Livre 3 : Programmer une base de données

4.0 01/09/2021

Bloc 2 - Conception et adaptation d'une base de données

Table des matières

I - Bases de données - Les procédures stockées et fonctions 3

1. Les procédures stockées, les fonctions 3
   - 1.1. Présentation des procédures stockées 3
   - 1.2. Exemples en SQL seul (ou SQL pur) 4
   - 1.3. Exemples en PL/pgSQL ou PL/SQL (Oracle) 6
   - 1.4. Les transactions 9

2. Exercice : Cas : "Au père tranquille" 10

II - Base de données - Les curseurs 11

1. Les curseurs 11
   - 1.1. Déclaration d'un curseur 11
   - 1.2. Ouverture d'un curseur 12
   - 1.3. Utilisation d'un curseur 13
   - 1.4. Fermeture d'un curseur 13
   - 1.5. Principales structures de contrôle 13
   - 1.6. Exemple d'utilisation d'un curseur 14

2. Exercice : Cas : Au père tranquille 15

3. Exercice : Cas : Meubléa 16

4. Exercice : Cas : Sondages 17

III - Bases de données - Les déclencheurs 19

1. Les déclencheurs ou triggers 19
   - 1.1. Qu'est ce qu'un déclencheur ? 19
   - 1.2. Comment faire un trigger ? 20

2. Exercice : Cas : Sondages 22

IV - Bases de données - Répartition des données 24

1. Les bases de données réparties 24
   - 1.1. Le Client / Serveur 24
   - 1.2. Données distribuées 31
   - 1.3. Exemple de synthèse 36

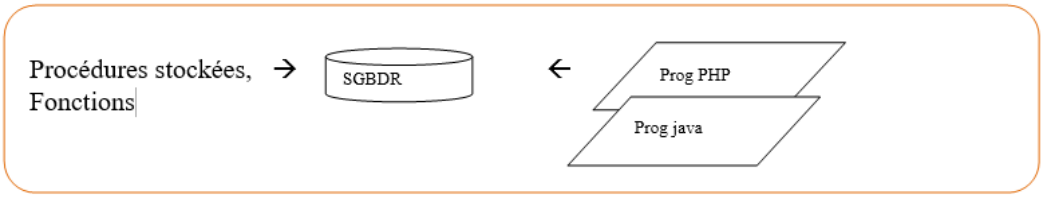
2. Exercice : Vas : Kimassur 37

Références 39

Bases de données - Les procédures stockées et fonctions

- Les procédures stockées, les fonctions
  - Présentation des procédures stockées
    - Intérêts
    - Structure générale d'une procédure, d'une fonction
  - Exemples en SQL seul (ou SQL pur)
  - Exemples en PL/pgSQL ou PL/SQL (Oracle)
  - Les transactions
- Exercice : Cas : "Au père tranquille"

1. Les procédures stockées, les fonctions



Plutôt que d'effectuer toutes les requêtes SQL depuis l'application cliente, il est préférable de déporter ces traitements sur le SGBD lui-même, c'est le rôle des procédures stockées.

PostgreSQL permet de programmer les procédures stockées dans différents langages, dont SQL pur.

1.1. Présentation des procédures stockées

a) Intérêts

Les procédures stockées ont plusieurs intérêts :

1. Le premier est de simplifier les codes interagissant avec la base de données. On aura un morceau de code fait une fois pour toute qui va s'occuper d'effectuer une opération qui a une signification fonctionnelle, et non pas seulement une signification physique dans la base de données.

Exemple :

Par exemple, si l'on désire effacer un élève de la base de données, cela peut nécessiter d'autres opérations qu'un simple "**DELETE FROM eleves**", on peut avoir besoin de modifier l'effectif de la classe, etc. On peut ainsi regrouper toutes ces opérations au sein d'une seule et unique procédure **supprimer_eleve()**.

2. Le second, qui découle directement du premier, est de factoriser le code. C'est à dire d'avoir un code unique, plutôt que plusieurs exemplaires du même code éparpillés au sein d'une ou plusieurs applications.

3. En terme de performances réseau, les procédures stockées apportent :

- une rapidité accrue puisque la requête est précompilée : les analyses lexicographiques et syntaxiques du code de la requête ont été faites une fois pour toute.
- moins d'échanges entre le serveur et le SGBDR dans le cas où la procédure stockée effectue plusieurs opérations. Avec un simple appel de procédure depuis l'application, toutes les opérations vont s'effectuer au sein même du SGBDR.

4. Et enfin, dans un soucis de sécurité, on peut permettre aux utilisateurs (et applications) d'accéder à des procédure stockées et non plus directement aux tables du SGBDR.

b) Structure générale d'une procédure, d'une fonction

¶ *Syntaxe* :

```
CREATE FUNCTION nom_fonction(parametres) RETURNS type AS

'code de la fonction'

LANGAGE 'nom du langage' ;
```

⚠ *Attention* :

Il s'agit de la création pure et simple d'une fonction ou d'une procédure stockée. Elle n'est pas exécutée. Pour ce faire, il convient de la lancer :

```
select nom_fonction(parametres) ;
```

⊕ *Complément* : PL/pgSQL et PostGreSQL

Lors de la création d'une fonction ou d'une procédure stockée, vous risquez d'être confronté au message suivant :

```
ERREUR:  le langage « plpgsql » n'existe pas
HINT:  Utiliser CREATE LANGUAGE pour charger le langage dans la base de données.
********** Erreur **********

ERREUR: le langage « plpgsql » n'existe pas
État SQL :42704
Astuce : Utiliser CREATE LANGUAGE pour charger le langage dans la base de données.

```

C'est normal bien que curieux. En effet, le langage est intégré par défaut mais non chargé. Pour le faire il suffit de créer la demande suivante :

```
CREATE LANGUAGE PLPGSQL ;
```

Il suffit de le faire UNE fois dans CHAQUE base de données. Mais seul un super-utilisateur peut le faire (typiquement postgres).

Bonne nouvelle ! ! ! Ce souci est corrigé depuis la version 9.0 de PostGreSQL. Ainsi, le langage PL/pgSQL est automatiquement chargé et utilisable par tous les utilisateurs de PostGreSQL.

1.2. Exemples en SQL seul (ou SQL pur)

Voici tout d'abord quelques exemples de fonctions en langage de requête (*SQL*)[*http://docs.postgresql.fr/9.4/xfunc-sql.html*]* :

⚠ *Attention* :

Elles ne peuvent rendre qu'un seul résultat !

1. Une fonction "cplus1()" en sql qui retourne "le nombre de tuples +1" de la table "salarié"

```
create function cplus1() returns int8  as
'select count(*) +1  as compte from salarie;'
language 'sql';

```

2. Une fonction en sql "nbs_no()" qui prend en argument un numéro de salarié et qui retourne pour un salarié donné, le nombre de personnes qu'il a sous ses ordres.

Le numéro est passé en paramètre du salarié est passé en paramètre.

```
create function nbs_no(numeric) returns int8 as
'select count(*) from salarie where numChef = $1;'
language 'sql';

```

3. Une fonction sbynum() en sql qui supprime une équipe par son numéro.

```
create function sbynum(int4) returns int8 as
'delete from equipe where equnum = $1;
 select 1;'
language 'sql';

```

4. Une fonction ag() en langage sql qui augmente de x le salaire des salariés qui ont pour chef y

x et y sont passés en paramètres.

Remarque **:**

La table SALARIE a la structure suivante : SALARIE (numS, salaire, #numChef).

Clé primaire : numS

Clé étrangère : #numChef en référence à numChef de la table CHEF

```
create function ag (numeric, numeric) returns int4 as
'update salarie set salaire = salaire + $1 where numChef = $2;
select 1;
language 'sql';

```

```
SELECT * FROM SALARIE;
```

donne :

```
nums  salaire  numchef
1     110.00  1
2     110.00  1
3     120.00  2
4     110.00  1
5     130.00  3
6     130.00  3
7     130.00  3
8     110.00  1
9     110.00  1
10    130.00  3
11    110.00  1

11 ligne(s)
```

L'instruction "select ag(50,3)" augmente de 50 tous les salariés qui ont pour chef le salarié qui a le numéro 3.

```
select ag(50,3);
```

augmente tous les salaires de 50 unités pour les salariés ayant pour chef le numéro 3. pour s'en convaincre :

```
SELECT * FROM SALARIE ;
```

```
ag
1

1 ligne(s)
```

Cela est peu convainquant. Une ligne a été modifiée ?

```
SELECT * FROM SALARIE ;
```

```
nums  salaire  numchef
1     110.00  1
2     110.00  1
3     120.00  2
4     110.00  1
8     110.00  1
9     110.00  1
11    110.00  1
5     180.00  3
6     180.00  3
7     180.00  3
10    180.00  3

11 ligne(s)
```

Les salariés 5, 6, 7 et 10 ont bien vu leur salaire augmenter... C'est plus convainquant !

⚠ *Attention* : A quoi sert select 1 ?

Le corps d'une fonction SQL doit être constitué d'une liste d'une ou de plusieurs instructions SQL séparées par des points-virgule.

Un point-virgule après la dernière instruction est optionnel.

Sauf si la fonction déclare renvoyer **void**, la dernière instruction doit être un SELECT ou un INSERT, UPDATE ou un DELETE qui a une clause RETURNING.

```
Erreur SQL :

ERREUR:  le type de retour ne correspond pas à la fonction déclarant renvoyer integer
DETAIL:  L'instruction finale de la fonction doit être un SELECT ou un
 INSERT/UPDATE/DELETE RETURNING.
CONTEXT:  Fonction SQL « ag »

Dans l'instruction :
create function ag (numeric, numeric) returns int4 as
'update salarie set salaire = salaire + $1 where numChef = $2; '
language 'sql';
```

✎ *Remarque* :

Cette fonction est tout à fait valable. Car elle renvoie un **void** :

```
CREATE FUNCTION nettoie_emp() RETURNS void AS
' DELETE FROM salarie WHERE salaire < 0; '
LANGUAGE SQL;
```

```
SELECT nettoie_emp();
```

```
nettoie_emp
-----------

  (1 row)
```

1.3. Exemples en PL/pgSQL ou PL/SQL (Oracle)

PL/pgSQL qui est un langage destiné à écrire des procédures stockées pour le système de base de données postgreSQL.

Ce langage est très proche d'autres langages utilisés pour différents SGBDR tel que pl/SQL pour Oracle.

Voici quelques exemples de fonctions en langage de procédures SQL (*PL/pgSQL*[http://docs.postgresql.fr/9.4/plpgsql.html]) :

***Exemple* : 1. Base de l'exemple : une table toute simple avec quelques données**

```
create table comptes(no integer primary key, solde decimal(10,2));
...
select * from comptes;

```

```
no | solde
---+---------
 1 | 100.00
 2 | 1000.00
 3 | 1500.00
 4 | 10.00

```

2. Création d'une fonction pour effectuer un virement d'un compte vers un autre

Remarque : Pour plus de lisibilité et de facilité, il est préférable de taper le code SQL de création de la procédure dans un fichier texte (virement.sql), puis d'exécuter ce script depuis psql avec la commande "**\i virement.sql"**, c'est beaucoup plus pratique.

```
create or replace function virement(integer,integer,decimal) returns integer AS '
DECLARE crediteur ALIAS FOR $1;
DECLARE debiteur ALIAS FOR $2;
DECLARE montant ALIAS FOR $3;
BEGIN
      UPDATE comptes SET solde = solde + montant WHERE no = crediteur;
      UPDATE comptes SET solde = solde - montant WHERE no = debiteur;
      return 0;
END;
'           LANGUAGE 'plpgsql';

```

Dans notre cas, notre fonction s'appelle **virement**, admet 2 paramètres numériques entiers (les numéros de comptes) et 1 décimal (le montant du virement).

Elle retournera une valeur entière (sans signification dans ce cas), et elle est codée en PL/pgSQL.

Ensuite, nous donnons des noms à nos paramètres, ce qui sera plus explicite que "1" & "2", etc.

Après, commence le bloc d'instruction PL/pgSQL proprement dit, où nous effectuons nos deux UPDATE successifs afin d'effectuer le virement :

3. Exécution de la fonction (avec vérification)

```
select virement(1,2,100);
```

Cela signifie : virement de 100 euros du compte 2 au compte 1.

```
virement
----------
   0

1 ligne(s)
```

```
select * from comptes;
```

```
no | solde
---+---------
 3 | 1500.00
 4 | 10.00
 1 | 200.00
 2 | 900.00

4 ligne(s)
```

4. Nouvelle version de la fonction

Maintenant, notre procédure devrait vérifier que le compte est suffisamment approvisionné avant de faire le virement. En voici la nouvelle version :

```
drop function virement(integer, integer, decimal);
create function virement(integer,integer,decimal) returns integer AS '
DECLARE crediteur ALIAS FOR $1;
DECLARE debiteur ALIAS FOR $2;
DECLARE montant ALIAS FOR $3;
DECLARE ancien_solde DECIMAL;

BEGIN
      SELECT solde INTO ancien_solde FROM comptes WHERE no = debiteur;
      IF (ancien_solde > montant) THEN
            UPDATE comptes SET solde = solde + montant WHERE no = crediteur;
            UPDATE comptes SET solde = solde - montant WHERE no = debiteur;
            return 0;
      END IF;
      return -1;
END;
'       LANGUAGE 'plpgsql';

```

Nous avons ici déclaré une nouvelle variable ancien_solde à laquelle nous avons affecté une valeur avec un ordre SELECT INTO. Dans ce cas, la clause INTO désigne une variable, et non pas une table à créer.

5. Ultime version de la fonction

Enfin pour terminer, voici une dernière version de la procédure qui va vérifier l'existence des comptes :

```
drop function virement(integer, integer, decimal);
create function virement(integer,integer,decimal) returns integer AS '
DECLARE crediteur ALIAS FOR $1;
DECLARE debiteur ALIAS FOR $2;
DECLARE montant ALIAS FOR $3;
DECLARE ancien_solde DECIMAL;
DECLARE test INTEGER;

BEGIN
      SELECT solde INTO ancien_solde FROM comptes WHERE no = debiteur;

      IF NOT FOUND THEN  [1]
            RAISE EXCEPTION ''Compte no % Inconnu'',debiteur;
      END IF;

      IF (ancien_solde > montant) THEN
            SELECT count(1) into test FROM comptes WHERE no = crediteur; [2]
            IF (test <> 1) THEN
                  RAISE EXCEPTION ''Compte no % Inconnu'',crediteur; [3]
            END IF;
            UPDATE comptes SET solde = solde + montant WHERE no = crediteur;
            UPDATE comptes SET solde = solde - montant WHERE no = debiteur;
            return 0;
      END IF;

      return 1;
END;
'     LANGUAGE 'plpgsql';

```

Ce code mérite quelques explications :

[1] : Le IF NOT FOUND sera vérifié si l'instruction le précédant n'a trouvé aucune ligne à manipuler.

[2] : Dans le même ordre d'idée, le SELECT count(1) INTO test nous sert à vérifier qu'il existe bien un (et un seul) compte ayant ce numéro.

Cette "bidouille" est inutile depuis la version 7.1 de postgreSQL, puisque désormais avec l'instruction GET DIAGNOSTICS test = ROW_COUNT on peut connaître directement le nombre de lignes manipulées par la dernière instruction SQL.

[3] : L'instruction RAISE EXCEPTION permet d'émettre des messages d'erreur. Notez que les chaînes de caractères sont encadrées de doubles apostrophes ".

En effet, le code de notre procédure étant elle même une chaîne de caractères (CREATE FUNCTION nom() RETURNS type AS 'code'...), il est nécessaire d'échapper les délimiteurs de chaînes au sein de la procédure.

Nous aurions également pu utiliser \' comme échappement. Il est intéressant de savoir également que le RAISE EXCEPTION à pour conséquence d'interrompre immédiatement le code en cours d'exécution.

Enfin, sachez également qu'il est possible d'appeler une procédure stockée au sein d'un ordre select. Si par exemple vous désirez que tous les comptes fassent un virement de 1 euro sur le compte numéro 1 :

```
select * from comptes;
```

```
no | solde
----+---------
 3 | 1500.00
 4 |   10.00
 2 |  797.00
 1 |  296.00

4 ligne(s)
```

```
select virement(1,no,1) FROM COMPTES WHERE no <>1;
```

```
virement
----------
000

3 ligne(s)
```

```
select * from comptes;
```

```
no | solde
----+---------
 3 | 1499.00
 4 |    9.00
 2 |  796.00
 1 |  299.00

4 ligne(s)
```

1.4. Les transactions

Une transaction est un ensemble cohérent de modifications faites sur les données afin d'en assurer l'intégrité. Une transaction est soit entièrement annulée, soit entièrement validée.

Nous retiendrons 3 instructions pour manipuler des transactions :

- BEGIN pour indiquer le début d'une transaction,
- COMMIT pour marquer la fin d'une transaction et la valider,
- ROLLBACK pour marquer la fin d'une transaction et l'annuler.

Tant qu'il n'y a pas eu COMMIT, seul l'utilisateur courant voit ses mises à jour. L'instruction COMMIT a pour effet d'intégrer de façon permanente les changements dans la base de données.

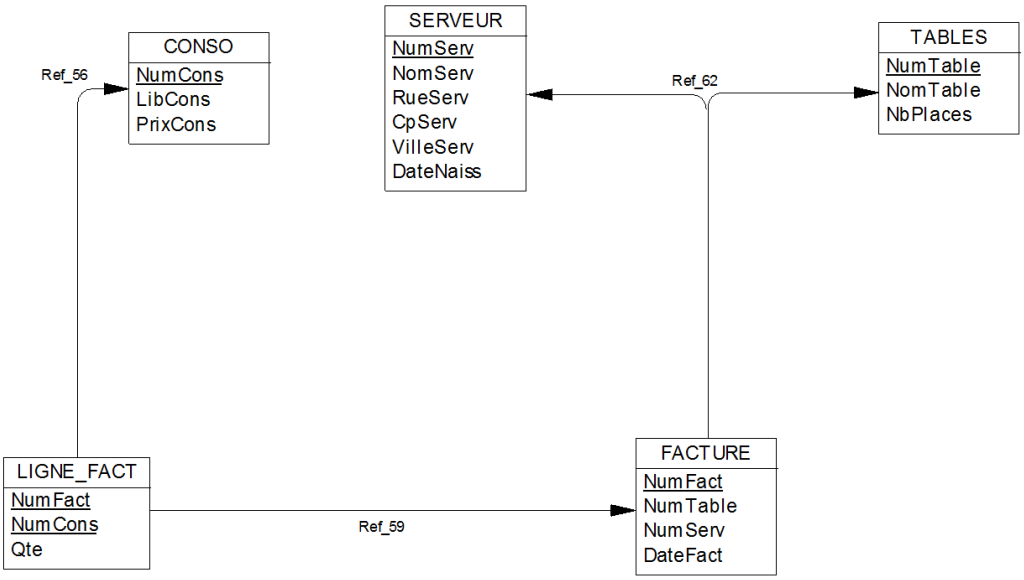
Si des erreurs sont détectées, toutes les modifications apportées aux données après l'exécution de l'instruction BEGIN peuvent être annulées afin que les données reviennent à l'état précédent de début de la transaction. Les COMMIT et ROLLBACK peuvent être implicites.

Dans notre appel de fonction précédent, pour être tout à fait rigoureux, nous devrions faire :

```
BEGIN TRANSACTION;
select virement(1,2,100);
COMMIT;
```

2. Exercice : Cas : "Au père tranquille"

On vous donne le modèle relationnel suivant décrivant le café "Au père tranquille" :



Modèle relationnel - "Au père tranquille"

Écrire et tester les fonctions ci-dessous :

Le texte de chaque fonction doit être placé dans un fichier texte. Pour chacune de ces fonctions, vous proposerez une version SQL et/ou une version en pl/pgSQL.

Question 1

1. Augmenter le prix de toutes les consommations selon la règle suivante : si le prix est inférieur à 3 euros, on augmente de 10%, sinon de 8%.

Question 2

2. Calculer et retourner le CA du serveur dont le nom est passé en paramètre

Question 3

3. Écrire une fonction qui supprime une facture et toutes les lignes de facture la concernant. Le numéro de la facture est transmis en paramètre.

Question 4

4. Écrire une fonction qui rend le montant total de la facture dont le numéro est passé en paramètre

Question 5

5. Ecrire une fonction recevant en paramètre un numéro de serveur et un numéro de table et insérant une nouvelle facture en respectant les règles suivantes :

- vérification de l'existence du numéro de serveur
- vérification de l'existence du numéro de table
- calcul du numéro de facture
- date de la facture = date du jour.

II Base de données - Les curseurs

- Les curseurs
  - Déclaration d'un curseur
  - Ouverture d'un curseur
    - Curseurs non liés à une requête
    - Curseurs liés à une requête
  - Utilisation d'un curseur
  - Fermeture d'un curseur
  - Principales structures de contrôle
    - Structures conditionnelles
    - Boucles ou répétitions
  - Exemple d'utilisation d'un curseur
- Exercice : Cas : Au père tranquille
- Exercice : Cas : Meubléa
- Exercice : Cas : Sondages

1. Les curseurs

Dans le chapitre précédent, nous avons vu qu'il était possible d'utiliser des fonctions afin de récupérer le résultat unique d'une requête ou de faire des opérations sur la base de données (INSERT, UPDATE ou DELETE). Comment récupérer le résultat d'une requête SELECT renvoyant plusieurs TUPLES ?

Le *curseur*[http://docs.postgresql.fr/9.4/plpgsql-cursors.html]* permet l'accès à chaque tuple du résultat du SELECT de façon séquentielle (c'est à dire ligne après ligne). Cet ensemble devra être déclaré (**DECLARE**), ouvert (**OPEN**), traité (**FETCH**) puis fermé (**CLOSE**). Il est possible d'avoir plusieurs curseurs actifs en même temps.

1.1. Déclaration d'un curseur

Les variables curseur sont du type de données spécial "**refcursor"**. Il existe deux moyens pour déclarer un curseur :

- déclarer un curseur comme une variable de type refcursor
- utiliser la syntaxe de déclaration de curseur qui est en général :

¶ Syntaxe :

```
nom CURSOR [ ( arguments ) ] FOR requête ;
```

Exemple :

```
DECLARE
    curs1 refcursor;
    curs2 CURSOR FOR SELECT * FROM employé;
    curs3 CURSOR (key integer) IS SELECT * FROM employé WHERE numEmp = key;

```

Remarque :

1. la première déclaration peut être utilisées avec n'importe quelle requête, alors que la seconde a une requête spécifiée et la dernière est liée à une requête paramétrée.
2. Il est évidemment possible de spécifier plusieurs paramètres en les séparant avec des virgules. Exemple : (key1 integer, key2 integer, ...)

1.2. Ouverture d'un curseur

Afin de pouvoir consulter les données retournées par le curseur, il est nécessaire de l'ouvrir. Les modes d'ouverture diffèrent suivant la déclaration du curseur.

a) Curseurs non liés à une requête

Syntaxe :

```
1 OPEN curseur-non-lié FOR SELECT ...;
```

Exemple :

Déclaration du curseur :

```
1 DECLARE curs1 refcursor;
```

Ouverture du curseur :

```
1 OPEN curs1 FOR SELECT * FROM employé WHERE numEmp = key;
```

Complément :

La requête peut être spécifiée comme une expression chaîne de la même façon que dans une commande.

```
1 OPEN curs1 FOR EXECUTE "SELECT" || $1 || "FROM employé";
```

b) Curseurs liés à une requête

Syntaxe :

```
1 OPEN curseur-lié [ ( arguments ) ];
```

Exemple :

Déclaration du curseur :

```
1 DECLARE curs2 CURSOR FOR SELECT * FROM employé;
```

Ouverture du curseur :

```
1 OPEN curs2;
```

Exemple :

Déclaration du curseur :

```
1 DECLARE curs3 CURSOR (key integer) IS SELECT * FROM employé WHERE numEmp = key;
2
```

Ouverture du curseur :

```
1 OPEN curs3(48);
```

1.3. Utilisation d'un curseur

La manipulation d'un curseur n'est possible qu'après l'avoir déclaré et ouvert.

Il est alors possible de récupérer les valeurs renvoyées par la requête à l'aide de l'instruction FETCH dont la syntaxe est la suivante :

¶ Syntaxe :

```
FETCH curseur INTO var1, var2...;
```

FETCH extrait la ligne suivante du résultat de la requête liée au curseur et stocke chacun des champs dans la liste des variables fournies après le mot-clé **INTO**.

Le variable prédéfinie FOUND permet de nous renseigner sur l'existence de résultat (elle vaut vrai si une ligne a été trouvée).

Exemple :

Dans la requête ayant ouvert le curseur, on souhaite récupérer pour chaque ligne la valeur de 3 champs (numéro, nom, salaire) :

```
FETCH curs1 INTO num, nom, salaire;
```

1.4. Fermeture d'un curseur

Il est nécessaire de fermer les curseurs ouverts pour libérer des ressources ou réutiliser un curseur.

¶ Syntaxe :

```
CLOSE curseur ;
```

Exemple :

```
CLOSE curs1 ;
```

1.5. Principales structures de contrôle

Il va être nécessaire d'utiliser certaines structures de contrôle pour pouvoir utiliser les curseurs (les boucles en particulier). Néanmoins les autres structures de contrôle peuvent également s'avérer utile.

a) Structures conditionnelles

```
IF ... THEN ... END IF ;
IF ... THEN ... ELSE ... END IF ;
IF ... THEN ... ELSE IF ... END IF ;
IF ... THEN ... ELSIF ... THEN ... ELSE ... END IF ;
```

Exemple :

```
IF var = 20 THEN
    result := ''parfait'';
ELSIF number > 10 THEN
    result := ''moyen'';
ELSIF number > 5 THEN
    result := ''insuffisant'';
ELSE
    result := ''mauvais'';
END IF;

```

b) Boucles ou répétitions

```
WHILE expression LOOP
    instructions
END LOOP;

FOR nom IN [ REVERSE ] expression .. expression LOOP
    instruction
END LOOP;

```

Exemple :

```
FOR i IN 1..30 LOOP
      RAISE NOTICE ''i is %'', i;
   ...
     END LOOP;
```

Remarque :

L'instruction ***RAISE***[http://docs.postgresql.fr/9.4/plpgsql-errors-and-messages.html]* permet d'afficher des messages et des erreurs.

Les niveaux possibles sont DEBUG, LOG, INFO, NOTICE, WARNING et EXCEPTION. EXCEPTION lève une erreur (ce qui annule habituellement la transaction en cours). Les autres niveaux ne font que générer des messages aux différents niveaux de priorité.

Exemple :

```
FOR i IN REVERSE 30..1 LOOP
      ...
     END LOOP;

```

1.6. Exemple d'utilisation d'un curseur

La fonction suivante nous permet d'afficher toutes les factures d'un serveur dont le numéro a été passé en paramètre.

```
create or replace function ListFact(integer) returns numeric as '
DECLARE nums ALIAS FOR $1;
DECLARE fact integer;
DECLARE cursfact refcursor;
BEGIN
  RAISE NOTICE ''Liste :'';
  OPEN cursfact FOR select numfact from facture where numserv = nums;
  FETCH cursfact INTO fact;
  WHILE FOUND LOOP
    RAISE NOTICE ''Facture numero : %'', fact;
    FETCH cursfact into fact;
  END LOOP;
  return 1;
END; '
language 'plpgsql';
```

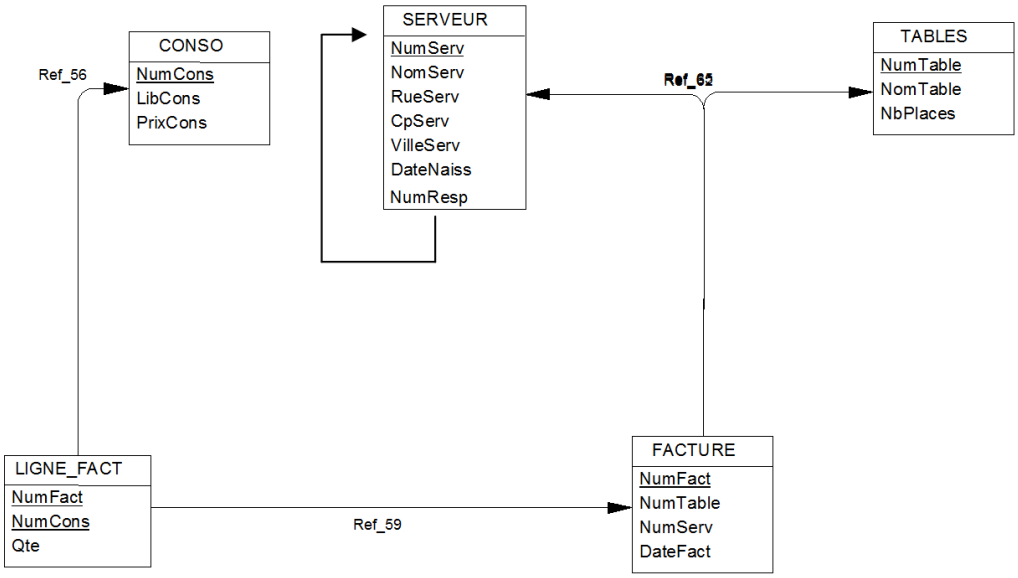
Exemple :

pour exécuter la fonction :

```
select ListFact(53);
```

2. Exercice : Cas : Au père tranquille

On vous donne le modèle relationnel suivant décrivant le café "Au père tranquille" :



La base de données a été légèrement modifiée. Le serveur peut avoir un numéro de responsable (qui est un serveur également) sauf celui a un responsable dont le n° est 0 qui est le chef des chefs.

Écrire et tester les fonctions ci-dessous :

Le texte de chaque fonction doit être placé dans un fichier texte. Pour chacune de ces fonctions, vous proposerez une version en PL/pgSQL.

Question 1

Écrire une fonction moyPrix sans paramètre qui renvoie le prix moyen des boissons.

Question 2

Utiliser ensuite cette fonction pour afficher les libellés et les tarifs des consommations qui coûtent plus que le tarif moyen.

Question 3

Utiliser ensuite cette fonction pour afficher les libellés et les tarifs des consommations dont le prix est égal au prix moyen à 10% près (c'est-à-dire ceux dont le tarif est compris entre 90% et 110% du tarif moyen).

Question 4

Écrire une fonction FctTable qui admette un numéro de facture en paramètre et qui renvoie comme résultat le nom de la table servie.

Question 5

Écrire une fonction AutreFactures qui admet un numéro de facture en paramètre et qui renvoie comme résultat le numéro et la date des factures faites par le même serveur, la facture passée en paramètre ne devant pas faire partie de la liste des factures.

Question 6

Écrire une fonction nbreLignes qui renvoie le nombre de lignes de la table dont le nom est passé en paramètre.

Question 7

Écrire une fonction qui supprime toutes les factures et les lignes de facture du serveur dont on passe le numéro en paramètre.

A tester avec les commande suivante :

```
BEGIN TRANSACTION;
  select * from facture;
  select * from ligne_fact;
  select * from suppfact(53);
  select * from facture;
  select * from ligne_fact;
  ROLLBACK;
```

Question 8

Écrire une fonction superieurs qui affiche les noms DES supérieurs du serveur dont le numéro est passé en paramètre.

Indice :

Pour commencer, écrivez une fonction qui affiche les nom DU responsable direct. Puis testez là.

3. Exercice : Cas : Meubléa

La société MEUBLEA, grande surface de meubles gérait son système d'information sur gros système. Son personnel avait accès à des terminaux texte pour les opérations courantes de gestion commerciale. Elle fonctionne comme suit :

- Les meubles sont exposés et les clients sont priés de noter le code produit figurant sur les étiquettes. Lorsqu'ils ont fait leur choix, ils contactent un vendeur qui enregistre une réservation pour tous les produits concernés.
- Le client, muni de son numéro de réservation se présente ensuite aux caisses, pour verser le montant de l'acompte du. Le solde sera à payer lors de la livraison (Cet aspect n'est pas à gérer dans ce TP).
- Les réservations qui n'auraient pas donné lieu à paiement de l'acompte dans la journée sont purement et simplement détruites après fermeture du magasin...

Après exécution de la procédure stockée, les lignes en caractères gras devront être supprimées et on devra trouver dans la table produit :

- 6 tables
- 16 chaises
- 4 canapés
- 10 fauteuils

Annexe : le script de création de la base :

```
create table produit (
    codep     integer           not null,
    desigp    varchar(30)       null    ,
    qtestock  integer           null    ,
    prixunit  float             null    ,
    constraint PK_PRODUIT primary key (CODEP)
    );

create table reservation(
    numr     integer              not null,
    dater    date                 null    ,
    confirme  integer            null    ,
    constraint PK_RESERVATION primary key (NUMR)
    );
```

```
15
16 create table concerner(
17     numr          integer          not null,
18     codep         varchar(10)      not null,
19     qter          integer          null    ,
20     constraint PK_CONCERNER2 primary key (CODEP, NUMR),
21     constraint fk_concernerProduit foreign key (codep) references produit(codep),
22     constraint fk_concernerreserv foreign key (numr) references reservation(numr)
23     ) ;
24
25 insert into produit values(1, 'table', 5, 150);
26 insert into produit values(2, 'chaise', 10, 10);
27 insert into produit values(3, 'buffet', 4, 1200);
28 insert into produit values(4, 'fauteuil', 8, 345);
29 insert into produit values(5, 'canape', 3, 2000);
30 insert into produit values(6, 'etagere', 8, 80);
31
32 insert into reservation values(1, '10/12/2005', 1);
33 insert into reservation values(2, '10/12/2005', 1);
34 insert into reservation values(3, '12/12/2005', 1);
35 insert into reservation values(4, '12/12/2005', 1);
36 insert into reservation values(5, current_date, 1);
37 insert into reservation values(6, current_date, 1);
38 insert into reservation values(7, current_date, 0);  /* non confirmee */
39 insert into reservation values(8, current_date, 0);  /* non confirmee */
40
41 insert into concerner values(1,1,2);
42 insert into concerner values(2,2,4);
43 insert into concerner values(3,4,2);
44 insert into concerner values(4,1,2);
45 insert into concerner values(5,5,1);
46 insert into concerner values(6,6,3);
47 insert into concerner values(7,1,1);
48 insert into concerner values(7,2,6);
49 insert into concerner values(8,5,1);
50 insert into concerner values(8,4,2);
```

Question

Écrire une procédure stockée, qui épure la base de toutes les réservations non confirmées de la journée.

Pour chacune de ces réservations, il faut remettre en stock les quantités qui en ont été déduites lors de l'enregistrement de la réservation.

4. Exercice : Cas : Sondages

Il s'agit de développer une application en JAVA permettant de :

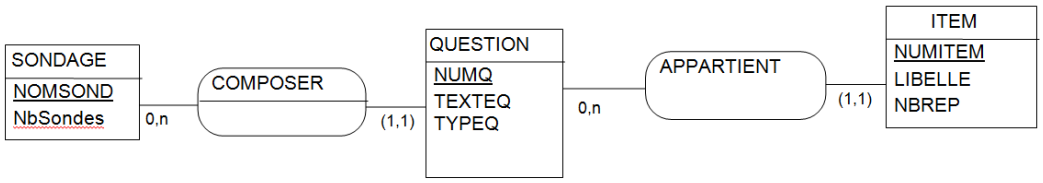
saisir les réponses aux questions d'un sondage

consulter les résultats chiffrés d'un sondage

créer un nouveau sondage

Dans cet exercice on ne s'occupera que de la partie données...

On vous donne le MCD suivant :



Modèle conceptuel de données (MCD) - Sondages

Remarques :

- Il existe deux types de questions : les question à choix unique (TYPEQ vaut 1) et les questions à choix multiple (TYPEQ vaut 2). Elles ne se différencient pas au niveau des données mais uniquement au niveau du comportement (méthodes de saisie des réponses, d'affichage et de calcul des résultats).
- Une question est composée d'au moins un et d'au plus 5 items.
- Le vocabulaire du domaine d'étude est illustré par les exemples suivant :

Vocabulaire du domaine d'étude

NOMSOND	ex : "aliments"
NBSONDES	Nombre de personnes ayant répondu au sondage (0 à la création du sondage)
TEXTEQ	Texte général de la question ex : "Combien de fois par semaine mangez-vous de la viande ?"
LIBELLE	de l'item 1 de la question 3 du sondage "aliments" : ex : "une fois par semaine"
NBREP	Nombre de fois où cet item a été retenu pour répondre à la question à laquelle il appartient. (0 à la création de la question).

Travail à faire :

Créer une nouvelle base données sous PostgreSQL et exécuter le fichier sondages.sql.

Écrire et tester les fonctions ci-dessous en pl/pgSQL :

Question 1

Écrire une fonction qui permet d'incrémenter le nombre de sondés à partir d'un nom de sondage passé en paramètre. On affichera un message d'erreur si le sondage n'existe pas.

Question 2

Écrire une fonction qui permet d'afficher pour chaque sondage le nombre de ses questions.

Question 3

Écrire une fonction ProportionItem qui admet un nom de sondage en paramètre et qui affiche pour chaque numéro et nom d'items le pourcentage de fois où ils ont été choisis pour ce sondage.

Question 4

Écrire une fonction qui supprime tous les sondages (ainsi que ses questions et réponses) n'ayant aucun sondé. Afficher le nom du sondage à l'écran avant de le supprimer. Elle retournera le nombre de sondages supprimés.

III Bases de données - Les déclencheurs

- Les déclencheurs ou triggers
  - Qu'est ce qu'un déclencheur ?
  - Comment faire un trigger ?
- Exercice : Cas : Sondages

1. Les déclencheurs ou triggers

Les triggers (aussi appelés déclencheurs en français) sont, au même titre que les contraintes, un élément fondamental dans la structure des bases de données. Alors qu'une contrainte permet de définir une règle algébrique, le trigger permet de définir une règle algorithmique. Il peut également être très utile d'un point de vue purement fonctionnel.

1.1. Qu'est ce qu'un déclencheur ?

Un trigger est une fonction affectée à une table et qui est déclenchée sur certaines opérations.

Étant stockés dans la base de données, ils permettent donc non seulement un développement plus rapide mais également un maintien global des règles d'intégrité.

Voici deux exemples mettant en évidence l'utilité des triggers :

***Exemple* : Contrainte logique**

Avec le mot-clé REFERENCES on a vu que l'on pouvait définir des contraintes de type clef étrangère. Mais vous pouvez avoir besoin de définir des contraintes de type " soit l'un, soit l'autre". C'est à dire deux champs qui sont des clefs étrangères, mais pour un tuple donné il n'est pas possible d'avoir les deux champs renseignés.

Exemple : Un intervenant travaille obligatoirement soit sur une région, soit pour une marque de véhicule, mais jamais pour les deux en même temps :

INTERVENANT(code, nom, prénom, codeMarque#, codeRégion#)

code : clé primaire

codeMarque : clé étrangère en référence à code de MARQUE

codeRégion : clé étrangère en référence à code de REGION

Pour cela, il est utile d'avoir une fonction déclenchée automatiquement lors de l'écriture d'une valeur dans ce champ, et qui va vérifier que cette condition sera valide avec la nouvelle valeur.

***Exemple* : Utilité fonctionnelle**

Si une de vos tables contient un champ texte, et que vous désirez garder une trace des versions successives du texte. Plutôt que de gérer cela au sein de votre application (avec le risque, au cours de la maintenance évolutive, qu'un développeur oublie d'intégrer cette gestion dans une nouvelle portion de code permettant de modifier le texte), le plus fiable et le plus simple sera de définir un trigger, déclenché sur le **UPDATE**, qui ira archiver l'ancienne version du texte avant de le mettre à jour (il sera aussi nécessaire d'effectuer un archivage sur le **DELETE**, mais c'est là un point de détail).

Un trigger peut être déclenché **avant** ou **après** une instruction de type **INSERT**, **DELETE** ou **UPDATE**. De plus, il peut être appelé pour chacune des lignes concernées par l'instruction, ou une seule fois pour l'instruction.

1.2. Comment faire un trigger ?

La première étape sera de créer une procédure stockée. Cette procédure (ou fonction) constitue le trigger proprement dit. Cette fonction n'admet aucun argument, et retourne le type OPAQUE (type indéterminé).

Lorsqu'une fonction est appelée en tant que trigger, elle hérite automatiquement lors de l'exécution d'un certain nombre de variables particulièrement utiles, parmi lesquelles on peut citer :

- NEW : La ligne telle qu'elle sera après l'exécution de l'instruction qui a déclenché le trigger.
- OLD : L'état de la ligne avant l'exécution de l'instruction qui a déclenché le trigger.

Exemple :

Voici un petit exemple pour y voir plus clair. Nous allons nous créer deux tables, une table pour y stocker des articles, et une autre table d'archivage afin de stocker les versions successives des articles :

```
create table article
(id integer not null primary key,
date_modif date not null default now(),
texte text,
status varchar);

create table archive
(id integer,
date_modif date not null,
texte text);
```

Maintenant, nous allons créer une fonction qui sera appelée lors d'une modification d'un article afin d'archiver la version précédente de l'article :

```
DROP FUNCTION arch_art();            /* Petite precaution */

CREATE FUNCTION arch_art() RETURNS OPAQUE AS '
BEGIN IF NEW.texte != OLD.texte THEN
             INSERT INTO archive(id, date_modif, texte) VALUES
(OLD.id,OLD.date_modif,OLD.texte);
      END IF;
      RETURN NEW;
END;'
LANGUAGE 'plpgsql';
```

La dernière ligne (RETURN NEW), retourne la ligne telle qu'elle devra être écrite dans la base de données. Dans notre cas, elle reste inchangée par rapport à l'ordre UPDATE original.

On remarque que le trigger sera appelé pour tout ordre UPDATE, et qu'il est donc nécessaire de vérifier que le texte de l'article à bel et bien été modifié avant de l'archiver.

Maintenant, nous allons créer le trigger proprement dit. Pour cela, nous indiquons à la base de données la table concernée, les opérations concernées, et la procédure stockée contenant le code du trigger :

Exemple :

```
DROP TRIGGER trg_arch_art ON article;

CREATE TRIGGER trg_arch_art BEFORE DELETE OR UPDATE ON article
FOR EACH ROW EXECUTE PROCEDURE arch_art() ;
```

Maintenant, vérifions que le trigger fonctionne correctement :

```
insert into article(id,texte) values(1,'Ceci est le premier article original') ;
```

```
INSERT 40942 1
```

```
=> select * from article ;
```

```
id | date_modif |          texte                       | status
---+------------+--------------------------------------+--------
 1 | 2008-11-19 | Ceci est le premier article original |

1 ligne(s)
```

```
select * from archive ;
```

```
id | date_modif | texte
---+------------+-------

0 ligne(s)
```

```
update article set texte='Ceci est le premier article version 2',date_modif=now();
```

```
UPDATE 1
```

```
select * from article ;
```

```
id | date_modif |           texte                      | status
---+------------+--------------------------------------+--------
 1 | 2008-11-19 | Ceci est le premier article version 2 |

1 ligne(s)
```

```
select * from archive ;
```

```
id | date_modif |                         texte
---+------------+---------------------------------------
 1 | 2008-11-19 | Ceci est le premier article original

1 ligne(s)
```

Modification du trigger (nouveau besoin du système d'information)

Maintenant, nous allons modifier notre trigger afin de refuser toutes modifications si son champ status indique 'publié'.

```
DROP FUNCTION arch_art();

CREATE FUNCTION arch_art() RETURNS OPAQUE AS '
BEGIN IF NEW.texte != OLD.texte THEN
           IF OLD.status = ''publié'' THEN
                RAISE EXCEPTION ''Article % déjà publié'',NEW.id;
           END IF;
           INSERT INTO archive(id, date_modif, texte) VALUES
(OLD.id,OLD.date_modif,OLD.texte);
      END IF;
RETURN NEW;
END; '
LANGUAGE 'plpgsql';
```

```
DROP TRIGGER trg_arch_art ON article;

CREATE TRIGGER trg_arch_art BEFORE DELETE OR UPDATE ON article
FOR EACH ROW EXECUTE PROCEDURE arch_art();
```

Exemple **:**

Voici une illustration de l'exécution de ce trigger :

```
select * from article ;
```

```
id | date_modif |               texte                  | status
---+------------+--------------------------------------+--------
 1 | 2008-11-19 | Ceci est le premier article version 2 |

1 ligne(s)
```

```
select * from archive ;
```

```
id | date_modif |                                texte
---+------------+---------------------------------------
 1 | 2008-11-19 | Ceci est le premier article original

1 ligne(s)
```

```
update article set status='publié' where id=1 ;
```

```
UPDATE 1
```

```
select * from article ;
```

```
id | date_modif |               texte                   | status
---+------------+---------------------------------------+--------
 1 | 2008-11-19 | Ceci est le premier article version 2 | publié

1 ligne(s)
```

```
select * from archive ;
```

```
id | date_modif |                                texte
---+------------+---------------------------------------
 1 | 2008-11-19 | Ceci est le premier article original

1 ligne(s)
```

```
update article set texte='Ceci est une tentative de modification',date_modif=now();
```

```
ERROR: Article 1 deja publie
```

Comme vous pouvez le voir, les triggers sont particulièrement pratiques pour débarrasser l'application cliente des archivages par exemple. Ils sont aussi très conseillés pour gérer des règles de sécurité, ainsi que les contraintes d'intégrités.

2. Exercice : Cas : Sondages

Il s'agit de développer une application en JAVA permettant de :

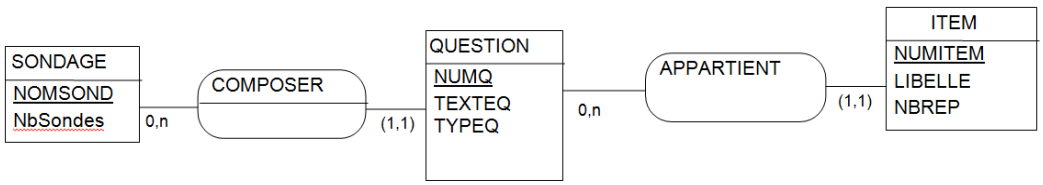
saisir les réponses aux questions d'un sondage

consulter les résultats chiffrés d'un sondage

créer un nouveau sondage

Dans cet exercice on ne s'occupera que de la partie données...

On vous donne le MCD suivant :



Modèle conceptuel de données (MCD) - Sondages

Remarques :

- Il existe deux types de questions : les question à choix unique (TYPEQ vaut 1) et les questions à choix multiple (TYPEQ vaut 2). Elles ne se différencient pas au niveau des données mais uniquement au niveau du comportement (méthodes de saisie des réponses, d'affichage et de calcul des résultats).
- Une question est composée d'au moins un et d'au plus 5 items.
- Le vocabulaire du domaine d'étude est illustré par les exemples suivant :

Vocabulaire du domaine d'étude

NOMSOND	ex : "aliments"
NBSONDES	Nombre de personnes ayant répondu au sondage (0 à la création du sondage)
TEXTEQ	Texte général de la question ex : "Combien de fois par semaine mangez-vous de la viande ?"
LIBELLE	de l'item 1 de la question 3 du sondage "aliments" : ex : "une fois par semaine"
NBREP	Nombre de fois où cet item a été retenu pour répondre à la question à laquelle il appartient. (0 à la création de la question).

Travail à faire :

Reprendre la base données sous PostgreSQL et exécuter le fichier sondages.sql si besoin est (les curseurs du chapitre précédent seront perdus, d'où l'importance de la sauvegarde des scripts dnas un fichier texte.).

Écrire et tester les fonctions ci-dessous en pl/pgSQL :

Question 1

Le nombre de réponses données pour un item ne peut être supérieur au nombre de sondés de ce sondage.

Question 2

Lors de l'enregistrement d'un nouvel item, le numéro de l'item doit être égal au numéro maximum de l'item de la même question + 1.

Question 3

Le nombre de points marqué par un internaute pour un exercice ne peut être supérieur au nombre de questions de cet exercice.

Question 4

Lorsqu'on modifie une question le champ réponse doit être égal à l'un des choix existants.

Question 5

Lorsqu'on supprime un choix pour une question, il ne peut s'agir de la bonne réponse.

Question 6

Lors de l'enregistrement d'un score, la date est forcément la date système.

IV Bases de données - Répartition des données

- Les bases de données réparties
    - Le Client / Serveur
        - Définitions et principe de fonctionnement
        - Intérêt d'une architecture client / Serveur
        - Typologies des architectures Client / Serveur
        - Le client / Serveur universel
    - Données distribuées
        - Analyse des besoins
        - Qu'est-ce-qu'une base de données répartie ?
            - Définition
            - Techniques de distribution des données
            - Synthèse concernant l'implantation de données réparties avec réplication
        - Règles et recommandations pour choisir le lieu d'implantation des tables
            - Critères à prendre en compte
            - Recommandations dans un cas simple d'architecture Client / Serveur
    - Exemple de synthèse
- Exercice : Vas : Kimassur

1. Les bases de données réparties

1.1. Le Client / Serveur

a) Définitions et principe de fonctionnement

- L'architecture client-serveur désigne l'ensemble des moyens matériels, logiciels et de communication permettant de mettre en place une application conforme au modèle client-serveur.
- Le modèle client-serveur est un mode de fonctionnement théorique basé sur la séparation des rôles. Est appelé client-serveur un modèle de fonctionnement logiciel dans lequel un programme utilise des services offerts par un autre programme indépendant, en communicant avec lui à l'aide de messages.

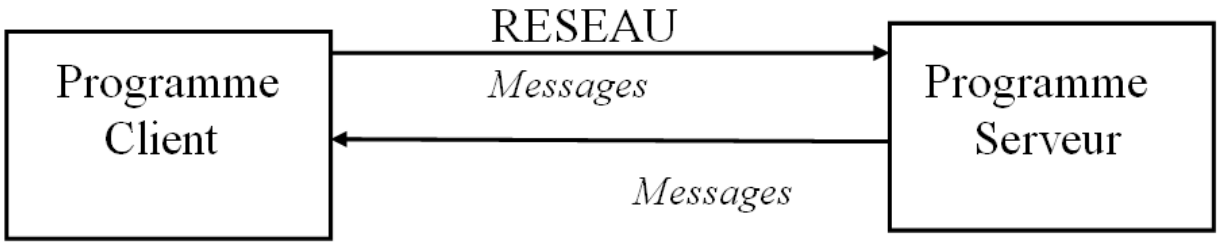
Le terme serveur fait référence à tout processus qui:

- reçoit une demande de service venant d'un client via un réseau,
- traite cette demande,
- et renvoie le résultat au demandeur (= le client).

Un serveur peut être spécialisé en serveur d'applications, de fichiers, de terminaux, ou encore de messagerie électronique.

Le terme client fait référence à tout programme "demandeur de service" via un réseau.

Principe de fonctionnement



Transmission des messages à travers un réseau

C'est l'application "cliente" qui prend l'initiative du dialogue. Le programme dit "client" demande par l'envoi d'un message, un service extérieur à un autre programme dit "serveur".

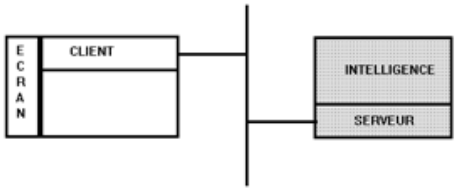
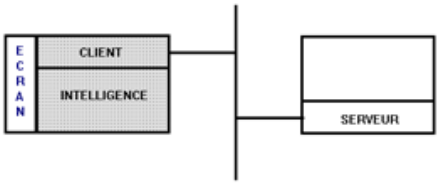
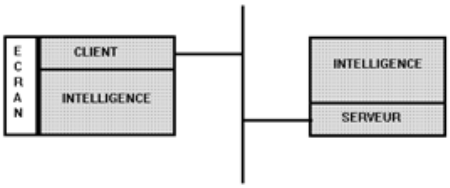
Une fois le service rendu, le programme serveur renvoie le résultat au client sous forme de message.

Une architecture client-serveur fournit donc des services distants (ex : bases de données, composants distribués, ...) à des clients et ce de manière transparente pour l'utilisateur.

Par abus de langage, on parlera de pseudo client-serveur (ou client-serveur en local) pour désigner un dialogue entre un client et un serveur situés sur un même poste.

b) Intérêt d'une architecture client / Serveur

- Répartir les tâches entre le client et le serveur.
- Réduire le trafic sur le réseau.
- Permettre à l'utilisateur de disposer d'une fenêtre unique sur le système d'information de l'entreprise.

Modèle centralisé multi-utilisateurs 	serveur = ordinateur central qui effectue pratiquement tous les traitements client = terminal sans puissance locale de traitement puis micros "intelligents" dont la puissance locale est gaspillée et sous-utilisée.
Modèle serveur de fichiers 	Les stations clientes effectuent la plupart des traitements, les serveurs se contentent de gérer le réseau et de stocker les fichiers et bases de données.
Modèle Client / Serveur 	Ce modèle correspond à une répartition judicieuse de la puissance de traitement entre les différentes stations interconnectées.

Modèles d'architecture Client / Serveur

c) Typologies des architectures Client / Serveur

Le découpage d'une application

Toute application de compose de trois parties appelés **modules fonctionnels**.

Modules fonctionnels d'une application

Interface Utilisateurs **(Modèle présentation)**	**Gestion et logique d'affichage**. • Gestion de la présentation : affichage proprement dit et réception des actions de l'utilisateur • Logique de présentation : description des objets à afficher et réaction aux événements déclenchés par l'utilisateur (validation de 1er niveau des saisies ...)
Logique de l'application **(Module traitements)**	**Ensemble des traitements applicatifs**. Peut être séparé en 2 parties : • Logique fonctionnelle (structures algorithmiques) • Manipulation des données
Gestion des données **(Module données)**	**Accès aux données**. (Pris en charge par le SGBD)

La répartition de ces composants entre le client et le serveur se fera en fonction :

- des types d'architectures retenues,
- de la capacité des machines,
- de la capacité du réseau.

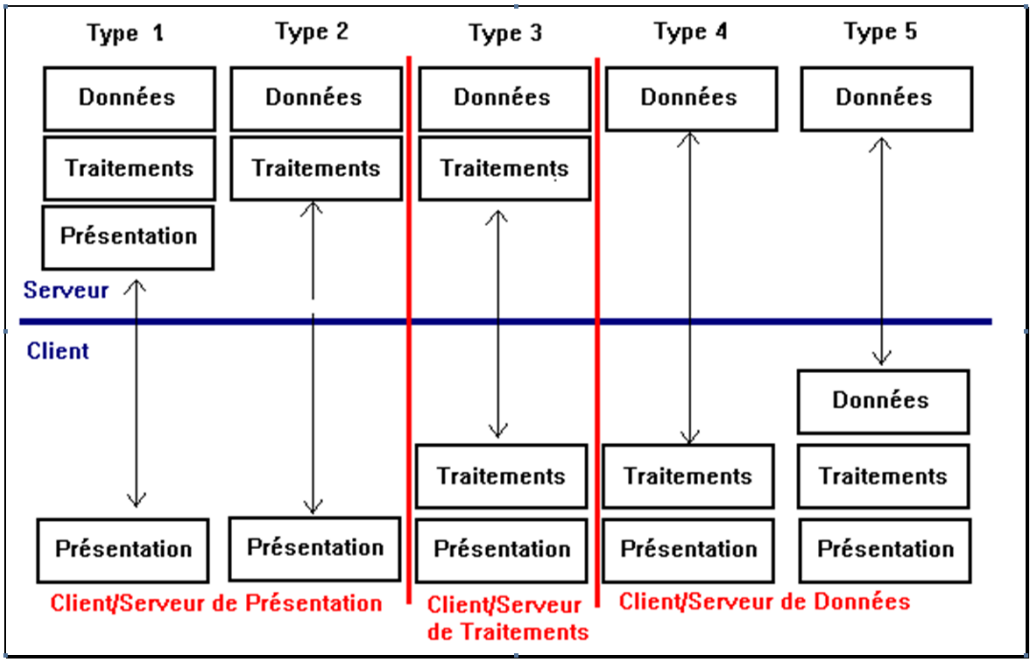
La classification du Gartner Group

Le *Gartner Group*[https://fr.wikipedia.org/wiki/Gartner]*, société américaine de consultants, a proposé une typologie des applications client-serveur. Cette typologie propose 5 types d'architecture client/serveur, établis en fonction de la localisation, sur le poste client et/ou sur le poste serveur des modules fonctionnels.

Il existe des combinaisons possibles de ces différents types d'architecture, cependant trois grands types se dégagent :

- Client-Serveur de présentation : types 1 et 2
- Client-Serveur de traitements : type 3
- Client-Serveur de données : types 4 et 5

Le schéma ci-après récapitule ces différents types d'architecture :



Les types de Client / Serveur - Gartner Group

Type 1 : la présentation distribuée

Données - Traitements - Présentation / Présentation

Elle permet d'améliorer l'interface utilisateur sans modifier l'architecture de l'application existante. Le poste de travail fournit une interface graphique évoluée à une application existante conçue à l'origine avec un affichage en mode caractère.

C'est ce qu'on appelle le '**rhabillage**' ou le '**revamping**' ou encore le 'remodelage' d'application. Cette approche permet de remplacer les terminaux par des micros.

Type 2 : la présentation déportée

Données - Traitements / Présentation

La partie "présentation" n'existe que sur le poste client : toute la gestion de l'interface homme-machine est déportée au niveau du poste client. Le cœur de l'application et l'ensemble des données résident sur le serveur.

C'est le cas des toutes les applications WEB où aucun traitement n'est effectué sur le poste client (ni scripts, ni applets).

Type 3 : les traitements distribués

Données - Traitements / Traitements - Présentation

Les traitements de l'application sont répartis entre le poste client et le poste serveur. Le serveur effectue la partie manipulation des données et gère l'accès aux données.

La partie cliente fait appel au serveur pour exécuter un certain nombre de services extérieurs. Elle envoie pour cela des requêtes qui vont activer des traitements, appelés procédures, localisés sur le serveur. Le client effectue donc la gestion de l'interface ainsi que la logique fonctionnelle de l'application.

Ce type de client-serveur est également qualifié de client-serveur de procédures.

Type 4 : la gestion des données distantes

Données / Traitements - Présentation

La partie cliente assure à la fois les fonctions de présentation et de traitement : la gestion du dialogue, la validation des saisies, la mise en forme des résultats et les traitements applicatifs (y compris la manipulation des données).

Le serveur héberge l'ensemble des données et assure uniquement l'accès à ces données, le plus souvent à l'aide d'un système de gestion de base de données relationnel (SGBDR). Le serveur est, dans ce cas, qualifié de serveur de données.

Dans ce modèle, l'application cliente envoie les requêtes au serveur de données, qui envoie en retour le résultat demandé.

Type 5 : la gestion de données distribuées

Données / Données - Traitements - Présentation

Les données sont réparties entre le poste client et un ou plusieurs postes serveurs.

Dans un tel environnement, les données réparties doivent pouvoir être gérées comme un ensemble unique et cohérent (bases de données réparties).

Le poste client devenant lui-même serveur de données, il se crée des liens de type serveur-serveur qui nécessitent une gestion des données à grande échelle et bien souvent, dans un environnement hétérogène.

Complément **:**

Types de Client / Serveur (CS)	Intérêts	Inconvénients
CS de présentation	• améliore l'interface utilisateur de l'application • allonge la durée de vie des applications existantes • constitue une solution intermédiaire avant le passage au vrai client-serveur.	• débits importants sur le réseau. • le serveur assure seul l'ensemble des fonctions.
CS de traitements	• limitation du trafic sur le réseau. • utilisation rentable de la puissance du poste de travail. • répartition équilibrée des traitements entre clients et serveurs	• processus de développement plus complexe car les traitements devront prendre en compte deux environnements différents.

Types de Client / Serveur (CS)	Intérêts	Inconvénients
CS de données	• répartition claire entre client et serveur • la sécurité, la gestion et le contrôle d'intégrité des données sont pris en charge par le serveur.	• génère un dialogue plus important que le mode traitements distribués (type 3).

Simulation : Exercice : Types de client / Serveur selon le le Gartner Group

A partir de la description et de la classification du Gartner Group, indiquer de quel(s) type(s) de client-serveur relèvent les applications mentionnées ci-dessous :

Description de l'application	Type(s) de client-serveur
La couche présentation et les traitements sont gérés par le poste de travail sous Windows. Les données sont stockées dans la base de données Oracle.	
La couche présentation et les traitements sont gérés par le poste de travail sous Windows. Les données et certains traitements sont stockés dans la base de données Oracle.	
La couche présentation et les traitements sont gérés par le poste de travail sous Windows. Les données sont stockées dans la base de données Oracle et une partie des données est répliquée dans une base de données locale gérée par SQL Anywhere de Sybase.	
L'application "Gestion des élèves" a été développée avec Access. La couche de présentation et les traitements sont gérés par un poste de travail sous Windows. Les données sont stockées dans la base de données Oracle.	
L'application « Taxe d'apprentissage » a été développée avec NetBeans. La couche de présentation et les traitements sont gérés par un poste de travail sous Windows. Les données et certains traitements sont stockés dans la base de données Oracle.	
L'application « Gestion des stages » a été développée avec Access. La présentation et les traitements sont gérés par un poste de travail sous Windows. Les données sont stockées dans la base de données Oracle et dans la base de données SQL Server (Microsoft).	

Description de l'application	Type(s) de client-serveur
L'application « Gestion des absences » a été développée avec NetBeans. La couche de présentation et les traitements sont gérés par un poste de travail sous Windows. Les données et certains traitements sont stockés dans la base de données Oracle. D'autre part, sur chaque poste en local, vous disposez du SGBDR SQL Anywhere qui gère également une partie des données.	
L'application "Gestion du budget" a été développée en Cobol et est interfacée avec la base de données relationnelle Oracle. Elle dispose d'une interface en mode caractère. La couche de présentation, les traitements et les données sont gérés par le serveur Unix. Certains postes de travail sont équipés avec Windows et l'outil de rhabillage Affinity qui permet de bénéficier d'une interface graphique.	

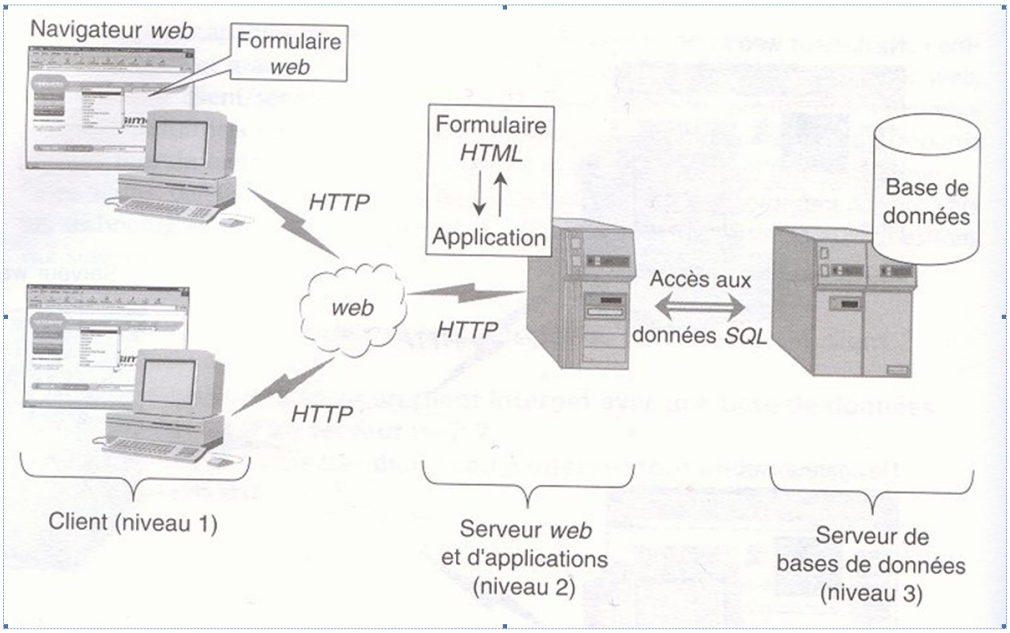
d) Le client / Serveur universel

Une nouvelle forme de client-serveur s'impose avec Internet : il s'agit d'un client-serveur dans lequel la partie présentation est assurée par un navigateur HTML ayant le rôle du client et la partie traitements est située sur un serveur Web et transmise par morceaux et en fonction des besoins vers le client.

Si l'application effectue des accès à une base de données, la partie serveur de données constitue généralement un 3ème niveau applicatif, le modèle a donc été baptisé client-serveur à 3 niveaux.

Un niveau pour chaque partie de l'application :

1. Au niveau du client ---------- Niveau Présentation (ou Interface)
2. Au niveau du serveur d'application web ------ Niveau Traitement (ou Application)
3. Au niveau du serveur de base de données ---- Niveau Données (Base de données)



Client / Serveur à 3 niveaux

Comme dans le cadre du client-serveur « classique », le client émet une requête, le serveur répond à sa demande. Cette requête est émise via son navigateur (Internet Explorer ou Firefox, Google Chrome, Opéra, ... par exemple) au serveur web en fournissant une adresse URL (adresse d'une page).

1er cas

Le serveur d'application web (ou serveur HTTP) répond à la requête du client en retournant la page écrite en langage HTML. Cette page est une page statique, elle contient toujours les mêmes informations, elle s'affiche dans le navigateur du client. C'est une interaction de type client-serveur à 2 niveaux.

2ème cas

La page demandée par le client est une page d'extension PHP par exemple. C'est donc un fichier contenant des instructions en langage PHP. Le serveur web transmet le fichier au module PHP.

Le module PHP construit une page HTML à la volée. Pour la construire, il fait appel à une base de données (Mysql ou PostGreSQL par exemple – niveau 3).

Le serveur HTTP (application – niveau 2) transmet la page HTML construite précédemment au client. Ce fichier HTML s'affiche à l'aide du navigateur (présentation – niveau 1). La page affichée est donc créée dynamiquement en utilisant les informations de la base de données.

C'est une interaction de type client-serveur à 3 niveaux ou client universel.

1.2. Données distribuées

a) Analyse des besoins

Pour répondre au mieux aux besoins d'accès aux données et plus généralement aux besoins des applications, nous allons nous poser un certain nombre de questions quant à l'architecture logicielle à mettre en place :

- Quels sont les outils à notre disposition ?
- Avec ces outils, quelles sont les solutions existantes pour implanter les données et les traitements de l'application ?
- Où placer les tables ? Sur chaque machine ? Sur un serveur ? Sur les deux ?
- Quels sont les critères sur lesquels nous allons nous baser pour choisir le lieu d'implantation des tables ?

Le besoin de rapprocher les données de l'utilisateur afin d'éviter la saturation du réseau et le nombre très important de requêtes à traiter par une seule machine a fait émerger le besoin de répartir les données sur plusieurs machines.

Les nouveaux SGBD (SGBD capables de gérer des données réparties) et le fait que le poste de travail soit aujourd'hui un micro-ordinateur doté d'une puissance de stockage et de traitement ont permis la mise en place d'architectures de type client-serveur (de type 5) dans lesquelles les données sont réparties sur le serveur et les postes de travail. On parle alors de **base de données réparties**.

Dans quel cas mettre en place une base de données réparties ?

- quand la quantité de données à traiter est importante ce qui nuit aux performances du SGBD et sature le réseau,

ou / et

- quand le nombre d'utilisateurs est important ce qui rend le partage de la base centrale inefficace.

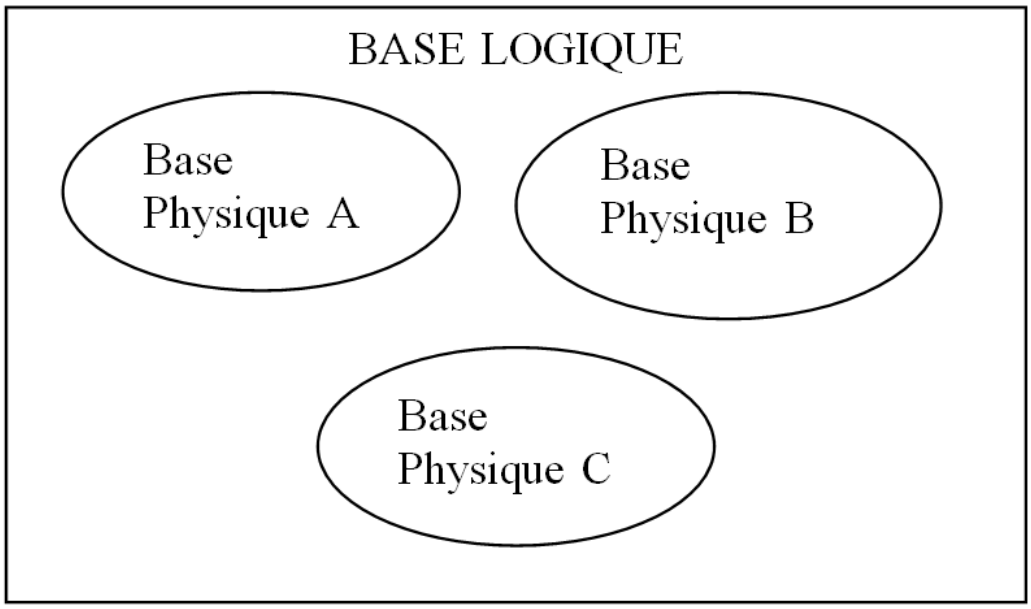
b) Qu'est-ce-qu'une base de données répartie ?

i) Définition

Une base de données réparties est un ensemble de bases de données situées sur différents postes et apparaissant aux applications comme une seule et même base de données. Le fait que les bases soient implantées sur des machines différentes est transparent pour l'utilisateur.

Dans une base de données répartie, on distingue, d'une part, la base de données logique et, d'autre part, les bases de données physiques.

- La base de données vue de l'utilisateur comme une seule et même base de données est appelée **base logique**.
- Chaque base de données regroupée au sein d'une base logique est appelée **base physique**.



Base Logique / Bases physiques

Un modèle logique des données réparti précise l'implantation des données permanentes sur chacun des postes de travail d'un système. Le MLD réparti est composé de 2 à n MLD locaux.

Chaque MLD local est propre à un type de poste de travail.

ii) Techniques de distribution des données

Les techniques de distribution (également appelées techniques de répartition) des données du MLD réparti sont de deux types :

1 - La distribution dite intégrale :

- avec segmentation des données au niveau table :

Exemple **:**

Au niveau d'une table, la segmentation (ou fragmentation) peut être horizontale (distribution des tuples) ou verticale (distribution d'attributs).

La segmentation horizontale est assez répandue, elle va par exemple permettre à une entreprise installée dans plusieurs régions d'implanter dans chaque région (donc au plus près des utilisateurs) les données (clients ...) relatives à la région. Par exemple, les clients de Paris se trouvent dans une table sur le site de Paris tandis que ceux de Dijon se trouvent dans une table sur le site de Dijon.

La segmentation verticale est très peu répandue.

- sans segmentation au niveau table :

on parle également de distribution intégrale si on choisit de mettre les différentes tables de la base logique dans des bases physiques différentes sans pour autant faire de la segmentation au niveau table.

2 - La distribution par réplication :

Elle consiste à dupliquer les données à partir d'un site d'origine par envoi de copies vers des sites de réplication.

Ces deux types de distribution peuvent être combinés, c'est le cas par exemple quand on fragmente horizontalement une table dans plusieurs régions et qu'on souhaite avoir une consolidation des différentes parties de la table au niveau du siège.

Quand il y a répartition de données, une requête émise par un client peut entraîner l'exécution de requêtes sur plusieurs sites afin d'assurer la cohérence des données.

Lors de requêtes de mise à jour (au sens large) sur le(ou les) site(s) distant(s) : c'est le cas quand il y a duplication des données, il faut assurer la propagation des mises à jour. Deux types de MAJ peuvent être utilisées :

- soit la MAJ synchrone (méthode PUSH) : la mise à jour des copies est assurée en temps réel. **Avantage** : données toujours à jour et intègres, **Inconvénient** : nécessaire disponibilité du réseau.
- soit la MAJ asynchrone (méthode PULL) où les modifications sont propagées à intervalles réguliers dans les copies ou bien à la demande d'un site répliquât (site sur lequel figure une copie). **Avantage** : les sites peuvent être indisponibles (sans que cela empêche les MAJ).

  Inconvénient : il faut travailler avec des données moins "fraîches".

Remarque **:**

1. La réplication peut être manuelle. Dans ce cas, c'est l'administrateur qui la demande explicitement.
2. Il est possible d'autoriser la réplication symétrique (le site ou le poste devient alternativement copie ou référence).

Simulation **: Exercice 2 : KES**

La société KES vend des terminaux « point de vente » (TPV) à des hypermarchés, supermarchés et petits commerçants. KES compte de nombreux sites en France (centres régionaux et agences). Elle dispose d'une informatique centralisée autour d'un ordinateur IBM (mainframe) installé à son siège social. Les postes de travail sont installés au siège social et dans les centres régionaux ; ce sont soit des terminaux passifs soit des micro ordinateurs dotés d'une carte d'émulation leur permettant, si nécessaire, de remplir les fonctions d'un terminal passif.

La société KES a décidé de renforcer l'autonomie financière et comptable des centres régionaux. Chaque centre régional sera organisé en centre de profit, il sera donc appelé à tenir sa propre comptabilité d'établissement, regroupant les agences qui dépendent de lui. Cependant, le siège social continuera d'exercer le suivi financier des différents centres régionaux. Il s'agit, en particulier, d'automatiser la procédure de consolidation des données financières émanant de ces différents centres.

Pour satisfaire ce besoin, le responsable informatique de la société KES propose de mettre en place une architecture client-serveur de données distribuées. Certaines données seront « répliquées », c'est-à-dire recopiées, pour permettre à la fois une gestion décentralisée et une consolidation des données comptables.

L'annexe 1 fournit un extrait du schéma relationnel des données relatives à la comptabilité de la société. Ayant consulté cette annexe, vous êtes chargé(e) de concevoir le scénario de répartition des données sur les différents sites (siège et centres régionaux).

Sur chaque site, seront stockées :

1. des tables de référence, pour lesquelles il est le seul autorisé à mettre à jour les données contenues ;
2. des tables répliquées, copies des tables de référence des autres sites, accessibles en consultation seulement.

Les responsables de la société KES souhaitent que les régions n'agissent que sur les informations les concernant directement. Par exemple, à propos des clients, le centre régional de Lyon ne pourrait consulter et mettre à jour que les données relatives aux clients de la région Rhône-Alpes.

ANNEXE 1 : EXTRAIT DU SCHÉMA RELATIONNEL DU SYSTÈME COMPTABLE

```
REGION (CodeRegion, NomRegion)
     CodeRegion : cle primaire
JOURNAL (CodeJournal, NomJournal).
     CodeJournal : cle primaire
COMPTE (NuméroCompte, ClasseCompte, NomCompte)
     NumeroCompte : Cle primaire
CLIENT (NumeroClient, NumroCompte#, CodeRegion#,...)
     NumeroClient : cle primaire
     NumeroCompte : cle etrangere en reference à la table COMPTE
     CodeRegion : cle etrangere en reference a la table REGION
OPÉRATION (CodeRegion#, NumeroOpération, CodeJournal#, LibelleOpération, DateOperation,
MontantOperation)
     CodeRegion + NumeroOperation : cle primaire
     CodeRegion : cle etrangere en reference a la table REGION
     CodeJournal : cle etrangere en refenrece a la table JOURNAL
LIGNE_ÉCRITURE (CodeRégion#, NuméroOpération#, NuméroLigneÉcriture, NuméroCompte#,
MontantMouvement, SensMouvement)
     CodeRegion + NumeroOperation + NumeroLigne : cle primaire
     CodeRegion : cle etrangere en reference a la table REGION
     NumeroOperation : cle etrangere en reference a la table OPERATION
     NumeroCompte : cle etrangere en reference à la table COMPTE

```

Travail à faire :

1. Proposer une répartition des données, en indiquant le nom des tables à héberger soit au siège social soit dans les centres régionaux et ce, en distinguant les tables de référence et les tables « répliquées ».

Au siège :

Tables de référence	Tables répliquées
RÉGION	OPÉRATION
JOURNAL	LIGNE_ÉCRITURE
COMPTE	
(1) CLIENT	(2) CLIENT : consolidation des données de la région

Dans les centres régionaux :

Tables de référence	Tables répliquées
(2) CLIENT de la région	(1) CLIENT : réplique partielle du siège
OPÉRATION	RÉGION
LIGNE_ÉCRITURE	JOURNAL
	COMPTE

Exercice KES

2. Indiquer les solutions qui permettent d'assurer la cohérence des données contenues dans les tables « répliquées » lorsque les données de référence sont mises à jour.

Il s'agit de choisir d'une part le mode de propagation des mises à jour (ou du rafraîchissement des copies), d'autre part les tables concernées.

1) Mise à jour des tables de référence au siège

Synchrone	Asynchrone
CLIENT	RÉGION JOURNAL COMPTE

2) Mise à jour des tables de référence dans la région

Synchrone	Asynchrone
CLIENT	OPÉRATION LIGNE_ÉCRITURE

Exercice KES

iii) Synthèse concernant l'implantation de données réparties avec réplication

1. Un segment logique (c'est une table ou partie de table) de type référence au sens strict n'est mis à jour que sur la machine chargée de le gérer (serveur ou poste de travail).
2. Un segment logique de type copie est mis à jour en temps réel ou à intervalles réguliers sous contrôle de la machine chargée de gérer le segment logique de référence. Il est également possible que la mise à jour soit effectuée à la demande de la machine qui accueille le segment logique de type copie.
3. Un segment logique de type dossier (dit également référence + copie) est à un instant référence sur une machine et copie sur d'autres.

c) Règles et recommandations pour choisir le lieu d'implantation des tables

Les données peuvent être implantées sur les différents sites selon le critère de volume ou bien selon des critères liés à leur utilisation.

i) Critères à prendre en compte

1 - Volume de données

- données volumineuses : sur le serveur,
- données peu volumineuses : sur le serveur ou sur poste.

2 - Critères liés à la l'utilisation des données

Les principaux sont :

1. le type d'utilisation :
    - données privées : données consultées et mises à jour par le site qui les accueille et inaccessibles aux autres,
    - données partagées : données consultées et mises à jour par plusieurs sites,
    - données protégées : données mises à jour par le site qui les accueille et consultables par les autres,
    - données consultables : données uniquement consultables par le site qui les accueille (c'est par exemple le cas des historiques).
2. le mode et la fréquence d'utilisation :
    - données en consultation : fréquente, peu fréquente,
    - données en mise à jour (au sens large) : fréquente, peu fréquente. Parfois, il est important d'aller jusqu'au niveau plus fin du type de mise à jour (modification, ajout, suppression).

3. le niveau de confidentialité (les données très confidentielles seront plutôt stockées sur le poste de travail habilité à les manipuler).

ii) Recommandations dans un cas simple d'architecture Client / Serveur

Implantation sur le poste de travail :

1. données privées,
2. données protégées,
3. copies de données partagées en consultation, à faible taux de mise à jour. Ces dernières sont destinées par exemple à permettre de faire des contrôles locaux, donc à réduire certains temps de réponse ou à permettre d'offrir des consultations ou des sélections locales dans des listes déroulantes.

Implantation sur le serveur :

Données partagées en mise à jour à fort taux de mise à jour.

1.3. Exemple de synthèse

L'illustration porte sur une banque qui dispose d'agences.

Les clients, les comptes et les agences sont mis à jour en agence. Une agence se charge uniquement de ses clients et de ses comptes et du tuple d'agence la concernant. Le siège consulte les clients, les comptes et les agences.

Les tarifs sont gérés par le siège et peu mis à jour. Ils sont consultés fréquemment en agence.

Les opérations bancaires sont volumineuses, elles sont souvent créées au niveau du siège, rarement en agence.

MLD global très simplifié de la gestion des comptes bancaires :

```
AGENCE (numagence, nomagence, adragence, telagence)
     numagence : cle primaire
CLIENT (numclient, nomclient, adrclient, telclient, #numagence)
     numclient : cle primaire
     numagence : cle etrangere reference table AGENCE
COMPTE (numcompte, soldecompte, typecompte, #numclient)
     numcompte : cle primaire
     numclient : cle etrangere reference table CLIENT
TARIF (typeoperation, prix)                    /* Exemple : virement */
     typeoperation : cle primaire
OPERATION (#numcompte, mois, an, numoperation, montant, ..., #typeoperation)
     numcompte + mois + an + numoperation : cle primaire
     numcompte : cle etrangere reference table COMPTE
     typeoperation : cle etrangere reference table TARIF

```

MLD local agence :

```
AGENCE (numagence, nomagence, adragence, telagence)                /* on peut la nommer agence_n (n désignant l'agence) */
     numagence : cle primaire
CLIENT (numclient, nomclient, adrclient, telclient, #numagence)    /* on peut la nommer client_n */
     numclient : cle primaire
     numagence : cle etrangere reference table AGENCE
COMPTE (numcompte, soldecompte, typecompte, #numclient)            /* on peut la nommer compte_n */
     numcompte : cle primaire
     numclient : cle etrangere reference table CLIENT
TARIF (typeoperation, prix)                                         /* Exemple : virement */
     typeoperation : cle primaire
```

MLD local siège :

```
AGENCE (numagence, nomagence, adragence, telagence)                    /* consolidation */
     numagence : cle primaire
CLIENT (numclient, nomclient, adrclient, telclient, #numagence)        /* consolidation */
     numclient : cle primaire
     numagence : cle etrangere reference table AGENCE
COMPTE (numcompte, soldecompte, typecompte, #numclient)               /* consolidation */
     numcompte : cle primaire
     numclient : cle etrangere reference table CLIENT
TARIF (typeoperation, prix)                                            /* Exemple :
virement */
     typeoperation : cle primaire
OPERATION (numcompte, mois, an, numoperation, montant, ..., #typeoperation)
     numcompte + mois + an + numoperation : cle primaire
     numcompte : cle etrangere reference table COMPTE
     typeoperation : cle etrangere reference table TARIF
```

Répartition des tables

Relation	Machine type AGENCE	Machine type SIÈGE
AGENCE	Référence pour le tuple la concernant (fragmentation sur numéro d'agence).	Copie avec MAJ périodique (tous les soirs par exemple).
CLIENT	Référence pour les clients gérés par l'agence (fragmentation sur numéro d'agence).	Copie avec MAJ périodique (tous les soirs par exemple).
COMPTE	Dossier (référence + copie) pour les comptes gérés par l'agence (fragmentation sur numéro d'agence). L'agence devient copie lors du relevé (le siège met alors le solde à jour). La MAJ est effectuée de façon asynchrone le soir.	Dossier. Le siège est site copie avec MAJ périodique (tous les soirs par exemple) sauf lors du relevé (le siège devient référence pour la MAJ du solde).
TARIF	Copie avec MAJ synchrone.	Référence.
OPERATION BANCAIRE	Non présent mais accessible en MAJ.	Référence.

2. Exercice : Vas : Kimassur

Présentation de la société ASSUR

La société Assur est spécialisée dans l'assurance automobile. Son siège social est situé à Paris et elle dispose de 3 succursales: une à Lyon, l'autre à Marseille et la troisième à Bordeaux.

La société a décidé de renforcer l'autonomie financière et comptable de ses succursales mais le siège social continuera cependant d'exercer le suivi de chacune d'entre elles.

Pour satisfaire ce besoin, le responsable informatique de la société Assur propose de mettre en place une architecture client-serveur de données distribuées. Certaines données seront "répliquées", c'est-à-dire recopiées sur les différents sites de la société, pour permettre à la fois une gestion décentralisée et une consolidation des données comptables.

- Dans les succursales, seront stockées des tables de référence, propres au site ainsi que des tables répliquées, copies des tables de référence du siège social.
- Enfin, les responsables de la société Assur souhaitent que les succursales n'agissent que sur les informations les concernant directement. Par exemple, à propos des clients, la succursale de Marseille ne pourrait consulter et mettre à jour que les données relatives aux clients de la région méditerranéenne.

Vous trouverez ci-dessous le MLD global de la base de données répartie.

MLD Global de la base de données répartie :

```
REGION (CodeRegion, NomRegion)
    CodeRegion : cle primaire
CLIENT (Numeroclient, NomClient, CodeRegion#)
    NumeroClient : cle primaire
    CodeRegion : cle etrangere reference table REGION
SINISTRE (NumSinistre, DateSinistre, NumeroClient#)
    NumSinistre : cle primaire
    NumeroClient : cle etrangere reference table CLIENT
```

A partir de la présentation de la société Assur, proposer un scénario de répartition des tables d'une base de données répartie.

Question 1

Proposer une répartition des données, en indiquant le nom des tables à héberger soit au siège social soit dans les succursales et ce, en distinguant les tables de référence et les tables "répliquées".

Question 2

En déduire le MLD réparti.

Question 3

Pour chaque table ou partie de table répliquée, indiquer le type de mise à jour à utiliser.

Références

http://docs.postgresql.fr/9.4/plpgsql.html	Fonctions en langage de procédures SQL
http://docs.postgresql.fr/9.4/plpgsql-cursors.html	Documentation PostGreSQL - Les curseurs
http://docs.postgresql.fr/9.4/plpgsql-errors-and-messages.html	Documentation PostGreSQL - L'instruction RAISE
http://docs.postgresql.fr/9.4/xfunc-sql.html	PostgreSQL - Les fonctions SQL (pur)
https://fr.wikipedia.org/wiki/Gartner	Gartner Group - Stanford