Université Sorbonne

Master DAC

Département informatique

BDLE

TP 1-2-3

14 octobre 2022

par

Oussama Sahli

Encadrant universitaire : Hubert Naacke/ Camelia Constantin/ Mohamed-amine Baazizi/ Bernd Amann

Table des matières

Table des figures

Introduction

Dans le tp 1, nous avons vu comment utiliser le langage SQL via l'interface PySpark. Pour cela, nous nous sommes familiariser avec les chaînes de traitements dans Spark, et nous avons réaliser différentes rquêtes en Spark SQL.

Dans le tp 2, on s'est intéressé à l'analyse multidimensionnelle en SQL. On a réalisé plusieurs requêtes multidimensionnelles.

1

TP 1

1.1 Exercice n°1

Soit la table User_visits présentée sur la figure 1.1. Elle permet de donner un aperçu des visites faites par différents utilisateurs sur les points d'intérêt. Dans cette table, on a :

— photoID : l'id de la photo.
— userID : l'id de l'utilisateur.
— date : la date.
— poiID : l'id du point d'intérêt.
— poiTheme : le thème du point d'intérêt.
— poiFreq : la fréquence du point d'intérêt.
— seqID : l'id de la séquence.

photoID	userID	date	poiID	poiTheme	poiFreq	seqID
7941504100	10007579@N00	1346844688	30	Structure	1538	1
4886005532	10012675@N05	1142731848	6	Cultural	986	2
4886006468	10012675@N05	1142732248	6	Cultural	986	2
4885404441	10012675@N05	1142732373	6	Cultural	986	2
4886008334	10012675@N05	1142732445	6	Cultural	986	2
4886009150	10012675@N05	1142916492	6	Cultural	986	3
7054481539	10012675@N05	1319327174	13	Cultural	964	4
6908387594	10012675@N05	1319328255	13	Cultural	964	4
6908381912	10012675@N05	1319331463	13	Cultural	964	4
6908398496	10012675@N05	1319331886	13	Cultural	964	4

FIGURE 1.1 – Schéma de la table User_visits, les 10 premiers n-uplets de la table

De plus, on dispose de la table POI, présentée sur la figure 1.2. Elle donne des informations concernant les différents points d'intérêt. Dans cette table, on a :

— poiID : l'id du point d'intérêt.
— poiName : le nom du point d'intérêt.
— latitude : la latitude du point d'intérêt.
— longitude : la longitude du point d'intérêt.
— theme : le thème du point d'intérêt.

.

poiID	poiName	latitude	longitude	theme
1	Air_Canada_Centre	43.64333	-79.37917	Sport
2	BMO_Field	43.63278	-79.41861	Sport
3	Maple_Leaf_Gardens	43.66222	-79.38028	Sport

FIGURE 1.2 – Schéma de la table POI, les 3 premiers n-uplets de la table

1.1.1 Les 10 POI les plus photographiés

On souhaite connaître les 10 points d'intérêt les plus photographiés. Pour cela, on peut utiliser seulement la table 'User_visits", car elle dispose des informations dont on a besoin, c'est à dire les poiID et les photoID. A chaque visite, on lui associe un photoID. Ainsi, il suffit de faire un 'group by' sur l'attribut poiID , tout en récupérant le nombre de photoID distinct qui lui sont associés. De ce fait, on aura le nombre de photo pour chaque poiID. Ensuite, pour récupérer les 10 poiID avec le nombre de photo le plus élevé, il suffit d'ordonner le résultat par le nombre de photo dans l'ordre décroissant, et de limiter le résultat à 10. Ainsi, cela s'écrit :

```
%%sql
select v.poiID, count(distinct(v.photoID)) as nbPhoto
from user_visits v
group by v.poiID
order by count(distinct(v.photoID)) desc
limit(10);
```

1.1.2 Les visites avec date détaillée

Pour cela, on utilise la table "User_visits", et sur l'attribut date on va extraire les composantes année, mois, jour, heure, minute et seconde. Pour cela, on utilise tout d'abord la fonction 'FROM_UNIXTIME' afin de convertir le timestamp en une date au format 'YYYY-MM-DD HH :MM :SS'. Puis, on utilise la fonction 'EXTRACT' pour à chaque venir extraire l'élément de la date qui nous intéresse pour construire l'attribut souhaité. Pour modifier le nom de l'attribut on utilise 'as'. Pour les secondes, il a fallut utiliser la fonction 'REGEXP_EXTRACT' pour extraire le nombre de secondes, car la valeur retournée, seulement pour ce cas là, était une chaîne de caractère. Ainsi, cela s'écrit :

```
%%sql
create or replace temp view Visite_Date as

select userID, seqID, poiID, FROM_UNIXTIME(date) as date,
EXTRACT(YEAR FROM FROM_UNIXTIME(date)) as annee, EXTRACT(MONTH FROM FROM_UNIXTIME(date)) as mois,
EXTRACT(DAY FROM FROM_UNIXTIME(date)) as jour, EXTRACT(HOUR FROM FROM_UNIXTIME(date)) as heure,
EXTRACT(MINUTE FROM FROM_UNIXTIME(date)) as minute,
CAST ( REGEXP_EXTRACT( EXTRACT(SECOND FROM FROM_UNIXTIME(date)) , "^([0-9]+)\..+", 1)  as Int) as seconde

from user_visits;

select * from Visite_Date
```

1.1.3 Le nombre de POI par utilisateur / Le nombre de POI par séquence

Pour chacune de ces deux requêtes, on va présenter la démarche de résolution de la requête.

Le nombre de POI par utilisateur :

— Pour chaque utilisateurs, on récupère le nombre de poiID distincts qui lui sont associés dans la table 'user_visits'. Pour cela, on fait un group by sur le userID, et on compte le nombre de poiID distincts qui lui sont associés. Pour trier le résultat selon le nombre de poiID par utilisateur dans l'ordre décroissant, on utilise l'instruction 'order by' en spécifiant l'ordre 'desc'.

Ainsi, cela s'écrit :

```
%%sql
select userID, count(distinct(poiID)) as nbPOI
from user_visits
group by(userID)
order by count(distinct(poiID)) desc;
```

Le nombre de POI par séquence :

— Cette requête est dans le même esprit que la requête précédente, sauf qu'ici on groupe sur le seqID. Pour chaque seqID, on récupère le nombre de poiID distinct qui lui sont associés. De plus, pour ne garder que les séquences qui contiennent au moins 3 POI distincts, on rajoute l'instruction 'having', en précisant que le nombre de POI distinct doit être supérieur ou égale à 3. Pour finir, on ordonne le résultat selon le nombre de POI dans l'ordre décroissant, puis selon le seqID dans l'ordre croissant (le premier tri se fait selon le premier attribut placé dans le 'order by'.

Ainsi, cela s'écrit :

```
%%sql
create or replace temp view Seq3plus as
select seqID, count(distinct(poiID)) as nbPOI
from user_visits
group by seqID
having count(distinct(poiID)) >=3
order by nbPOI desc, seqID asc;

select * from seq3plus
order by nbPOI desc, seqID;
```

1.1.4 Trajectoire

On commence par définir une fonction python 'trierPOI(t)', qui prend en paramètre une liste de tuples (date,poiID). Le but de cette fonction est de retourner la liste des POI visités dans l'ordre chronologique. Pour cela, je commence par trier la liste de tuples (date,POI) dans l'ordre croissant selon la valeur de date. Une fois que la liste est triée

selon la date, j'ajoute les POI correspondants au fur et à mesure dans une nouvelle liste que j'ai appelé 'result'. Il faut aussi faire attention à ne pas rajouter deux POI identiques successivement, d'où la condition dans le if.
Ensuite, on déclare la fonction créée dans l'environnement sql en utilisant 'spark.udf.register'. Le type du résultat de cette fonction est un tableau d'entiers. Et, on termine en invoquant la fonction en sql. Dans la requête sql, j'utilise 'collect_list(())' pour générer un tableau de tuples, qui servira de paramètre à la fonction 'trierPOI()'. J'invoque cette fonction pour chaque groupe de userID et seqID en utilisant un 'group by'. Ainsi, cela s'écrit :

```
def trierPOI(t):
  result=[]
  for n in sorted(t):
    if(len(result)!=0) :
        if( result[len(result)-1] != n[1]) :
          result.append(n[1])
    else:
        result.append(n[1])
  return result

spark.udf.register("trierPOI", trierPOI, ArrayType(IntegerType()))
```

```
create or replace temp view Trajectoire as

select userID, seqID, trierPOI(collect_list((date,poiID))) as listePOI
from user_visits
group by userID, seqID;

cache table Trajectoire;

select * from Trajectoire
where size(listePOI) == 6
order by seqID
```

1.1.5 Transitions

Je définit la fonction python 'transition(listePOI)', qui retourne la liste des déplacements entre les différents POI de cette liste. Pour qu'un déplacement entre deux POI soit comptabilisé, il faut que les deux POI soit différents (d'où la condition dans le deuxième if). Ainsi, on forme un tableau de tuples (p1,p2), où le tuple (p1,p2) correspond à un déplacement entre p1 et p2. Ensuite, on déclare la fonction créée dans l'environnement

sql en utilisant 'spark.udf.register'. Le type du résultat de cette fonction est un tableau de tuple, où les deux éléments du tuple sont des entiers.
Dans la requête sql, j'utilise la fonction 'collect_list()' afin de générer un tableau contenant les valeurs de POI pour chaque seqID. Et j'utilise la fonction explode, afin de générer un tuple (p1,p2) pour chaque valeur du tableau retourné par la fonction 'transition()'. Ainsi, cela s'écrit :

```
def transition(listePOI):
  result=[]
  if(len(listePOI)==1):
    return None
  for i in range (0, len(listePOI)-1):
    if(listePOI[i] != listePOI[i+1]):
      result.append((listePOI[i],listePOI[i+1]))
  return result

spark.udf.register("transition", transition, ArrayType( StructType( [StructField("p1" , IntegerType()), StructField("p2" , IntegerType()) ] ) ) )
```

```
%%sql
create or replace temp view Transition1 as
select seqID, explode(transition(collect_list(poiID))) as transition
from user_visits
group by seqID;


select t.seqID, t.transition.p1,  t.transition.p2
from Transition1 t
```

Puis, pour obtenir le nombre de transitions pour chaque déplacements, il suffit de grouper sur les éléments p1 et p2 de l'attribut transition (obtenue précédemment grâce au explode). Ainsi, cela s'écrit :

```
%%sql
create or replace temp view Transition2 as
select t.transition.p1,  t.transition.p2, count(*) as nbtransition
from Transition1 t
group by t.transition.p1,  t.transition.p2;

select * from Transition2
order by nbtransition desc;
```

Transition_relative :

Pour chaque point d'intérêt p1, on commence par calculer le nombre de transitions totale auxquelles il est impliqué. Cela s'écrit :

```
%%sql
create or replace temp view Total_transition as
select p1, sum(nbtransition) as total
from Transition2
group by p1
order by p1 asc;


select * from Total_transition
```

Ensuite, on joint les tables transition2 et total_transition afin de diviser le nombre de transitions par le nombre total de transitions impliqué par p1. Ainsi, on obtient pour chaque déplacement , sa transition relative. Ainsi, cela s'écrit :

```
%%sql
create or replace temp view Transition_relative as
select  t.p1, t2.p2, t2.nbtransition/t.total as rel
from Transition2 t2,Total_transition t
where t2.p1=t.p1 ;


select * from Transition_relative
order by  p1
```

1.1.6 DuréeVisitePOI

Pour chaque groupe de userID,seID,poiID on la calcule la différence entre le premier et dernier évènement, en utilisant max(date)-min(date) (on obtient ainsi la durée en seconde car ce sont timestamp). Ensuite, on utilise cette information comme table, pour calculer pour chaque poiID la durée moyenne de visite de ce POI. Ainsi, cela s'écrit :

```
%%sql
select poiID,avg(duree_visite) as duree_moyenne
from (select userID,seqID,poiID, max(date)-min(date) as duree_visite from user_visits group by userID,seqID,poiID) duree_visite
group by poiID;
```

1.1.7 Distances entre POIs

On commence par définir une fonction distance(a,b,c,d) en python. Cette fonction prend en entrée quatre paramètres qui sont :

— a : la latitude du point 1.
— b : la latitude du point 2.
— c : la longitude du point 1.
— d : la longitude du point 2.

.

Cette fonction retourne la distance en mètre entre deux points 1 et 2. On convertit toutes les mesures en radiant, puis on applique la formule de distance entre deux point (R est le rayon de la terre en mètres). Ensuite, on déclare la fonction créée dans l'environnement sql en utilisant 'spark.udf.register'. Le type du résultat de cette fonction est un double. Puis, pour faire correspondre chaque point avec tous les autres, on utilise 'full join', pour créer toutes les combinaisons possible entre les POI. Et on applique la fonction 'distance' dans la requête sql. Ainsi, cela s'écrit :

```
# les import doivent être fait dans la fonction udf

#retourne la distance en mètre
def distance(a,b,c,d):
  #on convertit les angles en radiant
  import math

  a=a* (math.pi / 180)
  b=b * (math.pi / 180)
  c=c * (math.pi / 180)
  d=d * (math.pi / 180)
  R=6367445  # rayon de la terre en mètre
  return R*math.acos(math.sin(a)*math.sin(b) + (math.cos(a)*math.cos(b)*math.cos(c-d)) )

spark.udf.register("distance", distance, DoubleType() ) # il faut faire attention à bien choisir le type ici, sinon la valeur retourné est null
```

```
%%sql
select p1.poiID,p2.poiID,distance(p1.latitude,p2.latitude,p1.longitude,p2.longitude) as distance
from POI p1 full join POI p2
```

1.2 Exercice n°2

Soit la table 'geonames' où les header n'ont pas été récupéré lors du charchement du fichier, et où les types sont mis en 'string' par défault.

1.2.1 Schéma de Geonames

On construit la table 'Geonames2' en se limitant aux 9 premiers attributs, et en définissant le type et le nom de chaque attribut. j'ai utilisé le lien :
'https ://spark.apache.org/docs/latest/sql-ref-datatypes.html', définit dans le cours, pour

être précis sur la défintion des types.

Soit :

- _co : Il correspond à l'attribut 'geonameID' et est un entier (d'après le fichier 'readme.txt'). De plus, on remarque qu'il peut s'écrire avec de très grand nombre, donc il est de type 'long'.
- _c1 : name est en varchar(200) d'après le fichier 'readme.txt'. Donc son type est varchar(200).
- _c3 : alternate_name est en varchar(10000) d'après le fichier 'readme.txt', donc son type est varchar(10000).
- _c4 : la latitude est en décimal d'après le fichier 'readme.txt'. Donc j'ai utilisé le type double.
- _c5 : la longitude est en décimal d'après le fichier 'readme.txt'. Donc j'ai utilisé le type double.
- _c6 : feature_class est en char(1) d'après le fichier 'readme.txt'. Donc j'ai utilisé le type char(1).
- _c7 : feature_code est en varchar(10) d'après le fichier 'readme.txt'. Donc j'ai utilisé le type varchar(10).
- _c8 : country_code s'écrit sous la forme de deux caractères d'après le fichier 'readme.txt'. Donc j'ai utilisé le type char(2).

.

Ainsi, cela s'écrit :

```
%%sql
create or replace temp view Geonames2 as
select cast(_c0 as long) as geonameID,
cast(_c1 as varchar(200)) as name,
cast(_c3 as varchar(10000)) as alternate_name,
cast(_c4 as double) as latitude,
cast(_c5 as double) as longitude,
cast(_c6 as char(1)) as feature_class,
cast(_c7 as varchar(10)) as feature_code,
cast(_c8 as char(2)) as country_code

from Geonames;

select * from Geonames2
```

1.2.2 Extrait pour le Canada

On récupère les n-uplets de la table 'Geonames2' précédement constuite, dont le country_code = CA (qui correspond au code pays du Canada). Ainsi, cela s'écrit :

```
%%sql
create or replace temp view Geonames_canada as
select geonameID, name, alternate_name, latitude, longitude, feature_class, feature_code from Geonames2
where country_code = 'CA';

cache table Geonames_canada;

select * from  Geonames_canada;
```

1.2.3 Association entre les POI et Geonames

On veut compléter la table POI avec des informations de la table Geonames2. Pour cela on utilise une jointure gauche (left outer join), pour venir alimenter la table POI sans perdre de n-uplets sur cette table si aucune correspondance n'est trouvé avec la table Geonames2. L'attribut de jointure entre les deux tables est poiID (geonameID dans la table Geonames2). Ainsi, cela s'écrit :

```
%%sql
select p.poiID, p.poiName, p.latitude, p.longitude, p.theme, g.name, g.feature_code
from  POI p left outer join Geonames2 g on  p.poiID=g.geonameID
```

2

TP 2-3

2.1 Exercice n°1

2.1.1 Identifier les thèmes

Soit la table 'themes' définit précédemment , qui contient la liste des thèmes distincts. Pour attribuer un numéro à chaque thème, on utilise la fonction 'row_number()' sur chaque n-uplet de la table 'themes'. L'attribution des numéros se fait selon l'attribut 'theme' trié dans l'ordre croissant, via l'instruction 'over (order by theme asc)'. Ainsi, le numéro 1 sera attribué au thème qui apparaît en premier dans la liste des thèmes triée dans l'ordre croissant, et ainsi de suite. Donc, cela s'écrit :

```
%%sql

create or replace temp view Theme1 as
select theme, row_number() over (order by theme asc) as theme_id
from Themes
order by theme asc;

select * from Theme1
```

2.1.2 Classement des séquences par leur plus grand nombre de POI distincts

On utilise la table 'user_visits', où pour chaque groupe de séquence (seqID), on compte le nombre de POI distincts. De plus,on attribut un rang pour chaque séquences selon le nombre de POI distincts qu'elle possède. Pour attribué le rang 1 à la séquence qui contient

le plus grand nombre de POI distincts, on trie le nombre de POI distincts dans l'ordre décroissant lors de l'attribution du rang. De plus, pour des séquences avec le même nombre de POI distincts, on souhaite qu'elles aient le même rang. Aussi, on souhaite que pour la prochaine séquence dont le nombre de POI distincts est différent, on veut que le rang soit initialisé selon le nombre de n-uplets (séquence) déjà traversés. On dit que le rang contient des ex aequos (valeurs non consécutives du rang). Ainsi, cela s'écrit :

```
%%sql

create or replace temp view TopSeq as
select seqID, count(distinct(poiID)) as nbPOI,
rank() over (order by count(distinct(poiID)) desc) as rang
from user_visits
group by seqID
;

select * from TopSeq
```

Identifier les check-ins

Pour qu'il n'y ait pas de doublons sur le triplet (seqID,poiID,date), je fais un group by sur les attributs userID,seqID,poiID,date. De cette manière , si on groupe sur ces attributs, on aura en sortit un n-plet pour chaque combinaison de ces attributs. J'ai décidé de rajouté l'attribut 'userID', car j'en aurai besoin dans les questions suivantes (cela ne pose de problèmes car à chaque séquence est associé un seul 'userID'). Ainsi, cela s'écrit :

```
%%sql
create or replace temp view Visite1 as
select userID,seqID,poiID,date
from user_visits
group by userID,seqID,poiID,date;

select seqID, poiID, date
from Visite1
where seqID in (298, 510)
order by seqID, date, poiID;
```

Montrer qu'il existe des séquences où une même date est associée à plusieurs POI distincts.

Pour chaque groupe de (seqID,date) (un groupe est formé de n-uplets avec le même seqID et la même date), on compte le nombre de POI distincts présent dans chaque groupe. Pour vérifier s'il y a des séquences où une même date est associée à plusieurs POI distincts, on rajoute l'instruction "having", qui servira à ne conserver que les n-uplets en sortit dont le nombre de POI distincts est supérieur ou égale à 2. Ainsi cela s'écrit :

```
%%sql
select seqID,date, count(distinct(poiID)) as nb_POI_meme_date
from visite1
group by seqID,date
having nb_POI_meme_date >=2
order by nb_POI_meme_date desc,seqID;
```

A partir de la table précédente, définir la table Visite2(seqID, poiID, date, num) avec *num* étant le numéro d'ordre dans une séquence

On utilise l'instruction 'partition by' afin de partitionner le résultat de la requête. Ainsi, une partition contient tous les n-uplets ayant la même valeur pour l'attribut seqID. De cette manière les numéros de deux séquences différentes ne seront pas liés. Ensuite, on attribut le numéro d'ordre, dans un séquence, selon la date (trié dans l'ordre croissant) et le poiID (trié dans l'ordre croissant). Ainsi, si dans une même séquences deux n-uplets ont la même date, celui des deux qui aura un numéro d'ordre le plus petit sera celui dont le POID est le plus petit. Par conséquent, dans une même il n'y aura pas de ex aequos (valeurs non consécutives du numéro d'ordre), car précédemment on avait fait en sorte qu'il n'y ait pas de doublons sur le triplet (seqID, poiID, date). Donc, cela s'écrit :

```
%%sql

create or replace temp view Visite2 as
select userID,seqID,date,poiID, rank() OVER( PARTITION BY seqID order by date,poiID) as num
from Visite1;

select *
from Visite2
where seqID in (298, 510)
order by num
```

2.1.3 Identifier les Visites de POI

POI précédent

Sur la table 'Visite1' définit précédemment, on partitionne le résultat de la requête selon l'attribut 'seqID'. Pour chaque partition, on trie le résultat selon numéro d'ordre dans l'ordre croissant, et on sélectionne la valeur du poiID du n-plet qui le précède dans la partition (via l'instruction lag (on choisit également de retourner la valeur null si aucun n-uplet précédent n'est trouvé –> lag(poiID,1)). Ainsi, cela s'écrit :

```
%%sql
create or replace temp view Visite3a as
select userID, seqID,date,poiID,num, lag(poiID, 1)  over (PARTITION BY seqID order by num) as precedent
from Visite2;

select * from Visite3a
where seqID in (298, 510)
order by seqID, date;
```

Début de visite

Pour chaque n-uplet on connaît la valeur du poiID de son précédent dans la même séquence. Ainsi, dans le cas où la valeur du poiID courant est égale à la valeur du poiID précédent (ou si le précèdent est null), on sait qu'il ne s'agit pas du premier tuple d'une série de photos consécutives concernant le même POI. Dans l'autre cas, si le poiID du précédent est différent du poiID courant, alors il s'agit bien du premier tuple d'une série de photos consécutives concernant le même POI. Ainsi, cela s'écrit :

```
%%sql
create or replace temp view Visite3b as
select userID,seqID, date, poiID, num, precedent,
case when poiID==precedent or precedent is null then 0 else 1 end  as debut
from Visite3a ;

select * from Visite3b
where seqID in (298, 510)
order by seqID, date;
```

Ordonner les POI visités

Chaque nouveau début (début=1) correspond à une nouvelle position de POI visité. Ainsi, pour obtenir la position du POI visité, il suffit de faire la somme cumulé de l'attribut début selon le numéro d'ordre attribué au POI dans la séquence. Et pour que la somme

cumulé des débuts soit indépendant selon les séquences, on utilise un 'partition by' sur l'attribut 'seqID'. Ainsi, cela s'écrit :

```
%%sql
create or replace temp view Visite3 as
select userID,seqID, num, date, poiID, sum(debut) OVER (partition by seqID order by num) as poiPosition
from visite3b
;

select * from Visite3
where seqID in (298, 510)
order by seqID, date, poiID;
```

2.1.4 Durée de visite d'un POI

Pour chaque groupe (userID,seqID,poiPosition, poiID), je calcule la différence entre la plus grande date du groupe et la plus petite date du groupe. Je décide de garder également date_debut (min(date)) et date_fin (max(date)) car je les utiliserai plus tard. Ainsi, cela s'écrit :

```
%%sql

create or replace temp view Visite4 as
select userID,seqID, poiPosition, poiID, min(date) as date_debut, max(date) as date_fin, max(date)-min(date) as duree
from Visite3
group by userID,seqID, poiPosition, poiID
;

select * from Visite4
where seqID in (298, 510)
order by seqID, poiPosition
```

Voici le résultat de la vérification :

```
%%sql
select round(avg(duree)/60,1) as duree_moyenne_en_minutes
from Visite4
where duree>0
```

1 entry Filter

duree_moyenne_en_minutes
65.8

On obtient bien que la durée moyenne de visite d'un POI en minute est égale à 65,8.

Nombre moyen de visites et nombre moyen de POI dans une séquence

On utilise la table visite4 qui contient toutes les visites de POI. On commence par compter le nombre de visite et le nombre de POI distinct pour chaque séquences, via l'instruction 'group by' sur l'attribut seqID. Ensuite, on utilise ce résultat pour faire la moyenne du nombre de visite et, la moyenne du nombre de POI distinct. Ainsi, cela

s'écrit :

```
%%sql
create or replace temp view SeqNbVisite as
select avg(nb_visite) as moyenne_nb_visite, avg(nb_POI) as moyenne_nb_POI
from (select seqID, count(*) as nb_visite, count(distinct(poiID)) as nb_POI
from Visite4
group by seqID);

select moyenne_nb_visite, moyenne_nb_POI
from SeqNbVisite
```

On obtient bien que le nombre moyen de visites dans une séquence est 1.30328 et, que le nombre moyen de POI dans une séquence est 1.255902.

Nombre de séquences selon leur nombre de visites.

Afficher le nombre de séquences pour chaque nombre de visites existant. Afficher aussi, pour chaque nombre de visite, le nombre cumulé de séquences ayant au moins ce nombre de visite :

Dans un premier temps, je commence par créer une table 't1', dans laquelle j'affiche le nombre de séquences pour chaque nombre de visites existant. Pour faire cela, je commence par calculer le nombre de visites pour chaque séquence (group by sur seqID). Ensuite, j'utilise ce résultat, tel que pour chaque nombre de visite je compte le nombre de séquences associés (via un group by sur le nombre de visite). Ainsi , j'obtiens la table t1. Cela me permet également de connaître le nombre de ligne dans ma table lorsque que j'utiliserai une fenêtre glissante juste après.

Dans un second temps, j'utilise une fenêtre glissante : ROWS, pour calculer la somme cumulée du nombre de séquences, selon le nombre de visites trié dans l'ordre croissant , entre le n-uplet courant et les m (m=17 ici,le but est d'aller jusqu'à la fin de la table) n-uplets suivants. Ce cumul correspond au nombre de séquences avec au moins nbVisite. Ainsi, cela s'écrit :

```
%%sql
create or replace temp view t1 as
select nbVisite, count(*) as nbSequences
from (select seqID, count(*) as nbVisite
from Visite4
group by seqID)
group by nbVisite
order by nbVisite;

select * from t1
```

Puis,

```
%%sql
SELECT nbVisite, nbSequences,
sum(nbSequences) OVER (order by nbVisite  rows between current row and 17 following) as nbre_sequences_avec_au_moins_nbVisite
FROM t1
```

Nombre de séquences selon leur nombre de POI distincts :

Dans un premier temps, je commence par créer une table 't2', dans laquelle j'affiche le nombre de séquences pour chaque nombre de POI distincts. Pour faire cela, je commence par calculer le nombre de POI distincts pour chaque séquences (group by sur seqID). Ensuite, j'utilise ce résultat, tel que pour chaque nombre de POI distinct je compte le nombre de séquence associé (via un group by sur le nombre de POI distincts). Ainsi, j'obtiens la table 't2'. Cela me permet également de connaître le nombre de ligne dans ma table lorsque que j'utiliserai une fenêtre glissante juste après.

Dans un second temps, j'utilise une fenêtre glissante : ROWS, pour calculer la somme cumulée du nombre de séquences, selon le nombre de POI distinct trié dans l'ordre croissant , entre le n-uplet courant et les m (m=11 ici,le but est d'aller jusqu'à la fin de la table) n-uplets suivants. Ce cumul correspond au nombre de séquences avec au moins nbPOI. Ainsi, cela s'écrit :

```
%%sql
create or replace temp view t2 as
select nbPOI, count(*) as nbSequences
from (select seqID, count(distinct(poiID)) as nbPOI
from Visite4
group by seqID)
group by nbPOI
order by nbPOI;

select * from t2
```

Puis,

```
%%sql
SELECT nbPOI, nbSequences,
sum(nbSequences) OVER (order by nbPOI  rows between current row and 11 following) as nbre_sequences_avec_au_moins_nbPOI
FROM t2
```

Nombre de séquences ayant au moins un POI visité 2 fois :

Je commence par calculer pour chaque groupe (seqID,poiID) le nombre de visite pour le POI. Je stocke ce résultat dans une table 'n'. Ensuite, je compte le nombre de seqID distinct sur la table 'n', en ne gardant que les goupes (seqID,poiID) dont le nombre de visites du POI est supérieur ou égale à 2. Ainsi, cela s'écrit :

```
%%sql
create or replace temp view n as
select seqID,poiID, count(*) as nb_visite_poi
from Visite4
group by seqID, poiID;

select count(distinct(seqID)) as nbSequences from n
where nb_visite_poi >=2;
```

2.1.5 Nombre de visites sur une semaine glissante

Définir la table Visite5a(userID, annee, mois, jour, nbVisite) : nbVisite est le nombre de visites qu'un utilisateur a fait chaque jour :

Dans un premier temps, j'utilise la table 'visite4' dans laquelle sont stockés les visites.

Dans cette table, j'avais conservé la date de début de chaque visite. Ainsi, je commence par créer une table 'c', dans laquelle à partir de la date de début d'une visite, j'extrais l'année, le mois et le jour. Je garde également la date en mettant à 0 les heures, minutes et secondes (afin de faire en sorte que des visites ayant lieu le même jour et à des heures différentes soient associées aux même jour) ; cela me sera utile pour la question suivante.

Dans un second temps, j'utilise la table c, afin de compter le nombre de visites pour chaque jour. Pour cela je fais un group sur les attributs userID,date,annee, mois, jour (j'inclus date dansle group by car j'en aurai besoin dans la question suivante).
Ainsi, cela s'écrit :

```
%%sql
create or replace temp view c as
select userID,seqID, FROM_UNIXTIME(date_debut,'yyyy-MM-dd 00:00:00') as date,
EXTRACT(YEAR FROM FROM_UNIXTIME(date_debut)) as annee, EXTRACT(MONTH FROM FROM_UNIXTIME(date_debut)) as mois,
EXTRACT(DAY FROM FROM_UNIXTIME(date_debut)) as jour
from visite4;
```

Puis,

```
%%sql
create or replace temp view Visite5a as
select userID, date, annee, mois, jour, count(*) as nbVisite
from c
group by userID, date, annee, mois, jour;

select * from visite5a
order by nbVisite desc
```

En déduire la table Visite5b(userID, anne, mois, jour, nbVisite7jours) : nbVisite7jours étant le nombre de visites effectuées sur une semaine glissante :

J'utilise la table précédemment construite qui s'appelle 'visite5a'. Sur cette table, je calcule la somme cumulée du nombre de visites selon la date (convertit en timestamp pour le calcul du range) triée dans l'ordre croissant en ne conservant que les n-uplets dont la date est comprise entre la date du n-uplet courant et 7 jours auparavant (7 jours = 604800 secondes). De plus, je partitionne le résultat de cette requête selon l'attribut 'userID' afin que le cumul du nombre de visites sur une semaine glissante soit indépendant entre chaque 'userID'. Ainsi, cela s'écrit :

```
%%sql
create or replace temp view Visite5b as
select userID,annee, mois, jour, date,
sum(nbVisite) OVER( PARTITION BY userID  ORDER BY  UNIX_TIMESTAMP(date)  RANGE between 604800 preceding and 0 following
 ) as nbVisite7jours
from visite5a ;

select * from visite5b
order by nbVisite7jours desc
```

2.1.6 Déplacements entre deux POI

Je commence par définir pour chaque visite d'une séquence la date de début de la prochaine visite dans cette même séquence. Pour cela je fais un partitionnement selon 'seqID' en triant les n-uplets selon la valeur de leur poiPosition, et où je garde seulement le n-uplet courant et le n-uplet suivant (via rows between current row and 1 following) . Cela me permet de récupérer la dernière valeur de l'attribut 'debut_date' dans la partition (via last_value(date_debut)), qui est celle de la prochaine visite. Ensuite, sur ce résultat je calcule la différence entre la date_debut de la prochaine visite et la date de fin du n-uplet courant. Ainsi, cela s'écrit :

```
%%sql
create or replace temp view Duree_Deplacement as
select seqID,poiPosition,poiID, date_debut, date_fin, date_debut_suivant-date_fin as deplacement

from
(select seqID,poiPosition,poiID, date_debut, date_fin, LAST_VALUE(date_debut)
over (partition by seqId order by poiPosition rows between current row and 1 following) as date_debut_suivant
from visite4 );
```

2.2 Exercice 2 : YFCC

Soit la table 'ycc_france', tel que le schéma est le suivant :

```
|-- Line: long (nullable = true)
|-- PhotoID: long (nullable = true)
|-- PhotoHash: string (nullable = true)
|-- UserNSID: string (nullable = true)
|-- UserNickname: string (nullable = true)
|-- DateTaken: string (nullable = true)
|-- DateUploaded: long (nullable = true)
|-- CaptureDevice: string (nullable = true)
|-- Title: string (nullable = true)
|-- Description: string (nullable = true)
|-- UserTags: string (nullable = true)
|-- MachineTags: string (nullable = true)
|-- Longitude: float (nullable = true)
|-- Latitude: float (nullable = true)
|-- Accuracy: integer (nullable = true)
|-- URL: string (nullable = true)
|-- DownloadURL: string (nullable = true)
|-- LicenseName: string (nullable = true)
|-- LicenseURL: string (nullable = true)
|-- ServerID: integer (nullable = true)
|-- FarmID: integer (nullable = true)
|-- Secret: string (nullable = true)
|-- SecretOriginal: string (nullable = true)
|-- Extension: string (nullable = true)
|-- Marker: integer (nullable = true)
```

FIGURE 2.1 – Schéma de la table 'ycc_france'

Dans cette table, chaque n-uplet a un attribut 'Line' unique et un attribut 'photoID' unique. De plus, les POI sont identifiés via les attributs 'latitude' et 'longitude'. Et l'attribut 'UserNSID' est l'id de l'utilisateur. Et l'attribut 'DateTaken' est la date à laquelle la photo a été prise, elle est sous la forme 'yyyy-MM-dd HH :mm :ss'.

2.2.1 Question 1

Je commence par extraire l'année, le mois et le jour de l'attribut 'DateTaken', et je stocke ce résultat dans la table 'y1' (dans laquelle je conserve égalements la latitude et la longitude du point pour pouvoir les utiliser et l'id de l'utilisateur). Ainsi, cela s'écrit :

```
%%sql
create or replace temp view y1 as
select Line, PhotoID, UserNSID,DateTaken,
EXTRACT(YEAR FROM DateTaken) as annee,
EXTRACT(MONTH FROM DateTaken) as mois,
EXTRACT(DAY FROM DateTaken) as jour, latitude, longitude
from yfcc_france
order by annee,mois,jour;
```

Ensuite, je crée la table y2 à partir de la table y1. Dans cette table, je fais en sorte que pour chaque groupe (annee, mois, jour), je récupère le nombre de POI distincts. Une séquence est définit par le triplet (annee,mois,jour). Car on sait qu'une séquence fait référence à un seul jour (précédemment on a vu que dans une séquence il y avait plusieurs timestamp, mais tous ces timestamp font référence à la même journée , mais à des moment de la journée différents ou non). De plus, je ne garde seulement que les séquences dont le nombre de POI distincts est supérieur ou égale à 3. Je garde également pour chaque séquence la liste de tous les tuples ((UserNSID,latitude,longitude)) correspondant à cette séquence. Ainsi, cela s'écrit :

```
create or replace temp view y2 as
select annee,mois,jour, count(distinct(latitude,longitude)) as nb_poi_distinct,
collect_list((UserNSID,latitude,longitude)) as nb_user_distinct
from y1
group by annee, mois, jour
having nb_poi_distinct >=3
order by nb_poi_distinct desc;

select * from y2
```

Ensuite, je définis une fonction 'poi_users_3(l,d_aprox)', qui retourne True si chaque point de la séquence est associé à au moins 3 utilisateur, sinon False. De plus, dans cette fonction je définis le paramètre d_aprox qui permet de considérer que deux points avec une distance de d_aprox mètre sont identiques. Dans cette fonction, je compte le nombre d'utilisateurs distincts pour chaque point de la séquence. J'ai réutilisé la fonction 'distance' que j'avais définis dans le tp1 précédemment. Ainsi, je crée la table y3 qui contient l'attribut poi_user_3, et je garde seulement les séquences qui où l'attribut poi_user_3 est égale à True (celles dont chaque points est associés à au moins 3 utilisateurs). Donc, cela s'écrit :

```
# les import doivent être fait dans la fonction udf

#retourne la distance en mètre
def distance(a,b,c,d):
  #on convertit les angles en radiant
  import math

  a=a* (math.pi / 180)
  b=b * (math.pi / 180)
  c=c * (math.pi / 180)
  d=d * (math.pi / 180)
  R=6367445  # rayon de la terre en mètre
  return R*math.acos(math.sin(a)*math.sin(b) + (math.cos(a)*math.cos(b)*math.cos(c-d)) )

spark.udf.register("distance", distance, DoubleType() ) # il faut faire attention à bien choisir le type ici, sinon la valeur retourné est null
```

Puis,

```
#on peut définir une fonction qui retourne true si chaque POI a au moins 3 user différents
def pois_users_3(l,d_aprox):
  D=dict()

  for t in l:
    point=(t[1],t[2])
    if(point not in D):
      # je regarde si il est proche d'un autre point d'une distance de d mètre
      ok=0
      #print(point)
      #print("d:")
      for cle, valeur in D.items():
        d=distance( cle[0],point[0],cle[1],point[1] )
        #print(d)
        if( d <= d_aprox):
          ok=1
          if(t[0] not in D[cle] ) : # si c'est un nouvel utilisateur je l'ajoute
            D[cle].append(t[0])
          break
      # sinon j'ajoute ce nouveau point
      if(ok==0): # n'est proche d'aucun point déjà trouvé.
        D[point]=[t[0]]


    elif (t[0] not in D[point] ):
      D[point].append(t[0])

  print(D)
  for cle, valeur in D.items():
      if(len(valeur) < 3):
        return False
  return True


spark.udf.register("pois_users_3", pois_users_3, BooleanType())
```

Et,

```
%%sql
create or replace temp view y3 as
select annee, mois, jour, nb_poi_distinct, pois_users_3(nb_user_distinct,10000) as poi_users_3
from y2
where pois_users_3(nb_user_distinct,10000) == true
```

Ainsi, dans la table y3 sont stockés les séquences qui vérifient les conditions de l'énoncer (une séquence est repérée via le triplet (annee,mois,jour)).

2.2.2 Question 2

Je commence par extraite de la date, l'annee, le mois et le jour sur la table 'yfcc_france'. je récupère aussi les dimensions qui serviront à l'analyse que je ferais plus tard. Les dimensions que je récupère sont l'accuracy (attribut Accuracy) et le titre (attribut Title).

La date composé des trois attributs annee, mois et jour est une dimension à trois niveaux. Ce résultat est stocké dans la table yfcc_france_date. Ainsi, cela s'écrit :

```
# ycc_france avec date
# je récupère l'annee, le mois et le jour de la date, ainsi que les dimensions à analyser plus tard (Accuracy et title)
%%sql
create or replace temp view yfcc_france_date as
select EXTRACT(YEAR FROM DateTaken) as annee,
EXTRACT(MONTH FROM DateTaken) as mois,
EXTRACT(DAY FROM DateTaken) as jour, Accuracy, Title
from yfcc_france
order by annee,mois,jour;

select * from yfcc_france_date
```

Ensuite, pour construire ma table de fait avec les dimensions date (annee,mois,jour), accuracy et title ; je joins la table 'yfcc_france' précédemment construite avec la table 'y3' qui contient les séquences. Ainsi, j'obtiens des informations supplémentaires sur toutes les séquences comprises dans la table 'y3'. De plus, je garde l'attribut 'nb_poi_distinct' qui me servira d'analyse juste après. J'enregistre tout cela dans la table 'analyse'. Ainsi, cela s'écrit :

```
%%sql
create or replace temp view analyse as
select y.annee, y.mois, y.jour, y.nb_poi_distinct, yd.Accuracy, yd.Title
from yfcc_france_date yd, y3 y
where yd.annee= y.annee and yd.mois= y.mois and yd.jour = y.jour
order by annee,mois,jour;

select * from analyse
```

Dans l'analyse que j'ai choisis de faire, j'analyse le nombre moyen de POI distincts sur un ensemble d'agrégation, en utilisant 'group by cube'. Cela me permet d'analyser le nombre moyen de POI distincts totale. Puis le nombre moyen de POI distincts sur chaque dimensions(annee et mois et jour et accuracy et title). Ensuite le nombre moyen de POI distincts sur chaque paire de dimensions (annee,mois) et (annee,jour), (annee,accuracy), etc. Et ainsi de suite jusqu'à calculer le nombre moyen de POI distincts sur chaque quintupler de dimension (car il y a 5 dimensions –> 2 dimensions + 1 dimension à trois niveaux = 5 dimensions). Ainsi, cela s'écrit :

```
%%sql
Select
annee, mois,jour, Accuracy, title,
avg(nb_poi_distinct) as nb_poi_moyen
From analyse
Group by cube (annee, mois,jour, Accuracy, title)
Order by annee, mois,jour, Accuracy, title
```

Et on obtient :

annee	mois	jour	Accuracy	title	nb_poi_moyen
NaN	NaN	NaN	NaN		6.733333333333333
NaN	NaN	NaN	NaN		10.0
NaN	NaN	NaN	NaN	04-12-01929.jpg	3.0
NaN	NaN	NaN	NaN	04-12-01934.jpg	3.0
NaN	NaN	NaN	NaN	100_0084	3.0
NaN	NaN	NaN	NaN	100_0086	3.0
NaN	NaN	NaN	NaN	432.sacredheart03	7.0
NaN	NaN	NaN	NaN	441.paris02	7.0
NaN	NaN	NaN	NaN	442.paris03	7.0
NaN	NaN	NaN	NaN	443.paris04	7.0

Show 10 per page

1 2 3 4 5 6 7 8 9 10

On peut lire sur cette image que par exemple, le nombre moyen de POI distinct pour le titre ’100_0084’ est 3.