Part II

Spark SQL and DataFrames

cube and rollup

- The method **cube(col1, .., coln)** of the **DataFrame** class can be used to create a multi-dimensional cube for the input DataFrame
  - On top of which aggregate functions can be computed for each "group"
- The method **rollup(col1, .., coln)** of the **DataFrame** class can be used to create a multi-dimensional rollup for the input DataFrame
  - On top of which aggregate functions can be computed for each "group"

cube and rollup

- Specify which attributes are used to split the input data in "groups" by using **cube(col1, .., coln)** or **rollup(col1, .., coln)**, respectively
- Then, apply the aggregate functions you want to compute for each group of the cube/rollup
  - The result is a DataFrame

cube and rollup

- The same aggregate functions/methods we already discussed for groupBy can be used also for cube and rollup

cube and rollup: Example

- Create a DataFrame from the purchases.csv file
  - The first line contains the header
  - The others lines contain the quantities of purchased products by users
    - Each line contains userid,productid,quantity
- Create a first DataFrame containing the result of the cube method. Define one group for each pair userid, productid and compute the sum of quantity in each group
- Create a second DataFrame containing the result of the rollup method. Define one group for each pair userid, productid and compute the sum of quantity in each group

cube and rollup: Example

- Input file

  userid,productid,quantity

  u1,p1,10

  u1,p1,20

  u1,p2,20

  u1,p3,10

  u2,p1,20

  u2,p3,40

  u2,p3,30

cube and rollup: Example

- Expected output - cube

userid,productid,sum(quantity)

null	null	150
null	p1	50
null	p2	20
null	p3	80
u1	null	60
u1	p1	30
u1	p2	20
u1	p3	10
u2	null	90
u2	p1	20
u2	p3	70

cube and rollup: Example

- Expected output - rollup

userid	productid	sum(quantity)
null	null	150
u1	null	60
u1	p1	30
u1	p2	20
u1	p3	10
u2	null	90
u2	p1	20
u2	p3	70

cube and rollup: Example

```
# Create a Spark Session object
spark = SparkSession.builder.getOrCreate()

# Read purchases.csv and store it in a DataFrame
dfPurchases = spark.read.load("purchases.csv",\
                              format="csv",\
                              header=True,\
                              inferSchema=True)

dfCube=dfPurchases.\
cube("userid","productid“).agg({"quantity": "sum"})

dfRollup=dfPurchases\
.rollup("userid","productid“)\
.agg({"quantity": "sum"})
```

Set transformations

- Similarly to RDDs also **DataFrames** can be combined by using set transformations
  - df1.**union**(df2)
  - df1.**intersect**(df2)
  - df1.**subtract**(df2)

Explain execution plan

- The method **explain()** can be invoked on a **DataFrame** to print on the standard output the execution plan of the part of the code that is used to compute the content of the DataFrame on which explain() is invoked

Broadcast join and DataFrames

- Spark SQL automatically implements a broadcast version of the join operation if one of the two input DataFrames is small enough to be stored in the main memory of each executor

Broadcast join and DataFrames

- We can suggest/force it by creating a broadcast version of a DataFrame
- E.g.,

```
dfPersonLikesBroadcast = dfUidSports\
.join(broadcast(dfPersons),\
dfPersons.uid == dfUidSports.uid)
```

Broadcast join and DataFrames

- We can suggest/force it by creating a broadcast version of a DataFrame
- E.g.,

```
dfPersonLikesBroadcast = dfUidSports\
.join(broadcast(dfPersons),\
dfPersons.uid == dfUidSports.uid)
```

In this case we specify that dfPersons must be broadcasted and hence Spark will execute the join operation by using a broadcast join