Spark Interview Guide : Miscellaneous Important Concepts

Created By: Sourav Banerjee
https://www.linkedin.com/in/sourav-banerjee-50b443106/

Part 1

Question 1: Difference between map and flatMap transformations in Spark (pySpark)

map: It returns a new RDD by applying a function to each element of the RDD. Function in the map can return only one item.

flatmap: Similar to map, it returns a new RDD by applying a function to each element of the RDD, but the output is flattened.

Also, function in flatMap can return a list of elements (0 or more)

Example1:-

sc.parallelize([3,4,5]).map(lambda x: range(1,x)).collect()

Output:

[[1, 2], [1, 2, 3], [1, 2, 3, 4]]

sc.parallelize([3,4,5]).flatMap(lambda x: range(1,x)).collect()

Output: notice o/p is flattened out in a single list

[1, 2, 1, 2, 3, 1, 2, 3, 4]

Example 2:

sc.parallelize([3,4,5]).map(lambda x: [x, x*x]).collect()

Output:

[[3, 9], [4, 16], [5, 25]]

sc.parallelize([3,4,5]).flatMap(lambda x: [x, x*x]).collect()

Output: notice flattened list

[3, 9, 4, 16, 5, 25]

Example 3:

There is a file greetings.txt in HDFS with the following lines:

Good Morning

Good Evening

Good Day

Happy Birthday

Happy New Year

lines = sc.textFile("greetings.txt")

lines.map

(lambda line: line.split()).collect()

Output:-

[['Good', 'Morning'], ['Good', 'Evening'], ['Good', 'Day'], ['Happy', 'Birthday'], ['Happy', 'New', 'Year']]

lines.flatMap(lambda line: line.split()).collect()

Output:-

['Good', 'Morning', 'Good', 'Evening', 'Good', 'Day', 'Happy', 'Birthday', 'Happy', 'New', 'Year']

We can do wordcount of file using flatMap:-

lines = sc.textFile("greetings.txt")

sorted(lines.flatMap(lambda line: line.split()).map(lambda w: (w,1)).reduceByKey(lambda v1, v2: v1+v2).collect())

Output:-

[('Birthday', 1), ('Day', 1), ('Evening', 1), ('Good', 3), ('Happy', 2), ('Morning', 1), ('New', 1), ('Year', 1)]

Question 2: Difference Between Where & Filter in Spark SQL.

Spark filter() or where() function is used to filter the rows from DataFrame or Dataset based on the given one or multiple conditions or SQL expression. You can use the where() operator instead of the filter if you are coming from an SQL background. Both these functions operate exactly the same.

Let’s create a DataFrame and view the contents:

```
val df = Seq(("famous amos", true),("oreo", true),("ginger snaps", false) ).toDF("cookie_type", "contains_chocolate")
```

df.show

()

7. +------------+------------------+

```
 | cookie_type|contains_chocolate|
+------------+------------------+
 | famous amos| true|
 | oreo| true|
 |ginger snaps| false|
 +------------+------------------+
```

Now let’s filter the DataFrame to only include the rows with contains_chocolate equal to true.

```
val filteredDF = df.where(col("contains_chocolate") === lit(true))

filteredDF.show
()
+-----------+------------------+
|cookie_type|contains_chocolate|
+-----------+------------------+
|famous amos| true|
| oreo| true|
+-----------+------------------+
```

There are various alternate syntaxes that give you the same result and the same performance.

```
df.where("contains_chocolate = true")
df.where($"contains_chocolate" === true)
df.where('contains_chocolate === true)
```

that all of these syntaxes generate the same execution plan, so they’ll all perform equally.

where is an alias for the filter, so all these work as well.

```
df.filter(col("contains_chocolate") === lit(true)
df.filter("contains_chocolate = true")
df.filter($"contains_chocolate" === true)
df.filter('contains_chocolate === true)
```

Question 3: Difference Between map() vs mapPartitions() in SparkSQL.

map()

map function is a transformation function that gets applied on every element of the RDD and returns a new RDD with transformed elements.

In our sample RDD, the map function will be called on each element in the RDD. So when we apply the map function in our sample RDD it will be called 50 times.

mapPartitions()

mapPartitions is a transformation function and gets applied once per partition in the RDD. In our sample RDD, mapPartitions will be called once per partition so it will be called 5 times because we have 5 partitions in our RDD.

When to use map() and when to use mapPartitions()?

Use mapPartitions function when you need to perform heavy initialization before you transform the elements in the RDD.

Let’s say you need a database connection to transformation elements in your RDD. It doesn’t make sense to initialize the database connection for every element in RDD. If we do that, we will end up initializing the database connection 50 times. Which is not ideal.

Ideally, we want to initialize database connection once per partition/task. So mapPartitions() is the right place to do database initialization as mapPartitions is applied once per partition.

Here is a code snippet that gives you an idea of how this can be implemented.\

```
val rddTransformed = rdd.mapPartitions(partition => {

/*DB init per partition - 5 times*/

val connection = new DBConnection()

/*map() applied on each element - 50 times*/

val partitionTransformed =

partition.map

( element => {

dosomethingWithDBConnection(element, connection)

}).toList

/*Below code calls once per partition - 5 times*/

connection.close() // close dbconnection here

partitionTransformed.iterator // returns iterator

})
```

Question 4: Difference Between Cache & Persist in Spark?

cache() and persist() functions are used to cache intermediate results of a RDD or DataFrame or Dataset. You can mark an RDD, DataFrame or Dataset to be persisted using the persist() or cache() methods on it. The first time it is computed in an action, the objects behind the RDD, DataFrame or Dataset on which cache() or persist() is called will be kept in memory or on the configured storage level on the nodes.

Cache(): cache is useful when the lineage of the RDD branches out. Let's say you want to filter the words of the previous example into a count for positive and negative words. You could do this like that:

```
val textFile = sc.textFile("/tmp/user.txt")

val wordsRDD = textFile.flatMap(line => line.split(" "))

wordsRDD.cache()

val positiveWordsCount = wordsRDD.filter(word => isPositive(word)).count()
```

val negativeWordsCount = wordsRDD.filter(word => isNegative(word)).count()

Before we look at the differences let’s look at the storage levels first.

cache() doesn’t take any parameters

cache() on RDD will persist the objects in memory.

RDD

rdd.cache()

cache() on DataFrame or Dataset will persist the objects in memory_and_disk (check storage levels below)

DataFrame

df.cache()

Dataset

ds.cache()

persist()

There are 2 flavours of persist() functions

persist() – without argument. When called without argument, calls cache() internally.

RDD

rdd.persist()

DataFrame

df.persist()

Dataset

ds.persist()

persist(StorageLevel) – with StorageLevel as argument

RDD

rdd.persist(StorageLevel.MEMORY_ONLY_SER)

DataFrame

df.persist(StorageLevel.DISK_ONLY)

Dataset

ds.persist(StorageLevel.MEMORY_AND_DISK)

Based on the provided StorageLevel, the behaviour of the persisted objects will vary.

Storage levels

Storage level defines 3 things –

1. whether the objects that are being persisted should be serialized or deserialized
2. whether the objects should be kept in memory or disk
3. whether the persisted objects should be replicated or not

MEMORY_ONLY

MEMORY_AND_DISK

MEMORY_ONLY_SER (Java and Scala)

MEMORY_AND_DISK_SER (Java and Scala)

DISK_ONLY

MEMORY_ONLY_2, MEMORY_AND_DISK_2, etc.

OFF_HEAP (experimental)

Which storage level to use?

MEMORY_ONLY option is CPU efficient and optimal for performance

MEMORY_ONLY_SER option if the RDD, DataFrame or Dataset has a lot of elements. Use an efficient serialization framework like Kyro. This option is space efficient.

Spilling data to disk is expensive so use DISK_ONLY and options like MEMORY_AND_DISK only when the computation involved in getting to the RDD, DataFrame or Dataset that is being persisted is expensive.

Use replication options like MEMORY_ONLY_2 only you have a mandate for full fast recovery.

Question 5: Difference Between Union and UnionAll in SparkSQL.

Using Spark Union and UnionAll you can merge data of 2 Dataframes and create a new Dataframe. Remember you can merge 2 Spark Dataframes only when they have the same Schema. Union All is deprecated since SPARK 2.0 and it is not advised to use any longer. Let's check with a few examples.

Note:- Union only merges the data between 2 Dataframes but does not remove duplicates after the merge.

SYNTAX of UNION in Spark Dataframe

The syntax is pretty straight forward

```
df1.union(df2)
```

where df1 and df2 are 2 dataframes with same schema.

Lets check this with an example.

```
val df1 = Seq(("Smith",23),("Rashmi",27)).toDF("Name","Age")

val df2 = Seq(("Smith",23),("Payal",27)).toDF("Name","Age")

df1.union(df2).show

+------+---+

| Name|Age|

+------+---+

| Smith| 23|

|Rashmi| 27|

| Smith| 23|

| Payal| 27|

+------+---+
```

Here we created 2 dataframes and did a union operation on them. Notice that the duplicate records are not removed.

SYNTAX of UNION ALL in Spark Dataframe

Syntax of union all is similar to union.

```
df1.unionAll(df2)
```

This works similar to union.

```
val df1 = Seq(("Smith",23),("Rashmi",27)).toDF("Name","Age")

val df2 = Seq(("Smith",23),("Payal",27)).toDF("Name","Age")

df1.unionAll(df2).show

+------+---+

| Name|Age|

+------+---+

| Smith| 23|

|Rashmi| 27|

| Smith| 23|

| Payal| 27|

+------+---+

warning: method unionAll in class Dataset is deprecated: use union()
```

Notice that as soon as you use unionAll you immediately get a warning that unionAll is deprecated and instead it suggests to use union.

Merge 2 Dataframes and Remove Duplicates.

In case you need to remove the duplicates after merging them you need to use distinct or dropDuplicates after merging them.

Lets check an example below.

```
val df1 = Seq(("Smith",23),("Rashmi",27)).toDF("Name","Age")

val df2 = Seq(("Smith",23),("Payal",27)).toDF("Name","Age")

df1.union(df2).

distinct.show

+------+---+

| Name|Age|

+------+---+

| Payal| 27|

|Rashmi| 27|

| Smith| 23|

+------+---+
```

Merge Multiple Dataframes.

You can merge N number of dataframes one after another by using the union keyword multiple times. We will see an example for the same. But what if there are 100’s of dataframes you need to merge. Will you be writing union as many times or is there a better way.

First let’s create 3 dataframes that we need to merge.

```
val df1 = Seq(("Smith",23),("Rashmi",27)).toDF("Name","Age")

val df2 = Seq(("Smith",23),("Payal",27)).toDF("Name","Age")

val df3 = Seq(("Kevin",29)).toDF("Name","Age")
```

Now the First method is to us union keyword multiple times to merge the 3 dataframes.

```
f1.union(df2).union(df3).show

+------+---+

| Name|Age|

+------+---+

| Smith| 23|

|Rashmi| 27|

| Smith| 23|

| Payal| 27|

| Kevin| 29|

+------+---+
```

Question 6: How Spark Internally Executes a Program: (Word Count Program)

I will try to explain how Spark works internally and what the components of execution are: jobs, tasks, and stages.

As we all know, Spark gives us two operations for performing any problem.

When we do a transformation on any RDD, it gives us a new RDD. But it does not start the execution of those transformations. The execution is performed only when an action is performed on the new RDD and gives us a final result.

So once you perform any action on an RDD, Spark context gives your program to the driver.

The driver creates the DAG (directed acyclic graph) or execution plan (job) for your program. Once the DAG is created, the driver divides this DAG into a number of stages. These stages are then divided into smaller tasks and all the tasks are given to the executors for execution.

The Spark driver is responsible for converting a user program into units of physical execution called tasks. At a high level, all Spark programs follow the same structure. They create RDDs from some input, derive new RDDs from those using transformations, and perform actions to collect or save data. A Spark program implicitly creates a logical directed acyclic graph (DAG) of operations.

When the driver runs, it converts this logical graph into a physical execution plan.

So, let's take an example of word count for better understanding:

```
val rdd = sc.textFile("address of your file")

rdd.flatMap(_.split(" ")).map(x=>(x,1)).reduceByKey(_ + _).collect
```

Here you can see that collect is an action that will collect all data and give a final result. As explained above, when I perform the collect action, the Spark driver creates a DAG.

In the first image, you can see that one job is created and executed successfully. Now, let's have a look at DAG and its stages.

In the second image, you can see that Spark created the DAG for the program written above and divided the DAG into two stages.

In this DAG, you can see a clear picture of the program. First, the text file is read. Then, the transformations like map and flatMap are applied. Finally, reduceBykey is executed.

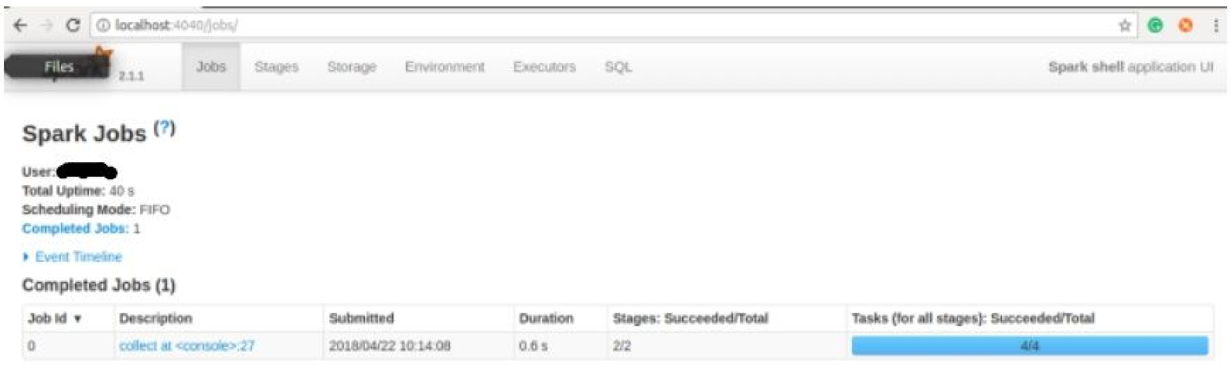
But why did Spark divide this program into two stages? Why not more than two or less than two? Basically, it depends on shuffling, i.e. whenever you perform any transformation where Spark needs to shuffle the data by communicating to the other partitions, it creates other stages for such transformations. And the transformation does not require the shuffling of your data; it creates a single stage for it.

Now, let's have a look at how many tasks have been created by Spark:

In this program, we have only two partitions, so each stage is divided into two tasks. And a single task runs on a single partition. The number of tasks for a job is:

(no of your stages * no of your partitions)

Now, I think you may have a clear picture of how Spark works internally.



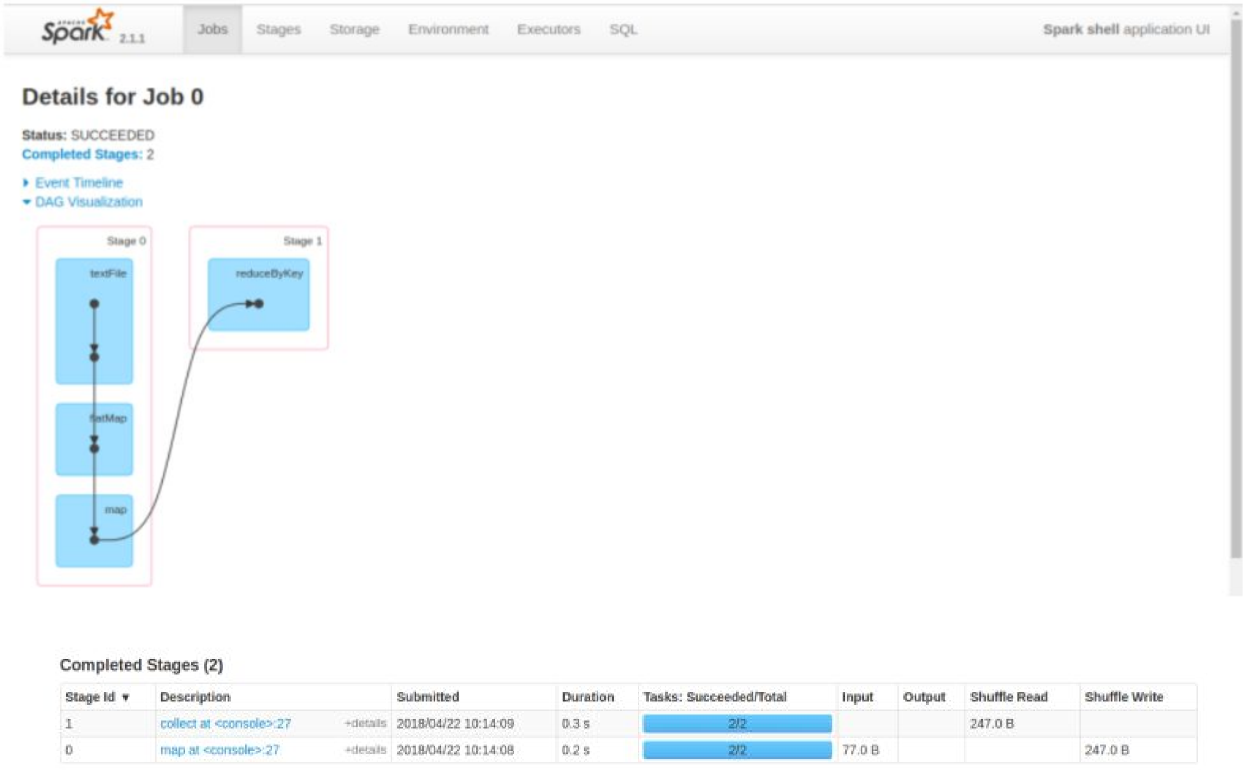
Spark Jobs (?)

User: [redacted]
Total Uptime: 40 s
Scheduling Mode: FIFO
Completed Jobs: 1

Event Timeline

Completed Jobs (1)

Job Id ▾	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
0	collect at <console>:27	2018/04/22 10:14:08	0.6 s	2/2	4/4



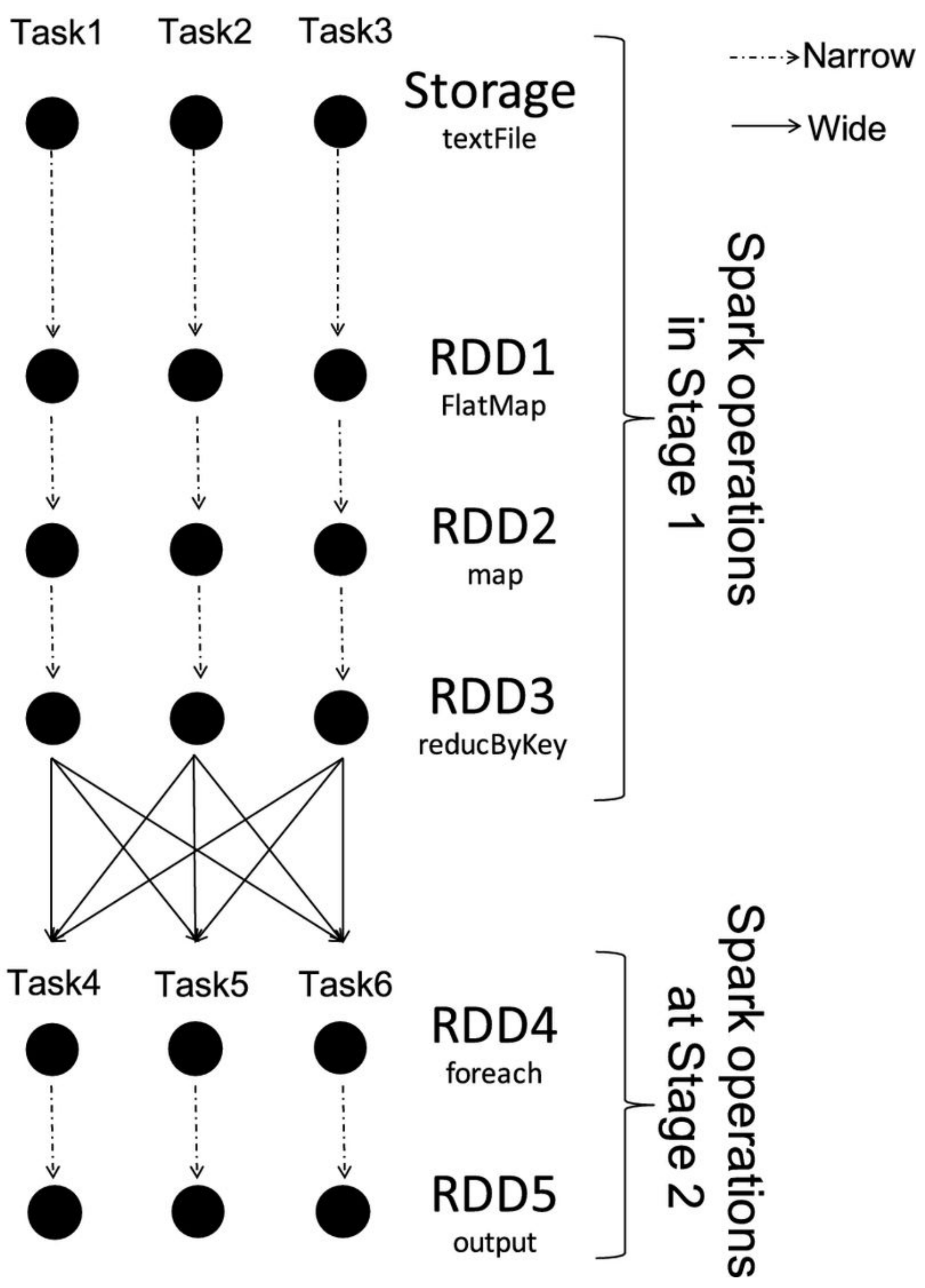
Details for Job 0

Status: SUCCEEDED
Completed Stages: 2

▸ Event Timeline
▾ DAG Visualization

Completed Stages (2)

Stage Id ▾	Description		Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
1	collect at <console>:27	+details	2018/04/22 10:14:09	0.3 s	2/2			247.0 B	
0	map at <console>:27	+details	2018/04/22 10:14:08	0.2 s	2/2	77.0 B			247.0 B



Question 7 : Difference between Repartition & Coalesce

In Spark, there are two ways of explicitly changing the number of partitions. We can use either the repartition function or coalesce. Both accept a numeric parameter, which tells Spark how many partitions we want to have. I will explain the difference between those two functions and when we should use them.

Repartition :

· Increase or decrease partitions.

· Repartition always involves a shuffle.

· Repartition works by creating new partitions and doing a full shuffle to move data around.

· Results in more or less equal sized partitions.

· Since a full shuffle takes place, repartition is less performant than coalesce.

Coalesce :

· Only decrease the number of partitions.

· Coalesce doesn’t involve a full shuffle.

· If the number of partitions is reduced from 5 to 2. Coalesce will not move data in 2 executors and move the data from the remaining 3 executors to the 2 executors. Thereby avoiding a full shuffle.

· Because of the above reason the partition size vary by a high degree.

· Since full shuffle is avoided, coalesce is more performant than repartition.

Finally, When you call the repartition() function, Spark internally calls the coalesce function with shuffle parameter set to true.

When to Use Coalesce Instead of Repartition?

According to the documentation, it is better to run repartition instead of coalesce when we want to do “drastic coalesce” (for example, merge everything into one partition). When we do it, the upstream calculations execute in parallel, so the entire computation is distributed across a larger number of workers.

I can think of only one use case when I am sure that the coalesce function is a better choice. If I have just used where or filter, and I think that some of the partitions may be “almost empty”, I will want to move the data around to have a similar number of rows in every partition. coalesce will not give me uniformed distribution, but it is going to be faster than repartition, and it should be good enough in this case.

Is Coalesce Worth Consideration In Practice?

Most likely no. Usually, the Spark jobs are full of the group by and join operations, so avoiding full reshuffles using coalesce in a few places will not make a huge difference.

Simply put Partitioning data means dividing the data into smaller chunks so that they can be processed in a parallel manner.

Using Coalesce and Repartition we can change the number of partitions of a Dataframe.

Coalesce can only decrease the number of partitions.

Repartition can increase and also decrease the number of partitions.

Also, Coalesce doesn’t do a full shuffle which means it does not equally divide the data into all partitions, it moves the data to the nearest partition.

Repartition on the other hand divides the data equally into all partitions hence triggering a full shuffle. This is the reason Repartition is a costly operation.

How can we find out the number of partition of a Dataframe

We can find using rdd.partitions.size

```
scala> val df = List(1,2,3,4,5,6,7,8,9,10,11,12).toDF("num")

df: org.apache.spark.sql.DataFrame = [num: int]

scala> df.rdd.partitions.size
```

```
res1: Int = 2
```

Now using Coalesce let reduce the number of partitions in Dataframe

```
scala> val df1 = df.coalesce(1)
df1: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [num: int]
scala> df1.rdd.partitions.size
res2: Int = 1
```

Can we increase the number of partitions using Coalesce

No, as you can see below even if we try to increase the number of partition the count remains same

```
scala> val df2 = df.coalesce(4)
df2: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [num: int]
scala> df2.rdd.partitions.size
res3: Int = 2
```

How to increase or decrease the number of partitions using Repartition

```
scala> df.rdd.partitions.size
res4: Int = 2
scala> df.repartition(1).rdd.partitions.size
res5: Int = 1
scala> df.repartition(4).rdd.partitions.size
res6: Int = 4
```

How does shuffle work with Coalesce and Repartition

Coalesce doesn't do a full shuffle which means it does not equally divide the data into all partitions, it moves the data to the nearest partition.

Which means

if we have 4 Partitions

Par1 a,b,c

Par2 d,e,f

Par3 g,h,i

Par4 j,k,l

And then we use coalesce to reduce the number of paritition to 2. Now the data could look like, depending on the location of partition

Par1 a,b,c,j,k,l

Par2 d,e,f,g,h,i

As you can see in the above case, data in Par1 and Par2 were not moved, hence reducing the overall data movement.

Repartition on the other hand divides the data equally into all partitions, so it pulls out all the data and divides the equally triggering a full shuffle. This is the reason Repartition is a costly operation.

Conclusion :

repartition redistributes the data evenly, but at the cost of a shuffle

- >coalesce works much faster when you reduce the number of partitions because it sticks input partitions together
- >coalesce doesn't guarantee uniform data distribution
- >coalesce is identical to a repartition when you increase the number of partitions

And with that, massive performance benefits await if you know how to make the right tradeoffs, as with anything in life.

Question 8 : Difference between broadcast variable and accumulator in spark

On a high level, accumulators and broadcast variables both are spark shared variables. In distributed computing, understanding closure is very important. Often it creates confusion among programmers in understanding the scope and life cycle of variables and methods while executing code in a cluster. Most of the time, you will end up getting :

org.apache.spark.SparkException: Job aborted due to stage failure: Task not serializable: java.io.NotSerializableException: ..

The above error can be triggered when you initialize a variable on the driver, but try to use it on one of the executors. In such case, Spark will first try to serialize the object to send it across executors, and fail if the object is not serializable and then throw java.io.NotSerializableException. From this, it is evident that the concept of shared variable(rather a global variable) in spark is different and that will introduce you to Accumulator and Broadcast Variables, two special kinds of shared variables that spark provides.

Accumulators:

Accumulators are a special kind of variable that we basically use to update some data points across executors. One thing we really need to remember is that the operation by which the data point update is happening has to be an associate and commutative operation. Let's take an example to have a better understanding of accumulators:

Let's say, you have a big text file and you need to find out the count of the number of lines having '_'

This is a good scenario where you can use an accumulator. So, first will read the file which will create an RDD out of it.

val inputRdd = sc.textFile("D:\\intelliJ-Workspace\\spark_application\\src\\main\\sampleFileStorage\\rawtext")

It will distribute the file in multiple partitions across all the executors. Now, we need to create an accumulator in the driver. We will be using an Accumulator of type Long as we are going to count the lines with '_'.

val acc = sc.longAccumulator("Underscore Counter")

Now, will check for each line of the inputRdd, if the line contains '_', will increase the accumulator count by 1. So, in this way, whenever each executor encounters '_' within a line, it will increase the value of the accumulator by 1; so, in the end if we print the value of the accumulator, will see the exact count of lines having '_'

```
inputRdd.foreach { line =>
if (line.contains("_"))
acc.add(1)
}
println(acc.value);
```

So, the bottom line is when you want spark workers to update some value, you should go with accumulator.

Broadcast Variables:

Broadcast variables are read-only variable that will be cached in all the executors instead of shipping every time with the tasks. Basically, broadcast variables are used as lookup without any shuffle as each executor will keep a local copy of it, so no network I/O overhead involves here. Imagine, you are executing a Spark Streaming application and for each input event, you have to use a lookup data set which is distributed across multiple partitions in multiple executors; so, each event will end up doing network I/O which will be a huge costly operation for a streaming application with such frequency.

Now the question is, how big a lookup dataset do you want to broadcast!! The answer lies in the amount of memory you are allocating for each executor. See, if we broadly look at memory management in spark, will observe that 75% of the total memory spark keeps for its own storage and execution. Out of that 75%, 50% is allocated for storage purposes and the other 50% is allocated for execution purposes.

For example, if you allocate 10GB of memory to an executor, then according to the formula, your spark storage memory would be:

(“Java Heap” — 300MB) * 0.75 * 0.5 = 3.64GB(approx) [300MB is reserved memory]

Spark stores broadcast variables in this memory region along with cached data. There is a catch here. This is the initial spark memory orientation. If spark execution memory grows big with time, it will start evicting objects from the storage region and as broadcast variables get stored with MEMORY_AND_DISK persistence level, there is a possibility that it also gets evicted from memory. So, will end up doing disk I/O which is again a costly operation in terms of performance. So, bottom line is, we have to carefully choose the size of the broadcast variable keeping all this in mind. If you want to know more about Spark memory management, please check this blog Spark memory management.

So, in the end, we can say that through the accumulator, spark gives executors provision to coordinate values with each other whereas, through broadcast variable, it provides the same dataset across multiple stages without shuffling.

Broadcast variables are created by wrapping with SparkContext.broadcast function as shown in the following Scala code

```
scala> val broadcastVar = sc.broadcast(Array(1, 2, 3))
broadcastVar: org.apache.spark.broadcast.Broadcast[Array[Int]] = Broadcast(0)
scala> broadcastVar.value
res1: Array[Int] = Array(1, 2, 3)
```

Question 9 : PySpark Window Functions

PySpark Window function performs statistical operations such as rank, row number, etc. on a group, frame, or collection of rows and returns results for each row individually. It is also popularly growing to perform data transformations. We will understand the concept of window functions, syntax, and finally how to use them with PySpark SQL and PySpark DataFrame API.

There are mainly three types of Window function:

· Analytical Function

· Ranking Function

· Aggregate Function

· Custom window definition

To perform window function operation on a group of rows first, we need to partition i.e. define the group of data rows using window.partition() function, and for row number and rank function we need to additionally order by on partition data using ORDER BY clause.

Syntax for Window.partition:

Window.partitionBy(“column_name”).orderBy(“column_name”)

Syntax for Window function:

DataFrame.withColumn(“new_col_name”, Window_function().over(Window_partition))

1.Analytical functions:

An analytic function is a function that returns a result after operating on data or a finite set of rows partitioned by a SELECT clause or in the ORDER BY clause. It returns a result in the same number of rows as the number of input rows. E.g. lead(), lag(), cume_dist().

Creating dataframe for demonstration:

Before we start with these functions, first we need to create a DataFrame. We will create a DataFrame that contains employee details like Employee_Name, Age, Department, Salary. After creating the DataFrame we will apply each analytical function on this DataFrame df.

```
# importing pyspark
from pyspark.sql.window import Window
import pyspark
# importing sparksessio
from pyspark.sql import SparkSession
# creating a sparksession object
# and providing appName
spark = SparkSession.builder.appName("pyspark_window").getOrCreate()
# sample data for dataframe
sampleData = (("Ram", 28, "Sales", 3000),
("Meena", 33, "Sales", 4600),
("Robin", 40, "Sales", 4100),
```

("Kunal", 25, "Finance", 3000),

("Ram", 28, "Sales", 3000),

("Srishti", 46, "Management", 3300),

("Jeny", 26, "Finance", 3900),

("Hitesh", 30, "Marketing", 3000),

("Kailash", 29, "Marketing", 2000),

("Sharad", 39, "Sales", 4100)

)

# column names for dataframe

columns = ["Employee_Name", "Age",

"Department", "Salary"]

# creating the dataframe df

df = spark.createDataFrame(data=sampleData, schema=columns)

# importing Window from pyspark.sql.window

# creating a window

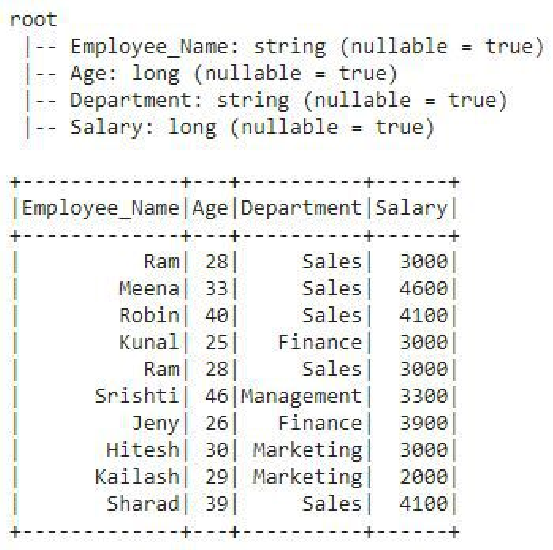
# partition of dataframe

windowPartition = Window.partitionBy("Department").orderBy("Age")

# print schema

df.printSchema()

# show df



```
root
 |-- Employee_Name: string (nullable = true)
 |-- Age: long (nullable = true)
 |-- Department: string (nullable = true)
 |-- Salary: long (nullable = true)

+-------------+---+----------+------+
|Employee_Name|Age|Department|Salary|
+-------------+---+----------+------+
|          Ram| 28|     Sales|  3000|
|        Meena| 33|     Sales|  4600|
|        Robin| 40|     Sales|  4100|
|        Kunal| 25|   Finance|  3000|
|          Ram| 28|     Sales|  3000|
|      Srishti| 46|Management|  3300|
|         Jeny| 26|   Finance|  3900|
|       Hitesh| 30| Marketing|  3000|
|      Kailash| 29| Marketing|  2000|
|       Sharad| 39|     Sales|  4100|
+-------------+---+----------+------+
```

This is the DataFrame on which we will apply all the analytical functions.

Example 1: Using cume_dist()

cume_dist() window function is used to get the cumulative distribution within a window partition. It is similar to CUME_DIST in SQL. Let's see an example:

# importing cume_dist()

# from pyspark.sql.functions

from pyspark.sql.functions import cume_dist

# applying window function with

# the help of DataFrame.withColumn

df.withColumn("cume_dist", cume_dist().over(windowPartition)).show()

Output: in the attachment

In the output, we can see that a new column is added to the df named "cume_dist" that contains the cumulative distribution of the Department column which is ordered by the Age column.

Example 2: Using lag()

A lag() function is used to access previous rows' data as per the defined offset value in the function. This function is similar to the LAG in SQL.

# importing lag() from pyspark.sql.functions

from pyspark.sql.functions import lag

df.withColumn("Lag", lag("Salary", 2).over(windowPartition)).show()

Output: in the attachment

In the output, we can see that lag column is added to the df that contains lag values. In the first 2 rows there is a null value as we have defined offset 2 followed by column Salary in the lag() function. The next rows contain the values of previous rows.

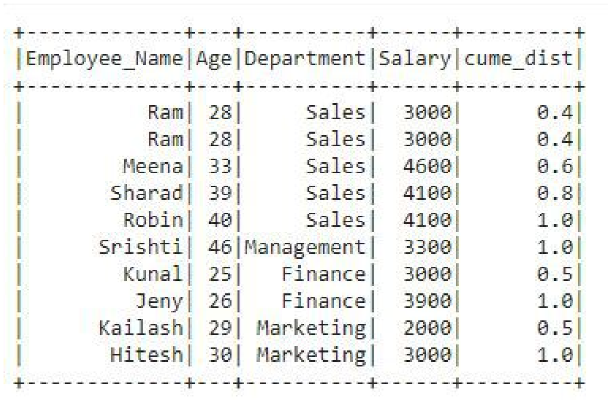
Example 3: Using lead()

A lead() function is used to access next rows data as per the defined offset value in the function. This function is similar to the LEAD in SQL and just opposite to lag() function or LAG in SQL.

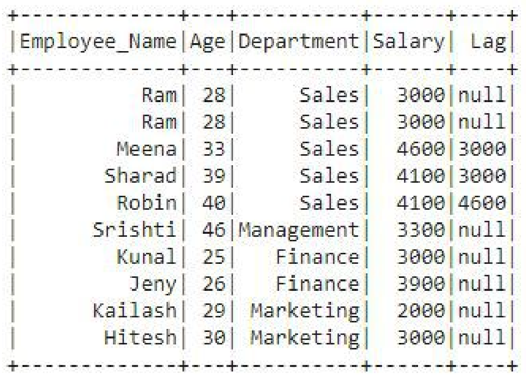
# importing lead() from pyspark.sql.functions

from pyspark.sql.functions import lead

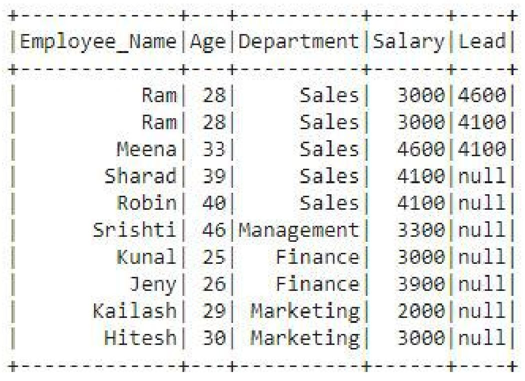
df.withColumn("Lead", lead("salary", 2).over(windowPartition)).show()



```
+-------------+---+----------+------+---------+
|Employee_Name|Age|Department|Salary|cume_dist|
+-------------+---+----------+------+---------+
|          Ram| 28|     Sales|  3000|      0.4|
|          Ram| 28|     Sales|  3000|      0.4|
|        Meena| 33|     Sales|  4600|      0.6|
|       Sharad| 39|     Sales|  4100|      0.8|
|        Robin| 40|     Sales|  4100|      1.0|
|      Srishti| 46|Management|  3300|      1.0|
|        Kunal| 25|   Finance|  3000|      0.5|
|         Jeny| 26|   Finance|  3900|      1.0|
|      Kailash| 29| Marketing|  2000|      0.5|
|       Hitesh| 30| Marketing|  3000|      1.0|
+-------------+---+----------+------+---------+
```



```
+-------------+---+----------+------+----+
|Employee_Name|Age|Department|Salary| Lag|
+-------------+---+----------+------+----+
|          Ram| 28|     Sales|  3000|null|
|          Ram| 28|     Sales|  3000|null|
|        Meena| 33|     Sales|  4600|3000|
|       Sharad| 39|     Sales|  4100|3000|
|        Robin| 40|     Sales|  4100|4600|
|      Srishti| 46|Management|  3300|null|
|        Kunal| 25|   Finance|  3000|null|
|         Jeny| 26|   Finance|  3900|null|
|      Kailash| 29| Marketing|  2000|null|
|       Hitesh| 30| Marketing|  3000|null|
+-------------+---+----------+------+----+
```



```
+-------------+---+----------+------+----+
|Employee_Name|Age|Department|Salary|Lead|
+-------------+---+----------+------+----+
|          Ram| 28|     Sales|  3000|4600|
|          Ram| 28|     Sales|  3000|4100|
|        Meena| 33|     Sales|  4600|4100|
|       Sharad| 39|     Sales|  4100|null|
|        Robin| 40|     Sales|  4100|null|
|      Srishti| 46|Management|  3300|null|
|        Kunal| 25|   Finance|  3000|null|
|         Jeny| 26|   Finance|  3900|null|
|      Kailash| 29| Marketing|  2000|null|
|       Hitesh| 30| Marketing|  3000|null|
+-------------+---+----------+------+----+
```

Question 10 : Ranking Functions in Spark

Ranking Function:

The function returns the statistical rank of a given value for each row in a partition or group. The goal of this function is to provide consecutive numbering of the rows in the resultant column, set by the order selected in the Window.partition for each partition specified in the OVER clause. E.g. row_number(), rank(), dense_rank(), etc.

Creating Dataframe for demonstration:

Before we start with these functions, first we need to create a DataFrame. We will create a DataFrame that contains student details like Roll_No, Student_Name, Subject, Marks. After creating the DataFrame we will apply each Ranking function on this DataFrame df2.

# importing pyspark

from pyspark.sql.window import Window

import pyspark

# importing sparksession

from pyspark.sql import SparkSession

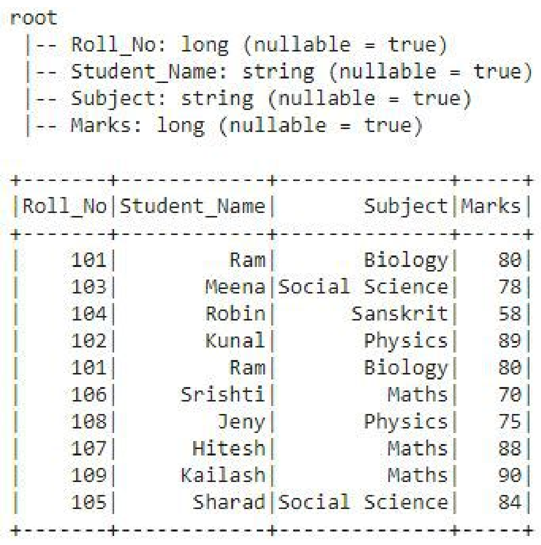
# creating a sparksession object and providing appName

spark = SparkSession.builder.appName("pyspark_window").getOrCreate()

# sample data for dataframe

```
sampleData = ((101, "Ram", "Biology", 80),
(103, "Meena", "Social Science", 78),
(104, "Robin", "Sanskrit", 58),
(102, "Kunal", "Phisycs", 89),
(101, "Ram", "Biology", 80),
(106, "Srishti", "Maths", 70),
(108, "Jeny", "Physics", 75),
(107, "Hitesh", "Maths", 88),
(109, "Kailash", "Maths", 90),
(105, "Sharad", "Social Science", 84)
)
# column names for dataframe
columns = ["Roll_No", "Student_Name", "Subject", "Marks"]
# creating the dataframe df
df2 = spark.createDataFrame(data=sampleData,schema=columns)
# importing window from pyspark.sql.window
# creating a window partition of dataframe
windowPartition = Window.partitionBy("Subject").orderBy("Marks")
# print schema
df2.printSchema()
# show df
df2.show
()
```

Output: find the attachment



```
root
 |-- Roll_No: long (nullable = true)
 |-- Student_Name: string (nullable = true)
 |-- Subject: string (nullable = true)
 |-- Marks: long (nullable = true)

+-------+------------+--------------+-----+
|Roll_No|Student_Name|       Subject|Marks|
+-------+------------+--------------+-----+
|    101|         Ram|       Biology|   80|
|    103|       Meena|Social Science|   78|
|    104|       Robin|      Sanskrit|   58|
|    102|       Kunal|       Physics|   89|
|    101|         Ram|       Biology|   80|
|    106|     Srishti|         Maths|   70|
|    108|        Jeny|       Physics|   75|
|    107|      Hitesh|         Maths|   88|
|    109|     Kailash|         Maths|   90|
|    105|      Sharad|Social Science|   84|
+-------+------------+--------------+-----+
```

Example 1: Using row_number().

row_number() function is used to gives a sequential number to each row present in the table. Let’s see the example:

# importing row_number() from pyspark.sql.functions

from pyspark.sql.functions import row_number

# applying the row_number() function

df2.withColumn("row_number", row_number().over(windowPartition)).show()

Output: Find in attachment

In this output, we can see that we have the row number for each row based on the specified partition i.e. the row numbers are given followed by the Subject and Marks column.

Example 2: Using rank()

The rank function is used to give ranks to rows specified in the window partition. This function leaves gaps in rank if there are ties. Let’s see the example:

# importing rank() from pyspark.sql.functions

from pyspark.sql.functions import rank

# applying the rank() function

df2.withColumn("rank", rank().over(windowPartition)).show()

Output: Find in attachment

In the output, the rank is provided to each row as per the Subject and Marks column as specified in the window partition.

Example 3: Using percent_rank()

This function is similar to rank() function. It also provides rank to rows but in a percentile format. Let’s see the example:

# importing percent_rank() from pyspark.sql.functions

from pyspark.sql.functions import percent_rank

# applying the percent_rank() function

df2.withColumn("percent_rank", percent_rank().over(windowPartition)).show()

Output: Find in attachment

We can see that in the output the rank column contains values in a percentile form i.e. in the decimal format.

Example 4: Using dense_rank()

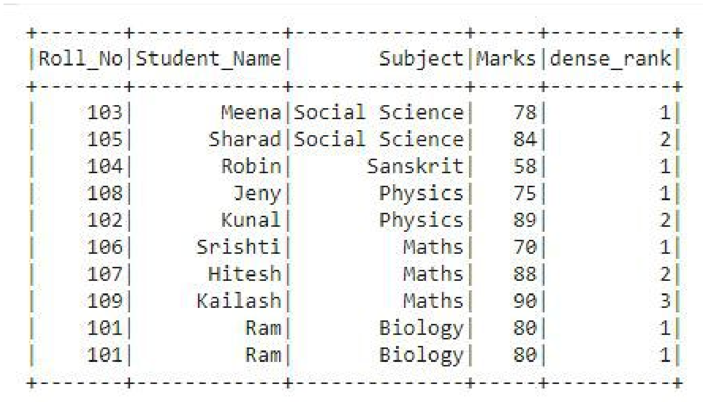
This function is used to get the rank of each row in the form of row numbers. This is similar to rank() function, there is only one difference the rank function leaves gaps in rank when there are ties. Let’s see the example:

# importing dense_rank() from pyspark.sql.functions

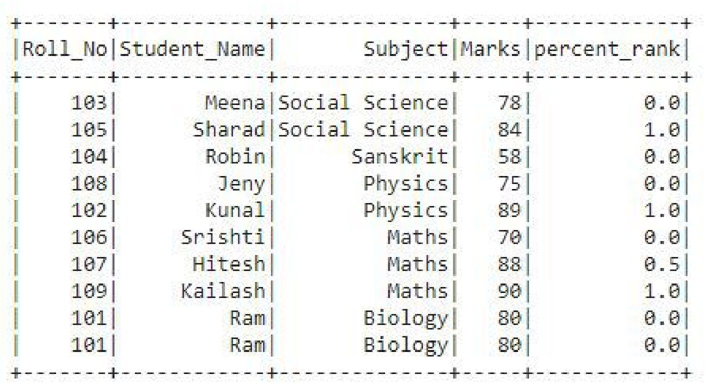
from pyspark.sql.functions import dense_rank

# applying the dense_rank() function

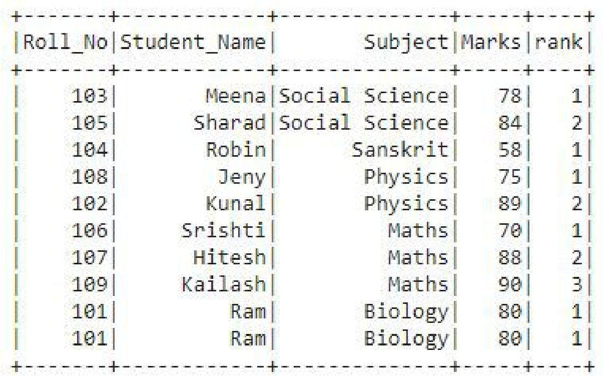
df2.withColumn("dense_rank", dense_rank().over(windowPartition)).show()



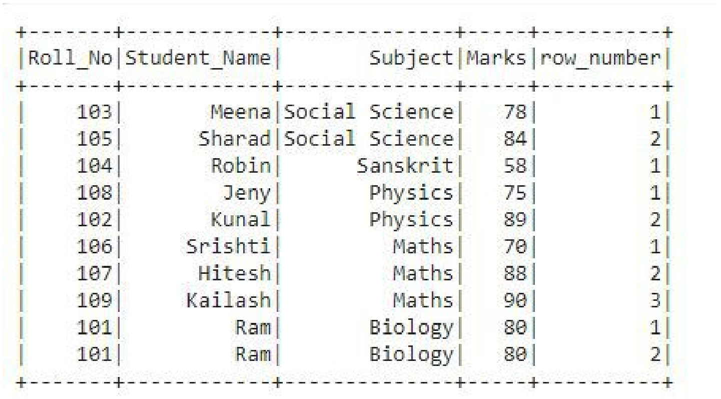
```
+-------+------------+--------------+-----+----------+
|Roll_No|Student_Name|       Subject|Marks|dense_rank|
+-------+------------+--------------+-----+----------+
|    103|       Meena|Social Science|   78|         1|
|    105|      Sharad|Social Science|   84|         2|
|    104|       Robin|      Sanskrit|   58|         1|
|    108|        Jeny|       Physics|   75|         1|
|    102|       Kunal|       Physics|   89|         2|
|    106|     Srishti|         Maths|   70|         1|
|    107|      Hitesh|         Maths|   88|         2|
|    109|     Kailash|         Maths|   90|         3|
|    101|         Ram|       Biology|   80|         1|
|    101|         Ram|       Biology|   80|         1|
+-------+------------+--------------+-----+----------+
```



```
+-------+------------+--------------+-----+------------+
|Roll_No|Student_Name|       Subject|Marks|percent_rank|
+-------+------------+--------------+-----+------------+
|    103|       Meena|Social Science|   78|         0.0|
|    105|      Sharad|Social Science|   84|         1.0|
|    104|       Robin|      Sanskrit|   58|         0.0|
|    108|        Jeny|       Physics|   75|         0.0|
|    102|       Kunal|       Physics|   89|         1.0|
|    106|     Srishti|         Maths|   70|         0.0|
|    107|      Hitesh|         Maths|   88|         0.5|
|    109|     Kailash|         Maths|   90|         1.0|
|    101|         Ram|       Biology|   80|         0.0|
|    101|         Ram|       Biology|   80|         0.0|
+-------+------------+--------------+-----+------------+
```



```
+-------+------------+--------------+-----+----+
|Roll_No|Student_Name|       Subject|Marks|rank|
+-------+------------+--------------+-----+----+
|    103|       Meena|Social Science|   78|   1|
|    105|      Sharad|Social Science|   84|   2|
|    104|       Robin|      Sanskrit|   58|   1|
|    108|        Jeny|       Physics|   75|   1|
|    102|       Kunal|       Physics|   89|   2|
|    106|    Srishti|         Maths|   70|   1|
|    107|      Hitesh|         Maths|   88|   2|
|    109|     Kailash|         Maths|   90|   3|
|    101|         Ram|       Biology|   80|   1|
|    101|         Ram|       Biology|   80|   1|
+-------+------------+--------------+-----+----+
```



```
+-------+------------+--------------+-----+----------+
|Roll_No|Student_Name|       Subject|Marks|row_number|
+-------+------------+--------------+-----+----------+
|    103|       Meena|Social Science|   78|         1|
|    105|      Sharad|Social Science|   84|         2|
|    104|       Robin|      Sanskrit|   58|         1|
|    108|        Jeny|       Physics|   75|         1|
|    102|       Kunal|       Physics|   89|         2|
|    106|    Srishti|         Maths|   70|         1|
|    107|      Hitesh|         Maths|   88|         2|
|    109|     Kailash|         Maths|   90|         3|
|    101|         Ram|       Biology|   80|         1|
|    101|         Ram|       Biology|   80|         2|
+-------+------------+--------------+-----+----------+
```

Question 11: Aggregate functions in Spark

An aggregate function or aggregation function is a function where the values of multiple rows are grouped to form a single summary value. The definition of the groups of rows on which they operate is done by using the SQL GROUP BY clause. E.g. AVERAGE, SUM, MIN, MAX, etc.

Creating Dataframe for demonstration:

Before we start with these functions, we will create a new DataFrame that contains employee details like Employee_Name, Department, and Salary. After creating the DataFrame we will apply each Aggregate function on this DataFrame.

# importing pyspark

import pyspark

# importing sparksessio

from pyspark.sql import SparkSession

# creating a sparksession

# object and providing appName

spark = SparkSession.builder.appName("pyspark_window").getOrCreate()

# sample data for dataframe

sampleData = (("Ram", "Sales", 3000),

("Meena", "Sales", 4600),

("Robin", "Sales", 4100),

("Kunal", "Finance", 3000),

("Ram", "Sales", 3000),

("Srishti", "Management", 3300),

("Jeny", "Finance", 3900),

("Hitesh", "Marketing", 3000),

("Kailash", "Marketing", 2000),

("Sharad", "Sales", 4100)

)

# column names for dataframe

columns = ["Employee_Name", "Department", "Salary"]

# creating the dataframe df

df3 = spark.createDataFrame(data=sampleData, schema=columns)

# print schema

df3.printSchema()

# show df

df3.show

()

Output: in the attachment

This is the DataFrame df3 on which we will apply all the aggregate functions.

Example: Let’s see how to apply the aggregate functions with this example

# importing window from pyspark.sql.window

from pyspark.sql.window import Window

# importing aggregate functions

# from pyspark.sql.functions

from pyspark.sql.functions import col,avg,sum,min,max,row_number

# creating a window partition of dataframe

windowPartitionAgg = Window.partitionBy("Department")

# applying window aggregate function

# to df3 with the help of withColumn

# this is average()

df3.withColumn("Avg", avg(col("salary")).over(windowPartitionAgg))

#this

is sum()

.withColumn("Sum", sum(col("salary")).over(windowPartitionAgg))

#this

is min()

.withColumn("Min", min(col("salary")).over(windowPartitionAgg))

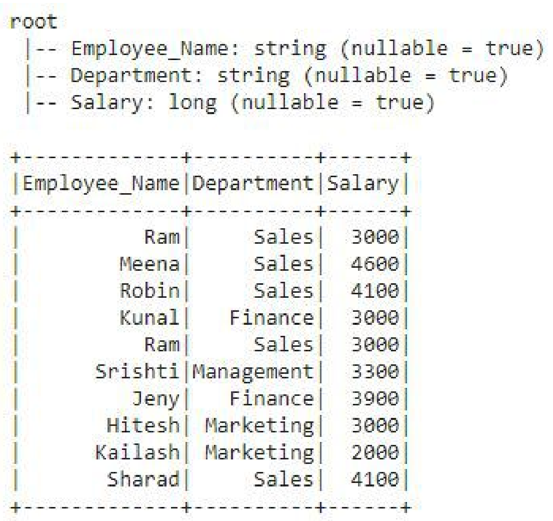
#this

is max()

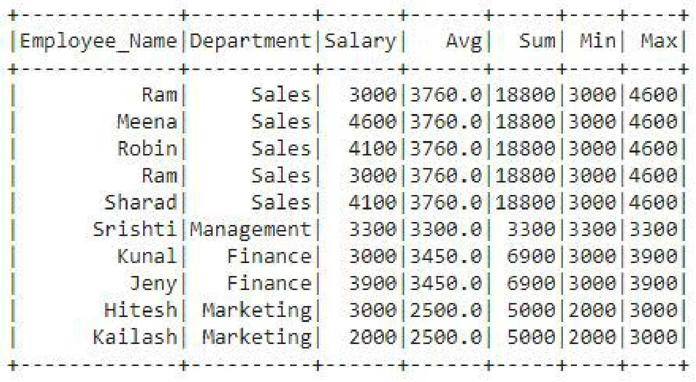
.withColumn("Max", max(col("salary")).over(windowPartitionAgg)).show()

Output: in the attachment

In the output df, we can see that there are four new columns added to df. In the code, we have applied all the four aggregate functions one by one. We got four output columns added to the df3 that contains values for each row. These four columns contain the Average, Sum, Minimum, and Maximum values of the Salary column.



```
root
 |-- Employee_Name: string (nullable = true)
 |-- Department: string (nullable = true)
 |-- Salary: long (nullable = true)

+-------------+----------+------+
|Employee_Name|Department|Salary|
+-------------+----------+------+
|          Ram|     Sales|  3000|
|        Meena|     Sales|  4600|
|        Robin|     Sales|  4100|
|        Kunal|   Finance|  3000|
|          Ram|     Sales|  3000|
|      Srishti|Management|  3300|
|         Jeny|   Finance|  3900|
|       Hitesh| Marketing|  3000|
|      Kailash| Marketing|  2000|
|       Sharad|     Sales|  4100|
+-------------+----------+------+
```



```
+-------------+----------+------+------+-----+----+----+
|Employee_Name|Department|Salary|   Avg|  Sum| Min| Max|
+-------------+----------+------+------+-----+----+----+
|          Ram|     Sales|  3000|3760.0|18800|3000|4600|
|        Meena|     Sales|  4600|3760.0|18800|3000|4600|
|        Robin|     Sales|  4100|3760.0|18800|3000|4600|
|          Ram|     Sales|  3000|3760.0|18800|3000|4600|
|       Sharad|     Sales|  4100|3760.0|18800|3000|4600|
|      Srishti|Management|  3300|3300.0| 3300|3300|3300|
|        Kunal|   Finance|  3000|3450.0| 6900|3000|3900|
|         Jeny|   Finance|  3900|3450.0| 6900|3000|3900|
|       Hitesh| Marketing|  3000|2500.0| 5000|2000|3000|
|      Kailash| Marketing|  2000|2500.0| 5000|2000|3000|
+-------------+----------+------+------+-----+----+----+
```

Question 12:Spark Submit Command Explained with Examples

· –executor-memory – Defines how much memory to set for each executor to run the application. Default 512MB. e.g

--executor-memory 2G

· –driver-memory – Defines how much memory to allocate for the application on the driver. The default is 1,024M.

--driver-memory 3G

· –num-executors: No of executor machines requested for the job. Dynamic allocation ensures that the initial number of executors is at least the number specified here. e.g.

--num-executors 12

· application-jar: This is the path to the bundled or compiled jar including your application and all dependencies. The URL must be globally visible inside of your cluster, for instance, an hdfs:// path or a file:// path that is present on all nodes. e.g.

/AA/BB/target/spark-project-1.0-SNAPSHOT.jar

· application-arguments: Arguments passed to the main method of your main class if any. Note that –

o Arguments passed before the .jar file will act as arguments to the JVM,

o Arguments passed after the jar file is considered as arguments passed to the Spark program.

/project/spark-project-1.0-SNAPSHOT.jar input1.txt input2.txt

· –jars: Mention all the dependency jars (separated by comma) needed to run the Spark Job. Note you need to give the Full path of the jars if the jars are placed in different folders. e.g.

-jars cassandra-connector.jar, some-other-package-1.jar, some-other-package-2.jar

· -files: If Spark needs any Additional files for its execution, those should be given using this option. Multiple files can be mentioned separated by a comma.

o In case of client deployment mode, the path must point to a local file.

o In case of cluster deployment mode, the path can be either a local file or a URL globally visible inside your cluster;

· -py-files – If the Spark application\job needs .zip, .egg, or .py files, you have to use this option. Two or more files should be comma-separated.

o In case of client deployment mode, the path must point to a local file.

o In case of cluster deployment mode, the path can be either a local file or a URL globally visible(within the cluster)

- -py-files dependency_files/egg.egg

Note Additional points below for PySpark job –

· If you want to run the Pyspark job in client mode, you have to install all the libraries (on the host where you execute the spark-submit) – imported outside the function maps.

· If you want to run the PySpark job in cluster mode, you have to ship the libraries using the option – archives in the spark-submit command.

Using most of the above a Basic skeleton for spark-submit command becomes –

./bin/spark-submit \

- -class<main-class> \
- -master<master-url> \
- -deploy-mode<deploy-mode> \
- -conf<key>=<value> \

... # options

<application-jar>\

[application-arguments] <--- here our app arguments

Let us combine all the above arguments and construct an example of one spark-submit command –

Spark-Submit Example 1 – Java\Scala Code:

export HADOOP_CONF_DIR=XXX

./bin/spark-submit\

- -classorg.com.sparkProject.examples.MyApp \
- -master yarn \
- -deploy-mode cluster \
- -executor-memory 20G \
- -num-executors 50 \
- -conf "spark.eventlog.enabled=true"
- -jars cassandra-connector.jar, some-other-package-1.jar, some-other-package-2.jar

/project/spark-project-1.0-SNAPSHOT.jar input1.txt input2.txt

#Argument

to the Program

Spark-Submit Example 2- Python Code:

Let us combine all the above arguments and construct an example of one spark-submit command –

./bin/spark-submit \

- -master yarn \
- -deploy-mode cluster \
- -executor-memory 5G \
- -executor-cores 8 \
- -py-files dependency_files/egg.egg
- -archives dependencies.tar.gz

mainPythonCode.py

value1 value2

#This

is the Main Python Spark code file followed by

#arguments

(value1,value2) passed to the program.

Example of how the arguments passed (value1, value2) can be handled inside the program.

import os

import sys

n = int(sys.argv[1])

a = 2

argspassed = []

for _ in

range(n):

argspassed.append(sys.argv[a])

a += 1

print(argspassed)

Spark-Submit Example 3 – Local:

./bin/spark-submit

- -class org.com.sparkProject.examples.MyApp \
- -master local[2] \

#Running

on 2-cores

/project/spark-project-1.0-SNAPSHOT.jar input.txt

Spark-Submit Example 4 – Standalone(Deploy Mode-Client):

./bin/spark-submit

- -class org.com.sparkProject.examples.MyApp \
- -master spark://<IP_Address:Port_No>

/project/spark-project-1.0-SNAPSHOT.jar input.txt

Spark-Submit Example 5 – Standalone (Deploy Mode-Cluster):

./bin/spark-submit

- -class org.com.sparkProject.examples.MyApp
- -master spark://<IP_Address:Port_No>
- -deploy-mode cluster

/project/spark-project-1.0-SNAPSHOT.jar input.txt

Spark-Submit Example 6 – Deploy Mode – Yarn Cluster:

export HADOOP_CONF_DIR=XXX ./bin/spark-submit

- -class org.com.sparkProject.examples.MyApp
- -master yarn
- -deploy-mode cluster
- -executor-memory 5G
- -num-executors10

/project/spark-project-1.0-SNAPSHOT.jar input.txt

Spark-Submit Example 7 – Kubernetes Cluster:

export HADOOP_CONF_DIR=XXX ./bin/spark-submit

- -class org.com.sparkProject.examples.MyApp
- -master k8s://<IP_Address>:443
- -deploy-mode cluster
- -executor-memory 5G
- -num-executors 10

/project/spark-project-1.0-SNAPSHOT.jar input.txt

Question 13: BUCKETING Concept in Spark and Hive.

Partitioning and bucketing are the most common optimization techniques we have been using in Hive and Spark. I shall be talking specifically on the bucketing and how is it should be used.

Bucketing works only when the given cases are met. And we shall be talking about those and how to get the most out of bucketing.

What is bucketing?

Spark and Hive Bucketing is an optimization technique. We provide the column by which the data needs to be partitioned.

We need to make sure that the bucketing conditions are met to get the most out of it. When it is applied properly it will improve the joins by avoiding shuffles which is been a pain point in the Spark.

Advantages of Bucketing the Tables in Spark:

Below are some of the advantages of bucketing (clustering) in Spark:

· We can partition the data based on the given column

· Optimised Joins when you use pre-shuffled bucketed tables.

· Evenly distribution of the data.

· Optimal access and query improvement.

How is Spark bucketing different from Hive bucketing?

Hive Bucketing is not compatible with Spark Bucketing.

Hive uses the Hive hash function to create the buckets whereas Spark uses the Murmur3. So here there would be an extra Exchange and Sort when we join Hive bucketed table with Spark Bucketed table.

In Hive, it needs the reducer per which the number of files to be created.

But whereas in Spark bucketing, we do not have a reducer so it would end up creating N number of files based on the number of tasks.

Check-in Image------------------

We tend to think that when we defined the number of buckets as 1024, the number of files should be 1024.

But this is not the case with Spark.

In Spark, each task associated while inserting data into the table will be multiplied by 1024 buckets.

For example, in our case,

We have 4 tasks running.

Check-in Image--------------------

So, 4(tasks)* 1024(number of buckets) = 4099

Check-in Image--------------------

Based on the above calculation the number of files it created is: ~4096

We need to be very cautious while using bucketing in spark.

If the number of tasks grows exponentially it might end up creating millions of files.

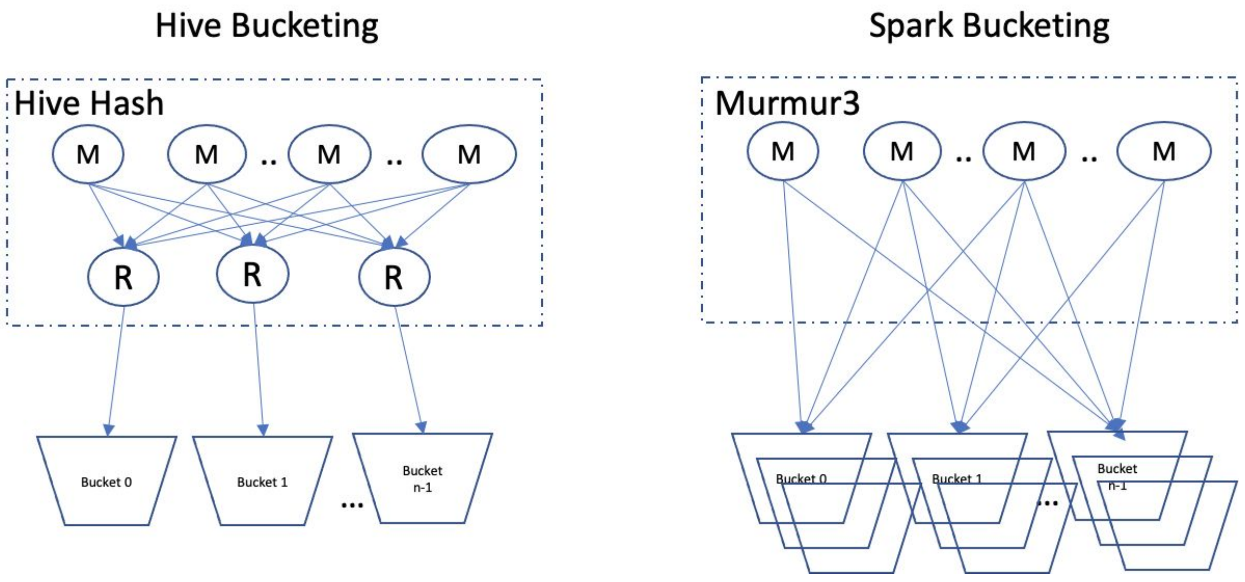
Limitation in Spark Bucketing:

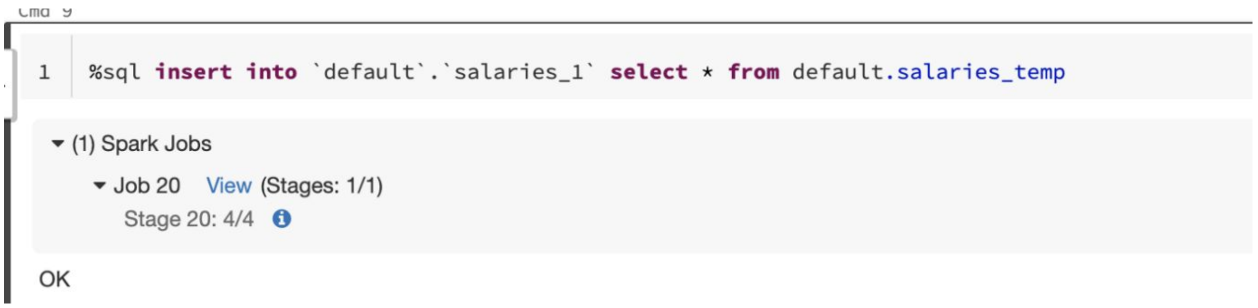
Spark Bucketing has its own limitations and we need to be very cautious when we create the bucketed tables and while we join them together.

To optimize the join and make use of bucketing in Spark we need to be sure of the below:

· Both the tables are Bucketed with the same number of buckets. If the bucket numbers are different in the joining tables, the pre-shuffle will not be applied.

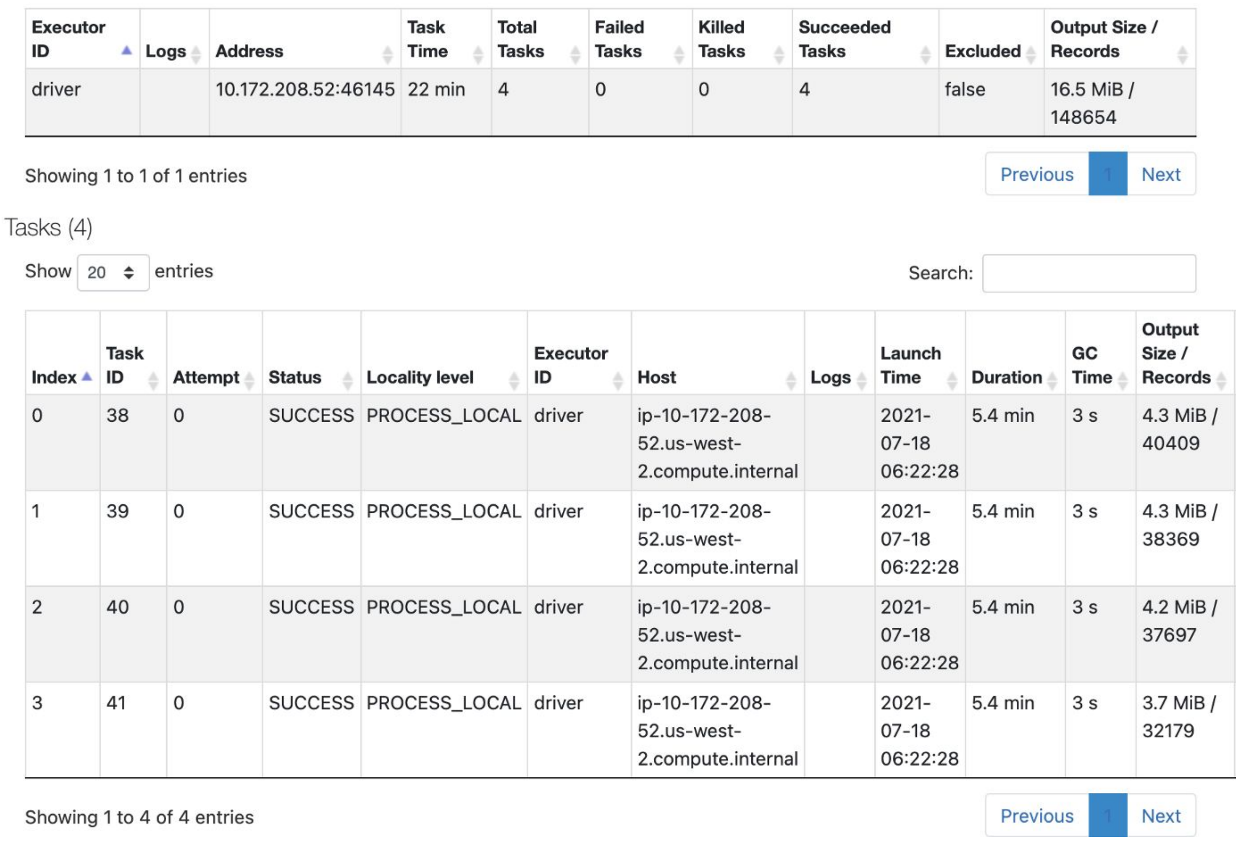
· Both the tables are bucketed on the same column for joining. As the data is partitioned based on the given bucketed column, if we do not use the same column for joining, you are not making use of bucketing and it will hit the performance.





```
%sql insert into `default`.`salaries_1` select * from default.salaries_temp
```

```
(1) Spark Jobs
  Job 20 View (Stages: 1/1)
    Stage 20: 4/4

OK
```



Executor ID	Logs	Address	Task Time	Total Tasks	Failed Tasks	Killed Tasks	Succeeded Tasks	Excluded	Output Size / Records
driver		10.172.208.52:46145	22 min	4	0	0	4	false	16.5 MiB / 148654

Showing 1 to 1 of 1 entries

Previous 1 Next

Tasks (4)

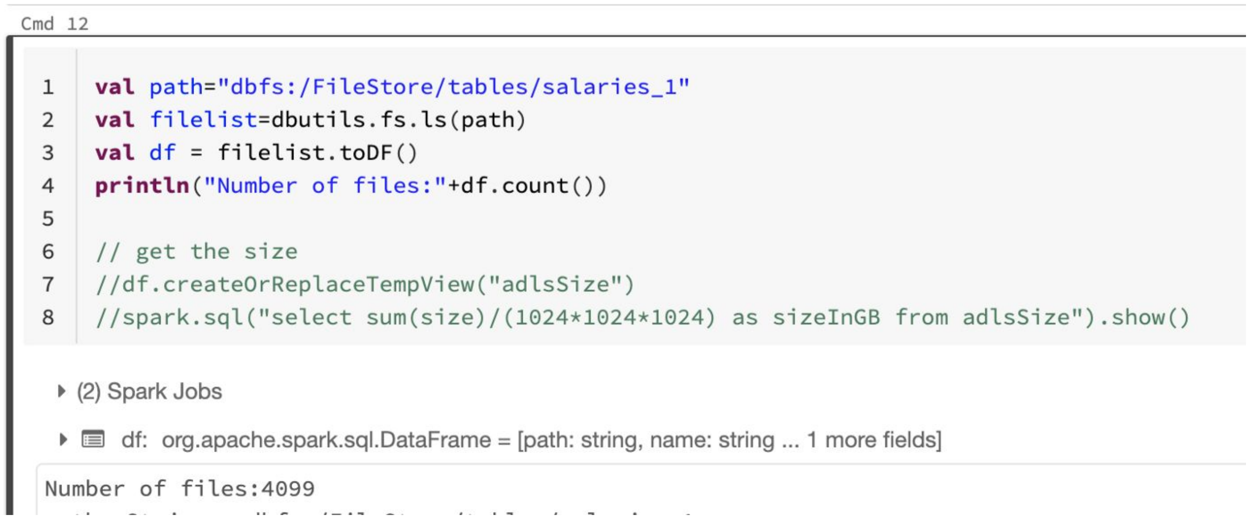
Show 20 entries

Search:

Index	Task ID	Attempt	Status	Locality level	Executor ID	Host	Logs	Launch Time	Duration	GC Time	Output Size / Records
0	38	0	SUCCESS	PROCESS_LOCAL	driver	ip-10-172-208-52.us-west-2.compute.internal		2021-07-18 06:22:28	5.4 min	3 s	4.3 MiB / 40409
1	39	0	SUCCESS	PROCESS_LOCAL	driver	ip-10-172-208-52.us-west-2.compute.internal		2021-07-18 06:22:28	5.4 min	3 s	4.3 MiB / 38369
2	40	0	SUCCESS	PROCESS_LOCAL	driver	ip-10-172-208-52.us-west-2.compute.internal		2021-07-18 06:22:28	5.4 min	3 s	4.2 MiB / 37697
3	41	0	SUCCESS	PROCESS_LOCAL	driver	ip-10-172-208-52.us-west-2.compute.internal		2021-07-18 06:22:28	5.4 min	3 s	3.7 MiB / 32179

Showing 1 to 4 of 4 entries

Previous 1 Next



Cmd 12

```scala
val path="dbfs:/FileStore/tables/salaries_1"
val filelist=dbutils.fs.ls(path)
val df = filelist.toDF()
println("Number of files:"+df.count())

// get the size
//df.createOrReplaceTempView("adlsSize")
//spark.sql("select sum(size)/(1024*1024*1024) as sizeInGB from adlsSize").show()
```

(2) Spark Jobs

df: org.apache.spark.sql.DataFrame = [path: string, name: string ... 1 more fields]

```
Number of files:4099
```

Question 14: Partition and Bucketing in Spark

Tips and Traps

- Bucketed column is only supported in Hive table at this time.
- A Hive table can have both partition and bucket columns.

· Suppose t1 and t2 are 2 bucketed tables and with the number of buckets b1 and b2 respectively. For bucket optimization to kick in when joining them:

- The 2 tables must be bucketed on the same keys/columns.

- Must join on the bucket keys/columns.

- `b1` is a multiple of `b2` or `b2` is a multiple of `b1`.

When there are many bucketed tables that might join with each other, the number of buckets needs to be carefully designed so that efficient bucket join can always be leveraged.

· Bucket for optimized filtering is available in Spark 2.4+. For example, if the table person has a bucketed column id with an integer-compatible type, then the following query in Spark 2.4+ will be optimized to avoid a scan of the whole table. A few things to be aware of here. First, you will still see a number of tasks close to the number of buckets in your Spark application. This is because the optimized job will still have to check all buckets of the table to see whether they are the right bucket corresponding to id=123. (If yes, Spark will scan all rows in the bucket to filter records. If not, the bucket will skip saving time.) Second, the type of value to compare must be compatible in order for Spark SQL to leverage bucket filtering. For example, if the id column in the person table is of the BigInt type and id = 123 is changed to id = “123” in the following query, Spark will have to do a full table scan (even if it sounds extremely stupid to do so).

```
:::sql

SELECT *

FROM persons

WHERE id = 123
```

· When you use multiple bucket columns in a Hive table, the hashing for a bucket on a record is calculated based on a string concatenating the value of all bucket columns. This means that to leverage bucket join or bucket filtering, all bucket columns must be used in joining conditions or filtering conditions.

The benefit of Partition Columns:

Spark supports partition pruning which skips scanning of non-needed partition files when filtering on partition columns. However, notice that partition columns do not help much on joining in Spark.

When to Use Partition Columns:

· Table size is big (> 50G).

· The table has low cardinality columns which are frequently used in filtering conditions.

How to Choose Partition Columns:

· Choose low cardinality columns as partition columns (since an HDFS directory will be created for each partition value combination). Generally speaking, the total number of partition combinations should be less than 50K.

· The columns are used frequently in filtering conditions.

· Use at most 2 partition columns as each partition column creates a new layer of the directory.

Bucket columns in Hive tables are similar to primary indexes in Teradata.

Benefits of Bucket Columns:

· Spark supports bucket pruning which skips scanning of non-needed bucket files when filtering on bucket columns.

· Bucket join will be leveraged when the 2 joining tables are both bucketed by joining keys of the same data type and bucket numbers of the 2 tables have a times relationship (e.g., 500 vs 1000).

· The number of buckets helps guide the Spark engine on parallel execution levels.

When to Use Bucket Columns:

· Table size is big (> 200G).

· The table has high cardinality columns which are frequently used as filtering and/or joining keys.

· medium size table but it is mainly used to join with a huge bucketize table, it is still beneficial to bucketize it

· the sort-merge join (without bucket) is slow due to shuffle not due to data skew

How to Configure Bucket Columns:

· Choose high cardinality columns as bucket columns.

· Try to avoid data skew.

· At least 500 buckets (as a small bucket number will cause poor parallel execution).

· Sorting buckets are optional but strongly recommended.

Small Files:

name node, each file is an object in name node, lots of small files put lots of pressure on the name node also, compute engine also has latencies managing jobs. Lots of small files might result in lots of small tasks which downgrade the performance of the Spark compute engine.

Frequent insert DML will introduce many small files in your tables' HDFS directory which will downgrade query performance and even impact HDFS name node stability.

To avoid too many small files:

· do not use "insert into tables values ..." in iterations.

· Trigger regular compact in your data processing compact table sales; # or compact table sales partition (site=0);

Analyze Tables/Columns:

Table and column level statistics info can help the Spark compute engine to accelerate the query performance with accurate estimation. It is suggested that you populate it after your regular data processing.

analyze table sales computer statistics -- table level analyze table sales compute statistics for columns item_id -- column level analyze tables in db_name compute statistics -- analyze all tables in db_name

Misc:

Do not use complex view definitions.

Good Practices:

· use partition/bucket columns in filtering conditions

· Spark 3 automatically handle simple situations of data skew, however, it doesn't work for complicated query especially when data skew happens in subqueries. It is suggested that you manually optimize your query for data skew issues at this time.

· cast data types to be the same for bucket joins

· consolidate UDFs -> use a subquery to invoke a UDF once

· use view to manage data access only. Avoid complex logic in a view definition. NEVER nest view logic in your data process flow!

· Add explain optimize before your query to analyze data and recommendation on your query!

· put small join table ahead (but I think Spark 3 handles this automatically now)

· A like X or A like Y -> A like any(X, Y)

Data Engineer

· use Parquet format, use Delta tables if you want to support update/delete, otherwise, use Hive tables

· partition/bucket on large tables and sort buckets

· reducing the number of small files

· avoid complex view definition

· data cache

· Bloom Filter Index

Question 15: Why do we use ORC Format in HIVE.?

ORC File Format:

The Optimized Row Columnar (ORC) file format provides a highly efficient way to store Hive data. It was designed to overcome the limitations of the other Hive file formats. Using ORC files improves performance when Hive is reading, writing, and processing data.

Compared with RCFile format, for example, ORC file format has many advantages such as:

1)a single file as the output of each task, which reduces the NameNode's load.

2)Hive type support including DateTime, decimal, and the complex types (struct, list, map, and union)

3)light-weight indexes stored within the file

3.1)skip row groups that don't pass predicate filtering

3.2)seek to a given row

4)block-mode compression based on data type

4.1)run-length encoding for integer columns

4.2)dictionary encoding for string columns

5)concurrent reads of the same file using separate RecordReaders

6)ability to split files without scanning for markers

7)bound the amount of memory needed for reading or writing

8)metadata stored using Protocol Buffers, which allows addition and removal of fields

File Structure:

An ORC file contains groups of row data called stripes, along with auxiliary information in a file footer. At the end of the file, a postscript holds compression parameters and the size of the compressed footer.

The default stripe size is 250 MB. Large stripe sizes enable large, efficient reads from HDFS.

The file footer contains a list of stripes in the file, the number of rows per stripe, and each column's data type. It also contains column-level aggregates count, min, max, and sum.

This diagram illustrates the ORC file structure:

Check-in Attachment-----------------

Stripe Structure:

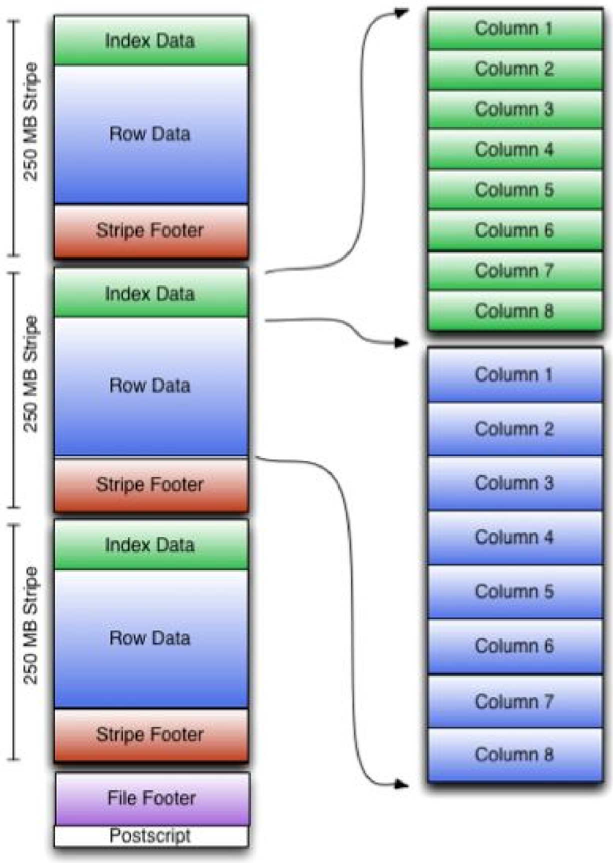
As shown in the diagram, each stripe in an ORC file holds index data, row data, and a stripe footer.

The stripe footer contains a directory of stream locations. Row data is used in table scans.

Index data include min and max values for each column and the row positions within each column. (A bit-field or bloom filter could also be included.) Row index entries provide offsets that enable seeking the right compression block and byte within a decompressed block. Note that ORC indexes are used only for the selection of stripes and row groups and not for answering queries.

Having relatively frequent row index entries enables row-skipping within a stripe for rapid reads, despite large stripe sizes. By default, every 10,000 rows can be skipped.

With the ability to skip large sets of rows based on filter predicates, you can sort a table on its secondary keys to achieve a big reduction in execution time. For example, if the primary partition is transaction date, the table can be sorted on state, zip code, and last name. Then looking for records in one state will skip the records of all other states.



HiveQL Syntax:

File formats are specified at the table (or partition) level. You can specify the ORC file format with HiveQL statements such as these:

1)CREATE TABLE ... STORED AS ORC

2)ALTER TABLE ... [PARTITION partition_spec] SET FILEFORMAT ORC

3)SET hive.default.fileformat=Orc

For example, creating an ORC stored table without compression:

create table Addresses (

name string,

street string,

city string,

state string,

zip int

) stored as orc tblproperties ("orc.compress"="NONE");

The parameters are all placed in the TBLPROPERTIES

Advantages:

1)Column stored separately

2)Knows Types - Uses Types specific encoders

3)Stores statistics (Min, Max, Sum, Count)

4)Has Lightweight Index

5)Skip over blocks of rows that that doesn’t matter

6)Larger Blocks - 256 MB by default, has an index for block boundaries

Query Vectorization:

Vectorization allows Hive to process a batch of rows together instead of processing one row at a time. Each batch is usually an array of primitive types. Operations are performed on the entire column vector, which improves the instruction pipelines and cache usage.

The hive Vectorization technique is a feature (in both MapReduce and Tez Engine) that greatly reduces the CPU usage for typical query operations like scans, filters, aggregates, and joins. A standard query execution system processes one row at a time. This involves long code paths and significant metadata interpretation in the inner loop of execution. Vectorized query execution streamlines operations by processing a block of 1024 rows at a time.

Enable Vectorization in Hive:

To enable vectorization, set this configuration parameter:

This technique is off by default, so your queries only utilize it if this variable is turned on to true. One must store the data in ORC format, and set the following variable as shown in Hive SQL to use vectorized query execution.

set hive.vectorized.execution.enabled=true

#default

false

set hive.vectorized.execution.reduce.enabled = true;

When vectorization is enabled, Hive examines the query and the data to determine whether vectorization can be supported. If it cannot be supported, Hive will execute the query with vectorization turned off.

Why use Hive Vectorization?

It is very useful when doing a row-wise transformation to Hive Table or doing any machine learning applications. Sometimes we might need to send data through standard input to Hive Custom Mapper and reducer. Instead of sending one record at a time we can use Hive containerization and send batches of data at once to custom Mapper and Reducer Scripts.

Question 16 : Why do we use Parquet Format in Spark.

Parquet - Overview:

Apache Parquet file format mainly backed up by Cloudera Service is an open-source format. It is a column-oriented data storage format inspired by Google Dremel paper. We can observe there is not much difference between Parquet and other columnar-storage file formats available in Hadoop namely RC File format and ORC format. It is efficient enough to handle a huge volume of data, i.e., even Terabytes of data can be handled with Parquet, as it maintains a good compression ratio and schema encryption.

Check-in Attachment----------------

Parquet file format is broadly classified as three components, namely

Parquet Header

Data Block

Parquet Footer

The header comprises of four-byte value as 'PAR1', this value indicates the Application that the file format processing is of type parquet.

Data Blocks in parquet are the group of records that consist of column chunks and column metadata. Column chunks are further divided into pages. Each page consists of the value of the particular column of the particular record in the given data-set. Column metadata contains the information the column such as type, path, encoding type, number of values, compressed size. The overall picture of the data block is shown in the below figure

Check-in attachment--------------

Footer contains the footer metadata, footer length of 4-bytes, and 'PAR1'. Footer's metadata consists of a version of the format, the schema of the data blocks, any key-value pairs, and the metadata of each column present in the data block.

Save as parquet using PySpark:

Now as we learned the architecture of Parquet, time for some hands-on activity. Let the data be a CSV file format and we read this CSV file as dataframe as shown in the below diagram. Our goal is to save this dataframe with students' data in parquet file format using Pyspark.

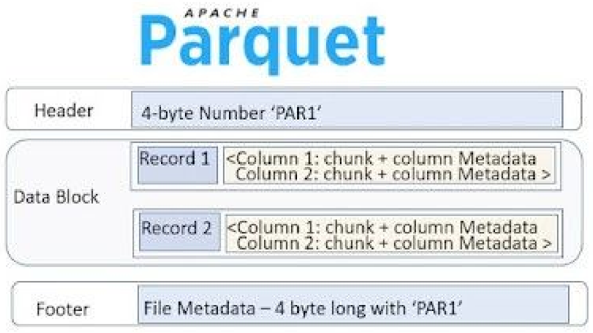
Check-in attachment------------

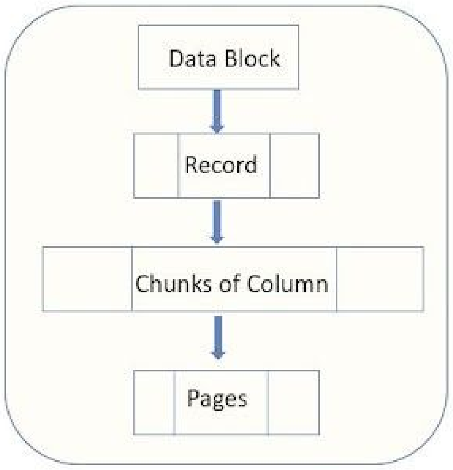
Snippet for saving the dataframe as a parquet file is given below.

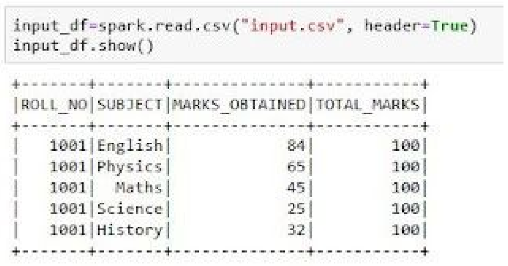
```
#Save

as parquet file

input_df.coalesce(1).write.format('parquet') \

.mode('overwrite') \

.save('out_parq')
```

Coalesce(1) in our snippet will combine all the partitions and results with one single partitioned file written to the target location as a parquet file. From the below figure, one can notice that the file format of data stored is in Parquet.

Note: Parquet File format unlike CSV or textfile format can't be read directly using Hadoop commands. To read the data in Parquet, we need to create a hive table on top of the data or else use the spark command to read the data.

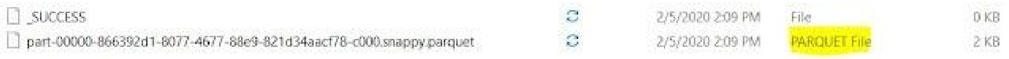

APACHE
Parquet
Header
4-byte Number 'PAR1'
Data Block
Record 1
<Column 1: chunk + column Metadata
Column 2: chunk + column Metadata >
Record 2
<Column 1: chunk + column Metadata
Column 2: chunk + column Metadata >
Footer
File Metadata – 4 byte long with 'PAR1'





```
input_df=spark.read.csv("input.csv", header=True)
input_df.show()
```

```
+-------+-------+--------------+-----------+
|ROLL_NO|SUBJECT|MARKS_OBTAINED|TOTAL_MARKS|
+-------+-------+--------------+-----------+
|   1001|English|            84|        100|
|   1001|Physics|            65|        100|
|   1001|  Maths|            45|        100|
|   1001|Science|            25|        100|
|   1001|History|            32|        100|
+-------+-------+--------------+-----------+
```



_SUCCESS	2/5/2020 2:09 PM	File	0 KB
part-00000-866392d1-8077-4677-88e9-821d34aacf78-c000.snappy.parquet	2/5/2020 2:09 PM	PARQUET File	2 KB

Read the parquet file using PySpark:

As I dictated in the above note, we can't read the parquet data using the Hadoop cat command. we can read either by having a hive table built on top of parquet data or using spark command to read the parquet data. We look at the method of reading parquet files using the spark command. Let us read the file that we wrote as parquet data in the above snippet.

#Read

the parquet file format

read_parquet=spark.read.parquet('out_parq\part*.parquet') read_parquet.show

()

The output of the above snippet will be the data in tabled structure as shown below.

Out[] : Check-in attachement

A real-time testing results with Numbers:

Sqoop import of a table with 13,193,045 records gave the output regular file size of 8.6 GB. but same Sqoop import of the table with the same 13,193,045 records as a parquet file gave an output file with just 1.6 GB which is ~ 500 % storage savings.

Also, we can create hive external tables by referring to this parquet file and also process the data directly from the parquet file.

Although, the time is taken for the sqoop import as a regular file was just 3 mins and for the Parquet file it took 6 mins as 4-part files. I was surprised to see this time duration difference in storing the parquet file. This part on time needs more research.

Bottom line:

Parquet files will for sure help in saving your disk space. if you are tight on your storage, then go for the Parquet file.

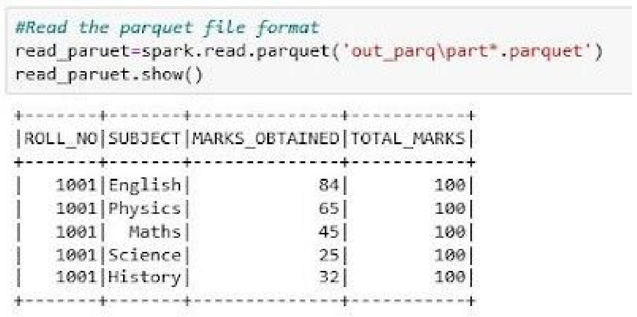
Performance:

As opposed to row-based file formats like CSV, Parquet is optimized for performance. When running queries on your Parquet-based file system, you can focus only on the relevant data very quickly. Moreover, the amount of data scanned will be way smaller and will result in less I/O usage. To understand this, let's look a bit deeper into how Parquet files are structured.

As we mentioned above, Parquet is a self-described format, so each file contains both data and metadata. Parquet files are composed of row groups, header, and footer. Each row group contains data from the same columns. The same columns are stored together in each row group:

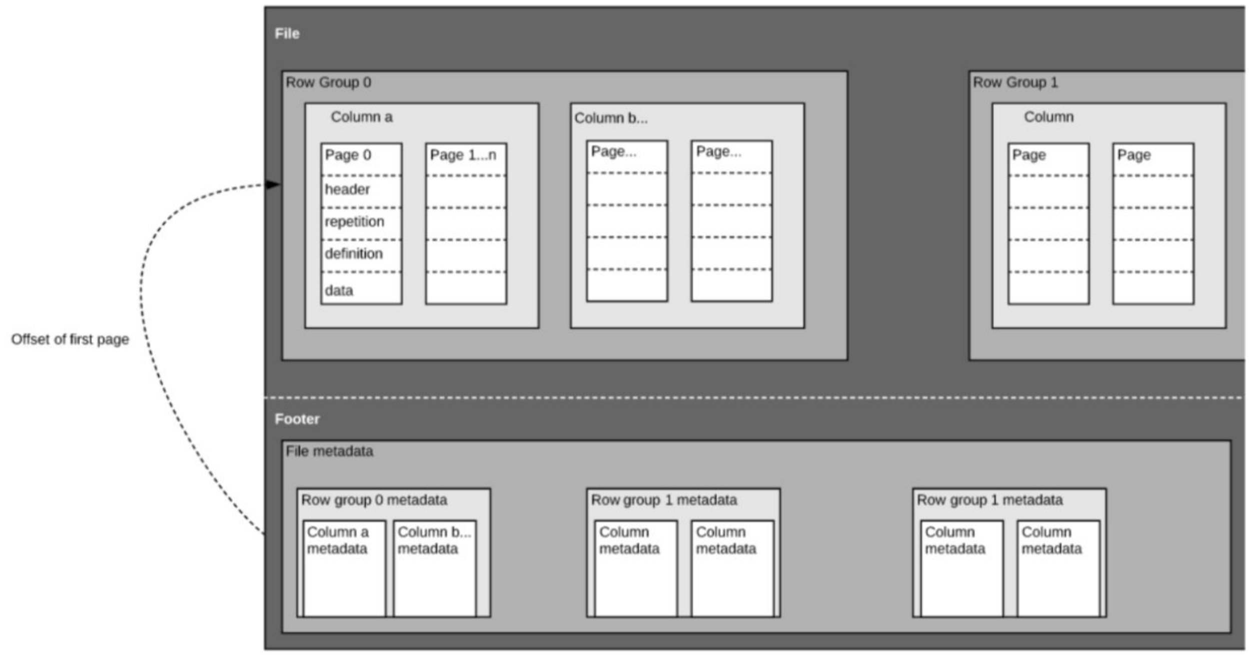
Check-in Attachment-----------

This structure is well-optimized both for fast query performance, as well as low I/O (minimizing the amount of data scanned). For example, if you have a table with 1000 columns, which you will usually only query using a small subset of columns. Using Parquet files will enable you to fetch only the required columns and their values, load those in memory and answer the query. If a row-based file format like CSV was used, the entire table would have to have been loaded in memory, resulting in increased I/O and worse performance.



```
#Read the parquet file format
read_paruet=spark.read.parquet('out_parq\part*.parquet')
read_paruet.show()
```

```
+-------+-------+--------------+-----------+
|ROLL_NO|SUBJECT|MARKS_OBTAINED|TOTAL_MARKS|
+-------+-------+--------------+-----------+
|   1001|English|            84|        100|
|   1001|Physics|            65|        100|
|   1001|  Maths|            45|        100|
|   1001|Science|            25|        100|
|   1001|History|            32|        100|
+-------+-------+--------------+-----------+
```



Schema evolution:

When using columnar file formats like Parquet, users can start with a simple schema, and gradually add more columns to the schema as needed. In this way, users may end up with multiple Parquet files with different but mutually compatible schemas. In these cases, Parquet supports automatic schema merging among these files.

Open source and non-proprietary :

Apache Parquet is part of the open-source Apache Hadoop ecosystem. Development efforts around it are active, and it is being constantly improved and maintained by a strong community of users and developers.

Storing your data in open formats means you avoid vendor lock-in and increase your flexibility, compared to proprietary file formats used by many modern high-performance databases. This means you can use various query engines such as Amazon Athena, Qubole, and Amazon Redshift Spectrum, within the same data lake architecture, rather than being tied down to a specific database vendor.

Apache Parquet Use Cases – When Should You Use It?

While this isn’t a comprehensive list, a few tell-tale signs that you should be storing data in Parquet include:

1)When you’re working with very large amounts of data. Parquet is built for performance and effective compression. Various benchmarking tests that have compared processing times for SQL queries on Parquet vs formats such as Avro or CSV (including the one described in this article, as well as this one), have found that querying Parquet results in significantly speedier queries.

2)When your full dataset has many columns, but you only need to access a subset. Due to the growing complexity of the business data, you are recording, you might find that instead of collecting 20 fields for each data event you’re now capturing 100+. While this data is easy to store in a data lake, querying it will require scanning a significant amount of data if stored in row-based formats. Parquet’s columnar and self-describing nature allows you to only pull the required columns needed to answer a specific query, reducing the amount of data processed.

When you want multiple services to consume the same data from object storage. While database vendors such as Oracle and Snowflake prefer you store your data in a proprietary format that only their tools can read, modern data architecture is biased towards decoupling storage from computing. If you want to work with multiple analytics services to answer different use cases, you should store data in Parquet.

Parquet File Format - Advantages:

· Nested data: Parquet file format is more capable of storing nested data, i.e., if data stored in HDFS have more hierarchy then parquet serves the best as it stores the data in a tree structure.

· Predicate Pushdown efficiency: Parquet is very useful in query optimization. Suppose, if you have to apply some filtering logic to the huge amount of data, then Spark Catalyst optimizer takes advantage of file format and push down lot of calculation to file format. Parquet as a part of metadata information will store some stats such as min, max, avg value of the partitions, which makes the query fetch data from metadata itself.

· Compression: Parquet is more compression efficient data storage than other file formats such as Avro, JSON, and CSV, etc.

· Less disk IO: Parquet with compression reduces your data storage by 75% on average, i.e., your 1TB scale factor data files will materialize only about 250 GB on disk. This reduces significantly input data needed for your Spark SQL applications. But in Spark 1.6.0, Parquet readers used push-down filters to further reduce disk IO. Push-down filters allow early data selection decisions to be made before data is even read into Spark.

· Higher scan throughput in Spark 1.6.0: The Databricks' Spark 1.6.0 release blog mentioned significant Parquet scan throughput because a "more optimized code path" is used. To show this in the real world, we ran query 97 in Spark 1.5.1 and in 1.6.0 and captured on data. The improvement is very obvious.

· Efficient Spark execution graph: In addition to smarter readers such as in Parquet, data formats also directly impact Spark execution graph because one major input to the scheduler is RDD count.

· Spark SQL is much faster with Parquet! The chart below compares the sum of all execution times of the 24 queries running in Spark 1.5.1. Queries taking about 12 hours to complete using flat CVS files vs. taking less than 1 hour to complete using Parquet, an 11X performance improvement.

· Spark SQL works better at large-scale with Parquet: Poor choice of storage format often cause exceptions that are difficult to diagnose and fix. At 1TB scale factor, for example, at least 1/3 of all runnable queries failed to complete using flat CSV files, but they all completed using Parquet files.

· Data security as Data is not human-readable

· Low storage consumption

· Efficient in reading Data in less time as it is columnar storage and minimizes latency.

· Supports advanced nested data structures. Optimized for queries that process large volumes of data.

· Parquet has a higher execution speed compared to other standard file formats like Avro, JSON, etc and it also consumes less disk space in comparison to AVRO and JSON.

Question 17: How to create Spark Session in different Programming languages(Mostly Pyspark(python), Scala)

SparkSession - New entry point of Spark :

In earlier versions of spark, spark context was the entry point for Spark. As RDD was the main API, it was created and manipulated using context API's. For every other API, we needed to use different contexts. For streaming, we needed StreamingContext, SQL sqlContext, and hive HiveContext. But as DataSet and Dataframe APIs are becoming new standard API's we need an entry point build for them. So, in Spark 2.0, we have a new entry point for DataSet and Dataframe APIs called as Spark Session.

SparkSession is essentially combination of SQLContext, HiveContext and future StreamingContext. All the API's available on those contexts are available on spark session also. Spark session internally has a spark context for actual computation.

Creating SparkSession with Scala :

SparkSession follows a builder factory design pattern. Below is the code to create a spark session.

```
val sparkSession = SparkSession.builder().
master("local")
.appName("spark session example")
.getOrCreate()
```

The above is similar to creating a SparkContext with local and creating an SQLContext wrapping it. If you need to create, a hive context you can use the below code to create a spark session with hive support.

```
val sparkSession = SparkSession.builder().
master("local")
.appName("App")
.enableHiveSupport()
.getOrCreate()
```

enableHiveSupport on factory enables hive support which is similar to HiveContext.

master() – If you are running it on the cluster you need to use your master name as an argument to master(). usually, it would be either yarn or mesos depending on your cluster setup.

Use local[x] when running in Standalone mode. x should be an integer value and should be greater than 0; this represents how many partitions it should create when using RDD, DataFrame, and Dataset. Ideally, the x value should be the number of CPU cores you have.

appName() – Used to set your application name.

getOrCreate() – This returns a SparkSession object if already exists, creates a new one if not exist.

Creating SparkSession with Pyspark:

Be default PySpark shell provides a "spark" object; which is an instance of SparkSession class. We can directly use this object where required in spark-shell. Start your "pyspark" shell from the $SPARK_HOME\bin folder and enter the below statement.

```
import pyspark
from pyspark.sql import SparkSession
spark = SparkSession.builder.master("local[1]") \
.appName('App') \
.getOrCreate()
```

Note: SparkSession object “spark” is by default available in the PySpark shell.

You can also create a new SparkSession using the newSession() method.

import pyspark

from pyspark.sql import SparkSession

sparkSession3 = SparkSession.newSession

This always creates a new SparkSession object.

Question 18: write a program to separate 1 to one side and 0 to one side in a binary string python.

Segregate 0s and 1s in an array

Given an array of 0s and 1s in random order. Segregate 0s on the left side and 1s on the right side of the array [Basically you have to sort the array]. Traverse array only once.

Input array = [0, 1, 0, 1, 0, 0, 1, 1, 1, 0]

Output array = [0, 0, 0, 0, 0, 1, 1, 1, 1, 1]

Method 1 (Count 0s or 1s)

1) Count the number of 0s. So, let’s understand with an example we have an array arr = [0, 1, 0, 1, 0, 0, 1] the size of the array is 7 now we will traverse the entire array and find out the number of zeros in the array, in this case, the number of zeros is 4 so now we can easily get the number of Ones in the array-by-Array Length – Number of Zeros.

2) Once we have counted, we can fill the array first we will put the zeros and then ones (we can get a number of ones by using the above formula).

Time Complexity: O(n)

```
def segregate0and1(arr, n) :

           # Counts the no of zeros in arr
           count = 0

           for i in range(0, n) :
                      if (arr[i] == 0) :
                                 count = count + 1

           # Loop fills the arr with 0 until count
           for i in range(0, count) :
                      arr[i] = 0

           # Loop fills remaining arr space with 1
           for i in range(count, n) :
                      arr[i] = 1


# Function to print segregated array
def print_arr(arr , n) :
           print( "Array after segregation is ",end = "")

           for i in range(0, n) :
                      print(arr[i] , end = " ")
```

```
# Driver function
arr = [ 0, 1, 0, 1, 1, 1 ]
n = len(arr)

segregate0and1(arr, n)
print_arr(arr, n)
```

Output:

Array after segregation is 0 0 1 1 1 1

Method 1 traverses the array two times. Method 2 does the same in a single pass.

Method 2 (Use two indexes to traverse)

Maintain two indexes. Initialize the first index left as 0 and the second index right as n-1.

Do the following while left < right

a) Keep incrementing index left while there are 0s at it

b) Keep decrementing index right while there are 1s at it

c) If left < right then exchange arr[left] and arr[right]

```
# Python program to sort a binary array in one pass
# Function to put all 0s on left and all 1s on right
def segregate0and1(arr, size):
	# Initialize left and right indexes
	left, right = 0, size-1
	while left < right:
		# Increment left index while we see 0 at left
		while arr[left] == 0 and left < right:
			left += 1
		# Decrement right index while we see 1 at right
		while arr[right] == 1 and left < right:
			right -= 1
		# If left is smaller than right then there is a 1 at left
		# and a 0 at right. Exchange arr[left] and arr[right]
		if left < right:
			arr[left] = 0
			arr[right] = 1
			left += 1
			right -= 1
	return arr
# driver program to test
arr = [0, 1, 0, 1, 1, 1]
arr_size = len(arr)
print("Array after segregation")
print(segregate0and1(arr, arr_size))
```

Output:

Array after segregation is 0 0 1 1 1 1

Time Complexity: O(n)

Another approach:

1. Take two pointer type0(for element 0) starting from beginning (index = 0) and type1(for element 1) starting from end (index = array.length-1).

Initialize type0 = 0 and type1 = array.length-1

2. It is intended to Put 1 on the right side of the array. Once it is done, then 0 will definitely be towards the left side of the array.

```
# Python program to sort a
# binary array in one pass
# Function to put all 0s on
# left and all 1s on right
def segregate0and1(arr, size):
           type0 = 0
           type1 = size - 1
           while(type0 < type1):
                       if(arr[type0] == 1):
                                   (arr[type0],
                                   arr[type1]) = (arr[type1],
                                                           arr[type0])
                                   type1 -= 1
                       else:
                                   type0 += 1
# Driver Code
arr = [0, 1, 0, 1, 1, 1]
arr_size = len(arr)
segregate0and1(arr, arr_size)
print("Array after segregation is", end = " ")
for i in range(0, arr_size):
                       print(arr[i], end = " ")
```

Output:

Array after segregation is 0 0 1 1 1 1

Time complexity: O(n)

Another approach:

Algorithm:

seg0s1s(A)

/* A is the user input Array and the element of A should be the combination of 0’s and 1’s */

Step 1: First traverse the array.

Step 2: Then check every element of the Array. If the element is 0, then its position is left side, and if 1 then it is on the right side of the array.

Step 3: Then concatenate two lists.

```
# Segregate 0's and 1's in an array list

def seg0s1s(A):
n = ([i for i in A if i==0] + [i for i in A if i==1])
print(n)

# Driver program
if __name__ == "__main__":
A=list()
n=int(input("Enter the size of the array ::"))
```

```
print("Enter the number ::")
for i in range(int(n)):
k=int(input(""))
A.append(int(k))
print("The New ArrayList ::")
seg0s1s(A)
```

Output:

Enter the size of the array::6

Enter the number::

1

0

0

1

1

0

The New ArrayList ::

[0, 0, 0, 1, 1, 1]

Another approach:

```
arr=[]
size = int(input("Enter the size of the array: "))

print("Enter the Element of the array(only 0s and 1s):")

for i in range(0,size):

num = int(input())

arr.append(num)

c=0

for i in range(0,size):

if arr[i]==0:

c+=1

for i in range(0,c):

arr[i]=0

for i in range(c,size):

arr[i]=1

print("After segregate 0s and 1s in an Array, Array is:")

print(arr)
```

Output:

```
Enter the size of the array: 3
Enter the Element of the array (only 0s and 1s):
0
1
0
After segregate 0s and 1s in an Array, Array is [0, 0, 1]
```

Last Approach:

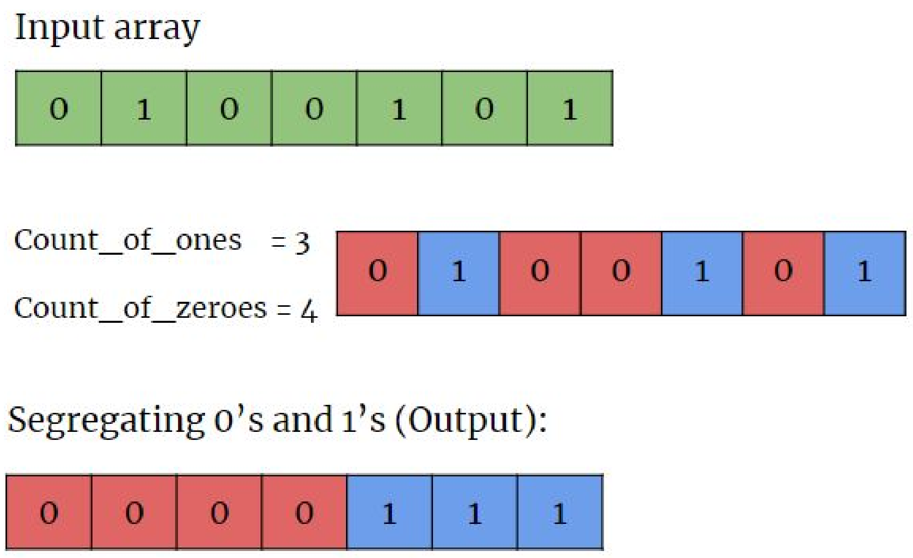
For the given array, find the sum of the array which gives the total count of 1’s in the array. Now, subtract that value from the size of the array, which gives a count of 0’s in the array. Print 0’s and 1’s with respect to their count values respectively.

```
def segregateOperation(array):
   total_sum,no_of_zeroes=0,0
   n=len(array)
   for i in array:
       total_sum+=i
   print("Array after Segregation of 0's and 1's: ",end=" ")
   no_of_zeroes=n-total_sum
   for i in range(no_of_zeroes):
       print(0,end=" ")
   for i in range(no_of_zeroes,n):
       print(1,end=" ")
if __name__=='__main__':
   array=[0,1,1,0,1,0,1]
   segregateOperation(array)
```

Output:

```
Array after segregation is: 0 0 0 0 1 1 1
```

Time complexity: O(n)



Question 19: Optimization Techniques in Spark

1)Data filtering as early as possible

2)File format selection

3)API Selection

4)Use of advance variables(Using Accumulators)

5)Parallelism using Coalesce/Repartition

6)Data Serialization

7)Caching and Persistence

8)Reduce Shuffle Operation

9)Setting up a limit for Broadcast Join

10)Compressing data while SQL execution

11)Memory and Resource Allocation

12)Hive Bucketing Performance

13)Predicate Pushdown Optimization

14)Zero Data Serialization/Deserialization using Apache Arrow

15)Garbage Collection Tuning using G1GC Collection

16)Memory Management and Tuning

17)Data Locality

18)Using Collocated Joins

19)Executor Size

20)Spark Windowing Function

21)Watermarks Technique

Mistakes to avoid while Optimising Apache Spark:

1)reduceByKey or groupByKey

2)Maintain the required size of the shuffle blocks

3)File Formats and Delimiters

4)Small Data Files

5)No Monitoring of Job Stages

6)ByKey, repartition, or any other operations which trigger shuffles

7)Reinforcement Learning

8)Serialization

9)Try to keep away from Skews as well as partitions too

10)Executor Resource Sizing

11)DAG Management

12)Shading

13)Try not to shuffle more

14)Reduce should be lesser than TreeReduce

15)Always try to lower the side of maps as much as possible

If anything misses from my end... please comment on your techniques. I can explain these things in the coming days...

Question 20: Different Joins in Spark with Examples (Dataframe with Pyspark).

Spark Join Types:

Like SQL, there is a variety of join types available in spark.

1)Inner Join – Keeps data from left and right data frame where keys exist in both

2)Outer join – keeps data from left and right data frame where keys exist in either left or right data frame

3)Let outer join – keeps data with keys in a left data frame

4)Right outer join – keeps data with keys in a right data frame

5)Left semi-join – Only gets data from the left data frame for which we have a matching key in the right data frame

6)Left anti join – Only gets data from the right data frame for which we do not have any matching key in the right data frame

7)Natural join – Joins two data frames with the same column names

8)Cross Join – joins every row from the left data frame with every other row in the right data frame

Now that we know all spark join types let us learn these using code and example data to understand them in a better way.

Setting Up Data:

Before we start, we will need data frames on which we can test join types. Instead of reading a lot of data, in this case, we will set up small data frames which we can easily validate.

Emp Dataframe:

```
emp = spark.createDataFrame([
(0,"Jack C", "2000-01-01", "D101"),
(1,"Teddy D", "2002-02-01", "D102"),
(2,"Maxi", "2003-04-01", "D104"),
(3,"Perkar J", "2005-06-01", "D104"),
(4,"Marky P", "2007-01-01", "D1022")
]).toDF("id","name","joining_date","dept_id")
emp.show
()
+---+-------+------------+-------+
| id|  name|joining_date|dept_id|
+---+-------+------------+-------+
| 0| Jack C| 2000-01-01|  D101|
| 1| Teddy D| 2002-02-01|  D102|
| 2|   Maxi| 2003-04-01|  D104|
| 3|Perkar J| 2005-06-01|  D104|
| 4| Marky P| 2007-01-01| D1022|
+---+-------+------------+-------+
```

Dept Dataframe:

```
dept = spark.createDataFrame([
("D101", "Support"),
("D102", "HR"),
("D103", "Marketing"),
("D104", "Sells")
]).toDF("id", "dept_name")
dept.show
()
```

```
+----+---------+
| id|dept_name|
+----+---------+
|D101| Support|
|D102|      HR|
|D103|Marketing|
|D104|   Sells|
+----+---------+
```

We have set up two simple data frames, one with employees and one for departments. We have department id as a common column between these two. Now, this is set up let us start with spark joins.

Inner Join:

Inner join returns data from the left and right data frame where the join key is present in both data frames. For example, it will return data from employee and department data frame where the same department id is present in both of them. Let us see that as an example.

```
emp.join(dept, emp["dept_id"] == dept["id"], "inner").show()
+---+-------+------------+-------+----+---------+
| id|  name|joining_date|dept_id| id|dept_name|
+---+-------+------------+-------+----+---------+
| 2|   Maxi| 2003-04-01|  D104|D104|   Sells|
| 3|Perkar J| 2005-06-01|  D104|D104|   Sells|
| 0| Jack C| 2000-01-01|  D101|D101| Support|
| 1| Teddy D| 2002-02-01|  D102|D102|      HR|
+---+-------+------------+-------+----+---------+
```

We can also format this code for better understanding by making join conditions and join types separate variables.

```
joinType = "inner"
joinCondition = emp["dept_id"] == dept["id"]
emp.join(dept, joinCondition, joinType).show()
```

By default, spark performs inner join. So when we want to do an inner join we can skip join type parameter and we will still receive the same result.

```
emp.join(dept, joinCondition).show()
+---+-------+------------+-------+----+---------+
| id|  name|joining_date|dept_id| id|dept_name|
+---+-------+------------+-------+----+---------+
| 2|   Maxi| 2003-04-01|  D104|D104|   Sells|
| 3|Perkar J| 2005-06-01|  D104|D104|   Sells|
| 0| Jack C| 2000-01-01|  D101|D101| Support|
| 1| Teddy D| 2002-02-01|  D102|D102|      HR|
+---+-------+------------+-------+----+---------+
```

Outer Join:

In the case of outer joins, Spark will take data from both data frames. If the matching key is not present in the left or right data frame, Spark will put “null” for that data.

```
joinType = "outer"
joinCondition = emp["dept_id"] == dept["id"]
emp.join(dept, joinCondition, joinType).show()
+----+-------+------------+-------+----+---------+
| id|  name|joining_date|dept_id| id|dept_name|
+----+-------+------------+-------+----+---------+
|  2|   Maxi| 2003-04-01|  D104|D104|   Sells|
|  3|Perkar J| 2005-06-01|  D104|D104|   Sells|
|  4| Marky P| 2007-01-01| D1022|null|    null|
|null|  null|       null|  null|D103|Marketing|
|  0| Jack C| 2000-01-01|  D101|D101| Support|
|  1| Teddy D| 2002-02-01|  D102|D102|      HR|
+----+-------+------------+-------+----+---------+
```

We can see that spark has picked all rows from both data frames and where it did not find a matching key like department id “D1022” it has inserted null for data coming from departments.

Left Outer Join:

Spark will pick all rows from a left data frame. For keys matching from the right data frame, it will get that data. And if there is no matching key in the right data frame, it will insert null.

```
joinType = "left_outer"
joinCondition = emp["dept_id"] == dept["id"]
emp.join(dept, joinCondition, joinType).show()
+---+-------+------------+-------+----+---------+
| id|  name|joining_date|dept_id| id|dept_name|
+---+-------+------------+-------+----+---------+
| 2|   Maxi| 2003-04-01|  D104|D104|   Sells|
| 3|Perkar J| 2005-06-01|  D104|D104|   Sells|
| 4| Marky P| 2007-01-01| D1022|null|    null|
| 0| Jack C| 2000-01-01|  D101|D101| Support|
| 1| Teddy D| 2002-02-01|  D102|D102|      HR|
+---+-------+------------+-------+----+---------+
```

Here only data from the employee data frame is selected and for department id “D1022” as there is no matching record in the department’s data frame, Spark has put null for that record.

Right Outer Join:

Like left outer join, the spark will only pick records from the right data frame while doing a right outer join. If there is no matching key in the left data frame, Spark will insert null.

```
joinType = "right_outer"

joinCondition = emp["dept_id"] == dept["id"]

emp.join(dept, joinCondition, joinType).show()

+----+-------+------------+-------+----+---------+
| id|  name|joining_date|dept_id| id|dept_name|
+----+-------+------------+-------+----+---------+
|  2|   Maxi| 2003-04-01|  D104|D104|   Sells|
|  3|Perkar J| 2005-06-01|  D104|D104|   Sells|
|null|  null|       null|  null|D103|Marketing|
|  0| Jack C| 2000-01-01|  D101|D101| Support|
|  1| Teddy D| 2002-02-01|  D102|D102|      HR|
+----+-------+------------+-------+----+---------+
```

Left Semi Joins:

In case of left semi joins, Spark only picks data from left data frame for which it finds a common key in right data frame. Like most join types, this is better understood with an example.

```
joinType = "left_semi"

joinCondition = emp["dept_id"] == dept["id"]

emp.join(dept, joinCondition, joinType).show()

+---+-------+------------+-------+
| id|  name|joining_date|dept_id|
+---+-------+------------+-------+
| 2|   Maxi| 2003-04-01|  D104|
| 3|Perkar J| 2005-06-01|  D104|
| 0| Jack C| 2000-01-01|  D101|
| 1| Teddy D| 2002-02-01|  D102|
+---+-------+------------+-------+
```

We can notice that spark has data only from the employee data frame and it has not picked employee id 4 as department id “D1022” is not present in the department data frame.

Left Anti Join:

Here, Spark picks data from left data frame only for those rows where it does not find a common key in right data frame. For our example, it should only pick employee id 4 as it does not have a matching dept_id in the department's data frame.

```
joinType = "left_anti"

joinCondition = emp["dept_id"] == dept["id"]

emp.join(dept, joinCondition, joinType).show()

+---+------+------------+-------+

| id| name|joining_date|dept_id|

+---+------+------------+-------+

| 4|Marky P| 2007-01-01| D1022|

+---+------+------------+-------+
```

Natural Join:

In the case of Natura joins, spark joins columns with the same name and performs an inner join, and returns us the result. Using implicate join condition is always dangerous. Like in our example, the spark will join employee id to department id which is clearly wrong.

```
emp.createOrReplaceTempView("emp")

dept.createOrReplaceTempView("dept")

spark.sql("select * from emp NATURAL JOIN dept").show()

+---+----+------------+-------+---------+

| id|name|joining_date|dept_id|dept_name|

+---+----+------------+-------+---------+

+---+----+------------+-------+---------+

# it is not showing any rows as the id in each data frame means different things.
```

In most cases, we should avoid using this join and specify our own joining condition.

Cross Join:

In Cross join, Sparks joins each row from the left data frame with each row in the right data frame. This will result in huge data created in the cluster. We should be careful while using cross join as it can break our application.

```
spark.sql("select * from emp CROSS JOIN dept").show()

+---+-------+------------+-------+----+---------+

| id|  name|joining_date|dept_id| id|dept_name|

+---+-------+------------+-------+----+---------+

| 0| Jack C| 2000-01-01|  D101|D101| Support|

| 0| Jack C| 2000-01-01|  D101|D102|      HR|
| 0| Jack C| 2000-01-01|  D101|D103|Marketing|
| 0| Jack C| 2000-01-01|  D101|D104|   Sells|
| 1| Teddy D| 2002-02-01|  D102|D101| Support|
| 1| Teddy D| 2002-02-01|  D102|D102|      HR|
| 1| Teddy D| 2002-02-01|  D102|D103|Marketing|
| 1| Teddy D| 2002-02-01|  D102|D104|   Sells|
| 2|   Maxi| 2003-04-01|  D104|D101| Support|
| 2|   Maxi| 2003-04-01|  D104|D102|      HR|
| 2|   Maxi| 2003-04-01|  D104|D103|Marketing|
| 2|    Maxi| 2003-04-01|  D104|D104|   Sells|
| 3|Perkar J| 2005-06-01|  D104|D101| Support|
| 3|Perkar J| 2005-06-01|  D104|D102|      HR|
| 3|Perkar J| 2005-06-01|  D104|D103|Marketing|
| 3|Perkar J| 2005-06-01|  D104|D104|   Sells|
| 4| Marky P| 2007-01-01| D1022|D101| Support|
| 4| Marky P| 2007-01-01| D1022|D102|      HR|
| 4| Marky P| 2007-01-01| D1022|D103|Marketing|
| 4| Marky P| 2007-01-01| D1022|D104|   Sells|
+---+-------+------------+-------+----+---------+
```

Cross join has created so many rows for our small data frames. You can imagine what will happen if you do cross join on very large data. So be careful when using this join type.