Chapter: Two
Working with input files,

Read CSV

Reading csv with header and infer the schema.

```
# python code
# read the csv file with header and inferSchema
sales_data = spark.read.csv('/mnt/e/Training_Data/5m-Sales-Records/
*.csv',header=True, inferSchema=True)
# print the schema
sales_data.printSchema()

root
 |-- Region: string (nullable = true)
 |-- Country: string (nullable = true)
 |-- Item Type: string (nullable = true)
 |-- Sales Channel: string (nullable = true)
 |-- Order Priority: string (nullable = true)
 |-- Order Date: string (nullable = true)
 |-- Order ID: integer (nullable = true)
 |-- Ship Date: string (nullable = true)
 |-- Units Sold: integer (nullable = true)
 |-- Unit Price: double (nullable = true)
 |-- Unit Cost: double (nullable = true)
 |-- Total Revenue: double (nullable = true)
 |-- Total Cost: double (nullable = true)
 |-- Total Profit: double (nullable = true)
 |-- id: integer (nullable = true)
```

Read Json

Reading JSON file and infer the schema. If the json is split across multi line, then we can use to keep the multi line option as True.

```
    # Python code
```

```
# read the json
sales_data_json = spark.read.option("multiline", "false").option("inf
erSchema","true").json('/mnt/e/Training_Data/5m-Sales-Records/
*.json')

sales_data_json.printSchema()

root
 |-- Country: string (nullable = true)
 |-- Item Type: string (nullable = true)
 |-- Order Date: string (nullable = true)
 |-- Order ID: long (nullable = true)
 |-- Order Priority: string (nullable = true)
 |-- Region: string (nullable = true)
 |-- Sales Channel: string (nullable = true)
 |-- Ship Date: string (nullable = true)
 |-- Total Cost: double (nullable = true)
 |-- Total Profit: double (nullable = true)
 |-- Total Revenue: double (nullable = true)
 |-- Unit Cost: double (nullable = true)
 |-- Unit Price: double (nullable = true)
 |-- Units Sold: long (nullable = true)
 |-- id: long (nullable = true)
```

Spark can read data from verities of files. However it's advisable to have the input files compress, preferably splittable compression, like BZ2, in this example.

Full list of splittable and non-splittable format

doop Codecs

odec	File Extension	Splittable?	Degree of Compression	Compression Spee
zip	.gz	No	Medium	Medium
zip2	.bz2	Yes	High	Slow
nappy	.snappy	No	Medium	Fast
ZO	.lzo	No, unless indexed	Medium	Fast

Codecs properties

Size of the files have impact on the reading data performance.

Here is an example. I have few uncompressed json files of 12.4 GB. Here is what happens when I read the files.

```
>>> corona_data = spark.read.json('/mnt/e/Training_Data/Corona_twiter_data/corona/*.json')
[Stage 9:>                                                          (0 + 4) / 100]
```

Stage Id	Description		Submitted	Duration	Tasks: Succeeded/Total	Input
9	json at NativeMethodAccessorImpl.java:0	+details	2021/04/11 14:48:06	10 min	100/100	12.4 GiB

Index	Task ID	Attempt	Status	Locality level	Executor ID	Host	Logs	Launch Time	Duration	GC Time	Input Size / Records	Errors
0	23	0	SUCCESS	PROCESS_LOCAL	0	172.31.16.1	stdout stderr	2021-04-11 14:48:06	20 s	50.0 ms	128.1 MiB / 39693	
1	24	0	SUCCESS	PROCESS_LOCAL	1	172.31.16.1	stdout stderr	2021-04-11 14:48:06	18 s	0.1 s	128.1 MiB / 39642	
2	25	0	SUCCESS	PROCESS_LOCAL	0	172.31.16.1	stdout stderr	2021-04-11 14:48:06	17 s	44.0 ms	128.1 MiB / 39566	
3	26	0	SUCCESS	PROCESS_LOCAL	1	172.31.16.1	stdout stderr	2021-04-11 14:48:06	16 s	0.1 s	128.1 MiB / 40082	
4	27	0	SUCCESS	PROCESS_LOCAL	1	172.31.16.1	stdout stderr	2021-04-11 14:48:23	24 s	0.1 s	128.1 MiB / 39720	
5	28	0	SUCCESS	PROCESS_LOCAL	0	172.31.16.1	stdout stderr	2021-04-11 14:48:23	24 s	63.0 ms	128.1 MiB / 39392	
6	29	0	SUCCESS	PROCESS_LOCAL	1	172.31.16.1	stdout stderr	2021-04-11 14:48:24	14 s	84.0 ms	128.1 MiB / 39726	
7	30	0	SUCCESS	PROCESS_LOCAL	0	172.31.16.1	stdout stderr	2021-04-11 14:48:26	13 s	41.0 ms	128.1 MiB / 39872	

Reading uncompressed json file took long time to finish

It took 10 min and 100 tasks to just to read those files.

Why 100 splits? To understand this, we should know how spark split the input data. When data is read from the file system, it is divided into input blocks, which are then sent to different executors. By default, it is 128 MB (128000000 bytes). So, to divide 12.4 GB by 128 MB is around 100. This is how spark split the files and run task in parallel. We can see input size as 128 MB.

I am using 2 nodes cluster, with 2 CPU cores in each node. Which means spark can have at max, 4 task running in parallel.

When using the *spark-xml* package, you can increase the number of tasks per stage by changing the configuration setting *spark.hadoop.mapred.max.split.size* to a lower value in the cluster's Spark configuration. This configuration setting controls the input block size.

Most of the times we use shared cluster and will not have permissions to change the cluster configurations. However, we can change the properties of the job we are running. I will put an example in the bellow section.

Lets compress this json files using bz2. Input files are compressed as bz2, with a total size of 1.8 GB.

The same spark read data statement takes 9.6 min and 15 tasks only, instead of 100 tasks. This is good if you use shared cluster, fewer task means, need few resources and spark will complete the task quickly .

```
>>> corona_data_bz = spark.read.json('/mnt/e/Training_Data/Corona_twiter_data/corona/*.bz2
[Stage 8:>                                                          (0 + 4) / 15]
```

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total
8	json at NativeMethodAccessorImpl.java:0 +details	2021/04/11 12:48:18	9.6 min	15/15

Index	Task ID	Attempt	Status	Locality level	Executor ID	Host	Logs	Launch Time	Duration
0	8	0	SUCCESS	PROCESS_LOCAL	1	172.31.16.1	stdout stderr	2021-04-11 12:48:18	2.4 min
1	9	0	SUCCESS	PROCESS_LOCAL	0	172.31.16.1	stdout stderr	2021-04-11 12:48:18	2.5 min
2	10	0	SUCCESS	PROCESS_LOCAL	1	172.31.16.1	stdout stderr	2021-04-11 12:48:18	2.4 min
3	11	0	SUCCESS	PROCESS_LOCAL	0	172.31.16.1	stdout stderr	2021-04-11 12:48:18	2.5 min
4	12	0	SUCCESS	PROCESS_LOCAL	1	172.31.16.1	stdout stderr	2021-04-11 12:50:42	2.5 min
5	13	0	SUCCESS	PROCESS_LOCAL	1	172.31.16.1	stdout stderr	2021-04-11 12:50:44	2.5 min
6	14	0	SUCCESS	PROCESS_LOCAL	0	172.31.16.1	stdout stderr	2021-04-11 12:50:49	2.6 min
7	15	0	SUCCESS	PROCESS_LOCAL	0	172.31.16.1	stdout stderr	2021-04-11 12:50:50	2.6 min
8	16	0	SUCCESS	PROCESS_LOCAL	1	172.31.16.1	stdout stderr	2021-04-11 12:53:09	2.6 min
9	17	0	SUCCESS	PROCESS_LOCAL	1	172.31.16.1	stdout stderr	2021-04-11 12:53:14	2.5 min
10	18	0	SUCCESS	PROCESS_LOCAL	0	172.31.16.1	stdout stderr	2021-04-11 12:53:24	2.6 min
11	19	0	SUCCESS	PROCESS_LOCAL	0	172.31.16.1	stdout stderr	2021-04-11 12:53:27	2.6 min
12	20	0	SUCCESS	PROCESS_LOCAL	1	172.31.16.1	stdout stderr	2021-04-11 12:55:42	2.2 min
13	21	0	SUCCESS	PROCESS_LOCAL	1	172.31.16.1	stdout stderr	2021-04-11 12:55:45	2.2 min
14	22	0	SUCCESS	PROCESS_LOCAL	0	172.31.16.1	stdout stderr	2021-04-11 12:56:03	37 s

Reading compressed json file took less time to complete with fewer tasks

We can further improve the performance by tweaking the input size using *spark.sql.files.maxPartitionBytes*.

Let’s try changing the input size. We will use bellow command to set the input split size as 32 MB.

```
spark.conf.set("spark.sql.files.maxPartitionBytes", 32000000)
```

This property will set the input file split to 32 MB, which means we will have more task to finish with fewer data(32MB) per task.
This property has no effect if we have a lot of small files, which are less than the max partition bytes.

<u>Scenario 1</u>: - Here is the capture of running 12.4 GB uncompressed data using 32MB as split size. We end up having 417 task and this time total run time is 7.7 min.

```
spark.conf.set("spark.sql.files.maxPartitionBytes", 32000000)
corona_data = spark.read.json('/mnt/e/Training_Data/
Corona_twiter_data/corona/*.json')
corona_data.count()
```

Stage Id	Description		Submitted	Duration	Tasks: Succeeded/Total	Input
24	json at NativeMethodAccessorImpl.java:0	+details	2021/04/11 23:33:57	7.7 min	417/417	12.4 GiB

Summary Metrics for 417 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	0.3 s	4 s	4 s	5 s	20 s
GC Time	2.0 ms	8.0 ms	17.0 ms	24.0 ms	85.0 ms
Input Size / Records	14.9 MiB / 4915	30.6 MiB / 9436	30.6 MiB / 9534	30.6 MiB / 9614	30.6 MiB / 9924

Tasks (417)

Show 20 entries Search:

Index	Task ID	Attempt	Status	Locality level	Executor ID	Host	Logs	Launch Time	Duration	GC Time	Input Size / Records	Errors
0	252	0	SUCCESS	PROCESS_LOCAL	0	172.31.16.1	stdout stderr	2021-04-11 23:33:57	0.4 s	4.0 ms	30.6 MiB / 9465	
1	253	0	SUCCESS	PROCESS_LOCAL	1	172.31.16.1	stdout stderr	2021-04-11 23:33:57	0.4 s	6.0 ms	30.6 MiB / 9520	
2	254	0	SUCCESS	PROCESS_LOCAL	0	172.31.16.1	stdout stderr	2021-04-11 23:33:57	0.4 s	4.0 ms	30.6 MiB / 9460	
3	255	0	SUCCESS	PROCESS_LOCAL	1	172.31.16.1	stdout stderr	2021-04-11 23:33:57	0.4 s	6.0 ms	30.6 MiB / 9390	
4	256	0	SUCCESS	PROCESS_LOCAL	1	172.31.16.1	stdout stderr	2021-04-11 23:33:58	0.3 s	2.0 ms	30.6 MiB / 9444	
5	257	0	SUCCESS	PROCESS_LOCAL	1	172.31.16.1	stdout stderr	2021-04-11 23:33:58	0.4 s	2.0 ms	30.6 MiB / 9398	

Set the input split size as 32 MB

<u>Scenario 2</u>: - Here is the capture of running 12.4 GB uncompressed data using

256MB as split size. We end up having 55 task and this time total run time is 13 min, double of previous time. Having a large chunk of data means that the executor needs to spend more time to process the data. Some times this can become a bottleneck with skewed data. I will show an example in Skewed data section.

Stage Id ▾	Description	Submitted	Duration	Tasks: Succeeded/Total	Input
25	json at NativeMethodAccessorImpl.java:0 +details	2021/04/11 23:48:03	13 min	53/53	12.4 GiB

Summary Metrics for 53 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	11 s	41 s	50 s	1.1 min	2.6 min
GC Time	31.0 ms	0.1 s	0.2 s	0.2 s	0.4 s
Input Size / Records	52.1 MiB / 16527	244.3 MiB / 75576	244.3 MiB / 75998	244.3 MiB / 76858	244.3 MiB / 78364

Set the input split size as 256 MB

<u>Scenario 3</u>: - Here is the capture of running 12.4 GB compressed data using 32MB as split size. We end up having 417 task and this time total run time is 7.7 minutes, same as scenario 1 and half of scenario 2's time.

Stage Id ▾	Description	Submitted	Duration	Tasks: Succeeded/Total	Input
24	json at NativeMethodAccessorImpl.java:0 +details	2021/04/11 23:33:57	7.7 min	417/417	12.4 GiB

Summary Metrics for 61 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	2 s	28 s	30 s	31 s	33 s
GC Time	8.0 ms	0.2 s	0.2 s	0.2 s	0.2 s
Input Size / Records	1.5 MiB / 4973	30.7 MiB / 92475	30.7 MiB / 93776	30.7 MiB / 97108	30.7 MiB / 100518

Set the input split size as 32 MB on compressed data

<u>Scenario 4</u>: - Here is the capture of running 1.8 GB compressed data using 256MB as split size. We end up having 8 task and this time total run time is 8.1 min better than scenario 2 but worse than scenario 3.

Stage Id ▾	Description	Submitted	Duration	Tasks: Succeeded/Total	Input
28	json at NativeMethodAccessorImpl.java:0 +details	2021/04/12 00:45:39	8.1 min	8/8	1828.5 MiB

Summary Metrics for 8 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	2.9 min	4.0 min	4.1 min	4.2 min	4.2 min
GC Time	1 s	1 s	2 s	2 s	2 s
Input Size / Records	165.4 MiB / 513900	244.3 MiB / 739712	244.3 MiB / 746033	244.3 MiB / 781497	244.3 MiB / 790344

Set the input split size as 256 MB on compressed data

<u>Scenario 5</u>: - Here is the capture of running 1.8 GB compressed data using 100MB as split size. We end up having 20 task and this time total run time is 6.8 min.

This is the best so far.

Stage Id ▾	Description	Submitted	Duration	Tasks: Succeeded/Total	Input
29	json at NativeMethodAccessorImpl.java:0 +details	2021/04/12 01:04:57	6.8 min	20/20	1830.4 MiB

Summary Metrics for 20 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	39 s	1.4 min	1.4 min	1.4 min	1.4 min
GC Time	0.3 s	0.6 s	0.6 s	0.6 s	0.7 s
Input Size / Records	43.5 MiB / 141733	95.5 MiB / 288365	95.6 MiB / 292094	95.6 MiB / 303927	95.6 MiB / 310682

Set the input split size as 100 MB on compressed data

<u>Scenario 6</u>: - Having a lot of small files is worst. This problem is called 'small files problem' and this will degrade the spark performance due to large amount of file open, close and listing dir etc. overhead.

Let's try same 12.4 GB of data split into 795 small files.

This time it took 13 minutes to complete the action.

Till now we are just reading the input data, Have you wondered, spark being a lazy evaluation, why is it running tasks, when we are not calling any action?

This is because spark is inferring the schema from the input data. This is spark default behavior. We can remove this step by supplying the schema definition while reading the data.

If you do not know the schema of the input data, then you can make spark read 10% or 20% of the input data and infer the schema.

Let's check out the examples

Supplying the schema:- Spark has 2 different methods to supply the schema, programmatically or passing a DDL string.

Defining a schema up front has few advantages

- Spark does not have to run a separate job to infer the schema and data types

- You can do a schema validation and detect error early, if input data do not match the required schema

Define Schema for CSV

1. Programmatically

```
# Python code
# import all data types
from pyspark.sql.types import *
# Define the schema
schema = StructType([
            StructField("Region", StringType(), False),
            StructField("Country", StringType(), False),
            StructField("Item Type", StringType(), False),
            StructField("Sales Channel", StringType(), False),
            StructField("Order Priority", StringType(), False),
            StructField("Order Date", StringType(), False),
            StructField("Order ID", IntegerType(), False),
            StructField("Ship Date", StringType(), False),
            StructField("Units Sold", IntegerType(), False),
            StructField("Unit Price", DoubleType(), False),
            StructField("Unit Cost", DoubleType(), False),
            StructField("Total Revenue", DoubleType(), False),
            StructField("Total Cost", DoubleType(), False),
            StructField("Total Profit", DoubleType(), False),
            StructField("id", IntegerType(), False)
      ])
# reading the csv with the schema
sales_data_sch = spark.read.csv('/mnt/e/Training_Data/5m-Sales-
Records/*.bz2',header=True, schema=schema)

sales_data_sch.printSchema()

root
 |-- Region: string (nullable = true)
 |-- Country: string (nullable = true)
 |-- Item Type: string (nullable = true)
 |-- Sales Channel: string (nullable = true)
 |-- Order Priority: string (nullable = true)
 |-- Order Date: string (nullable = true)
```

```
 |-- Order ID: integer (nullable = true)
 |-- Ship Date: string (nullable = true)
 |-- Units Sold: integer (nullable = true)
 |-- Unit Price: double (nullable = true)
 |-- Unit Cost: double (nullable = true)
 |-- Total Revenue: double (nullable = true)
 |-- Total Cost: double (nullable = true)
 |-- Total Profit: double (nullable = true)
 |-- id: integer (nullable = true)

# Show a row
sales_data_sch.show(1,False)
```

```
>>> sales_data_sch.show(1,False)
+--------------------+-------+---------------+-------------+--------------+----------+---------+---------+----------+----------+---------+-------------+--------
|Region              |Country|Item Type      |Sales Channel|Order Priority|Order Date|Order ID |Ship Date|Units Sold|Unit Price|Unit Cost|Total Revenue|Total Co
+--------------------+-------+---------------+-------------+--------------+----------+---------+---------+----------+----------+---------+-------------+--------
|Australia and Oceania|Palau  |Office Supplies|Online       |H             |3/6/2016  |517073523|3/26/2016|2401      |651.21    |524.96   |1563555.21   |1260428.
+--------------------+-------+---------------+-------------+--------------+----------+---------+---------+----------+----------+---------+-------------+--------
only showing top 1 row
```

Print of one sample record with schema defined Programmatically

2. Using DDL

```
# python code
# Define the schema using DDL
schema = "`Region` STRING, \
          `Country` STRING, \
          `Item Type` STRING, \
          `Sales Channel` STRING, \
          `Order Priority` STRING, \
          `Order Date` STRING, \
          `Order ID` INT, \
          `Ship Date` STRING, \
          `Units Sold` DOUBLE, \
          `Unit Price` DOUBLE, \
          `Unit Cost` DOUBLE, \
          `Total Revenue` DOUBLE, \
          `Total Cost` DOUBLE, \
          `Total Profit` DOUBLE, \
          `id` INT"

# reading the csv with the schema
sales_data_ddl = spark.read.csv('/mnt/e/Training_Data/5m-Sales-
Records/*.bz2',header=True, schema=schema)
```

```
sales_data_ddl.printSchema()

root
 |-- Region: string (nullable = true)
 |-- Country: string (nullable = true)
 |-- Item Type: string (nullable = true)
 |-- Sales Channel: string (nullable = true)
 |-- Order Priority: string (nullable = true)
 |-- Order Date: string (nullable = true)
 |-- Order ID: integer (nullable = true)
 |-- Ship Date: string (nullable = true)
 |-- Units Sold: double (nullable = true)
 |-- Unit Price: double (nullable = true)
 |-- Unit Cost: double (nullable = true)
 |-- Total Revenue: double (nullable = true)
 |-- Total Cost: double (nullable = true)
 |-- Total Profit: double (nullable = true)
 |-- id: integer (nullable = true)

sales_data_ddl.show(1,False)
```

```
>>> sales_data_ddl.show(1,False)
+---------------------+-------+---------------+-------------+--------------+----------+---------+---------+----------+----------+---------+-------------+--
|Region               |Country|Item Type      |Sales Channel|Order Priority|Order Date|Order ID |Ship Date|Units Sold|Unit Price|Unit Cost|Total Revenue|To
+---------------------+-------+---------------+-------------+--------------+----------+---------+---------+----------+----------+---------+-------------+--
|Australia and Oceania|Palau  |Office Supplies|Online       |H             |3/6/2016  |517073523|3/26/2016|2401.0    |651.21    |524.96   |1563555.21   |12
+---------------------+-------+---------------+-------------+--------------+----------+---------+---------+----------+----------+---------+-------------+--
only showing top 1 row
```

Print of one sample record with schema defined with DDL

Define Schema for JSON

The Schema definition for the JSON would be identical as CSV. Here is the example of define schema programmatically. We can also use the same DDL definition.

```
# python code

# import all types
from pyspark.sql.types import *

# Define the schema Programmatically
json_schema = StructType([
            StructField("Region", StringType(), False),
            StructField("Country", StringType(), False),
```

```
            StructField("Item Type", StringType(), False),
            StructField("Sales Channel", StringType(), False),
            StructField("Order Priority", StringType(), False),
            StructField("Order Date", StringType(), False),
            StructField("Order ID", IntegerType(), False),
            StructField("Ship Date", StringType(), False),
            StructField("Units Sold", IntegerType(), False),
            StructField("Unit Price", DoubleType(), False),
            StructField("Unit Cost", DoubleType(), False),
            StructField("Total Revenue", DoubleType(), False),
            StructField("Total Cost", DoubleType(), False),
            StructField("Total Profit", DoubleType(), False),
            StructField("id", IntegerType(), False)
      ])

# reading the csv with the schema
sales_data_json_sch = spark.read.option("multiline", "false").json('/
mnt/e/Training_Data/5m-Sales-Records/
5mSalesRecords_id.json', schema=json_schema)

sales_data_json_sch.printSchema()
root
 |-- Region: string (nullable = true)
 |-- Country: string (nullable = true)
 |-- Item Type: string (nullable = true)
 |-- Sales Channel: string (nullable = true)
 |-- Order Priority: string (nullable = true)
 |-- Order Date: string (nullable = true)
 |-- Order ID: integer (nullable = true)
 |-- Ship Date: string (nullable = true)
 |-- Units Sold: integer (nullable = true)
 |-- Unit Price: double (nullable = true)
 |-- Unit Cost: double (nullable = true)
 |-- Total Revenue: double (nullable = true)
 |-- Total Cost: double (nullable = true)
 |-- Total Profit: double (nullable = true)
 |-- id: integer (nullable = true)

sales_data_json_sch.show(5,False)
```

```
>>> sales_data_json_sch.show(5,False)
+---------------------------+-------+---------------+-------------+--------------+----------+---------+---------+----------+----------+---------+-------------+--------
|Region                     |Country|Item Type      |Sales Channel|Order Priority|Order Date|Order ID |Ship Date|Units Sold|Unit Price|Unit Cost|Total Revenue|Total Co
+---------------------------+-------+---------------+-------------+--------------+----------+---------+---------+----------+----------+---------+-------------+--------
|Australia and Oceania      |Palau  |Office Supplies|Online       |H             |3/6/2016  |517073523|3/26/2016|2401      |651.21    |524.96   |1563555.21   |1260428.
|Europe                     |Poland |Beverages      |Online       |L             |4/18/2010 |380507028|5/26/2010|9340      |47.45     |31.79    |443183.0     |296918.6
|North America              |Canada |Cereal         |Online       |M             |1/8/2015  |504055583|1/31/2015|103       |205.7     |117.11   |21187.1      |12062.33
|Europe                     |Belarus|Snacks         |Online       |C             |1/19/2014 |954955518|2/27/2014|1414      |152.58    |97.44    |215748.12    |137780.1
|Middle East and North Africa|Oman   |Cereal         |Offline      |H             |4/26/2019 |970755660|6/2/2019 |7027      |205.7     |117.11   |1445453.9    |822931.9
+---------------------------+-------+---------------+-------------+--------------+----------+---------+---------+----------+----------+---------+-------------+--------
```

Print of five JSON sample records with same schema as CSV

Best Way to infer schema

Some times you might not know the schema of the data beforehand. Or maybe the schema is too big to write using DDL or Programmatically.

Once I had a situation where schema of input data might change over the time and my program should identify the changes and adjust the ELT process accordingly. To handle this situation, I could not use the explicit schema. I would read the 30% of the input data to get the schema, instead of entire input data, using *samplingRation* option. This way I had saved time and processing cost.

```
# python code

# read data, no multi line, infer schema and sampling ration is 30%
sales_data_json_smpl = spark.read.option("multiline", "false").option
("inferSchema","true").option("samplingRation", 0.3).json('/mnt/e/
Training_Data/5m-Sales-Records/5mSalesRecords_id.json')

sales_data_json_smpl.printSchema()

root
 |-- Country: string (nullable = true)
 |-- Item Type: string (nullable = true)
 |-- Order Date: string (nullable = true)
 |-- Order ID: long (nullable = true)
 |-- Order Priority: string (nullable = true)
 |-- Region: string (nullable = true)
 |-- Sales Channel: string (nullable = true)
 |-- Ship Date: string (nullable = true)
 |-- Total Cost: double (nullable = true)
 |-- Total Profit: double (nullable = true)
 |-- Total Revenue: double (nullable = true)
 |-- Unit Cost: double (nullable = true)
 |-- Unit Price: double (nullable = true)
 |-- Units Sold: long (nullable = true)
 |-- id: long (nullable = true)
```

Conclusions

- While reading the input data specify the schema, if you know it
- If you do not know the schema, then read only 20% sample data to get the schema
- Use splittable compression instead of raw file format
- Do not have large quantity of small files, instead have few big files
- Keep the input data split size between 100 MB to 200 MB. Default 128 MB is good enough for most of the cases.