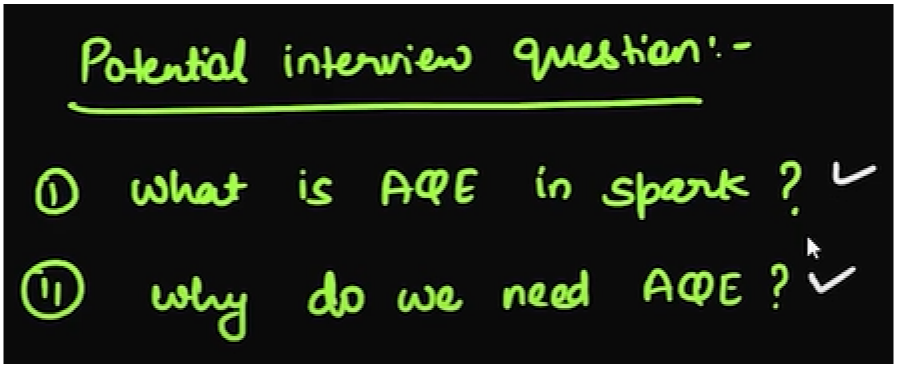
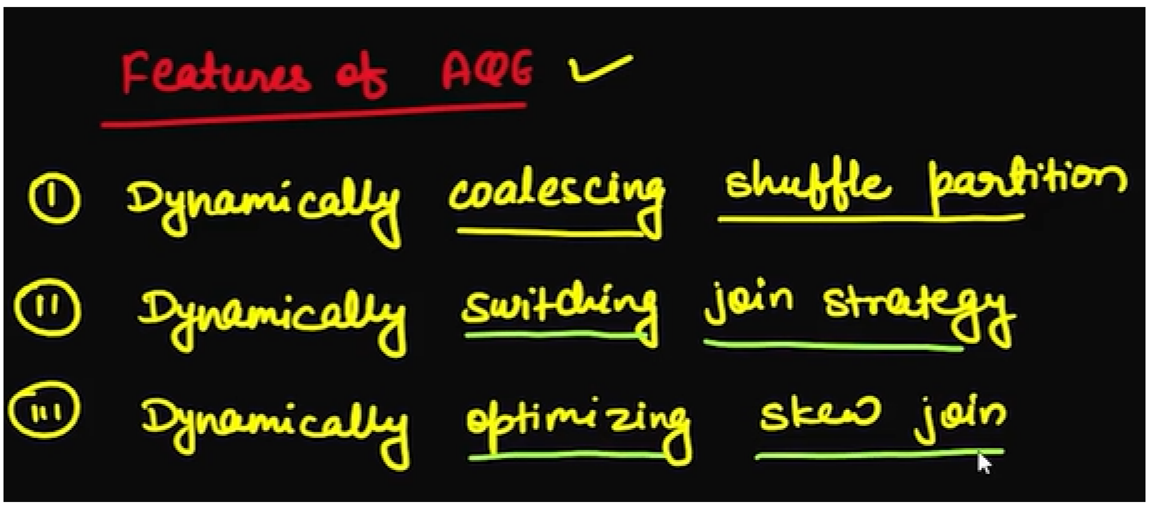
Adaptive Query Execution (AQE

1. Potential interview questions include




2. AQE gives us flexibility to change the query in the run time
3. It also changes the query dynamically in the run time. Example : https://g.co/gemini/share/5a456592b10f
4. It has 3 capabilities..which helps us in dynamical run time
5. 

6. Dynamically coalescing shuffle partition
7. Lets understand this by sample data

8. Here we have product_dimensional data..which helps us finding the old price of the product

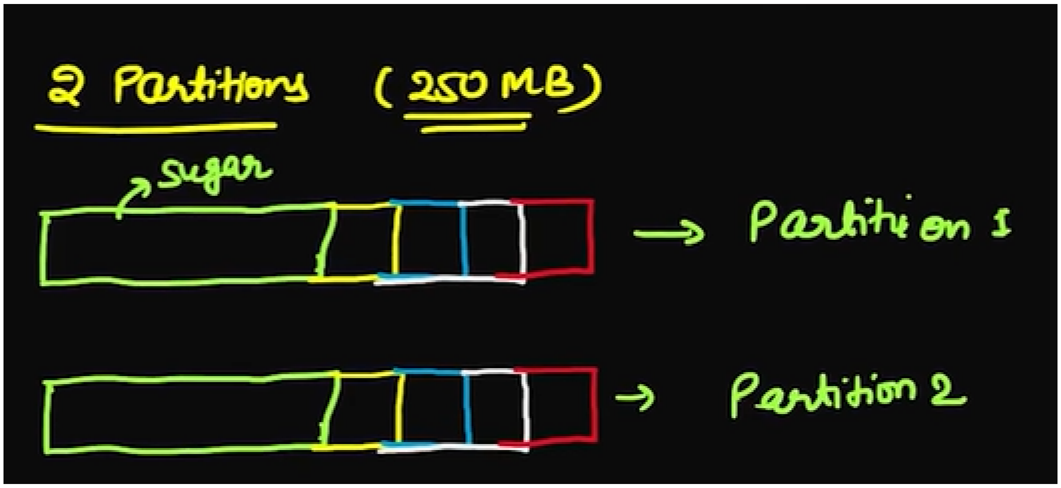
```
product_dim_df.show()

(1) Spark Jobs

+------+-----------+-------------+------------------+-------------------+-------------------+-------
|    id|       name|current_price|         old_price|       created_date|       updated_date|
+------+-----------+-------------+------------------+-------------------+-------------------+-------
|153690|refined oil|        110.0|              88.0|2023-05-17 00:00:00|2023-08-31 00:00:00|2023-08
| 75576|      maida|         20.0|              16.0|2023-03-02 00:00:00|2023-08-28 00:00:00|2023-03
| 17269|quaker oats|        212.0|169.60000000000002|2023-06-01 00:00:00|2023-07-16 00:00:00|2023-05
| 62652|      maida|         20.0|              16.0|2023-06-29 00:00:00|2023-08-26 00:00:00|2023-05
| 37409|      sugar|         50.0|              40.0|2023-08-10 00:00:00|2023-03-05 00:00:00|2023-06
| 15208|quaker oats|        212.0|169.60000000000002|2023-07-09 00:00:00|2023-05-12 00:00:00|2023-06
|204309|  dantkanti|        100.0|              80.0|2023-05-02 00:00:00|2023-07-11 00:00:00|2023-05
|106831|      besan|         52.0|              41.6|2023-07-14 00:00:00|2023-07-03 00:00:00|2023-05
|126342|refined oil|        110.0|              88.0|2023-04-11 00:00:00|2023-07-01 00:00:00|2023-05
|248306|   nutrella|         40.0|              32.0|2023-07-21 00:00:00|2023-03-25 00:00:00|2023-07
```

9. This is a product_fact_table…which is skewed(80% of product_name is sugar)

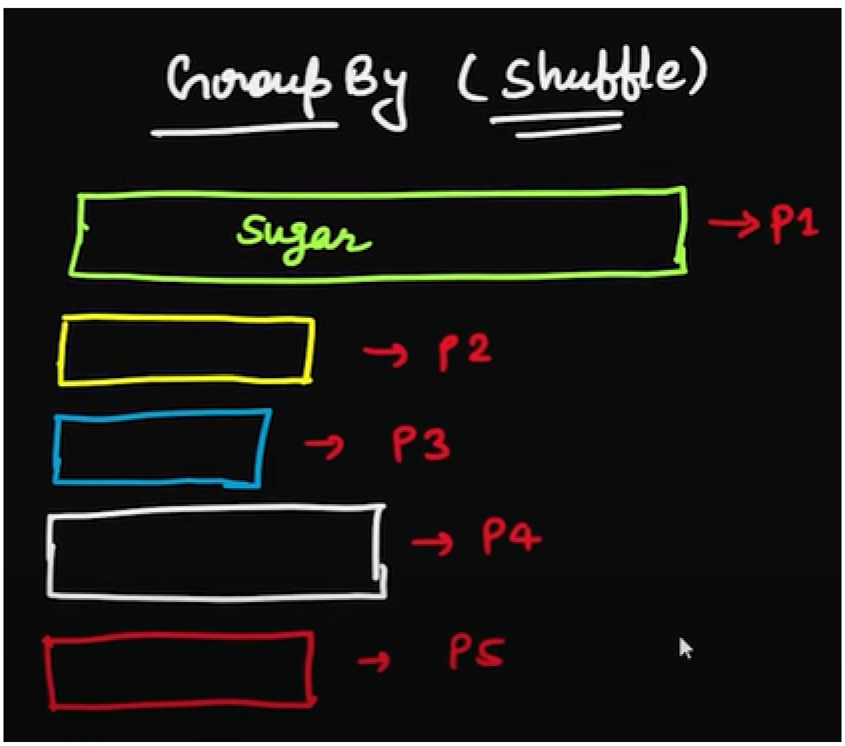
```
skewed_fact_tbl_df.show()

(1) Spark Jobs

+-----------+--------+------------+----------+---------------+-----+--------+----------
|customer_id|store_id|product_name|sales_date|sales_person_id|price|quantity|total_cost|
+-----------+--------+------------+----------+---------------+-----+--------+----------
|         12|     123|       sugar|2023-03-03|              8| 50.0|       3|     150.0|
|          6|     121|       sugar|2023-07-22|              1| 50.0|       3|     150.0|
|          8|     122|       sugar|2023-07-08|              5| 50.0|       4|     200.0|
|          4|     123|       sugar|2023-08-13|              8| 50.0|       1|      50.0|
|          5|     122|       sugar|2023-06-18|              5| 50.0|       1|      50.0|
|         20|     121|       sugar|2023-05-07|              2| 50.0|       8|     400.0|
|          1|     122|       sugar|2023-08-20|              6| 50.0|       6|     300.0|
|          3|     122|       sugar|2023-0[illegible]2|              5| 50.0|       9|     450.0|
|          2|     121|       sugar|2023-06-18|              2| 50.0|       8|     400.0|
|         11|     123|       sugar|2023-07-11|              8| 50.0|       1|      50.0|
|          7|     121|       sugar|2023-04-26|              1| 50.0|       4|     200.0|
|         20|     122|       sugar|2023-06-25|              6| 50.0|       4|     200.0|
|         11|     122|       sugar|2023-07-30|              5| 50.0|       2|     100.0|
|         13|     121|       sugar|2023-05-01|              3| 50.0|       3|     150.0|
|         20|     123|       sugar|2023-06-29|              9| 50.0|       7|     350.0|
|         18|     123|       sugar|2023-05-08|              7| 50.0|       8|     400.0|
|          9|     121|       sugar|2023-07-09|              3| 50.0|       8|     400.0|
|          1|     122|       sugar|2023-04-24|              5| 50.0|       1|      50.0|
```

10. Here product_dimension table is used for joins and to find the old price of the product
11. Now let's suppose we have this data in 2 partitions

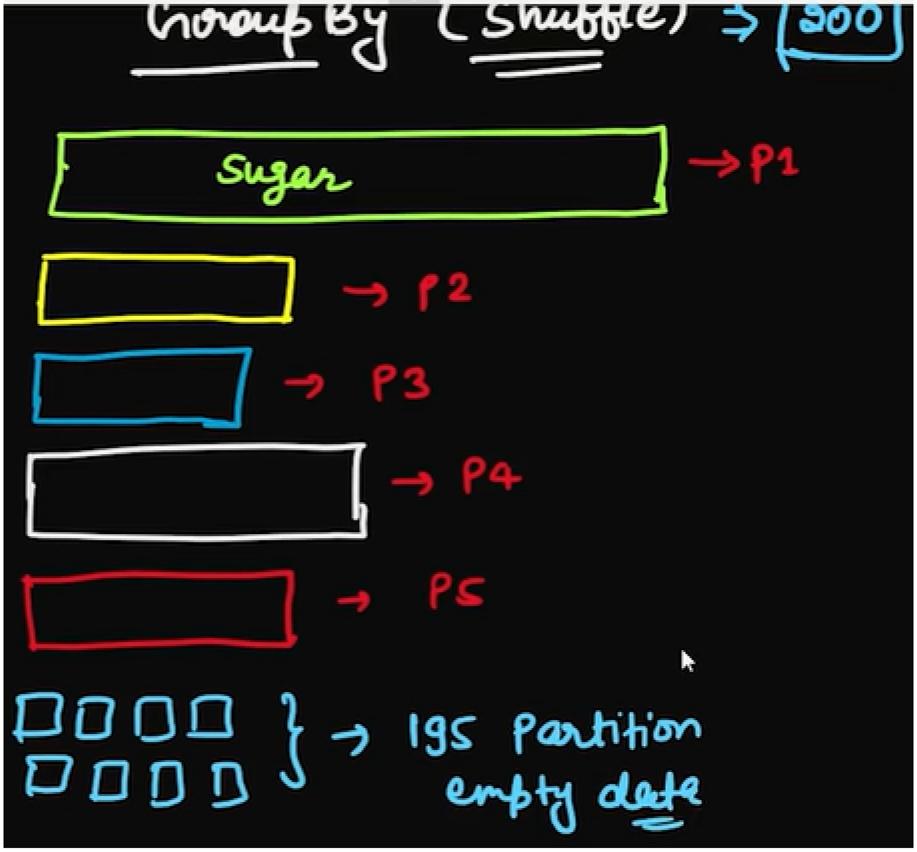
12. Each partition can store 128MB of data



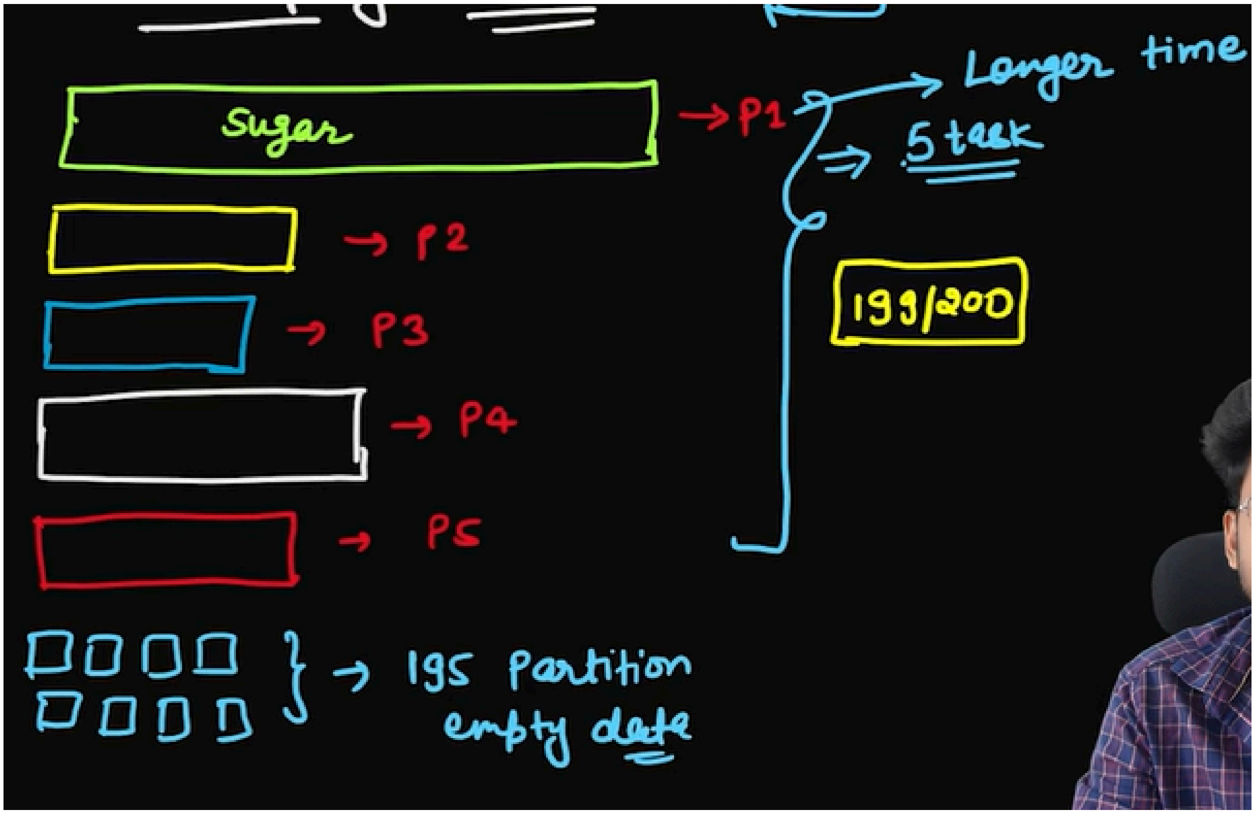

13. Now if we perform a group by on for getting total sales of a product)
14. Here if we apply a group by…then shuffling takes place
15. Which moves the data between partitions to perform group by easily
16. After the shuffling our data will look like this



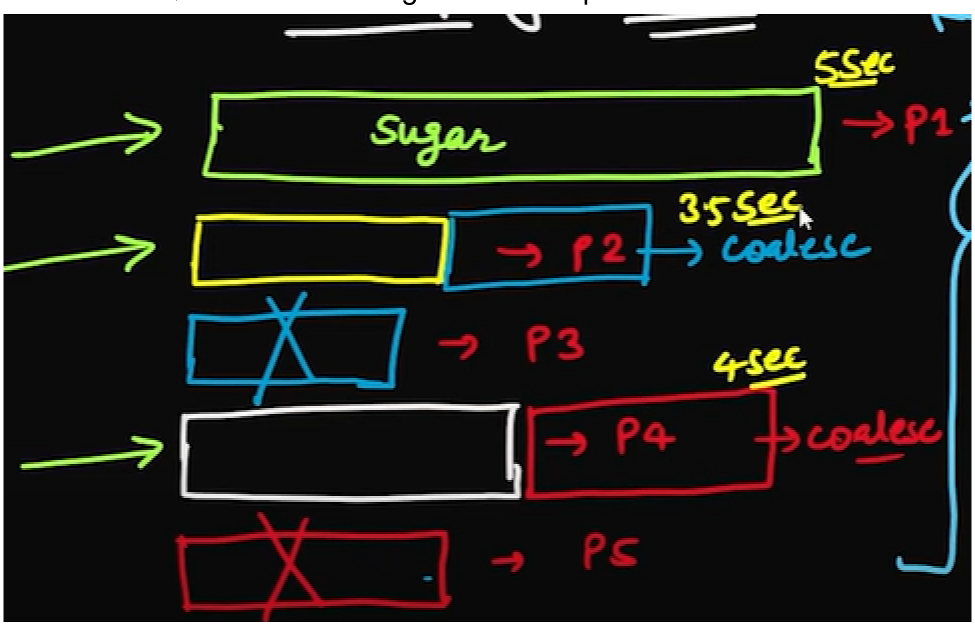
17. Here if we perform a shuffle..then by default we'll get 200 partitions




18. Now even if we have empty partitions..our spark will scan these entire partitions..even though it might finish in milliseconds…but our resources are getting wasted here
19. Now to process these partitions we need 5 tasks..one for each partition
20. Here partition 1 takes longer time…as it has large amount of data

21. 

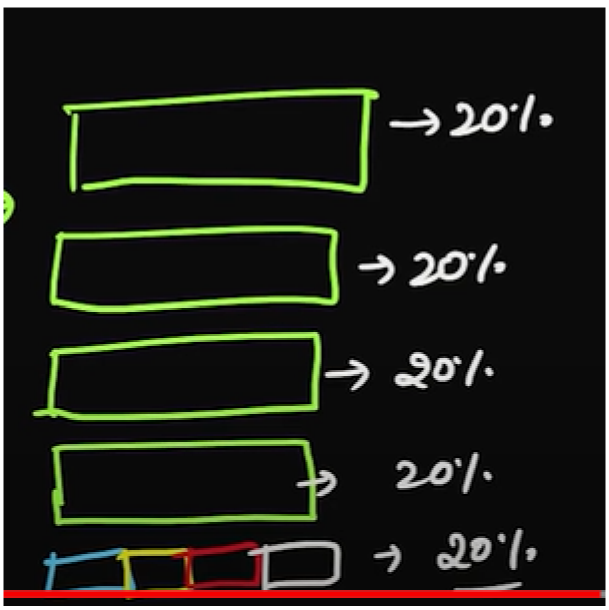

22. As we can see 199/200 tasks have been completed…but partition 1 takes longer time
23. Now here AQE comes into picture
24. Now what AQE does is…it merges the small partitions



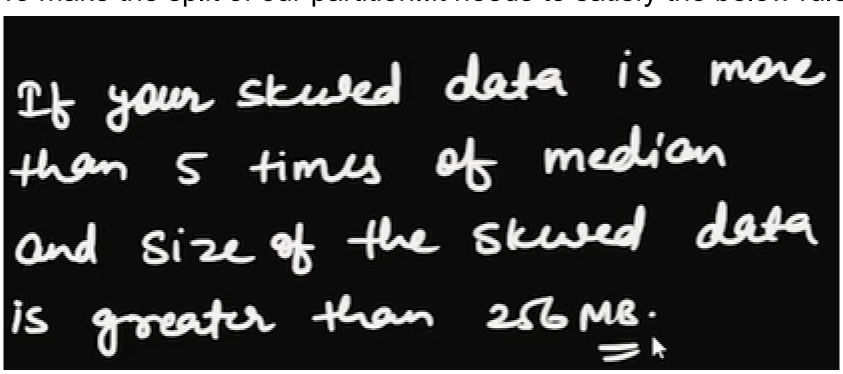

25. Here it merged p2,p3 and p4,p5…so eventually …tasks got reduced to 3…which frees cpu cores too

26. Another AQE scenario
27. Now we may have a doubt..as partition 1 has 80% of data..and p2,p3,p4,p5 has only 20% of data


28. Here we still can face the 199/200 issue(200th task needs more time)
29. So here we face data skewness
30. So here we need to split the data..such that every partition has even storage of data



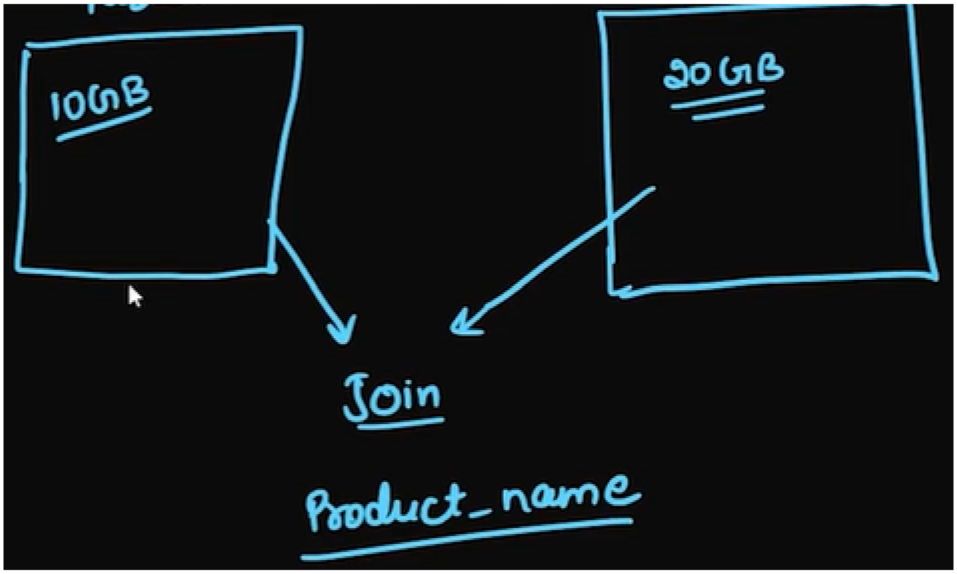

31. To make the split of our partition..it needs to satisfy the below rule




32. This is how it dynamically coalescing shuffle operation. Gemini explained : https://g.co/gemini/share/1d22256c4b26
33. Lets see how our code with AQE looks in sparkUI

34. dynamically switching join strategy

35. Lets consider we have 2 tables as shown below..and we join those tables on a condition

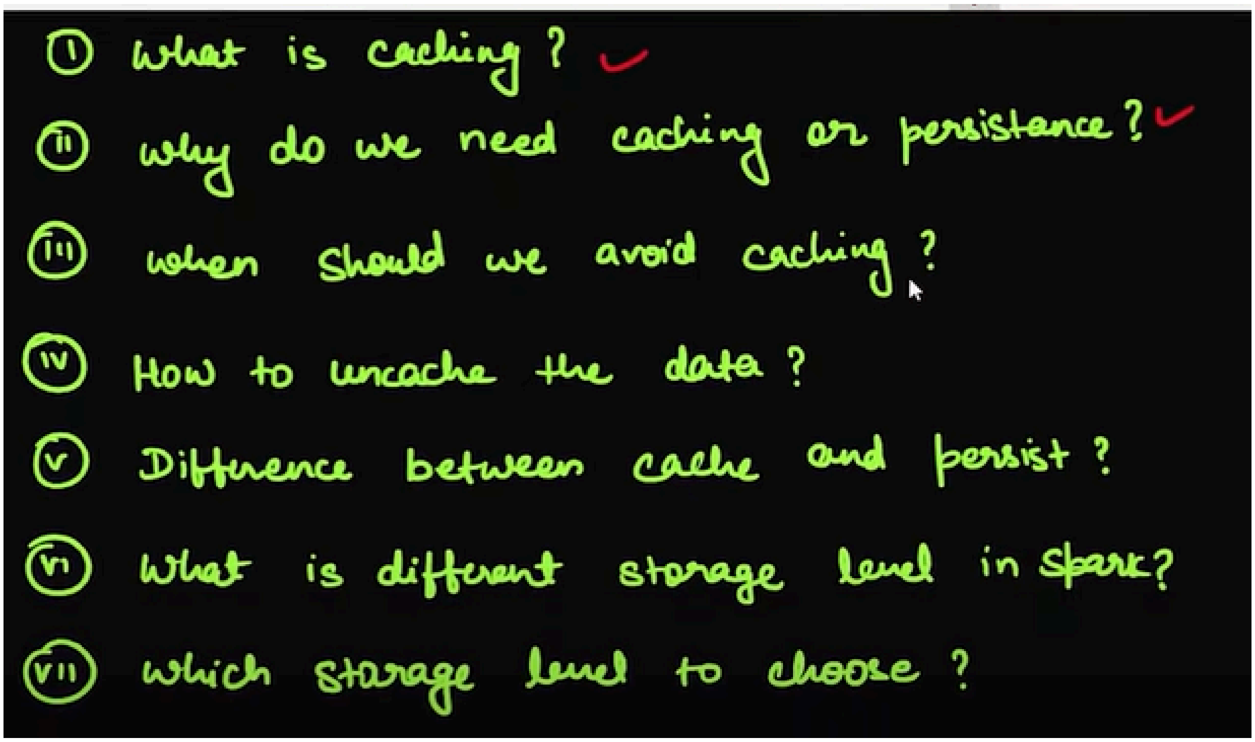



36. So now based on the tables sizes..it automatically joins using sort-merge join
37. But assume that we have made some transformations on data..and now the tables size have been reduced

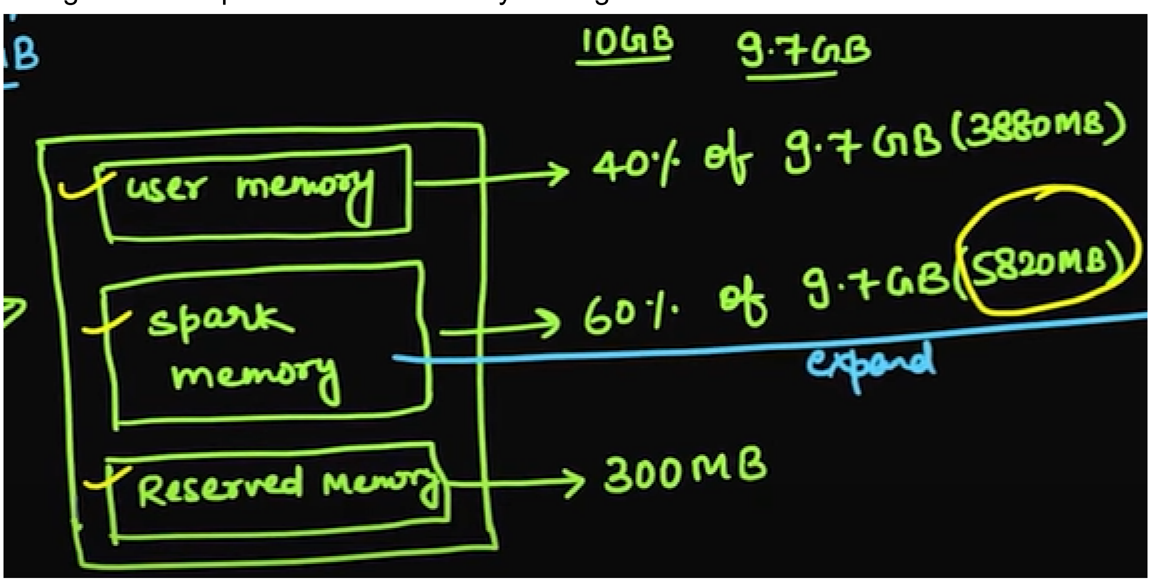

38. Now if the AQE is enabled..then it read the table size from the runtime statistics…then as the table2 size is very small…it changes the join to broadcast join
39. Explained : [https://g.co/gemini/share/379180883b2e](https://g.co/gemini/share/379180883b2e)
40. Dynamically optimizing skew join(refer online)

Cache and Persist

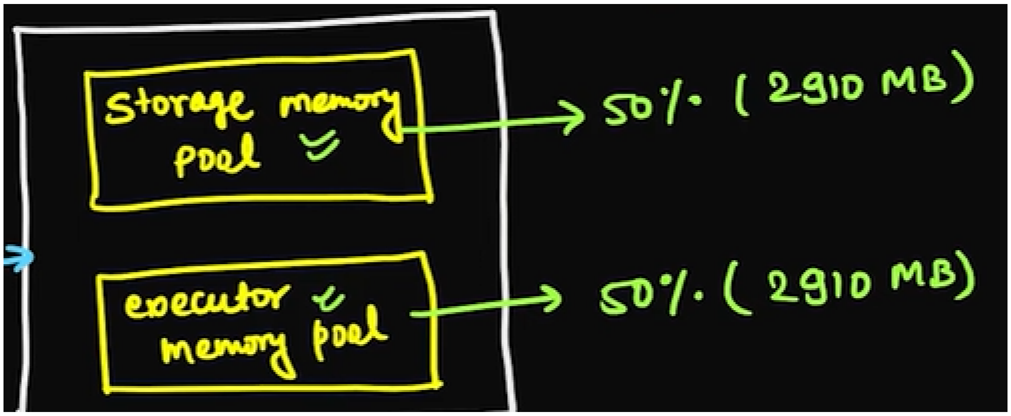
1. Potential interview questions



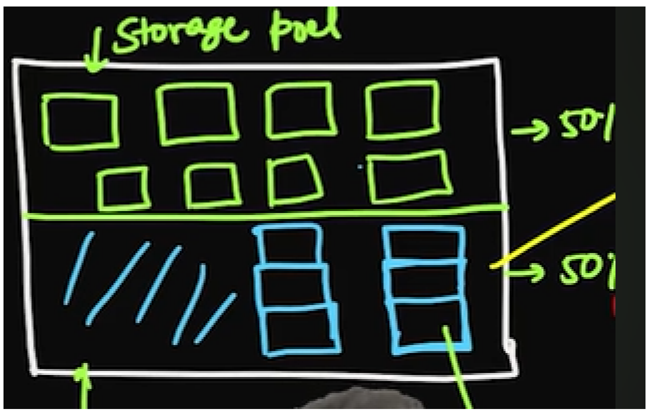

2. Caching is an optimizing technique..which stores the intermediate result.
3. Lets go back to spark executor memory management



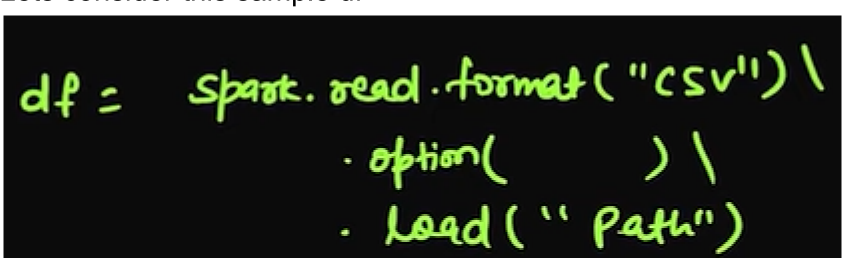
4. Inside spark memory we'll have



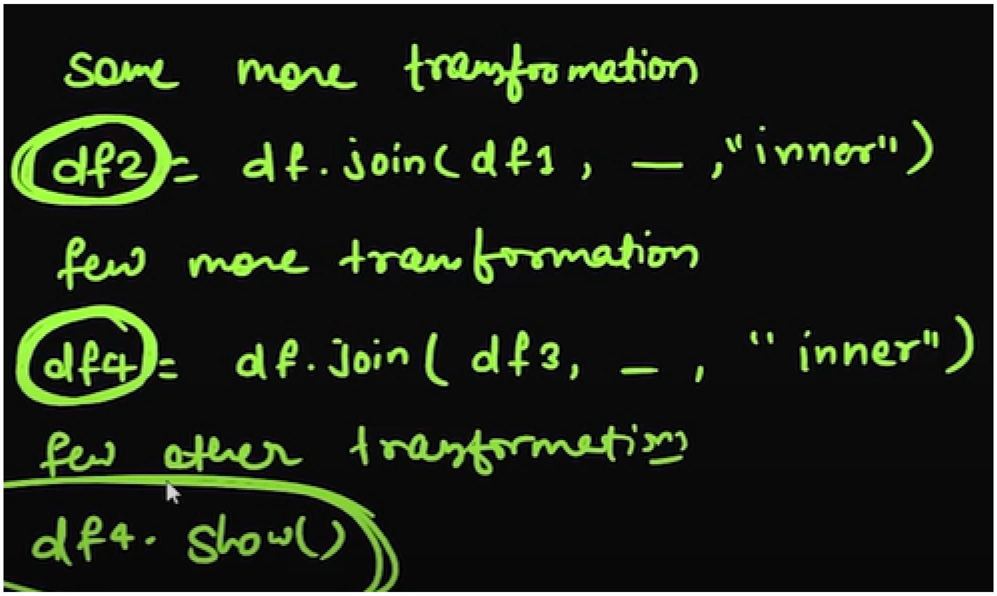

5. Now data caching happens at the storage pool



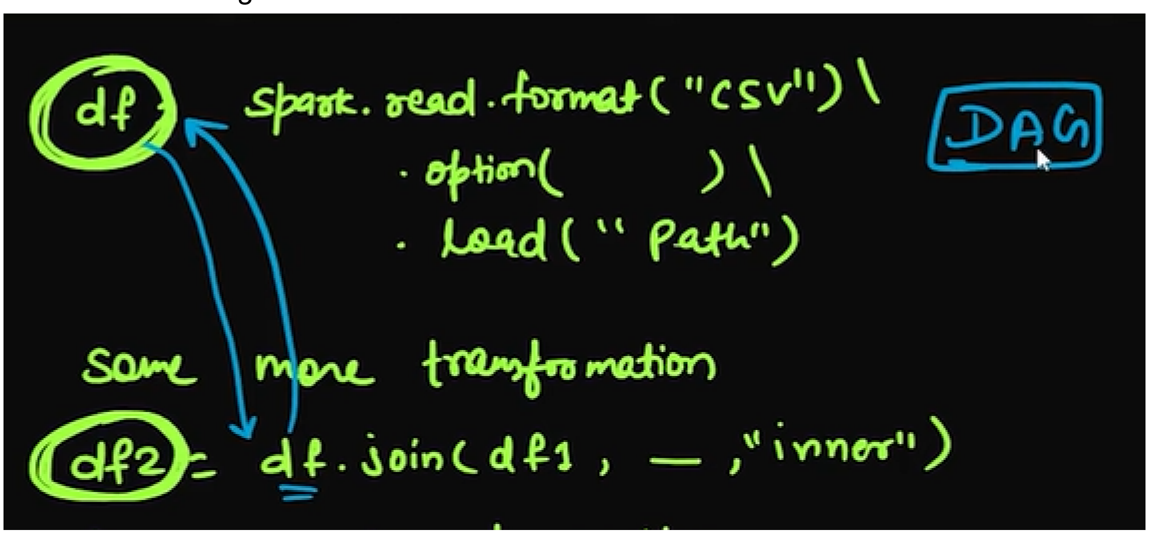

6. Now we know where the caching is happening
7. Actually caching is an optimization technique..which stores an intermediate result
8. What is an intermediate result?
9. Lets consider this sample df



10. So lets assume that we have some transformations and using joins on first_df

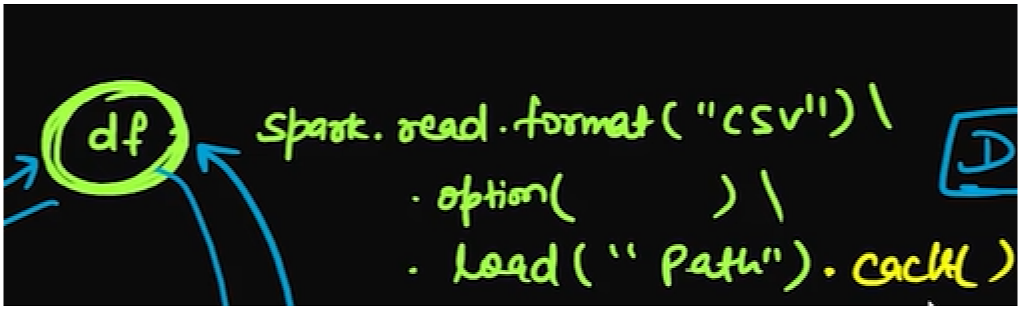



11. As we can see here…we have created few other dataframe and atlast we have hit action using df4.show()
12. Now here after the df2 has created…df1 will be wiped off…
13. Similarly after df4 has created…df2 will be wiped off and we made action using df4.show()..so it gives us df4
14. Now if we want to get df2

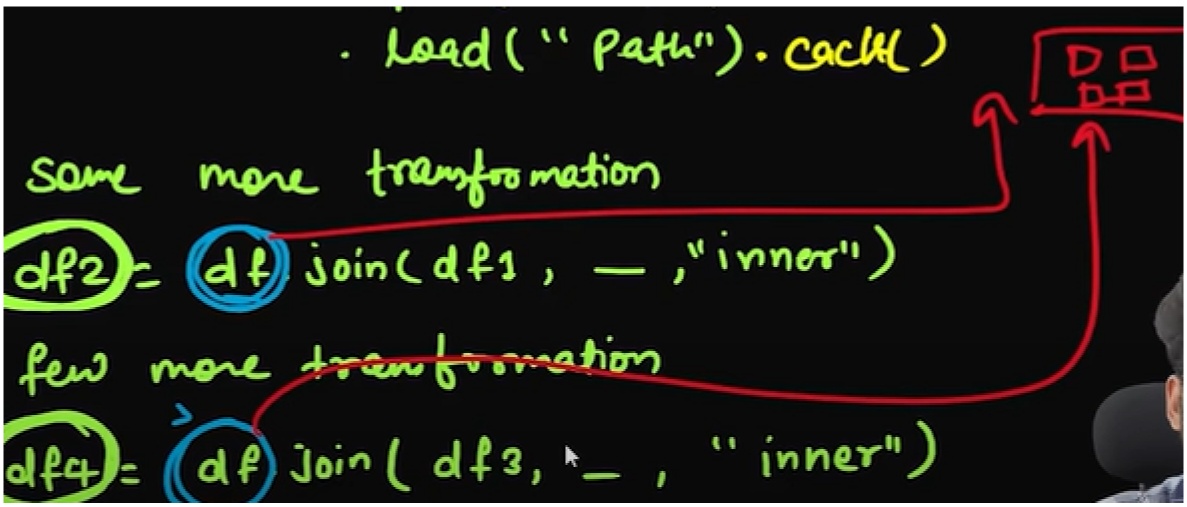



15. Using DAG it goes to df1 and gets the data and finishes its creation…but then again…it takes time(wastage of time) for recalculating using DAG

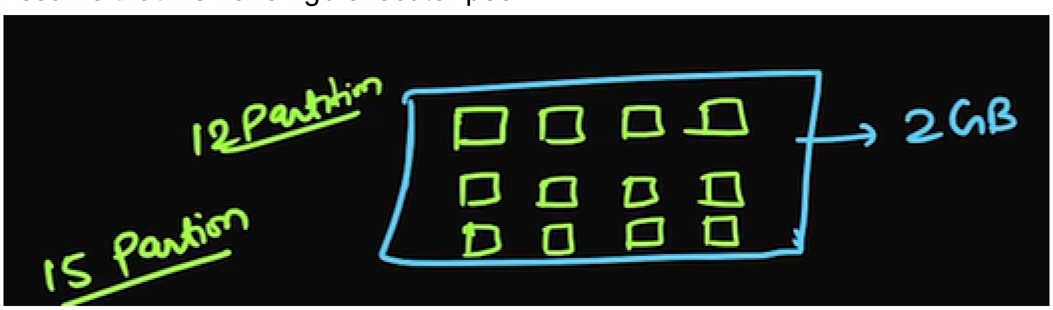
16. So here if we use .cache() while creating the df1




17. Then instead of storing the data in short lived..it stores the data spark memory executor pool. Now until our application is closed..we'll be having this data
18. There's a catch here…it removes the data on LRU basis
19. So now for df2 and df4..the df data will be coming from cache()

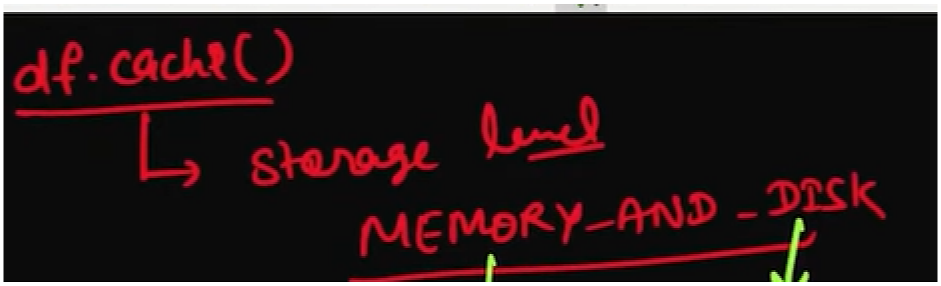



20. Assume that we have 2gb executor pool




21. Now what if our df size is above 2gb? ..then our executor pool will store until 2GB and leaves the rest of the data in short lived zone
22. Storage level concept

23. So when we use df.cache() if our executor pool is out of space…then the remaining partitions will go to disk




24. But R/W(I/O) from disks take more time than DAG
25. Now what if our partition is lost while doing read/write?
26. Then using DAG..we can recalculate our lost partitions
27. Persists
28. Persists give more flexibility than cache
29. When using persists..we have to specify the storage level

30.

In Apache Spark, both `cache` and `persist` serve the purpose of optimizing data processing by storing intermediate results for faster access. However, there's a subtle difference between them

**cache**

- **Shorthand:** `cache` is essentially a shorthand for `persist` with the default storage level, which is `MEMORY_ONLY`.
- **Storage Location:** When you use `cache`, the intermediate data is stored entirely in memory (assuming sufficient memory is available). This provides the fastest access but is limited by the amount of RAM on your Spark cluster nodes.

**persist**

- **Flexibility:** `persist` offers more control over where the data is cached. You can specify a `StorageLevel` object as an argument, allowing you to choose from various options like:
  - `MEMORY_ONLY` (same as `cache`): Stores data only in memory.
  - `MEMORY_AND_DISK`: Stores data in memory (if space permits) and spills to disk when necessary. This balances speed and capacity.
  - `DISK_ONLY`: Stores data only on disk (slower access, but suitable for large datasets that don't fit in memory).
  - Additional storage levels like `MEMORY_ONLY_SER` (serialized in memory) and `MEMORY_AND_DISK_SER` (serialized in memory and on disk) are also available for memory optimization.

```
# Create a DataFrame
data = spark.read.text("your_text_file.txt")
```

Use code with caution.

Here's how `cache` and `persist` differ in this case:

- `data.cache()` **:** This will store the entire DataFrame ( `data` ) in memory (assuming enough RAM). This is suitable for small to medium-sized datasets that you expect to use again soon.
- `data.persist(F.StorageLevel.MEMORY_AND_DISK)` **:** This will store `data` in memory if possible, but if memory becomes full, Spark will automatically spill it to disk. This is a good option for larger datasets that might not fit entirely in memory but for which you want faster access than relying solely on disk reads.

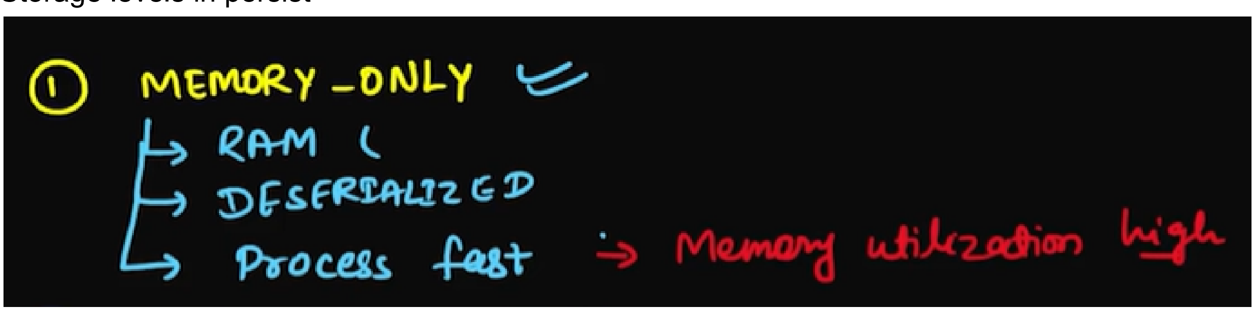
In essence, `cache` is a convenient way to use the default `MEMORY_ONLY` storage level, while `persist` provides more fine-grained control over how and where data is cached. When memory is abundant, both methods achieve the same result. However, with larger datasets or limited memory, using `persist` with an appropriate storage level like `MEMORY_AND_DISK` allows for more efficient data handling.

31. Lets see this practically

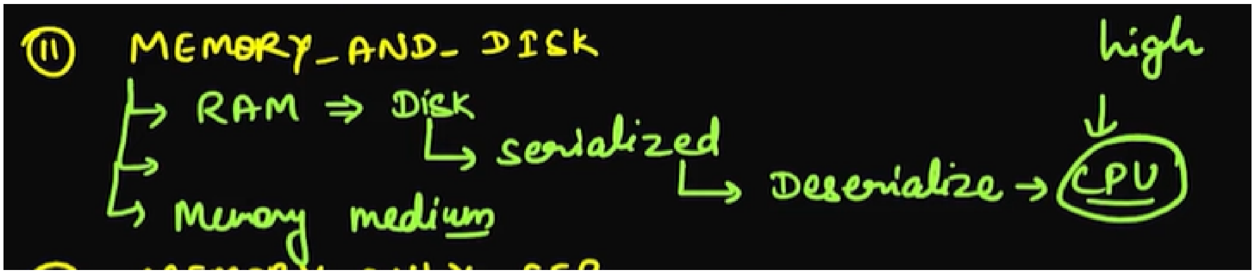
```
fact_df = spark.read.format("csv")\
    .option( key: "header", value: "true")\
    .option( key: "inferschema", value: "true")\
    .load("C:\\Users\\nikita\\Documents\\data_engineering\\spark_data\\skewed_data.csv")


fact_df.cache()
# fact_df.persist(StorageLevel.DISK_ONLY)
```

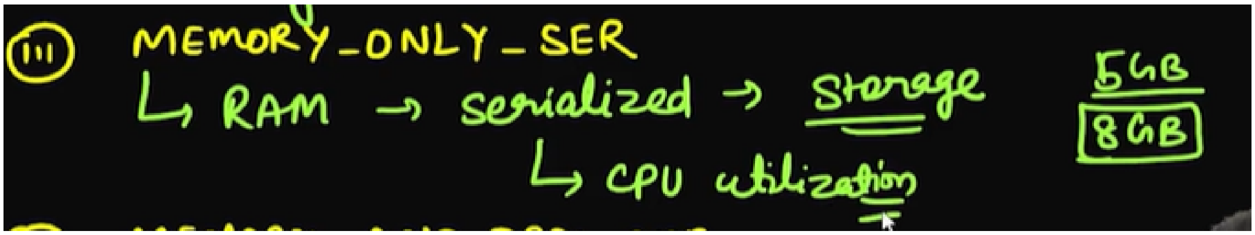
32. 
```
fact_df.show()
```

33. Please see this practical online
34. Storage levels in persist

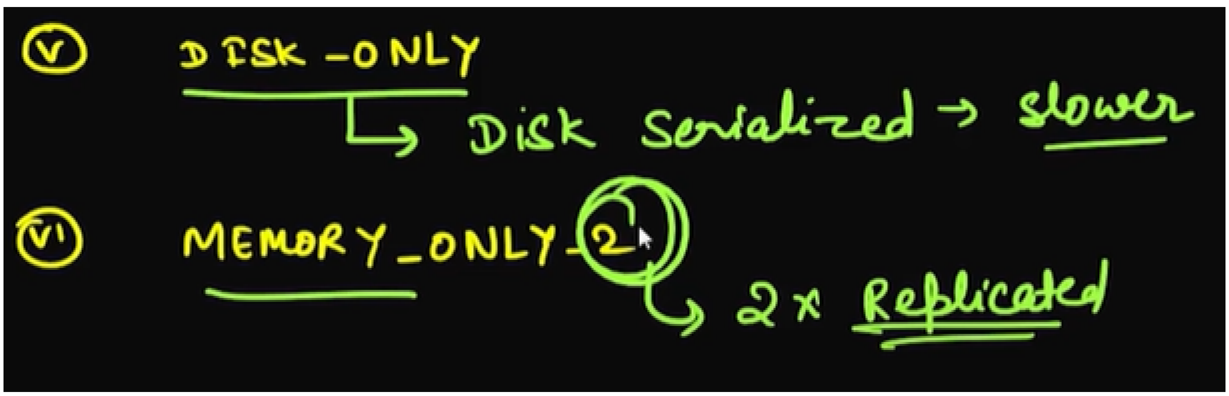



35. In memory only..the memory utilization will be very high..so if there are many shuffles and joins…its better to use memory and disk(storage level)

36. 


37. 


38. Using serialization(compresses data) we can get some extra space…but to read the data ..we need to deserialize it again ..which consumes the CPU

39. 


40. How to uncache data? Call unpersist()