# Apache Spark eco system & anatomy interview Q&As

Posted on [September 16, 2020](https://www.java-success.com/apache-spark-anatomy-interview-qas/)

Q01. Can you summarise the Spark eco system?  
A01. Apache Spark is a general purpose cluster **computing** system. It provides high-level API in Java, Scala, Python, and R. It has **6 components** Core, Spark SQL, Spark Streaming, Spark MLlib, Spark GraphX, and SparkR. All the functionalities being provided by Apache Spark are built on the top of **Spark Core**. Spark Core is the foundation of in-memory parallel and distributed processing of huge dataset with fault-tolerance & recovery.

Spark Eco System

The **Spark SQL** component is a distributed framework for structured data processing. **Spark Streaming** is an add on API, which allows scalable, high-throughput, fault-tolerant stream processing of live data streams. Spark can access data from sources like **Kafka**, **Flume**, **Amazon Kinesis** or **TCP socket**. **MLlib** in Spark is a scalable Machine learning library. **GraphX** in Spark is API for graphs. The key component of **SparkR** is SparkR DataFrame.

Q02. What are the key execution components of Apache Spark?  
A02. The 5 key components of Apache Spark are:

1) Spark **Driver**  
2) **Application Master**  
3) Spark **Session** (or Spark **Context** prior to Spark 2.0)  
4) **Executors**  
5) Cluster Resource Manager (E.g. **YARN**, Kubernetes, Mesos, Nomad & Spark’s own standalone cluster manager)

These components **PLAN**, **SCHEDULE**, **EXECUTE** & **MONITOR** the Spark application.

Apache Spark Execution

As shown below, Apache Spark uses **Master/Slave** architecture. The slave nodes are also known as the **worker nodes** or **core nodes**. A typical cluster will have 100’s to 1000’s of slave nodes. This is where you have the parallel execution of **tasks**. Tasks need to be planned, scheduled, queued, executed & monitored. If any executor crashes, its tasks will be sent to different executors to be processed again.

Apache Spark Architecture – Cluster Mode

Q03. What is a Driver?  
A03. A Spark **Driver** is a process where the main method runs. The Spark driver is the process which the clients used to submit the spark program. First it converts the user program into smaller execution units called **tasks** and after that it **schedules** the tasks on the executors.

For example, A driver initiates “**map**” tasks on the cluster executors against the data in the slave nodes, and each executor returns a subset of the data back to the Driver as a “**reduce**” operation to be combined & returned back to the client as a final result.

A Spark Driver contains components like DAGScheduler, TaskScheduler, etc responsible for converting user code into Spark jobs to be executed on the cluster. A Driver contains metadata of all the RDDs & their partitions.

The Spark driver runs on the port 4040 and UI is created automatically once the user submits the spark program to the spark driver. Sparkdriver:4040/jobs/

Spark UI

Q04. What are the different Spark modes of execution?  
A04. **Client Mode** & **Cluster Mode**. These modes change the behavior as to where the “Driver” runs.

In **Client Mode** a **Driver** component of spark job will run on the machine from which a job is submitted. For example, your local machine.

In a **client mode**, the Spark master plays the role of Cluster manager. Spark master negotiates the resources with slave (aka worker) nodes and tracks their status & monitor the progress. It also makes the resources available to spark driver.

Apache Spark – Client Mode

In **Cluster Mode** job submitting machine is remote from “spark infrastructure” as shown in the cluster diagram. The job will be submitted from a local machine or an edge node, but the Driver will be running in the cluster.

**spark-shell** should be used for **interactive**queries as it needs to be run in yarn-client mode so that the machine you’re running on acts as the driver. For **spark-submit**, you submit jobs to the cluster then the task runs in the cluster.

##### YARN client mode

Java

|  |  |
| --- | --- |
| 1  2  3  4 | $ spark-shell –-master yarn  $ spark-shell –-master yarn –-deploy-mode client |

##### YARN cluster mode

Java

|  |  |
| --- | --- |
| 1  2  3  4 | $ spark-submit –-class com.myapp.MySparkApp myspark.jar yarn-client  $ spark-submit –-class com.myapp.MySparkApp myspark.jar yarn-cluster |

You can also run Spark in **local mode**. This is a non-distributed single JVM deployment mode, where Spark spawns all the execution components – driver, executor, and master in the same single JVM. This is the only mode where a driver is used for execution.

Java

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | local[\*],local,local[2]…etc  ...  $ spark-shell –-master local[1]  $ spark-submit –-class com.myapp.MySparkApp myspark.jar local[1] |

Spark distribution comes with its own resource manager. When your program uses spark’s resource manager, execution mode is called **Standalone cluster mode**.

Java

|  |  |
| --- | --- |
| 1  2  3  4 | $ spark-shell –-master spark://hduser:7077  $ spark-submit –-class com.myapp.MySparkApp myspark.jar spark://hduser:7077 |

The only difference between Standalone mode and local mode is that in Standalone mode you are defining “containers” for the worker and spark master to run in your machine, but in local mode you are just running everything in the same JVM in your local machine.

Q05. What is an Application Master in Spark?  
A05. An **Application Master** is the process that requests resources from the cluster and make these available to the spark driver in-turn to execute the tasks in the executors. It is created on the same node as the Driver in the **cluster mode** when **spark-submit** is invoked. Each Spark application will have its **own** dedicated Application Master.

In a **client mode**, the Spark master plays the role of Cluster manager. Spark master negotiates the resources with slave nodes and tracks their status & monitor the progress. It also makes the resources available to spark driver.

Q06. What is a Spark Session or Spark Context?  
A06. A “**SparkContext**” is the **main entry point** for a Spark job prior Spark version 2.0. Starting from Apache Spark **2.0**, **Spark Session** is the new entry point for Spark applications. A Spark context is created by the Spark driver for each individual Spark programs when it is first submitted by the user.

Q07. What is a Cluster Resource Manager?  
A07. In a distributed computing, a **cluster resource manager** is responsible for monitoring the **containers** in the slave nodes and reserving the resources on these nodes upon request by the application master. The application master in turn makes these resources available to the spark driver program to execute the tasks and stages in executors. These containers are reserved based on the needs of the executors.

A **SparkSession** can connect to any cluster resource manager like YARN, Kubernetes, Mesos, etc.

Q08. What are the Spark executors?  
A08. Spark **executors** in the slave (aka worker or core) nodes are responsible for executing the assigned tasks. The results of each task are returned to the Spark Driver. Executors can be **statically allocated** via spark-submit arguments or **dynamically allocated** based on the overall work load by adding & removing executors. The dynamic allocation can adversely impact other spark jobs running in the cluster.

Executors only know of the tasks allocated to them and it’s the responsibility of the spark driver to coordinate a set of tasks with the correct dependencies.

Q09. How do you know which piece of code runs on driver or executor?  
A09. A Spark application consists of a **single Driver process** and **one or more Executor processes**. Driver process is responsible for a lot of things including directing the overall control flow of your application, restarting failed stages and the entire high level direction of how your application will process the data.

You can increase or decrease the number of Executors dynamically depending upon your usage, but the Driver will exist throughout the lifetime of your application.

As a rule of thumb everything that is executed inside functions like map, filter, flatMap, combineByKey, etc should be handled by executor nodes. Everything outside these are handled by the driver.

Java

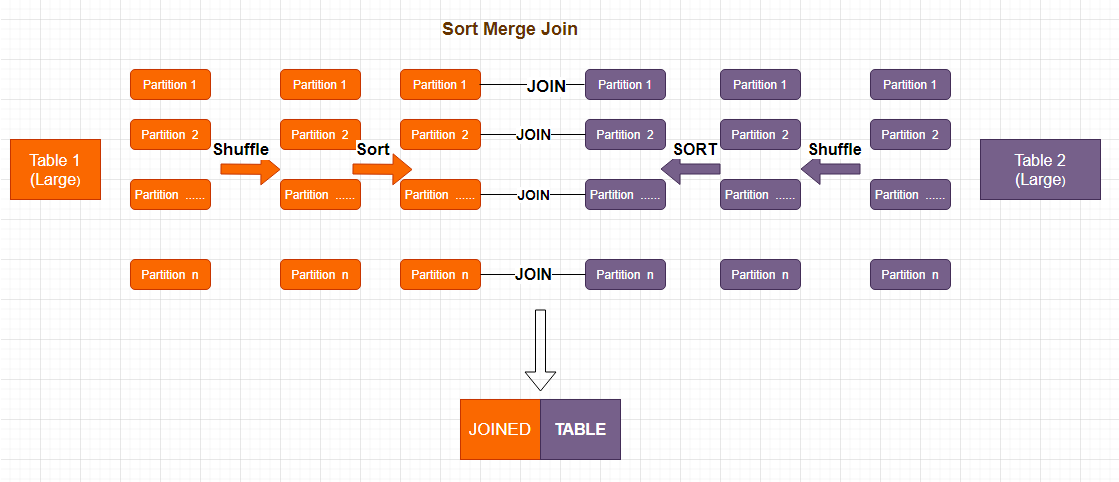
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | from pyspark.sql import SparkSession    # Runs in Driver  conf = SparkConf().setAppName(appName).setMaster(master)  sc = SparkSession\          .builder\          .appName("PythonWordCount")\      .config(conf=conf)          .getOrCreate()    # Runs in Driver. Driver splits linesRDD into tasks to be run in Executors  # Driver will send tasks to executors via Cluster Manager  linesRDD = sc.textFile("hdfs://...")    # Runs in executors as parallel tasks  wordsRDD = linesRDD.flatMap(lambda line: line.split(" ")    # Runs in executors as parallel tasks  wordCountRDD= wordsRDD.map(lambda word: (word, 1))    # Runs in executors as parallel tasks.  resultRDD = wordCountRDD.reduceByKey(lambda a, b: a + b)    # Runs in executors  resultRDD.saveAsTextFile("hdfs://...")    # Runs in Driver  spark.stop() |

**Spark SQL joins & performance tuning interview questions & answers**

Posted on [September 25, 2020](https://www.java-success.com/spark-sql-joins-performance-tuning-interview-questions-answers/)

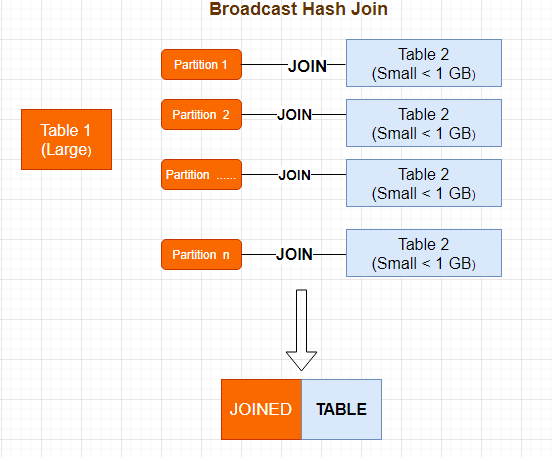
Q1. What are the different types of Spark SQL joins?  
A1. There are 3 types of joins.

1) **Sort Merge Join** – when both table 1 & table 2 are large. You need to shuffle & sort by the join keys.

[](https://www.java-success.com/wp-content/uploads/2020/09/Spark-SQL-SORT_MERGE_JOIN.png)

Spark SQL Sort Merge Join

2) **Broadcast Hash Join** – when table 1 is large & table 2 is small (i.e. under 1GB). The smaller table 2 will be copied to each node to avoid shuffling with the view to improve performance.

[](https://www.java-success.com/wp-content/uploads/2020/09/Spark2-SQL-BROADCAST_HASH_JOIN.png)

Spark SQL Broadcast Join

**Drawbacks**:

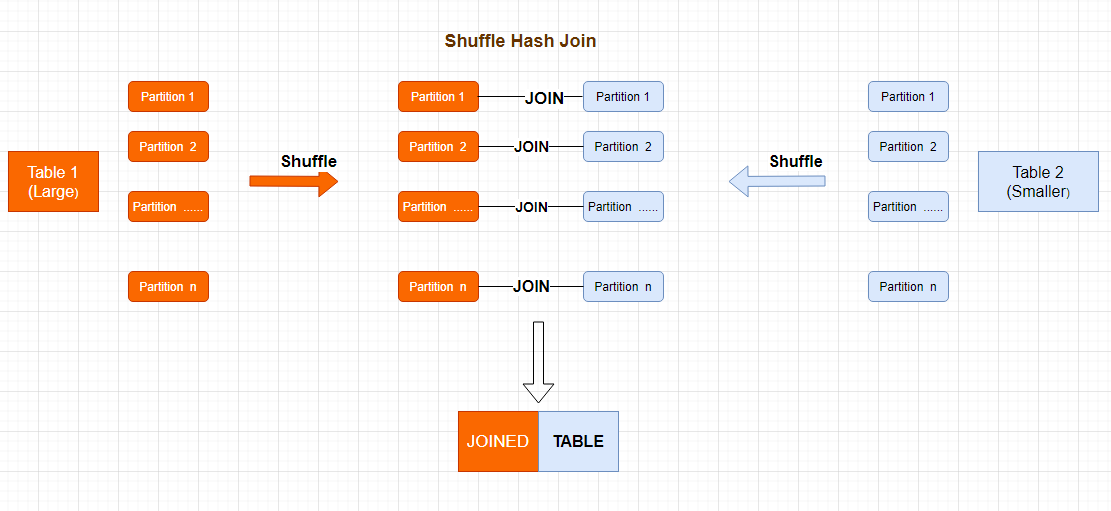
a) This algorithm can only be used to broadcast smaller tables, otherwise the redundant transmission of data is much greater than the cost of shuffle.

b) Since driver memory is also involved in broadcasting along with the executor memory, it can only be used for smaller tables as the driver can throw OutOfMemoryError.

Java

|  |  |
| --- | --- |
| 1  2  3 | /bin/spark2-submit --executor-cores 4 --driver-memory 2g --executor-memory 16g --conf spark.dynamicAllocation.enabled=true --conf spark.dynamicAllocation.minExecutors=4 --conf spark.dynamicAllocation.maxExecutors=20  --num-executors 6 |

3) **Shuffle Hash Join** – when table 1 is large & table 2 is relatively smaller, but greater than 1GB and “Broadcast Hash Join” cannot be performed because the broadcast table is first collected to the driver segment, and then distributed to each executor. This redundant transmission & potential Driver Memory Issue can adversely impact the job & performance.

[](https://www.java-success.com/wp-content/uploads/2020/09/Spark-SQL-SHUFFLE_HASH_JOIN.png)

Spark SQL Shuffle Hash Join

Q2. When will you use these joins in Spark SQL?  
A2. Use a **hint** in your query when joining a small table. These “smaller tables” are typically **dimension tables** that do not cost a lot to copy to each node. This reduces the amount of **shuffling** required to join big fact tables to smaller dimension tables.

Java

|  |  |
| --- | --- |
| 1  2  3  4  5 | SELECT /\*+ BROADCAST(Table\_2) \*/ COLUMN  FROM Table\_1 t1  JOIN Table\_2 t2  on t1.key= t2.key |

OR

Java

|  |  |
| --- | --- |
| 1  2  3 | table\_1\_df.join(table\_2\_df.hint("broadcast"), ["key"]) |

In spark 2.x, only broadcast hint was supported in SQL joins. This forces spark SQL to use broadcast join even if the table size is bigger than broadcast threshold. In Spark 3.0 you can add other join hints.

Java

|  |  |
| --- | --- |
| 1  2  3 | val joined\_df = table\_1.hint("shuffle\_hash").join(table\_2\_df,"key") |

Shuffle Hash Join is a join where both dataframes are partitioned using same partitioner. Here join keys will fall in the same partitions.

Spark SQL 2.4 added support for **COALESCE** and **REPARTITION** hints (using SQL comments):

Java

|  |  |
| --- | --- |
| 1  2  3 | SELECT /\*+ COALESCE(5) \*/ …​ |

Java

|  |  |
| --- | --- |
| 1  2  3 | SELECT /\*+ REPARTITION(3) \*/ …​ |

Q3. How will you know that what join the Spark SQL has used?  
A3. You can use “**explain**”

Java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | val sqlSimple = s"""SELECT /\*+ BROADCAST(Table\_2) \*/ COLUMN                      FROM Table\_1 t1                      JOIN Table\_2 t2                      on t1.key= t2.key"""    val df = spark.sql(sqlSimple).repartition(20)  df.explain() |

OR to shorten the plan

Java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | val sqlSimple = s"""SELECT /\*+ BROADCAST(Table\_2) \*/ COLUMN                      FROM Table\_1 t1                      JOIN Table\_2 t2                      on t1.key= t2.key"""    val df = spark.sql(sqlSimple).repartition(20)  df.explain(true) |

Q4. What other tips you have Spark SQL optimisation?  
A4.

**1)** As table joins can be expensive, join on numeric keys whenever possible and cast strings to integer if possible.

**2)** A common table expression (CTE) defines a temporary result set that a user can reference possibly multiple times.

Use CTE (i.e. Common Table Expression) as explained in [Common Table Expressions (i.e. CTE) in SQL using the “WITH” clause](https://www.java-success.com/sql-common-table-expressions-e-cte/) for scenarios where

a) **substring** is used in join.

Java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | with product\_mod AS  (SELECT p.name,         Cast(Substring(prod\_group,5,4) AS  INTEGER) AS pg1,         Cast(Substring(prod\_group,1,4) AS  INTEGER) AS pg2,  FROM   product p) |

b) If your query involves recalculating a complicated subset of data multiple times, move this calculation into a CTE.

**Spark interview Q&As with coding examples in Scala – part 1**

Posted on [October 1, 2020](https://www.java-success.com/spark-interview-qas-with-coding-examples-in-scala-part-1/)

Some of these basic Apache Spark interview questions can make or break your chance to get an offer.

Q01. Why is “===” used in the below Dataframe join?

Java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | df1.join (df2, $"df1Key" === $"df2Key")  df1.join (df2, $"df1Key" === $"df2Key", "inner")  // inner join  df1.join (df2, $"df1Key" === $"df2Key", "left")   // left outer join  df1.join(df2).where($"df1Key" === $"df2Key")  df1.join(df2).filter($"df1Key" === $"df2Key") |

A01. Comparisons with == and != are universal. This has a problem as they compare any two values, no matter what their **types** are. The Scala compiler does give warnings when two different types are compared as shown below:

Java

|  |  |
| --- | --- |
| 1  2  3  4 | scala> 1 == "some text"  <console>:12: warning: comparing values of types Int and String using `==' will always yield false |

But, this warning coverage is **NOT** comprehensive as no warning is issued for below code:

Java

|  |  |
| --- | --- |
| 1  2  3  4 | scala> "some text" == 1  res2: Boolean = false |

Another example would be when you use a **proxy** for some data structure, the proxy and the underlying data would have different types. If you accidentally compare a proxy with the underlying type using == or a pattern match, the code is still valid, but it will just always result in **false**.

Scala prides itself as a **strong static type system**. “**===**” is a **type safe equality operator**.

Java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | scala> "some text" === 1  res2: Boolean = false  <console>:13: error: type mismatch;  found   : Int(1)  required: String         "some text" === 1 |

Q02. When you join Dataframe, how do you know which join strategy is used by Spark?  
A02. There are 4 **join strategies**:

1) Broadcast Join  
2) Shuffle Hash Join  
3) Sort Merge Join  
4) BroadcastNestedLoopJoin

[**Learn more:** [Spark SQL joins & performance tuning interview questions & answers](https://www.java-success.com/spark-sql-joins-performance-tuning-interview-questions-answers/)].

You can use

Java

|  |  |
| --- | --- |
| 1  2  3 | resultDF.explain()   // physical query plan |

OR

Java

|  |  |
| --- | --- |
| 1  2  3 | resultDF.explain(true)   // physical & logical query plan |

**Sample output:** It is a “SortMergeJoin” in this example.

Java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | == Physical Plan ==  Union  :- SortMergeJoin [cID#8], [customerID#23], LeftOuter  :  :- \*(2) Sort [cID#8 ASC NULLS FIRST], false, 0  :  :  +- Exchange hashpartitioning(cID#8, 200)  :  :     +- \*(1) Project [\_1#4 AS cID#8, \_2#5 AS c2#9, \_3#6 AS c3#10]  :  :        +- \*(1) SerializeFromObject [assertnotnull(input[0, scala.Tuple3, true]).\_1 AS \_1#4, assertnotnull(input[0, scala.Tuple3, true]).\_2 AS \_2#5, assertnotnull(input[0, scala.Tuple3, true]).\_3 AS \_3#6]  ........... |

Q03. How do you remove duplicate rows in Spark?  
A03. You can use **distinct()** to remove rows that have the same values on **all columns**.

On **Databricks** notebook – [**Spark Tutorials on Databricks Notebook**](https://www.java-success.com/category/java-success-com/40-data-engineers/big-data-tutorials/tutorials-databricks/).

Java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | %scala      case class Employee(name: String, age: Int, salary: Double)    // Create the Employees  val employee1 = new Employee("Peter", 25, 35000.00)  val employee2 = new Employee("Peter", 26, 42000.00)  val employee3 = new Employee("John", 34, 45000.00)    val employees = Seq(employee1, employee2, employee3)  val df1 = employees.toDF()    df1.distinct().show() |

**Output:**

Java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | +-----+---+-------+  | name|age| salary|  +-----+---+-------+  |Peter| 25|35000.0|  |Peter| 26|42000.0|  | John| 34|45000.0|  +-----+---+-------+ |

Q04. What if you want to remove duplicates on selected columns?  
A04. Use **dropDuplicates()** to remove based on all columns

Java

|  |  |
| --- | --- |
| 1  2  3 | val distinctDF = dfEmployee.dropDuplicates() |

or to **deduplicate** rows based on **selected multiple columns**:

Java

|  |  |
| --- | --- |
| 1  2  3 | df1.dropDuplicates("name").show() |

**Output:**

Java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | +-----+---+-------+  | name|age| salary|  +-----+---+-------+  | John| 34|45000.0|  |Peter| 25|35000.0|  +-----+---+-------+ |

Q05. How do you get the count by name?  
Q05. Using the “**groupBy**”

Java

|  |  |
| --- | --- |
| 1  2  3 | df1.groupBy("name").count().show() |

**Output:**

Java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | +-----+-----+  | name|count|  +-----+-----+  | John|    1|  |Peter|    2|  +-----+-----+ |

Q06. How do you get the distinct name counts?  
A06. Function countDistinct(…) from org.apache.spark.sql.functions.\_

Java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | %scala      import org.apache.spark.sql.functions.\_    case class Employee(name: String, age: Int, salary: Double)    // Create the Employees  val employee1 = new Employee("Peter", 25, 35000.00)  val employee2 = new Employee("Peter", 26, 42000.00)  val employee3 = new Employee("John", 34, 45000.00)    val employees = Seq(employee1, employee2, employee3)  val df1 = employees.toDF()    df1.select(countDistinct("name")).show()  //countDistinct from functions package |

**Output:**

Java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | +--------------------+  |count(DISTINCT name)|  +--------------------+  |                   2|  +--------------------+ |

Q07. How do you aggregate salary by name?  
A07. Use “**groupBy(…)**” and “**agg()/sum()**”

Java

|  |  |
| --- | --- |
| 1  2  3 | df1.groupBy("name").sum("salary").show(); |

or

Java

|  |  |
| --- | --- |
| 1  2  3 | df1.groupBy("name").agg(sum("salary")).show(); |

**Output:**

Java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | +-----+-----------+  | name|sum(salary)|  +-----+-----------+  | John|    45000.0|  |Peter|    77000.0|  +-----+-----------+ |

Q08. How do you calculate average salary by name?  
A08. Use **agg()/avg()**.

Java

|  |  |
| --- | --- |
| 1  2  3 | df1.groupBy("name").agg(avg("salary")).show(); |

**Output:**

Java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | +-----+-----------+  | name|avg(salary)|  +-----+-----------+  | John|    45000.0|  |Peter|    38500.0|  +-----+-----------+ |

Q09. How will you aggregate salary by name & age with different combinations?  
A09. Use “**cube(….)**”

Java

|  |  |
| --- | --- |
| 1  2  3 | df1.cube("name", "age").sum().show(); |

**Output:**

Java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | +-----+----+--------+-----------+  | name| age|sum(age)|sum(salary)|  +-----+----+--------+-----------+  | null|null|      85|   122000.0|  |Peter|null|      51|    77000.0|  | null|  25|      25|    35000.0|  |Peter|  26|      26|    42000.0|  | null|  34|      34|    45000.0|  |Peter|  25|      25|    35000.0|  | John|  34|      34|    45000.0|  | null|  26|      26|    42000.0|  | John|null|      34|    45000.0|  +-----+----+--------+-----------+ |

Q10. How does rollup() differ from cube()?  
A10. **rollup(..)** returns a subset of cube(..). It computes hierarchical subtotals from left to right.

Java

|  |  |
| --- | --- |
| 1  2  3 | df1.rollup("name", "age").sum().show(); |

**Output:**

Java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | +-----+----+--------+-----------+  | name| age|sum(age)|sum(salary)|  +-----+----+--------+-----------+  | null|null|      85|   122000.0|  |Peter|null|      51|    77000.0|  |Peter|  26|      26|    42000.0|  |Peter|  25|      25|    35000.0|  | John|  34|      34|    45000.0|  | John|null|      34|    45000.0|  +-----+----+--------+-----------+ |

Q11. How will you rank salary by name?  
A11. **Window** aggregate functions to the rescue. These are functions that perform a calculation over a group of records called **window** that are in some relation to the current record.

Java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | import org.apache.spark.sql.expressions.Window    val byName = Window.partitionBy('name').orderBy('salary desc')    df1.withColumn("rank\_by\_name", rank().over(byName)).show() |

**Note:** Use “**withColumn**” to add a new column.

**Output:**

Java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | +-----+---+-------+------------+  | name|age| salary|rank\_by\_name|  +-----+---+-------+------------+  | John| 34|45000.0|           1|  |Peter| 26|42000.0|           1|  |Peter| 25|35000.0|           2|  +-----+---+-------+------------+ |

Q12. How will you display the average salary by name?  
A12.

Java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | import org.apache.spark.sql.expressions.Window    val byName = Window.partitionBy('name')    df1.withColumn("avg\_salary\_name", avg("salary").over(byName)).show() |

**Output:**

Java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | +-----+---+-------+---------------+  | name|age| salary|avg\_salary\_name|  +-----+---+-------+---------------+  | John| 34|45000.0|        45000.0|  |Peter| 25|35000.0|        38500.0|  |Peter| 26|42000.0|        38500.0|  +-----+---+-------+---------------+ |

Spark Scala on Databricks notebook

You can easily get started on Databricks to practice more examples with Scala by following [Getting started with Spark on Databricks](https://www.java-success.com/category/java-success-com/big-data-tutorials/tutorials-databricks/).

**Spark interview Q&As with coding examples in Scala – part 2**

Posted on [October 5, 2020](https://www.java-success.com/spark-interview-qas-with-coding-examples-in-scala-part-2/)

This extends [Spark interview Q&As with coding examples in Scala – part 1](https://www.java-success.com/spark-interview-qas-with-coding-examples-in-scala-part-1/) with the key optimisation concepts.

**Partition Pruning**

Q13. What do you understand by the concept **Partition Pruning**?  
A13. Spark & Hive table partitioning by year, month, country, department, etc will optimise **reads** by storing files in a hierarchy of directories based on the partitioning keys, hence reducing the amount of I/O needed to process your query/data. This will prevent full scanning of data by reading data only from a list of partitions, based on a filter on the partitioning key, skipping the rest.

For example, you can use active\_flag, year & month as keys to store large volume data.

Spark partition pruning

If the files are stored as above, and when you do the below operation only “**/my/base/folder/active\_flag=Y/**” and its year & month subfolders & parquet files will be scanned. The path to “active\_flag=N” will be ignored. This prevents full scan.

Java

|  |  |
| --- | --- |
| 1  2  3 | df.filter($"active\_flag" === "Y") |

Only “**/my/base/folder/active\_flag=Y/year=2020/**” and its month subfolders & parquet files will be scanned.

Java

|  |  |
| --- | --- |
| 1  2  3 | df.where($"active\_flag" === "Y" && year === "2020") |

Only “**/my/base/folder/active\_flag=Y/year=2020/month=01/**” and parquet files will be scanned.

Java

|  |  |
| --- | --- |
| 1  2  3 | df.where($"active\_flag" === "Y" && year === "2020" && month === 01) |

Q14. How will you write a dataframe data partitioned?  
A14. Using the “**partitionBy**” keyword.

Java

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | df.write.format("parquet") \    .partitionBy("active\_flag") \    .option("path", "/my/base/folder") \    .saveAsTable("mytable") |

Passing multiple columns:

Java

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | df.write.format("parquet") \    .partitionBy("active\_flag", "year", "month") \    .option("path", "/my/base/folder") \    .saveAsTable("mytable") |

**Predicate Push Down**

A “predicate” in maths & functional programming is a function that returns a boolean value. A predicate in SQL is a **WHERE** clause used to **filter** data. In general, a JOIN is performed before filtering the data in a WHERE clause. Predicate Push Down is an optimization technique used to apply filtering before a join to avoid loading unnecessary data into memory.

Q15. What is **Predicate/Filter Push Down**?  
A15. When you execute **where** or **filter** operators right after reading a dataset, and if partition filters are present the Spark catalyst optimizer pushes down the partition filters as shown below. This means the scan reads only the sub directories that match the partition filters as in active\_flag/2020/01/, hence reducing the disk I/O.

Predicate or Filter Push Down

**Dynamic Partitioning**

Q16. In reality, you will have a large **fact** table like orders & a small **dimension** table like products. Can you apply static partition pruning in this scenario?

Java

|  |  |
| --- | --- |
| 1  2  3  4 | SELECT \* from ORDERS o  JOIN PRODUCTS p on o.product\_id = p.product\_id  WHERE p.product\_category = "WHITE\_GOODS"; |

A16. In the above example, the fact table ORDERS will be a huge table with millions of rows & PRODUCTS is a small dimension table with 100K records. The static partition pruning is not beneficial on the small PRODUCTS table. The table that is more appealing and more attractive to pruning is the huge ORDERS table, which is partitioned by **order\_date** & **product\_id**.

You can first join the small dimension table with the huge fact table to an intermediate table with a static partitioning on “product\_category”. The obvious downside is that the join operation to create an intermediate table with all rows from both tables will be an expensive one. You are also **duplicating** the whole data with an intermediate table.

Spark 3.0 has introduced **Dynamic Partition Pruning** to optimise this type of scenarios by taking the filtered results from the dimension table, and then using them directly to limit the data from the fact table.

**Conditions** for Dynamic Partition Pruning are:

1) The tables that need to be pruned (i.e. often larger fact table) **must be partitioned by any one of the join key columns**. In the above example, the ORDERS table is partitioned by “product\_id”.

2) It works only with **equi-joins**(i.e. “=”). You cannot use it for p.product\_category != “WHITE\_GOODS”

The steps involved are:

1) Scan the dimension table PRODUCTS & apply filter (i.e. p.product\_category != “WHITE\_GOODS”). If the dimension table PRODUCTS is partitioned, then filter applied is pushed down before the scan process in dimension scan.

2) Spark creates an inner subquery from the dimension table PRODUCTS, which is broadcasted and hashed across all the executors. This subquery is meant for pruning unwanted partitions from the fact table ORDERS in the scanning phase.

3) Join only the selected partitions from the fact table ORDERS with the filtered dimension table PRODUCTS.

**Constant Folding**

Q17. What is Constant Folding?  
A17. Constant Folding is a operator optimization rule that replaces expressions that can be statically evaluated with their equivalent literal values.

Java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | %scala      import org.apache.spark.sql.functions.\_    val df1 = spark.range(5).select(lit(3) \* 2)  df1.show() |

**Output:**

Java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | +-------+  |(3 \* 2)|  +-------+  |      6|  |      6|  |      6|  |      6|  |      6|  +-------+ |

It is better to compute expression lit(3) \* 2 once, and then repeat 6 for each row. This is what the “constant folding” does.

Java

|  |  |
| --- | --- |
| 1  2  3 | df1.explain(true) // true means logical & physical plan |

**Output:**

Java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | == Parsed Logical Plan ==  Project [(3 \* 2) AS (3 \* 2)#155]  +- Range (0, 5, step=1, splits=Some(8))    == Analyzed Logical Plan ==  (3 \* 2): int  Project [(3 \* 2) AS (3 \* 2)#155]  +- Range (0, 5, step=1, splits=Some(8))    == Optimized Logical Plan ==  Project [6 AS (3 \* 2)#155]  +- Range (0, 5, step=1, splits=Some(8))    == Physical Plan ==  \*(1) Project [6 AS (3 \* 2)#155]  +- \*(1) Range (0, 5, step=1, splits=8) |

In the optimized logical plan & physical plan it says “Project [**6** AS (3 \* 2)#155]”. The Spark execution has the below phases:

source: https://www.learntospark.com/2020/02/spark-sql-catalyst-optimizer.html

Q18. What is a UDF? Is it a good practice to use them?  
A18. **UDF**s stands for “User Defined Functions”.

Java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | %scala      //my own udf fuction  val my\_upper: String => String = \_.toUpperCase      //register my function  import org.apache.spark.sql.functions.udf    val upperUDF = udf(my\_upper)    val df1 = Seq(    (1, "Aplle"),    (2, "Melon"),    (3, "Orange")  ).toDF("number", "word")    //using my udf upperUDF  val df2 = df1.withColumn("upper", upperUDF($"word"))  df2.show() |

**Output:**

Java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | +------+------+------+  |number|  word| upper|  +------+------+------+  |     1| Aplle| APLLE|  |     2| Melon| MELON|  |     3|Orange|ORANGE|  +------+------+------+ |

**Favor Spark SQL functions over UDFs** because Spark treats UDFs as blackbox, which result in losing many optimisations like: Predicate pushdown, Constant folding and many others.

The above can be achieved via built-in function “org.apache.spark.sql.functions.**upper**”

Java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | %scala      import org.apache.spark.sql.functions.upper    val df1 = Seq(    (1, "Aplle"),    (2, "Melon"),    (3, "Orange")  ).toDF("number", "word")    val df2 = df1.withColumn("upper", upper($"word")) // using built-in function upper(...)  df2.show() |

Avoiding UDFs is not always possible as not all functionality exists in Apache Spark functions.

**Column Projection**

Q19. What is Column Projection in Spark?  
A19. A “**Column Projection (i.e. selection)**” is to read only the required columns and skip the rest. For example, if you have a table with 100 columns, and your query requires only ten, then specify only those 3 columns in your select.

Java

|  |  |
| --- | --- |
| 1  2 | df1.select($"col1", $"col5", $"col99") |

Column oriented data formats like **Parquet** naturally stores data in a columnar fashion to save on disk I/O.

# Spark interview Q&As with coding examples in Scala – part 3

Posted on [October 8, 2020](https://www.java-success.com/spark-interview-qas-with-coding-examples-in-scala-part-3/)

This extends [Spark interview Q&As with coding examples in Scala – part 2](https://www.java-success.com/spark-interview-qas-with-coding-examples-in-scala-part-2/) with more coding examples on Databricks Note book.

**Prerequisite**: Create a free account as per [Databricks getting started](https://www.java-success.com/01-databricks-getting-started-pyspark/). Login to [community.cloud.databricks.com](https://community.cloud.databricks.com/login.html), and click on “**Clusters**” to create a Spark cluster.

Databricks Spark Cluster

You can now create a new workspace to write Spark code & attach it to this cluster. You can write the below Scala code in the workspace & run.

Q20. Given the below code, how will the data gets partitioned in the underlying file system?

Java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34 | %scala      case class Employee(name: String, age: Int, salary: Double, department: String)    // Create the Employees  val employee1 = new Employee("Peter", 25, 35000.00, "ENGINEERING")  val employee2 = new Employee("Rob", 26, 42000.00, "FINANCE")  val employee3 = new Employee("John", 34, 45000.00, "SALES")  val employee4 = new Employee("Sam", 34, 25000.00, "ENGINEERING")    val employees = Seq(employee1, employee2, employee3, employee4)  val df = employees.toDF()    df.write.format("parquet").partitionBy("department").option("path", "/employees").saveAsTable("my\_employees")  %scala      case class Employee(name: String, age: Int, salary: Double, department: String)    // Create the Employees  val employee1 = new Employee("Peter", 25, 35000.00, "ENGINEERING")  val employee2 = new Employee("Rob", 26, 42000.00, "FINANCE")  val employee3 = new Employee("John", 34, 45000.00, "SALES")  val employee4 = new Employee("Sam", 34, 25000.00, "ENGINEERING")    val employees = Seq(employee1, employee2, employee3, employee4)  val df = employees.toDF()    df.write.format("parquet")    .partitionBy("department")    .option("path", "/employees")    .saveAsTable("my\_employees") |

A20. If you click on “**Data**“, you can see the “**my\_employees**” table created.

Databricks my\_employees table

#### Partition Pruning

Click on “**my\_employees**” to view the schema & data.

Databricks my\_employees table

Under “# Partition Information”, “# col\_name” you can see “**department**” as the partition column.

Under “**Data** –> **Add Data**” –> **DBFS** (i.e. DataBricks File System) you can see the partitioned folders as shown below.

Databricks – DBFS partitioned

**/employees/department=ENGINEERING/** under which the parquet files for that department will be stored.

Q21. How will you read the stored data into a new Dataframe?  
A21. Read from the base folder path “**/employees**“. **Note**: Add a new cell with “**Add Cell Below**” in Databricks workspace as depicted below.

Databricks Notebook Commands

Java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | %scala      val readDF = spark.read.parquet("/employees");  readDF.show() |

**Output:**

Java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | +-----+---+-------+-----------+  | name|age| salary| department|  +-----+---+-------+-----------+  |Peter| 25|35000.0|ENGINEERING|  | John| 34|45000.0|      SALES|  |  Rob| 26|42000.0|    FINANCE|  |  Sam| 34|25000.0|ENGINEERING|  +-----+---+-------+-----------+ |

#### Predicate or Filter Pushdown

Q22. How will you demonstrate **predicate pushdown**?  
A22. If you use **filter** or **where** by “department”

Java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | %scala      val filteredDF = readDF.filter($"department" === "ENGINEERING");  filteredDF.show(); |

**Output:**

Java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | +-----+---+-------+-----------+  | name|age| salary| department|  +-----+---+-------+-----------+  |Peter| 25|35000.0|ENGINEERING|  |  Sam| 34|25000.0|ENGINEERING|  +-----+---+-------+-----------+ |

##### explain plan

Review the logical & physical plan where you can see “**PushedFilters:**” in the “== **Physical Plan** ==” indicating that the data is filtered on read itself by only reading from the subfolder “/employees/**department=ENGINEERING**/” to save IO.

Java

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | %scala      filteredDF.explain(true)  //true for physical plan in addition to logical |

**Output:**

Java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | == Parsed Logical Plan ==  'Filter ('department = ENGINEERING)  +- Relation[name#1152,age#1153,salary#1154,department#1155] parquet    == Analyzed Logical Plan ==  name: string, age: int, salary: double, department: string  Filter (department#1155 = ENGINEERING)  +- Relation[name#1152,age#1153,salary#1154,department#1155] parquet    == Optimized Logical Plan ==  Filter (isnotnull(department#1155) AND (department#1155 = ENGINEERING))  +- Relation[name#1152,age#1153,salary#1154,department#1155] parquet    == Physical Plan ==  \*(1) ColumnarToRow  +- FileScan parquet [name#1152,age#1153,salary#1154,department#1155] Batched: true, DataFilters: [], Format: Parquet, Location: InMemoryFileIndex[dbfs:/employees], PartitionFilters: [isnotnull(department#1155), (department#1155 = ENGINEERING)], PushedFilters: [], ReadSchema: struct<name:string,age:int,salary:double> |

**NOTE**: Spark uses “**lazy evaluation**“, which means the actual data is read from DBFS only when you invoke **actions** like filteredDF.show() or filteredDF.explain().

Q23. What will be the output for the below read?

Java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | %scala      val readDF = spark.read.parquet("/employees/department=SALES");  readDF.show() |

A23. Outputs the data by reading all the parquet files in the subfolder “/employees/**department=SALES**/”.

Q24. Can you run this as an SQL statement in the Notebook?  
A24. Yes.

Java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | %sql      SELECT \*  FROM my\_employees  WHERE department='SALES' |

**Output:**

Databricks workspace example

#### METADATA driven Static Pruning

Q25. Will the below code skip unnecessary files?

Java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | %scala      val readDF = spark.read.parquet("/employees")    val filteredDF = readDF.filter($"salary" > 40000.00)  filteredDF.show()    filteredDF.explain(true) |

A25. In addition to eliminating data at partition granularity as explained above with “department”, Delta Lake on Databricks dynamically skips unnecessary files where possible by automatically collecting metadata about data files.

In the above example, Delta Lake will collect metadata for each column, and for the “**salary**” column min & max salary amount will stored for each file, and when you query for [$”salary” > 40000.00], the files for “DEPARTMENT=ENGINEERING” will be skipped as its min (i.e. 25000.0) & max (i.e. 35000.0) value is outside 40000.00, hence will be skipped.

This can be further confirmed via “**filteredDF.explain(true)**“.

Java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | +----+---+-------+----------+  |name|age| salary|department|  +----+---+-------+----------+  |John| 34|45000.0|     SALES|  | Rob| 26|42000.0|   FINANCE|  +----+---+-------+----------+    == Parsed Logical Plan ==  'Filter ('salary > 40000.0)  +- Relation[name#114,age#115,salary#116,department#117] parquet    == Analyzed Logical Plan ==  name: string, age: int, salary: double, department: string  Filter (salary#116 > 40000.0)  +- Relation[name#114,age#115,salary#116,department#117] parquet    == Optimized Logical Plan ==  Filter (isnotnull(salary#116) AND (salary#116 > 40000.0))  +- Relation[name#114,age#115,salary#116,department#117] parquet    == Physical Plan ==  \*(1) Project [name#114, age#115, salary#116, department#117]  +- \*(1) Filter (isnotnull(salary#116) AND (salary#116 > 40000.0))     +- \*(1) ColumnarToRow        +- FileScan parquet [name#114,age#115,salary#116,department#117] Batched: true, DataFilters: [isnotnull(salary#116), (salary#116 > 40000.0)], Format: Parquet, Location: InMemoryFileIndex[dbfs:/employees], PartitionFilters: [], PushedFilters: [IsNotNull(salary), GreaterThan(salary,40000.0)], ReadSchema: struct<name:string,age:int,salary:double>    readDF: org.apache.spark.sql.DataFrame = [name: string, age: int ... 2 more fields]  filteredDF: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [name: string, age: int ... 2 more fields] |

You can see **PushedFilters: [IsNotNull(salary), GreaterThan(salary,40000.0)].**

#### Dynamic File Pruning

**Q26. Does Databricks support “Dynamic File Pruning”?  
A26. Dynamic File Pruning is for table joins where you have a huge fact table being probed is joined with a small dimension table. The filtered data from small table is dynamically broadcast to prune the huge table to skip files by scanning for only required files.**

**It is automatically enabled in Databricks Runtime 6.1 and higher, and for it to be applied the query must meet the below criteria:**

**1) The table being joined is in Delta Lake format.  
2) The join type is either INNER or LEFT-SEMI  
3) The join strategy is BROADCAST HASH JOIN  
4) Meet the below configuration settings & thresholds:**

**a) spark.databricks.optimizer.dynamicFilePruning (default=True)**

**b) spark.databricks.optimizer.deltaTableSizeThreshold (default is 10GB) minimum threshold on the table being probed to trigger Dynamic File Pruning.**

**c) spark.databricks.optimizer.deltaTableFilesThreshold (default is 1000) minimum threshold on the table being probed to trigger Dynamic File Pruning.**

#### Spark 3.0 Dynamic Partition Pruning (i.e. DPP)

**Q. When is DPP applied in Spark 3.0 onwards? What properties control the behavior?  
A. As explained above DPP improves job performance by dynamically inferring at runtime the specific partitions within a table that need to be read and processed for a specific query. By reducing the amount of data read and processed, significant time is saved in job execution.**

**When spark.sql.optimizer.dynamicPartitionPruning.enabled is set to true, which is the default, then the DPP will apply on the query, if the query itself is eligible. The second property is spark.sql.optimizer.dynamicPartitionPruning.reuseBroadcastOnly, which is a boolean flag controlling the use of the DPP. If this property is set to true (i.e. default), the DPP will apply only when the BroadcastExchange can be reused in the dynamic pruning filter.**

**The spark.sql.optimizer.dynamicPartitionPruning.useStats, defines whether the distinct count of the join attribute should be used, and the spark.sql.optimizer.dynamicPartitionPruning.fallbackFilterRatio sets the fallback value to use in the algorithm when the stats are disabled or unavailable.**

**Spark interview Q&As with coding examples in Scala – part 4**

Posted on [October 10, 2020](https://www.java-success.com/spark-interview-qas-with-coding-examples-in-scala-part-4/)

This extends [Spark interview Q&As with coding examples in Scala – part 3](https://www.java-success.com/spark-interview-qas-with-coding-examples-in-scala-part-3/) with more coding examples on Databricks Note book.

**Prerequisite**: Create a free account as per [Databricks getting started](https://www.java-success.com/01-databricks-getting-started-pyspark/). Login to [community.cloud.databricks.com](https://community.cloud.databricks.com/login.html), and click on “**Clusters**” to create a Spark cluster.

**In-memory lazy computation**

Q27. One of the key reasons why Apache Spark has made Hadoop MapReduce obsolete is that

“Spark computes **in-memory**, whilst Hadoop MapReduce has to read from and write to a disk. As a result, the speed of processing differs significantly. **Apache Spark can be up to 100 time faster** than MapReduce”

If Apache Spark processing in-memory, why do you need **cacheing** in Spark?

A27. Apache Spark operations are **lazy**. The

**Line 1:** read.parquet(…) is lazy, which states that the files in the folder need to be loaded, but the files are NOT loaded at this point.

**Line 2:** show() cannot be lazy as it requires observing the contents of the data. At this point the files in the folder “/employees” will be read, and the lines will displayed on the console. These operations are known as **actions**.

**Line 3:** count() cannot be lazy as it requires to count the number of rows. This is also an action. But, it will be executing the Line 1 again to read the data again from the folder “/employees”. This is very inefficient.

Java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | %scala    val readDF = spark.read.parquet("/employees")          //Line 1  readDF.show()                                          //Line 2 - Action    readDF.count()                                         //Line 3 - Action |

How can you prevent this loading twice? This is where **df.cache()** is useful. cache() is also a lazy operation.

When **Line 3:** readDF.show() action is executed the data will be loaded, **cached**, and displayed.

When **Line 4:** is executed the contents are read from cache and count the lines.

Java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | val readDF = spark.read.parquet("/employees")          //Line 1: Lazy  readDF.cache()                                         //Line 2: Lazy  readDF.show()                                          //Line 3 - Action    readDF.count()                                         //Line 4 - Action |

**Cacheing**

Q28. When should you cache in your code?  
A28. Identify the Dataframes that you will be reusing in your Spark application and cache them to prevent any re-computations.

1) Reading data from file systems (E.g. HDFS) & persistence object stores (E.g. AWS S3) requires expensive I/O operations. So, after you read data from the source and apply all the common operations/transformations, cache it if you are going to reuse the data.

2) By caching you create a **checkpoint** in your Spark application. This means, if any of the operations **fail** further down the execution, the lost Dataframes will be recomputed from the cache. There is no need to reload the data.

Databricks Cache

Q29. What if you don’t have enough memory?  
A29. If data does not fit the memory, it will be spilled into the local disk of executor which will be faster than reading from the source.

The readDF.cache() method invokes the readDF.**persist**(StorageLevel.MEMORY\_AND\_DISK). “**MEMORY\_AND\_DISK**” means spill to disk if does not fit in the memory.

Instead of cache(), you can invoke readDF.persist(…) various options like DISK\_ONLY, MEMORY\_ONLY, MEMORY\_AND\_DISK, and OFF\_HEAP. You can also replicate data with DISK\_ONLY\_2, MEMORY\_AND\_DISK\_2, MEMORY\_ONLY\_SER\_2, etc. “\_SER” is for serializing the data while storing, which saves memory footprint at the expense of additional processing.

Q30. Where do you check for cached data in Databricks?  
A30. Go to Clusters –> Spark UI –> Storage

Databricks Spark UI Cache

**Note**: Databricks support two types of cacheing. Delta cacheing & Apache Spark cacheing. The Delta cache is stored entirely on the local disk, so that memory is not taken away from other operations within Spark. Due to the high read speeds of modern SSDs, the Delta cache can be fully disk-resident without a negative impact on its performance.

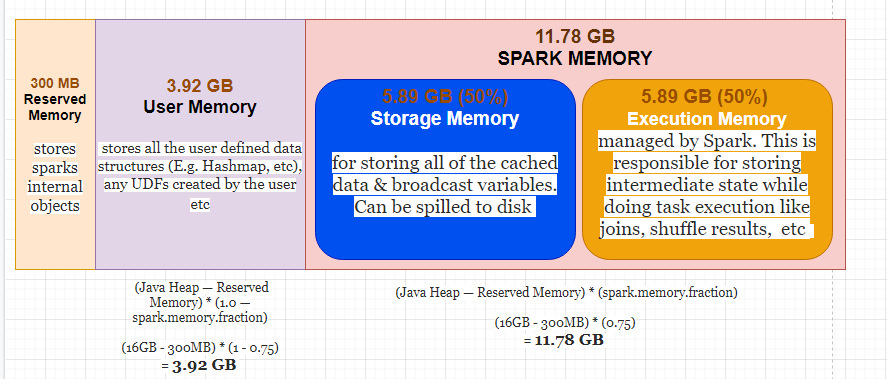
The Delta cache automatically detects when data files are created or deleted and updates its content accordingly. You can write, modify, and delete table data with no need to explicitly invalidate cached data.

You can check the current state of the Delta cache on each of the executors in the Storage tab in the Spark UI.

When a node reaches 100% disk usage, the cache manager discards the least recently used (i.e. **LRU**) cache entries to make space for new data.

**Spark memory Mgmt**

If you don’t understand Spark memory management, you will be dealing with **java.lang.OutOfMemoryError**.

[](https://www.java-success.com/wp-content/uploads/2020/09/Spark_executor_memory_2.png)

Spark Executor Memory Distribution

**Spark interview Q&As with coding examples in Scala – part 5**

Posted on [October 11, 2020](https://www.java-success.com/spark-interview-qas-with-coding-examples-in-scala-part-5/)

This extends [Spark interview Q&As with coding examples in Scala – part 4](https://www.java-success.com/spark-interview-qas-with-coding-examples-in-scala-part-4/) with more coding examples on Databricks Note book.

**Prerequisite**: Create a free account as per [Databricks getting started](https://www.java-success.com/01-databricks-getting-started-pyspark/). Login to [community.cloud.databricks.com](https://community.cloud.databricks.com/login.html), and click on “**Clusters**” to create a Spark cluster.

Given the below Data:

Databricks Employee Data

**Spark Transformations Vs. Actions**

Q31. What will be the output of the below code?

Java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | %scala    import org.apache.spark.sql.functions.\_  import org.apache.spark.sql.Row    val df1 = spark.read.parquet("/employees")  val df2 = df1.groupBy($"department").agg(sum($"salary").alias("dept\_total\_salary"))    val df3 = df2.filter($"dept\_total\_salary" > 42000.00)    val df4 = df3.map(r => r.getDouble(1) \* 1.1 )  val result:Array[Double]  =  df4.collect()     // action that triggers the above transformations    val count = df4.count()                        // action that re-triggers the above transformations |

**Note:** “r.getAs[Double](“dept\_total\_salary”)” is an alternative to “r.getDouble(1)”

A31. The output will be:

Java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | df1: org.apache.spark.sql.DataFrame = [name: string, age: int ... 2 more fields]  df2: org.apache.spark.sql.DataFrame = [department: string, dept\_total\_salary: double]  df3: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [department: string, dept\_total\_salary: double]  df4: org.apache.spark.sql.Dataset[Double] = [value: double]  result: Array[Double] = Array(66000.0, 49500.00000000001)  count: Long = 2 |

Q32. What do you understand by the terms “transformations” & “actions”? In the above code identify transformations & actions?  
A32. Spark has two types of operations.

1) **Transformations** refer to the operations applied on a Dataframe to create new Dataframe. In the above code groupBy, filter & map are transformations.

2) **Actions** refer to operations which are also applied on Dataframes, that instruct Spark to **perform computation** and send the result back to driver. In the above code collect() & count() are actions.

**Important**: Transformations are lazy in nature, which means they get executed only when we call an action like reduce(), show(), count(), collect(), etc. If you comment below lines the transformations will NOT be executed.

Java

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | //....  val result:Array[Double]  =  df4.collect()  val count = df4.count()  //... |

Q33. Is count($”department”) in below query a transformation or action?

Java

|  |  |
| --- | --- |
| 1  2  3 | val df2 = df1.groupBy($"department").agg(count($"department").alias("dept\_count")) |

A33. It is calling groupBy on Dataframe which returns **RelationalGroupedDataset** object, and count is invoked on grouped Dataset which returns a Dataframe, so its a **transformation** since it doesn’t gets the data to the driver.

Q34. Given the below operations, which ones are transformations & which ones are actions?

flatMap, mapValues, mapPartitions, sample, union, join, distinct, coalesce, getNumPartitions & reduce

A34. Except for **getNumPartitions** & **reduce**, the rest are transformation operations.

**Pipelining Vs Shuffling**

Q35. What do you understand by the terms narrow & wide transformations?  
A35. **Narrow Transformations** are those where each input partition will contribute to only one output partition. For example, map, flatMap, filter, sample, union, mapPartitions, etc are narrow transformations.

[Spark Narrow Vs. Wide transformations source: https://stackoverflow.com/questions/42799322/how-spark-realize-which-rdd-operation-need-to-be-split-into-seperate-stage]

**Wide transformation** will have input partitions contributing to many output partitions. This is known as a **shuffle** where Spark will exchange partitions across the cluster. With narrow transformations, Spark will automatically perform an operation called **pipelining** on narrow dependencies, this means that if we specify multiple filters on DataFrames they’ll all be performed in-memory. The same cannot be said for shuffles. When you perform a shuffle, Spark may write the results to disk. For example, join, distinct, reduceByKey, groupByKey, repartition, coalesce, intersection, etc are wide transformations.

**Shuffling** creates new **Stages**. More stages means more Data shuffling, which could lead to performance issues.

Spark Pipelining Vs Shuffling. Shuffling creates new stages

Q36. How will you identify contents & partitions of each Dataframe?  
A36. You can use the actions **show()** & **rdd.getNumPartitions**.

Java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | %scala    import org.apache.spark.sql.functions.\_  import org.apache.spark.sql.Row    val df1 = spark.read.parquet("/employees")    println("partitions: " + df1.rdd.getNumPartitions)  df1.show()      val df2 = df1.groupBy($"department")               .agg(sum($"salary").alias("dept\_total\_salary"))    println("partitions: " + df2.rdd.getNumPartitions)  df2.show()    val df3 = df2.filter($"dept\_total\_salary" > 42000.00)    println("partitions: " + df3.rdd.getNumPartitions)  df3.show()    val df4 = df3.map(r => r.getAs[Double]("dept\_total\_salary") \* 1.1 )    println("partitions: " + df4.rdd.getNumPartitions)  df4.show() |

**Output:**

Java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35 | partitions: 4  +-----+---+-------+-----------+  | name|age| salary| department|  +-----+---+-------+-----------+  |Peter| 25|35000.0|ENGINEERING|  | John| 34|45000.0|      SALES|  |  Rob| 26|42000.0|    FINANCE|  |  Sam| 34|25000.0|ENGINEERING|  +-----+---+-------+-----------+    partitions: 1  +-----------+-----------------+  | department|dept\_total\_salary|  +-----------+-----------------+  |ENGINEERING|          60000.0|  |      SALES|          45000.0|  |    FINANCE|          42000.0|  +-----------+-----------------+    partitions: 1  +-----------+-----------------+  | department|dept\_total\_salary|  +-----------+-----------------+  |ENGINEERING|          60000.0|  |      SALES|          45000.0|  +-----------+-----------------+    partitions: 1  +-----------------+  |            value|  +-----------------+  |          66000.0|  |49500.00000000001|  +-----------------+ |

Spark Wide & Narrow Transformations with partitions

Q37. What will be the output & no. of partitions for the below code? Why is BigDecimal used?

Java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | %scala    import org.apache.spark.sql.functions.\_  import org.apache.spark.sql.Row    val df1 = spark.read.parquet("/employees")    println("partitions: " + df1.rdd.getNumPartitions)  df1.show()    val df2 = df1.map(r => (r.getAs[String]("department"),  (BigDecimal(r.getAs[Double]("salary")) \* BigDecimal(1.1)).doubleValue()))  // creates a tuple with values as \_1, \_2, etc    println("partitions: " + df2.rdd.getNumPartitions)  df2.show()    val df3 = df2.groupBy($"\_1")               .agg(sum($"\_2").alias("dept\_total\_salary"))               .withColumnRenamed("\_1", "department")    println("partitions: " + df3.rdd.getNumPartitions)  df3.show()    val df4 = df3.filter($"dept\_total\_salary" > 46200.00)    println("partitions: " + df4.rdd.getNumPartitions)  df4.show() |

A37. The above code will have the below output:

Java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37 | partitions: 4  +-----+---+-------+-----------+  | name|age| salary| department|  +-----+---+-------+-----------+  |Peter| 25|35000.0|ENGINEERING|  | John| 34|45000.0|      SALES|  |  Rob| 26|42000.0|    FINANCE|  |  Sam| 34|25000.0|ENGINEERING|  +-----+---+-------+-----------+    partitions: 4  +-----------+-------+  |         \_1|     \_2|  +-----------+-------+  |ENGINEERING|38500.0|  |      SALES|49500.0|  |    FINANCE|46200.0|  |ENGINEERING|27500.0|  +-----------+-------+    partitions: 1  +-----------+-----------------+  | department|dept\_total\_salary|  +-----------+-----------------+  |ENGINEERING|          66000.0|  |      SALES|          49500.0|  |    FINANCE|          46200.0|  +-----------+-----------------+    partitions: 1  +-----------+-----------------+  | department|dept\_total\_salary|  +-----------+-----------------+  |ENGINEERING|          66000.0|  |      SALES|          49500.0|  +-----------+-----------------+ |

Spark Narrow Vs Wide Transformations

**Q.** Why use BigDecimal?  
**A.** In A36. you can see the output for department SALES is 49500.0000000000**1** and NOT 49500.00. This is because multiplying floating point values causes this inaccuracy. Hence, you need to first convert to BigDecimal and then perform the multiplication operation as shown above.

**Q.** What other recommendations will you make to the above code?  
**A.** Cacheing will prevent the data being read from “/employees” many times.

Java

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | ...  val df1 = spark.read.parquet("/employees")  df1.cache()  ... |

**Q.** Where will you check for function names like “**withColumnRenamed**“?  
**A.** You need to look for the **API Docs**. Google for “**Spark 3 API**“, and select “API Docs” –> “Scala”. This will take you to the classes & methods.

Apache Spark API

RDDs and Datasets are type safe means that the compiler knows the Columns and it’s data type of the Column whether it is Long, String, etc. Sparak 2.0 onwards:

**Dataframe = Dataset[Row]**

The “**withColumnRenamed**” is a method in “org.apache.spark.sql.**Dataset[Row]**“. If you drill down into this you will see all the methods supported by a Dataset.

<https://spark.apache.org/docs/latest/api/scala/org/apache/spark/sql/Dataset.html>

**Actions**

Dataset API : Actions

**Basic Dataset functions**

Spark API: Basic Dataset functions

**Streaming & Typed Transformations**

Spark API: Streaming & Typed transformations

**Untyped Transformations**

Where you will see agg(…), withColumnRenamed(..), etc.

Spark API: Untyped transformations

**Spark interview Q&As with coding examples in Scala – part 6**

Posted on [October 12, 2020](https://www.java-success.com/spark-interview-qas-with-coding-examples-in-scala-part-6/)

This extends [Spark interview Q&As with coding examples in Scala – part 5](https://www.java-success.com/spark-interview-qas-with-coding-examples-in-scala-part-5/) with more coding examples on Databricks Note book.

**Prerequisite**: Create a free account as per [Databricks getting started](https://www.java-success.com/01-databricks-getting-started-pyspark/). Login to [community.cloud.databricks.com](https://community.cloud.databricks.com/login.html), and click on “**Clusters**” to create a Spark cluster.

Given the below Data:

Databricks Employee Data

**groupBy & collect\_list**

Q38. How will you create a list of employees by “department” as a key?  
A38. Firstly, use “**map**” to create key/value pairs, and then **groupBy/collect\_list**.

Java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | %scala    import org.apache.spark.sql.functions.\_  import org.apache.spark.sql.Row    val df1 = spark.read.parquet("/employees")  df1.cache()    case class Employee(name: String, age: Int, salary: BigDecimal)    val df2 = df1.map(r => (r.getAs[String]("department"), new Employee(r.getAs[String]("name"), r.getAs[Int]("age"), BigDecimal(r.getAs[Double]("salary")))))  val df3 = df2.withColumnRenamed("\_1", "department")               .withColumnRenamed("\_2", "employee")    val df4 = df3.groupBy($"department")               .agg(collect\_list($"employee") as "employees")    df4.show(false)   //false to not truncate output |

**Output:**

Java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | +-----------+----------------------------------------------------------------------------+  |department |employees                                                                   |  +-----------+----------------------------------------------------------------------------+  |ENGINEERING|[[Peter, 25, 35000.000000000000000000], [Sam, 34, 25000.000000000000000000]]|  |SALES      |[[John, 34, 45000.000000000000000000]]                                      |  |FINANCE    |[[Rob, 26, 42000.000000000000000000]]                                       |  +-----------+----------------------------------------------------------------------------+ |

**map**

“**map**” is applied to each element. One input is transformed to one output.

Q39. How will you apply 10% salary increase to all employees?  
A39. **Note**: r.getAs[Seq[Row]] is to get the Array of StructType.

Java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | ...  val df5 = df4.map(r =>         (           r.getAs[String]("department"),r.getAs[Seq[Row]]("employees").map(r =>                new Employee(r.getAs[String]("name"), r.getAs[Int]("age"),(r.getAs[java.math.BigDecimal]("salary").multiply(java.math.BigDecimal.valueOf(1.1)))))          )).withColumnRenamed("\_1", "department")            .withColumnRenamed("\_2", "employees")  df5.show(false) |

**Output:**

Java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | +-----------+----------------------------------------------------------------------------+  |department |employee                                                                    |  +-----------+----------------------------------------------------------------------------+  |ENGINEERING|[[Peter, 25, 38500.000000000000000000], [Sam, 34, 27500.000000000000000000]]|  |SALES      |[[John, 34, 49500.000000000000000000]]                                      |  |FINANCE    |[[Rob, 26, 46200.000000000000000000]]                                       |  +-----------+----------------------------------------------------------------------------+ |

**select & explode**

Q40. How will you convert array of employees shown above to one employee per row?  
A40. use the explode function.

Java

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | ....  import org.apache.spark.sql.functions.\_    val df6 = df5.select($"department", explode($"employees").alias("employee"))  df6.show(false) |

**Output:**

Java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | +-----------+-------------------------------------+  |department |employee                             |  +-----------+-------------------------------------+  |ENGINEERING|[Peter, 25, 38500.000000000000000000]|  |ENGINEERING|[Sam, 34, 27500.000000000000000000]  |  |SALES      |[John, 34, 49500.000000000000000000] |  |FINANCE    |[Rob, 26, 46200.000000000000000000]  |  +-----------+-------------------------------------+ |

**withColumn & drop**

Q41. How will you split each field of employee into separate columns as name, age & salary?  
A41.

Java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | val df7 = df6.withColumn("name", $"employee.name")               .withColumn("age", $"employee.age")               .withColumn("salary", $"employee.salary")               .drop($"employee")      df7.show(false) |

**Output:**

Java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | +-----------+-----+---+------------------------+  |department |name |age|salary                  |  +-----------+-----+---+------------------------+  |ENGINEERING|Peter|25 |38500.000000000000000000|  |ENGINEERING|Sam  |34 |27500.000000000000000000|  |SALES      |John |34 |49500.000000000000000000|  |FINANCE    |Rob  |26 |46200.000000000000000000|  +-----------+-----+---+------------------------+ |

**cast**

Q42. Why is the salary displayed to 18 decimal places? Can it be displayed at 2 decimal places?  
A42. By default spark will infer the schema of the Decimal type (or BigDecimal) in a case class to be DecimalType(38, 18).

Java

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | import org.apache.spark.sql.types.DecimalType    val df8 = df7.withColumn("salary", $"salary".cast(DecimalType(10,2)))    df8.show(false) |

**Output:**

Java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | +-----------+-----+---+--------+  |department |name |age|salary  |  +-----------+-----+---+--------+  |ENGINEERING|Peter|25 |38500.00|  |ENGINEERING|Sam  |34 |27500.00|  |SALES      |John |34 |49500.00|  |FINANCE    |Rob  |26 |46200.00|  +-----------+-----+---+--------+ |

**printSchema**

Q43. In Q39, how did you find it is “Array of StructType” to use r.getAs[Seq[Row]] A43. In Scala, StructType is mapped to org.apache.spark.sql.**Row** & Array is a “Seq”. You can use printSchema on the df4.

Java

|  |  |
| --- | --- |
| 1  2  3 | df4.printSchema() |

**Output:**

Java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | root  |-- department: string (nullable = true)  |-- employees: array (nullable = true)  |    |-- element: struct (containsNull = false)  |    |    |-- name: string (nullable = true)  |    |    |-- age: integer (nullable = false)  |    |    |-- salary: decimal(38,18) (nullable = true) |

“employees” is of type “array” & “element” is of type “struct”.

**Complete code on Databricks**

Spark Scala on Databricks coding

Java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48 | %scala    import org.apache.spark.sql.functions.\_  import org.apache.spark.sql.Row    val df1 = spark.read.parquet("/employees")  df1.cache()    case class Employee(name: String, age: Int, salary: BigDecimal)    val df2 = df1.map(r => (r.getAs[String]("department"), new Employee(r.getAs[String]("name"), r.getAs[Int]("age"), BigDecimal(r.getAs[Double]("salary")))))  val df3 = df2.withColumnRenamed("\_1", "department")               .withColumnRenamed("\_2", "employee")    val df4 = df3.groupBy($"department")               .agg(collect\_list($"employee") as "employees")    df4.show(false)    df4.printSchema()    import org.apache.spark.sql.types.DecimalType    val df5 = df4.map(r =>         (           r.getAs[String]("department"),r.getAs[Seq[Row]]("employees").map(r =>                new Employee(r.getAs[String]("name"), r.getAs[Int]("age"),(r.getAs[java.math.BigDecimal]("salary").multiply(java.math.BigDecimal.valueOf(1.1)))))          )).withColumnRenamed("\_1", "department")            .withColumnRenamed("\_2", "employees")  df5.show(false)    import org.apache.spark.sql.functions.\_    val df6 = df5.select($"department", explode($"employees").alias("employee"))  df6.show(false)    val df7 = df6.withColumn("name", $"employee.name")               .withColumn("age", $"employee.age")               .withColumn("salary", $"employee.salary")               .drop($"employee")    df7.show(false)    val df8 = df7.withColumn("salary", $"salary".cast(DecimalType(10,2)))    df8.show(false) |

**Spark interview Q&As with coding examples in Scala – part 7**

Posted on [October 13, 2020](https://www.java-success.com/spark-interview-qas-with-coding-examples-in-scala-part-7/)

This extends [Spark interview Q&As with coding examples in Scala – part 6](https://www.java-success.com/spark-interview-qas-with-coding-examples-in-scala-part-6/) with more coding examples on Databricks Note book.

**Prerequisite**: Create a free account as per [Databricks getting started](https://www.java-success.com/01-databricks-getting-started-pyspark/). Login to [community.cloud.databricks.com](https://community.cloud.databricks.com/login.html), and click on “**Clusters**” to create a Spark cluster.

Given the below Data:

Databricks Employee Data

**flatMap**

Q44. How will you display salary & 10% bonus for each employee names as separate rows?  
A44. **flatMap** is a narrow transformation that takes an element & returns an array, list or sequence. The flatMap() is used to produce multiple output elements for each input element.

Java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | %scala    import scala.math.BigDecimal    val df1 = spark.read.parquet("/employees")  df1.cache()    case class Employee(name: String, age: Int, salary: BigDecimal)    // flatMap returns a List of Tuples  val df2 = df1.flatMap(r => {      List( (r.getAs[String]("name"),BigDecimal(r.getAs[Double]("salary"))),            (r.getAs[String]("name") + "\_bonus", BigDecimal(r.getAs[Double]("salary"))\*BigDecimal(0.1)))  })    df2.show(false) |

**Output:**

Java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | +-----------+------------------------+  |\_1         |\_2                      |  +-----------+------------------------+  |Peter      |35000.000000000000000000|  |Peter\_bonus|3500.000000000000000000 |  |John       |45000.000000000000000000|  |John\_bonus |4500.000000000000000000 |  |Rob        |42000.000000000000000000|  |Rob\_bonus  |4200.000000000000000000 |  |Sam        |25000.000000000000000000|  |Sam\_bonus  |2500.000000000000000000 |  +-----------+------------------------+ |

**map**

Q44. What will be the output if you had use a map instead of flatMap?  
A44. The map() transformation takes in a function and applies it to each element in the RDD and the result of the function is a new value of each element in the resulting RDD.

Java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | %scala    import scala.math.BigDecimal    val df1 = spark.read.parquet("/employees")  df1.cache()    case class Employee(name: String, age: Int, salary: BigDecimal)    val df2 = df1.map(r => {      List( (r.getAs[String]("name"),BigDecimal(r.getAs[Double]("salary"))),            (r.getAs[String]("name") + "\_bonus", BigDecimal(r.getAs[Double]("salary"))\*BigDecimal(0.1))) // List of Tuples  })  df2.show(false) |

**Output:**

Java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | +---------------------------------------------------------------------------+  |value                                                                      |  +---------------------------------------------------------------------------+  |[[Peter, 35000.000000000000000000], [Peter\_bonus, 3500.000000000000000000]]|  |[[John, 45000.000000000000000000], [John\_bonus, 4500.000000000000000000]]  |  |[[Rob, 42000.000000000000000000], [Rob\_bonus, 4200.000000000000000000]]    |  |[[Sam, 25000.000000000000000000], [Sam\_bonus, 2500.000000000000000000]]    |  +---------------------------------------------------------------------------+ |

**mapPartitions**

Q45. What is the purpose of **mapPartitions**?  
A45. If you have 10000 elements distributed across 10 partitions i.e. 1000 elements/partition. The map() transformation will call the function 10000 times for each element, whereas **mapPartitions** will call the function 10 times for each partition. It returns the result once all partitions are executed & data is held in memory.

Java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | %scala    import scala.math.BigDecimal    val df1 = spark.read.parquet("/employees")  df1.cache()    case class Employee(name: String, age: Int, salary: BigDecimal)    val df2 = df1.flatMap(r => {      List( (r.getAs[String]("name"),BigDecimal(r.getAs[Double]("salary"))),            (r.getAs[String]("name") + "\_bonus", BigDecimal(r.getAs[Double]("salary"))\*BigDecimal(0.1))) // List of Tuples  })  df2.show(false)    //it is iterator  val df3 = df2.mapPartitions(it => {     //any expensive operation like getting a database connection     //for each partition       it.map(row => {       //for each element in the partition       (row.\_1 + " earned " + row.\_2)       })    })    df3.show(false) |

**Output:**

Java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | +-----------+------------------------+  |\_1         |\_2                      |  +-----------+------------------------+  |Peter      |35000.000000000000000000|  |Peter\_bonus|3500.000000000000000000 |  |John       |45000.000000000000000000|  |John\_bonus |4500.000000000000000000 |  |Rob        |42000.000000000000000000|  |Rob\_bonus  |4200.000000000000000000 |  |Sam        |25000.000000000000000000|  |Sam\_bonus  |2500.000000000000000000 |  +-----------+------------------------+    +--------------------------+  |value                     |  +--------------------------+  |Peter earned 35000.0      |  |Peter\_bonus earned 3500.00|  |John earned 45000.0       |  |John\_bonus earned 4500.00 |  |Rob earned 42000.0        |  |Rob\_bonus earned 4200.00  |  |Sam earned 25000.0        |  |Sam\_bonus earned 2500.00  |  +--------------------------+ |

**mapValues**

Q46. What is **mapValues**? How does it differ from map?  
A46. **mapValues** applicable to only PairRDDs with RDD[(key, value)] tuples. mapValues operates on the value only, whilst map operates on the entire record of tuple with key and value (E.g. RDD[(key, value)]).

Java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | %scala    import scala.math.BigDecimal    val df1 = spark.read.parquet("/employees")  df1.cache()    case class Employee(name: String, age: Int, salary: BigDecimal)    val df2 = df1.map(x => (x.getAs[String]("name"),BigDecimal(x.getAs[Double]("salary"))))    df2.show(false)    val rdd1 = df2.rdd.mapValues(x => x\*BigDecimal(1.1))    //Dataframe to RDD  val df3 = rdd1.toDF                                     //RDD to Dataframe    df3.show(false) |

**Output:**

Java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | +-----+------------------------+  |\_1   |\_2                      |  +-----+------------------------+  |Peter|35000.000000000000000000|  |John |45000.000000000000000000|  |Rob  |42000.000000000000000000|  |Sam  |25000.000000000000000000|  +-----+------------------------+    +-----+------------------------+  |\_1   |\_2                      |  +-----+------------------------+  |Peter|38500.000000000000000000|  |John |49500.000000000000000000|  |Rob  |46200.000000000000000000|  |Sam  |27500.000000000000000000|  +-----+------------------------+ |

# Spark interview Q&As with coding examples in Scala – part 8

Posted on [October 24, 2020](https://www.java-success.com/spark-interview-qas-with-coding-examples-in-scala-part-8/)

This extends [Spark interview Q&As with coding examples in Scala – part 7](https://www.java-success.com/spark-interview-qas-with-coding-examples-in-scala-part-7/) with more coding examples on Databricks Note book.

**Prerequisite**: Create a free account as per [Databricks getting started](https://www.java-success.com/01-databricks-getting-started-pyspark/). Login to [community.cloud.databricks.com](https://community.cloud.databricks.com/login.html), and click on “**Clusters**” to create a Spark cluster.

Given the below Data:

Databricks Employee Data

**Pivot**

Q47. How will you display salary across different departments?  
A47. “**pivot**” to the rescue.

Java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | %scala    import scala.math.BigDecimal    val df1 = spark.read.parquet("/employees")  df1.cache()    case class Employee(name: String, age: Int, salary: BigDecimal, department: String)    val df2 = df1.groupBy("name")               .pivot("department")               .sum("salary")    df2.show(false) |

**Output:**

Java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | +-----+-----------+-------+-------+  |name |ENGINEERING|FINANCE|SALES  |  +-----+-----------+-------+-------+  |John |null       |null   |45000.0|  |Sam  |25000.0    |null   |null   |  |Rob  |null       |42000.0|null   |  |Peter|35000.0    |null   |null   |  +-----+-----------+-------+-------+ |

If you already know the distinct values of **department**, then a more performant code would be

Java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | %scala    import scala.math.BigDecimal    val df1 = spark.read.parquet("/employees")  df1.cache()    case class Employee(name: String, age: Int, salary: BigDecimal, department: String)    val df2 = df1.groupBy("name")               .pivot("department", Seq("ENGINEERING", "FINANCE", "SALES"))               .sum("salary")    df2.show(false) |

Q48. How will you display total & average salaries by age & department?  
A48.

Java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | %scala    import scala.math.BigDecimal    val df1 = spark.read.parquet("/employees")  df1.cache()    case class Employee(name: String, age: Int, salary: BigDecimal, department: String)    val df2 = df1.groupBy("age")               .pivot("department")               .agg(sum("salary"), avg("salary"))    df2.show(false) |

**Output:**

Java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | +---+-----------------------+-----------------------+-------------------+-------------------+-----------------+-----------------+  |age|ENGINEERING\_sum(salary)|ENGINEERING\_avg(salary)|FINANCE\_sum(salary)|FINANCE\_avg(salary)|SALES\_sum(salary)|SALES\_avg(salary)|  +---+-----------------------+-----------------------+-------------------+-------------------+-----------------+-----------------+  |34 |25000.0                |25000.0                |null               |null               |45000.0          |45000.0          |  |26 |null                   |null                   |42000.0            |42000.0            |null             |null             |  |25 |35000.0                |35000.0                |null               |null               |null             |null             |  +---+-----------------------+-----------------------+-------------------+-------------------+-----------------+-----------------+ |

**Unpivot with selectExpr and stack**

Q49. How will you Unpivot?  
A49. **selectExpr** & **stack** to the rescue.

Java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | %scala    import scala.math.BigDecimal  import org.apache.spark.sql.functions.\_    val df1 = spark.read.parquet("/employees")  df1.cache()    case class Employee(name: String, age: Int, salary: BigDecimal, department: String)    val df2 = df1.groupBy("name")               .pivot("department", Seq("ENGINEERING", "FINANCE", "SALES"))               .sum("salary")    println("=======pivoted========")  df2.show(false)    val df3 = df2.selectExpr("name", "stack(3, 'ENGINEERING', ENGINEERING, 'FINANCE', FINANCE, 'SALES', SALES) as (Department,Salary)")                .filter($"Salary".isNotNull)  // remove nulls    println("=======unpivoted========")  df3.show(); |

**Output:**

Java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | =======pivoted========  +-----+-----------+-------+-------+  |name |ENGINEERING|FINANCE|SALES  |  +-----+-----------+-------+-------+  |John |null       |null   |45000.0|  |Sam  |25000.0    |null   |null   |  |Rob  |null       |42000.0|null   |  |Peter|35000.0    |null   |null   |  +-----+-----------+-------+-------+    =======unpivoted========  +-----+-----------+-------+  | name| Department| Salary|  +-----+-----------+-------+  | John|      SALES|45000.0|  |  Sam|ENGINEERING|25000.0|  |  Rob|    FINANCE|42000.0|  |Peter|ENGINEERING|35000.0|  +-----+-----------+-------+ |

**agg & first**

Q50. How will you create a pivot without aggregating the values?  
A50. If your goal is pivoting and not aggregation, then you can use **first** (or any other function not restricted to numeric values).

Java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | %scala    import scala.math.BigDecimal    val df1 = spark.read.parquet("/employees")  df1.cache()    case class Employee(name: String, age: Int, salary: BigDecimal, department: String)    val df2 = df1.groupBy("name")               .pivot("department", Seq("ENGINEERING", "FINANCE", "SALES"))               .agg(first("age"))    println("=======pivoted========")  df2.show(false) |

**Output:**

Java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | =======pivoted========  +-----+-----------+-------+-----+  |name |ENGINEERING|FINANCE|SALES|  +-----+-----------+-------+-----+  |John |null       |null   |34   |  |Sam  |34         |null   |null |  |Rob  |null       |26     |null |  |Peter|25         |null   |null |  +-----+-----------+-------+-----+ |

**agg, first & struct**

Q51. What if you want to display the age along with salary?  
A51.

Java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | %scala    import scala.math.BigDecimal      val df1 = spark.read.parquet("/employees")  df1.cache()    case class Employee(name: String, age: Int, salary: BigDecimal, department: String)    val df2 = df1.groupBy("name")               .pivot("department", Seq("ENGINEERING", "FINANCE", "SALES"))               .agg(first(struct("age", "salary")))    println("=======pivoted========")  df2.show(false) |

**Output:**

Java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | =======pivoted========  +-----+-------------+-------------+-------------+  |name |ENGINEERING  |FINANCE      |SALES        |  +-----+-------------+-------------+-------------+  |John |null         |null         |[34, 45000.0]|  |Peter|[25, 35000.0]|null         |null         |  |Rob  |null         |[26, 42000.0]|null         |  |Sam  |[34, 25000.0]|null         |null         |  +-----+-------------+-------------+-------------+ |

**agg & collect\_set or collect\_list**

Java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | %scala    import scala.math.BigDecimal      val df1 = spark.read.parquet("/employees")  df1.cache()    case class Employee(name: String, age: Int, salary: BigDecimal, department: String)    val df2 = df1.groupBy("name")               .pivot("department", Seq("ENGINEERING", "FINANCE", "SALES"))               .agg(collect\_set(struct("age", "salary")))      println("=======pivoted========")  df2.show(false) |

**Output:**

Java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | =======pivoted========  +-----+---------------+---------------+---------------+  |name |ENGINEERING    |FINANCE        |SALES          |  +-----+---------------+---------------+---------------+  |Peter|[[25, 35000.0]]|[]             |[]             |  |John |[]             |[]             |[[34, 45000.0]]|  |Rob  |[]             |[[26, 42000.0]]|[]             |  |Sam  |[[34, 25000.0]]|[]             |[]             |  +-----+---------------+---------------+---------------+ |

**Spark interview Q&As with coding examples in Scala – part 9**

Posted on [November 1, 2020](https://www.java-success.com/spark-interview-qas-with-coding-examples-in-scala-part-9/)

This extends [Spark interview Q&As with coding examples in Scala – part 8](https://www.java-success.com/spark-interview-qas-with-coding-examples-in-scala-part-8/) with more coding examples on Databricks Note book.

**Prerequisite**: Create a free account as per [Databricks getting started](https://www.java-success.com/01-databricks-getting-started-pyspark/). Login to [community.cloud.databricks.com](https://community.cloud.databricks.com/login.html), and click on “**Clusters**” to create a Spark cluster.

**org.apache.spark.sql.functions.\_**

Q52. What are the different types of functions in Spark?  
A52. There are 4 types of functions:

1) **Built-in functions:** from org.apache.spark.sql.**functions** like to\_date(Column e), to\_utc\_timestamp(Column e), etc. Take values from a single row as input and, and return single value for each input row.

The API document for “**org.apache.spark.sql.functions**” can be found at https://spark.apache.org/docs/latest/api/scala/org/apache/spark/sql/functions$.html.

Java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | %scala      import org.apache.spark.sql.functions.\_;  import org.apache.spark.sql.types.\_    val someDF = Seq((1)  ).toDF("seq")    // withColumn() creates a new column  // def lit(literal: Any): Column  // Creates a Column of literal value.  // def unix\_timestamp(s: Column, p: String): Column  // Converts time string with given pattern to Unix timestamp (in seconds).  // def cast(to: DataType): Column  //Casts the column to a different data type (i.e. SQL timestamp type).  val df2 = someDF.withColumn("local\_datetime", unix\_timestamp(lit("2017-08-01 14:30:00"), "yyyy-MM-dd HH:mm:ss").cast(TimestampType))    // def to\_utc\_timestamp(ts: Column, tz: Column): Column  // to\_utc\_timestamp(ts: Column, tz: Column): Column  // Given a timestamp like '2017-07-14 02:40:00.0', interprets it as a time in the given time zone,  // and renders that time as a timestamp in UTC. For example, 'GMT+1' would yield '2017-07-14 01:40:00.0'.  val df3 = df2.withColumn("utc\_datetime", to\_utc\_timestamp(df2("local\_datetime"), lit("GMT+10")))    df3.show() |

**Output:**

Java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | +---+-------------------+-------------------+  |seq|     local\_datetime|       utc\_datetime|  +---+-------------------+-------------------+  |  1|2017-08-01 14:30:00|2017-08-01 04:30:00|  +---+-------------------+-------------------+ |

2) **UDF**s (User Defined Functions): as the name implies defined by the users. Take values from a single row as input and, and return single value for each input row.

Java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | %scala      val someDF = Seq(("John", "Australia"),  ("Peter", "AUS"),  ("Sam", "AU"),  ("Paul", "AUSTRALIA")    ).toDF("first\_name", "country")    //define a user defined function  def standardiseCountry = (country: String) => {    val allPossibleAustralia = Seq("AU", "AUS", "AUSTRALIA")    if (allPossibleAustralia.contains(country.toUpperCase())) {      "AUS"    }    else {      "UNKNOWN"    }  }    //Register it  val normalisedCountry = spark.udf.register("standardisedCountry",standardiseCountry)    //Use it  val df2 = someDF.withColumn("normalised\_country", normalisedCountry($"country"))  df2.show() |

**Output:**

Java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | +----------+---------+------------------+  |first\_name|  country|normalised\_country|  +----------+---------+------------------+  |      John|Australia|               AUS|  |     Peter|      AUS|               AUS|  |       Sam|       AU|               AUS|  |      Paul|AUSTRALIA|               AUS|  +----------+---------+------------------+ |

3) Aggregate functions: like SUM, MAX, AVG, MIN, etc, which operate on a group of rows and calculate a single return value for every group.

Java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46 | %scala      import org.apache.spark.sql.functions.\_    val someDF = Seq(("John", "USA", 35000.00),  ("Peter", "AUS", 45000.00),  ("Sam", "AUS", 75000.00),  ("Paul", "USA", 85000.0)    ).toDF("first\_name", "country", "salary")    //max salary by country  someDF.groupBy($"country")        .agg(max($"salary"))        .show()    //min salary by country  someDF.groupBy($"country")        .agg(min($"salary"))        .show()    //count of salary by country  someDF.groupBy($"country")        .agg(count($"salary"))        .show()    //avg salary by country  someDF.groupBy($"country")        .agg(avg($"salary"))        .show()    //sum of salary by country  someDF.groupBy($"country")        .agg(sum($"salary"))        .show()    someDF.groupBy($"country")        .agg(max($"salary"), min($"salary"), count($"salary"), avg($"salary"), sum($"salary"))        .show()    someDF.groupBy($"country")        .agg(max($"salary").alias("max\_salary"), min($"salary").alias("min\_salary"), count($"salary").alias("salary\_count"), avg($"salary").alias("avg\_salary"), sum($"salary").alias("total\_salary"))        .show() |

**Output:**

Java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50 | +-------+-----------+  |country|max(salary)|  +-------+-----------+  |    USA|    85000.0|  |    AUS|    75000.0|  +-------+-----------+    +-------+-----------+  |country|min(salary)|  +-------+-----------+  |    USA|    35000.0|  |    AUS|    45000.0|  +-------+-----------+    +-------+-------------+  |country|count(salary)|  +-------+-------------+  |    USA|            2|  |    AUS|            2|  +-------+-------------+    +-------+-----------+  |country|avg(salary)|  +-------+-----------+  |    USA|    60000.0|  |    AUS|    60000.0|  +-------+-----------+    +-------+-----------+  |country|sum(salary)|  +-------+-----------+  |    USA|   120000.0|  |    AUS|   120000.0|  +-------+-----------+    +-------+-----------+-----------+-------------+-----------+-----------+  |country|max(salary)|min(salary)|count(salary)|avg(salary)|sum(salary)|  +-------+-----------+-----------+-------------+-----------+-----------+  |    USA|    85000.0|    35000.0|            2|    60000.0|   120000.0|  |    AUS|    75000.0|    45000.0|            2|    60000.0|   120000.0|  +-------+-----------+-----------+-------------+-----------+-----------+    +-------+----------+----------+------------+----------+------------+  |country|max\_salary|min\_salary|salary\_count|avg\_salary|total\_salary|  +-------+----------+----------+------------+----------+------------+  |    USA|   85000.0|   35000.0|           2|   60000.0|    120000.0|  |    AUS|   75000.0|   45000.0|           2|   60000.0|    120000.0|  +-------+----------+----------+------------+----------+------------+ |

4) Window functions: are useful if you want to operate on a group of rows, but return a single value for every input row. For example, ranking a group of rows, calculating the cumulative total, etc.

Q52. What are the caveats of using Spark UDFs?  
A52. Firstly, and most importantly use UDFs only if you cannot find a suitable “**built-in**” function. This is because UDFs are a blackbox for Spark and so it does not even try to optimise them. The use of UDFs can lead to the loss of constant folding and of predicate pushdown optimizations. So, use UDFs only as a last resort, and when you use it:

Java

|  |  |
| --- | --- |
| 1  2  3  4 | //always check yourself using  dataframe.explain(true) |

Secondly, when using UDFs, user needs to handle all exceptional scenarios like handling null values. In the above UDF example, “country.toUpperCase()” can throw a “java.lang.**NullPointerException**” if the value passed for a country is null. So, proper testing is required for the UDFs.

Q53. What are the different types of window functions in Spark?  
A53.

**Ranking**: rank, dense\_rank, percent\_rank, row\_num, and ntile

**Aggregate**: min, max, avg, count, and sum.

**Analytical** : cume\_dist, lag, and lead

**Custom boundary** : rangeBetween and rowsBetween

Q54. Describe **ranking** window function with an example?  
A54. The example below shows how the ranking function can be used over a window of “**first\_name**” & “**country**“. As you can see it is very handy to rank and filter the latest salry by “**created\_dt**“.

Java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | %scala      import org.apache.spark.sql.functions.\_  import org.apache.spark.sql.expressions.Window    val format = new java.text.SimpleDateFormat("yyyy-MM-dd")    val someDF = Seq(("John", "USA", 35000.00, new java.sql.Date(format.parse("2017-06-02").getTime())),  ("Peter", "AUS", 45000.00, new java.sql.Date(format.parse("2017-07-02").getTime())),  ("Sam", "AUS", 75000.00, new java.sql.Date(format.parse("2017-02-02").getTime())),  ("Paul", "USA", 85000.0, new java.sql.Date(format.parse("2017-03-02").getTime())),  ("Sam", "AUS", 95000.00, new java.sql.Date(format.parse("2019-02-02").getTime())),  ("Paul", "USA", 105000.0, new java.sql.Date(format.parse("2020-03-02").getTime()))      ).toDF("first\_name", "country", "salary", "created\_dt")    someDF.show()    val winSpec = Window.partitionBy("first\_name", "country").orderBy($"created\_dt".desc)    val ranked\_salary\_df = someDF.withColumn("rank", rank().over(winSpec))    ranked\_salary\_df.show()    val current\_salry\_df = ranked\_salary\_df.filter($"rank" === 1)    current\_salry\_df.show() |

**Output:**

Java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32 | +----------+-------+--------+----------+  |first\_name|country|  salary|created\_dt|  +----------+-------+--------+----------+  |      John|    USA| 35000.0|2017-06-02|  |     Peter|    AUS| 45000.0|2017-07-02|  |       Sam|    AUS| 75000.0|2017-02-02|  |      Paul|    USA| 85000.0|2017-03-02|  |       Sam|    AUS| 95000.0|2019-02-02|  |      Paul|    USA|105000.0|2020-03-02|  +----------+-------+--------+----------+    +----------+-------+--------+----------+----+  |first\_name|country|  salary|created\_dt|rank|  +----------+-------+--------+----------+----+  |      John|    USA| 35000.0|2017-06-02|   1|  |      Paul|    USA|105000.0|2020-03-02|   1|  |      Paul|    USA| 85000.0|2017-03-02|   2|  |     Peter|    AUS| 45000.0|2017-07-02|   1|  |       Sam|    AUS| 95000.0|2019-02-02|   1|  |       Sam|    AUS| 75000.0|2017-02-02|   2|  +----------+-------+--------+----------+----+    +----------+-------+--------+----------+----+  |first\_name|country|  salary|created\_dt|rank|  +----------+-------+--------+----------+----+  |      John|    USA| 35000.0|2017-06-02|   1|  |      Paul|    USA|105000.0|2020-03-02|   1|  |     Peter|    AUS| 45000.0|2017-07-02|   1|  |       Sam|    AUS| 95000.0|2019-02-02|   1|  +----------+-------+--------+----------+----+ |

Q55. Describe **Aggregate** window function with an example?  
A55. You can calculate the average salary over a window as shown below.

Java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | %scala      import org.apache.spark.sql.functions.\_  import org.apache.spark.sql.expressions.Window    val format = new java.text.SimpleDateFormat("yyyy-MM-dd")    val someDF = Seq(("John", "USA", 35000.00, new java.sql.Date(format.parse("2017-06-02").getTime())),  ("Peter", "AUS", 45000.00, new java.sql.Date(format.parse("2017-07-02").getTime())),  ("Sam", "AUS", 75000.00, new java.sql.Date(format.parse("2017-02-02").getTime())),  ("Paul", "USA", 85000.0, new java.sql.Date(format.parse("2017-03-02").getTime())),  ("Sam", "AUS", 95000.00, new java.sql.Date(format.parse("2019-02-02").getTime())),  ("Paul", "USA", 105000.0, new java.sql.Date(format.parse("2020-03-02").getTime()))      ).toDF("first\_name", "country", "salary", "created\_dt")    val winSpec = Window.partitionBy("first\_name", "country")    val average\_salary\_df = someDF.withColumn("avg\_salary", avg("salary").over(winSpec))                                .dropDuplicates("first\_name", "country", "avg\_salary")    average\_salary\_df.show() |

**Output:**

Java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | +----------+-------+-------+----------+----------+  |first\_name|country| salary|created\_dt|avg\_salary|  +----------+-------+-------+----------+----------+  |      John|    USA|35000.0|2017-06-02|   35000.0|  |      Paul|    USA|85000.0|2017-03-02|   95000.0|  |     Peter|    AUS|45000.0|2017-07-02|   45000.0|  |       Sam|    AUS|75000.0|2017-02-02|   85000.0|  +----------+-------+-------+----------+----------+ |

Q56. Describe **Analytical** window function with an example?  
A56. “**lag**” analytical function looks at the “x” number of rows prior to the current row. If “x” is one, then it looks at its previous row.

Java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | %scala      import org.apache.spark.sql.functions.\_  import org.apache.spark.sql.expressions.Window    val format = new java.text.SimpleDateFormat("yyyy-MM-dd")    val someDF = Seq(("John", "USA", 35000.00, new java.sql.Date(format.parse("2017-06-02").getTime())),  ("Peter", "AUS", 45000.00, new java.sql.Date(format.parse("2017-07-02").getTime())),  ("Sam", "AUS", 75000.00, new java.sql.Date(format.parse("2017-02-02").getTime())),  ("Paul", "USA", 85000.0, new java.sql.Date(format.parse("2017-03-02").getTime())),  ("Sam", "AUS", 95000.00, new java.sql.Date(format.parse("2019-02-02").getTime())),  ("Paul", "USA", 105000.0, new java.sql.Date(format.parse("2020-03-02").getTime()))      ).toDF("first\_name", "country", "salary", "created\_dt")    val winSpec = Window.partitionBy("country").orderBy($"country".asc)    val lag\_salary\_df = someDF.withColumn("lag\_salary", lag("salary", 1).over(winSpec))    lag\_salary\_df.show() |

**Output:**

Java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | +----------+-------+--------+----------+----------+  |first\_name|country|  salary|created\_dt|lag\_salary|  +----------+-------+--------+----------+----------+  |     Peter|    AUS| 45000.0|2017-07-02|      null|  |       Sam|    AUS| 75000.0|2017-02-02|   45000.0|  |       Sam|    AUS| 95000.0|2019-02-02|   75000.0|  |      John|    USA| 35000.0|2017-06-02|      null|  |      Paul|    USA| 85000.0|2017-03-02|   35000.0|  |      Paul|    USA|105000.0|2020-03-02|   85000.0|  +----------+-------+--------+----------+----------+ |

Q57. Describe **Custom boundary** window function with an example?  
A57. Create a window where start of the window is “**one row prior to current**” and “**end is one row after current row**” with **rowsBetween**. You can find the average salary in that window.

Java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | %scala      import org.apache.spark.sql.functions.\_  import org.apache.spark.sql.expressions.Window    val format = new java.text.SimpleDateFormat("yyyy-MM-dd")    val someDF = Seq(("John", "USA", 35000.00, new java.sql.Date(format.parse("2017-06-02").getTime())),  ("Peter", "AUS", 45000.00, new java.sql.Date(format.parse("2017-07-02").getTime())),  ("Sam", "AUS", 75000.00, new java.sql.Date(format.parse("2017-02-02").getTime())),  ("Paul", "USA", 85000.0, new java.sql.Date(format.parse("2017-03-02").getTime())),  ("Sam", "AUS", 95000.00, new java.sql.Date(format.parse("2019-02-02").getTime())),  ("Paul", "USA", 105000.0, new java.sql.Date(format.parse("2020-03-02").getTime()))      ).toDF("first\_name", "country", "salary", "created\_dt")    val winSpec = Window.partitionBy("country").orderBy($"first\_name".asc).rowsBetween(-1, 1)    val average\_salary\_df = someDF.withColumn("avg\_salary", avg("salary").over(winSpec))    average\_salary\_df.show() |

**Output:**

Java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | +----------+-------+--------+----------+-----------------+  |first\_name|country|  salary|created\_dt|       avg\_salary|  +----------+-------+--------+----------+-----------------+  |     Peter|    AUS| 45000.0|2017-07-02|          60000.0|  |       Sam|    AUS| 75000.0|2017-02-02|71666.66666666667|  |       Sam|    AUS| 95000.0|2019-02-02|          85000.0|  |      John|    USA| 35000.0|2017-06-02|          60000.0|  |      Paul|    USA| 85000.0|2017-03-02|          75000.0|  |      Paul|    USA|105000.0|2020-03-02|          95000.0|  +----------+-------+--------+----------+-----------------+ |

What if you want to get a cumulative (i.e running) total?

Java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | //.....  val winSpec = Window.partitionBy("country").orderBy($"salary".asc).rowsBetween(Window.unboundedPreceding, Window.currentRow)    val cumulative\_salary\_df = someDF.withColumn("cumulative\_salary\_in\_range", sum("salary").over(winSpec))    cumulative\_salary\_df.show() |

**Output:**

Java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | +----------+-------+--------+----------+--------------------------+  |first\_name|country|  salary|created\_dt|cumulative\_salary\_in\_range|  +----------+-------+--------+----------+--------------------------+  |     Peter|    AUS| 45000.0|2017-07-02|                   45000.0|  |       Sam|    AUS| 75000.0|2017-02-02|                  120000.0|  |       Sam|    AUS| 95000.0|2019-02-02|                  215000.0|  |      John|    USA| 35000.0|2017-06-02|                   35000.0|  |      Paul|    USA| 85000.0|2017-03-02|                  120000.0|  |      Paul|    USA|105000.0|2020-03-02|                  225000.0|  +----------+-------+--------+----------+--------------------------+ |

Another example using **rangeBetween**:

Java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | %scala      import org.apache.spark.sql.functions.\_  import org.apache.spark.sql.expressions.Window    val format = new java.text.SimpleDateFormat("yyyy-MM-dd")    val someDF = Seq(("John", "USA", 35000.00, new java.sql.Date(format.parse("2017-06-02").getTime())),  ("Peter", "AUS", 45000.00, new java.sql.Date(format.parse("2017-07-02").getTime())),  ("Sam", "AUS", 75000.00, new java.sql.Date(format.parse("2017-02-02").getTime())),  ("Paul", "USA", 85000.0, new java.sql.Date(format.parse("2017-03-02").getTime())),  ("Sam", "AUS", 95000.00, new java.sql.Date(format.parse("2019-02-02").getTime())),  ("Paul", "USA", 105000.0, new java.sql.Date(format.parse("2020-03-02").getTime()))      ).toDF("first\_name", "country", "salary", "created\_dt")    val winSpec = Window.partitionBy("country").orderBy($"salary".asc).rangeBetween(Window.unboundedPreceding, 50000L)    val max\_salary\_df = someDF.withColumn("max\_salary\_in\_range", max("salary").over(winSpec))    max\_salary\_df.show() |

**Output:**

Java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | +----------+-------+--------+----------+-------------------+  |first\_name|country|  salary|created\_dt|max\_salary\_in\_range|  +----------+-------+--------+----------+-------------------+  |     Peter|    AUS| 45000.0|2017-07-02|            95000.0|  |       Sam|    AUS| 75000.0|2017-02-02|            95000.0|  |       Sam|    AUS| 95000.0|2019-02-02|            95000.0|  |      John|    USA| 35000.0|2017-06-02|            85000.0|  |      Paul|    USA| 85000.0|2017-03-02|           105000.0|  |      Paul|    USA|105000.0|2020-03-02|           105000.0|  +----------+-------+--------+----------+-------------------+ |

**Note**:

For **USA**, Row 1: **range start**= Window.unboundedPreceding = 35000, and range end = 35000 (i.e current) + 50000 = 85000, and 85000 is the max salary in that range.

For **USA**, Row 2: **range start**= Window.unboundedPreceding = 35000, and range end = 85000 (i.e current) + 50000 = 135000, and 105000 is the max salary.

For **USA**, Row 3: **range start**= Window.unboundedPreceding = 35000, and range end = 105000 (i.e current) + 50000 = 155000, and 105000 is the max salary in that range.

‹