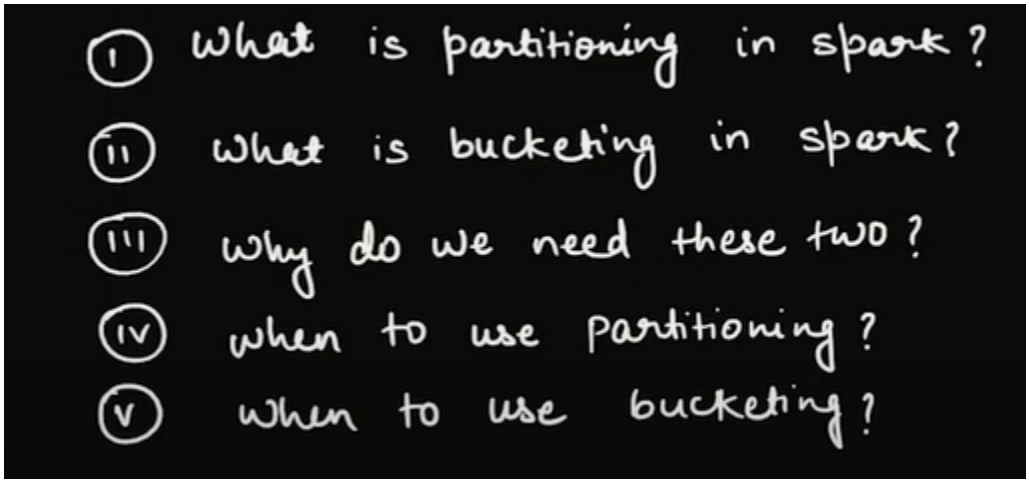
Partitioning and Bucketing

1. Potential interview question



2. Lets consider this sample data

Id	Name	age	salary
1	manish	15	50,000
8	vikash	25	60,000
2	Raushan	18	20,000
3	Rakesh	72	200000
7	Mukesh	14	14000
4	Sohan	35	35000
5	mohan	23	45000
6	Rani	28	37000

3. partitioning:

Partitioning:

- **Concept:** Partitioning divides data into smaller subsets based on the value of a specific column. This allows Spark to efficiently access relevant data during operations like filtering or joining.
- **Example:** Imagine a DataFrame containing website log data with a 'date' column. Partitioning by 'date' would create separate partitions for each unique date. When querying data for a specific date range, Spark only needs to access the corresponding partitions, speeding up the process.

4. Bucketing
    - **Concept:** Bucketing is similar to partitioning, but instead of using column values, it uses a hash function on a specified column to distribute data into a fixed number of buckets. This is useful for operations like joins where both tables are bucketed on the same column.
    - **Example:** Consider a DataFrame containing customer data with a 'customer_id' column. Bucketing this data by 'customer_id' with 10 buckets would distribute customer records across 10 buckets based on a hash of their ID. Now, joining this data with another DataFrame containing purchase information (also bucketed by 'customer_id') can be done efficiently by shuffling data only within each bucket, significantly reducing overall shuffle overhead.
5. Here if we have a look at our sample data..then we can say that their are no columns to perform partition on..hence we go to bucketing in this situation
6. Practical
7. Here we have used partition on address

```
#partition by address(country)
df.write.format("csv")\
    .option("header","true")\
    .option("mode","overwrite")\
    .option("path","/FileStore/tables/partition_by_address/")\
    .partitionBy("address")\
    .save()
```

8. Now it creates separate file for each address present on the file
9. So if in future ..if we need data where a person is in INDIA..then spark can just read a single partition named INDIA and it does not need to read entire data
10. It is one of the optimization technique.
11. If we give partition on two columns

```
df.write.format("csv")\
        .option("header","true")\
        .option("mode","overwrite")\
        .option("path","/FileStore/tables/patition_by_address_gender/")\
        .partitionBy("address","gender")\
        .save()
```

12. Then first it will create partition on address and in the address data it will create partitions on gender

13. Here it created partitions on address first

```
1  dbutils.fs.ls("/FileStore/tables/patition_by_address_gender/",)
```

```
Out[22]: [FileInfo(path='dbfs:/FileStore/tables/patition_by_address_gender/_SUCCES
dificationTime=1683865393000),
 FileInfo(path='dbfs:/FileStore/tables/patition_by_address_gender/address=INDIA/',
modificationTime=0),
 FileInfo(path='dbfs:/FileStore/tables/patition_by_address_gender/address=JAPAN/',
modificationTime=0),
 FileInfo(path='dbfs:/FileStore/tables/patition_by_address_gender/address=RUSSIA/'
0, modificationTime=0),
 FileInfo(path='dbfs:/FileStore/tables/patition_by_address_gender/address=USA/', m
ificationTime=0)]
```

14. Inside each address it partitioned on gender

```
1  dbutils.fs.ls("dbfs:/FileStore/tables/patition_by_address_gender/address=INDIA/")
```

```
Out[23]: [FileInfo(path='dbfs:/FileStore/tables/patition_by_address_gender/address=INDIA/gen
=f/', size=0, modificationTime=0),
 FileInfo(path='dbfs:/FileStore/tables/patition_by_address_gender/address=INDIA/gender=m/',
e=0, modificationTime=0)]
```

15. Where does the partition fails?
16. We cannot perform partition on a column which has unique values(like an ID column)
17. Bucketing
18.

```
#Bucketing
df.write.format("csv")\
    .option("header","true")\
    .option("mode","overwrite")\
    .option("path","/FileStore/tables/bucket_by_id/")\
    .bucketBy(3,"id")\
    .saveAsTable("bucket_by_id_table")
```

19. Here we have created 3 buckets on id
20. And while writing a bucket..we must use saveAsTable instead of save()
21. Should learn bucketing more(didn't understand properly)

How to create a dataframe in spark

1. Usually a DE pipelines involved reading data-tranformation-write-BI/ML/DA

2. Here the main part is transfornation
3. So we can apply transformation in 2 ways …using dataframe API and SparkSQL
4. We will concentrate more in dataframe API and will learn how to create df api
5. Practical
6. First we will create a sample data..using tuples inside a list

```
data= [( 1,   1),
( 2,   1),
( 3,   1),
( 4,   2),
( 5,   1),
( 6,   2),
( 7,   2)]
```

7. Next we define schema for our sample data

```
schema= [ 'id',  'num' ]
```

8. Now we will create a df using the sample data and its schema

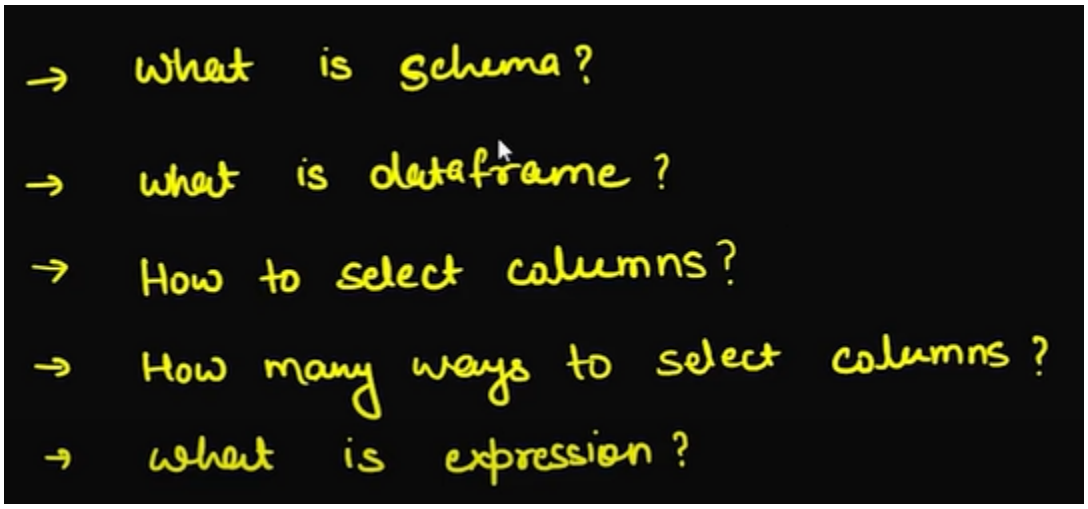
```
spark.createDataFrame(data=my_data,schema=my_schema).show()
```

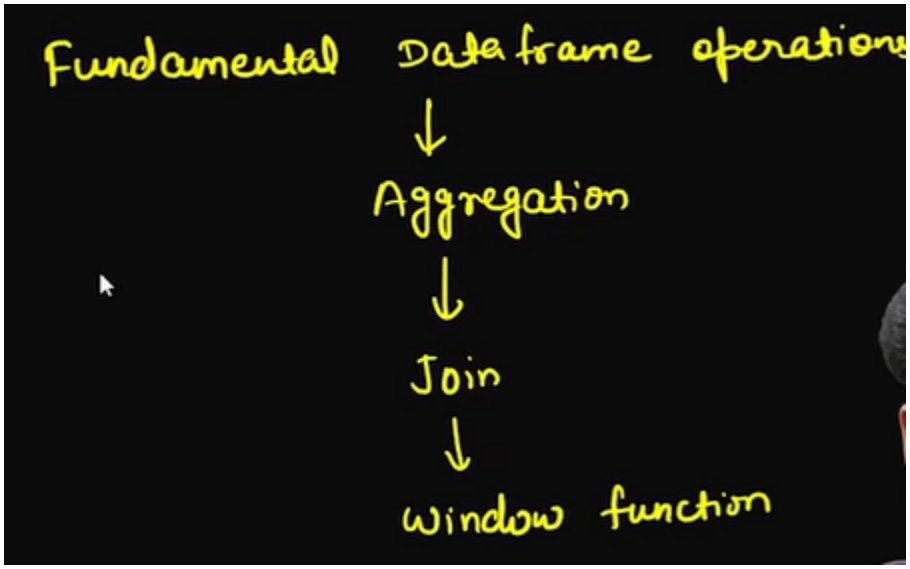
```
1  spark.createDataFrame(data=my_data,schema=my_schema).show()
```

```
▸ (3) Spark Jobs
+---+---+
| id|num|
+---+---+
|  1|  1|
|  2|  1|
|  3|  1|
|  4|  2|
|  5|  1|
|  6|  2|
|  7|  2|
+---+---+
```

9.

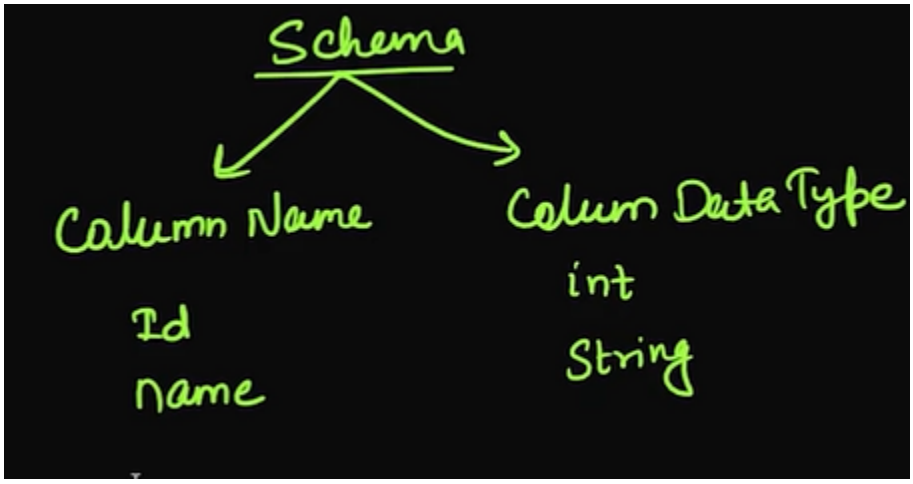
DataFrame transformation part1

1. 

2. We'll deal with fundamental df operation which is


3. We'll also learn sparkSQL too
4. So what is schema?

5. Schema is made up of column names and column data types



6. So printSchema gives us the schema of current_table

```
employee_df.printSchema()
```

```
root
 |-- id: integer (nullable = true)
 |-- name: string (nullable = true)
 |-- age: integer (nullable = true)
 |-- salary: integer (nullable = true)
 |-- address: string (nullable = true)
 |-- nominee: string (nullable = true)
```

7. To access the columns of the table we use

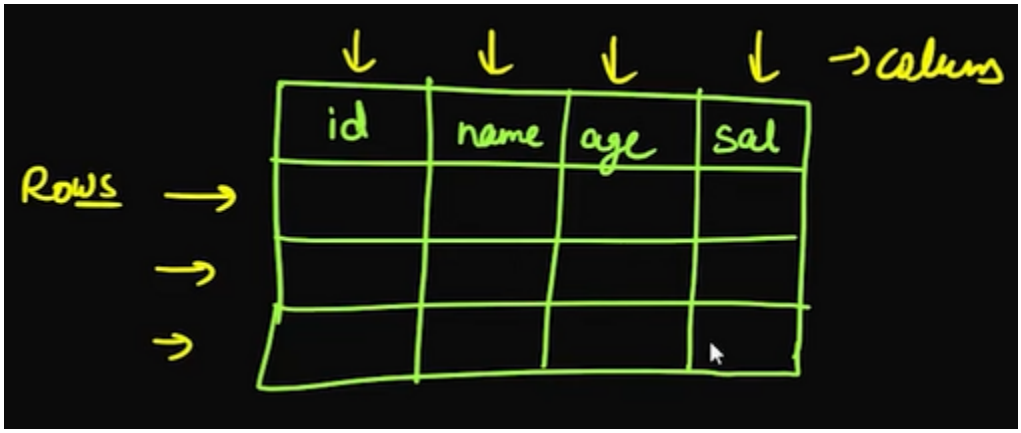
```
Cmd 36

employee_df.columns

Out[3]: ['id', 'name', 'age', 'salary', 'address', 'nominee']

Command took 0.22 seconds -- by manisnitt@gmail.com at 6/14/2023, 8:27:08 AM on My Cluster
```

8. To create a manual schema instead of inferschema we use

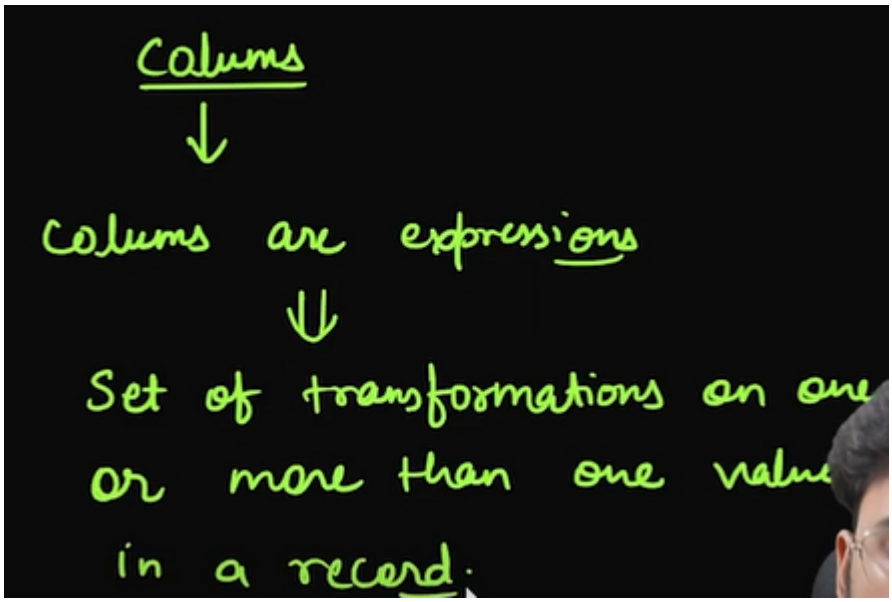
```
emp_schema= StructType(
                        [
                            StructField("id",IntegerType(),True),
                            StructField("name",StringType(),True),
                            StructField("age",IntegerType(),True),
                            StructField("salary",IntegerType(),True),
                            StructField("address",StringType(),True),
                            StructField("nominee",StringType(),True),
                            StructField("_corrupt_record", StringType(), True)
                        ])
```

9. So what is a df?

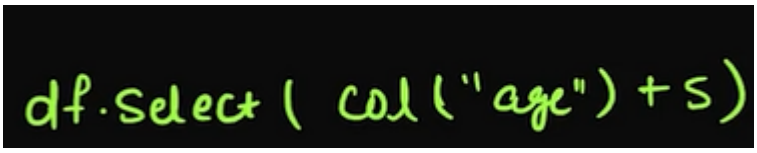
10. A df is made up of rows and columns



11. Here df is stored in a row format
12. Coming to columns



example



13. Selecting columns in spark `employee_df.select("name").show()` using string method
14. Using the string method..we cannot perform any operations on columns
15. So we use col method

16. `employee_df.select(col("id") +5 ).show()` to perform operations in column

```
1  employee_df.select(col("id") +5 ).show()
```

▸ (1) Spark Jobs

```
+--------+
|(id + 5)|
+--------+
|       6|
|       7|
|       8|
|       9|
|      10|
+--------+
```

17. To select multiple columns…using string method

```
1  employee_df.select("id","name","age").show()
```

▸ (1) Spark Jobs

```
+---+--------+---+
| id|    name|age|
+---+--------+---+
|  1|  Manish| 26|
|  2|  Nikita| 23|
|  3|  Pritam| 22|
|  4|Prantosh| 17|
|  5|  Vikash| 31|
+---+--------+---+
```

18. Selecting multiple columns using col method

```
1  employee_df.select(col("id"),col("name"),col("age")).show()
```

▸ (1) Spark Jobs

```
+---+--------+---+
| id|    name|age|
+---+--------+---+
|  1|  Manish| 26|
|  2|  Nikita| 23|
|  3|  Pritam| 22|
|  4|Prantosh| 17|
|  5|  Vikash| 31|
+---+--------+---+
```

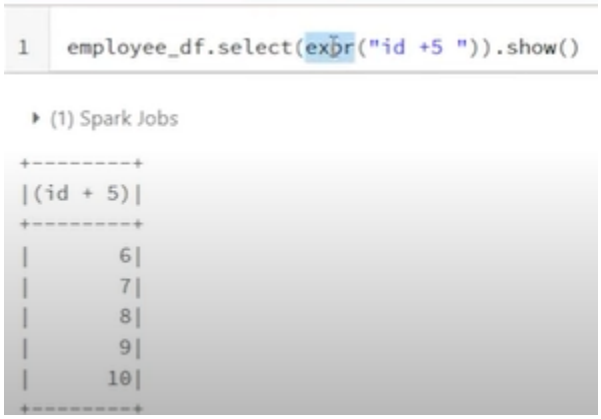
19. Here we have used multiple methods to access the

20. Here employee_df["salary"] …this type of accessing cols is used while using the join
21. Expression

22. **Expressions in PySpark DataFrames**

    Expressions are powerful tools used to manipulate, transform, and analyze data within DataFrames. They can be constructed using various components:

    - **Column References:** These directly refer to existing columns within the DataFrame.
    - **Operators:** PySpark supports a rich set of operators for arithmetic calculations, comparisons, string manipulations, and more (e.g., `+`, `-`, `*`, `/`, `==`, `!=`, `LIKE`, `CONCAT`).
    - **Functions:** PySpark offers a wide range of built-in functions for data processing and analysis (e.g., `abs`, `sqrt`, `lower`, `upper`, `count`, `avg`, `sum`).
    - **SQL Expressions:** The `expr()` function allows you to write SQL-like expressions directly within PySpark (useful for functions or operations not readily available as built-in functions).

23. 

24. We can also use expr for aliasing and concatening columns

```
1  employee_df.select(expr("id as employee_id "),expr("name as
   emplyee_name"),expr("concat(name,address)")).show()
```

▸ (1) Spark Jobs

```
+-----------+------------+--------------------+
|employee_id|emplyee_name|concat(name, address)|
+-----------+------------+--------------------+
|          1|       Manish|          Manishbihar|
|          2|       Nikita|   Nikitauttarpradesh|
|          3|       Pritam|       PritamBangalore|
|          4|     Prantosh|       PrantoshKolkata|
|          5|       Vikash|                null|
+-----------+------------+--------------------+
```

25. Spark SQL
26. It is same as SQL
27. Here we have created a temp table and performed sql operation

Cmd 52

```
1  employee_df.createOrReplaceTempView("emplooyee_tbl"
```

Command took 0.17 seconds -- by manisnitt@gmail.com at 6/14/2023,

Cmd 53

```
1  spark.sql("""
2
3  select * from emplooyee_tbl
4
5  """)
```