CONSIGNES GÉNÉRALES

DOCUMENTS NON AUTORISES
CE CAHIER D'EXAMEN COMPORTE 19 PAGES + 1 PAGE VIDE
LES RÉPONSES DOIVENT ÊTRE ÉCRITES DANS
LES ESPACES RÉPONSES DÉDIÉES
EN CAS DE BESOIN UTILISER LA PAGE VIDE EN FIN DU CAHIER, DANS CE CAS, IL FAUT LE SIGNALER DANS LES CASES RÉPONSES ALLOUÉES
L'USAGE DES CALCULATRICES EST INTERDIT
Il FAUT RESPECTER IMPÉRATIVEMENT LES NOTATIONS DE L'ÉNONCÉ
VOUS POUVEZ ÉVENTUELLEMENT UTILISER L'ANNEXE

Le sujet comporte deux parties qui traitent les aspects suivants :

Partie I : Programmation procédurale (Q1..Q6).

Partie II : Base de données relationnelle (Q7..Q22).

Partie I : Programmation Procédurale

L'objectif de ce problème est d'étudier la performance de quelques **algorithmes d'ordonnancement de processus** dans un système d'exploitation. Un **système d'exploitation** est un logiciel qui permet d'exploiter les ressources d'une machine, à titre d'exemples : Windows, Linux, MacOs, Android, etc.

Un **processus** est défini comme étant un programme chargé en mémoire centrale en vue d'être exécuté par le processeur. Un processus est donc né lors du chargement d'un programme et se termine à la fin de l'exécution de ce programme. Un processus peut être à l'un des états suivants :

- **Prêt :** s'il dispose de toutes les ressources nécessaires à son exécution à l'exception du processeur.
- **Elu :** s'il est en cours d'exécution par le processeur.
- **Bloqué :** s'il est en attente d'un événement ou bien d'une ressource pour pouvoir continuer son exécution.

Un **ordonnanceur** est un processus important du système d'exploitation qui gère l'allocation du temps processeur. Plusieurs processus peuvent être à l'état `"Prêt"`, dans ce cas l'ordonnanceur se charge de choisir parmi eux celui qui devra être traité en premier lieu selon une **stratégie d'ordonnancement**. Un ordonnanceur fait face à deux problèmes principaux :

— Comment élire un processus à exécuter parmi les processus prêts ?

— Durant combien de temps le processeur est alloué au processus élu ?

Dans ce qui suit on s'intéresse à l'étude de deux stratégies d'ordonnancement :

— **FCFS** (First-Come First-Served) : traite les processus dans l'ordre de leurs soumissions (temps d'arrivée) sans aucune considération de leurs durées d'exécution. L'organisation de la file d'attente des processus prêts est donc du type "First In First Out". L'algorithme consiste à choisir le processus disposant du temps d'arrivée minimal et l'exécuter jusqu'à l'achèvement de sa durée d'exécution. Ce procédé est appliqué itérativement jusqu'à épuisement des processus.

— **SJF** (Shortest Job First) : choisit de façon prioritaire le processus ayant la plus courte durée d'exécution en tenant compte des temps d'arrivée de tous les processus. Ce procédé est appliqué itérativement jusqu'à épuisement des processus.

Pour un échantillon donné de processus, un algorithme d'ordonnancement est jugé performant sur la base de la comparaison de deux critères, le temps de réponse moyen (TRM) en premier lieu et le temps d'attente moyen (TAM) en second lieu.

Dans ce qui suit, on se base sur un exemple pour illustrer le déroulement de la tâche d'ordonnancement pour plusieurs processus P_i où chaque P_i est caractérisé par :

- un nom NomP_i
- un temps d'arrivée TaP_i
- une durée d'exécution DeP_i
- un état EtatP_i : `"Prêt"`, `"Elu"` ou `"Bloqué"`
- un temps calculé de fin d'exécution TfP_i

Exemple : La table suivante présente un échantillon de processus à l'état `"Prêt"`. Le temps est exprimé en nombre de cycles d'horloge du processeur.

Processus $NomP_i$	Temps d'arrivée $\texttt{Ta}P_i$	Durée d'exécution $\texttt{De}P_i$
P_1	0	3
P_2	2	6
P_3	4	4
P_4	6	5
P_5	8	2

TABLE 1 – Processus à l'état "Prêt"

Les figures 1 et 2 illustrent respectivement les diagrammes de **GANT** associés aux déroulements des exécutions des processus donnés dans la table 1 selon les algorithmes d'ordonnancement **FCFS** et **SJF**. Les flèches verticales représentent les temps d'arrivées des processus.

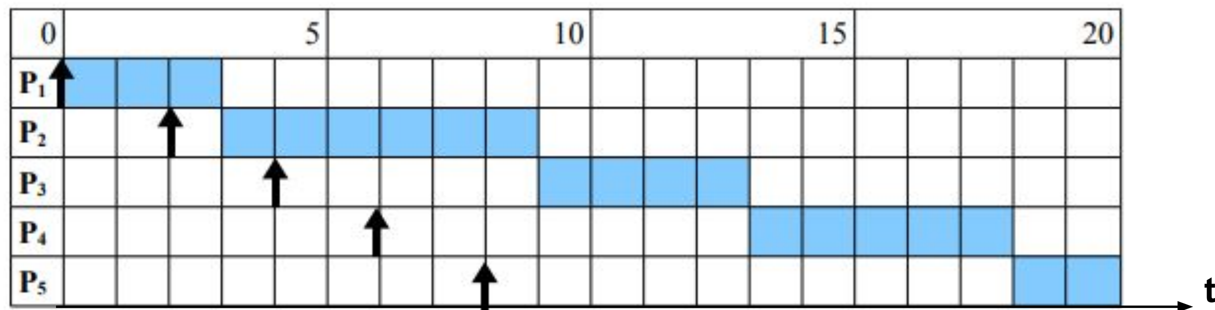


FIGURE 1 – Diagramme de déroulement de FCFS

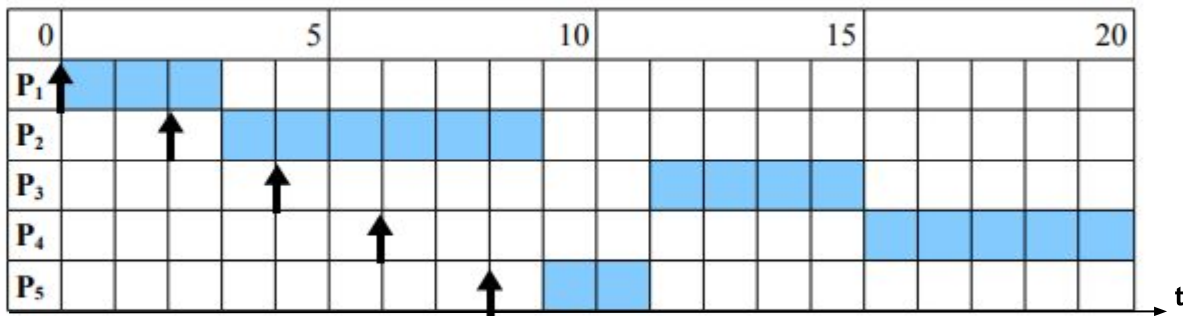


FIGURE 2 – Diagramme de déroulement de SJF

— **TRM** : moyenne des intervalles de temps séparant les temps d'arrivée et les temps de fin d'exécution de tous les processus. Ce temps de réponse est calculé selon la formule suivante :

$$\textbf{TRM} = \sum_{i=1}^{n} \frac{\texttt{Tr}P_i}{n} \text{ où } \begin{cases} \texttt{Tr}P_i = \texttt{Tf}P_i - \texttt{Ta}P_i & \text{temps de réponse d'un processus } P_i \\ \text{et} & \\ n & \text{nombre total de processus achevés} \end{cases}$$

Équation 1 – Calcul du TRM

— **TAM** : moyenne des intervalles de temps séparant les temps d'arrivées des processus et leurs mises en exécution. Ce temps d'attente est calculé selon la formule suivante :

$$\textbf{TAM} = \sum_{i=1}^{n} \frac{\texttt{Tat}P_i}{n} \text{ où } \begin{cases} \texttt{Tat}P_i = \texttt{Tr}P_i - \texttt{De}P_i & \text{temps d'attente d'un processus } P_i \\ \text{et} & \\ n & \text{nombre total de processus achevés} \end{cases}$$

Équation 2 – Calcul du TAM

Les tables 2 et 3 illustrent respectivement les temps décrits précédemment pour les processus donnés dans la table 1 selon les algorithmes d'ordonnancement **FCFS** et **SJF**.

P_i	Temps de réponse $\texttt{Tr}P_i$	Temps d'attente $\texttt{Tat}P_i$
P_1	3	0
P_2	7	1
P_3	9	5
P_4	12	7
P_5	12	10
	TRM	TAM
	8.6	4.6

Table 2 – Critères de performance selon FCFS

P_i	Temps de réponse $\texttt{Tr}P_i$	Temps d'attente $\texttt{Tat}P_i$
P_1	3	0
P_2	7	1
P_3	11	7
P_4	14	9
P_5	3	1
	TRM	TAM
	7.6	3.6

Table 3 – Critères de performance selon SJF

Pour le restant du travail :

— On dispose d'un fichier texte `"Processus.txt"` contenant la description d'un échantillon de processus (voir figure 3). Chaque ligne du fichier contient les informations, relatives à un processus donné, séparées par le caractère # et ayant le format suivant :

`"nomPi#taPi#dePi#etatPi\n"`

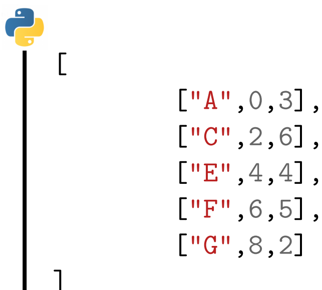
— On suppose qu'un processus ne peut apparaître qu'une seule fois.

```
A#0#3#Prêt
B#1#8#Bloqué
C#2#6#Prêt
D#3#4#Bloqué
E#4#4#Prêt
F#6#5#Prêt
G#8#2#Prêt
H#9#1#Bloqué
```

FIGURE 3 – Fichier "Processus.txt"

Dans le but de simuler un ordonnanceur de processus, on propose les structures ci-dessous décrites. Chaque processus prêt P_i est représenté par une liste contenant les 3 caractéristiques [NomP_i, TaP_i, DeP_i]. Les processus prêts sont représentés par une liste Lpp de listes de processus.

La liste Lpp des processus prêts de l'échantillon de la figure 3, est représentée comme suit :



```
[
        ["A",0,3],
        ["C",2,6],
        ["E",4,4],
        ["F",6,5],
        ["G",8,2]
]
```

Travail demandé

Dans la suite on propose d'implémenter les fonctions suivantes :

Q1. Écrire la fonction `LProcessusPrets`, qui à partir du fichier des processus intitulé `nomFichier`, retourne la liste `Lpp` des processus prêts.

Solution:

```
def LProcessusPrets(NomFichier):
    f = open(NomFichier, "r")
    L = f.readlines()
    f.close()
    Lpp=[]
    for i in L:
        L1=i.strip().split('#')
        if L1[-1]=="Prêt":
            Lpp.append([L1[0],int(L1[1]),int(L1[2])])
    return(Lpp)
```

Q2. Écrire la fonction `RechercherProcessus`, qui prend en entrée :

— la liste des processus prêts, `Lpp` ;

— le nom d'un processus `NomP` ;

retourne l'indice du processus de nom `NomP` s'il est dans la liste `Lpp` et `None` sinon.

Solution:

```
def Rechercherprocessus(Lpp,NomP):
    L=[p[0] for p in Lpp]
    if NomP in L:
        return L.index(NomP)
    return None
```

Q3. Ecrire la fonction `SupprimerProcessus`, qui prend en entrée :

— la liste des processus prêts `Lpp` ;

— le nom d'un processus `NomP` ;

supprime le processus ayant le nom `NomP` de la liste de processus s'il existe et déclenche une exception sinon.

Solution:

```python
def Supprimerprocessus(Lpp,NomP):
    try:
        ind=Rechercherprocessus(Lpp,NomP)
        assert ind!=None
        Lpp.pop(ind)
    except AssertionError:
        print("Ce nom du processus n existe pas")
```

Q4. Écrire la fonction `AppliquerFCFS` qui à partir de `Lpp`, applique l'algorithme d'ordonnancement **FCFS** et retourne comme résultat une liste à 3 éléments :

— une liste des noms des processus selon leurs ordres d'exécution ;

— le temps d'attente moyen (TAM) ;

— le temps de réponse moyen (TRM) ;

Pour la liste Lpp de l'échantillon décrit par la figure 3, l'appel de `AppliquerFCFS(Lpp)` donne le résultat suivant :

```
[ ["A","C","E","F","G"], 4.6, 8.6]
```

Solution:

```
def AppliquerFCFS(Lpp):
    Lpp=[[l[1],l[2],l[0]] for l in Lpp]
    Lpp.sort()
    n=len(Lpp)
    L=[]
    de=Tr=Ta=0
    for k in Lpp:
        de+=k[1]
        Tr+=de-(k[0]+k[1])
        Ta+=de-k[0]
        L.append(k[2])
    return [L,Ta/n,Tr/n]
```

Q5. Écrire la fonction `AppliquerSJF` qui à partir de `Lpp`, applique l'algorithme d'ordonnancement **SJF** et retourne comme résultat une liste à 3 éléments :

— une liste des noms des processus selon leurs ordres d'exécution ;

— le temps d'attente moyen (TAM) ;

— le temps de réponse moyen (TRM).

Pour la liste Lpp de l'échantillon décrit par la figure 3, l'appel de `AppliquerSJF(Lpp)` donne le résultat suivant :

```
[["A","C","G","E","F"], 3.6, 7.6]
```

Solution:

```python
def AppliquerSJF(Lpp):
    Lpp=[[l[1],l[2],l[0]] for l in Lpp]
    Lpp.sort()
    n=len(Lpp)
    L=[Lpp[0][2]]
    Tr=Ta=0
    de=Lpp[0][1]
    Tr+=de-(Lpp[0][0]+Lpp[0][1])
    Ta+=de-Lpp[0][0]
    del(Lpp[0])
    while True:
        if len(Lpp)==0:
            break
        Linf=[]
        for k in Lpp:
            if k[0]<=de:
                Linf.append([k[1],k[0],k[2]])
        Linf.sort()
        de+=Linf[0][0]
        Tr+=de-(Linf[0][1]+Linf[0][0])
        Ta+=de-Linf[0][1]
        L.append(Linf[0][2])
        for i in range(0,len(Lpp)):
            if Lpp[i][2]==Linf[0][2]:
                del(Lpp[i])
                break
    return [L,Ta/n,Tr/n]
```

Q6. Écrire les instructions Python permettant de :

— créer la liste des processus prêts `Lpp` ;

— créer le dictionnaire `dico` ayant comme clé le nom d'une stratégie d'ordonnancement, et comme valeur la liste résultat de l'appel de la stratégie en question appliquée à la liste Lpp.

— afficher le nom de la stratégie d'ordonnancement la plus performante sachant qu'en cas d'égalité des temps de réponse moyens, on examine les temps d'attente moyens.

Solution:

```python
Lpp1=LProcessusPrets("processus.txt")
print(Lpp1)
dico=dict()
dico.update({"FCFS":AppliquerFCFS(Lpp1)})
dico.update({"SJF":AppliquerSJF(Lpp1)})
print(dico)
L=[(j[1],j[2],i) for i,j in dico.items()]
L.sort()
print("la methode la plus rapide est ",L[0][2])
```

Partie II : Base de Données Relationnelle

Le ministère du transport fait appel à votre expertise pour manipuler une base de données relationnelle dans l'objectif de gérer en partie l'infrastructure ferroviaire à l'échelle nationale.

Règles de gestion et contraintes à considérer

— Dans notre contexte une gare ferroviaire de voyageurs est un lieu d'arrêt des trains pour la montée et la descente des passagers.

— Une gare peut jouer le rôle d'arrêt de départ et/ou arrêt transitoire et/ou arrêt de terminus.

— Dans une gare de départ ou de terminus un train est mis au repos pendant un certain nombre d'heures.

— Une ville peut contenir une ou plusieurs gares.

— Un voyage permet de desservir un ensemble de gares.

— Tout voyage par train doit nécessairement avoir une gare de départ, une gare de terminus et éventuellement un ensemble de gares d'arrêts transitoires.

Le schéma relationnel de la base de données ayant le nom `"ferroviaire.db"` est défini par les relations suivantes :

■ `Ville`(`idVille`, `nomVille`, `nbHabitants`)
La table `Ville`, stocke les différentes informations relatives aux villes, est caractérisée par les attributs suivants :

— `idVille` : identifiant pour distinguer les différentes villes, clé primaire, de type entier.

— `nomVille` : dénomination d'une ville, de type chaîne de caractères.

— `nbHabitants` : nombre d'habitants d'une ville, de type entier.

■ `Gare`(`idGare`, `nomGare`,#`idVille`)
La table `Gare`, stocke les informations des gares, est caractérisée par les attributs suivants :

— `idGare` : identifiant de la gare, clé primaire, de type chaîne de caractères.

— `nomGare` : dénomination distinctive attribuée à la gare de type chaîne de caractères.

— `idVille` : référence le numéro de la ville où se trouve la gare, de type entier, clé étrangère qui fait référence à la table Ville.

■ `Train`(<u>`idTrain`</u>, `dateMiseService`,`capacite`)
La table `Train`, stocke les informations relatives aux trains, est caractérisée par :

— `idTrain` : identifiant du train, clé primaire, de type chaîne de caractères.

— `dateMiseService` : date de mise en service du train, de type Date (`"AAAA-MM-JJ"`)

— `capacite` : capacité d'accueil maximale du train, de type entier.

■ `Voyage`(<u>`idVoyage`</u>, `dateHeureDepart`, `dateHeureArrivee`, #`idGareDepart`,# `idGareTerminus`, #`idTrain`)
La table `Voyage`, stocke les informations des voyages, est caractérisée par les attributs suivants :

— `idVoyage` : identifiant du voyage, clé primaire, de type chaîne de caractères.

— `dateHeureDepart` : date et heure de départ pour un voyage, assuré par un train, à partir de sa gare de départ, de type datetime (`"AAAA-MM-JJ HH:MM:SS"`).

— `dateHeureArrivee` : date et heure d'arrivée pour un voyage, assuré par un train, à sa gare terminus, de type datetime (`"AAAA-MM-JJ HH:MM:SS"`).

— `idGareDepart` : identifiant de la gare de départ de type chaîne de caractères, clé étrangère qui fait référence à la table `Gare`.

— `idGareTerminus` : identifiant de la gare terminus, clé étrangère qui fait référence à la table `Gare`, de type chaîne de caractères.

— `idTrain` : identifiant du train qui assure le voyage de type chaîne de caractères, clé étrangère qui fait référence à la table `Train`.

■ `Desservir`(<u>#`idVoyage`, #`idGareArret`</u>, `dateHeurePassage`)
La table `Desservir`, stocke l'ensemble des gares desservies pour un voyage, est caractérisée par les attributs suivants :

— `idVoyage` : identifiant d'un voyage, de type chaîne de caractères, clé étrangère qui fait référence à la table `Voyage`.

— `idGareArret` : identifiant d'une gare de type chaîne de caractères, clé étrangère qui fait référence à la table `Gare`.

— `dateHeurePassage` : date et heure de passage d'un train à la gare, de type datetime (`"AAAA-MM-JJ HH:MM:SS"`).

<u>**NB :**</u> `idVoyage` et `idGareArret` forment la clé primaire de la table `Desservir`.

Il est aussi à noter que la table `Desservir` ne stocke pas les gares de départ et de terminus pour un voyage donné.

Travail demandé

Algèbre relationnelle

Exprimer en algèbre relationnelle les requêtes permettant de :

Q7. Donner les noms des villes qui contiennent des gares desservies par le voyage d'identifiant "V23".

Solution:

$$\prod_{\texttt{nomVille}}\left(\texttt{Ville} \underset{\texttt{idVille}}{\bowtie} \texttt{Gare} \underset{\texttt{idGare}}{\bowtie} \sigma_{\texttt{idVoyage}=\texttt{"V23"}}(\texttt{Desservir})\right)$$

Q8. Quels sont les noms des villes dont le nombre d'habitants dépasse 100000 et qui ne contiennent aucune gare.

Solution:

$$\prod_{\texttt{nomVille}}\left(\sigma_{\texttt{nbHabitants}>100000}(\texttt{Ville}) \underset{\texttt{idVille}}{\bowtie} \left(\prod_{\texttt{idVille}}(\texttt{Ville}) - \prod_{\texttt{idVille}}(\texttt{Gare})\right)\right)$$

Q9. Donner toutes les informations relatives aux gares qui ne sont jusque-là desservies par aucun voyage. Il est à noter que la table Desservir n'inclus pas les gares de départ et de terminus d'un voyage.

Solution:

$$\texttt{Gare} \underset{\texttt{idGare}}{\bowtie} \prod_{\texttt{idGare}}(\texttt{Gare}) - \prod_{\texttt{idGare}}(\texttt{Desservir}) - \prod_{\texttt{idGareDepart}}(\texttt{Voyage}) - \prod_{\texttt{idGareTerminus}}(\texttt{Voyage})$$

SQL

Q10. Créer la table `Gare` en respectant les contraintes d'intégrités mentionnées ci-dessus. La table `Ville` est supposée déjà créée.

Solution:

```sql
/*Solution 1*/
CREATE TABLE Gare
(
idGare TEXT PRIMARY KEY,
nomGare TEXT,
idVille TEXT REFERENCES Ville
);

/*Solution 2*/
create table Gare(idGare text,nomGare  text,idVille int,
primary key(idGare),foreign key(idVille) references Ville(idVille))
```

Dans la suite on suppose que les tables de la base de données sont remplies en respectant les contraintes et les règles de gestion su citées.

Répondre aux questions suivantes par des requêtes SQL.

Q11. Supprimer les trains qui ont été mis en service avant 1993.

Solution:

```sql
delete from train where dateMiseService<"1993%"
```

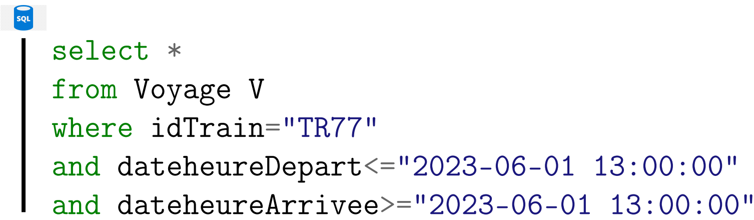
Q12. Donner le nombre des gares terminus.

Solution:

```sql
select count(distinct idGareTerminus)
from Voyage
```

Q13. Si le train d'identifiant "TR77" circule le premier juin 2023 à 08 :00 :00, déterminer toutes les informations relatives à son voyage.

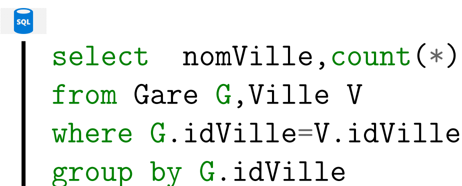
Solution:



```sql
select *
from Voyage V
where idTrain="TR77"
and dateheureDepart<="2023-06-01 13:00:00"
and dateheureArrivee>="2023-06-01 13:00:00"
```

Q14. Déterminer pour chaque nom de ville le nombre de gares qu'elle contient.

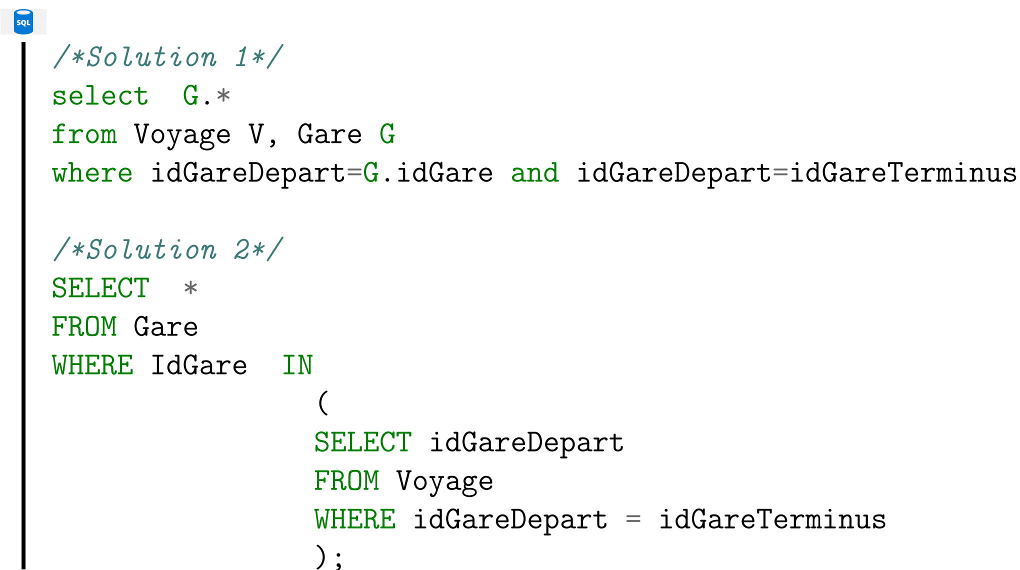
Solution:



```sql
select  nomVille,count(*)
from Gare G,Ville V
where G.idVille=V.idVille
group by G.idVille
```

Q15. Donner toutes les informations relatives aux gares qui sont à la fois gare de départ et gare terminus pour un même voyage.

Solution:



```sql
/*Solution 1*/
select  G.*
from Voyage V, Gare G
where idGareDepart=G.idGare and idGareDepart=idGareTerminus

/*Solution 2*/
SELECT  *
FROM Gare
WHERE IdGare  IN
                (
                SELECT idGareDepart
                FROM Voyage
                WHERE idGareDepart = idGareTerminus
                );
```

Q16. Donner toutes les informations relatives aux gares qui sont à la fois gare de départ et gare terminus.

Solution:

```sql
select *
from Gare
where idGare in (select  idGareDepart
                 from Voyage
                 intersect
                 select  idGareTerminus
                 from Voyage)
```

Q17. Déterminer pour chaque voyage le nom de la ville de départ et le nom de la ville d'arrivée ainsi que les dates et heures de départ et d'arrivée.

Solution:

```sql
select V1.nomVille DEPART,dateheureDepart,V2.nomVille ARRIVEE,dateheureArrivee
from Voyage V,Gare G1,Gare G2,Ville V1,ville V2
where  G1.idGare=idGareDepart
and G2.idGare=idGareTerminus
and G1.idville=V1.idVille
and G2.idville=V2.idVille
```

Q18. Déterminer pour le voyage "V23" la gare, date et heure de départ, la liste des gares desservies avec leurs dates et heures de passages et la gare, date et heure de terminus. On ne demande que les identifiants des gares.

Solution:

```sql
select idGareDepart,dateheureDepart,idGareArret,dateheurePassage,idGareTerminus,da
from Voyage V,Desservir D
where V.idVoyage=D.idVoyage and V.idVoyage like 'V23'
```

Q19. Donner toutes les informations des voyages qui desservent le maximum de gares.

Solution:

```sql
select V.*
from desservir D,Voyage V
where D.idVoyage=V.idVoyage
group by V.idVoyage
having count(*)=(select max(nb)
                 from (select count(*) as nb
                 from desservir
                 group by idVoyage))
```

SQLITE

Dans la suite les fonctions demandées doivent être écrites en Python en désignant par `cur` le curseur d'exécution de requêtes.

Q20. Écrire la fonction `trainV` qui prend comme paramètres :

- `cur` : le curseur d'exécution des requêtes ;
- `idV` : l'identifiant d'un voyage ;

retourne `idT` l'identifiant du train qui a assuré le voyage `idV`.

Solution:

```
def TrainV(cur,idV):
        cur.execute("select idTrain
                from Voyage V
                where idVoyage='{}'".format(idV))
        return(cur.fetchone()[0])
```

Q21. Écrire la fonction `circuitVoyage` qui prend comme paramètres :

- `cur` : le curseur d'exécution des requêtes ;
- `idV` : l'identifiant d'un voyage ;

retourne un dictionnaire dont :

- la clé est un `tuple` composé de l'identifiant du voyage `idV` et l'identifiant du train qui l'assure.
- la valeur est une liste de tuples où chaque tuple contient le nom de la ville, la gare, la date et heure dans un ordre chronologique croissant. L'extrait qui suit montre un exemple :

```
{('V01', 'TR10'):
        [('GABES', 'GARE-GAB', '2023-01-15 11:15:00'),
        ('SFAX', 'GARE-MAH', '2023-01-15 12:30:00'),
        ('SFAX', 'GARE-SF', '2023-01-15 13:15:00'),
        ('SOUSSE', 'GARE-SOU', '2023-01-15 15:15:00'),
        ('TUNIS', 'GARE-TUN', '2023-01-15 18:00:00')]
}
```

Solution:

```
def CircuitVoyage(cur,idV):
         cur.execute("select V1.nomVille,G1.nomgare,dateheureDepart,V2.nomVille,
                               G2.nomgare,dateheureArrivee
from Voyage ,Gare G1,Gare G2,Ville V1,ville V2
where  idVoyage='{}'
and G1.idGare=idGareDepart
and G2.idGare=idGareTerminus
and G1.idville=V1.idVille
and G2.idville=V2.idVille".format(idV))
         t=cur.fetchone()
         cur.execute("select  V.nomVille,nomgare,dateheurePassage
                      from desservir D,Gare G,Ville V
                      where  idVoyage='{}'
    and G.idGare=D.idGareArret
                      and G.idVille=V.idVille
    order by dateheurePassage asc".format(idV))
         L=cur.fetchall()
         L.insert(0,t[0:3])
         L.append(t[3:])
         return({(idV,TrainV(cur,idV)):L})
```

Q22. Écrire la fonction `circuitsVoyages` qui prend en paramètre `cur` et retourne la liste des dictionnaires des circuits de tous les voyages en faisant appel à la fonction `circuitVoyage` de la question **Q21**. L'extrait qui suit montre un exemple :

```
[
        {('V01', 'TR10'):
                [('GABES', 'GARE-GAB', '2023-01-15 11:15:00'),
                ('SFAX', 'GARE-MAH', '2023-01-15 12:30:00'),
                ('SFAX', 'GARE-SF', '2023-01-15 13:15:00'),
                ('SOUSSE', 'GARE-SOU', '2023-01-15 15:15:00'),
                ('TUNIS', 'GARE-TUN', '2023-01-15 18:00:00')
          ]},
```

```
        {('V02', 'TR12'):
                [('TUNIS', 'GARE-TUN', '2023-03-01 16:00:00'),
                ('SOUSSE', 'GARE-KAL', '2023-03-01 17:15:00'),
                ('SOUSSE', 'GARE-SOU', '2023-03-01 18:00:00')
         ]}
]
```

Solution:

```
def CircuitsVoyages(cur):
        cur.execute("select idVoyage from Voyage")
        V=[i[0] for i in cur.fetchall()]
        L=[]
        for idv in V:
                L.append(CircuitVoyage(cur,idv))
        return L
```