pyspark.sqlデータ加工処理

2021年9月10日

ZEAL®

目次

初めに

- 勉強会でglueを扱っているということもあり、
  主にデータ加工に利用するSparkSQLのmethodをSQLとの比較で纏めてみました。
- Spark v3.1.2で検証しております。
- 普段sparkを利用しない為、もっとよい実装方法があるかもしれません。。。
- 特にレビューもしていないので誤字・脱字等の細かいミスはご容赦ください。
- 1つの結果に対して、SQL同様複数の実装方法があるためすべては網羅しておりません。
- 関数は種類が多いため記載しておりません。データ型に関しても記載しておりません。
- SQL文を主に利用して処理を実装することも可能ですが、
  どの程度最適化されるのかが不明なことや、通常のメソッドで実装されているコードを読むことも今後あると思うので、「SQL・メソッド、どちらの処理も書ける」が現状ではベストだと思います。
- 通常のメソッドもSQLに相当近いので、SQLさえ書ければ学習コストは低いと思います。
  ⇒pythonが分からなくても、通常のpythonの実装とは全く違うのでほぼ問題ないと思います。

初めに

- データ加工以外にも、データ読み込み・書き込み・キャッシュの作成・パーティションの作成・削除等sparkを利用するために覚えることは色々とありますが（現状RDDの処理をほぼ覚えなくていいだけハッピーです）、まずは加工処理が書ければいいんじゃないかなと個人的には思ってます。
  逆にそこが書けないと何も実装できないので。。。
- SQLが書ければ構造化データの基本的な加工はほぼほぼ出来ると思うので、
  SQLライクなsparkSQLの各種メソッドを覚えれば、sparkでのデータ加工もかなり出来るようになると思います。
- 関数はSQLと同じで覚えるというよりはその都度調べればいいと思います（こういうのがあったなぁくらいのイメージがあればOK）。
  ⇒何度も使う関数は勝手に覚えるので大丈夫です。
- 後は、案件や興味に応じて、更によりよい実装・複雑なデータ処理・ストリーミング処理・MLなんかを覚えていけばいいのかなと。

FROM句

SQL

```
select
    *
from
    tbl
```

SparkSQL

```
df.show()   #df.select('*').show()と同意

#dfへのカラム追加
df2 = df.withColumn('col1', df.col2 + 100) ※ withColumn(カラム名, 式)

#dfのカラム削除
df2 = df.drop(df.col1) ※df.drop('col1')でも可

#dfのカラム名変更
df2 = df.withColumnRenamed('col1',  'col1_rename')
```

- DataFrame自体がテーブルに相当するものなので、特に「From」句は存在しません
- DFに対して各種メソッドを繋げて処理を実施します
- 「DFのshowメソッド」はDFのデータ内容を表示させます

SELECT句

SQL

```
select distinct
    col1
    ,col2 + 1 as col_add
    ,col3 || 'aaa' as col_concat
    ,case when col4 = 123 then 'xxx'
        when col5 = '345' then 'yyy'
        else null end as col_when
from
    tbl
```

SparkSQL

```
from pyspark.sql import functions as F

df.select('col1', (df.col2 + 1).alias('col_add'),
    F.concat(df.col3, F.lit('aaa')).alias('col_concat'),
    F.when(df.col4 == 123, 'xxx').when(df.col5 == 345,
'yyy').otherwise(None).alias('col_when')
    ).distinct().show()

#DFのselectExprメソッドを使うことでSQL形式で記載も可能
（式内で文字列を利用する場合はクォートに注意）
df.selectExpr('col1', 'col2 + 1 as col_add', 'col3 || "aaa" as col_concat',
'case when col4 = 123 then "xxx" else null end as col_when').show()
```

- select句は「DFのselectメソッド」で対応
- 全カラムをselectする「df.select('*')」も利用可能
- disticnt句は「DFのdistictメソッド」で対応
- カラム名の指定は「'カラム名'」でも「df.カラム名」でも可能
- カラムのメソッドを使いたい場合は「df.カラム名.メソッド名」を利用する
  ※'カラム名'.メソッド名だと文字列のメソッドと認識されるので注意
- カラムに対する四則演算が可能。「+,-,*,/,%」が利用可能
- SQLのas句は「カラムのaliasメソッド」が対応
- 文字列結合は「functionsのconcat関数」が対応
  「functionsのlit(リテラル)関数」はリテラルをカラム型（カラムインスタンス）に変換する関数
- SQLのcase when句は「functionsのwhen関数」で始めて、「カラムのwhenメソッド」を繋げることで対応
  else句はotherwiseメソッドが対応
  ※後続のwhenメソッドは「functionsのwhen関数」の戻り値のカラムに繋がないと例外になるので注意

JOIN句

SQL

```
select
    tbl1.col1, tbl2.co2
from
   tbl
left outer join
   tbl2
on
   tbl1.col1 = tbl2.co2
   and tbl1.co2 = tbl2.col2
```

SparkSQL

```
df1.join(df2, (df1.col1 == df2.col1) & (df1.col2 == df2.col2),
'left').select(df1.col1, df2.col2).show()

#クロスジョインの場合
df1.join(df2, None, 'cross').show() #joinキーは不要のためNoneを指定
df1.crossJoin(df2).show()
```

- join句は「DFのjoinメソッド」で対応
  ジョインの形式を表す第3引数には、
  「 inner(デフォルト) ,left,right,full,cross」等が指定可能

WHERE句

SQL

```
select
   *
from
  tbl
where
  col1 = ‘aaa’
  and col2 is not null
  and col3 like ‘%xxx’
  and col4 in(‘aaa’, ‘bbb’)
```

SparkSQL

```
df.filter((df.col1 == ‘aaa’) & (df.col2.isNotNull()) & (df.col3.like(‘%xxx’)) &
(df.col4.isin(‘aaa’, ‘bbb’))).show()

#betweenも使用可能
df.filter(df.col1.between(1, 100)).show()

#is nullは「DFのisNullメソッド」で対応
df.filter(df.col1.isNull()).show()

#SQL的な記載も可能
df.filter(‘col1 >= 5 and col2 = “xxx” and col3 is not null’).show()
```

- where句は「DFのfilterメソッド」で対応
  ※「DFのwhereメソッド」でも対応
- SQLと同様filter内ではwidow関数は実行できない
- 「functionsのwhen関数／カラムのwhenメソッド」の条件指定も基本同じ。
  SQL的な記載方法は「functionsのexpr関数」を利用することで可能
  df.select(F.when(F.expr(‘col1 is not null’), ‘xxx’)).otherwise(None).show()
- フィルター条件には論理演算子も利用可能
  ※「DFのJOINメソッド」の結合条件でも利用可能

SQL	sparkSQL
and	&
or	|
not	~

GROUPBY句

SQL

```
select
    col1
    ,col2
    ,sum(col3) as col3_sum
from
   tbl
group by
   col1
   ,col2
```

SparkSQL

```
from pyspark.sql import functions as F

df.groupBy(df.col1, df.col2).sum('col3').withColumnRenamed('sum(col3)', 'col3_sum').show()

df.groupBy(df.col1, df.col2).agg(F.sum(df.col3).alias('col3_sum')).show()

#複数カラムの集計を一度に実施したい場合
df.groupBy(df.col1, df.col2).agg(F.sum(df.col3), F.count('*')).show()
```

- groupby句は「DFのgloupByメソッド」で対応
- sum等の集計メソッドはgroupByメソッドに繋げて実行
  ※groupByメソッドで作成された（戻り値の）GroupedDataのメソッドを実行
- 「GroupedDataのaggメソッド」内で「functionsの集計関数」も実行可能
- 「 GroupedData の集計メソッド」を実行すると、カラム名が自動でついてしまう。カラム名も一度に修正したい場合は「 GroupedDataのaggメソッド内でfunctionsの集計関数」と「カラムのaliasメソッド」を組み合わせて対応する

HAVING句

SQL

```
select
    col1
,sum(col2) as col2_sum
from
   tbl
group by
   col1
having
    sum(col2) >= 100
```

SparkSQL

```
from pyspark.sql import functions as F

df.groupBy(df.col1).sum('col2').withColumnRenamed('sum(col2)',
'col2_sum').filter('col2_sum >= 100').show()

df.groupBy(df.col1).agg(F.sum(df.col2).alias('col2_sum')).filter('col2_sum
>= 100').show()

#以下も同じ
df2 = df.groupBy(df.col1).agg(F.sum(df.col2).alias('col2_sum'))
df2.filter(df2.col2_sum >= 100').show()
```

- having句は「DFのfilterメソッド」で対応
- 先行する集計の結果としてDF（集計列も含まれるDF）が返されるので、
  where句に対応するfilterと同じ考え方で処理すればよい

ORDERBY、LIMIT句

SQL

```
select
    *
from
   tbl
order by
   col1
   ,co2 desc
limit 100
```

SparkSQL

```
from pyspark.sql import functions as F

df.orderBy(df1.col1, df1.col2.desc()).limit(100).show()
df.orderBy('col1', df1.col2, ascending=[True, False]).limit(100).show()

#orderByはsortでも同じ
df.sort(df1.col1, df1.col2.desc()).limit(100).show()

#functionsのdesc関数を利用して降順指定も可能（asc関数もあります）
df.orderBy(df1.col1, F.desc(df1.col2)).limit(100).show()
```

- orderby句は「DFのorderByメソッド」で対応
- limit句は「DFのlimitメソッド」で対応
- デフォルトは昇順
  明示的に「カラムのascメソッド／functionsのasc関数」を利用することも可能
- 降順の場合は「カラムのdescメソッド」で対応
- 「 ascending 」引数を利用する場合は、orderByに渡した各カラムに対してbool値（昇順がTrueかFalseかを指定。Falseが降順となる）を指定する

WINDOW関数

SQL

```
select
    col1
    ,sum(col2) over(partition by col1 order by col3 rows between -2 and
current_row) as running_sum
    ,last(col4 ignore nulls) over(partition by col1 order by col3 rows
between unbounded preceding and unbounded following) as last_col
from
   tbl
```

SparkSQL

```
from pyspark.sql import functions as F, Window as W

#paritionby・orderbyともに「df.col・'カラム名'」どちらの指定も可能
wd = W.partitionBy(df.col1).orderBy('col3').rowsBetween(-2,
W.currentRow)
df.select(df.col1, F.sum(df.col2).over(wd).alias('running_sum'),
  F.last(df.col4,
ignorenulls=True).over(W.partitionBy('col1').orderBy('col3').rowsBetwee
n(W.unboundedPreceding,
W.unboundedFollowing).alias('last_col')).show()
```

- window関数は「WindowSpecオブジェクト」を「functionsの集計関数の集計結果カラムのoverメソッド」に渡すことで対応
- 「partitionByとorderByメソッド」を組み合わせて「window」を定義する。orderByのフレーム指定は「rowsBetweenメソッド」が利用可能。
  フレーム範囲の指定に利用できる「currentRow,unboundedPreceding,unboundedFollowing」もWindowオブジェクトの変数として用意されている。
- 定義された「WindowSpecオブジェクト」を、「functionsの集計関数とカラムのoverメソッド」と組み合わせて実行する
- last関数などの一部の関数には、引数にignorenullsが存在するので、SQLと同様の利用方法が可能
- WindowSpecの作成（定義）にて、利用するメソッドの繋ぎ順はどうでも良いが、SQL通りにpartitionBy、orderBy、rowsBetweenの順番が分かりやすいと思う

集合演算

SQL

```
select * from tbl union all select * from tbl2
select * from tbl union select * from tbl2

select * from tbl except all select * from tbl2
select * from tbl except select * from tbl2

select * from tbl intersect all select * from tbl2
select * from tbl intersect select * from tbl2
```

SparkSQL

```
#unionもunionAll同じ処理なので注意
df1.unionAll(df2).show()  # df1.union(df2).show()も同意
#union( distinct)はunionAll(union)とdistinctを組み合わせる
df1.unionAll(df2).distinct().show()

df1.subtract(df2).show()
df1.exceptAll(df2).show()

df1.intersect(df2).show()
df1.intersectAll(df2).show()
```

SQL	sparkSQL
union (distinct)	df.unionAll(df2).distinct() df.union(df2).distinct()
union all	df.unionAll(df2) df.union(df2)
except (distinct)	df.subtract(df2)
except all	df.exceptAll(df2)
intersect (distinct)	df.intersect(df2)
intersect all	df.intersectAll(df2)

演算子

SQL	sparkSQL
and	&
or	|
not	~
+	+
-	-
/	/
*	*
%	%
is null	DF.col.isNull()
is not null	DF.col.isNotNull()
like	DF.col.like()
between	DF.col.between()
in	DF.col.isin()
not in	~DF.col.isin() ※「~」は「not」

SQL	sparkSQL
>	>
<	<
>=	>=
<=	<=
=	==
<>	!=

APPENDIX

SparkSQL

```
#どうしてもSQLで処理をしたい人へ
#glue利用の場合は、SparkSession・SparkAppへの作成方法は任意に修正して下さい（glueと純粋なsparkでは違うと思うので）

from pyspark.sql import SparkSession
ss = SparkSession.builder.appName('test_spark').master('local[*]').getOrCreate()  #ローカル端末で実行を前提に記載

#df作成
df = ss.createDataFrame((('banana', 100), ('watermelon', 1500), ('peach', 300)), ['fruit', 'price'])

#dfをviewとして登録。SQL文のテーブルとして利用。
df.createOrReplaceTempView('fuits')

#SparkSessionのsqlメソッドでSQLを実行。SQLの実行結果がDFとして戻される
#ANSI標準のSQLが記載できるのである程度複雑な処理も記載可能（どこまで複雑なSQLが記載できるかは不明）
#SQLの実行では（各種メソッド利用に比べ）最適化が行われないことが場合によってはありうる様なので、そこだけ注意！
df2 = ss.sql('select * from fruits')
df2.show()

#selectExprメソッドやexpr関数を利用することで一部だけSQLの利用も可能
from pyspark.sql import functions as F

df2 = df.selectExpr('col1 as XXX', 'case when col2 is null then "YYY"  when col3 = "aaa" then "ZZZ" else null end as col4')
df2 = df.select(F.expr('col1 as XXX'), df.col2.alias('YYY'))
```