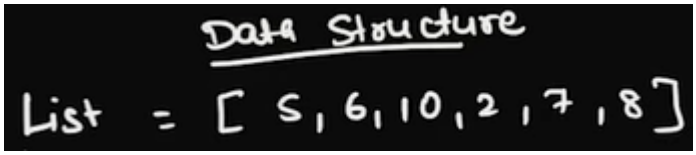
1. Questions



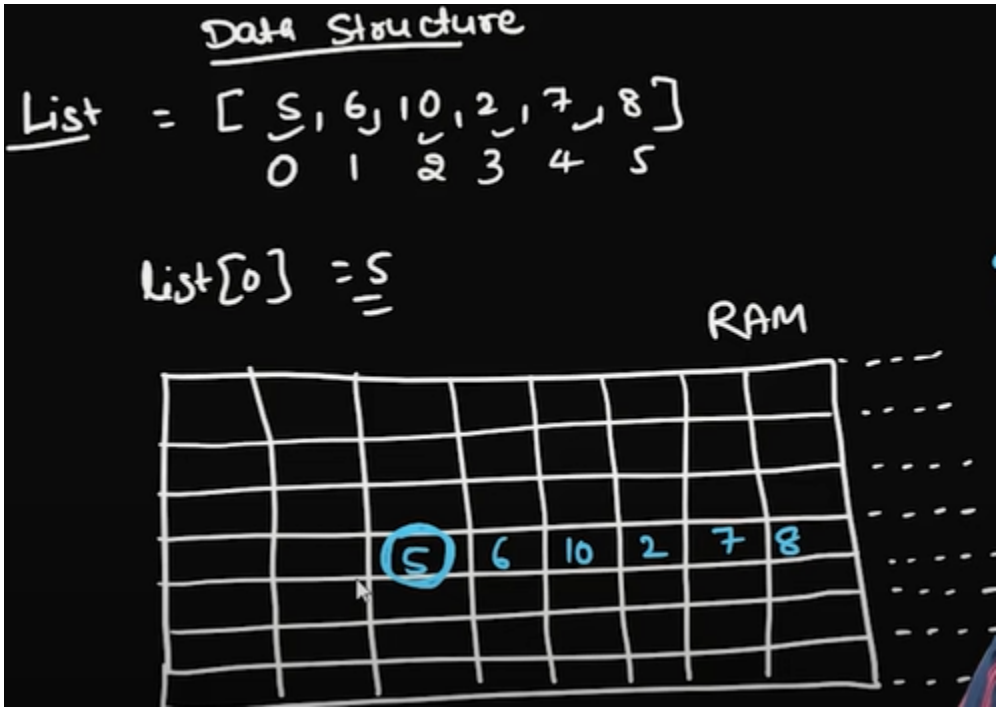
2. RDD stand for



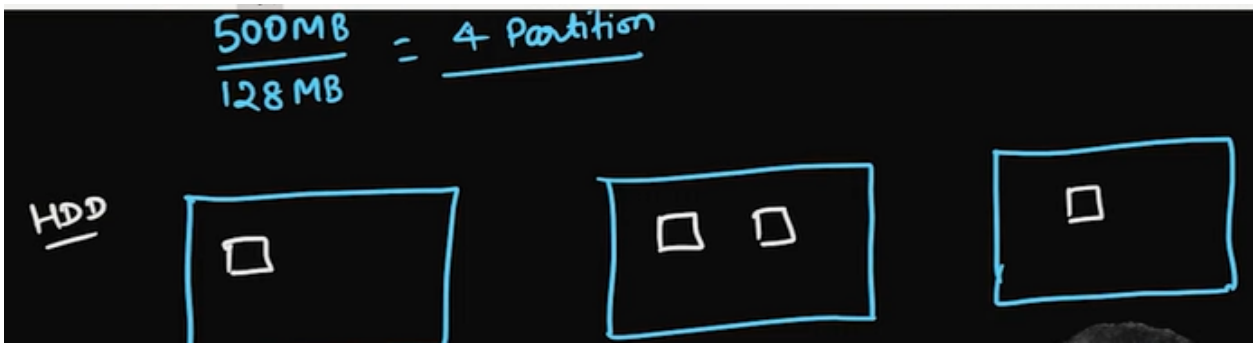
3. Let's consider a list data structure



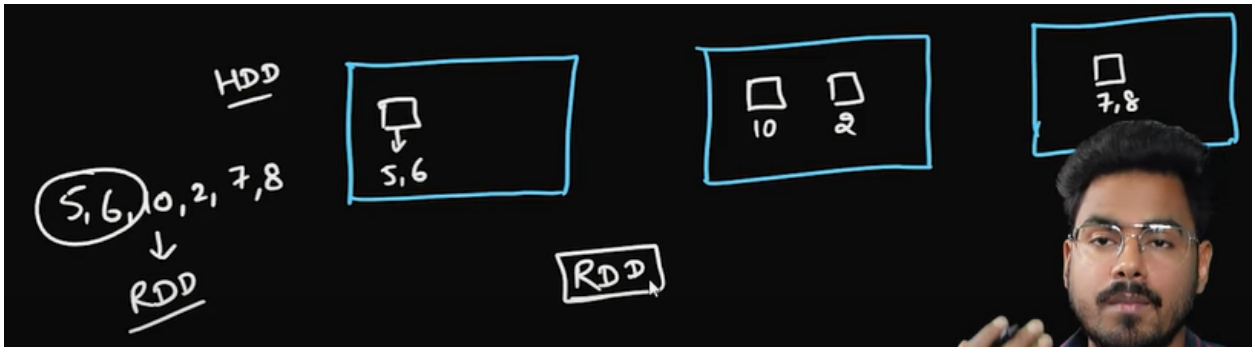
4. Lets consider our list in the ram

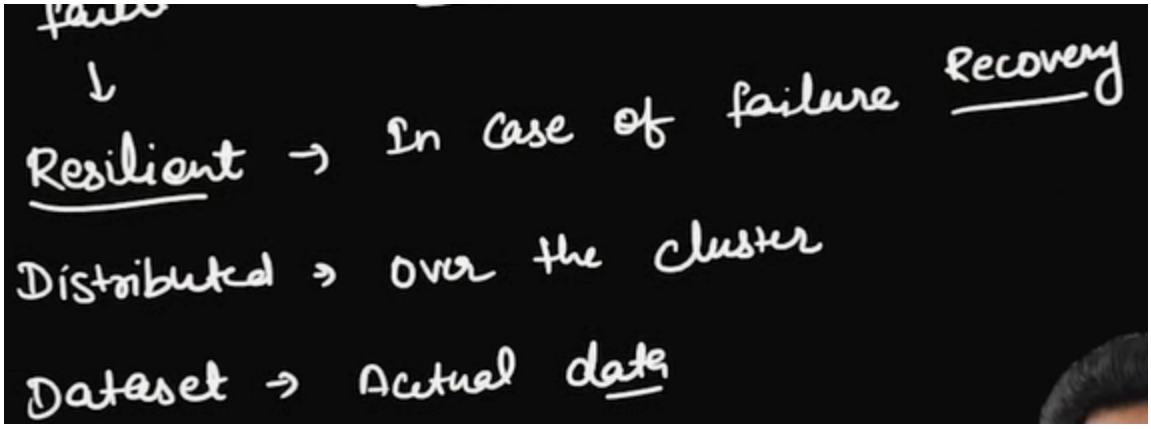


5. Now here RDD is also a type of data structure
6. Consider we have 500MB of data..and by default in hadoop each block stores 128MB..so here we have 4 partition



7. Now what RDD does is..retrieve this partitions of data from all this clusters
8. Suppose our list is in this partitions..then RDD retrieves the data from these clusters



9. 

10. More on RDD : https://g.co/gemini/share/16ec50a6aebf
11. So how is it resilient?
12. Lets suppose we have some data in RDD
13. Now we have applied some filter on RDD and this data will get stored in RDD1..and on RDD1 we have applied another filter and stored in RDD2

    1. **Immutable Data Structure**: RDDs are immutable, meaning once created, they cannot be modified. If you want to transform an RDD, you create a new RDD derived from the original one.
    2. **Lineage**: Every RDD in Spark maintains information about how it was derived from other RDDs. This lineage information forms a directed acyclic graph (DAG), recording the sequence of transformations applied to create the RDD.
    3. **Lazy Evaluation**: Spark employs lazy evaluation, which means transformations on RDDs are not executed immediately. Instead, they are recorded as operations in the DAG. This allows Spark to optimize the execution plan before actually executing it.
    4. **Resilience through Recomputation**: When a partition of an RDD is lost due to a worker failure, Spark can recompute the lost partition using the lineage information. Spark knows the series of transformations applied to the lost partition, so it can simply rerun those transformations to reconstruct the lost partition.
    5. **Data Replication (Optional)**: Additionally, Spark provides an option to replicate data partitions across multiple nodes for further fault tolerance. This is particularly useful for critical data that cannot be recomputed quickly.

14. 

15. So when do we need a RDD?
16. First we'll see disadvantages

17. **Spark's Catalyst Optimizer and RDDs**

    Spark leverages a powerful optimizer called Catalyst to analyze queries and transformations and generate efficient execution plans. However, Catalyst's effectiveness is limited when working with RDDs due to:

    - **Lack of Schema Information:** RDDs don't have a built-in schema that explicitly defines data types for each element. Catalyst relies on this schema information to make informed decisions about optimizations. Without it, the optimizer might have to make assumptions or perform additional checks during execution, impacting performance.
    - **Lambda Functions:** Transformations in RDDs often involve lambda functions for manipulating data. While these functions are flexible, Catalyst can't see "inside" them to understand the exact logic being applied. This limits the optimizer's ability to analyze the entire data processing pipeline and identify potential optimizations.

    Example: Filtering and Aggregation with Limited Optimization

    Imagine an RDD containing user data with elements like `[user_id, name, age, city]`. You want to filter for users above 18 from California and then calculate the total number of users in each city.

    1. **Filtering with Lambda:** You might use a transformation like `filter(data => data[2] > 18 && data[3] == "CA")`. Here, the lambda function encapsulates the filtering logic.
    2. **Limited Optimization:** Catalyst can't analyze the lambda function to see that you're filtering based on age and city. It might miss opportunities for optimization, such as:
        - Pushing down the filtering logic closer to the data source (if applicable) to reduce data movement across the cluster.
        - Reordering transformations for efficiency (e.g., filtering before grouping by city).

18. Advantages
19. RDD's handle unstructured data well and Structured data will be handled by dataframe API and dataset API
20. It is type safe(gives error if any before run time)

21. When do we need to use spark RDD?

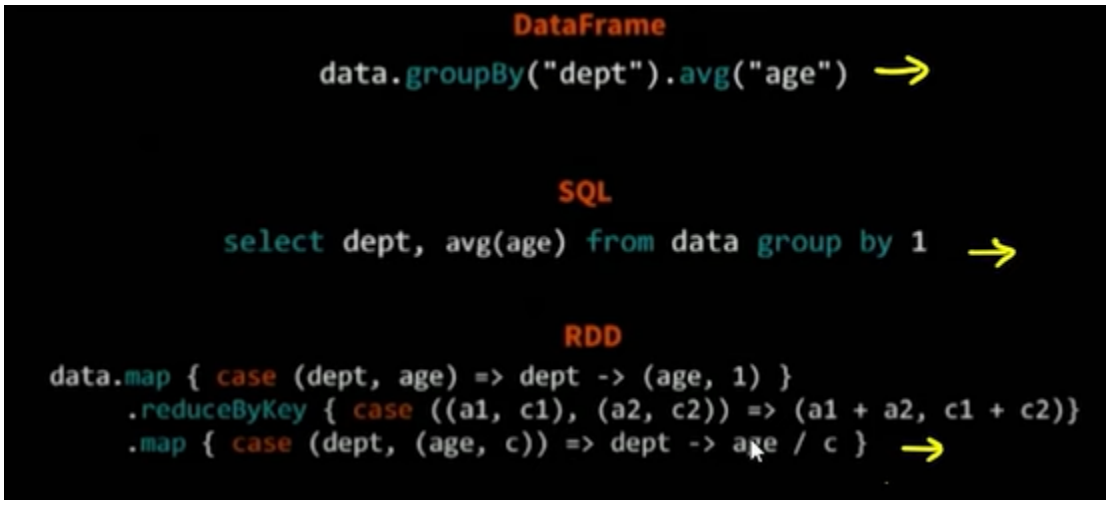
1. Low-Level Control and Customization:

- **Use Case:** If you need fine-grained control over data partitioning or want to implement custom logic that's not easily achievable with DataFrames/Datasets, RDDs offer more flexibility. You can define custom partitioners for specific data distribution across the cluster or write complex transformations using lambda functions.
- **Example:** Imagine you have a dataset with geospatial data and want to partition it based on specific geographic regions. You can achieve this by creating a custom partitioner function within an RDD.

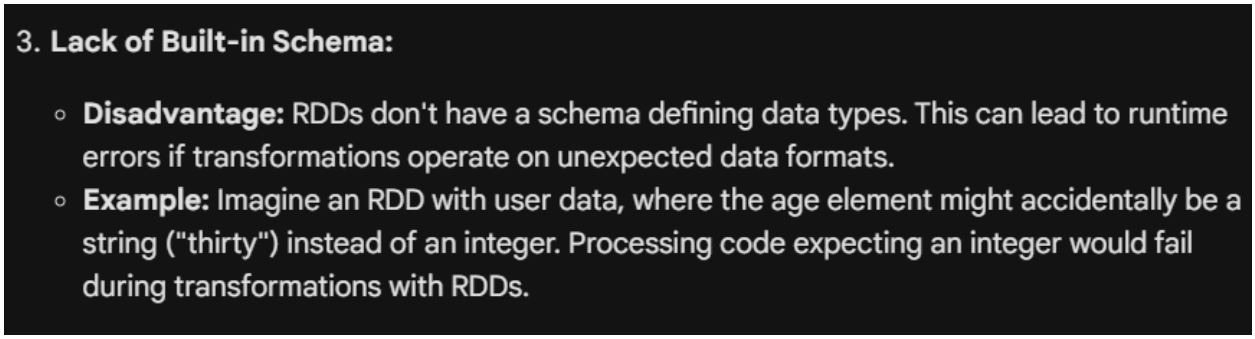
2. Unstructured or Semi-structured Data:

- **Use Case:** While Spark can work with DataFrames/Datasets for some unstructured or semi-structured data formats (like JSON), RDDs might be more straightforward for complex or custom data formats. You can directly manipulate the raw data within RDDs using functional programming techniques.
- **Example:** You might be working with sensor data logs containing various data types and inconsistent structures. RDDs allow you to handle this heterogeneity more easily compared to imposing a rigid schema upfront with DataFrames/Datasets.

22. Why we should not use RDD?
23. To write same code in dataframe,SQL and RDD



24. 



Spark Session and Spark Context

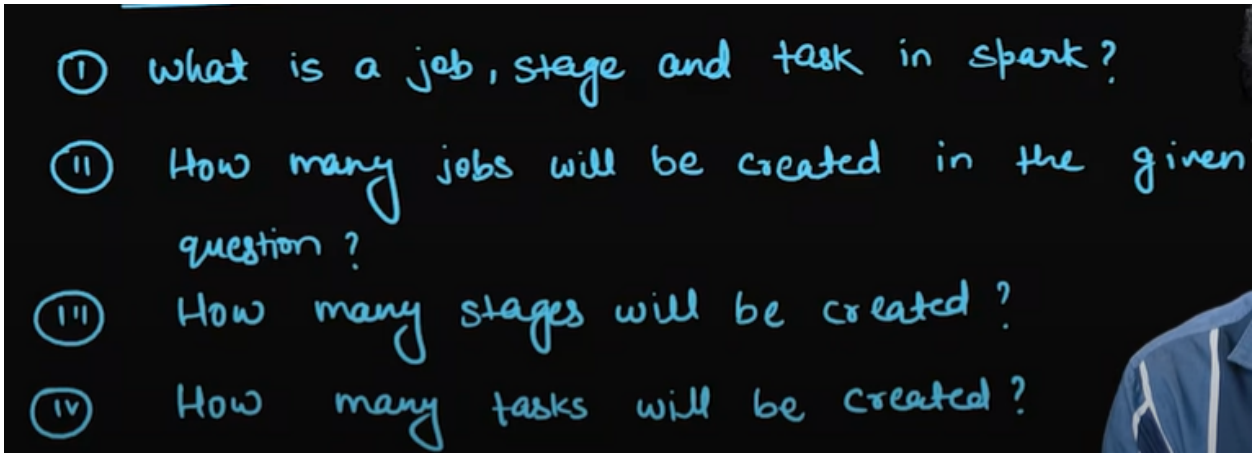
1. If we want to run a spark code in a cluster..then we need to have spark session in the cluster
2. Explained : https://g.co/gemini/share/6b295cd1ed9f

```
findspark.init()
from pyspark.sql import SparkSession
from pyspark.sql.functions import *
spark = SparkSession.builder.master("local[5]")\
        .appName("testing").getOrCreate()
print(spark)
spark2= spark.newSession()
print(spark2)
sc = spark.sparkContext
print(sc)
```

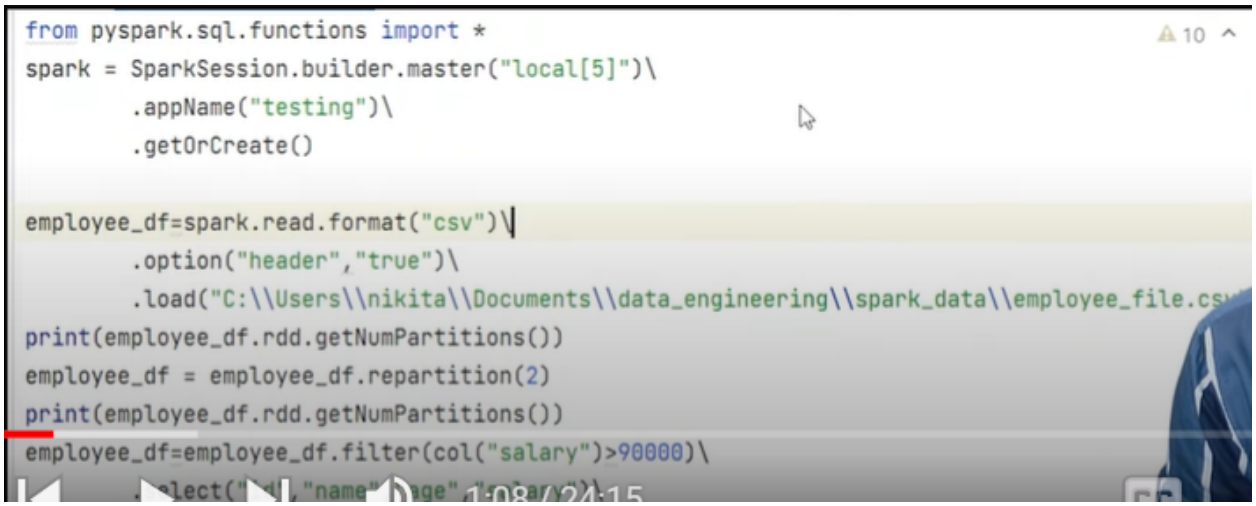
3.
4. Here in this code we have initiated spark session
5. We have assigned spark with node “local[5]” and appName of “testing”..and .getOrCreate()..if there’s existing spark session with this config then it gets that ..or else it creates new

Spark Job, Stages, Tasks

1. Potential interview questions



2. We may be also given a piece of code and may ask …how many stages or tasks will be created



3. So what is job,stage,task and application in spark?
4. Application : when we submit a program of spark in a spark cluster..then it is called a spark application

1. Application:

- Think of it as the **big picture.** It represents your entire Spark program, encompassing all the data processing logic you want to execute. This program defines what data you'll work with, the transformations you'll perform, and the desired outcome.

5. Job :

2. Job:

- Imagine a **specific action** within your application. A job is triggered by an **action** in your Spark program, such as `count()`, `collect()`, or `write.save()`. This action signifies the completion point for a particular data processing task.

Example: You might have a Spark program that reads a large CSV file, filters entries based on a specific criteria, and then counts the number of remaining entries. Here, the action could be `data_df.filter(...).count()`.

6. Stage is not but processing of a job

7. For example here first we have to read a csv file and then apply a filter…and after that apply another filter etc

- A job can be further divided into **logical units** called stages. Stages represent a set of related transformations that can be executed **without shuffling data across machines**. Shuffling involves moving data between different worker nodes in the Spark cluster, which can be resource-intensive.
- Stages are determined by Spark's optimizer based on the dependencies between transformations in your program. Transformations that don't require data exchange can be grouped into a single stage.

8. Task - here the actual thing is done

Task Breakdown:

1. **Job:** This program defines a single job triggered by the `count()` action, which calculates the number of unique visitors.
2. **Stages:** Spark might identify two stages in this example:
   - Stage 1: Reading the log file from HDFS and creating the initial DataFrame (`log_df`).
   - Stage 2: Filtering based on country (`filtered_df`) and then performing distinct count (`unique_visitors`).
3. **Tasks:**
   - **Stage 1 Tasks:** These tasks would be responsible for reading specific partitions of the log file from HDFS and converting them into DataFrame rows. The number of tasks here would depend on how the log file is partitioned in HDFS (e.g., if it's split into 10 partitions, there would be 10 tasks in this stage).
   - **Stage 2 Tasks:** These tasks would operate on the filtered DataFrame (`filtered_df`) created in the previous stage. They would likely involve:
     - Filtering tasks: Each task might process a partition of `filtered_df` to keep only entries for US visitors.
     - Counting tasks: These tasks would further process the filtered data (potentially another set of tasks depending on partitioning) to count the distinct IP addresses (unique visitors).

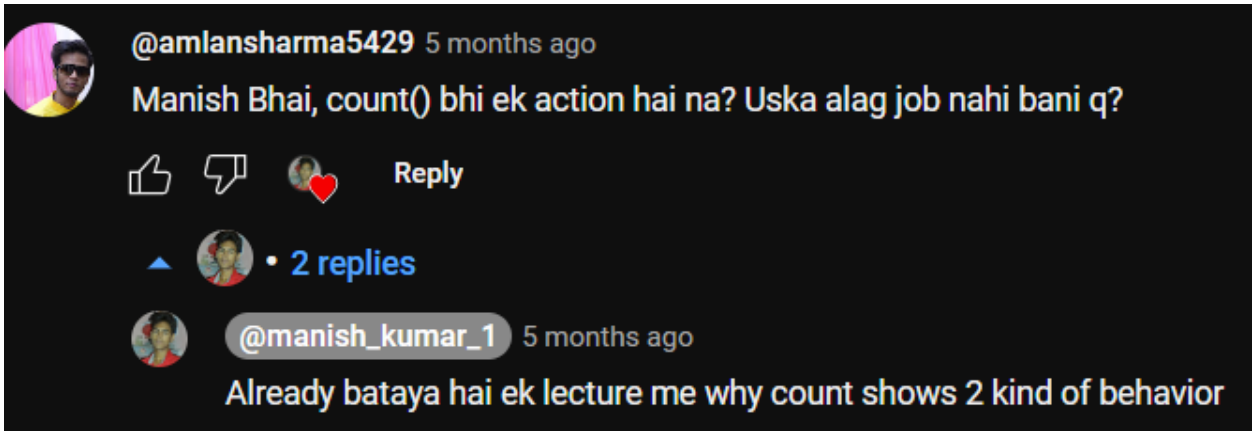
9. Now here each application may have multiple jobs and each job can have multiple stages and each stage can have multiple tasks

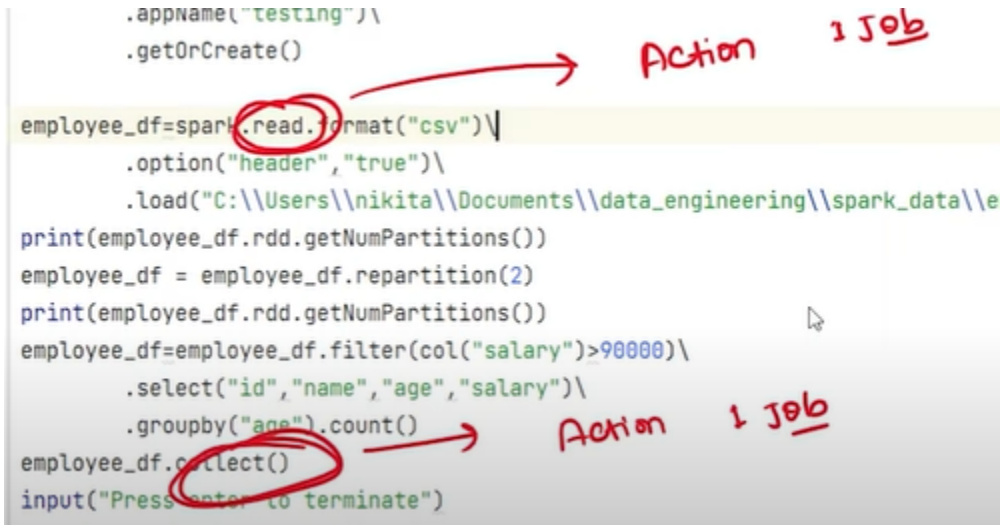
10. So lets get to the code

```
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")
```

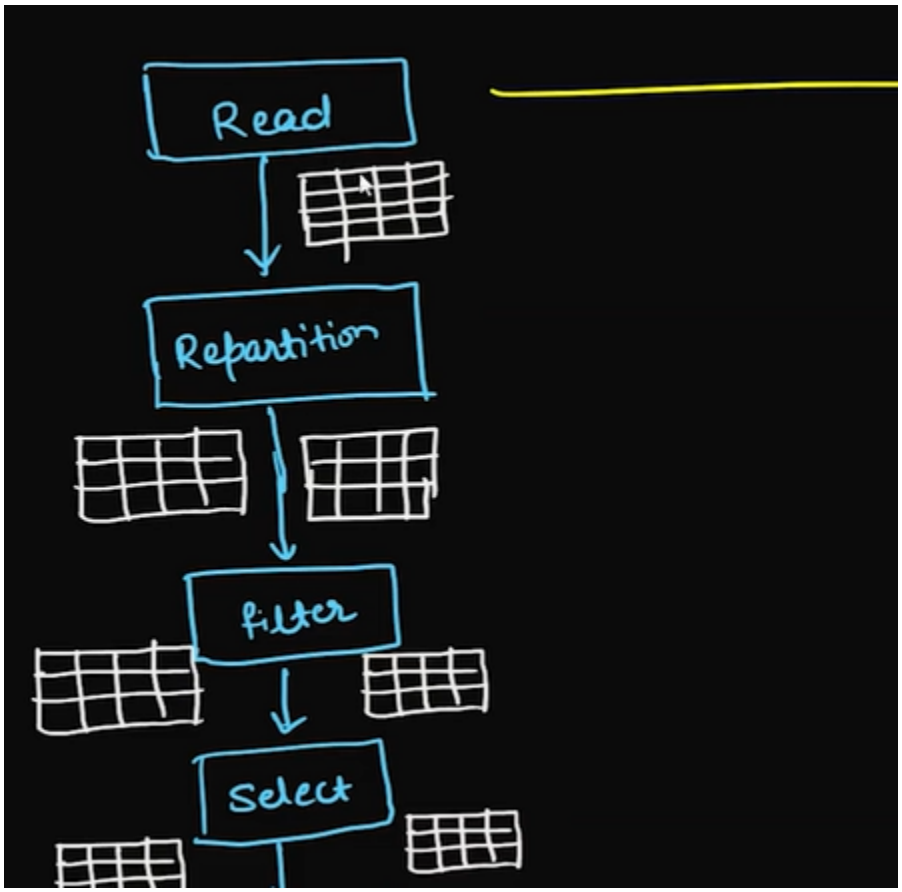
11. How many jobs have created in the above code?
12. So here we have 2 jobs ..which are read and collect()





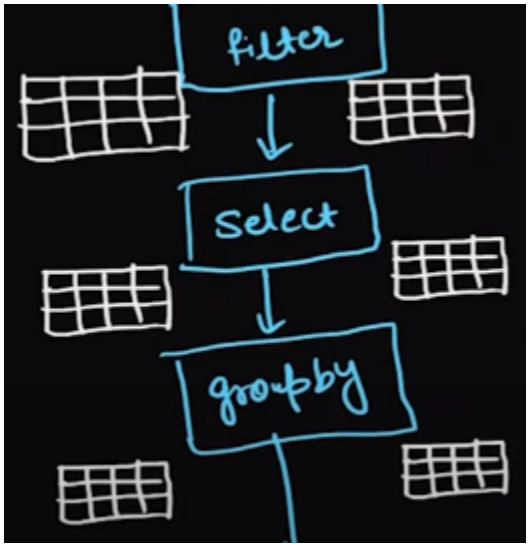
13. And also here employee_df.repartition(2),groupby are wide DT and filter,select are Narrow DT

14. Lets see our code in flow char

15. 

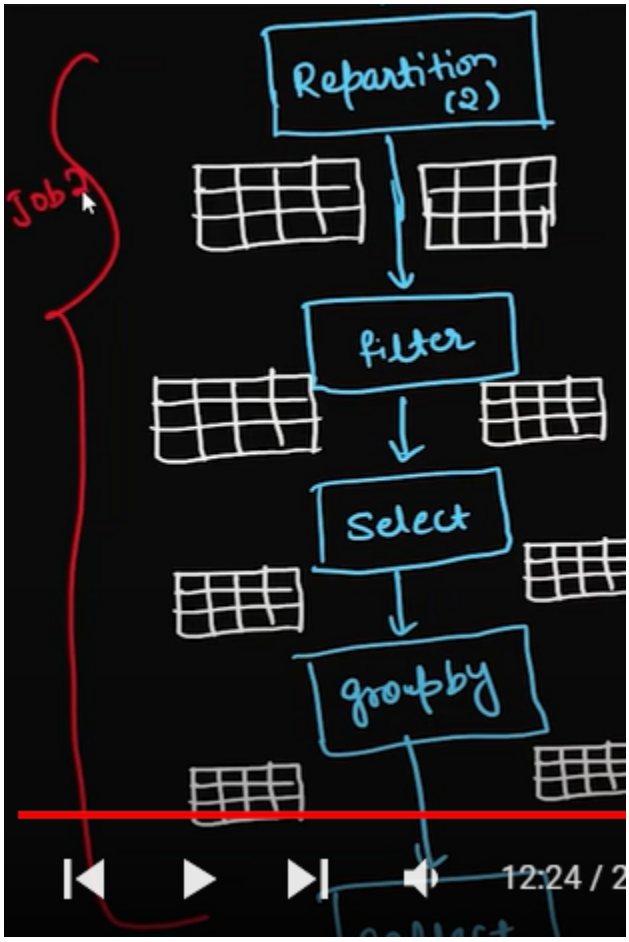
16. Here we have splitted our 128MB data into two repartitions

17. And now all the transformation will be made in these two repartitions



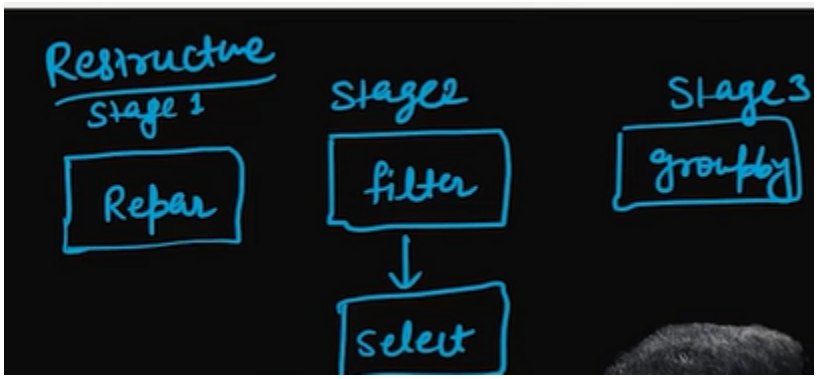
18. So now here the problem occurs at the group by..as it is wide DT..first it shuffles the data and then makes group by.
19. How many stages are created?
20. If there's a job..then there will min 1 stage and 1 task
21. Here read to repartition is our first job

22. Now until we hit the action the current job will not end

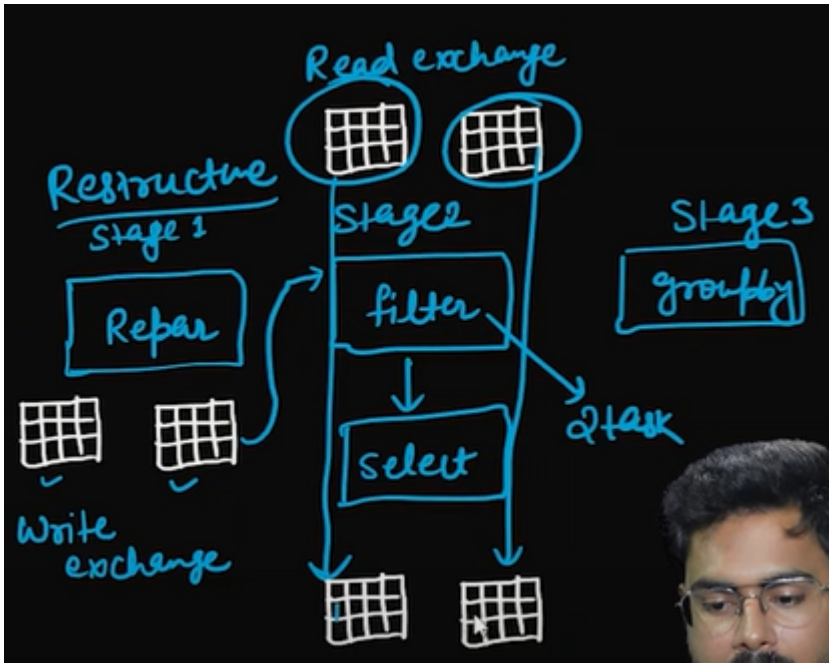


here in the end we have a collect() action ..so our job2 ends here

23. So here in job2 …repartition is our first stage(wide DT) …and in next stage there will be narrow DT(filter,select) and in stage3 we have group by which is wide DT



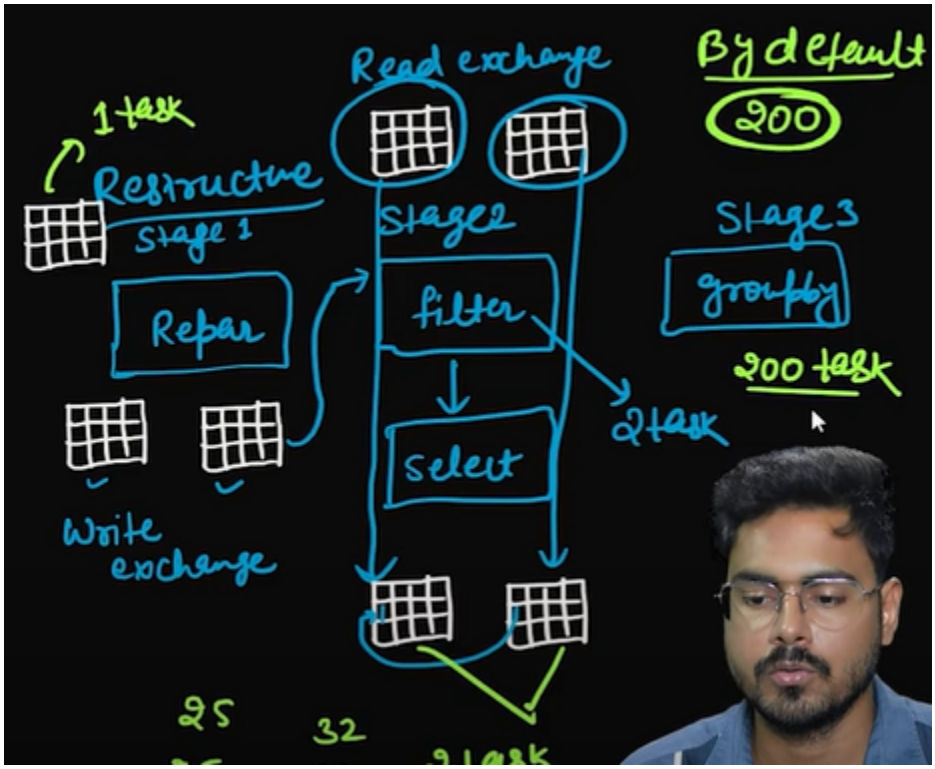
24. So in job1(read-repartition) we have 1 stage..and in job2(repartition - collect) we have 3 stages..total we have 4 stages here
25. How many tasks are created?

26. So when we use repartitions(2) ..it used write exchange and shuffles the datai into two sets of data 

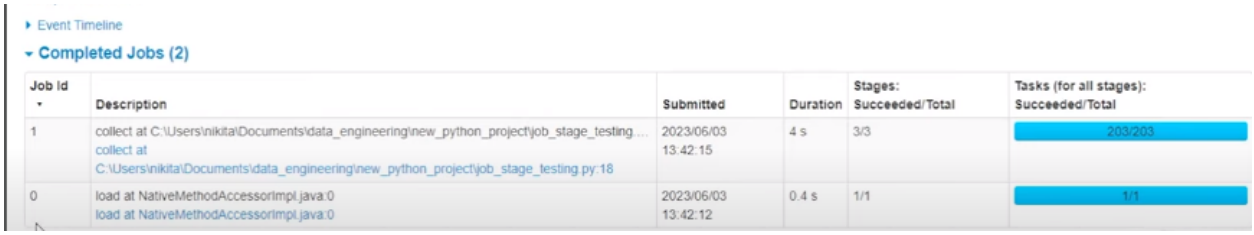
27. Now this data will be consumed by stage2 using read exchange ..as it needs to perform transformations(filter and select) which are our tasks..so total in stage2 we have 2 tasks
28. Coming to groupby..it needs to shuffle the data from these two repartitions.
29. So group by creates partitions for each col which is(group by("col")) value…by default it creates 200 partitions 

30. So for example ..if we groupby on age…then age=25 will be in one partition and age = 32 will be in one partition
31. So here no.of partitions = no.of tasks

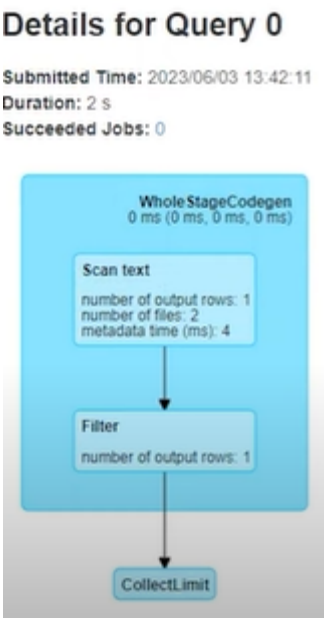
32. So here in total we have 203 tasks



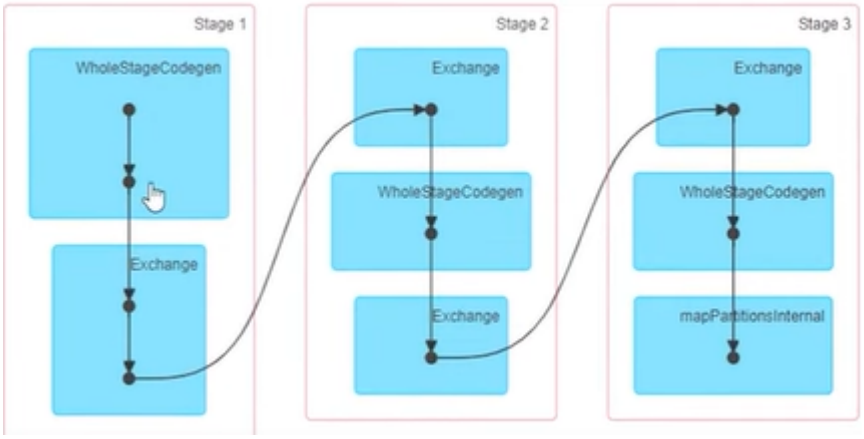
33. Lets run our sample code now…and check everything on its UI
34. If we want to see in the UI then we can go to localhost4040..see the above code created



35. Detailed job 0

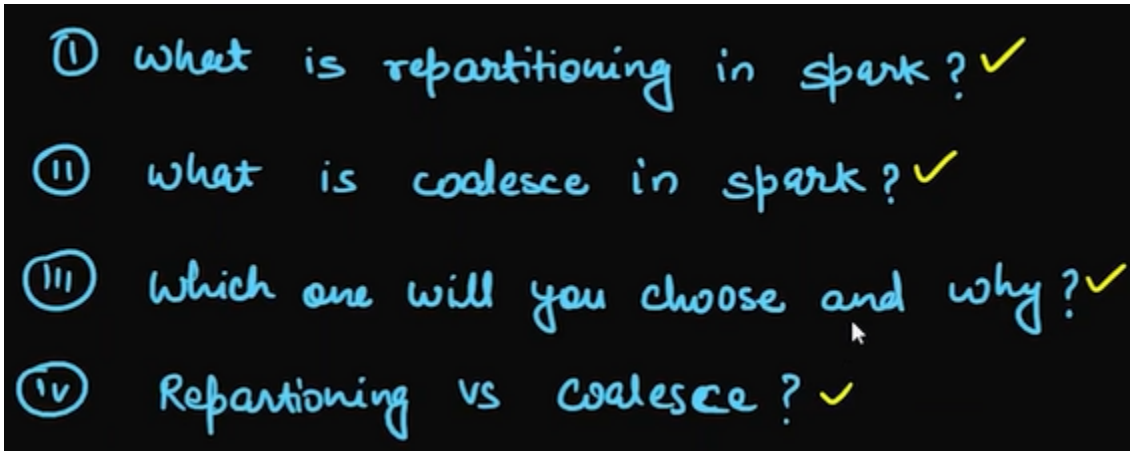


36. Detailed job 1

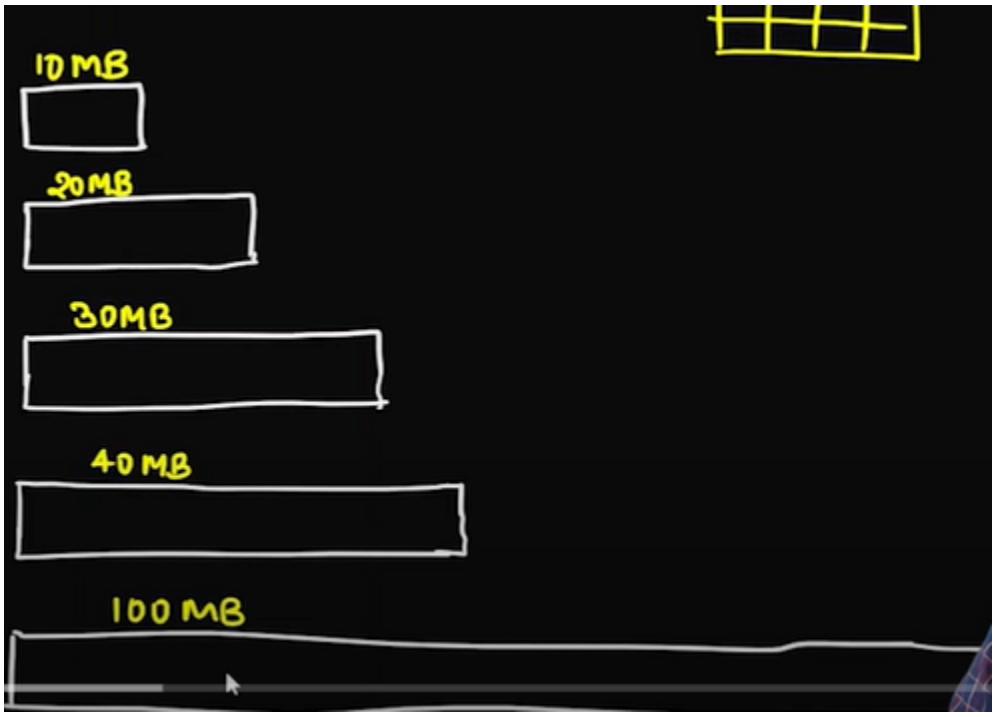


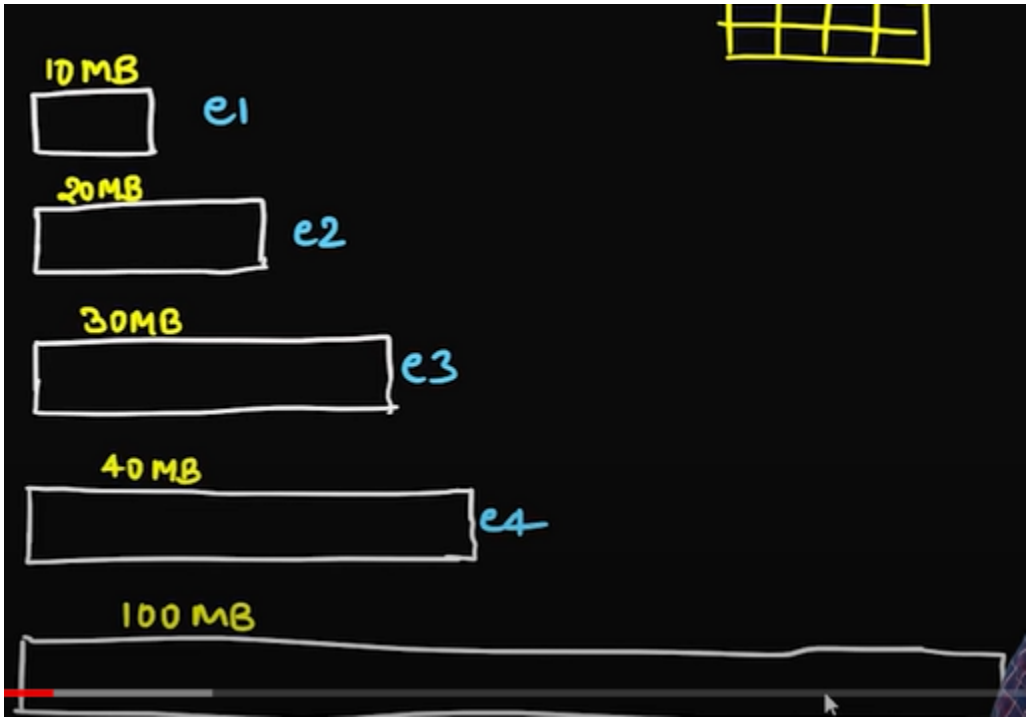
Repartition and coalesce

1. Potential question include



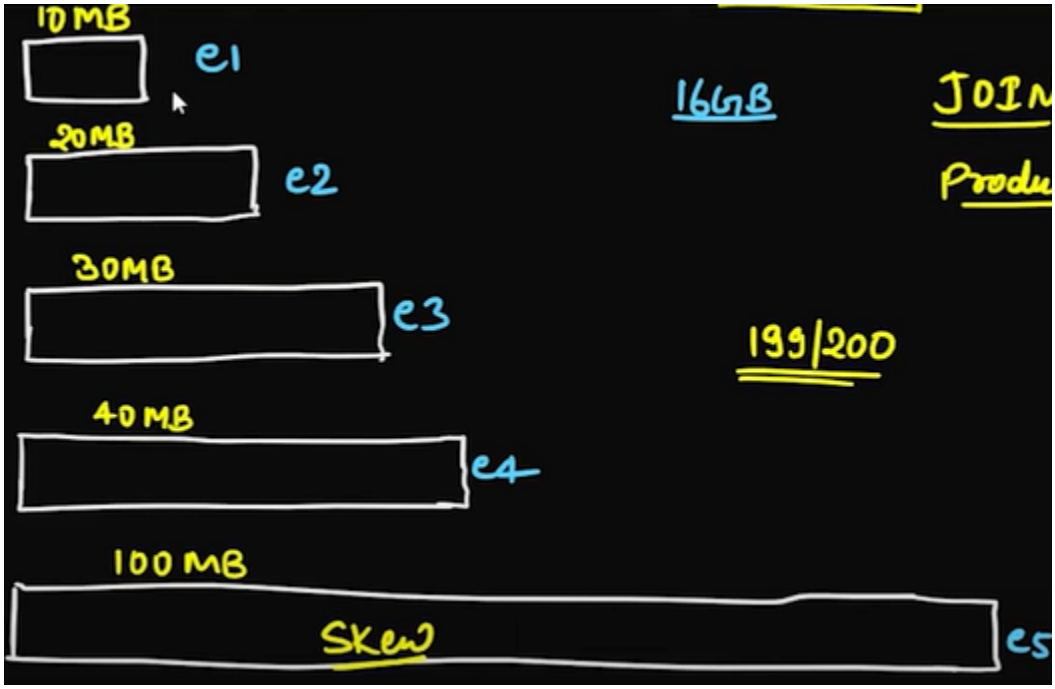
2. First let's understand repartition and coalesce came into existence
3. Lers suppose our data has partitioned like this

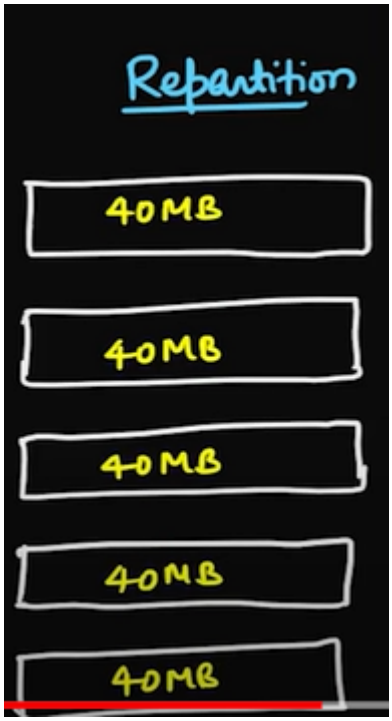


4. So if we process our data..then the partition with less data will get processed first and partition with large data ..then it takes more time for processing…lets assume each partition has one executor
5. 

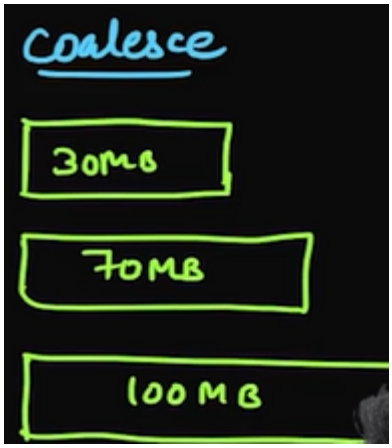
6. Now ..executore with small partitions will complete their processing fast..and will wait for e5 to complete its processing
7. Also if we assume we have 15gb ram…then each executor will have 3gb of ram
8. Small partitions will get executed fast…and the 3gb ram will get wasted for small tasks
9. So to deal with this we use repartition and coalesce
10. Repartition

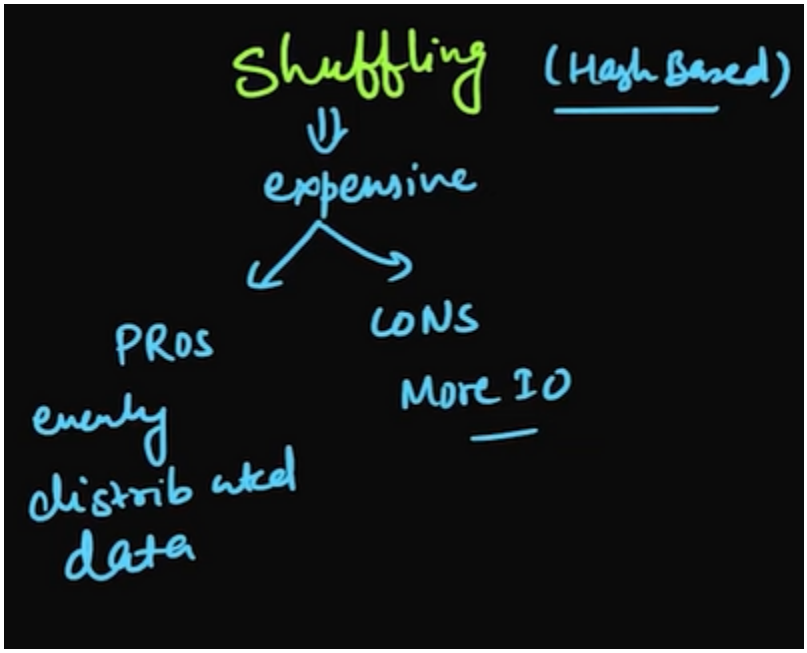
11. Here total we have 200MB of data and we have 5 executors

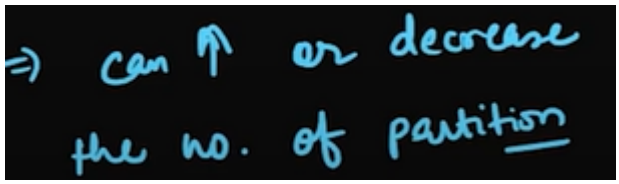




12. Repartition helps us evenly split our data partition
13. It shuffles the data and make our partition size even
14. Coalesce is used for merging partitions..

15. Here it merged and gave us 3 partitions 

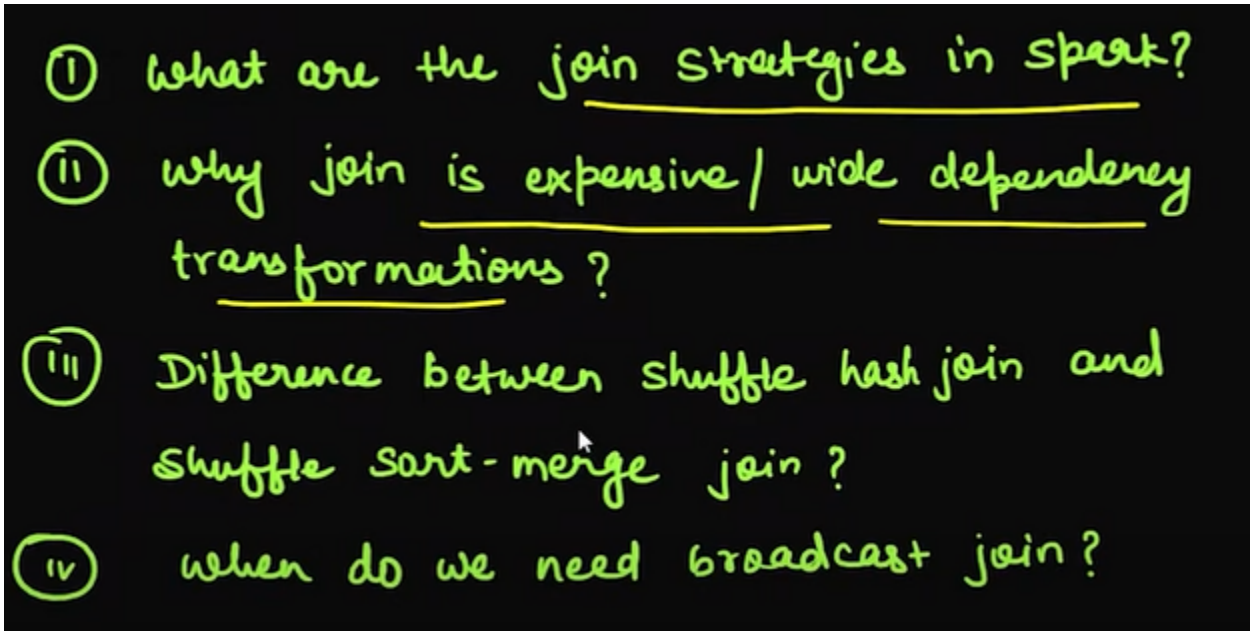
16. Repartition and coalesce explained : https://g.co/gemini/share/ab1bf4c3db4f
17. So in repartition there is shuffling and in coalesce no shuffling
18. 

19. In repar : 

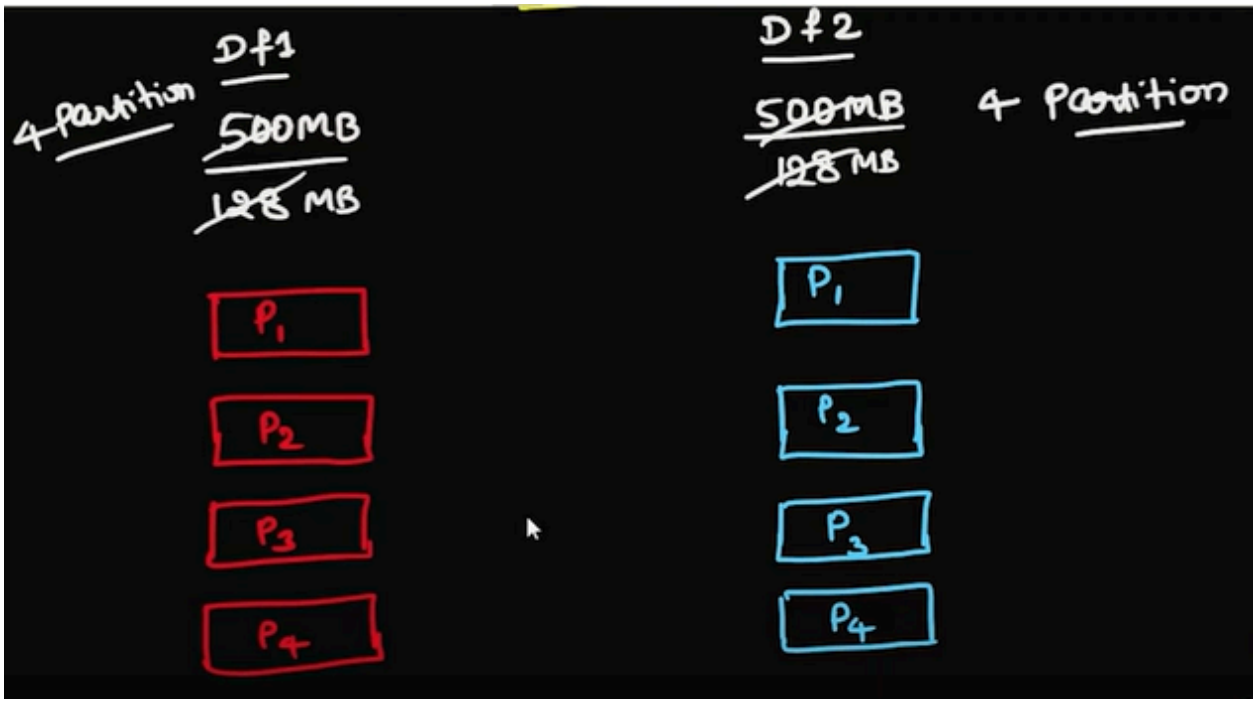
20. In coal : 

Spark Joins | Sort vs Shuffle

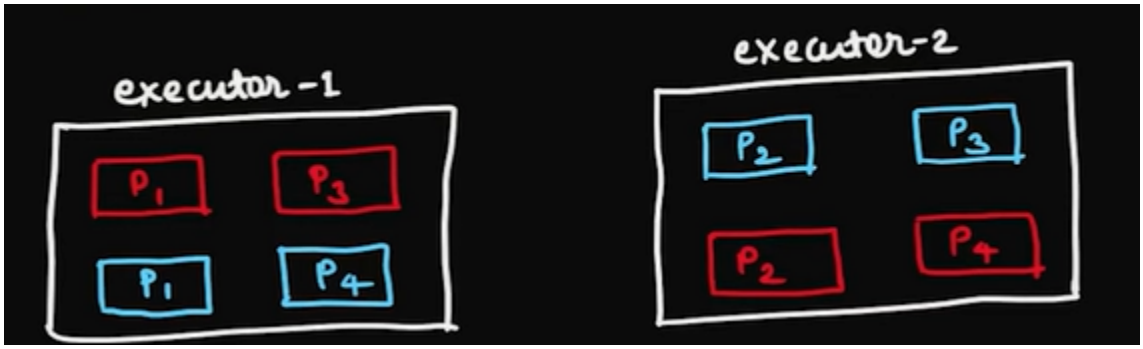
1. Potential interview questions



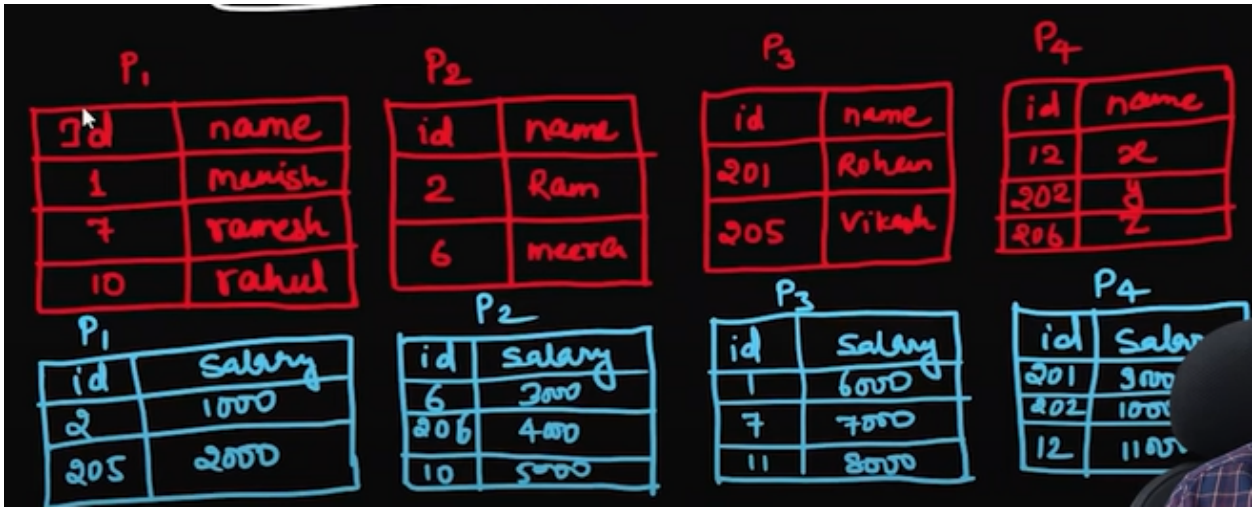
2. Basically joins are very expensive..bcuz they shuffle the data
3. Lets assume we have two dataframes of 500MB each…and by default each partition has 128MB
4. So here df1 has 4 partitions and df2 has 4 partitions



5. Here we have 2 executor and assume that our partition are like this due to repartitions



6. And the data in the partitions look like this



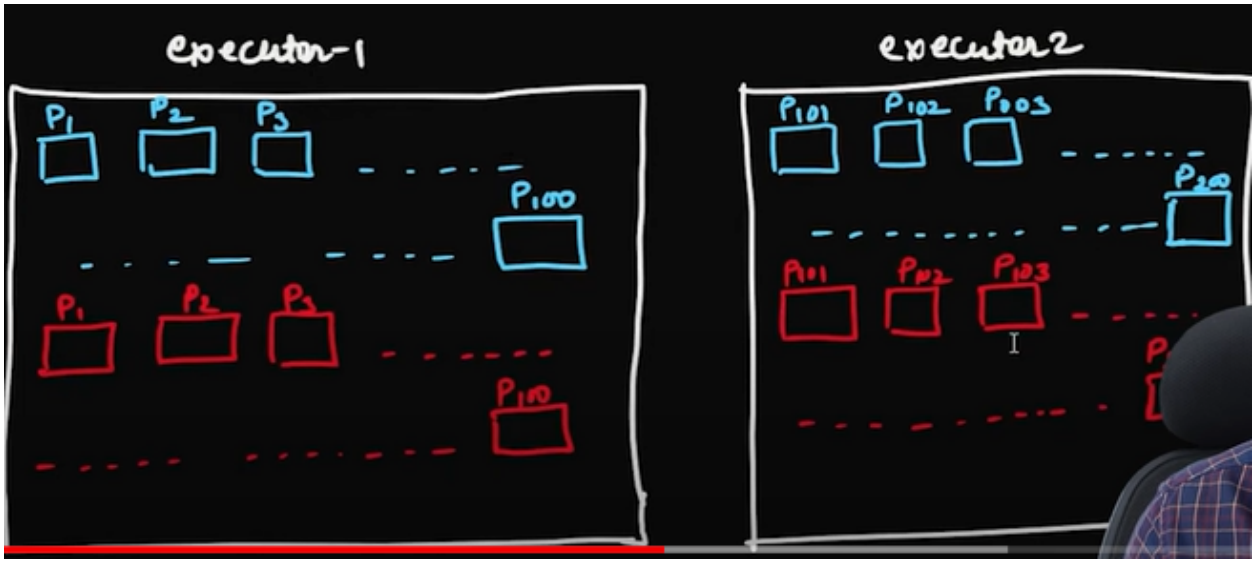
7. Df1 is emp data and df2 is salary data
8. Now we need salary of each emp
9. Lets consider id = 1

id	name	salary
1	manish	6000

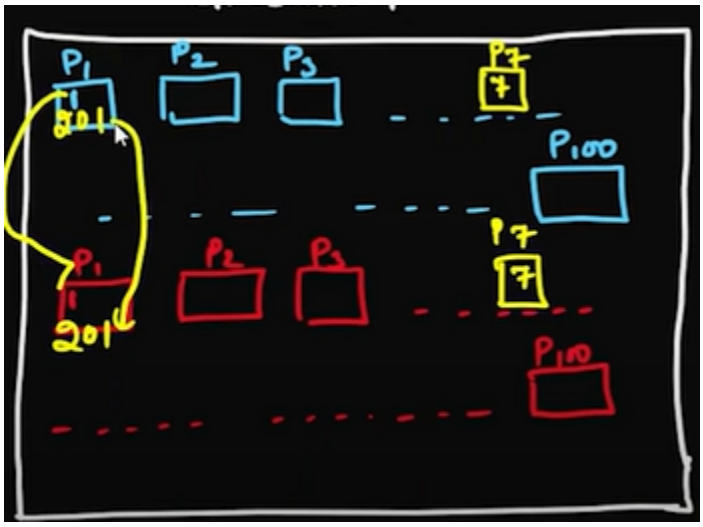
10. Now executor1 needs data from ex2

11. So now we shuffle the data using default partition

12. In default partition each executor will have 200 partitions

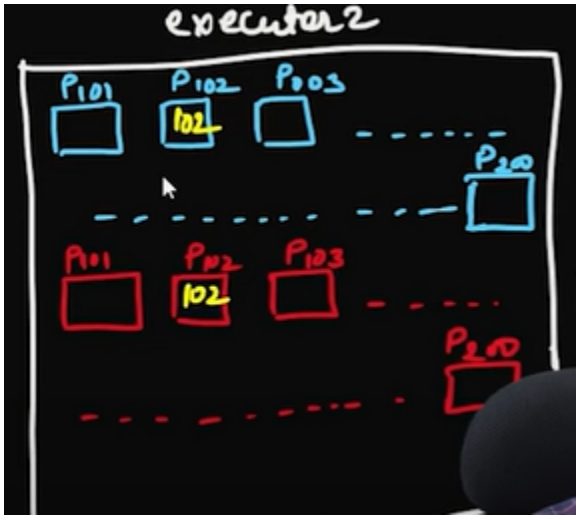


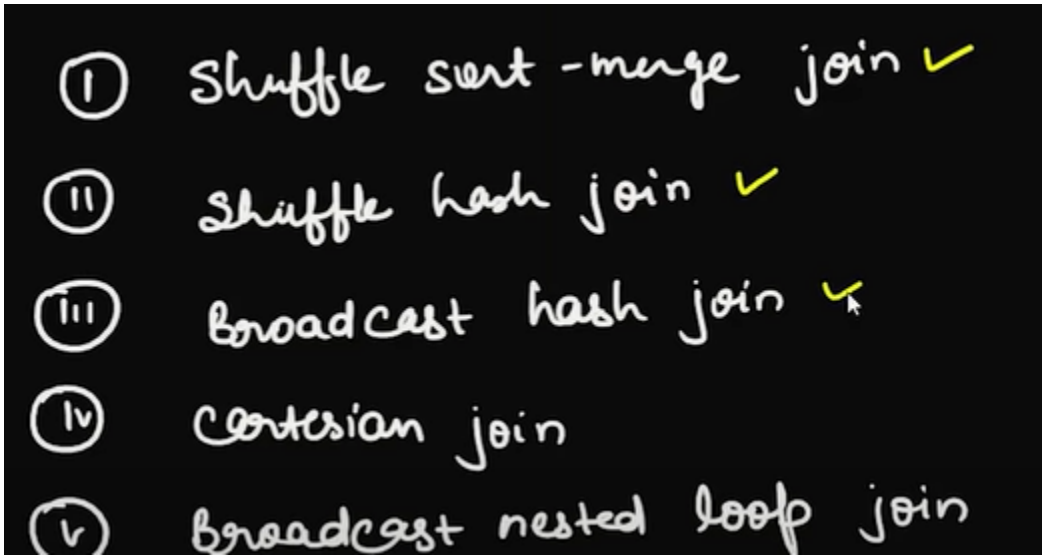
13. So in partition 1 we have data of id which when divided by 200 gives 1
14. Example 1/200,201/200..these data will get stores in partition 1
15. Similar id = 7…7/200 = 7..this data will get stores in p7
16. Now we can join this data using these partitions



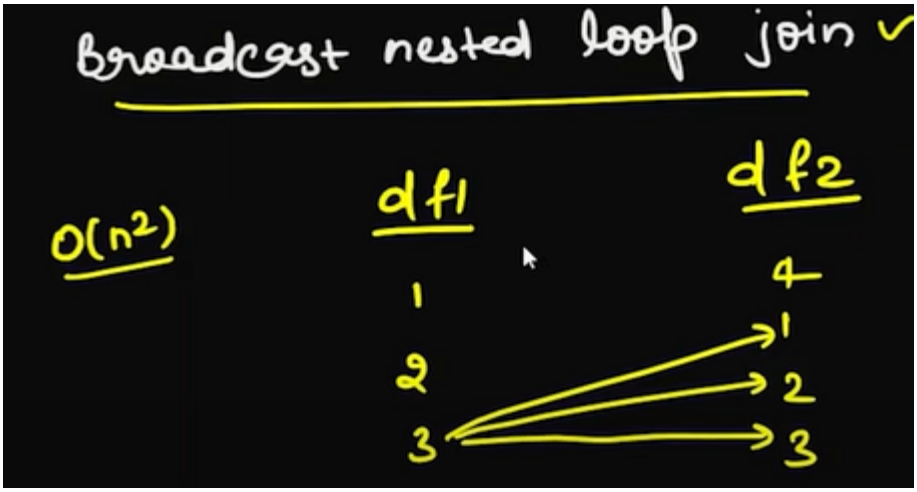
17. Here blue p1 stores id =1 and emp_name and red p1 stores id and emp_salary

18. If we have id = 102..then it goes to executor 2



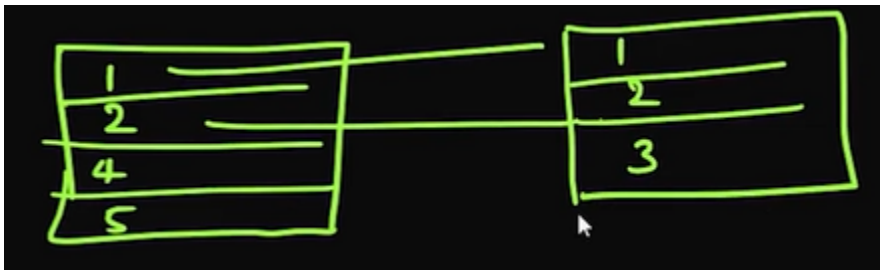
19. If there are lot of shuffles..then our cluster gets choked and performance will get degraded
20. Strategies in spark
21. 

22. 5th one is most expensive one

23. 

24. Shuffle sort merge join
25. Lets consider we have 2 df..which undergone shuffled already

26. After sorting we get our df's like this..and time complexity of sorting is nlogn



27. After sorting it joins the id's of 2 df

28. Shuffle Sort Merge Join (SSMJ) is the default join strategy in Apache Spark for large datasets (since Spark 2.3). It efficiently joins two DataFrames based on a shared key using a three-step process: shuffle, sort, and merge.

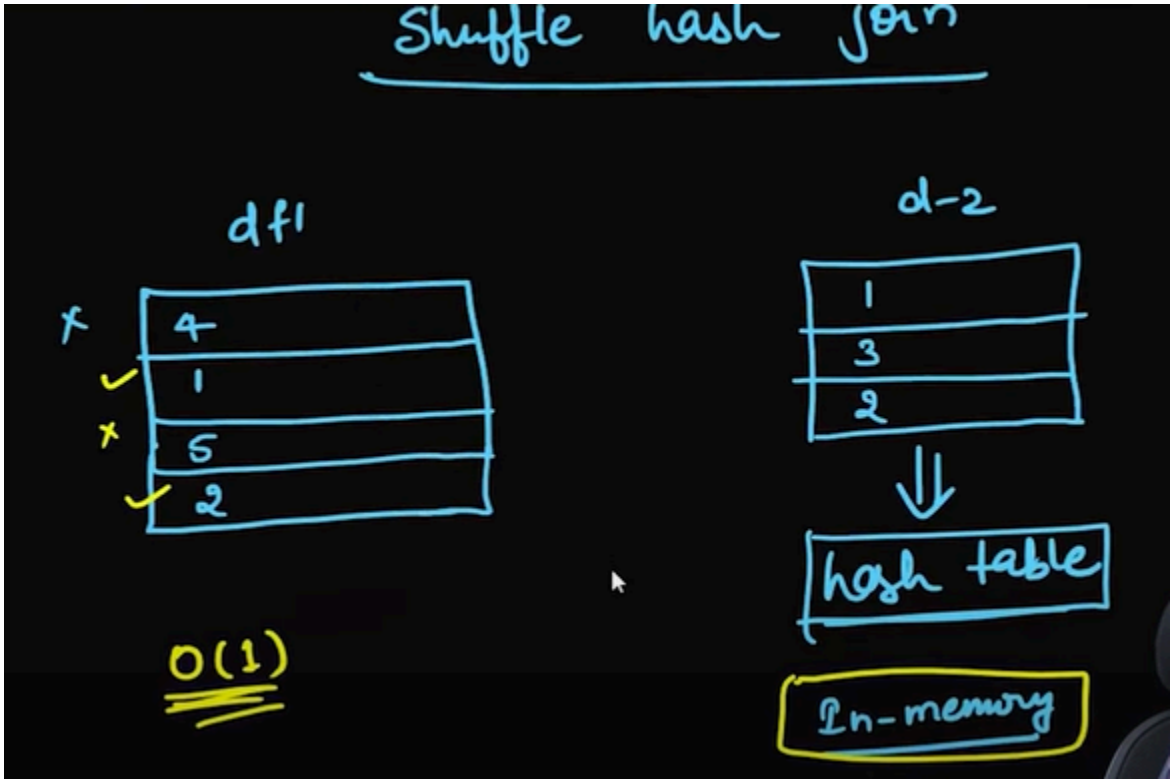
Here's how it works:

1. **Shuffle:**
   - Both DataFrames are shuffled and repartitioned based on the join key. Spark utilizes a hash function to determine the target partition for each record based on its join key value.
   - Records with the same join key are sent to the same executor node, ensuring they are co-located for efficient joining. This shuffle operation involves data movement across the network.
2. **Sort:**
   - On each executor node, the repartitioned data within each partition is sorted by the join key. Sorting allows for efficient merging in the next step.
3. **Merge:**
   - Each executor iterates through the sorted partitions of both DataFrames simultaneously.
   - By leveraging the sorted order, Spark can efficiently compare join keys and produce the joined output row by row without additional shuffling.

29. Shuffle hash join

1. **Shuffle and Partitioning:**
   - Similar to SSMJ, both DataFrames are shuffled and repartitioned based on the join key using a hash function. Records with the same join key are likely to end up on the same executor.
2. **Build and Probe Phase:**
   - Spark designates one DataFrame (usually the smaller one) as the "build" side. This DataFrame is used to construct an in-memory hash table on each executor node. Each entry in the hash table is a join key and a pointer to the corresponding records.
   - The other DataFrame becomes the "probe" side. Each record in the probe side is sent to the relevant executor based on its join key (using the same hash function as before).
   - On each executor, the probe side records are looked up in the build side's hash table using their join key. This lookup is efficient because the hash table provides direct access based on the key.
3. **Join and Output:**
   - If a matching join key is found in the hash table, Spark joins the records from both sides and generates the output row.
   - This process continues for all probe side records on each executor.

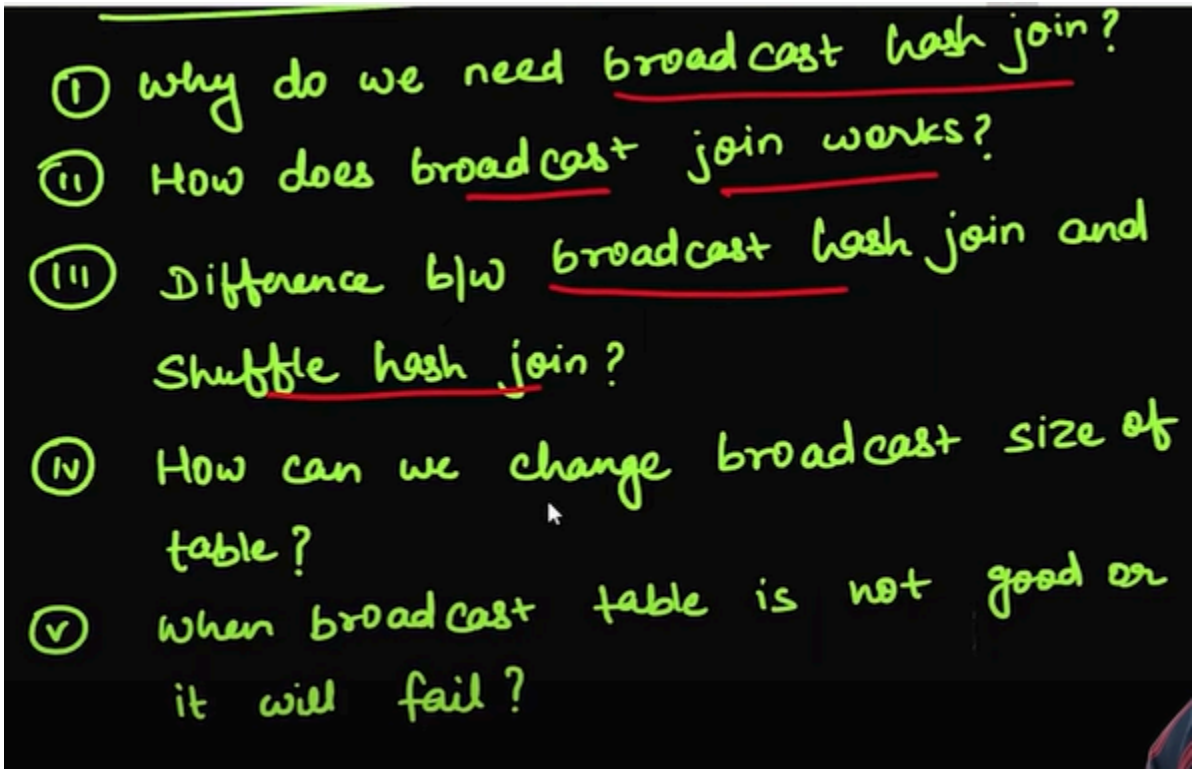
30. In shuffle hash join..here spark creates a hash table for smaller df



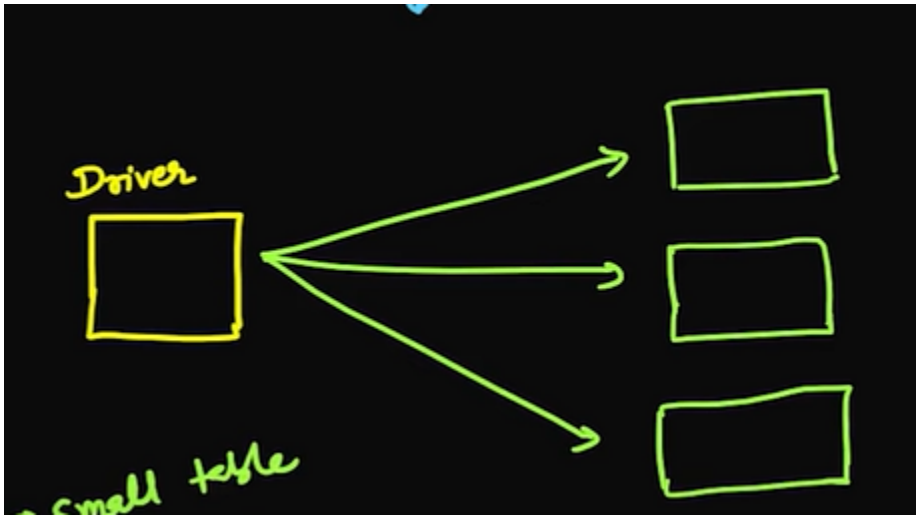
31. Then it iterates the df1 rows…and if the value of df1 is present in hashtable then it joins
32. Here time complexity is O(1) and space complexity is O(df2)

Broadcast join

1. Potential interview questions

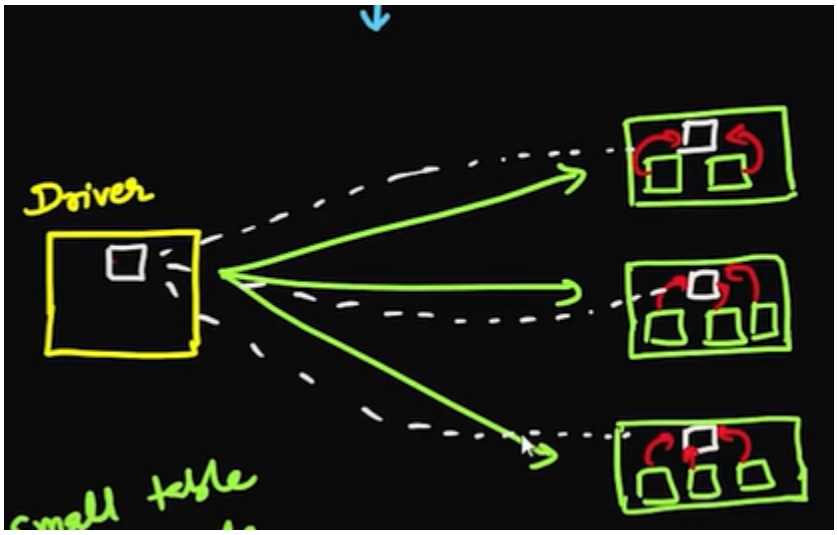


2. Lets assume we have a driver and 3 executors



3. And also assume we have one small table which 5mb and 1 large table which is 1GB

4. And our 1 gb file has 8 partition(128MB each) and these partitions are in these 3 executors
5. Now as the small table size is small..we'll send the small table to every executor …so that we can avoid shuffling which saves time and processing



6. So here our driver stores the small table and broadcast it to executors to perform join operation
7. But what if our small table size is 1gb and driver size is 2gb…then driver node will not be able to accommodate the small table and gives driver out of memory..if memory of driver becomes very less
8. Also if our small table size is large..then to send the table from driver to executors will also consume network speed..which slow down transfer speed
9. Similarly for executors also…if our small table size is 1 gb and executors has 2 or 3gb..then while performing joins..we'll get out of memory error
10. So while broadcasting we will consider small table size ..as per the executor and driver node capacity
11. Why do we need broadcast hash join?

12. The main answer is to get rid of shuffle

Concept:

- In a regular join, both DataFrames are shuffled and distributed across the cluster. This can be inefficient if one DataFrame is significantly smaller.
- Broadcast join keeps the smaller DataFrame (the "broadcast table") entirely in the memory of each worker node in the Spark cluster. This eliminates the need to shuffle it for each join operation.

Benefits:

- **Reduced Network Traffic:** No shuffling of the smaller DataFrame translates to less data movement across the network, leading to faster join processing.
- **Improved Efficiency:** Each worker node can directly access the broadcast table from memory, avoiding repetitive data transfers for each join.

13. Lets do this practically
14. So here is our customer_data

customer_id	customer_name	address	date_of_joining
1	manish	patna	30-05-2022
2	vikash	kolkata	12-03-2023
3	nikita	delhi	25-06-2023
4	rahul	ranchi	24-03-2023
5	mahesh	jaipur	22-03-2023
6	prantosh	kolkata	18-10-2022
7	raman	patna	30-12-2022
8	prakash	ranchi	24-02-2023

15. Sales data

customer_id	product_id	quantity	date_of_purchase
1	22	10	01-06-2022
1	27	5	03-02-2023
2	5	3	01-06-2023
5	22	1	22-03-2023
7	22	4	03-02-2023
9	5	6	03-03-2023
2	1	12	15-06-2023
1	56	2	25-06-2023
5	12	5	15-04-2023
11	12	76	12-03-2023

16. For practical follow vid