Bases de données – programmation

Tiago de Lima

UNIVERSITÉ D'ARTOIS
UFR des Sciences
Master Informatique
Année 1, Semestre 1

Année universitaire 2018–2019

Section 1

Présentation de l'unité

Objectif de l'unité

Nous abordons principalement dans ce module les techniques liées aux notions de transaction, de gestion d'accès concurrents, de reprise après panne, de bases de données distribuées, ainsi que les aspects procéduraux des bases de données (déclencheurs, PL/SQL).

- ...
- Aspects Procéduraux
  - PL/SQL
  - Procédures stockées
  - Déclencheurs (Triggers)
- Optimisation

Charge horaire et modalités de contrôle de connaissance (partie 2)

ECTS : 5

CM : 1,5 h $\times$ 3 sem $=$ 4,5 h (Semaines 1–3)

TD : 1,5 h $\times$ 3 sem $=$ 4,5 h (Semaines 1–3)

TP : 1,5 h $\times$ 5 sem $=$ 7,5 h (Semaines 2–6)

Évaluation : contrôle continu + examen

Plan

Sem.	CM	TD	TP
1	Fonctions	Fonctions	Fonctions
2	Déclencheurs	Déclencheurs	Fonctions
3	Optimisation	Optimisation	Déclencheurs
4	—	—	Optimisation
5	—	—	Catalogue

Site web de l'unité

Moodle : **`http://moodle.univ-artois.fr/`**

Vous trouverez :

- support des cours ;
- exercices ;
- notes ;
- d'autres informations.

Lecture supplémentaire

Site web officiel : PostgreSQL :

http://www.postgresql.org

http://www.postgresql.org/docs/

Site web en français : PostgreSQLFr :

http://www.posgresql.fr

http://docs.postgresql.fr

Cours 1

Fonctions

Sommaire

2. Étendre SQL

3. PL/pgSQL – Langage procédural PostgreSQL

Section 2

Étendre SQL

Étendre SQL

PostgreSQL est un SGBD extensible.

Il enregistre dans le **catalogue** non seulement l'information concernant les tables et les colonnes, mais aussi :

- les types de données
- les fonctions
- les méthodes d'accès
- etc.

Ainsi que les tables, ces informations peuvent être étendues et modifiées par l'utilisateur.

Le catalogue

Commandes pour consulter le catalogue :

```
> \?
...
  (options: S = show system objects, + = additional detail)
  \d[S+]                 list tables, views, and sequences
  \d[S+]  NAME           describe table, view, sequence, or
   index
  \da[S]  [PATTERN]      list aggregates
  \db[+]  [PATTERN]      list tablespaces
  \dc[S+] [PATTERN]      list conversions
  \dC[+]  [PATTERN]      list casts
  \dd[S]  [PATTERN]      show object descriptions not
   displayed elsewhere
  \ddp    [PATTERN]      list default privileges
  \dD[S+] [PATTERN]      list domains
  \det[+] [PATTERN]      list foreign tables
  \des[+] [PATTERN]      list foreign servers
  \deu[+] [PATTERN]      list user mappings
  \dew[+] [PATTERN]      list foreign-data wrappers
  \df[antw][S+] [PATRN]  list [only agg/normal/trigger/window]
  ...
```

Le catalogue

```
> \d
                      List of relations
 Schema |                 Name                 |   Type   | Owner
--------+--------------------------------------+----------+--------
 public | adherent                             | table    | delima
 public | adherent_id_adherent_seq             | sequence | delima

> \df
                                 List of functions
 Schema |      Name        | Result data type | Argument data
--------+------------------+------------------+---------------
 public | double           | integer          | x integer
 public | double_amende    | integer          | a adherent

> \dT
            List of data types
 Schema |        Name         | Description
--------+---------------------+-------------
 public | complexe            |
 public | element_inventaire  |
(2 rows)
```

Types de données

Les **types de base** sont implantés au niveau du langage SQL (ex. : `integer`, `varchar`, `boolean`, etc.).

Les **types composites**, ou types lignes, sont crées chaque fois qu'un utilisateur crée une table.

Il est également possible de définir un type composite autonome sans table associée.

Types de données

Exemple de création d'un type composite autonome :

```
1 create type complexe as (
2     r        double precision,
3     i        double precision
4 );
5 create type element_inventaire as (
6     nom              text,
7     id_fournisseur   integer,
8     prix             numeric
9 );
```

```
> \dT
            List of data types
 Schema |         Name         | Description
--------+----------------------+-------------
 public | complexe             |
 public | element_inventaire   |
(2 rows)
```

Types de données

Un **domaine** est un type de base qui possède des contraintes qui restreignent ses valeurs. Ceux-ci peuvent être créés à l'aide de l'instruction `CREATE DOMAIN`.

Les **pseudo-types** ne peuvent pas apparaître comme champs de table ou comme attributs de types composites, mais ils peuvent être utilisés comme des arguments et des résultats de fonctions.

Pseudo-types intéressants (les **types polymorphes**) :

- `anyelement`
- `anyarray`
- `anynonarray`
- `anyenum`
- `anyrange`

Fonctions utilisateur

Il y a quatre types de fonctions utilisateurs :

- fonctions en langage requête (SQL)
- fonctions en langage procédural (PL/pgSQL, PL/Tcl, etc.)
- fonctions internes
- fonctions en langage C

La création d'une fonction utilisateur se fait avec CREATE FUNCTION.

Syntaxe (simplifiée) :

```
CREATE [ OR REPLACE ] FUNCTION
nom ( [ mode ] [ nomarg ] type [ DEFAULT exp ] [, ...] )
[ RETURNS typeret ]
AS 'definition'
LANGUAGE lang
```

Fonctions utilisateur

Exemple :

```
1 create function ajoute(a integer, b integer) returns integer
2 as 'select a + b;'
3 language sql;
```

```
> \df
                              List of functions
 Schema | Name   | Result data type | Argument data types  |
--------+--------+------------------+----------------------
 public | ajoute | integer          | a integer, b integer |
(1 row)

> select ajoute(1, 2);
 ajoute
-------
      3
(1 row)
```

Fonctions en langage requête

Les fonctions SQL exécutent une liste arbitraire d'instructions SQL et retournent le résultat de la dernière requête de cette liste.

Si la dernière requête de la liste ne retourne aucune ligne, la valeur `NULL` est retournée.

Une fonction SQL peut retourner un ensemble :

```
RETURNS SETOF type
```

ou de façon équivalente :

```
RETURNS TABLE(colonnes)
```

Dans ce cas, toutes les lignes de la dernière requête sont retournées.

Dans le cas d'un résultat simple (pas d'ensemble), la première ligne du résultat de la dernière requête sera retournée.

Attention : « la première ligne » d'un résultat multiligne n'est pas bien définie sauf si `ORDER BY` est utilisé.

Arguments pour les fonctions SQL

Si le nom de l'argument est le même que celui d'une colonne dans la commande SQL en cours, le nom de la colonne est prioritaire.

Pour contourner, qualifiez le nom de l'argument avec le nom de la fonction (ex. : `nom_fonction.nom_argument`).

Les arguments peuvent être référencés en utilisant soit leurs noms soit leurs numéros :

- `$1` fait référence au premier argument
- `$2` fait référence au second argument
- et ainsi de suite.

Ceci fonctionnera si l'argument ait été déclaré avec un nom ou pas.

Les arguments peuvent seulement être utilisés comme valeurs de données, et non pas comme identifiants. Donc, ceci ne fonctionnera pas (où `arg` est l'argument de la fonction) :

```
1 insert into arg values (42);
```

Arguments pour les fonctions SQL

Il est possible de décrire les résultats d'une fonction avec des paramètres en sortie :

```
1 create function ajoute_produit (in x int, in y int,
2                                 out s int, out p int)
3 as 'select x + y, x * y;'
4 language sql;
```

```
> select ajoute_produit(1, 2);
 ajoute_produit
----------------
          (3,2)
(1 row)
```

Exemples de fonctions SQL

Débite un compte bancaire et retourne la nouvelle valeur du solde (notez la clause `RETURNING` en ligne 7) :

```
1 create function debiter(no_compte integer, val numeric)
2 returns integer
3 as $$
4   update compte
5     set solde = solde - val
6     where no_compte = debiter.no_compte
7   returning solde;
8 $$ language sql;
```

Pour débiter le compte 17 de 100 000 euros :

```
> select debiter(17, 100000);
```

Exemples de fonctions SQL

Double l'amende de l'adherent (version 1) :

```
1 create function double(x integer) returns integer
2 as 'select 2 * x;'
3 language sql;
```

```
> select nom_adherent, double(amende_adherent) from adherent;
    nom_adherent     | double
---------------------+--------
 Patamob             |      0
 Zeublouse           |      0
 Rivenbusse          |      0
 Comindieu           |      0
 Ardelpic            |      0
 Peulafenetre        |     40
 Locale              |      0
 Bierrekeuchprefere  |      0
(8 rows)
```

Exemples de fonctions SQL

Double l'amende de l'adherent (version 2) :

```
1 create function double_amende(a adherent) returns integer
2 as 'select a.amende_adherent * 2;'
3 language sql;
```

```
> select nom_adherent, double_amende(adherent.*)
  from adherent;
    nom_adherent    | double_amende
--------------------+---------------
 Patamob            |             0
 Zeublouse          |             0
 Rivenbusse         |             0
 Comindieu          |             0
 Ardelpic           |             0
 Peulafenetre       |            40
 Locale             |             0
 Bierrekeuchprefere |             0
(8 rows)
```

Exemples de fonctions SQL

En effet, cela peut simuler l'ajout d'une nouvelle colonne dans la table :

```
> select a.nom_adherent, a.double_amende from adherent a;
    nom_adherent    | double_amende
--------------------+---------------
 Patamob            |             0
 Zeublouse          |             0
 Rivenbusse         |             0
 Comindieu          |             0
 Ardelpic           |             0
 Peulafenetre       |            40
 Locale             |             0
 Bierrekeuchprefere |             0
(8 rows)
```

Exemples de fonctions SQL

Compte le nombre d'emprunts d'un adherent :

```
1 create function nb_emprunts(a adherent) returns bigint
2 as $$
3   select count(id_emprunt)
4     from emprunt
5     where id_adherent = a.id_adherent;
6 $$ language sql;
```

```
> select a.nom_adherent, a.nb_emprunts from adherent a;
    nom_adherent     | nb_emprunts
---------------------+-------------
 Patamob             |           0
 Zeublouse           |           0
 Rivenbusse          |           3
 Comindieu           |           1
 Ardelpic            |           0
 Peulafenetre        |           0
 Locale              |           0
 Bierrekeuchprefere  |           2
(8 rows)
```

Exemples de fonctions SQL

Les plus gros emprunteurs (notez le SETOF) :

```
1 create function max_emprunteurs() returns setof adherent
2 as $$
3   select a.* from adherent a natural join emprunt
4   group by id_adherent
5   having count(id_adherent) >=
6     (select max(adherent.nb_emprunts) from adherent);
7 $$ language sql;
```

```
> select max_emprunteurs();
    max_emprunteurs
------------------------
 (3,Rivenbusse,Elsa,0)
(1 row)
```

Exemples de fonctions SQL

```
> select * from max_emprunteurs();
 id_adherent | nom_adherent | prenom_adherent | amende_adherent
-------------+--------------+-----------------+----------------
           3 | Rivenbusse   | Elsa            |               0
(1 row)

> select nom_adherent from max_emprunteurs();
 nom_adherent
--------------
 Rivenbusse
(1 row)

> select nom_adherent, titre from max_emprunteurs()
         natural join emprunt
         natural join livre
         natural join oeuvre;
 nom_adherent |           titre
--------------+---------------------------
 Rivenbusse   | Dans les bois eternels
 Rivenbusse   | La perle et le croissant
 Rivenbusse   | Bienvenue au club
(3 rows)
```

Fonctions en langage procédural

PostgreSQL autorise l'écriture de fonctions définies par l'utilisateur dans d'autres langages que SQL.

Ces autres langages sont appelés des langages procéduraux (PL).

Ces langages ne sont pas compilés dans le serveur PostgreSQL. Ils sont fournis comme des modules chargeables.

Il y a actuellement quatre langages procéduraux disponibles dans la distribution PostgreSQL standard :

- PL/pgSQL
- PL/Tcl
- PL/Perl
- PL/Python

Dans cette unité, nous allons étudier PL/pgSQL.

Section 3

PL/pgSQL – Langage procédural PostgreSQL

PL/pgSQL – Langage procédural PostgreSQL

Les objectifs de PL/pgSQL :

- être utilisé pour la création des fonctions standards et triggers
- ajouter des structures de contrôle au langage SQL
- permettre d'effectuer des traitements complexes
- hériter de tous les types, fonctions et opérateurs définis par les utilisateurs
- être défini comme digne de confiance par le serveur
- être facile à utiliser

Quelques avantages :

- Les allers/retours entre le client et le serveur sont éliminés.
- Il n'est pas nécessaire de traiter ou transférer entre le client et le serveur les résultats intermédiaires dont le client n'a pas besoin.
- Une augmentation considérable des performances.

Structure d'une fonction

Syntaxe :

```
[ <<label>> ]
[ DECLARE
    déclarations ]
BEGIN
    instructions
END
```

Exemple :

```
1 create function prix_ttc(prix_ht real) returns real
2 as $$
3   declare
4     taxe real;
5   begin
6     taxe := 0.196;
7     return prix_ht * (1 + taxe);
8   end
9 $$ language plpgsql;
```

Structure d'une fonction

Chaque déclaration et chaque expression au sein du bloc est terminée par un point-virgule.

Un bloc qui apparaît à l'intérieur d'un autre bloc doit avoir un point-virgule après `END`.

Commentaires :

- `--` : commence un commentaire qui fini en fin de ligne.
- `/*` et `*/` : pour un bloc de commentaire (plusieurs lignes).

Tout ce qui n'est pas reconnu comme une instruction PL/pgSQL est présumé être une commande SQL et est envoyé au moteur principal de bases de données.

Déclarations

Toute les variables utilisées dans un bloc doivent être déclarées (sauf certaines variables de boucles `FOR`).

Syntaxe (simplifiée) :

```
nom [ CONSTANT ] type [ NOT NULL ] [ DEFAULT expr ];
```

Exemples :

```
1 id_utilisateur integer;
2 quantite numeric(5);
3 url varchar;
4 ma_ligne nom_table%ROWTYPE;
5 mon_champ nom_table.nom_colonne%TYPE;
6 une_ligne RECORD;
```

Si pas de valeur par défaut, la variable est assignée la valeur `NULL`.

Si la contrainte `NOT NULL` est spécifiée, il faut obligatoirement préciser une valeur (non nulle) par défaut et déclenche une erreur si l'on affecte la valeur `NULL`.

Instructions de base

Affectation :

```
variable := expression;
```

Requête avec résultat :

```
SELECT expressions_select INTO [STRICT] cible FROM ...;
INSERT ... RETURNING expressions INTO [STRICT] cible;
UPDATE ... RETURNING expressions INTO [STRICT] cible;
DELETE ... RETURNING expressions INTO [STRICT] cible;
```

où cible peut être une variable de type record, row ou une liste de variables ou de champs record/row séparées par des virgules.

Exemples :

```
1 select * into monrec from emp where nom = mon_nom;
2 select * into strict monrec from emp where nom = mon_nom;
```

Instructions de base

Si l'option `STRICT` est indiquée, la requête doit retourner exactement une ligne. Dans le cas contraire, une erreur sera rapportée à l'exécution.

Si `STRICT` n'est pas spécifié dans la clause `INTO`, alors cible sera configuré avec la première ligne retournée par la requête ou à `NULL` si la requête n'a retourné aucune ligne.

Notez que « la première ligne » n'est bien définie que si vous avez utilisé `ORDER BY`.

Vous pouvez vérifier la valeur de la variable spéciale `FOUND` pour déterminer si une ligne a été retournée :

```
1 select * into monrec from emp where nom = mon_nom;
2 if not found then
3     raise exception 'employe introuvable';
4 end if;
```

Instructions de base

Requête sans résultat :

```
PERFORM requête;
```

Écrivez la requête de la même façon qu'une instruction `SELECT` mais remplacez le mot clé initial `SELECT` avec `PERFORM`.

Une instruction SQL qui peut retourner des lignes comme `SELECT` sera rejetée comme une erreur si elle n'a pas de clause `INTO`.

Instructions dynamiques :

```
EXECUTE chaîne [ INTO [STRICT] cible ] USING exp, ... ];
```

Les expressions `USING` fournissent des valeurs à insérer dans l'instruction.

```
1 execute
2   'select * from adherent where prix < $1 and date < $2'
3   into resultat
4   using max_prix, max_date;
```

Instructions de base

Notez que `USING` peut seulement être utilisé avec des valeurs de données.

Si vous voulez utiliser des noms de tables ou colonnes déterminés dynamiquement, vous devez les insérer dans la chaîne.

```
1 execute 'select * from '
2   || nomtable::regclass
3   || ' where prix < $1 and date < $2'
4   into resultat
5   using max_prix, max_date;
```

Instructions de base

Obtention du status du résultat :
La première méthode pour déterminer l'effet d'une instruction est d'utiliser `GET DIAGNOSTICS`.

Syntaxe :

```
GET [ CURRENT ] DIAGNOSTICS variable = élément , ... ;
```

Les éléments d'état actuellement disponibles sont :

- `ROW_COUNT` : le nombre de lignes traitées par la dernière instruction.
- `RESULT_OID`, l'OID de la dernière ligne insérée par l'instruction SQL la plus récente dans une table contenant des OID.

La seconde méthode est la variable spéciale `FOUND` de type boolean.

Instructions de base

`FOUND` est initialisée à `false` au début de chaque instruction PL/pgSQL, ensuite :

- `SELECT INTO`, `PERFORM` et `FETCH` :
  `true` si au moins une ligne est retournée, `false` sinon.
- `UPDATE`, `INSERT` et `DELETE` :
  `true` si au moins une ligne est affectée, `false` sinon.
- `MOVE` : `true` si repositionne le curseur avec succès, `false` sinon.
- `FOR` et `FOREACH` : `true` s'il y a au moins une itération `false` sinon.
  `FOUND` n'est pas modifié à l'intérieur de la boucle, bien qu'il pourrait être modifié par l'exécution d'autres requêtes dans la boucle.
- `RETURN QUERY` et `RETURN QUREY EXECUTE` :
  `true` si au moins une ligne est retournée, `false` sinon.

Les autres instructions PL/pgSQL ne changent pas l'état de `FOUND`.

`EXECUTE` modifie la sortie de `GET DIAGNOSTICS` mais pas `FOUND`.

`FOUND` est une variable locale à l'intérieur de chaque fonction PL/pgSQL.

Instructions de base

Ne rien faire du tout :

```
NULL;
```

Structures de contrôle

Retour d'une fonction :

```
RETURN expression;
```

```
RETURN NEXT expression;
```

Quand une fonction qui renvoye `SETOF type`, les éléments individuels à renvoyer sont spécifiés par une séquence de `RETURN NEXT` suivies de la commande finale `RETURN`.

Si vous déclarez la fonction avec des paramètres en sortie, écrivez seulement `RETURN` sans expression.

`RETURN` peut être utilisée pour quitter rapidement une fonction du type `VOID`.

Structures de contrôle

IF :

```
IF expr_booleenne THEN
   instructions
[ ELSIF expr_booleenne THEN
   instructions
... ]
[ ELSE
   instructions ]
END IF;
```

```
1 if nombre = 0 then
2     resultat := 'zero';
3 elsif nombre > 0 then
4     resultat := 'positif';
5 elsif nombre < 0 then
6     resultat := 'negatif';
7 else
8     resultat := 'NULL';
9 end if;
```

Structures de contrôle

CASE simple :

```
CASE expr_recherche
   WHEN expr , ... THEN
      instructions
   [ WHEN expr , ... THEN
      instructions
   ... ]
   [ ELSE
      instructions ]
END CASE;
```

```
1 case x
2     when 1, 2 then
3         msg := 'un ou deux';
4     else
5         msg := 'autre valeur que un ou deux';
6 end case;
```

Si aucune correspondance n'est trouvée et il n'y a pas de bloc ELSE, une exception CASE_NOT_FOUND est levée.

Structures de contrôle

CASE recherché :

```
CASE
   WHEN expr_booléenne THEN
      instructions
   [ WHEN expr_booléenne THEN
      instructions
   ... ]
   [ ELSE
      instructions ]
END CASE;
```

```
1 case
2     when x between 0 and 10 then
3         msg := 'valeur entre zero et dix';
4     when x between 11 and 20 then
5         msg := 'valeur entre onze et vingt';
6 end case;
```

Si aucune correspondance n'est trouvée et il n'y a pas de bloc ELSE, une exception CASE_NOT_FOUND est levée.

Boucles simples

LOOP :

```
[ <<label>>]
LOOP
    instructions
END LOOP [ label ];
```

EXIT :

```
EXIT [ label ] [ WHEN expr_booléenne ];
```

CONTINUE :

```
CONTINUE [ label ] [ WHEN expr_booléenne ];
```

```
1 <<un_bloc>>
2 begin
3     -- quelques traitements
4     if stocks > 100000 then
5         exit un_bloc;  -- cause la sortie du bloc
6     end if;
7 end;
```

Boucles simples

WHILE :

```
[ <<label>> ]
WHILE expr_booléenne LOOP
    instructions
END LOOP [ label ];
```

FOR avec entier :

```
[ <<label>> ]
FOR nom IN [ REVERSE ] expr .. expr [ BY expr ] LOOP
    instructions
END LOOP [ label ];
```

```
1 for i in reverse 10..1 by 2 loop
2     -- prend les valeurs 10,8,6,4,2 dans la boucle
3 end loop;
```

REVERSE indique que la valeur de l'étape est soustraite, plutôt qu'ajoutée. Si la limite inférieure est supérieure à la limite supérieure (ou inférieure dans le cas du REVERSE), le corps de la boucle n'est pas exécuté du tout. Aucune erreur n'est renvoyée.

Boucles dans les résultats de requête

FOR avec requête :

```
[ <<label>> ]
FOR cible IN requête LOOP
   instructions
END LOOP [ label ];
```

`cible` est une variable de type record, row ou une liste de variables scalaires séparées par une virgule.

La cible est affectée successivement á chaque ligne résultant de la requête et le corps de la boucle est exécuté pour chaque ligne.

Si la boucle est terminée par une instruction `EXIT`, la dernière valeur ligne affectée est toujours accessible après la boucle.

Boucler dans des tableaux

FOREACH :

```
[ <<label>> ]
FOREACH cible [ SLICE nombre ] IN ARRAY expr LOOP
   instructions
END LOOP [ label ];
```

`cible` est peut être une variable scalaire ou une liste de variables lors d'une boucle dans un tableau de valeurs composites.

Avec une valeur `SLICE` positive, `FOREACH` itère au travers des morceaux du tableau plutôt que des éléments seuls. La variable cible doit être un tableau et elle reçoit les morceaux successifs de la valeur du tableau.

Boucler dans des tableaux

Exemple :

```
1  create function print_lignes(int[][]) returns void
2  as $$
3      declare
4          x int[];
5      begin
6          foreach x slice 1 in array $1 loop
7              raise notice 'ligne = %', x;
8          end loop;
9      end;
10 $$ language plpgsql;
```

```
> select print_lignes(array[[1,2,3],[4,5,6],[7,8,9],
        [10,11,12]]);

NOTICE:  ligne = {1,2,3}
NOTICE:  ligne = {4,5,6}
NOTICE:  ligne = {7,8,9}
NOTICE:  ligne = {10,11,12}
```

Récupérer les erreurs

EXCEPTION :

```
[ <<label>> ]
[ DECLARE
   déclarations ]
BEGIN
   instructions
EXCEPTION
   WHEN condition [ OR condition ... ] THEN
      instructions
   [ WHEN condition [ OR condition ... ] THEN
      instructions
    ... ]
END;
```

Récupérer les erreurs

Exemple :

```
1  ...
2  insert into mon_tableau(prenom, nom) values('Tom', 'Jones');
3  begin
4    update mon_tableau set prenom = 'Joe' where nom = 'Jones';
5    x := x + 1;
6    y := x / 0;
7    exception
8      when division_by_zero then
9        raise notice 'recuperation de l''erreur...';
10   return x;
11 end;
12 ...
```

L'exécution de la ligne 6 échouera avec une erreur `division_by_zero`. Cela sera récupérée par la clause `EXCEPTION`. La valeur retournée par `RETURN` sera la valeur incrémentée de `x` mais les effets de l'instruction `UPDATE` auront été annulés. L'instruction `INSERT` précédant le bloc ne sera pas annulée.

Utilisez l'instruction `RAISE` pour rapporter des messages et lever des erreurs.

Syntaxe (simplifiée) :

```
RAISE [ niveau ] 'format' [, expr ... ];
```

Les niveaux :

- `DEBUG` et `LOG` : écrit un message sur le fichier de log du serveur.
- `INFO`, `NOTICE` et `WARNING` : idem mais envoie également le message au client.
- `EXCEPTION` : (niveau par défaut) envoie un message et annule la transaction en cours.

Exemple :

```
1 raise exception 'Nonexistent ID --> %', usr_id
```

Curseurs

Les **curseurs** permettent la lecture du résultat d'une requête quelques lignes à la fois.

Cela évite le surcharge de mémoire quand le résultat contient un grand nombre de lignes.

Les boucles `FOR` en PL/pgSQL utilisent automatiquement un curseur en interne pour éviter les problèmes de mémoire.

Néanmoins, les curseurs sont un moyen efficace de retourner de grands ensembles de lignes à partir des fonctions.

Déclaration des variables curseur

Syntaxe :

```
nom [ [ NO ] SCROLL ] CURSOR [ (arguments) ] FOR requête;
```

`SCROLL` indique que le curseur sera capable d'aller en sens inverse.

`NO SCROLL` indique que les récupérations en sens inverses seront rejetées.

Si rien n'est indiqué, cela dépendra de la requête.

`arguments` est une liste de paires `nom type` qui définit les noms devant être remplacés par les valeurs des paramètres dans la requête.

```
1 declare
2     curs1 refcursor;        -- variable curseur non liee
3     curs2 cursor for select * from tenk1;
4     curs3 cursor (cle integer) for select * from tenk1
5                                    where unique1 = cle;
6     x integer;
7     curs4 cursor for select * from tenk1 where unique1 = x
```

Ouverture de curseurs

Avant d'être utilisé le curseur doit être ouvert (c'est l'action équivalente de l'instruction SQL `DECLARE CURSOR`).

Syntaxe (curseur non lié) :

```
OPEN var_curseur [ [ NO ] SCROLL ] FOR requête;
```

Syntaxe (curseur lié) :

```
OPEN var_curseur [ ( [ nom_arg := ] val_arg, ... ) ] ];
```

Exemples :

```
1 open curs1 for select * from foo where cle = 42;
2 open curs2;
3 open curs3(42);
4 open curs3(cle := 42);
5 x := 42;
6 open curs4;
```

Utilisation de curseurs

Syntaxe :

```
FETCH [ direction { FROM | IN } ] curseur INTO cible;
```

Récupère la prochaine ligne à partir d'un curseur et la place dans `cible`.

S'il n'y a pas de ligne suivante, `cible` est affectée à `NULL`.

`direction` peut être une des variantes suivantes :
`NEXT`, `PRIOR`, `FIRST`, `LAST`, `ABSOLUTE nombre`, `RELATIVE nombre`, `FORWARD` ou `BACKWARD`.

Omettre `direction` est équivalent à `NEXT`.

Les valeurs direction qui nécessitent d'aller en sens inverse risquent d'échouer sauf si le curseur a été déclaré ou ouvert avec l'option `SCROLL`.

```
1 fetch curs1 into rowvar;
2 fetch curs2 into foo, bar, baz;
3 fetch last from curs3 into x, y;
4 fetch relative -2 from curs4 into x;
```

Utilisation de curseurs

```
MOVE [ direction { FROM | IN } ] curseur;
```

Fonctionne exactement comme l'instruction `FETCH` sauf qu'elle ne fait que repositionner le curseur et ne retourne donc pas les lignes du déplacement.

La variable `FOUND` peut être lue pour vérifier s'il y avait bien les lignes correspondant au déplacement.

Exemples :

```
1 move curs1;
2 move last from curs3;
3 move relative -2 from curs4;
4 move forward 2 from curs4;
```

Fermeture de curseurs

Syntaxe :

```
CLOSE curseur;
```

Ferme le un curseur ouvert.

Ceci peut être utilisé pour libérer des ressources avant la fin de la transaction ou pour libérer la variable curseur pour pouvoir la réouvrir.

Exemple

```
1 close curs1;
```

Boucler dans les résultats d'un curseur

Syntaxe :

```
[ <<label>> ]
FOR var_record IN curseur [ ( [ arg := ] val, ... ) ]
LOOP
    instructions
END LOOP [ label ];
```

`curseur` doit être lié mais pas ouvert.

L'instruction `FOR` ouvre automatiquement le curseur, et le ferme en sortie de la boucle.

`var_record` est définie automatiquement avec le type record et existe seulement dans la boucle.

Chaque ligne renvoyée par le curseur est successivement affectée à la variable d'enregistrement et le corps de la boucle est exécuté.

Boucler dans les résultats d'un curseur

Exemple :

```
1  -- Changer le prix des tous les livres d'un certain theme.
2  create function change_prix(t varchar, p real) returns void
3  as $$
4      declare
5          curs1 cursor(th varchar) for
6              select livre.*
7                  from livre natural join oeuvre
8                      natural join theme
9                  where theme.nom_theme = t
10                     and ouvre.id_theme = theme.id_theme
11                     and livre.id_oeuvre = ouvre.id_oeuvre;
12     begin
13         for ligne in curs1(t) loop
14             update livre set prix = p
15                 where ligne.id_livre = livre.id_livre;
16         end loop;
17     end;
18 $$ language plpgsql;
```

Cours 2

Déclencheurs

Aperçu

Un **déclencheur** (ou **trigger**) spécifie que la base de données doit exécuter automatiquement une fonction donnée chaque fois qu'un certain type d'opération est exécuté.

Ceci sert, par exemple, à implémenter une contrainte d'intégrité de la base qui n'a pas pu être exprimée par une contrainte sur une table.

On différencie la fonction déclencheur (la fonction qui est appelée par le déclencheur) du déclencheur lui-même.

Les fonctions pour déclencheurs peuvent être écrites dans la plupart des langages procéduraux disponibles incluant PL/pgSQL, PL/Tcl, PL/Perl, et PL/Python ou bien en C.

Fonctions pour déclencheurs en PL/pgSQL

Une fonction pour déclencheurs doit avoir une signature particulière :

- pas d'argument ;
- type de retour `TRIGGER`.

Exemple :

```
1 create function nouvel_etat() returns trigger
2 as $$
3 ... mise a jour etc...
4 $$ language plpgsql;
```

Une fonction pour déclencheur **doit être créée avant** de pouvoir être associé à un déclencheur.

Déclencheurs

Syntaxe (simplifiée) :

```
CREATE TRIGGER nom
    { BEFORE | AFTER } { evenement [ OR ... ] }
    ON nom_table
    [ FOR EACH { ROW | STATEMENT } ]
    EXECUTE PROCEDURE nom_fonction()
```

où evenement fait partie de :

- INSERT
- UPDATE [OF nom_colonne, [,...]]
- DELETE
- TRUNCATE

Exemple :

```
1 create trigger tr_nouvel_etat
2     after insert on histo_article
3     for each row
4     execute procedure nouvel_etat();
```

Déclencheurs

Quand une fonction PL/pgSQL est appelée en tant que déclencheur, plusieurs variables spéciales sont créées automatiquement dans le bloc de plus haut niveau. Parmi elles :

- `NEW`
  Type `RECORD`. Cette variable contient la nouvelle ligne pour les opérations `INSERT` et `UPDATE` et est non initialisée pour l'opération `DELETE`.
- `OLD`
  Type `RECORD`. Cette variable contient l'ancienne ligne pour les opérations `UPDATE` et `DELETE` et est non initialisée pour l'opération `INSERT`.

Exemple

```
1  create function nouvel_etat() returns trigger as $$
2  declare
3      date_d date;
4      date_f date;
5      etat int;
6  begin
7      select date_debut_etat, id_etat, date_fin_etat
8          into date_d, etat, date_f
9          from histo_article
10         where id_article = new.id_article
11         order by date_debut_etat desc offset 1 limit 1;
12     if found then
13         if date_f is null then
14             update histo_article
15                 set date_fin_etat = current_date
16                 where id_etat = etat
17                 and id_article = new.id_article
18                 and date_debut_etat = date_d;
19         end if;
20     end if;
21     return null;
22 end;
```

Déclencheur

Une fonction pour déclencheurs doit renvoyer soit `NULL` soit une valeur `RECORD` ayant exactement la structure de la table pour laquelle le déclencheur a été lancé.

Déclencheurs `BEFORE` :

- `NULL` : annule l'action qui a déclenché la fonction ;
- non `NULL` : l'opération se déroule avec la valeur renvoyée.

Pour les déclencheurs `AFTER`, la valeur de retour est toujours ignorée. Néanmoins, un déclencheur peut toujours annuler l'opération en cours avec une exception.

Déclencheurs

Exemple :

```
1  create table comande_article(id_article int,
2                               id_commande int,
3                               qte int);
4
5  create function au_mois_dix() returns trigger
6  as $$
7  begin
8      if (new.qte < 10) then
9          new.qte = 10;
10     end if;
11     return new;
12 end;
13 $$ language plpgsql
14
15 create trigger tr_commande
16     before insert or update
17     on commande_article
18     for each row
19     execute procedure au_moins_dix();
```

Déclenchements en cascade

Plusieurs déclencheurs peuvent être lancés par une même action. Ils sont exécutés les uns à la suite des autre par **ordre alphabétique (! !)** sur leur nom. L'élément retourné par un déclencheur devient l'élément entrant (`NEW`) du suivant.

Le premier déclencheur qui retourne `NULL` annule l'ensemble de l'action sur l'enregistrement courant.

La programmation des déclencheurs nécessite donc d'une analyse fine de leurs enchaînements (et de préférence simple, avec peu de niveau de déclenchement en cascade), et doit être bien documentée.

Déclencheurs par instruction

Les déclencheurs par instruction (`FOR EACH STATEMENT`) sont exécutés une seule fois pour une instruction, même si cette instruction concerne plusieurs enregistrements.

Les déclencheurs `BEFORE` par instruction sont exécutés après tous les déclencheurs `BEFORE` par enregistrement (et de même pour les déclencheurs `AFTER` par enregistrement et par instruction).

Les pseudo-variables `NEW` et `OLD` ne sont pas accessibles dans un déclencheur par instruction.

Les déclencheurs par instruction sont utiles notamment pour les actions qui nécessitent d'accéder à la table concernée par le déclencheur.

Déclencheurs par instruction

Par exemple, dans la base `biblio`, nous voulons enregistrer quelques informations statistiques dans la table `proprietes(id, nom, valeur)`. Notamment, le nombre de livres empruntés actuellement :

```
1  create function maj_nb_emprunts() returns trigger as $$
2  declare
3      nb int;
4  begin
5      select count(id_livre) into nb from emprunt;
6      update proprietes
7          set valeur = nb
8          where nom = 'nb_emprunts';
9  end;
10 $$ language plpgsql;
11
12 create trigger tr_maj_nb_emprunts
13     after insert or delete
14     on emprunt
15     for each statement
16     execute procedure maj_nb_emprunts();
```

Règles de visibilité de modifications

Un déclencheur par enregistrement ou par instruction `AFTER` voit toutes les modifications effectuées.

Un déclencheur par instruction `BEFORE` ne voit aucune modification.

Un déclencheur par enregistrement `BEFORE` :

- ne voit pas les modifications induites par l'enregistrement qu'il est en train de traiter (puisqu'elles n'ont pas encore été faites) ;
- voit toutes les modifications faites par les enregistrements qui ont déjà été traités (dans la même instruction).
  Attention : cela peut s'avérer dangereux puisqu'il n'est pas possible de savoir dans quel ordre les enregistrements seront traités (il n'y a pas de clause `ORDER BY` sur un `INSERT`, `UPDATE` ou `DELETE`).

Exemple

```
1  create or replace function ajout_emprunt() returns trigger
2  as $$
3  declare
4      v_du Adherent.du%TYPE;
5      v_nb_emprunts integer;
6      v_sorti Livre.sorti%TYPE;
7  begin
8      select du into v_du
9          from Adherent
10         where id_adherent = new.id_adherent;
11
12     if not found then
13         raise exception 'Adherent inconnu.';
14     end if;
15     if v_du <> 0 then
16         raise notice 'L adherent doit %', v_du;
17         return null;
18     end if;
```

Exemple (cont.)

```
    select sorti into v_sorti
        from Livre
        where id_livre = new.id_livre;
    if not found then
        raise notice 'Livre inconnu : %', new.id_livre;
        return null;
    end if;
    if v_sorti then
        raise exception 'Livre sorti.';
    end if;

    select count(*) into v_nb_emprunts
        from Emprunt
        where id_adherent = new.id_adherent;
    if ((v_nb_emprunts is not null) and
        (v_nb_emprunts > 4)) then
        raise notice 'Trop d emprunts en cours.';
        return null;
    end if;
    return new;
end;
$$ language plpgsql;
```

Exemple (cont.)

```
1 create trigger ajout_emprunt
2     before insert
3     on table Emprunt
4     for each row
5     execute procedure ajout_emprunt();
```

Informations additionnelles du déclencheur

D'autres variables disponibles pour les fonctions pour déclencheurs :

- `TG_OP` : quel événement à déclenché la fonction (`INSERT`, `UPDATE` ou `DELETE`) ;
- `TG_WHEN` : quand la fonction à été déclenchée (`AFTER` ou `BEFORE`) ;
- `TG_LEVEL` : le niveau de déclenchement (`ROW` ou `STATEMENT`) ;
- `TG_RELNAME` : le nom de la table associée au déclencheur.
- etc.

Cours 3

Optimisation

Ce qui se passe lors d'une requête

1. **Connexion :** le client envoie une requête au serveur.
2. **Analyse syntaxique :**
   - 2.1 Vérification de la correction de la requête.
   - 2.2 Vérification de l'existence des tables et des attributs.
   - 2.3 Simplifications (ex. : `x<=123` et `x>=123` simplifié en `x=123`).
   - 2.4 Détection des incohérences (ex. : `x=123` et `x=124`).
3. **Création de l'arbre de la requête :** traduction en expression algébrique.
4. **Réécriture de l'arbre :** notamment pour les vues ;
5. **Optimisation :**
   - 5.1 Analyse de l'arbre de requête.
   - 5.2 Les plans d'exécution amenant aux mêmes résultats sont évalués et le moins coûteux est choisi.
6. **Traitement :**
   - 6.1 Le plan d'exécution est mis en oeuvre.
   - 6.2 Accès aux données, évaluation des conditions.
   - 6.3 Le résultat est retourné.

Section 4

Les index

Les index

Soit la table :

```
1 create table test1 (
2     id integer,
3     contenu varchar
4 );
```

et la requête :

```
1 select contenu from test1 where id = constante;
```

Sans aucune préparation, le système doit lire la table `test1` dans son intégraliteé pour trouver les lignes recherchées.

Ce processus peut être beaucoup plus efficace avec un **index**.

Les index

Syntaxe (simplifiée) création :

```
CREATE INDEX nom
     ON nom_table ( nom_colonne [ ,... ] )
```

Syntaxe (simplifiée) suppression :

```
DROP INDEX nom
```

Exemple :

```
1 create index test1_id_idx on test1 (id);
```

Les index

Le système utilise l'index dans les requêtes lorsqu'il juge que ceci est plus efficace qu'une lecture complète de la table.

Les index peuvent être utilisés aussi dans les jointures.

Il faut néanmoins lancer la commande `ANALYZE` régulièrement pour permettre à l'optimiseur de requêtes de prendre les bonnes décisions. (Plus d'informations plus tard.)

Les index peuvent aussi bénéficier aux commandes `UPDATE` et `DELETE` à conditions de recherche.

Les index sont mis à jour automatiquement lorsque les tables sont modifiées. Les opérations de manipulation de données sont donc plus lourdes. C'est pourquoi les index qui sont peu, voire jamais, utilisés doivent être supprimés.

Les clés primaires ou uniques sont automatiquement indexées.

Les index

Quand faut-il indexer d'autres colonnes ?

- grandes tables ;
- requêtes concernant un petit nombre de lignes d'une grande table ;
- colonnes dont le nombre de valeurs différentes est très grand (valeurs presque uniques) ;
- colonnes fréquemment utilisées dans des jointures (clés étrangères) ;
- colonnes avec des valeurs `NULL` dont les requêtes ne s'intéressent principalement qu'aux valeurs non nulles.

Section 5

Algèbre relationnelle

Optimisation et algèbre relationnelle

Exemple :

```
select ue.*, d.libelle_diplome
    from diplome d natural join unite_enseignement ue
    where d.nb_etudiants > 100;
```

Admettons :

- 28 diplômes,
- 271 unités d'enseignement,
- 3 diplômes avec plus de 100 étudiants inscrits.
- chaque unité d'enseignement concerne un seul diplôme.
- cette requête retourne 70 enregistrements.

Jointure naturelle puis sélection :
28×271 lect. $+$ 271 écrit. $+$ 271 lect. $+$ 70 écrit. $=$ **8200 E/S**

Sélection puis jointure naturelle :
28 lect. $+$ 3 écrit. $+ 3 \times 271$ lect. $+$ 70 écrit. $=$ **914 E/S**

Optimisation et algèbre relationnelle

Pour l'optimisation : utilisation des règles de réécriture autorisées par les propriétés de l'algèbre relationnelle.

- favoriser les sélections à un niveau bas, car cet opérateur est considéré comme plus réducteur ;
- tandis que les produits sont remontés le plus tard possible (le plus haut dans l'arbre de la requête).

Est-ce que cela est toujours vrai ?

Optimisation et algèbre relationnelle

Quelles sont les oeuvres dont un exemplaire a été acheté il y a plus d'un mois et dont l'auteur est Pierre Bourdieu ?

```
1 select o.titre
2     from livre l, oeuvre o
3     where l.date_achat + cast('1 month' as interval)
4         < current_date
5     and o.id_oeuvre = l.id_oeuvre
6     and l.nom_auteur = 'Pierre Bourdieu';
```

Admettons :

- 1500 livres ;
- 300 oeuvres ;
- 98% des achats ont plus d'un mois ;
- 30% des oeuvres ont un exemplaire à la bibliothèque ;
- 3 titres de Pierre Bourdieu ;
- 2 exemplaires chacun, tous achetés il y a plus d'un mois.

Optimisation et algèbre relationnelle

Solution 1. 2 sélections puis une jointure :

- Sélection des livres : 1500 lectures + 1470 écritures.
- Sélection des oeuvre : 300 lectures puis 3 écritures.
- Jointure : 3×1470 lectures (soit 4410) puis 6 écritures

Total : $1500 + 1470 + 300 + 3 + 4410 + 6 =$ **7689 E/S**.

Solution 2. 1 sélection, 1 jointure puis 1 sélection :

- Sélection des oeuvres : 300 lectures puis 3 écritures.
- Jointure : 3×1500 lectures (soit 4500) puis 6 écritures.
- Sélection sur le résultat de la jointure : 6 lectures puis 6 écritures

Total : $300 + 3 + 4500 + 6 + 6 + 6 =$ **4821 E/S**.

Il s'agit d'un cas où une jointure est plus réductrice qu'une sélection.
(**Ce cas est rare !**)

Optimisation

Une bonne optimisation ne peut pas se contenter d'une réécriture algébrique.

Il faut tenir compte :

1. des chemins d'accès aux données ;
2. des différents algorithmes implantant une même opération algébrique ;
3. des informations statistiques sur la base de données.

Organisation physique des données

Organisation des données en mémoire secondaire :

- Le disque est divisé en blocs physiques (ou pages) de tailles égales.
- Accès à un bloc par son adresse (par exemple le numéro de cylindre + le numéro du premier secteur + le numéro de face).
- Le bloc est l'unité d'échange entre la mémoire secondaire et la mémoire principale.

Organisation des données dans un SGBD :

- Les données sont stockées dans des fichiers.
- Un fichier occupe un ou plusieurs blocs (pages) sur un disque.
- L'accès aux fichiers est géré par un logiciel spécifique : le Système de Gestion de Fichiers (SGF).
- Un fichier est caractérisé par son nom.

Le plan d'exécution

L'instruction `EXPLAIN` :

- affiche le plan d'exécution (*query plan*) choisi par PostgreSQL pour une requête ;
- affiche également les coûts **estimés** pour la requête.

Les coûts estimés :

- **Temps de préparation :** temps pour preparer les tables avant leur examen (pour les trier par exemple) ;
- **Temps total :** si tous les enregistrements sont récupérés (ce qui n'est pas le cas avec une clause `LIMIT`, par exemple) ;
- **Nombre d'enregistrements retournés.**
- **Taille moyenne des enregistrements.**

Exemples

```
jeanperrin=# \d ue_ee
                Table "public.ue_ee"
         Colonne           |  Type   | Modifications
---------------------------+---------+---------------
 id_element_enseignement | integer | not null
 id_unite_enseignement   | integer | not null
 maj_auto                | boolean | default false
 nb_gpe                  | real    | default 0
Index: ue_ee_pkey primary key btree
       (id_unite_enseignement, id_element_enseignement)
Foreign Key constraints: ue_ee_del_cascade_ue
   FOREIGN KEY (id_unite_enseignement)
   REFERENCES unite_enseignement(id_unite_enseignement)
       ue_ee_del_cascade_ee
   FOREIGN KEY (id_element_enseignement)
   REFERENCES element_enseignement(id_element_enseignement)
```

Exemples

La clé primaire de cette table est une clé composée `(id_element_enseignement, id_unite_enseignement)`.

```
jeanperrin=# explain select * from ue_ee;
                          QUERY PLAN
---------------------------------------------------------------
 Seq Scan on ue_ee  (cost=0.00..9.71 rows=571 width=13)
(1 row)
```

- Coût de préparation estimé : 0.00 accès pages-disques.
- Coût total estimé : 9.71 accès pages-disques.
- Nombre d'enregistrements estimé : 571
- Taille moyenne d'un enregistrement (en octets) : 13.
- Le scan (examen) de la table est effectué séquentiellement.

Exemples

Un second exemple, avec une condition de sélection :

```
jeanperrin=# explain select * from ue_ee where nb_gpe>2;
                          QUERY PLAN
-----------------------------------------------------------
 Seq Scan on ue_ee  (cost=0.00..11.14 rows=190 width=13)
   Filter: (nb_gpe > 2::double precision)
(2 rows)
```

- Le coût total estimé augmente : les 571 enregistrements devront être parcourus, avec en plus le temps nécessaire au test de la condition `WHERE`.
- Le filtre limite le nombre de lignes retournées (190), mais le parcours est toujours séquentiel.

Exemples

```
jeanperrin=# explain select * from ue_ee
where id_unite_enseignement < 18 ;
                          QUERY PLAN
-----------------------------------------------------------
 Seq Scan on ue_ee  (cost=0.00..11.14 rows=190 width=13)
   Filter: (id_unite_enseignement < 18)
```

Même résultat estimé, car utilisation d'un filtre !

Exemples

Utilisation d'un des attributs de la clé primaire dans la condition :

```
jeanperrin=# explain select * from ue_ee
where id_unite_enseignement = 18;
                                  QUERY PLAN
-------------------------------------------------------------------------
 Index Scan using ue_ee_pkey on ue_ee  (cost=0.00..10.45
                                        rows=3 width=13)
   Index Cond: (id_unite_enseignement = 18)
(2 rows)
```

- Le nombre d'enregistrements visités, grâce à l'utilisation de l'index, chute à 3.
- Le scan de la table est effectué en utilisant un index sur la clé primaire. Cela améliore le temps d'exécution par rapport aux deux requêtes précédentes.

Exemples

Utilisation complète de la clé primaire :

```
jeanperrin=# explain select * from ue_ee
where id_unite_enseignement = 12
and id_element_enseignement = 80;
                        QUERY PLAN
-----------------------------------------------------
 Index Scan using ue_ee_pkey on ue_ee
       (cost=0.00..5.57 rows=1 width=13)
   Index Cond: ((id_unite_enseignement = 12)
           AND (id_element_enseignement = 80))
(2 rows)
```

- L'utilisation de l'index permet d'obtenir un résultat bien meilleur qu'un examen complet de la table, même sans condition (premier exemple).

Exemples

Enfin, un exemple qui combine scan indexé et filtrage des résultats :

```
jeanperrin=# explain select * from ue_ee
where id_unite_enseignement = 18 and nb_gpe > 3;
                                  QUERY PLAN
-----------------------------------------------------------
 Index Scan using ue_ee_pkey on ue_ee
       (cost=0.00..10.46 rows=1 width=13)
   Index Cond: (id_unite_enseignement = 18)
   Filter: (nb_gpe > 3::double precision)
(3 rows)
```

Avec plusieurs tables

Il y a trois stratégies possibles pour faire une jointure :

- **Nested Loop Join :** la table de droite est examinée une fois pour chaque enregistrement de la table de gauche ;
- **Merge Sort Join :** chaque relation est triée sur les attributs de jointure avant que la jointure ne soit effectuée ;
- **Hash Join :** la relation de droite est d'abord examinée et enregistrée dans une table de hachage, en utilisant les attributs de jointure comme des clés pour la table de hachage.

Avec plusieurs tables

- Le plan d'exécution final comporte :
  - **des noeuds de scan** indexés ou séquentiels ;
  - **des noeuds de jointure** ;
  - plus d'autres noeuds intermédiaires comme **des noeuds de tri**, **de fonctions d'agrégations** ou **de calculs**.
- La plupart de ces noeuds procèdent en plus à des opérations de sélection (**filtrage**) ou de projection.
- Une des prérogatives de l'optimiseur est d'attacher les conditions de sélection (clause `WHERE`) aux noeuds appropriés dans le plan d'exécution.

Avec plusieurs tables

```
jeanperrin=# select count(*) from enseignant;
 count
-------
   140
(1 ligne)

jeanperrin=# select count(*)
from discipline_unite_enseignement;
 count
-------
   284
(1 ligne)
```

Avec plusieurs tables

```
jeanperrin=# explain select nom, id_unite_enseignement
jeanperrin-# from enseignant
jeanperrin-# natural join discipline_unite_enseignement ;
    QUERY PLAN
-----------------------------------------------------------------
 Merge Join  (cost=9.39..85.02 rows=4942 width=23)
   Merge Cond: ("outer".id_discipline="inner".id_discipline)
   ->  Index Scan using discipline_unite_enseignem_pkey
       on discipline_unite_enseignement
       (cost=0.00..12.45 rows=284 width=8)
   ->  Sort  (cost=9.39..9.74 rows=140 width=15)
         Sort Key: enseignant.id_discipline
         ->  Seq Scan on enseignant
               (cost=0.00..4.40 rows=140 width=15)
(6 lignes)
```

Avec plusieurs tables

Exemples d'une jointure sur une clé primaire :

```
base_test=# explain select *
base_test-# from enseignant natural join statut;
                              QUERY PLAN
-----------------------------------------------------------
 Hash Join  (cost=1.09..5.28 rows=7 width=522)
   Hash Cond: ("outer".id_statut = "inner".id_statut)
   ->  Seq Scan on enseignant
         (cost=0.00..3.40 rows=140 width=452)
   ->  Hash  (cost=1.07..1.07 rows=7 width=70)
         ->  Seq Scan on statut
               (cost=0.00..1.07 rows=7 width=70)
(5 lignes)
```

Avec plusieurs tables

Pour comparer avec les autres possibilités de jointure, nous pouvons modifier les valeurs d'environnement :

- `set enable_hashjoin = on/off;`
- `set enable_mergejoin = on/off;`
- `set enable_nestloop = on/off;`

Avec plusieurs tables

```
base_test=# set enable_hashjoin = off;
SET
base_test=# explain select *
base_test-# from enseignant natural join statut;
          QUERY PLAN
-----------------------------------------------------
 Merge Join  (cost=9.56..10.38 rows=7 width=522)
   Merge Cond: ("outer".id_statut = "inner".id_statut)
   ->  Sort  (cost=8.39..8.74 rows=140 width=452)
         Sort Key: enseignant.id_statut
         ->  Seq Scan on enseignant
                 (cost=0.00..3.40 rows=140 width=452)
   ->  Sort  (cost=1.17..1.19 rows=7 width=70)
         Sort Key: statut.id_statut
         ->  Seq Scan on statut
                 (cost=0.00..1.07 rows=7 width=70)
(8 lignes)
```

Avec plusieurs tables

```
base_test=# set enable_mergejoin = off;
SET
base_test=# explain select *
base_test-# from enseignant natural join statut;
                                QUERY PLAN
-----------------------------------------------------------------
 Nested Loop  (cost=0.00..37.12 rows=7 width=522)
   Join Filter: ("inner".id_statut = "outer".id_statut)
   ->  Seq Scan on statut
         (cost=0.00..1.07 rows=7 width=70)
   ->  Seq Scan on enseignant
         (cost=0.00..3.40 rows=140 width=452)
(4 lignes)
```

Avec plusieurs tables

Encore un exemple avec une fonction d'agrégation :

```
base_test=# explain select sum(nb_heures_faites)
base_test-# from enseignant_element_enseignement
base_test-# where id_enseignant <50;
                    QUERY PLAN
-----------------------------------------------------
 Aggregate  (cost=1.36..1.36 rows=1 width=4)
   ->  Seq Scan on enseignant_element_enseignement
           (cost=0.00..1.34 rows=9 width=4)
         Filter: (id_enseignant < 50)
(3 lignes)
```

Et c'est fini ?

Lorsqu'il s'agit de fonctions (d'agrégation ou pas) définies par l'utilisateur, la commande `EXPLAIN` ne détaille pas le plan d'exécution pour les requêtes SQL internes aux fonctions.

D'après les exemples précédents, on voit que le SGBD choisit le meilleur plan d'exécution. Est-ce que cela dédouane le programmeur de tout effort à effectuer ?

Exemple

```
base_test=# explain select id_enseignant
base_test-# from enseignant_element_enseignement
base_test-# where id_element_enseignement in
base_test-# (select id_element_enseignement
base_test-# from element_enseignement where
base_test-# id_nature_enseignement = 3);
                    QUERY PLAN
-----------------------------------------------------------
Seq Scan on enseignant_element_enseignement
          (cost=0.00..337.15 rows=14 width=4)
Filter: (subplan) SubPlan
   -> Materialize (cost=12.44..12.44 rows=3 width=4)
   -> Seq Scan on element_enseignement
           (cost=0.00..12.44 rows=3 width=4)
      Filter: (id_nature_enseignement = 3)
          (6 lignes)
```

Exemple

```
base_test=# explain select id_enseignant
from enseignant_element_enseignement
     natural join element_enseignement
where  id_nature_enseignement = 3;
      QUERY PLAN
-----------------------------------------------------
 Hash Join  (cost=12.44..13.85 rows=1 width=12)
   Hash Cond: ("outer".id_element_enseignement
               = "inner".id_element_enseignement)
   ->  Seq Scan on enseignant_element_enseignement
       (cost=0.00..1.27 rows=27 width=8)
   ->  Hash  (cost=12.44..12.44 rows=3 width=4)
         ->  Seq Scan on element_enseignement
             (cost=0.00..12.44 rows=3 width=4)
               Filter: (id_nature_enseignement = 3)
         (6 lignes)
```

EXPLAIN ANALYZE.

L'instruction `EXPLAIN ANALYZE` permet de comparer les estimations de coûts faits par `EXPLAIN` et les ressources réellement nécessaires pour l'exécution de la requête.

Les éléments chiffrés ne peuvent pas se comparer :

- car les unités ne correspondent pas :
  - l'unité de mesure est la milliseconde pour `EXPLAIN ANALYZE` ;
  - l'unité de mesure est nombre d'accès aux pages-disques converti en temps CPU pour `EXPLAIN` ;
- car certaines actions sont prises en compte d'un côté et pas de l'autre (dans le cas d'un `INSERT` par exemple, le temps mis pour procéder à l'insertion n'est pas pris en compte dans les coûts estimés).

Néanmoins, on peut comparer les différents ratios proposés par différents plans d'exécution.

Exemple

```
base_test=# explain analyze select id_enseignant
base_test-# from enseignant_element_enseignement
base_test-# where id_element_enseignement in (
base_test-#       select id_element_enseignement
base_test-#       from element_enseignement
base_test-#       where id_nature_enseignement = 3);
                        QUERY PLAN
-----------------------------------------------------------
 Seq Scan on enseignant_element_enseignement
(cost=0.00..337.15 rows=14 width=4)
(actual time=2.49..6.37 rows=11 loops=1)
   Filter: (subplan)
   SubPlan
     ->  Materialize
         (cost=12.44..12.44 rows=3 width=4)
         (actual time=0.05..0.11 rows=106 loops=27)
           ->  Seq Scan on element_enseignement
               (cost=0.00..12.44 rows=3 width=4)
               (actual time=0.10..1.19 rows=124 loops=1)
                 Filter: (id_nature_enseignement = 3)
 Total runtime: 6.48 msec
(7 lignes)
```

Exemple

```
base_test=# explain analyze select id_enseignant
base_test-# from enseignant_element_enseignement
base_test-# natural join element_enseignement
base_test-# where  id_nature_enseignement = 3;
                        QUERY PLAN
-----------------------------------------------------------------
 Hash Join  (cost=12.44..13.85 rows=1 width=12)
            (actual time=1.31..1.42 rows=11 loops=1)
   Hash Cond: ("outer".id_element_enseignement
               = "inner".id_element_enseignement)
   ->  Seq Scan on enseignant_element_enseignement
       (cost=0.00..1.27 rows=27 width=8)
       (actual time=0.01..0.07 rows=27 loops=1)
   ->  Hash  (cost=12.44..12.44 rows=3 width=4)
       (actual time=1.25..1.25 rows=0 loops=1)
         ->  Seq Scan on element_enseignement
             (cost=0.00..12.44 rows=3 width=4)
             (actual time=0.06..1.05 rows=124 loops=1)
               Filter: (id_nature_enseignement = 3)
 Total runtime: 1.52 msec
(7 lignes)
```

Attention

Il est important de ne pas interpréter les résultats de la commande `EXPLAIN` comme étant des résultats valables pour toute table.

- Les estimations des coûts et donc les choix de plans d'exécution sont **dépendants des tables**.
- Ainsi, une petite table n'excédant pas une page-disque, même si elle est indexée, ne sera jamais scannée autrement que séquentiellement.

Les statistiques utilisées par le SGBD

Les informations sont de deux ordres :

- **Informations factuelles :** le nombre d'enregistrements, le nombre de pages-disques, etc. de chaque table et index ;
- **Informations statistiques :** les valeurs les plus courantes et leur coefficient de distribution, la taille moyenne, etc. sur chaque attribut de chaque table.

Les statistiques utilisées

Les premières informations se trouvent dans la table `pg_class` :

```
base_test=# select relname, relkind, reltuples, relpages
from pg_class
where relname like 'elt_enseign%';
            relname      | relkind | reltuples | relpages
-------------------------+---------+-----------+----------
elt_enseign_pkey         | i       |       515 |        4
elt_enseign_code_ee_key  | i       |       515 |        4
elt_enseign              | r       |       515 |        6
(3 lignes)
```

Les statistiques utilisées

Ces informations **ne sont pas tenues à jour à chaque modification** d'une table (ce n'est pas nécessaire et plus efficace).

Elles sont mises à jour par les instructions :
`VACUUM`, `ANALYZE` et `CREATE INDEX`.

L'instruction `ANALYZE` sert à collecter les statistiques sur la base. Elle peut être appelée :

- sans paramètre pour analyser toute la base ;
- avec un nom de table ;
- avec un nom de table et des attributs pour analyser que ces éléments.

L'instruction `VACUUM` déclenche le garbage-collector. Il est recommande de l'utiliser accompagné de la commande `ANALYZE`.

Les statistiques utilisées

La seconde catégorie d'information se trouve dans la table `pg_statistics`, et aussi dans la vue associée `pg_stats`.

```
base_test=# select most_common_vals,
base_test-# most_common_freqs,histogram_bounds
base_test-# from pg_stats where tablename='ue_ee';
 most_common_vals | most_common_freqs | histogram_bounds
------------------+-------------------+------------------
(0 lignes)

base_test=# vacuum analyze;
VACUUM
base_test=# select attname, most_common_vals,
base_test-# most_common_freqs
base_test-# from pg_stats where tablename='ue_ee'
base_test-# and not (attname like 'id_%');
   attname |most_common_vals|   most_common_freqs
-----------+----------------+---------------------------
maj_auto   | {t,f}          | {0.740806,0.259194}
nb_gpe     | {1,0,2}        | {0.553415,0.189142,0.134851}
(4 lignes)
```

Les statistiques utilisées

```
base_test=# \d pg_stats
          View "pg_catalog.pg_stats"
      Colonne       |  Type    | Modifications
--------------------+----------+---------------
 schemaname         | name     |
 tablename          | name     |
 attname            | name     |
 null_frac          | real     |
 avg_width          | integer  |
 n_distinct         | real     |
 most_common_vals   | text[]   |
 most_common_freqs  | real[]   |
 histogram_bounds   | text[]   |
 correlation        | real     |
```

Les statistiques utilisées

La vue `pg_stats` :

- `null_frac` : le pourcentage d'enregistrements dont la valeur est nulle ;
- `n_distinct` : des informations sur le nombre de valeurs distinctes ;
- `correlation` : une information sur le tri de la table sur cet attribut (utile pour les index).

Optimiser les requêtes

Optimiser les requêtes, c'est. . .tout un art !

Les points importants :

- penser aux optimisations en algèbre relationnelle lors de l'écriture des requêtes ;
- créer des index sur les colonnes (hors clés primaires) fréquemment utilisées comme critère de recherche ;
- maîtriser les calculs de plans d'exécution par l'utilisation de `jointures` pour imposer un ordre ;
- penser à régulièrement mettre à jour les informations statistiques sur la base.

Section 6

Utilisation des statistiques

Exemples d'estimations de lignes

```
EXPLAIN SELECT * FROM tenk1;

                         QUERY PLAN
-------------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..458.00 rows=10000 width=244)
```

Le nombre de pages et de lignes est trouvé dans `pg_class` :

```
SELECT relpages, reltuples FROM pg_class WHERE relname = 'tenk1';

 relpages | reltuples
----------+-----------
      358 | 10000
```

Le planificateur récupère ensuite le nombre de pages actuel dans la table (c'est une opération peu coûteuse, ne nécessitant pas un parcours de table).

Si c'est différent de `relpages`, alors `reltuples` est modifié en accord pour arriver à une estimation actuelle du nombre de lignes.

Dans ce cas, la valeur de `relpages` est mise à jour.

Exemples d'estimations de lignes

Avec `WHERE` :

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 1000;

                                 QUERY PLAN
-------------------------------------------------------------------------
 Bitmap Heap Scan on tenk1  (cost=24.06..394.64 rows=1007
                             width=244)
   Recheck Cond: (unique1 < 1000)
   ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..23.80
                                            rows=1007 width=0)
         Index Cond: (unique1 < 1000)
```

Le planificateur examine la condition `WHERE` et cherche la fonction de sélectivité à partir de l'opérateur < dans `pg_operator`.

Exemples d'estimations de lignes

La fonction `scalarlt-sel` récupère l'histogramme pour `unique1` à partir de `pg_statistics` (vue `pg_stats`).

```
SELECT histogram_bounds FROM pg_stats
WHERE tablename='tenk1' AND attname='unique1';

                    histogram_bounds
------------------------------------------------------
 {0,993,1997,3050,4040,5036,5957,7057,8029,9016,9995}
```

La fraction de l'histogramme occupée par verb< 1000 est traitée. C'est la sélectivité.

$$\begin{aligned} \textit{selectivite} &= (1 + (1000 - 993)/(1997 - 993))/10 \\ &= 0.100697 \end{aligned}$$

La valeur 1000 est dans la seconde partie (993-1997), donc en supposant une distribution linéaire des valeurs à l'intérieur de chaque partie, nous pouvons calculer la sélectivité comme :

$$\begin{aligned} \textit{lignes} &= \textit{reltuples} * \textit{selectivite} \\ &= 1007 \end{aligned}$$

Exemples d'estimations de lignes

Égalité dans le `WHERE:`

```
EXPLAIN SELECT * FROM tenk1 WHERE stringu1 = 'CRAAAA';

                        QUERY PLAN
-----------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..483.00 rows=30 width=244)
   Filter: (stringu1 = 'CRAAAA'::name)
```

Pour une estimation d'égalité, l'histogramme n'est pas utile.

Exemples d'estimations de lignes

La liste des valeurs les plus communes (most common values) est utilisé pour déterminer la sélectivité.

```
SELECT null_frac, n_distinct, most_common_vals,
most_common_freqs FROM pg_stats
WHERE tablename='tenk1' AND attname='stringu1';

null_frac          | 0
n_distinct         | 676
most_common_vals   |
{EJAAAA,BBAAAA,CRAAAA,FCAAAA,FEAAAA,GSAAAA,
JOAAAA,MCAAAA,NAAAAA,WGAAAA}
most_common_freqs  | {0.00333333,0.003,0.003,0.003,0.003,0.003,
0.003,0.003,0.003,0.003}
```

Comme `CRAAAA` apparaît dans la liste, la sélectivité est l'entrée correspondante dans la liste des fréquences 0.03.

$$\begin{aligned} \textit{lignes} &= \textit{reltuples} * \textit{freq} \\ &= 30 \end{aligned}$$

Exemples d'estimations de lignes

Avec une constante qui n'est pas dans MCV :

```
EXPLAIN SELECT * FROM tenk1 WHERE stringu1 = 'xxx';

                        QUERY PLAN
-----------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..483.00 rows=15 width=244)
   Filter: (stringu1 = 'xxx'::name)
```

L'approche est d'utiliser le fait que la valeur n'est pas dans la liste, combinée avec la connaissance des fréquences pour tout les MCV

Exemples d'estimations de lignes

Ajouter toutes les fréquences pour les MCV et les soustraire d'un, puis les diviser par le nombre des autres valeurs distinctes :

$$\begin{aligned} \mathit{selectivite} &= (1 - \mathit{sum}(\mathit{mcv}))/(\mathit{num\_distinct} - \mathit{num\_mcv}) \\ &= 0.0014559 \end{aligned}$$

Le nombre de lignes est calculé comme d'habitude :

$$\begin{aligned} \mathit{lignes} &= \mathit{reltuples} * \mathit{selectivite} \\ &= 15 \end{aligned}$$

Exemples d'estimations de lignes

Avec plus d'une condition dans le `WHERE` :

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 1000
AND stringu1 = 'xxx';
                                  QUERY PLAN
-----------------------------------------------------------------------
 Bitmap Heap Scan on tenk1  (cost=23.80..396.91 rows=1
                             width=244)
   Recheck Cond: (unique1 < 1000)
Filter: (stringu1 = 'xxx'::name)
->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..23.80
                                         rows=1007 width=0)
      Index Cond: (unique1 < 1000)
```

Le planificateur suppose que les deux conditions sont indépendantes :

$$\begin{aligned} \mathit{selectivite} &= \mathit{selectivite}(< 1000) * \mathit{selectivite}(\mathit{string}1 =' \mathit{xxx}') \\ &= 0.100697 * 0.0014559 \\ &= 0.0001466 \\ \mathit{lignes} &= 10000 * 0.0001466 \\ &= 1 \end{aligned}$$

Exemples d'estimations de lignes

Jointure :

```
EXPLAIN SELECT * FROM tenk1 t1, tenk2 t2
WHERE t1.unique1 < 50 AND t1.unique2 = t2.unique2;
                                  QUERY PLAN
-------------------------------------------------------------------------
 Nested Loop  (cost=4.64..456.23 rows=50 width=488)
   ->  Bitmap Heap Scan on tenk1 t1  (cost=4.64..142.17
                                      rows=50 width=244)
         Recheck Cond: (unique1 < 50)
         ->  Bitmap Index Scan on tenk1_unique1
                              (cost=0.00..4.63 rows=50 width=0)
               Index Cond: (unique1 < 50)
   ->  Index Scan using tenk2_unique2 on tenk2 t2
                              (cost=0.00..6.27 rows=1 width=244)
Index Cond: (unique2 = t1.unique2)
```

Il s'agit d'un opérateur $=$. Nous calculons les cardinalités (nombre de lignes) et ensuite la séléctivité :

$$\begin{aligned} lignes &= (outer\_cardinality * inner\_cardinality) * selectivite \\ &= (50 * 10000) * 0.0001 = 50 \end{aligned}$$

Cours 4

Administration

Sommaire

7. Un peu sur les catalogues

8. Un peu sur les droits et schémas

9. Tâches de maintenance

Section 7

Un peu sur les catalogues

Quelques tables importantes

- `pg_attribute` : les colonnes (attributs) des tables.
- `pg_class` : les relations (tables, indexes, views, sequénces, etc.).
- `pg_constraint` : le restrictions (clés primaires, uniques, étrangères, checks).
- `pg_stats` : statistiques.
- `pg_tables` : les tables.
- `pg_triggers` : les déclencheurs.

pg_attribute

Contient des informations sur les colonnes (attributs) des tables.

Quelques attributs :

Attribut	Type	Description
attrelid	oid	Id de la table qui le contient.
attname	name	Nom de l'attribut.
atttypeid	oid	Id du type.
attnum	int2	Numéro de la colonne (négatif si crée automat.).
attnotnull	bool	Si restriction non-nulle.
atthasdef	bool	Si valeur par défaut.

pg_class

Contient des informations sur les tables, indexes, séquences, views, etc.

Quelques attributs :

Attribut	Type	Description
oid	oid	Id de la table (caché).
relname	name	Nom de la table
relowner	oid	Id du propriétaire.
relpages	int4	Estimation de la taille.
reltuples	float4	Estimation du nombre de lignes.
relkind	char	r = relation, i = index, s = sequence, v = view, etc.

pg_constraint

Contient des informations sur restrictions.

Quelques attributs :

Attribut	Type	Description
oid	oid	Id de la restriction (caché).
conname	name	Nom de la restriction.
conname	char	c = check, f = foreign, p = primary, u = unique, etc.
conrelid	oid	Id de la table.
confrelid	oid	Id de la table de référence (si clé étrangère).
conkey	int2[]	Liste des colonnes restreintes.
confkey	int2[]	Liste des colonnes référencées (si clé étrangère).
consrc	text	Expression de la restriction (si check).

pg_stats

La view pg_stats contient des informations statistiques sur les tables (estimatives).

Quelques attributs :

Attribut	Type	Description
tablename	name	Nom de la table.
attname	name	Nom de la colonne.
avg_width	integer	Taille moyenne en bytes.
n_distinct	real	Si positif, nombre de valeur distincts. Si négatif, nombre de linges sur nombre de valeurs distincts.
most_common_vals	anyarray	Valeurs les plus trouvés.
most_common_freqs	real[]	Fréquences des valeurs les plus trouvés
histogram_bounds	anyarray	Liste de valeurs qui divisent les valeurs la colonne en groupes de valeurs de même population.

pg_tables

La view `pg_tables` contient des informations factuelles sur les tables (estimatives).

Quelques attributs :

Attribut	**Type**	**Description**
tablename	name	Nom de la table.
tableowner	name	Nom du propriétaire.
hasindexes	boolean	Si contient des indexes.
hastriggers	boolean	Si contient des déclencheurs.

pg_triggers

La view pg_triggers contient des informations sur les déclencheurs.

Quelques attributs :

Attribut	Type	Description
tgrelid	oid	Id de la table du déclencheur.
tgname	name	Nom du déclencheur.
tgfoid	oid	Id de la fonction du déclencheur.

Section 8

Un peu sur les droits et schémas

Les rôles

Les utilisateurs et les groupes sont des éléments d'un SGBD qui sont partagé par toutes les bases.

Ils sont regroupés sous une même notion de rôle.

```
CREATE ROLE nom [ OPTION ]
```

où option est un ou plusieurs parmi :

- `SUPERUSER`
- `CREATEDB`
- `CREATEROLE`
- `LOGIN`
- `VALID UNITL date`
- etc.

Les rôles

Les utilisateurs sont de rôles crées avec l'option `LOGIN`.

Il est possible d'associer un rôle à un utilisateur :

```
GRANT role TO role1 [ , role2, ... ]
```

et supprimer un rôle :

```
REVOKE role FROM role1 [, role2, ... ]
```

Pour consulter les utilisateurs :

```
select username from pg_roles;
```

ou bien :

```
\du
```

Pour consulter les rôles :

```
select rolename from pg_roles;
```

ou bien :

```
\dg
```

Les rôles, les fonctions et les déclencheurs

Les fonctions et les déclencheurs autorisent les utilisateurs à insérer du code que d'autres utilisateurs peuvent exécuter sans en avoir l'intention.

Ces mécanismes permettent la création d'un « cheval de Troie ».

Les fonctions sont exécutées à l'intérieur du processus serveur avec les droits au niveau système d'exploitation du démon serveur de la base de données.

La seule protection réelle est d'effectuer un fort contrôle sur ceux qui peuvent définir des fonctions.

Droits

Syntaxe (simplifiée) :

```
GRANT droit ON table TO role
```

ou les droits peuvent être :

- SELECT
- INSERT
- UPDATE
- DELETE
- TRUNCATE
- REFERENCES
- TRIGGER
- ALL

Pour consulter ses droits :

\z

Schémas

Un schéma est une structure logique qui ressemble des objets (tables, vues, index, fonctions, etc.).

C'est un espace de nommage à l'intérieur d'une base (PostgreSQL ne propose pas de paquetage). La table `maTable` du schéma `toto` sera appelée `toto.maTable`.

C'est un moyen de regrouper dans une même entité logique des objets qui sont fortement liées entre eux.

C'est aussi un moyen simple de permettre à une équipe de travailler sur une même base.

Par défaut, l'utilisateur travaille dans le schéma `public`.

Schémas

L'utilisateur peut créer des schémas :

```
create schema mon_schema;
```

et des objets dans le schema :

```
create table mon_schema.table;
```

Supprimer un schéma :

```
drop schema mon_schema cascade;
```

`CASCADE` permet de supprimer tous les objets du schéma.

Association d'un schéma à un utilisateur (qui en dévient le propriétaire) :

```
create schema mon_schema authorization utilisateur;
```

Section 9

Tâches de maintenance

Tâches de maintenance

Tout SGBD requiert que certains tâches soient réalisées régulièrement une performance optimale.

Ces tâches répétitives et peuvent être facilement automatisées avec des outils standard comme les scripts cron.

Les opérations de maintenance :

- sauvegarde ;
- nettoyage (`vacuum`) ;
- mise à jour des statistiques ;
- ré-indexation ;
- gestion du fichier de traces.

Nettoyage

Le SGBD nécessite des opérations de nettoyages périodiques (`VACUUM`).

Pour de nombreuses installations, il est suffisant de laisser travailler le démon `autovacuum`.

En fonction des cas, les paramètres de cet outil peuvent être optimisés pour obtenir de meilleurs résultats

VACUUM

L'instruction `VACUUM` est responsable pour :

1. récupération ou ré-utilisation de l'espace disque occupé par les lignes supprimées ou mises à jour ;
2. mettre à jour les statistiques utilisées par l'optimiseur ;
3. mettre à jour la carte de visibilité qui accélère les parcours d'index seuls ;

Il y a deux variantes :

- **`VACUUM` standard :**
  Permet l'exécution en parallèle des instructions `SELECT`, `INSERT`, `UPDATE` et `DELETE`.
  Mais la définition d'une table ne peut être modifiée (`ALTER TABLE`).
- **`VACUUM FULL:`**
  Peut récupérer davantage d'espace disque.
  Mais s'exécute beaucoup plus lentement.
  Nécessite un verrou exclusif sur la table.

VACUUM produit un nombre important d'entrées/sorties. Cela peut entraîner de mauvaises performances pour les autres sessions actives.

Récupérer l'espace disque

Les lignes ne sont pas immédiatement supprimées par `UPDATE` et `DELETE` pour garantir la consistance des accès concurrents.

La forme standard de `VACUUM` élimine les versions d'enregistrements morts dans les tables et les index, et marque l'espace comme réutilisable.

`VACUUM FULL` écrit une nouvelle version complète du fichier de la table. Cela réduit la taille de la table mais peut prendre beaucoup de temps et requiert aussi un espace disque supplémentaire.

Le but d'un netoyage régulier est de lancer des `VACUUM` standard suffisamment souvent pour éviter d'avoir recours à `VACUUM FULL`.

Le démon `autovacuum` essaie de fonctionner de cette façon, et n'exécute jamais de `VACUUM FULL`.

`VACUUM FULL` peut retourner une table à sa taille minimale, mais cela ne sert pas à grand chose, si cette table recommence à grossir dans un futur proche.

Récupérer l'espace disque

Nous pouvons planifier le `VACUUM`, par exemple la nuit, quand la charge est faible.

Le difficulté est que la table peut avoir un pic d'activité inattendu pendant la journée et un `VACUUM FULL` devient nécessaire.

Le démon `autovacum` planifie les vacuum de façon dynamique, en réponse à l'activité de mise à jour.

Il est donc peu raisonnable de le désactiver totalement, sauf si l'activité de la base est extrêmement prévisible.

Un compromis possible est de régler les paramètres du démon afin qu'il ne réagisse qu'à une activité exceptionnellement lourde de mise à jour. Cela peut éviter de perdre le contrôle, tout en laissant les `VACUUM` planifiés faire le gros du travail quand la charge est normale.

Récupérer l'espace disque

Si le contenu d'une table est supprimé périodiquement, il est préférable d'envisager l'utilisation de `TRUNCATE`

Cette instruction supprime le contenu entier de la table immédiatement sans nécessiter de `VACUUM FULL` pour réclamer l'espace.

Mise à jour des statistiques

L'optimiseur de requêtes s'appuie sur des informations statistiques.

Ces statistiques sont collectées par `ANALYZE` ou `VACUUM ANALYSE`.

Le démon `autovacuum` exécute automatiquement `ANALYZE` quand le contenu d'une table aura changé suffisamment.

À l'instar du nettoyage, les statistiques doivent être plus souvent collectées pour les tables intensément modifiées que pour celles qui le sont moins.

Toutefois, un administrateur peut préférer de planifier manuellement ses opérations, surtout si l'activité de la table n'a pas d'impact sur les statistiques des colonnes « importantes ».

Mise à jour des statistiques

Une règle simple pour décider quelles statistiques collecter est de voir comment évoluent les valeurs minimum et maximum des données.

Par exemple :

- une colonne *A* qui contient la date de mise à jour de la ligne ;
- une autre colonne *B* contient les URL des pages accédées sur un site web.

Sur laquelle de ces deux colonnes doit-on exécuter `ANALYZE` le plus fréquemment ?

Réponse : sur *A* parce que la distribution probabiliste des données changera plus rapidement. Notez que la valeur maximum changera le plus souvent.

Ré-indexation

Dans certains cas, reconstruire périodiquement les index par la commande `REINDEX` ou une série d'étapes individuelles de reconstruction en vaut la peine.

Si, sur une page, quelques clés d'index ont été supprimés, la page reste allouée. En conséquence, l'espace disque peut être très mal utilisé. Dans de tels cas, la réindexation périodique est recommandée.

De plus, pour les index B-tree, un index récemment construit est légèrement plus rapide : les pages logiquement adjacentes sont habituellement aussi physiquement adjacentes dans un index nouvellement créé (ceci ne s'applique pas aux index non B-tree).

Fichier de trace

Sauvegarder les journaux de trace du SGBD est une bonne idée.

Néanmoins, les journaux ont tendance à être volumineux et vous ne voulez pas les sauvegarder indéfiniment. Vous avez donc besoin d'en faire une « rotation ».

Si vous redirigez simplement `stderr` de postgres dans un fichier, vous aurez un journal des traces mais la seule façon de le tronquer sera d'arrêter et de relancer le serveur.

Il existe un programme interne de rotation que vous pouvez utiliser en configurant le paramètre `logging_collector` à `true` dans `postgresql.conf`.

Vous pouvez aussi utiliser un programme externe de rotation. Par exemple, l'outil `rotatelogs` inclus dans la distribution Apache ou bien `syslog`. Pour cela, envoyez via un tube la sortie `stderr` du serveur dans le programme désiré.

Sauvegardes

Il y a trois approches fondamentalement différentes pour sauvegarder les données de PostgreSQL :

- la sauvegarde SQL ;
- la sauvegarde au niveau du système de fichiers ;
- l'archivage continu.

Sauvegarde SQL

Création d'un fichier contenant des instructions SQL (« fichier dump »), qui recréent une base de données identique à celle sauvegardée :

```
$ pg_dump base_de_donnees > fichier_dump
```

La sauvegarde peut être effectuée depuis n'importe quel ordinateur ayant accès à la base. L'utilisateur doit avoir accès en lecture à toutes les tables à sauvegarder. Normalement, il faut être super-utilisateur de la base.

Avantages :

- la sortie de `pg_dump` peut être rechargée dans des versions plus récentes de PostgreSQL ;
- c'est la seule méthode qui fonctionnera lors du transfert d'une base de données vers une machine d'architecture différente (comme par exemple d'un serveur 32 bits à un serveur 64 bits).

Restauration de la sauvegarde SQL

La base doit être créée à partir de `template0` avant d'être restaurée :

```
$ createdb -T template0 base_de_donnees
```

Tous les utilisateurs possédant des objets ou ayant certains droits sur les objets de la base sauvegardée doivent exister préalablement à la restauration de la sauvegarde.

Pour restaurer une sauvegarde SQL :

```
$ psql base_de_donnees < fichier_dump
```

Il est possible de transmettre la base données directement à un autre :

```
$ pg_dump -h serveur1 base | psql -h serveur2 base
```

Sauvegarde au niveau du système de fichiers

Cette stratégie consiste à copier les fichiers utilisés par PostgreSQL :

```
$ tar -cf sauvegarde.tar /usr/local/pgsql/data
```

Avantage :

1. Généralement plus rapide qu'une sauvegarde SQL.

Inconvénients :

1. Généralement plus volumineuse qu'une sauvegarde SQL. (le fichier dump ne contient pas le contenu des index, mais l'instruction pour les recréer).
2. Le serveur doit être arrêté pour obtenir une sauvegarde utilisable et le même pour la restauration.
3. Cela fonctionne uniquement pour les sauvegardes et restaurations complètes d'un cluster de bases de données.

Archivage continu

PostgreSQL maintient des journaux WAL (*writea head log*) dans le répertoire `pg_xlog`.

Si le système s'arrête brutalement, la base de données peut être restaurée dans un état cohérent en utilisant les entrées des journaux depuis le dernier point de vérification.

Il est donc possible de combiner une sauvegarde de système de fichiers avec la sauvegarde des fichiers WAL.

Si la récupération est nécessaire, la sauvegarde des fichiers est restaurée, puis les fichiers WAL sauvegardés sont rejoués pour amener la sauvegarde jusqu'à la date actuelle.

Archivage continu

Avantages :

1. Il n'est pas nécessaire de disposer d'une sauvegarde des fichiers parfaitement cohérente comme point de départ.
2. Une sauvegarde continue est obtenue en continuant simplement à archiver les fichiers WAL (intéressant pour les grosses bases).
3. Il est possible de restaurer l'état de la base telle qu'elle était à tout point depuis la dernière sauvegarde.
4. À tout moment, une deuxième machine peut être montée et disposer d'une copie quasi-complète de la base.

Inconvénients :

1. Cette méthode ne supporte que la restauration d'un cluster de bases de données complet.
2. Un peu plus compliqué.