Comprendre les bases de données

Auteur : Hervé LEFEBVRE <aegir@free.fr>

9 mars 2002

Version compilée des articles parus sur <http://www.linuxfrench.net>, ce document reprends l'introduction à *Postgresql* ainsi que les 11 épisodes suivants dédiés à la compréhension des bases de données, édités entre le 10 octobre 2001 et le 4 février 2002.
Contact : christian.sahastume@wanadoo.fr

Table des matières

1 Article 1 : présentation de Postgresql

PostgreSQL est un serveur de base de données (DBMS) transactionnel très puissant. En terme de taille, les limitations techniques n'existent quasiment pas puisqu'une simple table peut contenir jusqu'à 64 téra-octets de données.

1.1 Concurrence d'accès par multi-version

PgSQL étant un système transactionnel, il se doit de gérer la concurrence d'accès. La plupart des SGBD sur la marché utilisent pour cela un mécanisme de verrouillage. Lorsqu'une donnée est accédée en lecture, un verrou partagé y est placé. Lorsqu'une donnée est accédée en écriture, un verrou exclusif y est placé. Un verrou partagé permet toujours d'accéder en lecture sur cette donnée, mais toute opération d'écriture sur cette même donnée sera bloquée en attendant que le verrou soit libéré. En revanche, un verrou exclusif bloque toute opération de lecture ou d'écriture sur cette donnée.

Ainsi, si l'on prend terminaux, et que depuis chacun d'eux on se connecte sur le SGBD. Depuis le premier terminal, effectuons les opération suivantes :

```
market=# create table toto( a integer);
CREATE

market=# insert into toto values(1);
INSERT 53526 1

market=# begin transaction;
BEGIN
market=# update toto set a=2;
UPDATE 1

market=#
```

À ce moment là, prenons le second terminal pour effectuer l'opération suivante :

```
market=# select * from toto;
```

Sur beaucoup de SGBD (par exemple sur SQL-Server), le second terminal sera alors bloqué. L'instruction SELECT restera en attente, sans renvoyer le moindre résultat. L'explication est simple, sur le premier terminal nous avons placé un verrou exclusif sur la ligne de la table TOTO avec une opération d'écriture (UPDATE), au sein d'une transaction (BEGIN TRANSACTION). Par conséquent, l'opération de lecture restera bloquée jusqu'à ce que le verrou exclusif soit libéré, ce qui arrivera à la fin de la transaction (que ce soit avec un COMMIT ou bien un ROLLBACK).

Avec postgreSQL, le second terminal ne sera pas bloqué. L'ordre SELECT aura immédiatement pour résultat :

```
a
---
 1
(1 row)
```

Cela s'explique par le mécanisme de concurrence d'accès par multi-version (Multi-Version Concurrency Control). Tant que la transaction du premier terminal n'est pas terminée, le second terminal peut consulter la version de la base de données telle qu'elle était avant que la transaction ne commence. Bien entendu, si le second terminal ouvre à son tour une transaction, et cherche à modifier la même donnée, il sera bloqué de la même manière qu'avec un SGBD classique.

1.2 Héritage, structures orientées objet

PostgreSQL est certe une base de données relationelle, mais également orientée objet. Cela permet non seulement n'avoir une structure commune à plusieurs tables, facilitant d'autant la maintenance du modèle physique, mais également d'avoir une notion de tables hierarchiques. Un court exemple vaut mieux qu'un long discours :

```
market=# create table prospect(nom varchar);
CREATE

market=# create table client (date_commande date) inherits (prospect);
CREATE

market=# insert into prospect values('LinuxFrench');
INSERT 53580 1

market=# select * from prospect;
     nom
-------------
 LinuxFrench
(1 row)

market=# select * from client;
 nom | date_commande
------+---------------
(0 rows)

market=# insert into client(nom) select nom from prospect;
INSERT 53581 1

market=# select * from client;
     nom     | date_commande
-------------+---------------
 LinuxFrench |
(1 row)

market=# select * from prospect;
     nom
-------------
 LinuxFrench
 LinuxFrench
(2 rows)

market=# select * from ONLY  prospect;
     nom
-------------
 LinuxFrench
(1 row)
```

Dans cet exemple nous avons donc une table "client" qui hérite de la table "prospect". Si nous avons une ligne dans chcune des tables, nous ne pourront obtenir que le client en interrogeant la table client, en revanche si on interroge la table prospect, nous aurons soit l'ensemble des prospects et des clients (une ligne de chaque dans notre exemple), soit uniquement les prospects en utilisant le mot-clef ONLY lors du SELECT.

1.3 Procédures stockées

PostgreSQL permet d'implémenter des procédures stockées sous forme de fonctions. Ces procédures peuvent être écrites dans différents langages :

- Le SQL "standard" (SQL92).
- Le PL/pgSQL, qui est un langage procédural directement dérivé de SQL.
- C
- Tcl
- Perl
- Python
- Ruby

Le moins qu'on puisse dire, c'est qu'il y a le choix. Pour ma part, j'utilise le PL/pgSQL qui est très proche du PL/SQL d'Oracle.

On appécie dans postgreSQL le fait de pouvoir créer plusieurs fonctions ayant le même nom, mais prenant des paramètres différents. Ainsi, il est beaucoup plus agréable et lisible d'avoir deux fonctions :

- supprimerClient(integer) supprimerClient(varchar)

plutôt que

- supprimerClientParIdentifiant(integer) supprimerClientParNom(varchar)

Cependant, une limitation regrettable (et oui, PostgreSQL n'est pas parfait !), il est impossible d'ouvrir et de fermer une transaction dans une procédure stockée. En effet, PostrgreSQL ne gère pas encore les transactions imbriquées. Si dans votre applicatif vous désirez faire appel à une procédure qui doit effectuer une série de mises à jour, vous êtes donc condamné à faire pécéder votre appel d'un "BEGIN TRANSACTION".

1.4 Triggers, contraintes

Une procédure stockée (ou fonction selon la terminologie de PostgreSQL), peut être appelée dans le cadre d'un trigger (aussi appelé déclencheur). Un trigger est une opération qui sera appelée lors d'un accès à une table. Un trigger peut être appelé dans le cadre d'un SELECT et/ou INSERT, et/ou UPDATE. L'appel peut être effectué soit pour chacune des lignes manipulées, soit une fois par ordre SQL effectué. De plus un trigger peut intervenir avant et/ou après que la manipulation des données soit réellement effective.

Les triggers sont très pratiques pour effectuer des vérifications de cohérence ou d'intégrité des données avant d'accepter une modification dans la base de données. Mais il peuvent également servir à des opérations plus fonctionnelles, par exemple archiver dans une table spécifique les données avant qu'elles ne soient modifiées afin de gérer les versions successives d'un document.

PostgreSQL gère également les contraintes. Celles-ci sont déclarées lors de la création de la table. Nous avons bien sû les contraintes classiques : NULL/NO NULL, UNIQUE, PRIMARY KEY, et REFENCES (clef étrangère) mais également CHECK qui est moins courante. Cette contrainte permet d'écrire une expression booleenne qui acceptera ou n'acceptera pas les nonnées. Par exemple on peut vouloir vérifier qu'il y a au minimum un des 2 champs d'une ligne qui sont renseignés (TEL_FIXE ou TEL_GSM par exemple).

En déclarant une contrainte de clef étrangère, on peut également spécifier le comportement en cas de suppression de la donnée référencée (refuser, suppression en cascade, passage de la clef étrangère à NULL ou bien

à la valeur par défaut définie pour cette colonne).

Essayez-le !

PostgreSQL est une base de données très puissante. J'ai personellement travaillé plusieurs années avec des SGBDR propriétaires (Oracle et Sybase) sur de très grosses bases de donnée dans des domaines variés (Bancaire, système de réservation, base documentaire...), et je dois dire qu'il ne manque plus qu'une seule chose à pgSQL pour se frotter à des applications aussi critiques : les transactions imbriquées dans les procédures stockées. On peut aussi lui trouver quelques manques, par exemple les procédures stockées ne peuvent pas renvoyer des tuples. Mais la même limitation existe avec Oracle. Il y a également les tables temporaires de session (tables #xxx de Sybase) qui sont absentes. Elles n'existent pas non plus dans Oracle, et y sont - sous Oracle tout comme pgSQL - avantageusement remplacées par des mécanismes de sous-requêtes qui sont absents de Sybase.

En terme de performances, on peut dire que globalement pgSQL tient très bien la charge. Il est tout à fait adapté à un serveur Intranet par exemple.

2 Episode 1 : introduction, tables, lignes et colonnes

Les bases de données... Elles envahissent notre vie privée, elles deviennent incontournables en informatique, alors voici une série d'articles qui a la prétention de vous former à la compréhension, utilisation et conception des bases de données. Les articles seront publiés au rythme de un par jour ou tous les deux jours.

2.1 Un peu de vocabulaire

Comme tout le monde le sait, le jargon technique à pour principale conséquence (intérêt ?) de rendre inaccessible une technologie aux non-initiés. Alors voici une base permettant de décoder - voire décrypter - le domaine des bases de données.

Une base de données (BD) - database ou DB en anglais - est un système informatique permettant d'écrire, de socker et d'accéder à des données.

Fondamentalement, la notion de base de données est donc très vaste. Par exemple, si vous utilisez un fichier de traitement de texte pour y noter vos rendez-vous, il s'agit là d'une base de données. De même, votre disque dur est une base de données. On distingue cependant les bases de données **structurées** des autres. Votre fichier de traitement de texte n'est à priori pas structuré, pas plus que ne l'est votre disque dur (encore que cela peut se discuter). Par contre, quand vous envoyez un email, on peut considérer que votre mailer se comporte comme une base de données structurée, puisque qu'il va archiver les emails tout en stockant dans des champs destinés à cet effet la date, l'expéditeur, le destinataire, le texte etc.

Un système de gestion de base de données - SGBD ou DBMS en anglais - fait référence à un système informatique expressément conçu pour la gestion des données la plupart du temps structurées, à l'exception notable des SGBD à vocation documentaire (données non structurées), qui ne feront pas l'objet de cette série d'articles.

Les bases de données relationnelles sont celles qui établissent des relations entre différentes entités. Par exemple, si vous avez un dictionnaires et un atlas, et que dans le dictionnaire vous y lisez « Le Louvre : musée français situé à paris cf : atlas, Paris, p. 230 », vous avez là une relation entre deux entités (le dictionnaire et l'atlas), puisqu'en lisant une donnée dans le dictionnaire vous avez récupéré des informations vous permettant d'accéder à une donnée dans l'autre entité : l'atlas. Signalons que les bases de données relationnelles ne sont pas les seules existantes. Nous avons aussi les bases de données hiérarchiques (quasiment disparues aujourd'hui, surtout depuis qu'il a été démontré que tout ce qui peut être fait en hiérarchique peut l'être également

en relationnel). Il y a aussi les bases de données objets, censées révolutionner les SGBD, mais qui peinent à émerger. Nous n'évoquerons dans cette série d'articles que les bases de données relationnelles. Nous ne ferons qu'avoir un aperçu des capacités orientées objets de pgSQL. Un système informatique permettant de gérer des bases de données relationnelles est appelé SGBDR ou RDBMS en anglais.

Pour accéder aux données dans un SGBDR, on utilise en général un langage nommé SQL (Structured Query Language ou Langage Structuré de Requête). Le SQL est un langage **ensembliste**. Il s'agit en fait d'une transposition informatique des opérations algébriques sur les ensembles. Dans un SGBDR les données sont effectivement des ensembles, et pour y accéder on peut dire par exemple « Je veux **l'union** entre les clients qui ont achetés plus de 3 fois cette année, et ceux qui ont dépensés au total pour plus de 10 000 FRF ». Ce qui serait complètement différent de « **l'intersection** entre les clients qui ont achetés plus de 3 fois cette année, et ceux qui ont dépensés au total pour plus de 10 000 FRF » (dans un cas nous avons un OU, dans l'autre un ET).

2.2 Les tables

Dans un SGBDR, les données sont stockées dans des **tables**, voici un exemble :

```
  lact_code | lact_lib
------------+-------------------------------------------
         15 | Industries alimentaires
         16 | Industrie du tabac
         17 | Industrie textile
         18 | Industrie de l'habillement et des fourrures
         19 | Industrie du cuir et de la chaussure
```

Dans cette table nous avons 5 lignes (on parle aussi de **tuples**), et deux colonnes portant le nom de lact_code et de lact_lib. Au lieu de parler de colonnes, on peut aussi parler de **champs**.

Chaque colonne possède un type particulier, correspondant aux données qui y seront stockées. Ici, le champ lact_code est d'un type numérique entier, tandis que lact_lib est d'un type chaîne de caractères.

2.3 NULL, une valeur à part

Il y a dans les bases de données une valeur particulière qui s'appelle **NULL**. Null signifie « indéterminé ». Attention, cela ne signifie pas « vide », c'est complètement différent.

Par exemple, si vous avez dans une table un champ **nombre_enfants**, si vous y mettez 0, c'est que vous n'avez pas d'enfants, si vous ne renseignez pas ce champ, alors il contiendra la valeur **NULL**.

De même, si une table contient le champ **surnom** . Si le champ contient la valeur NULL, c'est qu'on ne sait pas si le personne à un surnom. Par contre si le champ est vide (c'est à dire s'il contient la chaîne vide : « »), cela signifie que la personne n'a pas de surnom. Il faut bien faire attention à ne pas confondre la chaine vide, et NULL, ce sont deux valeurs complètement différentes [1] (déterminée dans un cas, indéterminée dans l'autre).

La valeur NULL, n'a pas de type. Elle ne peut pas être impliquée dans la moindre opération mathématique puisque sa valeur est indéterminée. L'opération aura donc un résultat indéterminé. 1 + NULL donne NULL, la concaténation d'une chaîne de caractères avec NULL donne également NULL. En quelque sorte, NULL est un élément absorbant pour la quasi-totalité des opérations.

```
martin=> SELECT * FROM eleves;
id |  nom   |  prenom  | age | classe
----+--------+----------+-----+--------
```

```
 5 | MARTIN | Philippe |     |
 6 | TUX    | Family   |     |
(2 rows)
```

3 Episode 2 : Installation de postgreSQL, CREATE TABLE, SELECT, INSERT, DELETE.

L'article précédent était passablement théorique. Nous allons désormais passer à un apprentissage pratique. Pour cela, il vous faut un SGBDR opérationel sur votre machine.

3.1 Installer postgreSQL

Nous allons parler ici de l'installation sur une Mandrake 8, si vous êtes équipé d'une autre distribution, vous pouvez soit utiliser le paquetage spécifique à votre distribution, soit télécharger les sources et les compiler.

Avec une mandrake, l'installation est particulièrement simple. Si PostgreSQL n'est pas déjà intallé, un simple **urpmi postgresql** devrait suffire à installer tout les éléments nécessaires.

Après l'installation, le plus simple est d'effectuer un redémarrage de la machine afin que le service soit lancé correctement.

Ensuite, en tant que superutilisateur, taper la commande **su - postgres** , vous serez ainsi identifié comme utisateur ayant les droits d'administration du SGBDR. Il vous faut désormais vous créer un utilisateur de base de données, et une base de données. Attention, le SGBDR possède sa propre gestion des utilisateurs. Donc, si votre nom d'utilisateur Linux est **martin** il vous faut également créer un utilisateur **martin** dans postgresql. Cela s'effectue avec la ligne de commande : **createuser** martin. Si yous répondez **Y** aux deux questions qui vous sont posées à ce moment là, ce nouvel utilisateur aura également le droit de créer des utilisateurs et des bases de données.

La création de la base de données se fait très simplement avec la commande **createdb martin**. Nous avons donc désormais un utilisateur et une base de données tout deux nommés **martin**.

Ensuite, lorsque vous êtes en ligne de commande en tant qu'utilisateur Linux martin, vous pouvez vous connecter au SGBDR avec la commande **psql**, vous serez connecté en tant qu'utilisateur **martin**, et dans la base de données **martin** . Psql est un client permettant d'envoyer des ordres SQL à la base de données. Vous pouvez quitter à tout moment psql pour revenir au shell avec les touches CTRL+D.

3.2 Création de table

Une table est créée avec l'instruction SQL **CREATE TABLE**. Si nous voulons créer une table *ELEVES*, comportant 3 colonnes : le nom, le prenom et l'age, voici comment s'y prendre :

```
martin=# CREATE TABLE eleves (
martin(# nom varchar,
martin(# prenom varchar,
martin(# age integer);
CREATE
martin=#
```

La commande aurait pu être tapée sur une seule ligne peu importe, psql n'envoie l'ordre SQL au SGBD que lorsqu'il rencontre le « ; » qui marque la fin de l'instruction SQL. Le nom de la table à créer (eleves) suit immédiatement l'ordre CREATE TABLE, puis la liste des colonnes est spécifiée entre parenthèses. Pour

chaque colonne, on donne le nom et le type. Voici donc 2 exemples de type : **integer** qui est un entier signé sur 32 bits, et **varchar** qui signifie « chaîne de caractères de longueur variable ».

Oh, un dernier détail, avec postgreSQL aucune différence n'est faite entre les majuscules et les minuscules dans les ordres SQL. On aurait donc pu taper **Create Table EleVeS**, cela n'aurait rien changé.

3.3 Ajouter et consulter des données

Pour mettre des données dans notre table, on utilise l'instruction **INSERT INTO**.

```
martin=# INSERT INTO eleves VALUES('DUPONT','Martin',25);
INSERT 53638 1
martin=#
```

La valeur 53638 qu'affiche psql lorsque l'ordre est exécuté correspond à **l'oid** attribué à cette nouvelle ligne (l'oid est le numéro d'identification unique d'un objet dans pgSQL). Le numéro que vous obtiendrez sera certainement différent. Le chiffre 1 qui se trouve après l'oid indique que l'instruction INSERT a affecté une seule ligne.

On peut désormais consulter les données avec l'ordre SELECT :

```
martin=# SELECT nom FROM eleves;
  nom
--------
 DUPONT
(1 row)

martin=# SELECT nom, prenom FROM eleves;
  nom   | prenom
--------+--------
 DUPONT | Martin
(1 row)

martin=# SELECT * FROM eleves;
  nom   | prenom | age
--------+--------+-----
 DUPONT | Martin |  25
(1 row)

martin=#
```

L'argument « ***** » est bien pratique puisqu'il permet de sélectionner d'un seul coup toutes les colonnes. Cependant il est déconseillé de l'utiliser dans le code d'une application. Il va en effet renvoyer les colonnes dans l'ordre ou elles ont été déclarées. À la moindre évolution du schéma de votre base de données vous risqueriez d'avoir des dysfonctionnements liés à l'utilisation de cet argument.

Un SELECT peut également servir à effectuer des opérations. Par exemple, l'opérateur « || » sert à effectuer une concaténation entre deux chaînes de caractères.

```
martin=# SELECT nom||' '||prenom AS np, nom AS name FROM eleves;
      np       |  name
---------------+--------
 DUPONT Martin | DUPONT
```

```
(1 row)

martin=#
```

Ainsi que vous pouvez le voir, on peut changer le nom de référence d'une colonne avec **AS** , cela est bien pratique lorsqu'on effectue des opérations complexes ou bien lorsqu'on veut faire référence à la valeur d'une colonne avant qu'elle n'ait été modifiée par une opération. Dans le jargon on appelle cela des *ALIAS* de colonnes.

3.4 Opération ensembliste

Ainsi que je l'ai déjà dit, SQL permet de manipuler des ensembles. Un exemple ? Et bien nous avons vu les les instructions INSERT et SELECT. Le SELECT retourne un ensemble de données issues d'une table, tandis que l'INSERT pertmet dajouter un ensemble de données dans une table. Un petit exemple, et les choses seront plus claires :

```
martin=# SELECT * FROM eleves;
  nom   | prenom | age
--------+--------+-----
 DUPONT | Martin |  25
(1 row)

martin=# INSERT INTO eleves SELECT * FROM eleves;
INSERT 53639 1

martin=# SELECT * FROM eleves;
  nom   | prenom | age
--------+--------+-----
 DUPONT | Martin |  25
 DUPONT | Martin |  25
(2 rows)

martin=#
```

Vous avez compris ? Nous venons d'utiliser l'ensemble des données qui ont été retournées par un SELECT comme argument d'un INSERT. Comme le SELECT ne retournait qu'une seule ligne, nous avons ajouté une deuxième ligne dans notre table. Si nous recommencions une seconde fois la même opération, nous obtiendrions 4 lignes identiques dans la table eleves.

3.5 Faire le ménage

Comment supprimer les données de notre table ? Tout simplement avec une instruction DELETE :

```
martin=# DELETE FROM eleves;
DELETE 2

martin=\#
```

Voilà, cette instruction SQL vient d'effacer sauvagement les 2 lignes que contenait notre table ! Il peut être également utile de supprimer définitivement la table du schéma du SGBD, il faudra alors la créer à nouveau avec un CREATE TABLE :

```
martin=# DROP TABLE eleves;
DROP

martin=\#
```

4 Episode 3 : WHERE et UPDATE.

Jusqu'à présent, nous avons manipulé des tables dans leur intégralité. Nous allons apprendre dans cet article à ne sélectionner que quelques lignes précises dans une table.

4.1 La clause WHERE

La clause **WHERE** est utilisable avec la plupart des instructions SQL (mis à part INSERT, car cela n'aurait pas de sens !). Elle permet de caractériser les éléments qui constituerons le sous-ensemble qui sera manipulé par l'instruction SQL.

La clause WHERE est suivie d'une expression booleenne. Si elle est vérifiée, le tuple fera partie du sous-ensemble, sinon il en sera exclus. Voici une première utilisation du WHERE :

```
martin=# CREATE TABLE eleves(nom varchar, prenom varchar, age integer,
classe char(4));
CREATE

martin=# INSERT INTO eleves VALUES('DUPONT', 'Martin', 6, 'CP');
INSERT 53698 1

martin=# INSERT INTO eleves VALUES('DURAND', 'Theo',15, '3A');
INSERT 53699 1

martin=# INSERT INTO eleves VALUES('DUPOND', 'Léa',12, '6B');
INSERT 53700 1

martin=# SELECT {*} FROM eleves WHERE age > 10;
  nom   | prenom | age | classe
--------+--------+-----+--------
 DURAND | Theo   |  15  | 3A
 DUPOND | Léa    |  12  | 6B
(2 rows)

martin=#
```

Tout d'abord, nous découvrons un nouveau type lors de la création de la table : **char(4)** . Cela signifie que ce champ contiendra une chaine d'une longueur exacte de caractères. Si on essaye d'un mettre une chaine plus courte, '6B' par exemple, le SGBD ajoutera automatiquement 2 espaces pour compléter la chaîne.

Ensuite, après avoir inséré 3 lignes dans notre table, nous effectuons un SELECT doté d'une clause WHERE : **age > 10** .

Cela signifie tout simplement que notre instruction SQL va sélectionner un sous-ensemble de données dans lequel le champ age contient une valeur strictement supérieur à 10.

Beaucoup d'opérateurs logiques sont disponibles. Ainsi l'opérateur LIKE permet de faire un test sur une partie d'une chaine :

```
martin=# SELECT {*} FROM eleves WHERE nom LIKE '%PO%';
   nom   | prenom | age | classe
--------+--------+-----+--------
 DUPONT | Martin |   6 | CP
 DUPOND | Léa    |  12 | 6B
(2 rows)

martin=# SELECT * FROM eleves WHERE nom LIKE 'DU%' AND classe='CP '
  nom   | prenom | age | classe
--------+--------+-----+--------
 DUPONT | Martin |   6 | CP
(1 row)
```

4.2 UPDATE

Maintenant que nous savons désigner précisément une ligne, nous pouvons utiliser l'instruction UPDATE pour effectuer des mises à jour. Par exemple, si nous voulons corriger l'age et la classe de l'élève DUPONT :

```
martin=# UPDATE eleves SET age=16, classe='3A' WHERE nom='DUPONT';
UPDATE 1

martin=# select * from eleves;
  nom   | prenom | age | classe
--------+--------+-----+--------
 DURAND | Theo   |  15 | 3A
 DUPOND | Léa    |  12 | 6B
 DUPONT | Martin |  16 | 3A
(3 rows)
martin=#
```

4.3 DISTINCT et INTO

Profitons de l'état dans lequel se trouve la base de données pour voir deux nouvelles clauses à l'instruction SELECT. Si nous voulons avoir la liste des classes ayant des élèves, il nous faut un moyen pour ne selectionner qu'une seule fois chaque valeur présente de la colonne classe. C'est le but de la clause DISTINCT :

```
martin=# SELECT classe FROM eleves; classe
--------
3A
6B
3A
(3 rows)

martin=# SELECT DISTINCT classe FROM eleves;
classe
--------
3A
6B
(2 rows)
```

Ainsi, si nous voulons créer une nouvelle table contenant la liste des classes, rien n'est plus simple. Nous pouvons utiliser la clause INTO qui va créer une table afin d'y stocker l'ensemble de tuples ramené par l'instruction SELECT. La clause INTO va se comporter un peu comme le redirecteur > en ligne de commandes Linux :

```
martin=# SELECT DISTINCT classe INTO liste_classes FROM eleves;
SELECT

martin=# select * from liste_classes;
classe
--------
3A
6B
(2 rows)
```

Ainsi que vous pouvez le voir, la base de données à automatiquement créé la table « liste_classes » afin s'y stocker le résultat de notre SELECT DISTINCT. Signalons qu'il est impératif que la table liste_classe n'existe pas avant d'exécuter cette commande, faute de quoi nous aurions une erreur « Relation 'liste_classes' already exists »

5 Episode 4 : indexes, clefs et jointures.

Nous avons appris à créer et manipuler des ensembles et sous-ensembles de données. C'est bien, mais nous parlons dans ces articles de SGBDR, et le « R » signifie « Relationnel ». Nous devons donc apprendre à établir des relations.

5.1 Les index

Les index ont pour but principal d'accélérer l'accès aux données. En effet, si vous recherchez une valeur dans une table (par exemple la liste des élèves ayant un age de 10 ans), avec la requête suivante :

```
SELECT * FROM eleves WHERE age=10 ;
```

Le SGBDR va devoir parcourir l'ensemble de la table et d'en passer en revue chaque ligne afin d'en extraire celles qui répondent à la condition spécifiée dans la clause WHERE.

Cela n'est pas gênant lorsque la table ne contient que quelques dizaines de lignes, mais le temps de recherche va être directement proportionnel au nombre de lignes contenues dans la table. Si vous doublez le nombre de lignes dans la table, alors le temps d'exécution de la requête sera doublé. Si la clause FROM contient 2 tables (nous verrons cela un peu plus loin dans cet article), et que la volumétrie de la base de données à doublé, alors le temps d'exécution de la requête sera multiplié par 4 !

Autant dire que ces perspectives ne sont pas satisfaisantes pour une base de données. Heureusement on peut placer des index sur un ou plusieurs champs d'une table. Cela fonctionne sur le même principe que l'index d'un livre, plutot que de lire tout le livre à la recherche d'une information, vous cherchez l'entrée correspondante dans l'index qui vous indiquera directement le numéro de la page à lire.

Un index sous postgreSQL est en général un **B-TREE** , c'est à dire un arbre équilibré. Signalons que PostgreSQL permet d'autres types d'index, mais le type par défaut est le B-TREE. Le SGBD va donc stocker une représentation arborescente des valeurs. À chaque embranchement (noeud), vous avez une valeur. Si la valeur que vous recherchez est supérieure, vous prenez l'embrachement de droite, sinon celui de gauche. Les feuilles de l'arbre donnent directement l'adresse de la ligne concernée. Ainsi, avec une table de 1000 enregistrements,

si l'arbre est binaire et équilibré, vous obtiendrez la(les) ligne(s) recherchée en seulement 10 comparaisons (2 puissance 10 = 1024) au lieu des 1000 nécessaires s'il n'y a pas d'index. Si vous doublez le nombre de lignes de la table, vous ne doublerez pas le nombre de comparaison comme précédemment. Il suffira d'ajouter un « étage » à l'arbre, seule une 11ème comparaison sera nécessaire. La progression du temps d'exécution par rapport à la volumétrie est donc devenue logarithmique, ce qui est beaucoup plus satifaisant.

Pour créer un index sur notre table **eleves** afin d'accélerer les recherches sur l'age, on utilise l'instruction CREATE INDEX :

```
martin=# CREATE INDEX idx_eleve_age ON eleves(age);
CREATE

martin=#
```

Un index peut également porter sur plusieurs colonnes. Dans ce cas, la valeur de la clef de l'index qui sera calculée par le SGBD sera en quelque sorte la concaténation de la valeur des deux colonnes. Par exemple pour créer un index portant sur le nom **et** le prénom :

```
CREATE INDEX idx_eleve_nom ON eleves(nom,prenom);
```

Il faut être prudent et réfléchir un minimum avant de créer un index sur plusieurs colonnes. Dans notre exemple, si nous faisons une recherche sur le nom de l'élève, l'index sera utilisé et la recherche sera rapide. Il en sera de même bien sûr si la recherche porte sur le nom **et** le prenom (et encore, il faut que le nom recherché soit complet). Cependant, pour une recherche sur le prénom l'index ne sera **pas** utilisé. En effet, comme je l'ai dis, la clef d'index calculée provient de la concaténation de la valeurs des colonnes dans l'ordre où elles sont indiquées dans l'instruction CREATE INDEX. Il faut donc placer ces colonnes par ordre décroissant d'importance.

Un index peut servir à vérifier l'unicité d'une valeur. On parle alors de "clef". Par exemple si on suppose qu'il n'existe pas d'homonymes, on peut créer un index unique sur le NOM+PRENOM :

```
CREATE UNIQUE INDEX pk_eleve_nom ON eleves(nom,prenom);
```

Maintenant que cet index est créé, toute tentative d'insérer deux fois le même élève dans la table se soldera par un échec.

Avant de passer à la suite, il est important de signaler que, si les index accélèrent l'accès aux données, les opérations d'écritures sont quand à elles ralenties puisque le SGBD devra maintenir l'arbre équilibré.

5.2 Jointures

Créons nous une nouvelle table "professeurs" :

```
martin=# CREATE TABLE professeurs(classe char(4), nom varchar);
CREATE

martin=# INSERT INTO professeurs VALUES('3A','M. Hiboo');
INSERT 5053839 1

martin=# INSERT INTO professeurs VALUES('6B','Mme Grenouille');
INSERT 5053840 1
```

Si nous voulons la liste des élèves, avec le nom du professeur de leur classe, la première (mauvaise) idée serait de faire :

```
martin=# SELECT * FROM eleves, professeurs;
  nom   | prenom | age | classe | classe |   nom
--------+--------+-----+--------+--------+---------------
 DURAND | Theo   |  15 | 3A     | 3A     | M. Hiboo
 DURAND | Theo   |  15 | 3A     | 6B     | Mme Grenouille
 DUPOND | Léa    |  12 | 6B     | 3A     | M. Hiboo
 DUPOND | Léa    |  12 | 6B     | 6B     | Mme Grenouille
 DUPONT | Martin |  16 | 3A     | 3A     | M. Hiboo
 DUPONT | Martin |  16 | 3A     | 6B     | Mme Grenouille
(6 rows)

martin=#
```

Que c'est-il passé ? Et bien nous avons effectué un produit cartésien entre les deux ensembles que constituent les tables. C'est à dire que pour chaque ligne de la table élève, et chacune de la table professeur, le **select** nous a ramené un tuple constitué de l'élève et du professeur. Nous avons donc obtenus toutes les combinaisons possibles entre les élèves et les professeurs ! C'est intéressant mais ce n'est pas du tout ce que nous voulions.

L'opération que nous désirons effectuer, s'appelle une jointure. Pour chaque élève, nous désisons le professeur dont la classe est celle de l'élève. Ce type de jointure est appelée « jointure interne ». Nous avons deux moyens de la réaliser, le premier est d'utiliser une clause WHERE spécifiant l'égalité entre le champ classe des deux tables :

```
martin=# SELECT eleves.nom, eleves.classe, professeurs.nom
martin-# FROM eleves, professeurs
martin-# WHERE eleves.classe=professeurs.classe;
  nom   | classe |      nom
--------+--------+---------------
 DURAND | 3A     | M. Hiboo
 DUPONT | 3A     | M. Hiboo
 DUPOND | 6B     | Mme Grenouille
(3 rows)
martin=#
```

L'autre méthode est d'utiliser le mot-clé **JOIN** . Cette méthode est préférable, afin de permettre les jointures externes (expliquées plus loin). Elle impose cependant que le champ servant de clef dans chacune des tables porte le même nom. Voici comment faire :

```
 martin=# SELECT eleves.nom, eleves.classe, professeurs.nom
 martin=# FROM eleves JOIN professeurs USING (classe);
 nom   | classe |      nom
--------+--------+---------------
 DURAND | 3A     | M. Hiboo
 DUPONT | 3A     | M. Hiboo
 DUPOND | 6B     | Mme Grenouille
(3 rows)

martin=#
```

Nous avons (heureusement) le même résultat qu'avec la jointure exprimée dans la clause WHERE. Voici ce qu'on appelle une base de données **relationnelle**, des ralations sont établies entre les différentes tables, en se basant sur des clefs.

Maintenant, nous allons ajouter un nouvel élève sans lui affecter de classe (champ classe à NULL), puis nous allons faire un jointure interne comme précédemment, puis une jointure externe :

```
martin=# INSERT INTO eleves VALUES('FAMILY','Tux',10,null);
INSERT 5053897 1
martin=# SELECT eleves.nom, eleves.classe, professeurs.nom
martin=# FROM eleves JOIN professeurs USING (classe);
nom   | classe |      nom
--------+--------+----------------
 DURAND | 3A     | M. Hiboo
 DUPONT | 3A     | M. Hiboo
 DUPOND | 6B     | Mme Grenouille
(3 rows)

martin=# SELECT eleves.nom, eleves.classe, professeurs.nom
martin=# FROM eleves LEFT OUTER JOIN professeurs USING (classe);
  nom   | classe |      nom
--------+--------+----------------
 DURAND | 3A     | M. Hiboo
 DUPONT | 3A     | M. Hiboo
 DUPOND | 6B     | Mme Grenouille
 FAMILY |        |
(4 rows)

martin=#
```

Avec la jointure interne, le nouvel élève n'apparaît pas dans la liste, car il n'a pas de professeur. Avec la jointure externe à gauche, par contre l'élève apparaît bien. Le terme **LEFT** (à gauche) signifie que c'est la table se trouvant à gauche de l'expression (eleves) qui est susceptible d'avoir des clefs à NULL.

6 Episode 5 : Les aggrégats.

Nous avons déjà un aperçu assez complet de la manipulation des ensembles et relations. Cependant, un SGBDR peut faire mieux que ramener des données brutes. Il peut les manipuler, grouper, compter etc.

Un erreur courante de la part des débutants en SGBD, est d'effectuer dans leur application un simple *SELECT*, et ensuite à l'aide de boucles (en PHP, en C, ... peu importe) et effectuent des calculs sur les lignes ramenées par le *SELECT*.

En effet, tout ce qui peut être effectué par le SGBD de manière ensembliste doit l'être. Un SGBD est une merveille d'optimisation. Il est doté de pleins d'artifices pour accéder aux données pertinentes le plus rapidement possible. Même si vous programmez en assembleur, vous ne compterez pas le nombre de lignes ramenées par un SELECT aussi vite que peut le faire le SGBD. C'est pourquoi, les aggrégats sont si importants. Ils font souvent la différence entre une application médiore et une autre rapide. Voici la fonction d'aggrégat la plus simple : *COUNT*.

```
aegir=> SELECT COUNT(1) FROM eleves;
count
-------
    3
(1 row)
```

```
aegir=>
```

Que signifie le « **1** » qui se trouve en paramètre du **COUNT** ? Et bien il s'agit tout simplement l'expression dont il faut caompter le nombre de lignes. Si je fais :

```
aegir=> SELECT 1 FROM eleves;
?column?
----------
        1
        1
        1
(3 rows)
```

Notre SELECT nous a ramené la valeur 1 pour chacune des lignes présentes dans la table. De la même manière que **COUNT(1)**, nous aurions pu faire **COUNT(nom)**. Attention cependant à la bonne vieille valeur NULL qui, elle, n'est pas comptée :

```
aegir=> insert into eleves values('TUX','Family',null,null);
INSERT 14284416 1
aegir=> SELECT COUNT(nom) FROM eleves;
 count
-------
     4
(1 row)

aegir=> SELECT COUNT(classe) FROM eleves;
count
-------
     3
(1 row)

aegir=>
```

6.1 Grouper les aggrégats

Si au lieu de compter bêtement le nombre de lignes présentes dans la table eleves, nous souhaitons compter le nombre d'éleves par classe, rien n'est plus facile. Il suffit de grouper notre aggrégat par classe avec la clause **GROUP BY**.

```
 aegir=> select count(1) as total,classe from eleves group by classe;
total | classe
-------+--------
     2 | 3A
     1 | CP
     1 |
(3 rows)
```

Nous pouvons éliminer les élèves qui n'ont pas de classe avec un clause WHERE :

```
aegir=> select count(1) as total,classe from eleves
aegir=> where classe IS NOT NULL group by classe;
total | classe
-------+--------
```

```
     2 | 3A
     1 | CP
(2 rows)
```

Et enfin nous pouvon spécifier quels sont les groupes qui nous intéressent avec la clause **HAVING**. Par exemple si seules les classes comportant plus d'un élève nous intéressent :

```
aegir=> select count(1) as total,classe from eleves
aegir=> where classe IS NOT NULL group by classe HAVING count(1) > 1;
total | classe
-------+--------
     2 | 3A
(1 row)
```

Attention à ne pas confondre WHERE et HAVING. WHERE sert à spécifier sur quel ensemble de lignes nous désirons travailler, tandis que HAVING sert à déterminer les groupes que nous voulons.

Il existe beaucoup d'autres fonctions d'aggrégat. Signalons **SUM(colonne)** qui va faire la somme d'une colonne pour un groupe, MAX et MIN pour trouver les valeurs minimales et maximales. Par exemple **select max(age), min(age) from eleves group by classe ;** retournera les ages des élèves les plus vieux et les plus jeunes de chaque classe. D'autres telles que AVG() permettent de faire quelques calculs statistiques (moyenne arithmétique dans ce cas).

7 Episode 6 : Les transaction

Les transactions sont une notion absolument indispensable en SGBD. Elles garantissent que l'état des données est toujours cohérent, quelles que soient les circonstances.

Un transaction peut se définir (de manière simpliste) comme étant une suite d'opérations qui s'effectuent soit totalement, soit pas du tout, mais jamais partiellement. Ainsi quand vous allez chez votre boulanger pour y acheter votre baguette, vous effectuez une transaction constituée de 2 opérations : vous donnez l'argent au boulanger, et lui vous donne une baguette. On peut même considérer qu'il y a une troisième opération s'il doit vous rendre la monnaie. Si jamais une des opérations se déroule mal (vous avez oublié votre porte-monnaie, ou bien quand vous voyez la gueule de la baguette qu'il vous présente vous changez d'avis...) alors la totalité des opérations est annulée (« finalement, reprenez votre baguette »). Il n'est pas envisageable que seulement une partie des opérations se déroulent (vous ne repartirez pas avec votre baguette si vous ne payez pas).

Le mot **transaction** a une connotation financière, cependant en SGBDR les transactions couvrent toutes les manipulations qui nécessitent plusieurs opérations. Par exemple, si vous voulez transférer un ancien élève de la table eleve vers la table diplome , vous devrez d'abord insérer l'élève dans cette nouvelle table, puis le supprimer de la table eleves. Il n'est pas envisageable qu'à cause d'un crash du serveur par exemple, la seconde opération ne soit pas effectuée. Notre base de données deviendrait incohérente, il faut absolument que les 2 opérations s'exécutent totalement ou pas du tout.

D'un point de vue pratique, c'est très simple. Une transaction se commence par l'instruction **BEGIN TRANSACTION** , ensuite toutes les modifications effectuées dans la base de données ne sont visibles qu'à l'intérieur de cette même transaction. Les modifications ne deviendront effectives dans la base de données qu'après que la transaction soit validée avec l'ordre **COMMIT**.

Si dans vos traitements une anomalie apparaît (erreur SQL, solde insuffisant sur un compte pour effectuer un virement etc.), alors vous pouvez annuler toutes les modifications effectuées en terminant la transaction par

l'ordre **ROLLBACK** .

Concrètement, comment-cela se passe-t-il ?

Les instructions de modifications de la base de données sont journalisées (écrites au fur et à mesure) dans *un journal de transactions*. Aucune modification n'est effectuée dans la base par le SGBD tant que l'opération n'a pas été **physiquement** écrite dans le journal de transaction. J'insiste sur le **physiquement** . Je veux dire par là que le SGBD ne s'est pas contenté d'ecrire dans le journal, il a également vidé les buffers, les caches etc.

Voici un exemple de transaction :

```
aegir=> CREATE TABLE TEST(A INTEGER);
CREATE

aegir=> BEGIN TRANSACTION;
BEGIN

aegir=> INSERT INTO TEST VALUES(1);
INSERT 14284476 1

aegir=> SELECT * FROM TEST;
a
---
1
(1 row)

aegir=> ROLLBACK;
ROLLBACK

aegir=> SELECT * FROM TEST;
a
---
(0 rows)
```

La transaction s'étant terminée par un ROLLBACK, les modifications inscrites dans le journal de transactions n'ont pas été répercutées sur la base de données.

Il faut noter que certaines instructions, telles que le *TRUNCATE TABLE* et *DROP TABLE* ne sont pas journalisées. Par conséquent, impossible de faire un ROLLBACK dessus. PostgreSQL journalise cependant le CREATE TABLE, ce qui n'est pas le cas de tous les SGBDR.

Il faut noter également que lorsque vous ouvrez une transaction, toutes les opérations de modifications que vous y effectuez seront journalisées. Si ces opérations sont volumineuses, le journal va donc grossir. Ainsi, un UPDATE qui porte sur une table de 15 millions de ligne ne va pas modifier la taille de la base de données. Les lignes de la table étant déjà présente, on se contente de modifier la valeur d'un colonne. Cependant, il faut bien journaliser les opérations effectuées. Donc le journal va grossir, jusqu'à ce que la transaction soit terminée. Si notre instruction UPDATE à pris 10 minutes à s'exécuter, il y a fort à parier qu'un ROLLBACK en fin de transaction prendra également 10 minutes, puisqu'il va falloir « dérouler » le journal depuis le début de la transaction pour en annuler les instructions.

À noter, 2 possibilités intéressantes qu'apporte la journalisation (mais qui à ma connaissance n'est pas encore proposé dans pgSQL, mais cela doit être assez simple à coder pour ceux qui en ont besoin) :

- La sauvegarde au fil de l'eau. Plutôt que de sauvegarder une base tous les jours, il peut être intéressant de la sauvegarder en permanence pour être certain de ne jamais perdre de données. La méthode est de sauvegarder sa base de données à un instant T, puis d'envoyer sur une bande ou un disque de secour tout ce qui arrive dans le fichier de log. Ainsi, il est possible à tout moment de reprendre la sauvegarde T, puis d'y appliquer les opérations de modifications qui y ont été effectuées.
- La réplication. Cela consiste à avoir plusieurs serveurs distincts qui contiennent en permanence une base de données identique, dans un souci de performance, équilibrage de charge, ou de haute disponibilité. Pour cela, le principe est un peu le même que pour la sauvegarde au fil de l'eau. Les serveurs envoient aux autres les ajouts effectuées dans leur journal de transactions.

Au sujet des mécanismes de verrouillage, je renvoie le lecteur à ce précédent article. (Voir 1er article))

8 Episode 7 : Contraintes et séquences.

Pour pouvoir établir des relations entre des tables, il faut établir un schéma de base de données adéquat, propre, et structuré. Les contraintes aident beaucoup pour cela.

Tout d'abord, reprenons notre bonne vieille table "eleves". On peut vouloir interdire l'insertion dans la base de données d'un élève dont le nom est indéterminé. Pour cela, à la création de la table, nous pouvons mettre tout simplement la **contrainte de colonne** : NOT NULL.

```
martin=> CREATE TABLE eleves(
martin(> nom VARCHAR NOT NULL,
martin(> prenom VARCHAR,
martin(> age INTEGER,
martin(> classe CHAR(2) );
CREATE

martin=>
```

Maintenant, vous pouvez toujours essayer de mettre une ligne dont le champ "nom" est à NULL, le SGBD refusera obstinément :

```
martin=> INSERT INTO eleves VALUES(null,null,null,null);
ERROR:~ ExecAppend: Fail to add null value in not null attribute nom

martin=> INSERT INTO eleves VALUES('DUPONT',null,null,null);
INSERT 22664 1

martin=> UPDATE eleves SET nom=null;
ERROR:~ ExecReplace: Fail to add null value in not null attribute nom

martin=>
```

Maintenant, une petite question : le nom d'un élève est-il suffisant pour l'identifier ? Bien sur que non. Si vous dites à la base de données que l'élève « MARTIN » passe dans la classe 3A :

```
UPDATE eleves SET classe='3A' WHERE nom='MARTIN';
```

Vous allez mettre tous les élèves portant ce nom dans la classe ! Vous pourriez alors dire qu'un élève est identifié par son nom **et** son prénom, mais les homonymes ne sont pas rares. Au cours de ma scolarité j'ai rencontré au moins trois "Philippe Martin", et j'ai moi-même un homonyme qui jouait comme moi aux échecs

dans la même ligue... La FFE s'arrachait les cheveux pour calculer nos classements jusqu'à ce que j'utilise tous les prénoms de mon état civil :-)

Il nous faut donc une caractéristique unique à chaque élève. On peut penser au numéro INSEE (numéro de sécurité sociale) par exemple. L'idée ne serait pas mauvaise en soit. En effet chaque individu ayant son propre numéro INSEE, nous avons bien là une caractéristique unique que nous appelons dans le jargon des bases de données une **Clef Primaire**. On appelle une **clef** tout ce qui permet de caractériser des tuples. Ainsi, le nom de l'élève est une clef, de même que le couple (nom,prénom). C'est une convention, c'est tout. Mais on appelle **clef primaire** une caractéristique qui permet d'identifier un tuple **à coup sûr** . Je veux dire par là qu'on est absolument certain que chaque tuple possède cette clef, et qu'il n'existe pas deux tuples qui ont la même clef.

Donc, le numéro d'INSEE est une clef primaire tout à fait valable. On à l'habitude d'appeler ce genre de clef une clef primaire **fonctionnelle** . Pourquoi ? Parce que dans ce cas là, la clef a une véritable signification, c'est une caractéristique qui comporte des informations. D'autres exemples de clef fonctionnelles seraient, pour un livre par exemple, le numéro ISBN, ou à la rigueur le triplet (nom de l'auteur, titre, éditeur)... encore que pour cette dernière ce ne serait pas très bien choisi. Un roman a pu être édité deux fois par le même éditeur, mais avec des traductions révisées par exemple.

Par expérience, je vais vous dire qu'il est rarement opportun de choisir une clef fonctionnelle comme clef primaire. Même si sur le papier dans un cours magistral de SGBD à l'université c'est beau, c'est propre, carré, dans la pratique cela devient vite ingérable.

Si l'on revient à l'exemple de notre élève, choisir le numéro INSEE de l'élève comme clef primaire aurait une première conséquence immédiate : impossible d'inscrire ou de préinscrire l'élève si vous n'avez pas son numéro INSEE ! (Quoi ? Vous n'avez pas le numéro du formulaire rose qu'on vous a envoyé ? Faudra revenir au guichet avec sinon on ne peut rien faire ! ;-)).

Une autre conséquence un peu plus technique, c'est que si vos clefs primaires sont fonctionnelles elles sont bien souvent composées de plusieurs champs (nom+prénom), donc dès que voudrez faire des jointures, vous allez être contraint à écrire des clauses WHERE interminables, ce qui est particulièrement bugogène et pas spécialement efficace en terme de performances.

C'est pourquoi on utilise des **clefs informatiques** . Il s'agit d'une valeur, sans signification, que l'on attribue à chaque tuple et qui servira de clef primaire. Voici quelques avantages pratiques de la méthode :

- Ces valeurs sont générées automatiquement lorsqu'on insère un nouveau tuple, donc pas de risque de ne pas pouvoir enregistrer une fiche à cause d'un renseignement manquant.
- Pas de risque de « collision » (homonymes...)
- Une clef primaire sur une seule colonne s'avèrera très pratique pour les jointures et les requêtes complexes.
- Une clef primaire numérique est performante en terme de temps d'accès et n'occupe que peu de volume de stockage.

Pour générer ces valeurs automatiques, on utilise des séquences. Ce sont des objets qui se manipulent un peu comme des tables :

```
 martin=> CREATE SEQUENCE seq_eleves INCREMENT 1 START 1;
CREATE
martin=> select nextval('seq_eleves');
nextval
---------
      1
```

```
(1 row)

martin=> select nextval('seq_eleves');
nextval
---------
      2
(1 row)

martin=> select nextval('seq_eleves');
nextval
---------
      3
(1 row)

martin=> select last_value from seq_eleves;
last_value
------------
         3
(1 row)

martin=> select nextval('seq_eleves');
nextval
---------
      4
(1 row)

martin=> select last_value from seq_eleves;
last_value
------------
         4
(1 row)
```

Comme vous pouvez le voir, la séquence de fait que générer des valeurs numériques successives. À la création nous lui avons demandée de commencer à compter à partir de la valeur 1, puis d'ajouter 1 à chaque appel de **nextval()**.

Reprenons notre table eleves. Nous allons ajouter une colonne **id** (identifiant de l'élève) qui sera une clef primaire, alimentée par notre séquence

```
martin=> DROP TABLE  eleves;
DROP
martin=> CREATE TABLE eleves(
martin(> id INTEGER NOT NULL DEFAULT NEXTVAL('seq_eleves') PRIMARY KEY,
martin(> nom VARCHAR NOT NULL,
martin(> prenom VARCHAR,
martin(> age INTEGER,
martin(> classe CHAR(2) );
NOTICE:  CREATE TABLE/PRIMARY KEY will create implicit index 'eleves_pkey'
for table 'eleves'
CREATE
martin=>
```

Dans la nouvelle définition de la table eleves, nous avons donc un champ qui ne peut pas contenir NULL, qui - si aucune valeur n'est fournie lors de l'insertion d'un nouveau tuple - sera alimenté par la sequence seq_eleves, et qui plus est a été déclaré comme clef primaire. Le SGBDR à répondu qu'il créait automatiquement un index pour pouvoir gérer cette clef primaire. Maintenant, voyons ce que cela donne :

```
martin=> INSERT INTO eleves(nom,prenom) VALUES ('MARTIN','Philippe');
INSERT 22816 1
martin=> SELECT * FROM eleves;
id |  nom   |  prenom  | age | classe
----+--------+----------+-----+--------
 5 | MARTIN | Philippe |     |
(1 row)

martin=> INSERT INTO eleves(id,nom,prenom) VALUES (5,'TUX','Family');
ERROR:  Cannot insert a duplicate key into unique index eleves_pkey
martin=> INSERT INTO eleves(nom,prenom) VALUES ('TUX','Family');
INSERT 22818 1
martin=> SELECT * FROM eleves;
id |  nom   |  prenom  | age | classe
----+--------+----------+-----+--------
 5 | MARTIN | Philippe |     |
 6 | TUX    | Family   |     |
(2 rows)

martin=>
```

Maintenant, créons une nouvelle table *messages* qui va contenir des messages envoyés par les parents d'un élève. Cette table va être simpliste, elle va contenir la date et l'heure du message, le texte, ainsi que l'identifiant de l'élève concerné. Nous établissons là une relation entre la table messages et la table eleves. L'identifiant d'élève que nous mettons dans la table message est ce qu'on appelle une **clef étrangère** , c'est à dire que la valeur que nous allons mettre dans cette colonne est la clef primaire d'une autre table. Voici comment faire :

```
martin=> CREATE TABLE messages (
martin(> id INTEGER NOT NULL REFERENCES eleves,
martin(> texte TEXT,
martin(> dateheure DATETIME NOT NULL DEFAULT now());
NOTICE:~ CREATE TABLE will create implicit trigger(s) for FOREIGN
KEY check(s)
CREATE

martin=>
```

Avec la contrainte de colonne REFERENCES on indique que la valeur du champ doit être celle d'une clef primaire de la table eleves. Signalons que des options permettent de définir le comportement de la base de données en cas de suppression d'un élève (les messages sont supprimés ou bien l'id passe à NULL, ou bien la suppression est refusée tant qu'il existe des messages). Le type **TEXT** est l'équivalent d'un VARCHAR, mais selon les normes SQL il permet de contenir plus de caractères. Signalons aussi le type datetime qui est une date/heure, et que l'on initialise par défaut à la valeur renvoyée par now(), c'est à dire l'heure courante.

Lors de la création de la table, le SGBD nous a informés de la création de **triggers**. Nous verrons dans un prochain article ce que sont les triggers. Pour l'instant, ne vous en souciez pas, ceux là sont automatiquement gérés par le SGBD afin de répondre aux contraintes indiquées dans le CREATE TABLE.

Si nous voulons ajouter le numéro d'INSEE de l'élève, nous savons que ce numéro est unique, mais nous n'avons pas voulu l'utiliser comme clef primaire afin de pouvoir insérer dans notre table des élèves dont on ne connait pas le numéro d'INSEE. On peut néanmoins ajouter cette colonne avec la contrainte UNIQUE :

```
martin=> DROP TABLE eleves;
NOTICE:~ DROP TABLE implicitly drops referential integrity trigger
from table "messages"
DROP

martin=> CREATE TABLE eleves(
martin(> id INTEGER NOT NULL DEFAULT NEXTVAL('seq_eleves') PRIMARY
KEY,
martin(> nom VARCHAR NOT NULL,
martin(> prenom VARCHAR,
martin(> age INTEGER,
martin(> classe CHAR(2),
martin(> insee CHAR(15) NULL UNIQUE );
NOTICE: CREATE TABLE/PRIMARY KEY will create implicit index 'eleves_pkey'
for table 'eleves'
NOTICE:CREATE TABLE/UNIQUE will create implicit index 'eleves_insee_key'
for table 'eleves'
CREATE

martin=>
```

Ainsi notre SGBD nou laissera insérer des tuples sans renseigner le champ INSEE. Cependant, si celui-ci est renseigné, le SGBD verifiera bien qu'il n'a pas déjà été déjà utilisé.

8.1 Les contraintes de tables

Les contraintes que nous venons d'évoquer peuvent éventuellement porter sur plusieurs colonnes. Par exemple bien que chaque wagon d'un train possède des places numérotées de 1 à 90, le triplet (numéro de train, numéro de voiture, numéro de siège) est quant à lui unique.

Ces contraintes de table s'ajoutent après la déclaration des colonnes dans la création de la table :

```
martin=> CREATE TABLE reservations(
martin(> no_train INTEGER NOT NULL,
martin(> no_voiture INTEGER NOT NULL,
martin(> no_siege INTEGER NOT NULL,
martin(> UNIQUE (no_train,no_voiture,no_siege)
martin(> );
NOTICE:~ CREATE TABLE/UNIQUE will create implicit index 'reservations_no_train_key'
for table 'reservations'
CREATE

martin=>
```

Comme vous le constatez, le SGBD crée un index pour gérer cette contrainte. Soyez donc attentif à l'ordre dans lequel vous énumérez vos colonnes dans la contrainte UNIQUE afin d'obtenir de meilleures performances (voir l'article sur les index).

Vous pouvez également dans une contrainte de table définir une clef étrangère vers une table dont la clef primaire est composée de plusieurs colonnes. Beaucoup d'autres options de contraintes sont ainsi disponible, j'invite le lecteur à consulter son manuel de référence afin d'en avoir la liste exhaustive.

Néanmoins, ces quelques contraintes de colonnes et de tables (NOT NULL, UNIQUE, REFERENCES, PRIMARY KEY) sont en général suffisantes pour mettre en oeuvre de petites bases de données. J'appelle petite base de données des schémas d'un trentaine de tables maximum dont la volumétrie de la plus grosse d'entre elle est inférieure à 250 000 lignes environ. Le site LinuxFrench.net utilise une « petite base de données ».

9 Episode 8 : Procédures stockées, fonctions.

Plutot 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. Nous allons ici parler de celles écrites en pl/pgSQL. Pourquoi ce choix ? Parce que ce langage est très proche du pl/SQL d'Oracle, et bien que différent, semblable au transact-SQL de Sybase.

Il nous faut donc commencer par activer le langage pl/pgsql pour notre base de données. Si vous utilisez une Mandrake 8.x voici la marche à suivre :

```
[root@localhost src]# su - postgres
bash-2.05$ export PGLIB=/usr/lib/
bash-2.05$ createlang plpgsql martin
bash-2.05$
```

« martin » étant bien sûr le nom de notre base de données. Sous postgreSQL les procédures stockées sont en fait des fonctions qui peuvent être utilisées dans un ordre SELECT comme n'importe quelle autre fonction. Si vous utilisez une autre distribution, cherchez le répertoire où se trouve **plpgsql.so**, initialisez PGLIB de manière à pointer sur ce répertoire, et créez le langage avec **createlang.**

Les procédures stockées ont plusieurs intérêts :

- Le premier est d'avoir une entité **fonctionelle** . C'est à dire, d'avoir un morceau de code fait une fois pour toute qui va s'occuper 'effectuer une opération qui a une signification fonctuinelle, et non pas seulement une signification physique dans la base de données. 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 supprimer les messages des parents, etc. On peut ainsi regrouper toutes ces opérations au sein d'une seule et unique procédure *supprimer_eleve()*.
- Le second, qui découle directement du premier, est de **factoriser** le code. C'est à dire d'avoir un code unique, plutot que plusieurs exemplaires du même code éparpillés au sein de l'application. Ainsi, s'il faut modifier la structure d'une table par exemple, on évite beaucoup de problèmes : il suffira de mettre à jour les procédures stockées concernées au lieu de devoir auditer l'ensemble du code de l'application pour trouver quels appels sont fait sur cette table.
- En terme de performances réseau, les procédures stockées apportent également beaucoup. En effet, lorsqu'il faut effectuer des opérations successives, cela se traduit par des allers-retours réseaux coûteux en terme de performance entre le SGBDR et l'application. Alors qu'avec un simple appel de procédure depuis l'application, toutes les opérations vont s'effectuer au sein même du SGBDR.
- Toujours en terme de performances, cela économise les analyses lexcicographiques et syntaxiques du code de la requête. Ces choses là ont été faites une fois pour toute. Certains SGBD (hélàs, ce n'est pas

encore le cas de pgSQL) effectuent même une compilation complète de la procédure, avec un pré-calcul des plans d'exécution.

Assez de théorie ! Voyons un peu concrètement ce que cela donne. Voici une table toute simple avec quelques données :

```
aegir=> create table comptes(no integer primary key, solde decimal(10,2));
...
aegir=> select * from comptes;
no |  solde
----+---------
 1 |  100.00
 2 | 1000.00
 3 | 1500.00
 4 |   10.00
```

Maintenant, créons une fonction pour effectuer un virement d'un compte vers un autre. 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 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';
```

Voyons tout d'abord la structure générale de création d'une procédure (fonction) :

```
CREATE FUNCTION nom_fonction(paramètres) RETURNS type AS 'code de
la fontion' LANGAGE 'nom du langage' ;
```

Dans notre cas, notre fonction s'appelle **virement** , admet 2 paramètres numériques entiers et 1 décimal. 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 updates successifs afin d'effectuer le virement :

```
aegir=> select * from comptes;
no |  solde
----+---------
 2 | 1000.00
 3 | 1500.00
 4 |   10.00
 1 |  100.00
(4 rows)
```

```
aegir=> select virement(1,2,100);
virement
----------
       0
(1 row)

aegir=> select * from comptes;
no |  solde
----+---------
 3 | 1500.00
 4 |   10.00
 1 |  200.00
 2 |  900.00
(4 rows)
```

Bien sûr, pour être tout à fait rigoureux, nous devrions faire :

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

Maintenant, notre procédure devrait vérifier que le compte est suffisamment approvisionné avant de faire le virement. En voici la nouvelle version (les nouveautés sont en *italique*) :

```
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.

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
        RAISE EXCEPTION ''Compte no % Inconnu'',debiteur;

  END IF;

  IF (ancien_solde > montant) THEN

SELECT count(1) into test FROM comptes WHERE no=crediteur;
        IF (test <> 1) THEN
                RAISE EXCEPTION ''Compte no % Inconnu'',crediteur;
        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. Le **IF NOT FOUND** sera vérifié si l'instruction le précédant n'a trouvé aucune ligne à manipuler. 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 connaitre directement le nombre de lignes manipulées par la dernière instruction SQL.

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 chaine de caractères **(CREATE FUNCTION nom() RETURNS type AS 'code'...)**, il est nécessaire *d'échapper* les délimiteurs de chaines 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. Ainsi, si nous nous trouvons dans une transaction (ce qu'il faut impérativement faire afin de ne pas courrir le risque d'avoir des virements qui ne sont que partiellement effectués), l'opération sera annulée.

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 franc sur le compte numéro 1 (hum ;-)) :

```
aegir=> select * from comptes;
no |  solde
----+---------
 3 | 1500.00
 4 |   10.00
 2 |  797.00
 1 |  296.00
(4 rows)

aegir=> BEGIN TRANSACTION; select virement(1,no,1) FROM COMPTES WHERE no <>1; COMMIT;
BEGIN
virement
----------
       0
       0
       0
(3 rows)

COMMIT
aegir=> select * from comptes;
no |  solde
----+---------
 3 | 1499.00
 4 |    9.00
 2 |  796.00
 1 |  299.00
(4 rows)

aegir=>
```

10 Episode 9 : 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.

10.1 Qu'est-ce que c'est ?

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

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

- Contrainte logique. Avec le mot-clée *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 clef étrangères, mais pour un tuple donné il n'est pas possible d'avoir les deux champs renseignés. 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.
- 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.

10.2 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, parmis 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.

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

```
martin=> create table article(id integer not null primary key, date_modif
date not null default now(), texte text, status varchar);
NOTICE:~ CREATE TABLE/PRIMARY KEY will create implicit index 'article_pkey'
for table 'article'
CREATE

martin=> create table archive(id integer, date_modif date not null,
texte text);
CREATE

martin=>
```

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();
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 :

```
DROP TRIGGER trg_arch_art ON article;
CREATE TRIGGER trg_arch_art BEFORE DELETE OR UPDATE ON article
FORE ACH ROW EXECUTE PROCEDURE arch_art();
```

Maintenant, vérifions que le trigger fonctionne correctement :

```
martin=> insert into article(id,texte) values(1,'Ceci est le premier article original');
INSERT 40942 1
martin=> select * from article;
id | date_modif |               texte                | status
----+------------+------------------------------------+--------
 1 | 2001-11-19 | Ceci est le premier article original |
(1 row)

martin=> select * from archive;
id | date_modif | texte
----+------------+-------
(0 rows)

martin=> update article set texte='Ceci est le premier article version 2',
date_modif=now();
UPDATE 1
martin=> select * from article;
id | date_modif |                texte                | status
----+------------+-------------------------------------+--------
 1 | 2001-11-19 | Ceci est le premier article version 2 |
(1 row)

martin=> select * from archive;
id | date_modif |               texte
----+------------+-------------------------------------
 1 | 2001-11-19 | Ceci est le premier article original
(1 row)

martin=>
```

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();
```

Voici une illustration de l'exécution de ce trigger :

```
martin=> select * from article;
id | date_modif |                 texte                 | status
----+------------+---------------------------------------+--------
 1 | 2001-11-19 | Ceci est le premier article version 2 |
(1 row)

martin=> select * from archive;
id | date_modif |               texte
----+------------+-------------------------------------
 1 | 2001-11-19 | Ceci est le premier article original
(1 row)

martin=> update article set status='publié' where id=1;
UPDATE 1
martin=> select * from article;
id | date_modif |                 texte                 | status
----+------------+---------------------------------------+--------
 1 | 2001-11-19 | Ceci est le premier article version 2 | publié
(1 row)

martin=> select * from archive;
id | date_modif |               texte
----+------------+-------------------------------------
 1 | 2001-11-19 | Ceci est le premier article original
(1 row)

martin=> update article set texte='Ceci est une tentative de modification',
date_modif=now();
ERROR:  Article 1 déjà publié
martin=>
```

Comme vous pouvez le voir, les triggers sont particulièrement pratiques pour débarasser l'application cliente de la gestion des règles de workflow par exemple. Ils sont aussi très conseillés pour gérer des règles de sécurité.

11 Episode 10 : Optimisations et performances.

Les performances sont une problématique souvent évoquée au sujet des bases de données. On parle souvent de tuning (ajustement des paramètres pour affiner les performances). Il est nécessaire de mettre les mains dans le cambouis, et comprendre comment fonctionnent les rouages d'un SGBDR avant de pouvoir traiter les problèmes de performances.

11.1 Les niveaux d'optimisation

Le problème des performances n'est pas un problème global au système d'information. Il faut le traiter niveau par niveau.

- Le niveau conceptuel. Cela va consister à concevoir correctement le schéma physique de sa base de données. Créer les index adéquats, choisir correctement les types de données etc.
- Le niveau applicatif. Il s'agira avant tout d'écrire correctement ses requêtes.
- Le niveau serveur. D'abord le choix du matériel, la configuration de celui-ci, et ensuite le paramétrage du serveur de base de données proprement dit.

Pour chacun de ces points, il est nécessaire d'avoir une bonne compréhension du fonctionnement interne des SGBD, afin de pouvoir choisir des options judicieuses.

11.2 Le niveau conceptuel

La principale question est celle des index. Bien sûr, les champs déclarés en tant que clef primaire se voient automatiquement indexés. Mais d'une manière générale, tous les champs qui sont susceptibles d'être utilisés comme clef d'accès doivent (enfin, tout est relatif hein ?) être indexés.

Par exemple, dans notre table ELEVES, nous avons une clef primaire infomatique (id) sur laquelle le SGBD a automatiquement placé un index. Mais il serait de bon ton de créer des index sur "nom", "prenom" et "numero_insee". Maintenant, il faut se poser la question de savoir si la création d'un idex composé de plusieurs champs est judicieuse ou pas. Il faut bien comprendre le fonctionnement d'un index. Dans une table, chaque ligne est dotée d'un object identifier - ou OID - qui est en quelque sorte son adresse. Le SGBDR utilise en interne ces OID pour désigner les tuples. Un OID est unique dans l'ensemble de la base de données. Imaginons une table de 8 lignes très simple, contenant les entiers de 1 à 8 :

OID	valeur
222	1
223	2
224	3
225	4
226	5
227	6
228	7
229	8

Si la table n'est pas indexée, et que vous faites une requête dotée d'un clause WHERE valeur=5, le SGBDR va examiner chaque ligne de la table, et renvoyer l'OID de chaque tuple où le champ valeur est à 5. Vous allez me dire qu'il suffira d'examiner seulement 5 lignes pour trouver le tuple qui nous intéresse. Non ! Le SGBDR devra examiner les 8 lignes afin de vérifier qu'il n'existe pas un autre tuple avec valeur=5. C'est pourquoi l'utilisation d'une simple contrainte UNIQUE lors de la création de la table peut influer sur les performances !

Quand on crée un index sur une valeur, un arbre équilibré est créé, l'organisation ressemblera donc très schématiquement à la suivante :

Noeud 1	Noeud 2	Noeud 3	Noeud 4
		1 (222)	
	2 (223)		3 (224)
		4 (225)	
5 (226)			
		6 (227)	
	7 (228)		
		8 (229)	

Ainsi, si nous cherchons la valeur 6 par exemple, on accède au premier noeud. La valeur de ce noeud est 5, donc inférieure à la clef recherchée. Nous passons donc au noeud numéro 2, en prenant la branche inférieure.

Sa valeur est 7, on emprunte donc la branche supérieure du noeud, et on obtient le noeud 6, qui nous indique que la ligne qui nous intéresse comporte l'OID 227. Il ne nous aura fallu que 3 comparaisons au lieu de 8. De plus, alors qu'avec une organisation séquentielle le nombre de comparaisons croît linéairement en fonction du nombre de lignes que comporte la table, avec un BTree, le nombre de comparaisons augmente logarithmiquement.

Lorsqu'on crée un index composé, les clefs du BTree sont tout simplement la concaténation des champs composant l'index. Par conséquent, si votre index est composé de (nom,prénom), il vous est possible de parcourir l'arbre même si vous ne faite qu'une recherche sur le nom. L'index vous trouvera rapidement tous les élèves dont le nom est "Dupont". Par contre, si votre recherche est effectuée uniquement sur le prénom, l'index sera inutilisable : comment pourriez-vous effectuer la comparaison des clefs ? (Dupont|Jean est-il supérieur ou inférieur à « Albert » ?)

11.3 Écriture de requêtes, tables directrices et statistiques

Prenons par exemples nos deux tables **eleves** et **classes** :

```
martin=# select * from eleves;
id |   nom   | prenom | age | classe | insee
----+---------+--------+-----+--------+-------
 7 | Alpha   | A      |  15 | 3A     |
 8 | Bravo   | B      |  15 | 3A     |
 9 | Delta   | D      |  15 | 3B     |
10 | Epsilon | E      |  16 | 3B     |
11 | Gamma   | E      |  16 | 3A     |
12 | Mu      | M      |  16 | 3A     |
13 | Nu      | N      |  16 | 3B     |
14 | Omega   | O      |  16 | 3B     |
15 | Chi     | Q      |  16 | 3A     |
(9 rows)

martin=# select * from classes;
classe
--------
3B
3A
(2 rows)

martin=#
```

Nous allons utiliser la directive **EXPLAIN** pour avoir un aperçu du fonctionnement interne du SGBDR :

```
martin=# explain select * from eleves,classes where eleves.classe=classes.classe;
NOTICE:  QUERY PLAN:

Nested Loop  (cost=0.00..32.50 rows=10 width=68)
 ->  Seq Scan on eleves  (cost=0.00..0.00 rows=1 width=56)
 ->  Seq Scan on classes  (cost=0.00..20.00 rows=1000 width=12)

EXPLAIN

martin=#
```

PostgreSQL nous indique ici qu'il va effectuer un scan séquenciel , c'est à dire examiner chaque ligne l'une après l'autre, sur les deux tables. Il n'a pas vraiment le choix puisque je n'ai pas créé le moindre index. La table directrice choisie est **eleves**. C'est à dire qu'il commence par examiner la table **eleves**, et pour chacune des lignes de cette table, il effectuera un scan sur la table **classe** afin d'effectuer la jointure demandée.

PostgreSQL effectue une estimation du « coût » de chaque opération, afin de choisir la meilleure stratégie. La stratégie choisie est ce qu'on appelle le **plan d'exécution**.

On peut voir ici que postgreSQL estime que le scan sur la table **eleves** est d'un coût nul, tandis que le scan sur la table **classe** coûtera entre 0 et 20 (c'est un indice, donc sans unité). Pour faire cette estimation de coût, pgSQL se base sur le type d'opérations à effectuer (comparer un entier, comparer un chaîne de caractères etc.) ainsi que sur le nombre de lignes que comporte chaque table. On peut voir ici que pgSQL a estimé que la table **classes** comportait 1000 lignes, tandis que la table **eleves** comporte 1 seule ligne.

Il n'est donc pas étonnant qu'il ait choisi d'effectuer un scan de la table **classes** pour chaque ligne de la table **eleves** puisque cette dernière ne comporte qu'une seule ligne... mais le seul problème c'est que **ces chiffres sont complètement faux ! ! !**. En effet notre table **classes** comporte 2 lignes, et **eleves** 9 lignes. Alors que se passe-t-il ?

pgSQL utilise des statistiques pour faire ces évaluations. C'est ce qu'on appelle **l'optimisation statistique** . Mais il faut bien que ces statistiques soient à jour afin que l'optimiseur choisisse le plan d'exécution le plus efficace. Avec la commande **VACUUM** (ou bien avec l'utilitaire **vacuumdb** en ligne de commande, pgSQL va examiner l'état de chaque table afin de mettre à jour ses statistiques :

```
martin=# vacuum;
VACUUM
martin=# explain select * from eleves,classes where eleves.classe=classes.classe;
NOTICE:  QUERY PLAN:

Merge Join  (cost=2.26..2.40 rows=2 width=68)
 ->  Sort  (cost=1.23..1.23 rows=9 width=56)
       ->  Seq Scan on eleves  (cost=0.00..1.09 rows=9 width=56)
 ->  Sort  (cost=1.03..1.03 rows=2 width=12)
       ->  Seq Scan on classes  (cost=0.00..1.02 rows=2 width=12)

EXPLAIN
```

On voit immédiatement que pour la même requête, le plan d'exécution à changé. De plus, le nombre de ligne estimé dans les tables est désormais correct. Il faut donc penser à exécuter **VACUUM** après avoir effectué des mises à jour massives sur la base de données

Si vous créez un index sur la table classes (create index idx1 on eleves(classe) ;) vous pourrez vous rendre compte que le plan d'exécution ne change pas. Pourquoi ? Parce que tant que la table ne contient que quelques lignes, il est plus coûteux d'utiliser un index qu'effectuer un scan séquenciel.

Nous pouvons nous « amuser » à remplir notre table eleves :

```
martin=# insert into eleves(nom,prenom,age,classe)
(select nom,prenom,age,classe from eleves);
INSERT 0 9
martin=# insert into eleves(nom,prenom,age,classe)
(select nom,prenom,age,classe from eleves);
```

```
INSERT 0 18
...
martin=# insert into eleves(nom,prenom,age,classe)
(select nom,prenom,age,classe from eleves);
INSERT 0 294912
martin=# insert into eleves(nom,prenom,age,classe)
(select nom,prenom,age,classe from eleves);
INSERT 0 589824
```

Nous avons désormais une table contenant plus d'un million de lignes. Voyons ce qui se passe sans index, puis avec un index :

```
 martin=# vacuum;
VACUUM
martin=# explain select * from eleves,classes where eleves.classe=classes.classe;
NOTICE:  QUERY PLAN:

Nested Loop  (cost=0.00..74731.18 rows=23593 width=68)
 ->  Seq Scan on classes  (cost=0.00..1.02 rows=2 width=12)
 ->  Seq Scan on eleves  (cost=0.00..22619.48 rows=1179648 width=56)

EXPLAIN
martin=# create index idx1 on eleves(classe);
CREATE
martin=# explain select * from eleves,classes where eleves.classe=classes.classe;
NOTICE:  QUERY PLAN:

Nested Loop  (cost=0.00..61562.29 rows=23593 width=68)
 ->  Seq Scan on classes  (cost=0.00..1.02 rows=2 width=12)
 ->  Index Scan using idx1 on eleves  (cost=0.00..30633.18 rows=11796 width=56)

EXPLAIN
martin=#
```

Avec un million de lignes, postgreSQL préfère scanner un index (on le comprend !). Il utilise alors la table classes comme table directrice. Enfin, signalons une chose pas forcément évidente, les SGBDR n'utilisent **qu'un seul index** par table. Si la table comporte plusieurs index utilisables pour traiter une requête, l'optimiseur statistique choisira celui qui lui semble le plus approprié. Par exemple en cas de clause where du type *WHERE age=15 AND nombre_freres > 3* et que ces deux champs sont indexés, il est fort probable que l'optimiseur statistique optera pour l'index sur **age** , car il considère qu'un test d'égalité ramènera moins de lignes qu'un test de comparaison. Ce qui n'est pas forcément judicieux, mais souvenez-vous en lorsque vous rechercherez pourquoi une requête est longue à s'exécuter :-)

11.4 Le matériel

Une question reviens souvent : « quel matériel dois-je acheter pour faire tourner un serveur pgSQL qui va faire xxxx ». Il n'y a pas de réponse ! Cela dépend !

Cela dépend de la volumétrie des tables, de la concurrence d'accès, du nombre de postes clients (ou instances d'applications) connectés simultanément, du type de requêtes, et des contraintes de disponibilités...

D'abord, en ce qui concerne la RAM : plus il y en a de disponible, mieux c'est ! En règle générale, prévoir entre 4 et 8 Mo de RAM par connexion active, cela met à l'abri.

En ce qui concerne le sous-système disque. Il faut toujours du SCSI. D'abord parce que le SCSI consomme moins de ressources CPU que l'EIDE, ensuite parce que les accès peuvent être parallélisés sur plusieurs périphériques, alors qu'en EIDE ils seront sérialisés. IDonc le minimum est d'avoir 2 disques SCSI, l'un qui contiendra les bases de données, et l'autre consacré au système (/tmp /var etc.). Si vous êtes fortuné, ou bien si vous avez de fortes exigeances pour votre SGBDR, utilisez 2 chaînes SCSI. Une première consacrée au système, et une seconde en RAID-5 hardware. Non seulement cela pourra vous permettre une haut disponibilité quand à votre sous-système disques, mais cela permettra également de paralléliser des accès concurrents.

11.5 La configuration du serveur

PostgreSQL est doté de nombreux paramétrages, comme tout autres SGBDR. Je ne les passerais pas en revue ici. D'abord parce que je n'ai personellement jamais « joué » assez longtemps avec les paramètres de pgSQL pour pouvoir donner des conseils judicieux. Je vous renvoie donc à la documentation, et en particulier le Manuel d'administration.

Je signale néanmoins 4 paramètres qui me paraissent primordiaux dans les performances d'un serveur :

- SHARED_BUFFERS qui spécifie le nombre de pages de 8 Ko qui seront utilisées comme buffer par postgreSQL.
- SORT_MEM qui spécifie la mémoire qui sera disponible pour que l'optimiseur puisse effetuer ses tris internes. Lorsqu'une telle opération ne pourra tenir en mémoire, des fichiers temporaires seront utilisés.
- WAL_FILES indique le nombre de fichiers de log créés d'avance.
- WAL_BUFFERS nombre de buffers utilisés pour le logging.

D'une manière générale, je dirais que tant que vous avez de la RAM disponible, il ne faut pas hésiter à augmenter les deux premiers paramètres

12 Episode 11 : Les utilisateurs et leurs droits : CREATE GROUP, GRANT.

Il est indispensable d'introduire des éléments de sécurité dans sa base de données. Ainsi, si par exemple un pirate exploite une faille de votre serveur web, il ne pourra pas détruire votre base de données pas plus qu'il ne pourra consulter les données confidentielles qu'elle contient.

Disons le tout net, postgreSQL est encore un peu à la traîne en ce qui concerne les droits d'accès sur une base de données. Néanmoins, les principales fonctionalités sont présentes.

12.1 Utilisateurs

Les SGBDR utilisent souvent les concepts de **DBA** et **DBO**. Le DBA est le « *database administrator* ». C'est à dire celui qui administre le serveur de SGBDR. C'est en quelque sorte le super-utilisateur. Avec postgreSQL, il s'agit en général de l'utilisateur **postgres**. Lorsque vous êtes connecté sur Linux sous le compte **postgres**, vous pouvez alors créer ou supprimer des bases de données ou des utilisateurs.

Le DBO est le « **database owner** ». C'est le propriétaire de la base de données. Il peut créer de nouvelles tables, les supprimer, gérer les droits des utilisateurs sur cette base de données, consulter les tables, y écrire etc. Sur un serveur, il y a donc un DBA, et autant de DBO que le serveur contient de bases de données. Le DBO est celui qui a créé la base de données.

Il faut bien faire la distinction entre utilisateur et base de données. En effet PostgreSQL tout comme Oracle peut facilement prêter à confusion ouisque par défaut un utilisateur du même nom que la base de données est créé.

Par exemple, nous allons nous créer une base de données *linuxfrench*, et un utilisateur *linuxfrench_dbo* qui nous servira par la suite à administrer la base de données, et un utilisateur *linuxfrench_user* qui sera l'utilisateur utilisé par l'application (par exemple par les scripts PHP d'un serveur web) :

```
[root@localhost dua]# su - postgres
bash-2.05$ createdb linuxfrench
CREATE DATABASE
bash-2.05$ createuser -W linuxfrench_dbo
Shall the new user be allowed to create databases? (y/n) n
Shall the new user be allowed to create more new users? (y/n) y
Password:
CREATE USER
bash-2.05$ createuser  linuxfrench_user
Shall the new user be allowed to create databases? (y/n) n
Shall the new user be allowed to create more new users? (y/n) n
CREATE USER
bash-2.05$
```

Maintenant on va se connecter en DBO pour créer une table :

```
linuxfrench=# create table toto(a integer);
CREATE
linuxfrench=# insert into toto values (1);
INSERT 1475451 1
linuxfrench=#
```

Et enfin, nous allons nous reconnecter en tant que linuxfrench_user :

```
 linuxfrench=# \q
[aegir@localhost aegir]$ psql -U linuxfrench_user -d linuxfrench
Welcome to psql, the PostgreSQL interactive terminal.

Type:  \copyright for distribution terms
       \h for help with SQL commands
       \? for help on internal slash commands
       \g or terminate with semicolon to execute query
       \q to quit

linuxfrench=> select * from toto;
ERROR:  toto: Permission denied.
linuxfrench=>
```

Notre utilisateur linuxfrench_user ne peut pas accéder à la table **toto**. En effet cette table à été créée par l'utilisateur **linuxfrench_dbo** , par défaut seul ce dernier utilisateur à donc le droit de manipuler la table. Nous allons voir comment ajouter des droits à l'utilisateur linuxfrench_user.

12.2 GRANT et REVOKE

```
[aegir@localhost aegir]$ psql -U linuxfrench_dbo -d linuxfrench
Welcome to psql, the PostgreSQL interactive terminal.

Type:  \copyright for distribution terms
       \h for help with SQL commands
       \? for help on internal slash commands
       \g or terminate with semicolon to execute query
       \q to quit

linuxfrench=# GRANT select ON toto TO linuxfrench_user;
CHANGE
linuxfrench=# \q
[aegir@localhost aegir]$ psql -U linuxfrench_user -d linuxfrench
Welcome to psql, the PostgreSQL interactive terminal.

Type:  \copyright for distribution terms
       \h for help with SQL commands
       \? for help on internal slash commands
       \g or terminate with semicolon to execute query
       \q to quit

linuxfrench=> select * from toto;
a
---
1
(1 row)

linuxfrench=> insert into toto values(2);
ERROR:  toto: Permission denied.
linuxfrench=>
```

Nous nous sommes d'abord connectés en dbo afin de donner à l'utilisateur *linuxfrench_user* le droit d'accéder à la table **toto** . Ceci se fait grâce à la commande GRANT. On peut voir que si l'utilisateur peut désormais effectuer des SELECT sur la table, il lui est en revanche toujours impossible de faire un INSERT.

En fait, avec la commande GRANT, on peut donner des droits pour chaque commande SQL : SELECT, INSERT, UPDATE, DELETE.

Ainsi, si votre application est un site web par exemple, il est préférable de ne donner à l'utilisateur qui va être utilisé dans vos scripts PHP que le droit d'effectuer des SELECT. En effet, en général les visiteurs d'un site ne vont jamais écrire dans la base de données. Jamais ou presque... Vous avez ainsi peut être une table **contacts** qui va servir à enregistrer les adresses email des visiteurs qui souhaitent recevoir un courriel d'information. Dans ce cas là, la situation est complètement inversée. Il nous faut pouvoir insérer de nouvelles lignes dans la table, par contre par mesure de confidentialité on va interdire à l'utilisateur de lire le contenu de la table. Cette table n'est normalement utilisée par l'application publique que pour ajouter de nouvelles adresses email :

```
GRANT insert ON contacts TO linuxfrench_user;
```

Par sécurité, on va enlever les autres droits (théoriquement c'est inutile puisque ces droits n'ont jamais été attribués à linuxfrench_user sur la table contacts)

```
REVOKE select, update, delete, rule ON contacts FROM linuxfrench_user;
```

(le privilège RULE consiste à pouvoir créer des règles avec la commande CREATE RULE).

Signalons que vous pouvez utiliser **ALL** pour désigner l'ensemble des privilèges :

```
REVOKE ALL ON backups FROM linuxfrench_user
```

12.3 Les groupes

Gérer les droits au niveau de l'utilisateur n'est en général pas une bonne idée. En effet, les utilisateurs on en ajoute et on en enlève au fil de la maintenance de l'application. Il est donc préférable de créer des groupes d'utilisateurs. On gèrera les droits au niveau du groupe, et on ajoutera tout simplement des utilisateurs dans ce groupe lorsqu'on en aura besoin.

```
[aegir@localhost aegir]$ psql -U linuxfrench_dbo -d linuxfrench
Welcome to psql, the PostgreSQL interactive terminal.

Type:  \copyright for distribution terms
       \h for help with SQL commands
       \? for help on internal slash commands
       \g or terminate with semicolon to execute query
       \q to quit

linuxfrench=# CREATE GROUP utilisateurs WITH USER linuxfrench_user;
CREATE GROUP
linuxfrench=#
```

Voilà, il nous reste à ajouter les droits voulus pour ce groupe, et nous allons aussi créer un nouvel utilisateur :

```
linuxfrench=# REVOKE ALL ON toto FROM GROUP utilisateurs;
CHANGE
linuxfrench=# GRANT select ON toto TO GROUP utilisateurs;
CHANGE
linuxfrench=# CREATE USER linuxfrench_stagiaire WITH PASSWORD 'stagiaire' NOCREATEDB
NOCREATEUSER IN GROUP utilisateurs;
CREATE USER
linuxfrench=#
```

Maintenant nous allons tester notre nouvel utilisateur :

```
 [aegir@localhost aegir]$ psql -U linuxfrench_stagiaire -d linuxfrench
Welcome to psql, the PostgreSQL interactive terminal.

Type:  \copyright for distribution terms
       \h for help with SQL commands
       \? for help on internal slash commands
       \g or terminate with semicolon to execute query
       \q to quit

linuxfrench=> select * from toto;
a
---
```

```
1
(1 row)

linuxfrench=> insert into toto values(1);
ERROR:  toto: Permission denied.
linuxfrench=>
```

12.4 Suite... ?

Voilà, cette série d'articles est terminée. J'espère qu'elle vous aura aidés à aborder simplement les SGBDR. Je répète que les articles ne sont absolument pas exhaustif, il est nécessaire de se reporter à la documentation de postgreSQL. Ces articles n'ont pour but que de vous permettre de vous immerger plus facilement dans le monde SQL.

J'ai gardé une petite surprise pour la fin... dans un ultime article je vous donnerais des conseils pratiques. Ce ne sont pas des règles formelles, mais plutôt des habitudes que j'ai acquises au fil de mon expérience, et à force de faire des erreurs :o)

Donc... À bientôt.

13 Conclusion

Voici une conclusion à la série d'articles d'initiation aux bases de données. Je vais essayer, au regard de mon expérience personnelle, de vous donner quelques astuces et bonnes habitudes destinées à vous simplifier la vie.

13.1 Scripts de création de la base

A mon avis, il faut impérativement proscrire les outils « graphiques » ou « interactifs » pour créer ses bases de données. Il est, de loin, préférable de travailler sur des scripts. Ce sont de simples fichiers textes qui contiennent les instructions créant les objets de la base de données. Voici par exemple un extrait d'un tel script :

```
\echo ******************************
\echo * TABLE users
\echo ******************************
drop sequence seq_users;
create sequence seq_users;
drop table users;
create table users(
user_id integer not null default nextval('seq_users') primary key,
user_login varchar,
user_password varchar,
user_profil integer not null references profils,
user_lastlogin timestamp);

\echo ******************************
\echo * TABLE sessions
\echo ******************************
drop sequence seq_sessions;
create sequence seq_sessions;
drop table sessions;
create table sessions(
...etc...
```

Quels sont les avantages d'un script ? Ils sont nombreux. D'abord, vous maitrisez parfaitement la nature et les types d'objets créés. Les interfaces graphiques ont la facheuse habitude d'interpréter de façon obscure des « méta-types ».

Ensuite, vous pouvez reproduire à l'identique les opérations d'initialisation de base autant de fois que vous le désirez, à travers un simple telnet ou ssh. C'est très pratique pour préparer une mise en production, c'est aussi très pratique pendant le développement. En effet, les développeurs font souvent des tests un peu hasardeux. Lorsqu'il y a des dégâts, il suffit de « rejouer les scripts » pour se retrouver avec une base de développement « propre ».

En ce qui me concerne, j'ai pour habitude de maintenir 5 scripts pour chaque base de données :

- create_base.sql, la création des objets proprement dits, voir l'exemple plus haut ;
- grant_base.sql, la création des utilisateurs et groupes par défaut, ainsi que l'exécution des GRANT adéquats sur les objets de la base de données ;
- function_base.sql, création des fonctions (procédures stockées) ;
- init_base.sql, insertion dans les tables des valeurs de références (liste des pays...) ;
- go_base.sql, appelle tout simplement successivement les 4 scripts précédents puis effectue un **vacuumdb** (dans le cas de postgresql).

Je précise que sous postgreSQL, un script est appelé avec **\i script.sql** sur la ligne de commandes psql.

13.2 UNION / ALL des performances très différentes

Lorsque vous effectuez une union de requêtes avec la clause UNION, il faut savoir que le SGBDR va se créer une table temporaire pour pouvoir éliminer les doublons. Si vous avez la certitude que chacune de vos requêtes ramènera un résultat unique, alors pensez à utiliser la clause **UNION ALL** plutôt que **UNION.** Le SGBDR renverra alors le résultat brut, sans chercher à éliminer les doublons. Les performances en seront bien meilleures.

13.3 Utilisez les alias de tables

Dans vos requêtes, utilisez systématiquement les alias de tables. Cela rendra les clauses WHERE sera beaucoup moins verbeuses (et donc plus lisibles, et plus rapides à taper !). Par exemple au lieu de :

```
SELECT classe.nom, eleves.nom, prenom
FROM eleves, classes
WHERE eleves.classe_id =  classes.classe_id
AND classes.nom LIKE '3%'
```

préférez :

```
SELECT c.nom, e.nom, prenom
FROM eleves e, classes c
WHERE e.classe_id =  c.classe_id
AND c.nom LIKE '3%'
```

13.4 Ne présumez jamais du résultat d'une opération impliquant un NULL

Je le dis, et je le répète : « null signifie indéterminé ». Donc NULL+1 vous ne pouvez pas savoir ce que ça va donner. Le premier réflexe est d'essayer :

```
aegir=# select 1+1 ;
?column?
----------
       2
(1 row)

aegir=# select 1+null ;
?column?
----------

(1 row)
```

Et d'en conclure que null+x donne NULL. C'est **FAUX**. Nous avons là typiquement un résultat qui peut changer d'un SGBDR à un autre, d'une version d'un SGBDR à une autre version d'un même SGBDR, ou même parfois suivant l'humeur d'un SGBDR. Lorsqu'un champ intervenant dans un calcul est susceptible de contenir la valeur NULL, utilisez la fonction COALESCE qui retourne le premier paramètre non null :

```
aegir=# select 1+coalesce(null,0) ;
?column?
----------
       1
(1 row)
```

13.5 Nommez systématiquement vos colonnes

Il faut proscrire les « SELECT * » ou alors les « INSERT INTO ma_table VALUES(...) ». Nommez toujours explicitement vos colonnes, et donnez un nom aux colonnes calculées :

```
« SELECT valeur1, valeur2+valeur3 valeur_calculee... »
```

ou :

```
« INSERT INTO ma_table(col1,col2,col3) VALUES (v1,v2,v3) ».
```

Cela peut paraître laborieux, mais vous évitera bien des problèmes lorsque le schema d'une table sera modifié par un ALTER TABLE par exemple.

13.6 Travaillez toujours de manière ensembliste

Le développeur est souvent tenté d'exécuter une requête, puis dans le code de son application cliente de parcourir le résultat de cette requête afin d'effectuer une opération sur chaque ligne. C'est une très mauvaise idée. Bien souvent cela aboutit à des programmes dont la complexité (en nombre d'opérations exécutées,et donc en terme de performances) est exponentielle, alors qu'une seule requête bien conçue aurait pu faire l'ensemble des opérations.

13.7 Pensez au produit cartésien d'une table sur elle-même

Voici un exemple typique où beaucoup vont adopter une approche itérative au lieu d'une approche ensembliste. Considérons la table suivante. Elle contient disons, vos billets de trains. La colonne *depart_retour* contient D ou R suivant que c'est votre billet de départ ou de retour, *date_heure l'heure* à laquelle votre train part (pour les départ) ou arrive (pour les retours), et *destination*, le lieu où s'effectue votre déplacement.

```
martin=# \d voyages
                  Table "voyages"
  Attribute   |           Type           | Modifier
--------------+--------------------------+----------
depart_retour | character(1)             |
date_heure    | timestamp with time zone |
destination   | character varying(128)   |

martin=# select * from voyages;
depart_retour |       date_heure       | destination
--------------+------------------------+-------------
D             | 2002-01-02 10:00:00+01 | PARIS
R             | 2002-01-05 11:30:00+01 | PARIS
D             | 2002-01-07 06:30:00+01 | ANGERS
R             | 2002-01-08 22:25:00+01 | ANGERS
D             | 2002-01-15 06:35:00+01 | PARIS
R             | 2002-01-17 22:10:00+01 | PARIS
(6 rows)
```

Si je veux calculer la durée exacte de chacun des déplacements, beaucoup seront tentés de faire une première requête qui ramènera les billets de départ (SELECT * FROM voyages WHERE depart_retour=’D’), et ensuite effectuer depuis l’application cliente (en PHP, C++ etc.) une nouvelle requête qui ramènera le billet de retour pour chacun de billet aller trouvés (SELECT MIN(date_heure) FROM voyages WHERE depart_retour =’R’ AND date_heure > ’xx-xx-xxx’) , et ensuite calculer la différence des dates, toujours depuis l’application cliente.

Alors que les choses peuvent s’effectuer très simplement, en une seule requête :

```
martin=# select MIN(r.date_heure) - d.date_heure, d.destination
martin=# FROM voyages r, voyages d
martin-# WHERE r.date_heure > d.date_heure AND d.depart_retour=’D’
martin-# AND r.depart_retour=’R’
martin-# GROUP BY d.date_heure, d.destination;
  ?column?   | destination
-------------+-------------
3 days 01:30 | PARIS
1 day 15:55  | ANGERS
2 days 15:35 | PARIS
(3 rows)

martin=#
```

Dans cette requête, on met deux fois la même table dans la clause FROM, en prenant bien évidemment garde à *aliaser* les tables sous les noms r et d afin de pouvoir les différencier dans la requête
.

On rappelle que mettre deux tables dans la clause FROM effectue, de base, un produit cartésien. C’est à dire que pour chacun des éléments du premier ensemble, le SGBDR ramène tous les éléments du second ensemble. Comme nous voulons effectuer un calcul pour chacun des départs en déplacement, le premier ensemble est simple à définir : nous voulons tous les départs, donc WHERE d.depart_retour=’D’.

Pour le second ensemble, nous voulons le retour dont la date survient immédiatement après le départ considéré. Par « immédiatement après » je veux dire qu’il s’agit du retour le plus proche dont la date est postérieure

à la date de départ.

Donc, première chose, nous voulons des retours :

```
WHERE r.depart_retour='R'.
```

Ensuite, le retour doit être postérieur au départ considéré :

```
WHERE r.date_heure > d.date_heure.
```

Nous avons donc déja une requête qui comme premier ensemble contient tous les départs, et pour chacun des éléments de ce premier ensemble, ramènera un second ensemble constitué de tous les retours postérieurs à ce départ. Ce n'est pas encore ce qu'on veut, mais on peut exécuter la requête pour vérifier ce résultat :

```
martin=# SELECT d.date_heure as depart, r.date_heure as retour
martin=# FROM voyages d, voyages r
martin-# WHERE d.depart_retour='D' AND r.depart_retour='R'
martin=# AND r.date_heure > d.date_heure;
        depart         |         retour
-----------------------+-----------------------
2002-01-02 10:00:00+01 | 2002-01-05 11:30:00+01
2002-01-02 10:00:00+01 | 2002-01-08 22:25:00+01
2002-01-02 10:00:00+01 | 2002-01-17 22:10:00+01
2002-01-07 06:30:00+01 | 2002-01-08 22:25:00+01
2002-01-07 06:30:00+01 | 2002-01-17 22:10:00+01
2002-01-15 06:35:00+01 | 2002-01-17 22:10:00+01
(6 rows)

martin=#
```

Nous retrouvons donc dans la colonne **depart** nos trois départs différents, et dans la colonne de droite les retours postérieurs à chacun de ces départs.

Mais puisque nous voulons le plus petit (en terme de date) de ces retours pour chacun des départs considérés, un aggrégat MIN(date_heure) est tout indiqué. Reste à savoir sur quel critère de groupe nous voulons effectuer l'aggrégat. Et bien puisque nous voulons « le plus petit retour **pour chacun des départs** », c'est simplissime : GROUP BY d.date_heure, d.destination.

D'une manière générale ces jointures d'une table sur elle-même sont très pratiques lorsqu'on doit ramener des lignes en tenant compte d'autres lignes de la même table.

Voilà, j'espère que ces quelques conseils vous seront profitables. Je vous dis à bientôt pour d'autres sujets...