Architectures for big data – Exam 20/01/2022

ERP Dataset

ERP is an acronym that stands for enterprise resource planning (ERP). It's a business process management software that manages and integrates a company's financials, supply chain, operations, commerce, reporting, manufacturing, and human resource activities. One important ERP entity is the **Invoice**: it is the only way for a business to business company to get money from customer or to pay suppliers. Each **invoice** is made by several part like the header – the part with general information about customers/suppliers that define the invoice – the list of items, the list of payments, details about the customers, details about the shipping, ...
The exam dataset focuses only on the financial part, so header and payments.

- **Invoice Header**
  - **Description**
    Header of Invoice fiscal documents of our company, with all the information: it is created in the moment an invoice is issued. For legal reason, **is not possible to change an invoice** right after: if there is the need to change it, it should be put in "Cancelled" status and then created one new from scratch! At the beginning of each fiscal year, the DocumentNumber start back from "00000000"
    - 2 Signs:
      - "+" – if the money must be paid from the receiver – or "-" – if the money must be paid from our company
    - CustomerId could be both a Customer if the invoice is from Account Receivable or a Supplier if the invoice is from Account Payable
    - PostingDate: the date the invoice is issued (i.e., created)
    - DueDate is the day the last day the customer can pay us as per the contract done with the customer
  - **Schema**
    DocumentNumber:Int | FiscalYear:Int | PostingDate:Datetime | CustomerId:String | GrossValue:Float | Sign:String ["+","-"]| DueDate:timestamp| DocumentType:string["Invoice", "Debit Note", "Credit Note", "Cancelled"]

- **Invoice Payments**
  - **Description**
    Set of payment given or taken for a given Invoice Header. Each invoice could be paid with multiple payments: every time a new payment is done, a "Closed" payment line is added with the amount of money paid as well as an "Open" line is eventually created with the amount of money still due
    - PaymentStatus
      - Open ➔ the customer has not already paid us – it represents the actual amount of money still the customer should give us
      - Closed ➔ amount of money already paid
  - **Schema**
    DocumentNumber:Int | FiscalYear:Int | PaymentTimestamp:Datetime | PaymentStatus:String | GrossValue:Float | Sign:String

Questions

Q1 – 12 points

[2 points] Define the type of each table (Log or Registry): which are the keys of these tables?

Invoice Header is a registry table, while payment is a log. For the first the keys are DocumentNumber and FiscalYear, while for the second DocumentNumber, FiscalYear, PaymentTimestamp, and PaymentStatus

[2 points] Architect a strategy to integrate them through a CDC job. You can propose a pseudo-code solution for the CDC job, a SQL based solution, a mix of the two.

For Payment table, I use this statement to cdc data
Select * from payment where PaymentTimestamp > $LASTMAXPaymentTimestamp order by PaymentTimestamp

To get **$LASTMAXPaymentTimestamp** I just need to read the last cdc file and read the **PaymentTimestamp** of the last row. This operation make this job completely stateless. I could add an additional check, to check if the last row is well formatted: if not I could delete till I get a fully formatted line and then get the last valid **$LASTMAXPaymentTimestamp**.

For the Header, I could use the standard diff approach: I create everytime a file with tuples Khash,Hash ➔ hash is the hash of all the columns of each row except the values of keys, while khash is the hash of the value of key columns. Then everytime I would like to check for fresh rows, I just make a full table scan, apply same hashing technique and then make a diff between the actual one and the previous (_old represent the previous iteration, _new the new one):

- If Khash_old == Khash_new and Hash_old == Hash_new: do nothing
- If Khash_old == Khash_new and Hash_old != Hash_new: that row has been updated
- If Khash_old not in Khash_new list: the row has been inserted
- Else: the row has been deleted

The status of each invoice must be dynamically computed: if the sum of the closed lines in the payment table is the same of the given InvoiceHeader GrossValue, it means the invoice has a “cleared” status; viceversa, the invoice has a “still open” status. Furthermore, is interesting computing for each invoice if it is “on time” or “overdued”:

- “on time” is
  - any “still open” invoice with a due date greater than today
  - any “cleared” invoice that has been “fully paid” before the due date
- “overdued” is
  - any “still open” invoice with a due date in the past
  - any “cleared” invoice that has been “fully paid” after the due date

[4 points] Architect a spark based job that reads the cdc files created after one month of CDC daily jobs to create an “InvoiceStatusObject”: imagine it is the first time it runs so there is no need to add any incremental logic, just read all the files and compute the logic and write the result. Imagine previous step write a file with the timestamp in the file name in the same folder, and you can use wildcard expression to read multiple files together.

“InvoiceStatusObject”: DocumentNumber:Int | FiscalYear:Int | PostingDate:Datetime | CustomerId:String | GrossValue:Float | Sign:String | DueDate:timestamp | AmountPaid: float | AmountStillToPay:float | Status:string [“still open”, “fully paid”]|Ageing:string[“on time”, “overdued”]

I decided to ignore the open payments considering only the closed one.

- So first [1] I create a new PaymentStatusRdd, where I put for each invoice the sum of all payments received (closed one), as well as the last payment date. Consider that from the text “well as an “Open” line is eventually created with the amount of money still due”, after the last payment a new Open line with 0 of due value is not created (this was not considered in any correction).
- Then I RIGHT OUTER JOIN – to be sure not to lose any invoice – with the header: from the first part of the solution I would have lost not paid invoice if I had used an INNER JOIN

```
[1] PaymentStatusRdd = PaymentsRdd.filter(lambda x: x.get(“status”)==”closed”)
    .map(lambda x: ((x.get("DocumentId"),x.get("FiscalYear")), (x.get("GrossValue") if  x.get("sign")=="+" else -
x.get("GrossValue"),x.get("PaymentDate")))).reduceByKey(lambda x,y: (x[0]+y[0],max([x[1],y[1]]))).persist()
                                #Here I have for each Invoice, the amount of money customer has paid to me and the last payment date

def mapperFunction(row):
        finalRow = row[1][1].copy()
        finalRow["GrossValue"] = row[1][1].get("GrossValue") if  row[1][1].get("sign")=="+"
                                                                else -row[1][1].get("GrossValue")
        if row[1][0] == None:
                row[1][0][0] = 0
                row[1][0][1] = row[1][1].get(“postingDate”)
        finalRow["AmountStillToPay"] = finalRow["GrossValue"]-row[1][0][0]
        finalRow["Status"] = "still open" if finalRow["AmountStillToPay"]!=0 else "fully paid"
        if finalRow["Status"] == "cleared":
                finalRow["Ageing"] = "on time" if row[1][1].get("dueDate")>row[1][0][1] else "overdued"
        else:
                finalRow["Ageing"] = "on time" if row[1][1].get("dueDate")> today() else "overdued"
        #it could be useful to take the last payment date for the cdc later
        finalRow["lastPaymentDate"] = row[1][0][1]
        return finalRow

invoiceStatusObject = PaymenyStatusRdd.rightOuterJoin(invoiceHeader.map(lambda x: ((x.get("DocumentId"),x.get("FiscalYear")),x)))\
                                                        .map(lambda x: mapperFunction(x)).persist()
invoiceStatusObject.count()
```

[4 points] Architect a last job that apply the cdc to get changes over the “InvoiceStatusObject”

To get updated version, I should:

1. Read all Invoices Headers – or all Invoices Headers that are still open and new ones
2. Read Fresh Payments
    a. Incrementally apply to the old invoices
    b. Simply join for the new ones

    This version will be called snapshotInvoiceStatusObject in the rest of correction

Q2 12 points

Assumption

you have already read the 2 tables and you have a session with InvoiceHeaderRDD and InvoicePaymentsRDD where you find read any cdc created over time.

The aesthetic part of the code (e.g., names used for variables) as well as the distributability will be considered.

Write the code to compute:

1. [2 points] How many customers has my company?
   invoiceHeader.map(lambda x: x.get("CustomerId")).distinct().count()
2. [2 points] Extract the list of "Still Open" invoices with this structure
   DocumentNumber:Int | FiscalYear:Int | PostingDate:Datetime | CustomerId:String | GrossValue:Float | Sign:String | DueDate:timestamp | listOfClosedPayments: list<{"PaymentDate":datetime,"Value":float}> | OpenPayment:dict<{"PaymentDate":datetime,"Value":float}>

```
def mapperFun(row):
#it takes the result of the join and give me back (DocumentNumber:Int | FiscalYear:Int (PostingDate | CustomerId | GrossValue | Sign | DueDate | openPayment:dict<paymentDate | paymentValue> | closedPayment:list< paymentDate | paymentValue > ))
def reducerFun(left,right):
    result = left.copy()
    result["closedPayment"] += right["closedPayment"]
    result["openPayment"] = left["openPayment"] if left["openPayment ]["paymentDate"]>right["openPayment ]["paymentDate"] else right
   return result


openPaymentInvoice = snapshotInvoiceStatusObject.filter(lambda x: x.get("Status")=="Still Open").map(lambda x: x.get("DocumentNumber"),x.get("FiscalYear")),x) ).join(paymentsRdd.map(lambda x: x.get("DocumentNumber"),x.get("FiscalYear")),x)).map(lambda x: mapperFun(x))\
.reduceByKey(lambda x,y: reduceFun(x,y)).persist()
openPaymentInvoice.count()
```

3. [2 points] Total gross value of "Still Open" invoices posted during 2021 that should be still paid
   snapshotInvoiceStatusObject.filter(lambda x: x.get("status")=="Still Open").map(lambda x: x.get("still open value")).sum()
4. Create a score of "reliability":
   a. [2 points] By Document. The score for a document is based on the amount of day after posting an invoice needed to pay it: each day after the due date must be count as +5 (e.g., if an invoice is posted on 01/01 with a due date of 01/20 and paid on 01/15, the score for this invoice is +15. The same invoice paid on 01/25 generate a score of +45). This score must be multiplied by the GrossAmount of that given invoice.

```
def addScore(x):
    score = (x.get("DueDate") – x.get("lastPaymentDate")).days()
   if score >0:
      x.get("score") = (x.get("lastPaymentDate")-x.get("postingDate")).days()*x.get("GrossValue")
  else:
    x.get("score") =( (x.get("DueDate")-x.get("postingDate")).days() +  (x.get("LastPaymentDate")-x.get("dueDate")).days())*x.get("GrossValue")
   return x


documentScore = snapshotInvoiceStatusObject.filter(lambda x: x.get("status")=="cleared").map(lambda x: addScore(x))
```

   b. [2 points] The monthly score for each customer is the average score of each invoice in that month.

monthlyScore = documentScore.map(lambda x:
((x.get("customerId"),x.get("PostingDate").yearMonth()),(x.get("score"),1)).reduceByKey(lambda x,y: (x[0]+y[0]),(x[1]+y[1]))

c. [2 points] The global score for each customer is the median score of each month

customerRate = monthlyScore.map(lambda x: (x[0],[x[1]])).reduceByKey(lambda x,y: x+y).map(lambda x: (x[0],median(x[1]))

#in this case the median is acceptable, we know we have very few data points for each customer so the list will have less than 1K points

What are the main pillars of any Software Architectures? Describe each of them with practical examples to show the added value

12:25