Part 2

Question 21: Difference between Client mode & Cluster mode in Spark
Question 22 : Explain the MPP Architecture in Spark.
Question 23: Finding Nth Lowest Salary per Department using py-spark and SQL.
Question 24: Explain Broadcast Join in Spark.
Question 25: Explain about Sort-Merge Join in Spark.
Question 26: Explain About Serialization in Spark?
Question 27: Pivot, Unpivot Data with SparkSQL & PySpark – Databricks
Question 28: Pyspark SQL Functions:
Question 29: Optimizations in Azure Databricks. (Now its time to focus on Azure with Bigdata)

Question 21: Difference between Client mode & Cluster mode in Spark

Whenever we submit a Spark application to the cluster, the Driver or the Spark App Master should get started. And the Driver will be starting N number of workers. Spark driver will be managing spark context objects to share the data and coordinates with the workers and cluster manager across the cluster. Cluster Manager can be Spark Standalone or Hadoop YARN or Mesos. Workers will be assigned a task and it will consolidate and collect the result back to the driver. A spark application gets executed within the cluster in two different modes – one is cluster mode and the second is client mode.

Let’s try to understand the Difference Between Spark Cluster & Client Deploy Modes. It is a confusing concept for many. We will see if we can understand it and get clarified.

But first some Basics. Spark has a notion of a Worker or Slave node(s) which is used for computation. Each such nodes have N no. of Executor processes running on it. If Spark assigns a driver to be run on such an arbitrary Worker or Slave node that doesn’t mean that the Worker\Slave node can’t run additional Executor processes to do any computation task.

But please note Spark doesn’t select a Master node – A master node is fixed in the environment. What spark does is choose – were to run the driver, which is where the SparkContext will live for the lifetime of the app.

For any Spark job, the Deployment mode is indicated by the flag deploy-mode which is used in a spark-submit command. It determines whether the spark job will run in cluster or client mode. Let's see what these two modes mean –

·— deploy-mode cluster –

o In cluster deploy mode, all the slave or worker-nodes act as an Executor.

o But one of them will act as Spark Driver too. This means is where the SparkContext will live for the lifetime of the app.

o This basically means One specific node will submit the JAR (or .py file) and we can track the execution using web UI. However, note this Particular node will also act as an executor at the same time.

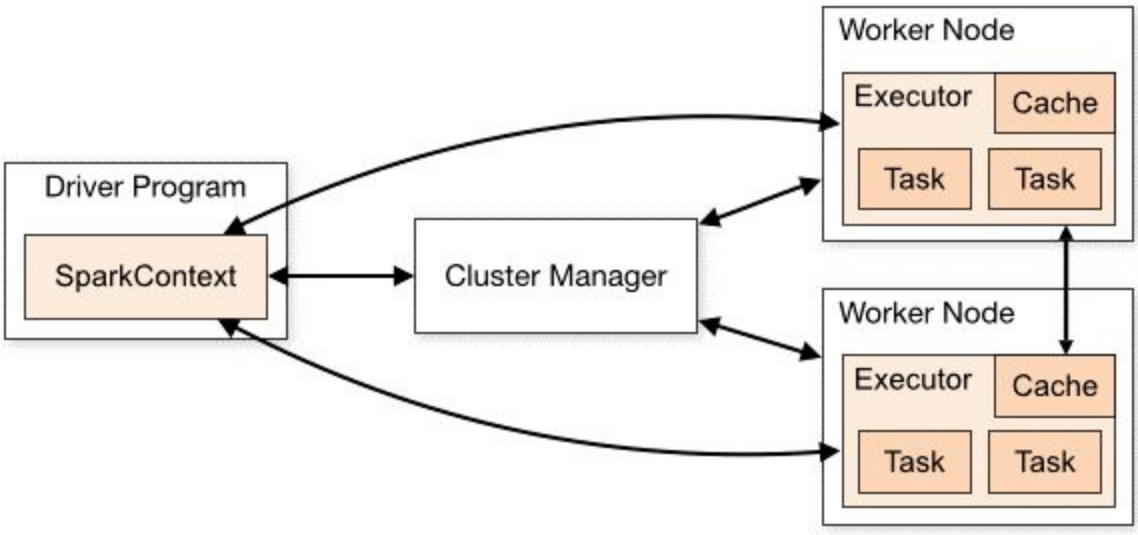
· — deploy-mode client –

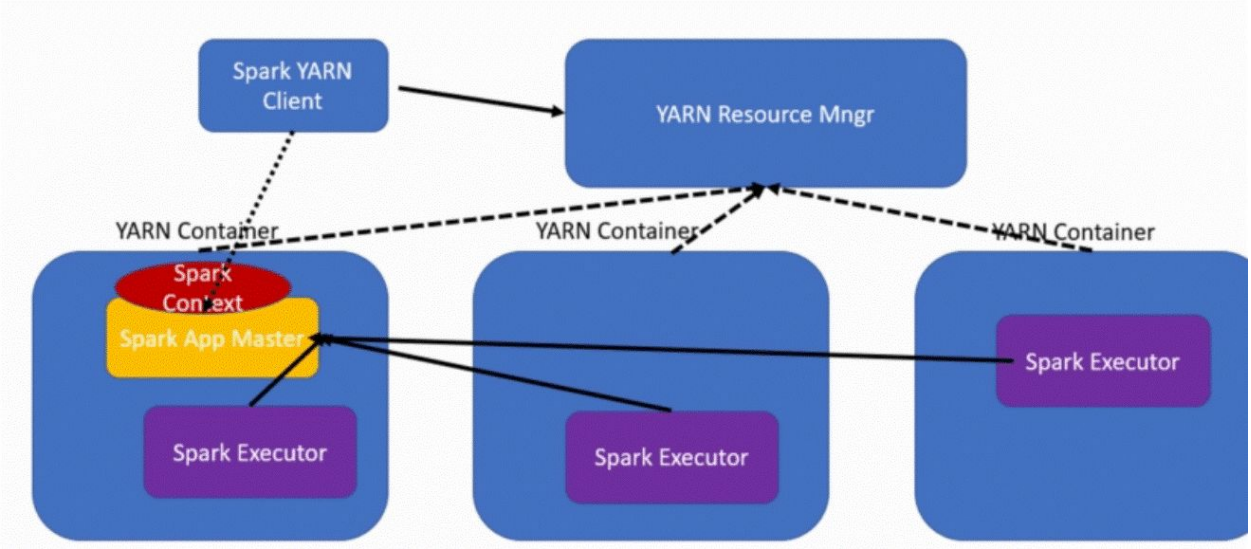
o In client mode, the node where the spark-submit is invoked, will act as the Spark driver. This means is where the SparkContext will live for the lifetime of the app.

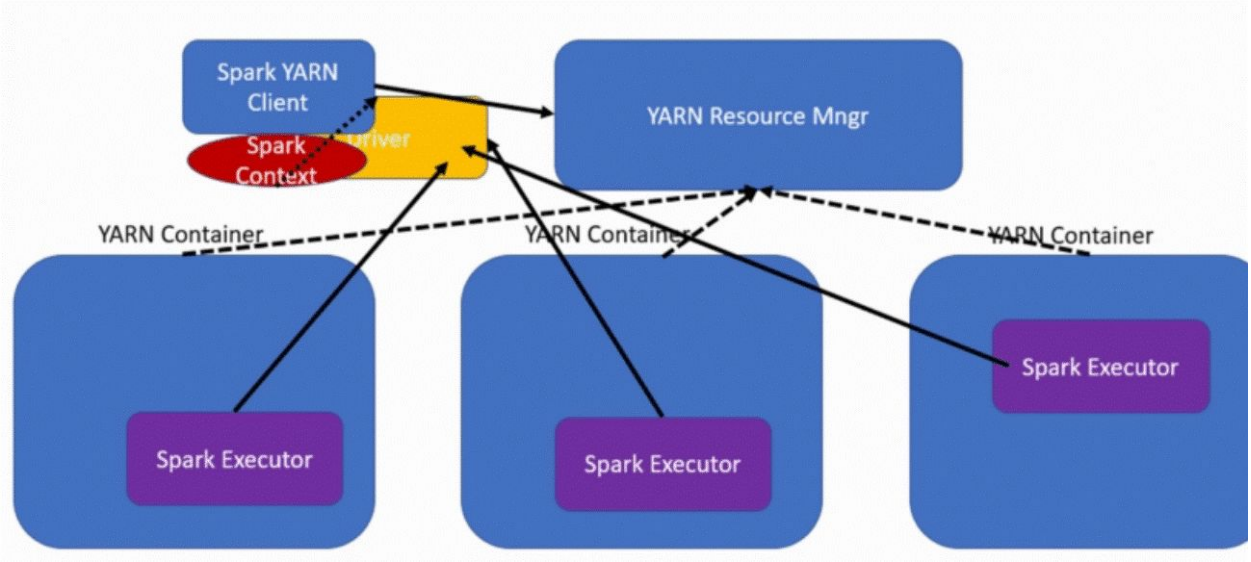
o But this node WILL NOT execute the DAG as this it is designated JUST as a driver for the Spark cluster.

o However, all the other nodes will act as executors for running the job.

o We can track the execution of the jobs through the Web UI.







How to submit spark application in cluster mode:

First, go to your spark installed directory and start a master and any number of workers on a cluster using commands:

./sbin/

start-master.sh

./sbin/

start-slave.sh

spark://<<hostname/ipaddress>>:portnumber - worker1

./sbin/

start-slave.sh

spark://<<hostname/ipaddress>>:portnumber - worker2

Then, run command:

./bin/spark-submit --class org.apache.spark.examples.SparkPi --master spark://<<hostname/ipaddress>>:portnumber --deploy-mode cluster ./examples/jars/spark-examples_2.11-2.3.1.jar 5(number of partitions)

NOTE: Your class name, Jar File, and partition number could be different.

spark-submit --class org.apache.spark.examples.SparkApp \

--master yarn-client \

--deploy-mode cluster \

--num-executors 10 \

--driver-memory 512m \

--executor-memory 512m \

--executor-cores 1 \

/sample/jars/spark-app*.jar 10

How to submit spark application in client mode:

First, go to your spark installed directory and start a master and any number of workers on a cluster. Commands are mentioned above in Cluster mode. Then run the following command:

./bin/spark-submit --class org.apache.spark.examples.SparkPi --master spark://<<hostname/ipaddress>>:portnumber --deploy-mode client ./examples/jars/spark-examples_2.11-2.3.1.jar 5(number of partitions)

Meanwhile, it requires only change in deploy-mode which is the client in Client mode and cluster in Cluster mode.

spark-submit --class org.apache.spark.examples.SparkApp \

- -master yarn-client \
- -deploy-mode client \

- -num-executors 10 \
- -driver-memory 512m \
- -executor-memory 512m \
- -executor-cores 1 \

/sample/jars/spark-app*.jar 10

Question 22 : Explain the MPP Architecture in Spark.

Massively Parallel Processing:

A massively parallel processing (MPP) system consists of a large number of small homogeneous processing nodes interconnected via a high-speed network. The processing nodes in an MPP machine are independent—they typically do not share a memory, and typically each processor may run its own instance of an operating system, although there may be systemic controller applications hosted on leader processing nodes that instruct the individual processing nodes in the MPP configuration on the tasks to perform.

Nodes on MPP machines may also be connected directly to their own I/O devices, or I/O may be channelled into the entire system via high-speed interconnects. Communication between nodes is likely to occur in a coordinated fashion, where all nodes stop processing and participate in an exchange of data across the network, or in an uncoordinated fashion, with messages targeted for specific recipients being injected into the network independently.

Because data can be streamed through the network and targeted for specific nodes, an MPP machine is nicely suited for data-parallel applications. In this case, all processors execute the same program on different data streams. In addition, because individual processors can execute different programs, an MPP machine is nicely suited to coarse-grained parallelism and can be configured for pipelined execution as well.

Multiprocessing:

Even if a computer has more than one processor physically installed, it might not be able to perform multiprocessing. To perform multiprocessing, the operating system must be capable of recognizing the presence of multiple processors and be able to use them.

Some operating systems—such as Windows 9x—do not support multiprocessing. Even among those that do, not all multiprocessing operating systems are created equal.

There are three methods of supporting multiple processing:

- Asymmetric multiprocessing (AMP or ASMP)
- Symmetric multiprocessing (SMP)
- Massively parallel processing (MPP)

With asymmetric multiprocessing, each processor is assigned specific tasks. One primary processor act as the “master” and controls the actions of the other, secondary processors.

Symmetric multiprocessing makes all the processors available to all individual processes. The processors share the workload, distributed more or less equally, thus increasing performance. Symmetric multiprocessing is also called tightly coupled multiprocessing because the multiple processors still use just one instance of the operating system and share the computer's memory and I/O resources.

Massively parallel processing is a means of crunching huge amounts of data by distributing the processing over hundreds or thousands of processors, which might be running in the same box or in separate, distantly located computers. Each processor in an MPP system has its own memory, disks, applications, and instances of the operating system. The problem being worked on is divided into many pieces, which are processed simultaneously by multiple systems.

MPP processors can have up to 200 or more processors working on the application and most commonly communicate using a messaging interface. MPP works by allowing messages to be sent between processes through an “interconnect” arrangement of data paths.

There are several types of MPP database architectures, each with its own benefits:

Grid computing– uses multiple computers in distributed networks. This type of architecture uses use resources opportunistically based on their availability. This architecture reduces costs for server space but also limits bandwidth and capacity at peak times or when there are too many requests.

Computer clustering – links the available power into nodes that can connect with each other to handle multiple tasks at once.

Advantages of MPP databases include:

1) Allowing more people in an organization to run their own data analyses and queries simultaneously without experiencing lag or longer response times.

2) Centralizing data in a single location.

3) Making it easier to uncover insights, and build dashboards that contain more relevant information than those built from data that is fragmented.

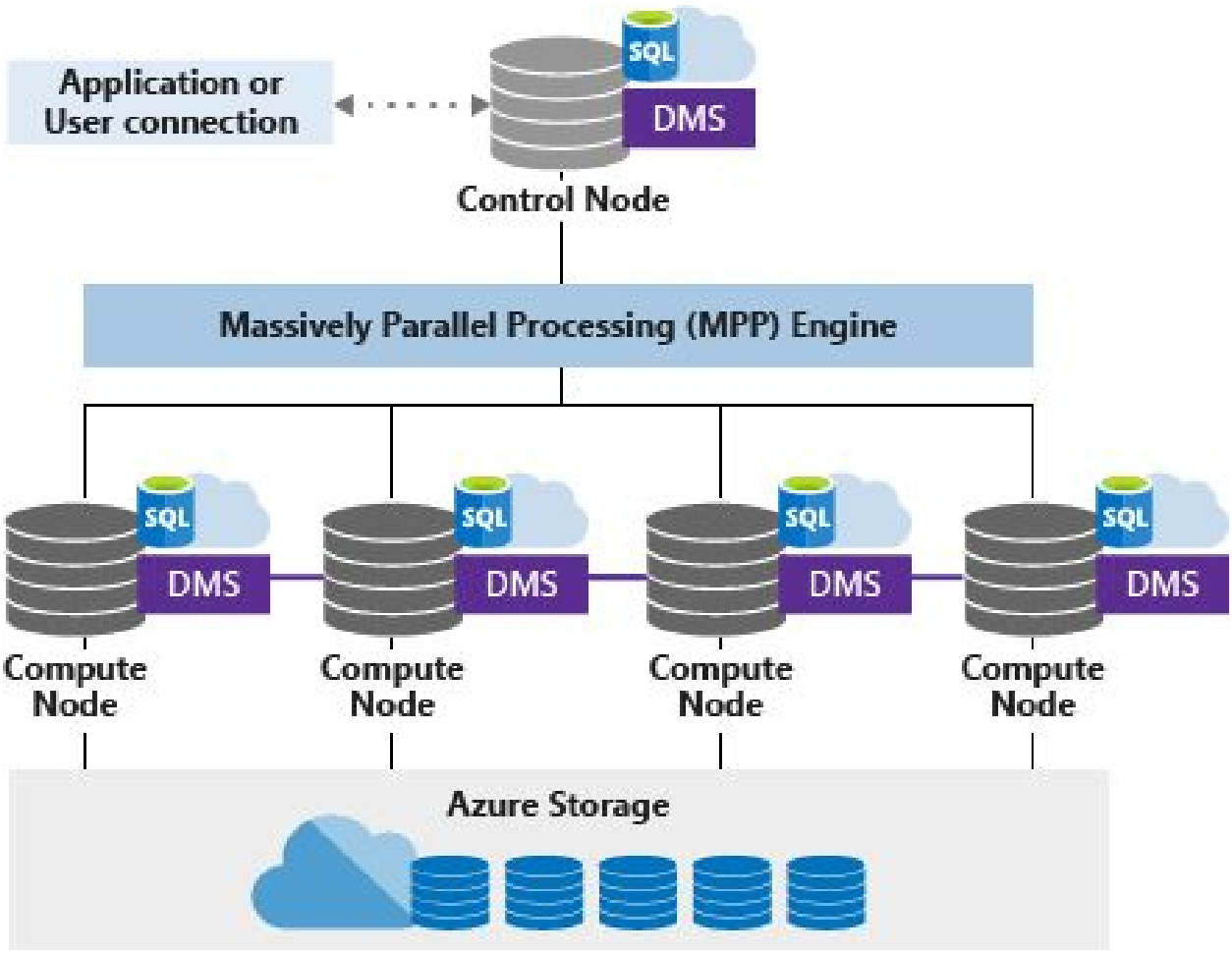
What Can I Use MPP Databases For?

The amount of data organizations produce today means that companies cannot rely on single servers or must pay handsomely for physical server capacity to handle massive datasets. Instead, MPP is becoming an increasingly popular alternative in a variety of settings.

In business intelligence, for instance, MPP databases mean that more people in an organization can run their own data analyses and queries simultaneously without experiencing lag or longer response times. Especially for larger organizations, this degree of flexibility grants more stakeholders information on demand.

MPP databases are also useful for centralizing data in a single location. Instead of having to break up massive datasets, MPP allows them to be stored in a single location and accessed from different points. This includes storing a variety of data such as marketing, web, operational, logistics, and HR data.

For larger organizations, this centralized resource makes it easier to uncover insights, connect data dots that may not be apparent at first, and even build dashboards that contain more relevant information than those built from data that is fragmented. Finally, MPP is usually best suited to handle structured data sets as opposed to models such as data lakes.



Question 23: Finding Nth Lowest Salary per Department using py-spark and SQL.

At first, we will create the spark context object.

```
# importing sparksession
from pyspark.sql import SparkSession
# creating a sparksession object
# and providing appName
spark = SparkSession.builder.appName("pyspark_window_function").getOrCreate()
```

Now we will create our hypothetical data set for employees with Salaries.

```
employees_Salary = [("James", "Sales", 2000),
```

(“sofy”, “Sales”, 3000),

(“Laren”, “Sales”, 4000),

(“Kiku”, “Sales”, 5000),

(“Sam”, “Finance”, 6000),

(“Samuel”, “Finance”, 7000),

(“Yash”, “Finance”, 8000),

(“Rabin”, “Finance”, 9000),

(“Lukasz”, “Marketing”, 10000),

(“Jolly”, “Marketing”, 11000),

(“Mausam”, “Marketing”, 12000),

(“Lamba”, “Marketing”, 13000),

(“Jogesh”, “HR”, 14000),

(“Mannu”, “HR”, 15000),

(“Sylvia”, “HR”, 16000),

(“Sama”, “HR”, 17000),

]

#Create

the spark Data frame and assign schema with it

employeesDF = spark.createDataFrame(employees_Salary,schema=”””employee_name STRING, dept_name STRING, salary INTEGER”””)

Now the data will look like the below -

employeesDF.show

()

check-in screenshot----------

Now we will apply the window function, so the data will be partition by department id and order by salary so that we can fetch the Lowest salary per department.

#libraries

for window and rank

from pyspark.sql.window import Window

from pyspark.sql.functions import rank

#Create

window specification for applying window function

windowPartition = Window.partitionBy(“dept_name”).orderBy(“salary”)

#Apply

the window specification

employeeDF = employeesDF.withColumn(“rank”, rank().over(windowPartition))

Now when we run below query data will look like below -

employeesDF.show

()

Check-in Screenshot----------

Now we will write the final spark query to get the lowest salaried employees

for each department.

employeeDF.filter(“rank=1”).show()

Check-in Screenshot----------

So, we have found the lowest salary for each department using the py-spark Window function.

Now it’s in SQL Window Function,

select * from (

SELECT employee_name, salary, dept_name, rank() over(partition by e.dept_name order by e.salary) rnk

FROM employees e, departments d

WHERE e.dept_name=d.dept_name_id

)

where rnk=1;

```
employeesDF.show()
```

```
+-------------+---------+------+
|employee_name|dept_name|salary|
+-------------+---------+------+
|        James|    Sales|  2000|
|         sofy|    Sales|  3000|
|        Laren|    Sales|  4000|
|         Kiku|    Sales|  5000|
|          Sam|  Finance|  6000|
|       Samuel|  Finance|  7000|
|         Yash|  Finance|  8000|
|        Rabin|  Finance|  9000|
|       Lukasz|Marketing| 10000|
|        Jolly|Marketing| 11000|
|       Mausam|Marketing| 12000|
|        Lamba|Marketing| 13000|
|       Jogesh|       HR| 14000|
|        Mannu|       HR| 15000|
|       Sylvia|       HR| 16000|
|         Sama|       HR| 17000|
+-------------+---------+------+
```

```
+-------------+---------+------+----+
|employee_name|dept_name|salary|rank|
+-------------+---------+------+----+
|        James|    Sales|  2000|   1|
|         sofy|    Sales|  3000|   2|
|        Laren|    Sales|  4000|   3|
|         Kiku|    Sales|  5000|   4|
|       Jogesh|       HR| 14000|   1|
|        Mannu|       HR| 15000|   2|
|       Sylvia|       HR| 16000|   3|
|         Sama|       HR| 17000|   4|
|          Sam|  Finance|  6000|   1|
|       Samuel|  Finance|  7000|   2|
|         Yash|  Finance|  8000|   3|
|        Rabin|  Finance|  9000|   4|
|       Lukasz|Marketing| 10000|   1|
|        Jolly|Marketing| 11000|   2|
|       Mausam|Marketing| 12000|   3|
|        Lamba|Marketing| 13000|   4|
+-------------+---------+------+----+
```

```
[15]: from pyspark.sql.window import Window
      from pyspark.sql.functions import rank

      windowPartition = Window.partitionBy("dept_name").orderBy("salary")

      employeeDF = employeesDF.withColumn("rank", rank().over(windowPartition))
      employeeDF.filter("rank=1").show()
```

```
+-------------+---------+------+----+
|employee_name|dept_name|salary|rank|
+-------------+---------+------+----+
|        James|    Sales|  2000|   1|
|       Jogesh|       HR| 14000|   1|
|          Sam|  Finance|  6000|   1|
|       Lukasz|Marketing| 10000|   1|
+-------------+---------+------+----+
```

Question 24: Explain Broadcast Join in Spark.

Spark broadcast joins are perfect for joining a large DataFrame with a small DataFrame.

Broadcast joins cannot be used when joining two large DataFrames.

This post explains how to do a simple broadcast join and how the broadcast() function helps Spark optimize the execution plan.

Spark splits up data on different nodes in a cluster so multiple computers can process data in parallel. Traditional joins are hard with Spark because the data is split.

Broadcast joins are easier to run on a cluster. Spark can “broadcast” a small DataFrame by sending all the data in that small DataFrame to all nodes in the cluster. After the small DataFrame is broadcasted, Spark can perform a join without shuffling any of the data in the large DataFrame.

Example:

Let’s create a DataFrame with information about people and another DataFrame with information about cities. In this example, both DataFrames will be small, but let’s pretend that the peopleDF is huge and the citiesDF is tiny.

val peopleDF = Seq(

("andrea", "medellin"),

("rodolfo", "medellin"),

("abdul", "bangalore")

).toDF("first_name", "city")

```
peopleDF.show
()
+----------+---------+
|first_name|    city|
+----------+---------+
|   andrea| medellin|
|  rodolfo| medellin|
|    abdul|bangalore|
+----------+---------+
val citiesDF = Seq(
("medellin", "colombia", 2.5),
("bangalore", "india", 12.3)
).toDF("city", "country", "population")
citiesDF.show
()
+---------+--------+----------+
|    city| country|population|
+---------+--------+----------+
| medellin|colombia|       2.5|
|bangalore|  india|      12.3|
+---------+--------+----------+
```

Let’s broadcast the citiesDF and join it with the peopleDF.

```
peopleDF.join(
broadcast(citiesDF),
peopleDF("city") <=> citiesDF("city")
).show()
+----------+---------+---------+--------+----------+
```

|first_name| city| city| country|population|

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

| andrea| medellin| medellin|colombia| 2.5|

| rodolfo| medellin| medellin|colombia| 2.5|

| abdul|bangalore|bangalore| india| 12.3|

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

The Spark null safe equality operator (<=>) is used to perform this join.

When one of the data frames is small and fits in the memory, it will be broadcasted to all the executors, and a Hash Join will be performed.

The property spark.sql.autoBroadcastJoinThreshold can be configured to set the Maximum size in bytes for a dataframe to be broadcasted.

Here, spark.sql.autoBroadcastJoinThreshold=-1 will disable the broadcast Join whereas default spark.sql.autoBroadcastJoinThreshold=10485760, i.e 10MB.

Which table will be broadcasted in the below conditions?

The Join side with the hint will be broadcasted irrespective of autoBroadcastJoinThreshold, if a broadcast hint is specified on either side of the join.

The side with a smaller physical data size will be broadcasted, if broadcast hints are specified on both sides of the Join.

The table will be broadcasted to all the executor nodes if there is no hint and the physical size of the table < autoBroadcastJoinThreshold.

If the broadcast side is small, BHJ can perform faster than other Join algorithms as there is no shuffling involved.

Is broadcasting always good for performance? Not at all!

The broadcasting table is a network-intensive operation. When the broadcasted table is big, it may lead to OOM or performs worse than other algorithms.

if you give more resources to the cluster, the non-broadcasted version will run faster than the broadcasted one as the broadcasting operation is expensive in itself. If we are increasing the number of executors, those executors need to receive the table. By increasing the number of executors, we are increasing the broadcasting cost too.

Now, imagine you are broadcasting a medium-sized table. When you run the code, everything is fine and super-fast. But in the future when a medium-sized table is no more “medium”, then your code will break with OOM.

Skewness:

When you want to join the two tables, ‘Skewness’ is the most common issue developers face. When the Join key is not uniformly distributed in the dataset, the Join will be skewed. Spark cannot perform operations in parallel when the Join is skewed, as the Join’s load will be distributed unevenly across the Executors.

If one table is very small, we can decide to broadcast it straightaway! Observe what happened to the tasks during the execution: one of the tasks took much more time.

Question 25: Explain about Sort-Merge Join in Spark.

Sort merge join: As the name suggests, Sort merge join perform the Sort operation first and then merges the datasets.

This is Spark’s default join strategy, Since Spark 2.3 the default value of spark.sql.join.preferSortMergeJoin has been changed to true.

Spark performs this join when you are joining two BIG tables, Sort Merge Joins minimize data movements in the cluster, highly scalable approach, and perform better when compared to Shuffle Hash Joins.

Performs disk IO operations same as the Map-Reduce paradigm which makes this join scalable.

Three phases of sort Merge Join –

· Shuffle Phase: The 2 big tables are repartitioned as per the join keys across the partitions in the cluster.

· Sort Phase: Sort the data within each partition parallelly.

· Merge Phase: Join the 2 Sorted and partitioned data. This is basically merging of dataset by iterating over the elements and joining the rows having the same value for the join key.

Find the Diagram----------

SMJ performs better than other joins most of the time and has a very scalable approach as it does away with the overhead of hashing and does not require the entire data to fit

inside the memory.

Let’s examine this sort-merge join with an example. Two data frames A and B have four key columns (1,2,3,4) and let’s say we have 2 node clusters.

Find the Diagram----------

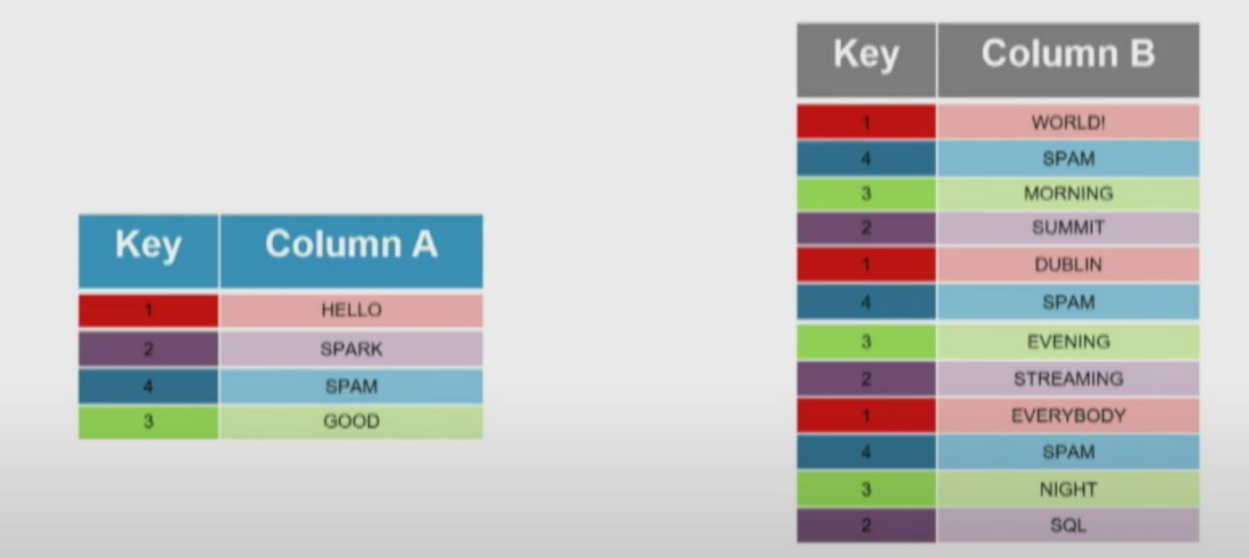
Sort Phase: As you can see, both A and B are sorted by the Join key i.e. Key column, and sorted data is split into 2 partitions. Each partition should have specific key data. There should not be any overlapping of Keys between partitions which is the whole idea of shuffling.

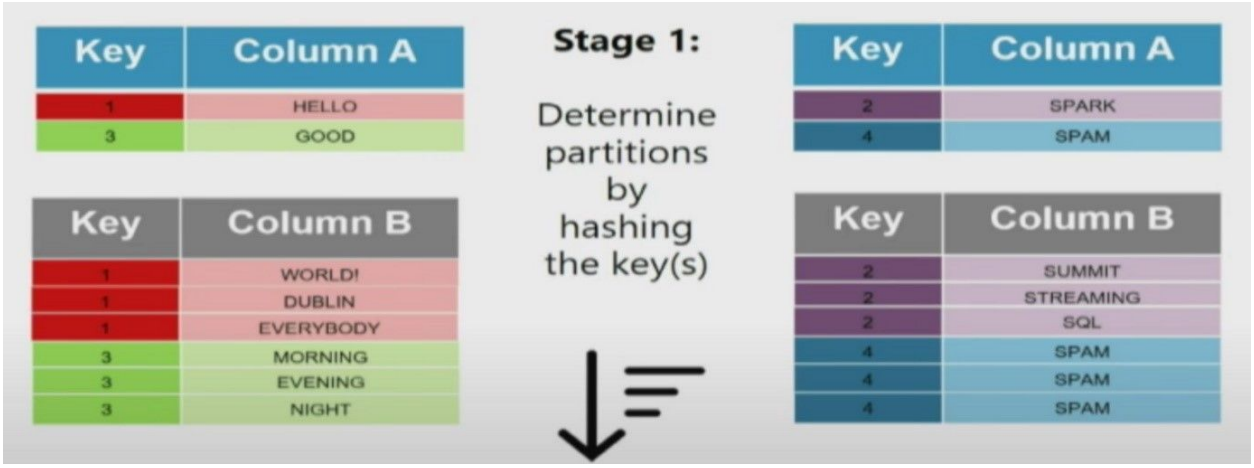
Find the Diagram----------

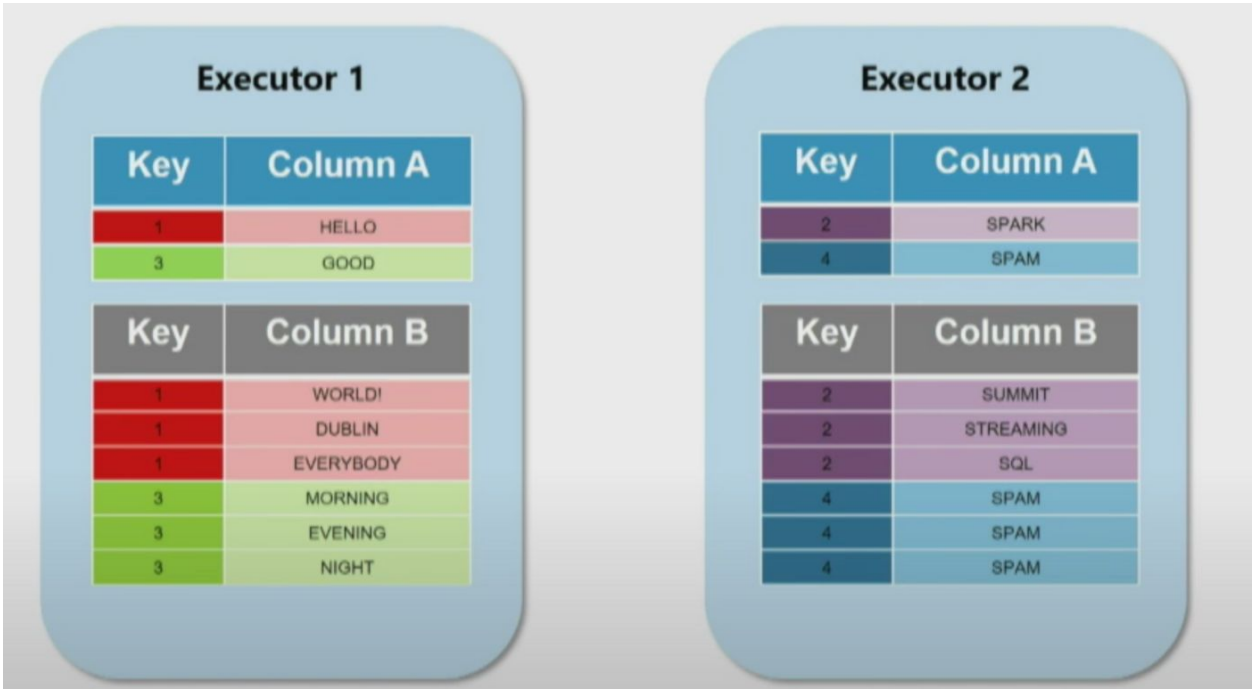
Assign sorted partitions to executors

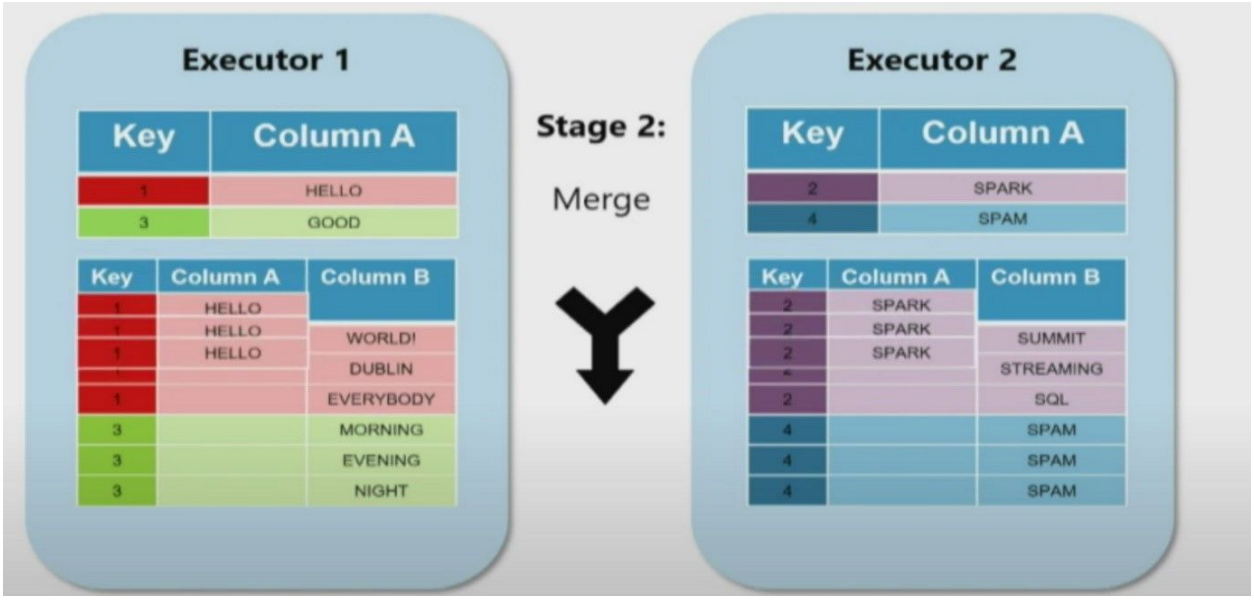
Find the Diagram----------

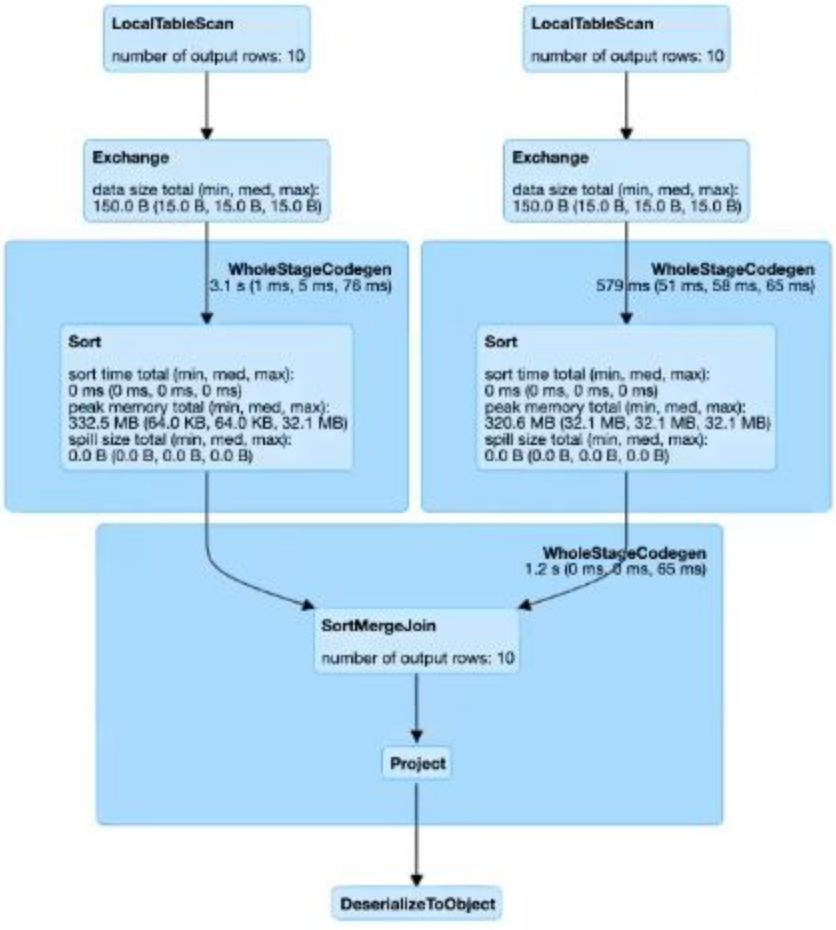
Merge Phase: Merging sorted data by keys is a very simple and quickest operation.




Key
Column A
1
HELLO
3
GOOD
Key
Column B
1
WORLD!
1
DUBLIN
1
EVERYBODY
3
MORNING
3
EVENING
3
NIGHT
Stage 1:
Determine partitions by hashing the key(s)
Key
Column A
2
SPARK
4
SPAM
Key
Column B
2
SUMMIT
2
STREAMING
2
SQL
4
SPAM
4
SPAM
4
SPAM


Executor 1
Key
Column A
1
HELLO
3
GOOD
Key
Column B
1
WORLD!
1
DUBLIN
1
EVERYBODY
3
MORNING
3
EVENING
3
NIGHT
Executor 2
Key
Column A
2
SPARK
4
SPAM
Key
Column B
2
SUMMIT
2
STREAMING
2
SQL
4
SPAM
4
SPAM
4
SPAM





The property which leads to setting the Sort-Merge Join:

spark.sql.join.preferSortMergeJoin

In the class involved in sort-merge join we should mention

org.apache.spark.sql.execution.joins.SortMergeJoinExec

Below is the simple script which shows you how Sort-Merge-Join works.

```
from pyspark.sql import SparkSession
spark = SparkSession.builder\
.appName("sort-merge-analysis")\
.master("yarn")\
.config("spark.sql.join.preferSortMergeJoin", "true")\
.config("spark.sql.autoBroadcastJoinThreshold", "1")\
.config("spark.sql.defaultSizeInBytes", "100000")\
.enableHiveSupport()\
.getOrCreate()
orders = spark.read.json('retail_db_json/orders')
order_item = spark.read.json('retail_db_json/order_items')
wide_table = order_item.join(orders , orders.order_id==order_item.order_item_order_id)
wide_table.show
()
```

Here is the query Execution plan:

```
== Physical Plan ==
*SortMergeJoin [order_item_order_id#26L], [order_id#10L], Inner
:- *Sort [order_item_order_id#26L ASC NULLS FIRST], false, 0
: +- Exchange hashpartitioning(order_item_order_id#26L, 200)
:    +- *Project [order_item_id#25L, order_item_order_id#26L,
order_item_product_id#27L, order_item_product_price#28, order_item_quantity#29L,
order_item_subtotal#30]
:       +- *Filter isnotnull(order_item_order_id#26L)
```

: +- *FileScan json

Question 26: Explain About Serialization in Spark?

In distributed systems, data transfer over the network is the most common task. If this is not handled efficiently, you may end up facing numerous problems, like high memory usage, network bottlenecks, and performance issues.

Serialization plays an important role in the performance of any distributed application.

Serialization refers to converting objects into a stream of bytes and vice-versa (de-serialization) in an optimal way to transfer it over nodes of network or store it in a file/memory buffer.

Spark provides two serialization libraries and modes are supported and configured through spark.serializer property.

Java serialization (default):

Java serialization is the default serialization that is used by Spark when we spin up the driver. Spark serializes objects using Java's ObjectOutputStream framework. The serialization of a class is enabled by the class implementing the java.io.Serializable interface. Classes that do not implement this interface will not have any of their states serialized or de-serialized. All subtypes of a serialized class are themselves serialized.

A class is never serialized, only the object of a class is serialized.

Java serialization is slow and leads to large serialized formats for many classes. We can fine-tune the performance by extending java.io.Externalizable.

Kryo serialization (recommended by Spark):

public class KryoSerializer

extends Serializer

implements Logging, java.io.Serializable

Kryo is a Java serialization framework that focuses on speed, efficiency, and a user-friendly API.

Kryo has less memory footprint, which becomes very important when you are shuffling and caching a large amount of data. However, it is not natively supported to serialize to

the disk. Both methods, saveAsObjectFile on RDD and objectFile on SparkContext support Java serialization only.

Kryo is not the default because of the custom registration and manual configuration requirement.

Default Serializer:

When Kryo serializes an object, it creates an instance of a previously registered Serializer class to do the conversion to bytes. Default serializers can be used without any setup on our part.

Custom Serializer:

For more control over the serialization process, Kryo provides two options. We can write our own Serializer class and register it with Kryo or let the class handle the serialization by itself.

Let's see how we can set up Kryo to use in our application.

val conf = new SparkConf()

.set("spark.serializer", "org.apache.spark.serializer.KryoSerializer")

.set("spark.kryoserializer.buffer.mb","24")

val sc = new SparkContext(conf)val sc = new SparkContext(conf)

val spark =

SparkSession.builder().appName(“KryoSerializerExample”)

.config(someConfig) .config(“spark.serializer”, “org.apache.spark.serializer.KryoSerializer”) .config(“spark.kryoserializer.buffer”, “1024k”) .config(“spark.kryoserializer.buffer.max”, “1024m”) .config(“spark.kryo.registrationRequired”, “true”)

.getOrCreate }

The buffer size is used to hold the largest object you will serialize. It should be large enough for optimal performance.

KryoSerializer is a helper class provided by Spark to deal with Kryo. We create a single instance of KryoSerializer that configures the required buffer sizes provided in the configuration.

Finally, you can follow these guidelines by Databricks to avoid serialization issues:

- Make the object/class serializable.
- Declare the instance within the lambda function.
- Declare functions inside an object as much as possible.
- Redefine variables provided to class constructors inside functions.

Serialization Rules:

Before we get into examples let’s explore the basic rules around serialization with respect to Spark code.

When will objects need to be Serialized?

When you perform a function on an RDD (Spark’s Resilient Distributed Dataset), or on anything that is an abstraction on top of this (e.g. Dataframes, Datasets), it is common that this function will need to be serialized so it can be sent to each worker node to execute on its segment of the data.

What gets Serialized?

The rules for what is Serialized are the same as in Java more generally — only objects can be serialized.

The function is passed to map (or similar Spark RDD function) itself will need to be Serialized (note this function is itself an object). If references to other objects are made within this function, then those objects will also need to be serialized. The whole of these objects will be serialized, even when accessing just one of their fields.

Examples:

Examples including code and explanations follow, though I strongly encourage you to try running the examples yourself and trying to figure out why each one works or doesn’t work —

For each of these examples assume we have a testRdd containing Integers.

val testRdd: RDD[Int]

Basic(ish) Examples

We’ll start with some basic examples that draw out the key principles of Serialization in Spark.

1 — basic spark map

object Example {

def myFunc =

testRdd.map

(_ + 1)

}

- *PASSES**

A very simple example — in this case the only thing that will be serialized is a Function1 object which has an apply method that adds 1 to its input. The Example object won't be serialized.

2 — spark map with external variable

object Example {

val num = 1

def myFunc =

testRdd.map

(_ + num)

}

- *FAILS**

Very similar to the above, but this time within our anonymous function we're accessing the num value. Therefore the whole of the containing Example object will need to be serialized, which will actually fail because it isn't serializable.

3 — spark map with external variable — the first way to fix it

object Example extends Serializable {

val num = 1

def myFunc =

testRdd.map

(_ + num)

}

- *PASSES**

One solution people often jump to is to make the object in question Serializable. It works, but may not be desirable as ideally, we want to be serialized as little as possible.

4 — spark map with external variable — a flawed way to fix it

```
object Example {

val num = 1

def myFunc = {

lazy val enclosedNum = num

testRdd.map

(_ + enclosedNum)

}

}
```

- *FAILS**

In this case, we create an enclosedNum value inside the scope of myFunc — when this is referenced it should stop trying to serialize the whole object because it can access everything required in the scope of myFunc. However, because enclosedNum is a lazy val this still won't work, as it still requires knowledge of num and hence will still try to serialize the whole of the Example object.

5 — spark map with external variable — properly fixed!

```
object Example {

val num = 1

def myFunc = {

val enclosedNum = num

testRdd.map

(_ + enclosedNum)

}

}
```

- *PASSES**

Similar to the previous example, but this time with enclosedNum being a val, which fixes the previous issue.

Upping the difficulty — examples with nested objects

The same principles apply in the following examples, just with the added complexity of a nested object.

6 — nested objects, a simple example

object Example {

val outerNum = 1

object NestedExample extends Serializable {

val innerNum = 10

def myFunc =

testRdd.map

(_ + innerNum)

}

}

- *PASSES**

A slightly more complex example but with the same principles. Here innerNum is being referenced by the map function. This triggers the serialization of the whole of the NestedExample object. However, this is fine because it extends Serializable. You could use the same enclosing trick as before to stop the serialization of the NestedExample object too.

7 — nested objects gone wrong

object Example {

val outerNum = 1

object NestedExample extends Serializable {

val innerNum = 10

def myFunc =

testRdd.map

(_ + outerNum)

}

}

- *FAILS**

In this case outerNum is being referenced inside the map function. This means the whole Example object would have to be serialized, which will fail as it isn't Serializable.

8 — nested objects, using enclosing in the inner object

object Example {

val outerNum = 1

object NestedExample extends Serializable {

val innerNum = 10

val encOuterNum = outerNum

def myFunc =

testRdd.map

(_ + encOuterNum)

}

}

- *PASSES**

In this example, we have fixed the previous issue by providing encOuterNum. Now the map references only values in the NestedExample object, which can be serialized.

Question 27: Pivot, Unpivot Data with SparkSQL & PySpark – Databricks

PySpark pivot() function is used to rotate/transpose the data from one column into multiple Dataframe columns and back using unpivot(). Pivot() It is an aggregation where one of the grouping column values is transposed into individual columns with distinct data.

pivot(pivot_col, values=None)

pivot_col — Name of the column to Pivot

values — List of values that will be translated to columns in the output DataFrame.

We consider the table SparkTable before pivoting data.

```
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.functions import expr
data = [("Banana",1000,"USA"), ("Carrots",1500,"USA"), ("Beans",1600,"USA"),
("Orange",2000,"USA"),("Orange",2000,"USA"),("Banana",400,"China"),
("Carrots",1200,"China")("Beans",1500,"China"), ("Orange",4000,"China"),
("Banana",2000,"Canada"),("Carrots",2000,"Canada"),("Beans",2000,"Mexico")]
columns= ["Product","Amount","Country"]
df = spark.createDataFrame(data = data, schema = columns)
df.printSchema()
df.show
(truncate=False)
+-------+------+-------+
|Product|Amount|Country|
+-------+------+-------+
|Banana |1000 |USA    |
|Carrots|1500 |USA    |
|Beans |1600 |USA    |
|Orange |2000 |USA    |
|Orange |2000 |USA    |
|Banana |400  |China |
|Carrots|1200 |China |
|Beans |1500 |China |
|Orange |4000 |China |
|Banana |2000 |Canada |
```

|Carrots|2000 |Canada |

|Beans |2000 |Mexico |

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

After Pivoting:

pivotDF = df.groupBy("Product").pivot("Country").sum("Amount")

pivotDF.printSchema()

pivotDF.show

(truncate=False)

(OR)

spark.sql("select * from SparkTable").groupBy("Product").pivot("Country").sum("Amount").show()

Output:

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

|Product|Canada|China|Mexico|USA |

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

|Orange |null |4000 |null |4000|

|Beans |null |1500 |2000 |1600|

|Banana |2000 |400 |null |1000|

|Carrots|2000 |1200 |null |1500|

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

Unpivot is a reverse operation, we can achieve by rotating column values into row values. PySpark SQL doesn't have unpivot function hence will use the stack() function. Below code converts column countries to rows. Unpivot all of the country columns into one single country column.

from pyspark.sql.functions import expr

unpivotExpr = "stack(3, 'Canada', Canada, 'China', China, 'Mexico', Mexico) as (Country,Total)"

unPivotDF =

pivotDF.select

("Product", expr(unpivotExpr)) \

.where("Total is not null")

unPivotDF.show

(truncate=False)

unPivotDF.show

()

(OR)

spark.sql("SELECT Product,stack(3, 'Canada', Canada, 'China', China, 'Mexico', Mexico) as (Country,Total) from PivotTable").show()

It converts pivoted column “country” to rows.

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

|Product|Country|Total|

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

| Orange| China| 4000|

| Beans| China| 1500|

| Beans| Mexico| 2000|

| Banana| Canada| 2000|

| Banana| China| 400|

|Carrots| Canada| 2000|

|Carrots| China| 1200|

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

Question 28: Pyspark SQL Functions:

i)PySpark Aggregate Functions:

PySpark SQL Aggregate functions are grouped as “agg_funcs” in Pyspark. Below is a list of functions defined under this group.

approx_count_distinct

avg

collect_list

collect_set

countDistinct

count

grouping

first

last

kurtosis

max

min

mean

skewness

stddev

stddev_samp

stddev_pop

sum

sumDistinct

variance, var_samp, var_pop

ii)PySpark Window Functions:

Ranking Functions:

1)row_number(),

2)rank(),

3)dense_rank(),

4)percent_rank(),

5)ntile()

Analytic Functions:

1)cume_dist(),

2)lag(),

3)lead()

Aggregate Functions:

1)sum(),

2)first(),

3)last(),

4)max(),

5)min(),

6)mean(),

7)stddev()

iii)Pyspark JSON Functions:

from_json() – Converts JSON string into Struct type or Map type.

to_json() – Converts MapType or Struct type to JSON string.

json_tuple() – Extract the Data from JSON and create them as a new column.

get_json_object() – Extracts JSON element from a JSON string based on json path specified.

schema_of_json() – Create schema string from JSON string

iv)PySpark SQL Date and Timestamp Functions:

1)Date Functions

2)Timestamp Functions

3)Date and Timestamp Window Functions

Question 29: Optimizations in Azure Databricks. (Now its time to focus on Azure with Bigdata)

Azure Databricks provides optimizations for Delta Lake that accelerate data lake operations, supporting a variety of workloads ranging from large-scale ETL processing to ad-hoc, interactive queries. Many of these optimizations take place automatically; you get their benefits simply by using Azure Databricks for your data lakes.

- Optimize performance with file management
  - Compaction (bin-packing)
  - Data skipping
  - Z-Ordering (multi-dimensional clustering)
  - Tune file size
  - Notebooks
  - Improve interactive query performance
  - Frequently asked questions (FAQ)
- Auto Optimize
  - How Auto Optimize works
  - Enable Auto Optimize
  - When to opt in and opt out
  - Example workflow: Streaming ingest with concurrent deletes or updates
  - Frequently asked questions (FAQ)
- Optimize performance with caching
  - Delta and Apache Spark caching
  - Delta cache consistency
  - Use Delta caching
  - Cache a subset of the data
  - Monitor the Delta cache
  - Configure the Delta cache
- Dynamic file pruning
- Isolation levels
  - Set the isolation level
- Bloom filter indexes
  - How Bloom filter indexes work
  - Configuration

- Create a Bloom filter index
- Drop a Bloom filter index
- Display the list of Bloom filter indexes
- Notebook
- Low Shuffle Merge
- Optimized performance
- Optimized data layout
- Availability
- Optimize join performance
- Range join optimization
- Skew join optimization
- Optimized data transformation
- Higher-order functions
- Transform complex data types

Azure Databricks Performance Notes:

1) Storage Optimized Spark cluster type.

2) Enable the Delta cache - spark.databricks.io.cache.enabled true.

3) Set an appropriate number of shuffling partitions.

4) Use Auto Optimize for your write workload.

5) Clean up your files with the Vacuum command.