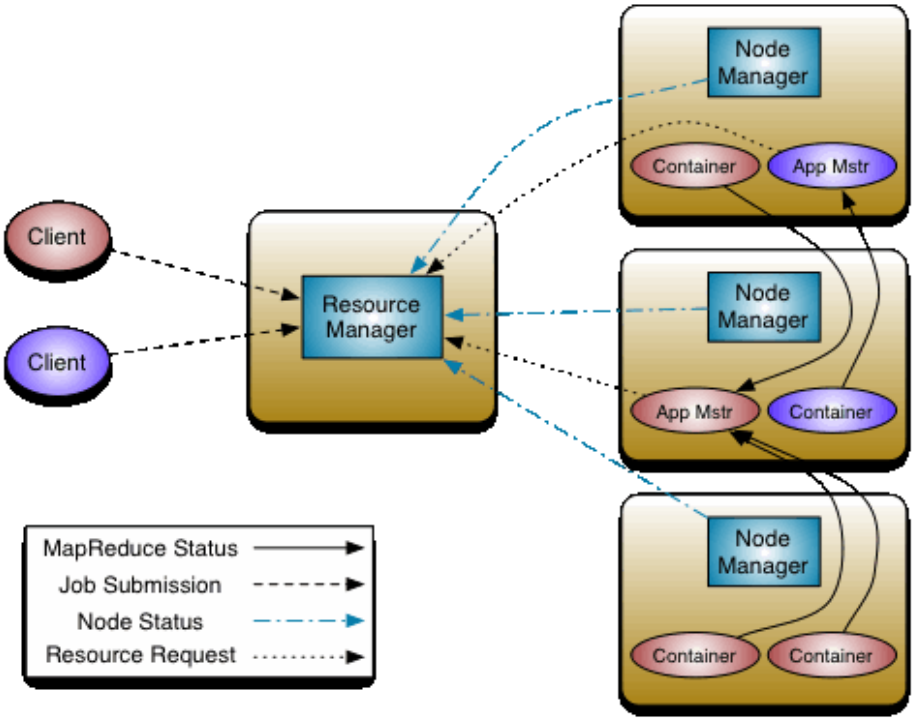
Q1 - Uber mode in Hadoop YARN

Hadoop YARN consists of 3 main components - A global Resource Manager (RM) on master node, Node Manager (NM) on individual worker nodes and a per-application Application Master (AM).

When submitting any job (MR / Spark / PIG etc..) on Hadoop Cluster, Resource Manager (RM), which is the ultimate authority for scheduling resources for job execution, negotiates with Node Managers (NM) on worker nodes, to launch first container with an Application master (AM) process.

The per-application ApplicationMaster takes over and has the responsibility of negotiating appropriate resource containers from the RM.



Based on job requirements (# of map and reduce tasks to be executed), ApplicationMaster negotiates with the ResourceManager to get that many resource containers for running the tasks.

However, If a job is small ApplicationMaster may decide to run the job sequentially in the similar JVM where ApplicationMaster itself is running. This way of running a job is known as **“uber mode” in yarn.**

Application Master makes the decision of whether a job is sufficiently small for execution (“uber mode”), based on below configuration parameters. (These parameters are in **mapred-site.xml**.

- **mapreduce.job.ubertask.enable** - Setting this parameter as true enables the small-jobs "ubertask" optimization, which runs "sufficiently small" jobs sequentially within a single JVM. Default is false.

```
▼<property>
   <name>mapreduce.job.ubertask.enable</name>
   <value>false</value>
   <final>false</final>
   <source>mapred-default.xml</source>
 </property>
```

- **mapreduce.job.ubertask.maxmaps**- Threshold for number of maps, beyond which job is considered too big for the ubertasking optimization. Default value is 9. Users may override this value, but only downward.

```
▼<property>
   <name>mapreduce.job.ubertask.maxmaps</name>
   <value>9</value>
   <final>false</final>
   <source>mapred-default.xml</source>
 </property>
```

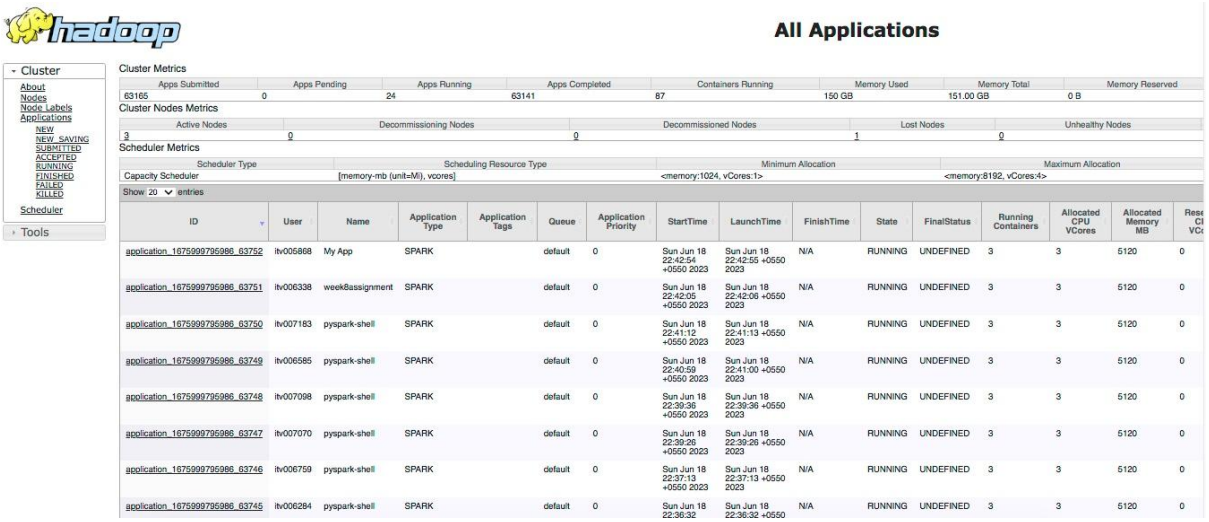
- **mapreduce.job.ubertask.maxreduces**- Threshold for number of reduces, beyond which job is considered too big for the ubertasking optimization.

```
▼<property>
   <name>mapreduce.job.ubertask.maxreduces</name>
   <value>1</value>
   <final>false</final>
   <source>mapred-default.xml</source>
 </property>
```

- **mapreduce.job.ubertask.maxbytes**- Threshold for number of input bytes, beyond which job is considered too big for the uber tasking optimization. If no value is specified, dfs.block.size is used as a default which means HDFS block size in case of HDFS.

Q2 - Resource Manager in a Multi-Node Cluster

Cluster Metrics

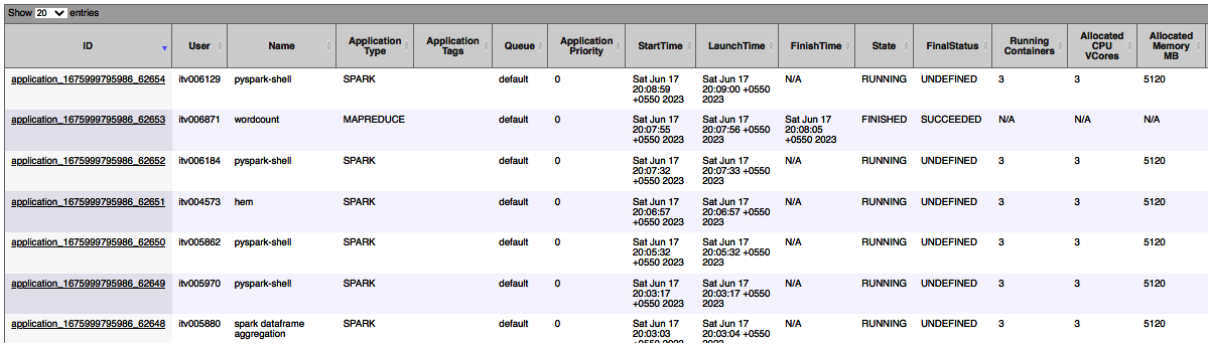


YARN Resource Manager (RM) orchestrates the scheduling of resources between different applications running on the cluster. RM dashboard provides cluster metrics of overall status of cluster resources.

Active Nodes - Total Worker nodes available in the cluster
Containers Running - Count of containers (executors) currently running in the cluster.
Memory Total - Total Memory available in the cluster for jobs
Memory Used - Memory currently in use.
VCores Total - Total virtual CPU cores available in the cluster
VCores Used - Total Virtual CPU cores currently in use.

Additionally, We can also view the list of different applications and their current executionstatus.

In the figure below, We can see SPARK application in running status and MAPREDUCE application in finished state.

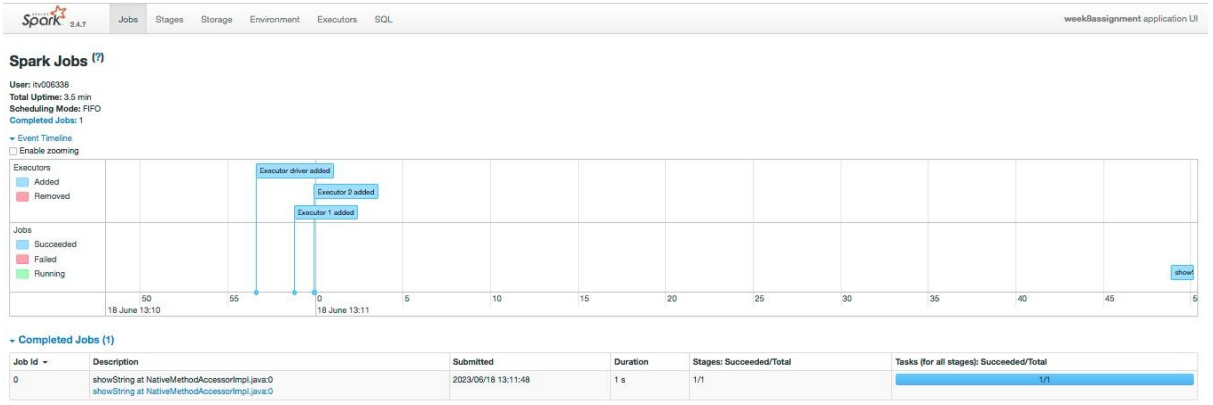


Show 20 entries

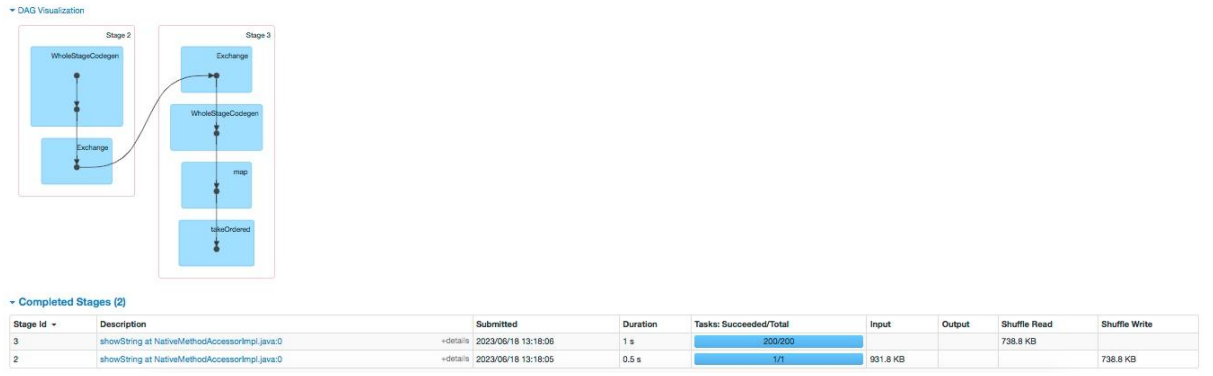
ID	User	Name	Application Type	Application Tags	Queue	Application Priority	StartTime	LaunchTime	FinishTime	State	FinalStatus	Running Containers	Allocated CPU VCores	Allocated Memory MB
application_1675999795986_62654	itv006129	pyspark-shell	SPARK		default	0	Sat Jun 17 20:08:59 +0550 2023	Sat Jun 17 20:09:00 +0550 2023	N/A	RUNNING	UNDEFINED	3	3	5120
application_1675999795986_62653	itv006871	wordcount	MAPREDUCE		default	0	Sat Jun 17 20:07:55 +0550 2023	Sat Jun 17 20:07:56 +0550 2023	Sat Jun 17 20:08:05 +0550 2023	FINISHED	SUCCEEDED	N/A	N/A	N/A
application_1675999795986_62652	itv006184	pyspark-shell	SPARK		default	0	Sat Jun 17 20:07:32 +0550 2023	Sat Jun 17 20:07:33 +0550 2023	N/A	RUNNING	UNDEFINED	3	3	5120
application_1675999795986_62651	itv004573	hem	SPARK		default	0	Sat Jun 17 20:06:57 +0550 2023	Sat Jun 17 20:06:57 +0550 2023	N/A	RUNNING	UNDEFINED	3	3	5120
application_1675999795986_62650	itv005862	pyspark-shell	SPARK		default	0	Sat Jun 17 20:05:32 +0550 2023	Sat Jun 17 20:05:32 +0550 2023	N/A	RUNNING	UNDEFINED	3	3	5120
application_1675999795986_62649	itv005970	pyspark-shell	SPARK		default	0	Sat Jun 17 20:03:17 +0550 2023	Sat Jun 17 20:03:17 +0550 2023	N/A	RUNNING	UNDEFINED	3	3	5120
application_1675999795986_62648	itv005880	spark dataframe aggregation	SPARK		default	0	Sat Jun 17 20:03:03	Sat Jun 17 20:03:04 +0550	N/A	RUNNING	UNDEFINED	3	3	5120

Job, Stages & Task

Click on Job Tab to view Jobs (Every action creates a Job in Spark application)



For every action, You can click to view Stages. Note - Every Wide transformation like groupBy, creates 200 partitions by default.



Click on SQL Tab to view SQL queries being executed as part of Spark application.

Q3 - Window Functions

We will be working on a grocery dataset to demonstrate different window functions.

- Load the dataset

```
groceries_schema = "order_id string, location string, item string, order_date string, quantity integer"

grocery_df = spark.read.format("csv").schema(groceries_schema).option("header","true").option("dateFormat","dd/MM/YYYY").load(f"/user/{username}/groceries.csv")
```

```
+--------+---------+---------+----------+--------+
|order_id| location|     item|order_date|quantity|
+--------+---------+---------+----------+--------+
|      o1|  Seattle| Bananas|01/01/2017|       7|
|      o2|     Kent|  Apples|02/01/2017|      20|
|      o3| Bellevue| Flowers|02/01/2017|      10|
|      o4|  Redmond|    Meat|03/01/2017|      40|
|      o5|  Seattle|Potatoes|04/01/2017|       9|
|      o6| Bellevue|   Bread|04/01/2017|       5|
|      o7|  Redmond|   Bread|05/01/2017|       5|
|      o8| Issaquah|   Onion|05/01/2017|       4|
|      o9|  Redmond|  Cheese|05/01/2017|      15|
|     o10| Issaquah|   Onion|06/01/2017|       4|
|     o11|   Renton|   Bread|05/01/2017|       5|
|     o12| Issaquah|   Onion|07/01/2017|       4|
|     o13|Sammamish|   Bread|07/01/2017|       5|
|     o14| Issaquah|  Tomato|07/01/2017|       6|
|     o15| Issaquah|    Meat|08/01/2017|       3|
|     o16| Issaquah|    Meat|09/01/2017|       5|
|     o17| Issaquah|    Meat|10/01/2017|       6|
|     o18| Bellevue|   Bread|11/01/2017|       7|
|     o19| Bellevue|   Bread|12/01/2017|      54|
|     o20| Bellevue|   Bread|13/01/2017|      34|
+--------+---------+---------+----------+--------+
only showing top 20 rows
```

- **Change the date column from string type into date type**

```
grocery_df_new = grocery_df.withColumn("order_date",to_date(col("order_date"),
'd/MM/yyyy'))

grocery_df_new.printSchema()
```

```
root
 |-- order_id: string (nullable = true)
 |-- location: string (nullable = true)
 |-- item: string (nullable = true)
 |-- order_date: date (nullable = true)
 |-- quantity: integer (nullable = true)
```

- RUNNING TOTAL

We will partition by item and order by order_date

```
mywindow = Window.partitionBy("item") \
.orderBy("order_date") \
.rowsBetween(Window.unboundedPreceding, Window.currentRow)

result_df = grocery_df_new.withColumn("running_total",sum("quantity").over(mywindow))
result_df.show()
```

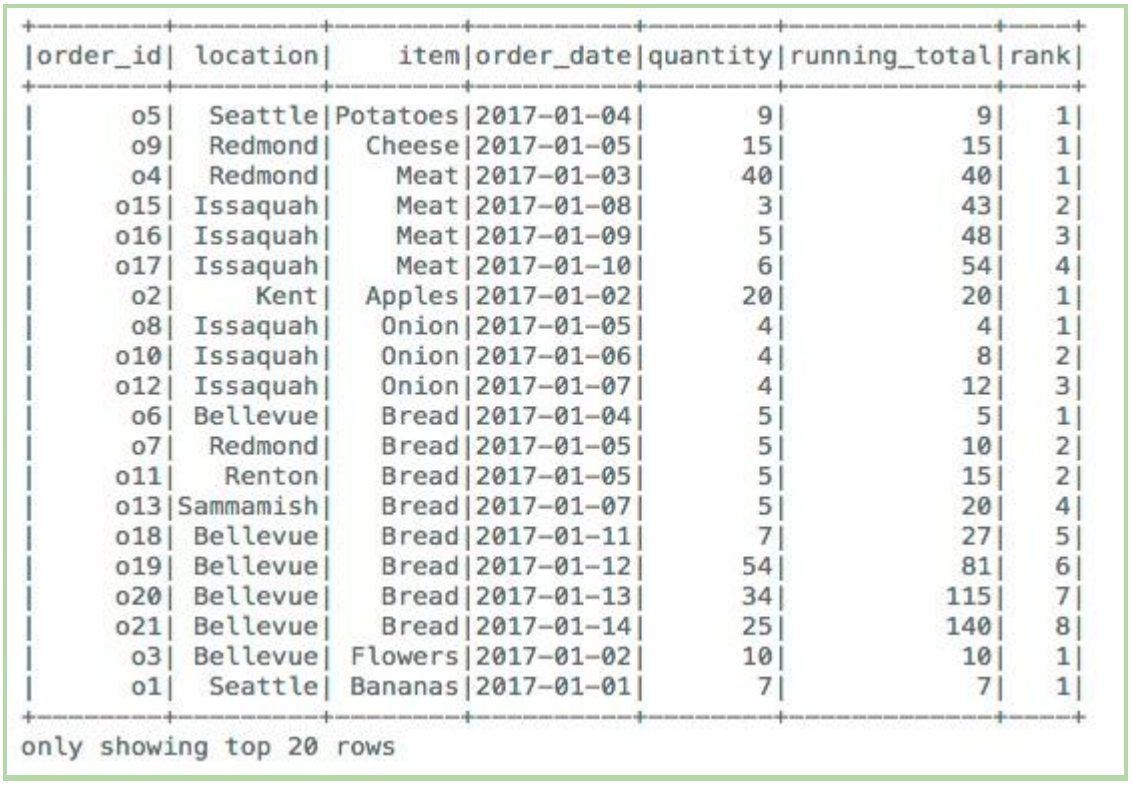
order_id	location	item	order_date	quantity	running_total
o5	Seattle	Potatoes	2017-01-04	9	9
o9	Redmond	Cheese	2017-01-05	15	15
o4	Redmond	Meat	2017-01-03	40	40
o15	Issaquah	Meat	2017-01-08	3	43
o16	Issaquah	Meat	2017-01-09	5	48
o17	Issaquah	Meat	2017-01-10	6	54
o2	Kent	Apples	2017-01-02	20	20
o8	Issaquah	Onion	2017-01-05	4	4
o10	Issaquah	Onion	2017-01-06	4	8
o12	Issaquah	Onion	2017-01-07	4	12
o6	Bellevue	Bread	2017-01-04	5	5
o7	Redmond	Bread	2017-01-05	5	10
o11	Renton	Bread	2017-01-05	5	15
o13	Sammamish	Bread	2017-01-07	5	20
o18	Bellevue	Bread	2017-01-11	7	27
o19	Bellevue	Bread	2017-01-12	54	81
o20	Bellevue	Bread	2017-01-13	34	115
o21	Bellevue	Bread	2017-01-14	25	140
o3	Bellevue	Flowers	2017-01-02	10	10
o1	Seattle	Bananas	2017-01-01	7	7

only showing top 20 rows

- ## RANK function

```
result_df = result_df.withColumn("rank",rank().over(mywindow))
result_df.show()
```

Notice the results for item “Bread”. Both entries on ‘2017-01-05’ share rank 2, while the next entry is marked as rank 4th .



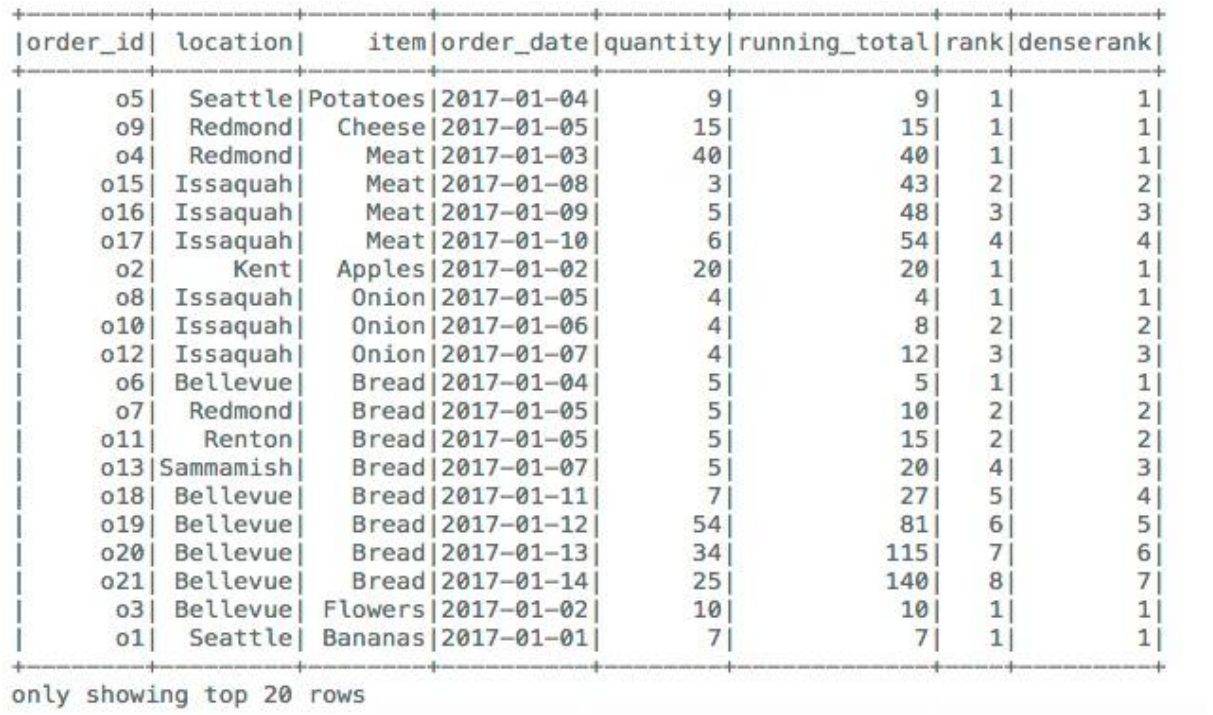
order_id	location	item	order_date	quantity	running_total	rank
o5	Seattle	Potatoes	2017-01-04	9	9	1
o9	Redmond	Cheese	2017-01-05	15	15	1
o4	Redmond	Meat	2017-01-03	40	40	1
o15	Issaquah	Meat	2017-01-08	3	43	2
o16	Issaquah	Meat	2017-01-09	5	48	3
o17	Issaquah	Meat	2017-01-10	6	54	4
o2	Kent	Apples	2017-01-02	20	20	1
o8	Issaquah	Onion	2017-01-05	4	4	1
o10	Issaquah	Onion	2017-01-06	4	8	2
o12	Issaquah	Onion	2017-01-07	4	12	3
o6	Bellevue	Bread	2017-01-04	5	5	1
o7	Redmond	Bread	2017-01-05	5	10	2
o11	Renton	Bread	2017-01-05	5	15	2
o13	Sammamish	Bread	2017-01-07	5	20	4
o18	Bellevue	Bread	2017-01-11	7	27	5
o19	Bellevue	Bread	2017-01-12	54	81	6
o20	Bellevue	Bread	2017-01-13	34	115	7
o21	Bellevue	Bread	2017-01-14	25	140	8
o3	Bellevue	Flowers	2017-01-02	10	10	1
o1	Seattle	Bananas	2017-01-01	7	7	1

only showing top 20 rows

- DENSE RANK function

```
result_df = result_df.withColumn("denserank",dense_rank().over(mywindow))
result_df.show()
```

Again, notice the results for item “Bread”. Both entries on ‘2017-01-05’ share rank 2, however the next entry on ‘2017-01-07’ gets the next rank of 3.



order_id	location	item	order_date	quantity	running_total	rank	denserank
o5	Seattle	Potatoes	2017-01-04	9	9	1	1
o9	Redmond	Cheese	2017-01-05	15	15	1	1
o4	Redmond	Meat	2017-01-03	40	40	1	1
o15	Issaquah	Meat	2017-01-08	3	43	2	2
o16	Issaquah	Meat	2017-01-09	5	48	3	3
o17	Issaquah	Meat	2017-01-10	6	54	4	4
o2	Kent	Apples	2017-01-02	20	20	1	1
o8	Issaquah	Onion	2017-01-05	4	4	1	1
o10	Issaquah	Onion	2017-01-06	4	8	2	2
o12	Issaquah	Onion	2017-01-07	4	12	3	3
o6	Bellevue	Bread	2017-01-04	5	5	1	1
o7	Redmond	Bread	2017-01-05	5	10	2	2
o11	Renton	Bread	2017-01-05	5	15	2	2
o13	Sammamish	Bread	2017-01-07	5	20	4	3
o18	Bellevue	Bread	2017-01-11	7	27	5	4
o19	Bellevue	Bread	2017-01-12	54	81	6	5
o20	Bellevue	Bread	2017-01-13	34	115	7	6
o21	Bellevue	Bread	2017-01-14	25	140	8	7
o3	Bellevue	Flowers	2017-01-02	10	10	1	1
o1	Seattle	Bananas	2017-01-01	7	7	1	1

only showing top 20 rows

- Lag & Lead function

```
lagleadwindow = Window.partitionBy("item").orderBy("order_date")

result_df = result_df.withColumn("previous-lag",lag("quantity").over(lagleadwindow))

result_df = result_df.withColumn("next-lead",lead("quantity").over(lagleadwindow))

result_df.show()
```

Lag - Within the window, print the column value of previous row, as current row.
Lead - Within the window, print the column value of next row, as current row.

Lead and Lag can be used to calculate the difference between column value of current row and previous/next row.

order_id	location	item	order_date	quantity	running_total	rank	denserank	previous-lag	next-lead
o5	Seattle	Potatoes	2017-01-04	9	9	1	1	null	null
o9	Redmond	Cheese	2017-01-05	15	15	1	1	null	null
o4	Redmond	Meat	2017-01-03	40	40	1	1	null	3
o15	Issaquah	Meat	2017-01-08	3	43	2	2	40	5
o16	Issaquah	Meat	2017-01-09	5	48	3	3	3	6
o17	Issaquah	Meat	2017-01-10	6	54	4	4	5	null
o2	Kent	Apples	2017-01-02	20	20	1	1	null	null
o8	Issaquah	Onion	2017-01-05	4	4	1	1	null	4
o10	Issaquah	Onion	2017-01-06	4	8	2	2	4	4
o12	Issaquah	Onion	2017-01-07	4	12	3	3	4	null
o6	Bellevue	Bread	2017-01-04	5	5	1	1	null	5
o7	Redmond	Bread	2017-01-05	5	10	2	2	5	5
o11	Renton	Bread	2017-01-05	5	15	2	2	5	5
o13	Sammamish	Bread	2017-01-07	5	20	4	3	5	7
o18	Bellevue	Bread	2017-01-11	7	27	5	4	5	54
o19	Bellevue	Bread	2017-01-12	54	81	6	5	7	34
o20	Bellevue	Bread	2017-01-13	34	115	7	6	54	25
o21	Bellevue	Bread	2017-01-14	25	140	8	7	34	null
o3	Bellevue	Flowers	2017-01-02	10	10	1	1	null	null
o1	Seattle	Bananas	2017-01-01	7	7	1	1	null	null

only showing top 20 rows

Q4 - Dealing with Nulls in Apache Spark

When the value of a column specific to a row is not known, at the time the row came into existence, it is represented as NULL in SQL.

Create a DataFrame with NULL Value.

```
df = spark.createDataFrame([("shivam",40),("Rajesh",30),("Mayank",None)],["name","age"])
df.show()
```

name	age
shivam	40
Rajesh	30
Mayank	null

‘None’ keyword is used in Python to denote missing or unknown value

Create a DataFrame with Nullable Property.

Each column in a DataFrame has a nullable property that can be set to True or False.

If nullable is set to False then the column cannot contain null values.

```
schema = StructType([
  StructField("name", StringType(), True),
  StructField("age", IntegerType(), False)])

df = spark.createDataFrame([("shivam",40),("Rajesh",30),("Mayank",None)],schema)
df.show()
```

It throws a ValueError: field age: This field is not nullable, but got None

Comparison Operators with Null Value

The result of comparison operators is unknown or NULL when one of the operarands or both the operands are unknown or NULL. In order to compare the NULL values for equality,

Spark provides a null-safe equal operator (‘<=>’), which returns False when one of the operand is NULL and returns ‘True when both the operands are NULL.

Drop Rows having Null Values

```
#na func to drop rows with null values
#rows having atleast a null value is dropped

df.na.drop().show()
```