Université Paris Diderot – L2 Informatique 13 janvier 2017

Bases de Données (BD3) – Corrigé de l'examen (durée : 3 heures)

Documents autorisés : trois feuilles A4 recto-verso et personnelles. Les ordinateurs et les téléphones mobiles sont interdits.

Le barême est donné à titre indicatif.

Exercice 1 [**Requêtes** : *12 points*]
Soit la base de données **BANQUE** contenant les tables suivantes :

- **AGENCE (*Num_Agence, Nom, Ville, Actif)**
- **CLIENT (*Num_Client, Nom, Prenom, Ville)**
- **COMPTE (*Num_Compte, Num_Agence#, Num_Client#, Solde)**
- **EMPRUNT (*Num_Emprunt, Num_Agence#, Num_Client#, Montant)**

Les clefs primaires sont précédées d'une étoile (*) et les clefs étrangères sont suivies d'un astérisque (#).

1. Sans utiliser DISTINCT, donnez une requête équivalente en SQL :

```
SELECT DISTINCT Num_Client
FROM COMPTE
WHERE solde < 1000
OR solde > 100000 ;
```

Une première solution exploite le fait que l'opérateur d'union est un opérateur ensembliste et élimine donc les doublons :

```
(SELECT Num_Client
FROM COMPTE
WHERE solde < 1000)
UNION
(SELECT Num_Client
FROM COMPTE
WHERE solde > 100000) ;
```

Une seconde solution détourne le `GROUP BY` (utilisé d'ordinaire dans le cadre des requêtes d'agrégation) :

```
SELECT Num_Client
FROM COMPTE
WHERE solde < 1000
OR solde > 100000
GROUP BY Num_Client ;
```

Une troisième solution exploite le fait que Num_Client est clef primaire de client :

```
SELECT Num_Client
FROM CLIENT
WHERE Num_Client IN
(SELECT  Num_Client
FROM COMPTE
WHERE solde < 1000
OR solde > 100000) ;
```

J'ai également vu cette solution, un peu laborieuse mais correcte (intersect est un opérateur ensembliste et élimine donc les doublons) :

```
(SELECT Num_Client
FROM COMPTE
WHERE solde < 1000 OR solde > 100000)
INTERSECT
(SELECT Num_Client
FROM COMPTE
WHERE solde < 1000 OR solde > 100000) ;
```

Une erreur rencontrée fréquemment dans les copies :

```
SELECT Num_Client
FROM COMPTE
GROUP BY Num_Client
HAVING  solde < 1000
OR solde > 100000;
```

Un attribut figurant dans le `HAVING` doit figurer dans le `GROUP BY` ou bien être utilisé dans un agrégat. À un même numéro de client peuvent en effet être associés plusieurs soldes différents (un client peut avoir plusieurs comptes), il y a donc ambiguïté.

Attention : ce n'est pas parce que l'information concernant chaque compte d'un client n'apparait qu'une seule fois qu'il n'y a pas de doublons (autre erreur fréquente). Un client peut en effet avoir plusieurs comptes.

2. Ecrivez les requêtes SQL correspondant aux questions suivantes :
   (a) Les clients n'ayant pas de compte dans la même agence que Liliane Bettencourt. (Tableau résultat : **Num_Client**).
   Attention, la question est plus difficile qu'il n'y parait. Malgré les apparences la requête suivante par exemple ne convient pas :

   ```
   SELECT Num_Client
   FROM COMPTE
   WHERE Num_Agence NOT IN
   (SELECT  Num_Agence
   FROM COMPTE NATURAL JOIN CLIENT
   WHERE Client.Nom='Bettencourt'
   AND Client.Prenom='Liliane') ;
   ```

   Cette requête retourne les clients qui ont un compte dans une agence dans laquelle Liliane Bettencourt n'a pas de compte.
   En fait, il fallait écrire quelque chose de plus compliqué, par exemple :

```
SELECT Num_Client
FROM Client
EXCEPT
(Select Num_Client
FROM Compte
WHERE Num_Agence IN
(SELECT  Num_Agence
FROM COMPTE NATURAL JOIN CLIENT
WHERE Client.Nom='Bettencourt'
AND Client.Prenom='Liliane')) ;
```

ou bien

```
SELECT Num_Client
FROM Client
WHERE Num_Client NOT IN
(Select Num_Client
FROM Compte
WHERE Num_Agence IN
(SELECT  Num_Agence
FROM COMPTE NATURAL JOIN CLIENT
WHERE Client.Nom='Bettencourt'
AND Client.Prenom='Liliane')) ;
```

Il existe bien sûr plusieurs variantes en fonction de la manière dont sont faites les jointures, e.g., :

```
SELECT Num_Client
FROM Client
WHERE Num_Client NOT IN
(SELECT Num_Client
FROM Compte C1, Compte C2, Client C
WHERE C1.Num_Agence=C2.Num_Agence
AND C.Num_Client=C2.Num_Client
AND C.Nom='Bettencourt'
AND C.Prenom='Liliane')) ;
```

Il était également possible d'utiliser NOT EXISTS.

(b) Les agences ayant un actif plus élevé que toutes les agences de Saint-Ouen. (Tableau résultat : **Num_Agence**).

```
SELECT Num_Agence
FROM Agence
WHERE Actif > ALL
(SELECT Actif
FROM Agence
WHERE Ville='Saint Ouen') ;
```

Attention, utiliser `> ANY` au lieu de `> ALL` sélectionne les agences ayant un actif plus élevé qu'au moins une agence de Saint Ouen.
Une solution équivalente à celle utilisant `> ALL` :

```
SELECT Num_Agence
FROM Agence
WHERE Actif >
(SELECT Max(Actif)
FROM Agence
WHERE Ville='Saint Ouen') ;
```

Attention, la syntaxe `.... > MAX (SELECT Actif FROM...)` est incorrecte.

(c) Le solde moyen des comptes clients, pour chaque agence dont le solde moyen est supérieur à 10000. (Tableau résultat : **Num_Agence**, **Solde_Moyen**).

```
SELECT AVG(Solde) as Solde_Moyen
FROM Compte
GROUP BY Num_Agence
HAVING AVG(Solde) > 10000 ;
```

Attention, la requête suivante, qui a l'air innocente, fonctionne avec MySQL, mais pas avec Postgres (ERROR : column "Solde_Moyen" does not exist) :

```
SELECT AVG(Solde) as Solde_Moyen
FROM Compte
GROUP BY Num_Agence
HAVING Solde_Moyen > 10000 ;
```

J'ai évidemment compté tous les points, mais l'exemple permet de pointer que le standard de SQL est implémenté différemment d'un système à l'autre.
J'ai compté la moitié des points (ce qui était déjà trop gentil) pour la requête suivante, qui est incorrecte (jamais d'agrégat dans le WHERE) :

```
SELECT AVG(Solde) as Solde_Moyen
FROM Compte
GROUP BY Num_Agence
WHERE AVG(Solde) > 10000 ;
```

(d) Le nombre de clients de l'agence de nom "Paris-BNF" dont la ville n'est pas renseignée dans la relation CLIENT. (Tableau résultat : **Nombre**).

```
SELECT COUNT(DISTINCT num_client) as Nombre
FROM Client, Compte, Agence
WHERE Client.Num_client=Compte.Num_client
AND Agence.Num-Agence=Compte.Num-Agence
AND Agence.Nom='Paris-BNF'
AND Client.Ville IS NULL ;
```

Attention à la jointure naturelle sur `Client` et `Agence`. La jointure sera entre autres faite sur les deux attributs ville, ce qui forcera une contrainte supplémentaire sur les données (si un client possède un compte dans une agence domiciliée dans une ville différente de celle où il habite, alors ce compte n'apparaitra pas). Attention également à ne pas oublier le mot clef `Distinct` : nous ne voulons compter chaque client ayant un compte dans l'agence "Paris-BNF" qu'une seule fois, même s'il y possède plusieurs comptes. Une solution alternative sans `Distinct` :

```
SELECT SUM(num_client) as Nombre
FROM Client
WHERE  Ville IS NULL
AND Num_Client IN
(SELECT Num_Client
FROM Compte NATURAL JOIN Agence
WHERE Agence.Nom='Paris-BNF') ;
```

Attention également à ne pas écrire de condition du type `Agence.ville not in (Select Client.ville from Client)` (vu dans une copie). Au delà du fait que ce n'est pas ce qu'on veut (la ville pourrait ne pas être renseignée tout en n'appartenant pas à cette table), null not in (...) n'est ni vrai, ni faux, c'est indéterminé, donc la condition ne sera pas satisfaite si l'attribut est nul... Pour la même raison on n'écrit jamais `VILLE = NULL` (qui est de toutes façons syntaxiquement incorrect), mais `VILLE IS NULL`.

(e) Les clients ayant un compte dont le solde est supérieur à la somme totale de tous les actifs des agences de Saint-Ouen. (Tableau résultat : **Num_Client**).

```
SELECT Num_Client
FROM Compte
WHERE Solde >
(SELECT SUM(Actif)
FROM Agence
WHERE Ville='Saint-Ouen') ;
```

Si l'on veut éviter les doublons liés au fait qu'un même client pourrait avoir plusieurs comptes satisfaisant cette propriété, on peut utiliser le mot clef `DISTINCT` :

```
SELECT DISTINCT Num_Client
FROM Compte
WHERE Solde >
(SELECT SUM(Actif)
FROM Agence
WHERE Ville='Saint-Ouen') ;
```

(f) Les clients dont la somme du solde de tous les comptes est inférieure à l'actif de chaque agence. (Tableau résultat : **Num_Client**).

```
SELECT Num_Client
FROM Compte
GROUP BY Num_Client
HAVING SUM(Solde) <
(SELECT MIN(Actif)
FROM Agence) ;
```

Un exemple parmi beaucoup d'autres d'erreur rencontrée dans les copies :

```
SELECT Num_Client
FROM Compte, Agence
GROUP BY Num_Client
HAVING SUM(Solde) < Actif ;
```

Plusieurs problèmes ici : le produit cartésien sur `Compte` et `Agence` "multiplie les soldes" et pour chaque `Num_Client`, `SUM(Solde)` sera donc beaucoup plus élevé que prévu. Par ailleurs, `< Actif` n'est syntaxiquement pas correct, car `Actif` n'est qu'un nom d'attribut et ne définit pas le résultat d'une requête. L'objet de l'opérateur de comparaison n'est donc pas défini à droite.

(g) Les clients ayant un compte dans toutes les agences de Saint-Ouen. (Tableau résultat : **Num_Client**).
Une première solution sans utiliser l'agrégation, mais avec double imbrication et négations :

```
SELECT Num_Client
FROM Client
WHERE NOT EXISTS
(SELECT * FROM Agence
WHERE Ville='Saint-Ouen'
AND NOT EXISTS
(SELECT * FROM Compte
WHERE Client.Num_Client=Compte.Num_Client
AND Compte.Num_Agence=Agence.Num_Agence));
```

Afin d'écrire cette requête, on commence par traduire "Pour toute agence de Saint-Ouen, le client y a un compte" en "il n'existe pas d'agence de Saint-Ouen telle que le client n'y a pas de compte" (en calcul relationnel : $\forall x \varphi(x) \equiv \neg(\exists x \neg \varphi(x))$, avec φ une formule quelconque du calcul relationnel).
Une requête équivalente avec `NOT IN` au lieu du second `NOT EXISTS` :

```
SELECT Num_Client
FROM Client
WHERE NOT EXISTS
(SELECT * FROM Agence
WHERE Ville='Saint-Ouen'
AND Num_Client NOT IN
(SELECT Num_Client FROM Compte
WHERE Compte.Num_Agence=Agence.Num_Agence));
```

Une autre solution sans négation, mais avec de l'agrégation :

```
SELECT Num_Client
FROM Compte, Agence
WHERE Compte.Num_Agence=Agence.Num_Agence
AND Ville='Saint-Ouen'
GROUP BY Num_Client
HAVING COUNT(DISTINCT Num_Agence)=
(SELECT COUNT(DISTICT Num_Agence)
FROM Agence
WHERE Ville='Saint-Ouen');
```

Cette dernière solution est basée sur le fait que si un client a un compte dans chacune des agences de Saint-Ouen et qu'il y a exactement n agences à Saint-Ouen, alors ce client a un compte dans exactement n agences à Saint-Ouen.

3. Ecrivez en algèbre relationnelle les requêtes correspondant aux questions suivantes :
   (a) Les clients résidant à Paris, avec un compte dont le solde est supérieur à 10000 et un emprunt dont le montant est inférieur à 100000. (Tableau résultat : **Num_Client**.)
   J'ai compté cette solution comme juste :

$$\Pi_{Num\_Client}(\sigma_{ville=`Paris'}(Client) \bowtie \sigma_{solde>1000}(Compte) \bowtie \sigma_{montant<100000}(Emprunt))$$

   En revanche, cette réponse n'était pas complétement correcte. Il y avait une subtilité : ici pour qu'un client soit retenu dans la réponse, il faut que le solde et l'emprunt en question relèvent de la même agence, contrainte qui n'est pourtant pas impliquée par l'énoncé. Voici une solution générant potentiellement plus de réponses (tout client qui a un emprunt et un solde satisfaisant les réquisits, mais dans deux agences différentes, sera également retenu) :

$$\Pi_{Num\_Client}(\sigma_{ville=`Paris'}(Client)) \bowtie \sigma_{solde>1000}(Compte)) \\ \bowtie \Pi_{Num\_Client}(\sigma_{montant<100000}(Emprunt))$$

   Solution alternative vue dans une copie utilisant l'intersection :

$$\Pi_{Num\_Client}(\sigma_{ville=`Paris'}(Client)) \cap \Pi_{Num\_Client}(\sigma_{solde>1000}(Compte)) \\ \cap \Pi_{Num\_Client}(\sigma_{montant<100000}(Emprunt))$$

   Solution alternative vue dans une copie utilisant le produit cartésien :

$$\Pi_{Num\_Client}(\sigma_{Client.Num\_Client=Compte.Num\_Client \wedge Client.Num\_Client=Emprunt.Num\_Client} \\ (\sigma_{ville=`Paris'}(Client) \times \sigma_{solde>1000}(Compte) \times \sigma_{Montant<100000}(Emprunt)))$$

   Attention, la requête précédente est plus efficace que celle-ci (version originellement vue dans la copie en question) :

$$\Pi_{Num\_Client}(\sigma_{solde>1000 \wedge Montant<100000 \wedge Client.Num\_Client=Compte.Num\_Client} \\ \wedge Client.Num\_Client=Emprunt.Num\_Client(\sigma_{ville=`Paris'}(Client) \times Compte \times Emprunt))$$

   En effet les tailles des tables intermédiaires seront ici minimisées, les sélections ayant été faites le plus tôt possible (avant les coûteux produits cartésiens).
   (b) Les clients n'ayant contracté aucun emprunt. (Tableau résultat : **Num_Client**.)

$$\Pi_{Num\_Client}(Client) - \Pi_{Num\_Client}(Emprunt)$$

   Attention, la requête suivante ne convient pas :

$$\Pi_{Num\_Client}(\sigma_{montant=0}(Emprunt))$$

Elle retourne en effet tous les clients pour lesquels il existe dans la table `Emprunt` un tuple associé pour lequel la valeur de `montant` est égal à 0. Mais cela ne nous dit rien au sujet de l'absence ou de la présence dans la table d'autres tuples associés à ce même client. Par ailleurs, ce n'est pas parce que le montant est égal à 0 quelque part, qu'un emprunt n'a jamais été contracté, au contraire...

(c) Les clients ayant un compte dans la même agence que Liliane Bettencourt. (Tableau résultat : **Num_Client**.)

$$\Pi_{Num\_Client}(\Pi_{Num\_Agence}(\sigma_{Prenom='Liliane' \wedge Nom='Bettencourt'}(Client) \bowtie Compte) \bowtie Compte)$$

Attention à la proposition suivante, rencontrée fréquemment dans les copies :

$$\Pi_{Num\_Client}((\sigma_{Prenom='Liliane' \wedge Nom='Bettencourt'}(Client) \bowtie Compte) \bowtie Compte)$$

Ici la jointure sera faite entre autres sur Num_Client. Seules les infos concernant Liliane Bettencourt seront donc retenues, ce qui n'a pas de sens, puisque quelles que soient les données, aucun autre numéro de client que celui de Liliane Bettencourt figurera dans la réponse...

J'ai vu également des solutions très bizarres avec des clauses du type $\sigma_{Num\_Agence=E}$ avec E une requête SQL. Le signe = n'a pas de sens ici, parce qu'on parle d'ensemble. L'appartenance ensembliste pourrait théoriquement en avoir un (situation rencontré dans une copie), mais ce n'est pas quelque chose qui est admis dans la syntaxe standard de l'algèbre relationnelle.

J'ai en revanche vu une solution intéressante utilisant l'intersection.

Exercice 2 [**Normalisation** : *6 points*]

On s'intéresse ici à la conception du schéma d'une base de données d'une agence immobilière. On retient l'ensemble d'attributs {`Num_Client`, `Nom_Client`, `Num_App`, `Adr_App`, `DateD_Loc`, `DateF_Loc`, `Montant`, `Num_Prop`, `Nom_Prop`}, ainsi que les dépendances fonctionnelles suivantes :

- `Num_Client -> Nom_Client`
- `Num_Client, Num_App -> DateD_Loc, DateF_Loc`
- `Num_App -> Adr_App, Montant, Num_Prop, Nom_Prop`
- `Num_Prop -> Nom_Prop`

1. Supposez que tous les attributs sont groupés dans une seule relation. Proposez une clef candidate déterminant tous les attributs de cette relation. Justifiez votre réponse.

   On peut commencer par remarquer que `Num_Client` et `Num_App` ne se trouvent en partie droite d'aucune dépendance fonctionnelle (DF). Toute clef candidate devra donc les contenir. On vérifie maintenant qu'il s'agit bien d'une clef pour la relation, au moyen de l'algorithme de clôture (cf. transparent 12 du cours sur la normalisation). On ajoute d'abord à notre ensemble de départ l'attribut `Nom_Client` grâce à la première DF. La seconde DF nous permet ensuite d'ajouter `DateD_Loc, DateF_Loc`. Enfin la troisième nous permet d'obtenir les derniers attributs de la relation `Adr_App, Montant, Num_Prop, Nom_Prop`. On en déduit que

`(Num_Client, Num_App)` détermine tous les attributs de la relation et est donc bien une clef. Reste à prouver qu'elle est minimale et donc bien clef candidate. Il suffit pour cela de remarquer que ni la clôture de `Num_Client`, ni celle de `Num_App` ne contiennent tous les attributs de la relation. On peut alors réutiliser l'algorithme de clôture. Dans le premier cas celui-ci ne permet d'obtenir que `Nom_Client` et dans le second uniquement `Adr_App, Montant, Num_Prop, Nom_Prop`. Ni `Num_Client` ni `Num_App` pris isolément ne sont donc des clefs pour la relation et `(Num_Client, Num_App)` est donc bien une clef candidate.

2. Proposez une instance très simple (5 tuples) sur ce schéma et étant donné cette instance, identifiez au moins une anomalie.

   Voici par exemple une instance possible :

Num_Client	Nom_Client	Num_App	Adr_App	DateD_Loc	DateF_Loc	Montant	Num_Prop	Nom_Prop
CR76	Jean DUPONT	PG4	12, rue de la Gare	01.07.93	31.08.95	3500	CX40	Jeanne MOULIN
CR76	Jean DUPONT	PG16	7, av. de la République	01.09.95	01.09.96	4500	CX93	Alain MULLER
CR56	Claire SERRON	PG4	12, rue de la Gare	01.09.92	10.06.93	3500	CX40	Jeanne MOULIN
CR56	Claire SERRON	PG36	3, Grande Rue	10.10.93	01.12.94	3800	CX93	Alain MULLER
CR56	Claire SERRON	PG16	7, av. République	01.01.95	10.08.95	4500	CX93	Alain MULLE

   Attention, une anomalie est un terme spécifique dans ce contexte, il ne s'agit pas de tout problème de modélisation au sens large. On parle d'anomalies de suppression, insertion, modification. Celles-ci peuvent entrainer redondance, valeurs de données nulles ou incohérences.

   En voici un exemple. Supposons qu'on veuille modifier ou corriger l'adresse de l'appartement de numéro PG4, apparaissant deux fois dans la table (pour deux locataires différents à différentes périodes), il nous faudra modifier deux tuples différents. Au delà du fait que l'information concernant l'adresse de l'appartement apparait de façon redondante dans la table, ce qui n'est déjà pas satisfaisant en soi, nous faisons face à un autre problème. Si l'un des deux tuples concernés n'est pas modifié, il y aura deux adresses différentes associées à un même appartement. On parle ici d'une anomalie de modification, potentiellement génératrice d'incohérence.

   Imaginons maintenant que Jeanne Moulin souhaite proposer un nouvel appartement à la location. Si l'on souhaite enregistrer cet appartement dans la base de données avant qu'il n'ait trouvé de locataire (ce qui semble assez normal), de nombreuses valeurs d'attribut seront nulles pour le tuple correspondant. On parle dans ce cas d'une anomalie d'insertion.

3. Normalisez ce schéma. Le schéma obtenu est-il en FNBC, 3FN ?

   La forme normale de Boyce Codd (FNBC) étant la meilleure forme normale (bien qu'elle ne puisse pas être obtenue dans tous les cas), on commence par appliquer l'algorithme de décomposition pour la FNBC. On sélectionne (par exemple) la première DF et on remarque que bien que `Num_Client` détermine `Nom_Client`, cet attribut ne détermine pas la valeur des autres attributs de la relation (ie, ce n'est pas une clef). La relation n'est donc pas en FNBC. En suivant la procédure décrite en cours (cf. transparent 45 du cours sur la normalisation) on obtient les trois ensembles X={Num_Client}, Y={Nom_Client}, Z= {Num_App, Adr_App, DateD_Loc, DateF_Loc, Montant, Num_Prop, Nom_Prop}. On décompose alors notre schéma en deux relations, l'une contenant {Num_Client, Nom_Client} et l'autre contenant tous les attributs sauf {Nom_Client}. On peut ensuite appliquer le même raisonnement en considérant la dernière DF. On conserve notre premier ensemble tel quel et on

décompose notre deuxième ensemble en {Num_Prop, Nom_Prop} et {Num_Client, Num_App, Adr_App, DateD_Loc, DateF_Loc, Montant, Num_Prop}. Les deux premiers ensembles sont non problématiques. On considère maintenant la troisième DF et on décompose le 3ème ensemble en {Num_App, Adr_App, Montant, Num_Prop} et {Num_Client, Num_App, DateD_Loc, DateF_Loc}. Toutes les relations obtenues sont maintenant en FNBC et on obtient le schéma suivant :

- {Num_Client, Nom_Client}
- {Num_Prop, Nom_Prop}
- {Num_App, Adr_App, Montant, Num_Prop}
- {Num_Client, Num_App, DateD_Loc, DateF_Loc}

Le schéma est en FNBC (et donc également en troisième forme normale 3FN) ; mais surtout, il est bien sans perte de dépendance et nous n'avons donc pas besoin d'appliquer l'algorithme de mise en troisième forme normale. (S'il y avait eu perte de dépendance, alors on aurait toujours pu appliquer l'algorithme de mise en 3FN, dont on est sûr qu'il mènera à un résultat SPI et SPD.)

Exercice 3 [Information Incomplète : *3 points*]

Soit la table :

ACTIVITE	Intitulé	Titulaire	Heures_Cours	Heures_Tp
	Java	PHE	30	15
	Labo prog	PHE		45
	BD	JLH	30	
	Projet qualité	NHA		30
	Modélisation	JLH	20	10
	Conception	VEN	45	
	Mise en oeuvre	VEN	60	
	Labo gestion	NHA		45

1. Donnez le résultat des requêtes suivantes :

   (a)
   ```
   SELECT Titulaire, SUM(Heures_Cours) + SUM(Heures_Tp) as Charge
   from ACTIVITE
   group by Titulaire;
   ```

   (b)
   ```
   SELECT Titulaire, SUM(Heures_Cours + Heures_Tp) as Charge
   from ACTIVITE
   group by Titulaire ;
   ```

   La réponse se trouvait sur les transparents 19 et 20 du cours concernant l'information incomplète dans SQL. Le résultat de la première requête était :

REPONSE	Titulaire	Charge
	JLH	60
	NHA	
	PHE	90
	VEN	

   Le premier et le troisième tuple du résultat auraient été identiques si les valeurs nulles de la table avaient été remplacées par 0. En revanche le calcul aurait été différent pour

le second et le quatrième tuple. Au lieu d'obtenir des valeurs nulles, nous aurions obtenu 75 et 105.

Nous détaillons le calcul menant à une valeur nulle uniquement pour le second tuple (l'autre calcul est laissé au lecteur comme exercice de révision). Toutes les valeurs de H_Cours concernant NHA sont nulles, la somme de ces valeurs est donc nulle (cf. transparent 28 du cours). La somme des H_TP en revanche n'est pas nulle, elle est égale à 75. Mais null+75=null (cf. transparent 27 du cours). Le résultat de la requête pour NHA est donc nul.

Le résultat de la seconde requête était :

REPONSE	Titulaire	Charge
	JLH	40
	NHA	
	PHE	45
	VEN	

Nous détaillons ici uniquement le calcul associé au premier tuple. Le calcul s'effectue ici d'abord tuple par tuple dans la table `ACTIVITE`. Les troisièmes et cinquièmes tuples sont les seuls à concerner JLH. Dans le premier cas on calcule 30+NUL=NUL, et dans le second 20+10=30. On effectue ensuite la somme de null et de 30 au moyen de l'opérateur d'agrégation `SUM` de SQL et le résultat est 30 (cf. transparent 28 : "Ces fonctions travaillent sur des ensembles dont les éventuelles valeurs null sont ignorées").

Ici aussi nous demandons au lecteur de refaire les autres calculs lui même afin de bien s'assurer qu'il a compris la logique des nuls dans SQL.

2. Que pensez-vous de ces résultats ? Comment procéderiez-vous s'il vous fallait calculer le nombre d'heures total de chaque enseignant ?

   Ces résultats sont bien évidemment problématiques. J'avais mentionnée la meilleure façon de faire face à ce problème en cours. On pouvait modifier les requêtes par exemple de la façon suivante (cf. transparents 21 et 22 du cours + transparent 13 discutant comment réaliser l'union des différentes parties d'un ensemble contenant des nulls) :

```
WITH heures AS (select TITULAIRE, sum(H_COURS) as CHARGE
from ACTIVITE
where H_COURS IS NOT NULL
group by TITULAIRE
UNION ALL
SELECT TITULAIRE, sum(H_TP) as CHARGE
from ACTIVITE
where H_TP IS NOT NULL
group by TITULAIRE
)
SELECT titulaire, SUM(CHARGE) as charge
FROM heures
GROUP BY titulaire ;
```

   ou bien :

```
SELECT titulaire, SUM(CHARGE) as charge
FROM (select TITULAIRE, sum(H_COURS) as CHARGE
```

```
from ACTIVITE
where H_COURS IS NOT NULL
group by TITULAIRE
UNION ALL
SELECT TITULAIRE, sum(H_TP) as CHARGE
from ACTIVITE
where H_TP IS NOT NULL
group by TITULAIRE
) as heures
GROUP BY titulaire ;
```

De cette façon, toutes les valeurs nulles sont exclues du calcul et l'on parvient au résultat attendu sans encombre, qui est :

REPONSE	Titulaire	Charge
	JLH	60
	NHA	75
	PHE	90
	VEN	105

Attention, `<> NULL` n'est pas correct (déjà souligné ailleurs dans ce corrigé). Je ne laisserai plus passer cette erreur dans les prochains examens.

Certains étudiants ont proposé de modifier la conception du schéma de la base de données en introduisant lors de la création des tables soit des valeurs par défaut à 0, soit des contraintes de non nullité sur les attributs problématiques. Pourquoi pas. En revanche, il s'agit là d'une solution radicale qu'il n'est pas toujours aisé de mettre en oeuvre une fois que les tables ont été créées. Mais j'ai apprécié le fait que ces étudiants avaient un peu réfléchi et j'ai donné tous les points correspondant à la question. Je serai cependant moins généreuse si la situation se représente.