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

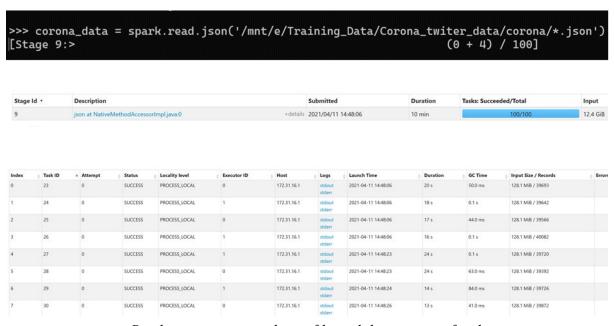
doop Codecs

odec	File Extension	Splittable?	Degree of Compression	Compression Speed
zip	.gz	No	Medium	Medium
zip2	.bz2	Yes	High	Slow
nappy	.snappy	No	Medium	Fast
20	.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.



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.

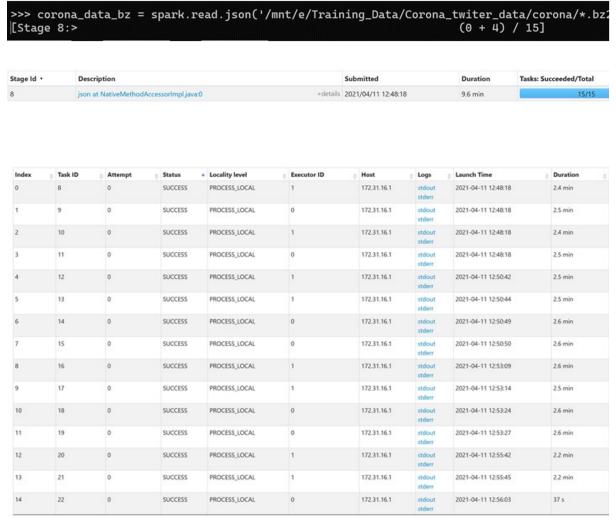
I am using 2 nodes cluster, with 2 CPU cores in each node. Which means spark can have at max, 4 task running in parallel.

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.



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.

Scenario 1: - 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.

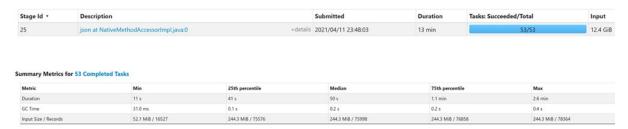
spark.conf.set("spark.sql.files.maxPartitionBytes", 32000000)
corona_data = spark.read.json('/mnt/e/Training_Data/
Corona_twiter_data/corona/*.json')
corona data.count()

Stage Id	•	Description					Submitted		Duration	Tasks: Succe	Input	
24		json at NativeMethodAccessorImpl.java:0		Impl.java:0	+details 2021/0			23:33:57	7.7 min	417/417		12.4 GiE
ımmary l	Metrics for 4	117 Completed To	asks									
Metric			Min		25th percentile		Med	dian	75th percent	tile	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 / 96	14	30.6 MiB / 9924	
ks (417)												
iks (417)	• entries										Search:	
		▲ Attempt	§ Status	Locality level	© Executor ID	(Host	Logs	& Launch Time	Duration	g GC Time	Search:	į Err
how 20	• entries	Attempt 0	Status SUCCESS	Locality level	Executor ID 0	Host 172.31.16.1	Logs stdout stderr	© Launch Time 2021-04-11 23:33:57	Duration 0.4 s	GC Time		o Err
how 20	entries						stdout				Input Size / Records	φ Er
now 20	entries Task ID 252	0	SUCCESS	PROCESS_LOCAL		172.31.16.1	stdout stderr stdout	2021-04-11 23:33:57	0.4 s	4.0 ms	Input Size / Records 30.6 MiB / 9465	ψ Er
now 20	task ID 252 253	0	SUCCESS	PROCESS_LOCAL PROCESS_LOCAL	0	172.31.16.1 172.31.16.1	stdout stderr stdout stderr stdout	2021-04-11 23:33:57 2021-04-11 23:33:57	0.4 s	4.0 ms	Input Size / Records 30.6 MiB / 9465 30.6 MiB / 9520	ψ Er
how 20	• entries • Task ID 252 253 254	0	SUCCESS SUCCESS	PROCESS_LOCAL PROCESS_LOCAL PROCESS_LOCAL	0 1	172.31.16.1 172.31.16.1 172.31.16.1	stdout stderr stdout stderr stdout stderr stdout	2021-04-11 23:33:57 2021-04-11 23:33:57 2021-04-11 23:33:57	0.4 s 0.4 s	4.0 ms 6.0 ms 4.0 ms	Input Size / Records 30.6 MiB / 9465 30.6 MiB / 9520 30.6 MiB / 9460	ψ Err

Set the input split size as 32 MB

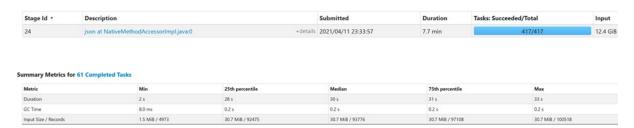
Scenario 2: - Here is the capture of running 12.4 GB uncompressed data using

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.



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.



Set the input split size as 32 MB on compressed data

Scenario 4: - 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.



Set the input split size as 256 MB on compressed data

Scenario 5: - 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 Durati		Tasks: Succeede	ed/Total	Input	
29	json at NativeMethodAccessorImpl java:0 +deta		2021/04/12 01:04:57	6.8 min	6.8 min		1830.4 MiE
ummary Metrics fo	or 20 Completed Tasks	25th percentile	Median	75th per	rcentile	Max	
Metric	A. W. Carlotte S. Carlotte Carlotte Carlotte S. Carlotte Car	25th percentile 1.4 min	Median	75th per 1.4 min		Max 1.4 min	
	Min						

Set the input split size as 100 MB on compressed data

Scenario 6: - 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)
      1)
# 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 Reven	ue Total Co
Australia and Oceania	Palau	Office Supplies	Online	Н	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)
```

Region	Country	 Item Type	Sales Channel	Order Priority	Order Date	Order ID	 Ship Date	Units Sold	Unit Price	Unit Cost	Total Revenue
Australia and Oceania	Palau	Office Supplies	+ Online	+ н	+ 3/6/2016	517073523	+ 3/26/2016	2401.0	651.21	524.96	 1563555.21

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.

```
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)
      1)
# 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(,False)											
Region	Country	Item Type	Sales Channel	Order Priority	Order Date	Order ID	Ship Date	Units Sold	Unit Price	Unit Cost	Total Revenue	Total C
Australia and Oceania	Palau	Office Supplies	Online	н	3/6/2016	 517073523	3/26/2016	2401	651.21	524.96	 1563555.21	1260428
Europe	Poland	Beverages	Online	jL .	4/18/2010	380507028	5/26/2010	9340	47.45	31.79	443183.0	296918.
North America	Canada	Cereal	Online	М	1/8/2015	504055583	1/31/2015	103	205.7	117.11	21187.1	12062.3
Europe	Belarus	Snacks	Online	c	1/19/2014	954955518	2/27/2014	1414	152.58	97.44	215748.12	137780.
Middle East and North Africa	Oman	Cereal	Offline	н	4/26/2019	970755660	6/2/2019	7027	205.7	117.11	1445453.9	822931.

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.