Running Notes

Week12: Apache Spark Structured API Part-2

Structured API's session - 9
==========

1. we read the data from a source and create a dataframe

2. we do bunch of transformations and actions - processing

3. we write the output to target location - sink

saveModes
==========

1. append (putting the file in the existing folder)

2. overwrite (first delete the existing folder, and then it will create a new one)

3. errorIfExists (will give error if output folder already exist)

4. ignore (if folder exist it will ignore)

normally when we are writing a dataframe to our target.

then we have few options to control the file layout

spark file layout
==========

1. Number of files and file size

2. partitioning and bucketing

3. sorted data - sortBy

Note: number of output files is equal to the number of partitions in your dataframe.

1. simple repartiton
==========

it can help you increase the parallelism

df.repartition(4)

with a normal repartition you wont be able to skip some of the partitions for performance improvement

partition pruning is not possible.

2. partitionBy

is equivalent to your partitioning in hive.

it provides partition pruning

3. bucketBy(4,"order_id")

maxRecordsPerFile

csv, parquet, json ...

avro is external and not supported by default.

we need to add a jar.

spark 2.4.4 2.11

spark avro 2.4.4 2.11

Structured API's session - 11
==========

sometimes we have a requirement to save the data in a persistent manner in the form of table.

when data is stored in the form of table then we can connect tableau, power bi etc... for reporting purpose.

table has 2 parts
==========

data	metadata
spark warehouse	catalog metastore
spark.sql.warehouse.dir	in memory (on terminating application it is gone)

we can use hive metastore to handle spark metadata

spark hive 2.4.4 2.11

bucketBy works when we say saveAsTable

Structured API's session - 12
==========

1. Dataframe reader - taking the data from source

2. tranformations to process your data

3. Dataframe writer - to write your data to target location

Transformations
==========

1. Low level Transformations
==========

map

filter

groupByKey

Note: we can perform low level tranformations using raw rdds

some of these are even possible with dataframes and datasets..

2. High level Transformations
==========

select

where

groupBy

Note: These are supported by Dataframes and Datasets..

since this is an unstructured file.

I will load this file as a rdd (raw rdd)

each line of the rdd is of string type..

use a map transformation which is low level transformation.

input to the map tranformation is:

1 2013-07-25 11599,CLOSED

output:

1,2013-07-25,11599,CLOSED

In my map tranformation I will use a regular expression

I will associate the output with the case class

1,2013-07-25,11599,CLOSED

so that we have structure associated..

input to the map is a raw line.

output from the map will be structured line.

if we have schema associated/structure associated we can convert our rdd to a dataset

rdd

then we imposed structure on top of rdd

on structured rdd we call .toDS method to convert it to dataset..

I can do whatever higher level transformations I want to use.

Idea is to give structure to your data and then use high level transformations.

do this as early as possible..

Structured API's session - 13
==========

how to refer a column in a dataframe/dataset

1. column string
==========

ordersDf.select("order_id","order_status").show

2. column object
==========

```
ordersDf.select(column("order_id"),col("order_date"),$"order_customer_id", 'order_status).show
```

```
ordersDf.select(column("order_id"),col("order_status")).show
```

column
col

both of these can be used in pyspark, spark with scala.

scala specific
==========

$"order_id"
'order_id

syntactic sugar but available only for scala

we cannot mix both columns strings and column object in the same statement.

column expression
==========

Note: we cannot mix columns strings with column expression

nor we can mix column object with column expression

column string - select("order_id")

column object - select(column("order_id"))

column expression - concat(x,y)

there is a way to convert column expression to a column object

Structured API's session - 14
==========

UDF (user defined functions)

Structured API's

whenever we want to add a new column we use .withColumn

df.withColumn("adult",)

column object expression udf
==========

df.withColumn("adult",parseAgeFunction(col("age")))

basically we register the function with the driver.

the driver will serialize the function and will send it to each executor.

sql/string expression udf
==========

session takeaways
==========

if you want to add a new column to a dataframe then use .withColumn transformation

how to convert dataframe to dataset , it is by using case class val ds = df.as[Person]

how to convert dataset to a dataframe, using .toDF()
val df1 = ds.toDf()

creating our own user defined function is spark

1. column object expression
it is not registered in catalog.

2. sql expression (easier)
the function is registered in catalog.
so that we will be able to use it with spark sql also.

whenever we register a UDF with driver.

driver will serialize it (convert it into bytes)

and will send it to each executor.

Structured API's session - 15
==========

1,"2013-07-25",11599,"CLOSED"
2,"2014-07-25",256,"PENDING_PAYMENT"
3,"2013-07-25",11599,"COMPLETE"
4,"2019-07-25",8827,"CLOSED"

1. I want to create a scala list - done
2. from the scala list I want to create a dataframe
   orderid, orderdate, customerid, status - done

3. I want to convert orderdate field to epoch timestamp (unixtimestamp) - number of seconds after 1st january 1970 - done

4. create a new column with the name "newid" and make sure it has unique id's
   - done

5. drop duplicates - (orderdate , customerid) - done

6. I want to drop the orderid column - done

7. sort it based on orderdate -

if I want to add a new column or if I want to change the content of a column I should be using .withColumn

Structured API's session - 16
==========

Aggregate transformations

1. Simple aggregations

2. grouping aggregates

3. window aggregates

order_data.csv it is 46 mb file

Simple aggregations
==========

when after doing the aggregations we get a single row.

total number of records, sum of all quantities.

grouping aggregates
==========

in this we will be doing a group by

in the output there can be more than one record.

window aggregates
==========

so we will be dealing with a fixed size window.

Simple aggregations
==========

1. load the file and create a dataframe. I should do it using standard dataframe reader api. - done

Simple Aggregate

totalNumberOfRows, totalQuantity, avgUnitPrice, numberOfUniqueInvoices

2. calculate this using column object expression - done

3. do the same using string expression - done

4. Do it using spark sql - done.

Structured API's session - 17
==========

Grouping Aggregates

group the data based on Country and Invoice Number

I want total quantity for each group, sum of invoice value

1. do it using column object expression - done

2. do it using string expression - done

3. do it using spark sql - done

Structured API's session - 18
==========

simple aggregations

grouping aggregations

window aggregations..

1. parition column - country
2. ordering column - weeknum
3. the window size - from 1st row to the current row

Structured API's session - 19

==========

there are 2 kind of joins

1. Simple join (Shuffle sort merge join)

2. Broadcast join

we have 2 datasets
==========

orders - order_customer_id

customers - customer_id

kind of joins which are possible
==========

1. inner (matching records from both the tables)

we wont see the customer who never placed a order.

2. outer - matching records + non matching records from left table + non matching records from right table

3. left - matching records + non matching records from the left table

4. right - matching records + non matching records from the right table

Lets a some customers never placed a order.. but I do not want to miss on these customers details.

Structured API's session - 20
==========

1. showcasing how your code can lead to ambiguous column names.

this happens when we try to select a column name which is coming from 2 different dataframes..

how to solve this issue..
==========
there are 2 ways to solve this problem

1. this is before the join

you rename the ambiguous column in one of the dataframe

.withColumnRenamed("old_column_name","new_column_name")

2. once the join is done we can drop one of those columns.

.drop
==========

2. how to deal with null's

problem statement
==========
whenever order_id is null show -1

coalesce

Structured API's session - 21
==========

internals of a normal join operation

shuffle..

Simple join involves - Shuffle sort merge join

executor1 - node 1
==========
orders
15192,2013-10-29 00:00:00.0,2,PENDING_PAYMENT
33865,2014-02-18 00:00:00.0,2,COMPLETE

(2,{15192,2013-10-29
00:00:00.0,PENDING_PAYMENT})
(2,{33865,2014-02-18 00:00:00.0,COMPLETE})

customers
3,Ann,Smith,XXXXXXXXX,XXXXXXXXX,3422 Blue
Pioneer Bend,Caguas,PR,00725

(3,{Ann,Smith,XXXXXXXXX,XXXXXXXXX,3422 Blue
Pioneer Bend,Caguas,PR,00725})

it will write the output into the exchange.

exchange is nothing but like a buffer in the executor..

from this exchange spark framework can read it and do the shuffle.

exchange

executor2 - node 2
==========

2,Mary,Barrett,XXXXXXXXX,XXXXXXXXX,9526 Noble Embers Ridge,Littleton,CO,80126

orders
35158,2014-02-26 00:00:00.0,3,COMPLETE
15192,2013-10-29 00:00:00.0,2,PENDING_PAYMENT

exchange

executor3 - node 3
==========

exchange

15192,2013-10-29 00:00:00.0,2,PENDING_PAYMENT
2,Mary,Barrett,XXXXXXXXX,XXXXXXXXX,9526 Noble
Embers

all the records with the same key go to the same reduce exchange.

Structured API's session - 22
==========

1. simple - shuffle
2. broadcast - this does not require a shuffle.

whenever we are joining 2 large dataframes then it will invoke a simple join and shuffle will be required.

when you have one large dataframe and the other dataframe is smaller. in that case you can go with the broadcast join.