Travaux Pratiques de bases de données

Licence

— PostgreSQL—

Contraintes statiques

▸ **Exercice 1.**

- Construisez une table add qui a trois champs entiers, nommés a,b,somme.

  ```
   CREATE TABLE addition(a INTEGER, b INTEGER, Somme INTEGER);
  ```

  On veut à présent que pour une ligne donnée la somme soit egale à a+b. Une possibilite est de le verifier à l'insertion. Pour cela on peut recreer la table en ajoutant la contrainte statique :

  ```
  CONSTRAINT addition_ok CHECK(somme=a+b)
  ```

  ```
   CREATE TABLE addition(a INTEGER, b INTEGER, somme INTEGER,
  CONSTRAINT addition_ok CHECK(somme=a+b) );

   CREATE TABLE addition2(a INTEGER, b INTEGER,
     somme INTEGER  CONSTRAINT addition_ok CHECK(somme=a+b) );
  ```

- inserer plusieurs valeurs dans votre table.

  ```
  insert into addition values (2,3,5);
  INSERT 53748 1
  tppg1=> insert into addition values (2,3,51);
  ERROR:  ExecAppend: rejected due to CHECK constraint addition_ok
  ```

▸ **Exercice 2.** Creer une table avec un champs chaine de caractère, en ajoutant

```
check (reponse in ('oui','non'))
```

```
create table oui_non ( reponse char(3),
    constraint dadada check (reponse in ('oui','non')) );
create table voui_nan ( reponse char(3) check (reponse in ('oui','non')));
```

Voila ce que ca donne:

```
insert into oui_non values ('oui');
INSERT 53760 1
tppg1=> insert into oui_non values ('ou');
ERROR:  ExecAppend: rejected due to CHECK constraint dadada
```

▸ **Exercice 3.** A votre avis quel est l'effet de la contrainte addition_ok sur les insertions

```
insert into addition values (2,3,NULL);
insert into addition values (2,NULL,6);
insert into oui_non values(NULL);
```

Rq : de manière équivalente on peut ecrire

```
insert into addition (a, b) values (2, 3);
```

tout marche.

Au moment de la création d'une table, on peut, si on le desire, preciser qu'un champs ne peut pas prendre la valeur `NULL`. (C'est obligatoirement le cas pour une clé). La contrainte correspondante est `NOT NULL`

On reprend le problème de la somme. On veut maintenant qu'au moment de l'insertion le champs `somme` se remplisse automatiquement.

Les contraintes que nous venons de voir peuvent être qualifiées de statiques. Des modifications dynamiques seront faites par ce que l'on appelle les **triggers**.

Triggers

Supposons que l'on ait deux tables, **A** et **B**. La première contient des entiers, et la seconde n'est composée que d'une ligne, qui tient le compte du nombre d'elements de A. On veut qu'a chaque insertion dans **A** la table **B** soit automatiquement mise à jours.

```
create table A (val integer);
create table B (nb  integer);

insert into B values (0);
```

On va pour cela associer l'execution d'une fonction/procedure avec l'insertion dans la base.

```
create function incremente() returns opaque as
'begin
update b set nb=nb+1;
return new;
end;'
language 'plpgsql';
```

Si on veut la supprimer, on entre `drop function incremente();`

A chaque ligne (each row) inseree dans la table **A**, on souhaite declencher (trigger) cette fonction. Le code correspondant est :

```
create trigger keep_count
       after insert on A
       for each row
       execute procedure incremente();
```

Si on veut supprimer un trigger, on utilise la commande : `drop trigger keep_count on a;`

▸ **Exercice 4.** Pour maintenir le compte exact, il faut incrementer à l'ajout et decrementer à la suppression. C'est votre tour pour la suppression. Que se passe t'il si **B** contient plusieurs lignes ? Essayez de supprimer plusieurs lignes en une seule commande.

Bon, alors si **B** a plusieurs lignes, chacune est affectee par le trigger.

```
create function decremente() returns opaque as
'begin
update b set nb=nb-1;
return new;
```

```
end;'
language 'plpgsql';

create trigger keep_count2
       after delete on A
       for each row
       execute procedure decremente();

delete from a where val=34;
```

Les conditions possibles pour le declenchement d'un trigger sont {`before`,`after`}×{`insert`,`update`,`delete`}.

On veut a present qu'une table **C** vale en permanence la somme des elements de **A**. On va avoir besoin d'extraire la valeur d'une des lignes insérées. Pour cela on utilisera l'un des mots clés **:old,:new**. Voici un exemple :

```
create table c (total integer);
insert into c values (0);

create function somme_update() returns opaque as
'begin
  update c set total=total+new.val-old.val;
  return new;
 end;'
language 'plpgsql';


create trigger keep_sum
       after update on A
       for each row
       execute procedure somme_update();
```

Au moment ou le trigger se declenche, la ligne concernee par l'insertion, la suppression ou l'update est dupliquee. L'une d'entre elle est identifiee par `new`, l'autre par `old`.

▸ **Exercice 5.** Completer la definition de **C** pour l'insertion et la suppression. Que valent dans ce cas `new` et `old`.

Si on insere alors new est definie, et old est a null dans tout ses champs, reciproquement pour delete.

Rq :les histoires de after et before peuvent se faire imbriquer les triggers, c'est une source d'erreur statique (le parseur de postgres les refuses, il faut donc traiter ces coherences avec PHP par exemple)

QQ remarques propres a oracle/postgres : il ne peut pas y avoir 2 trigger qui se declanchent avec texto les memes premisses du genre before insert je fais +1 mais before insert je fais aussi +2 sur la ligne a inserer ...

en oracle on peut ecrire `create or replace` qui dispense de bcp de drop.

▸ **Exercice 6.** Reprenez l'exercice 1 pour faire en sorte que l'insertion de n'importe quelle valeur (a, b) par `insert into addition (a,b) values (..  , ..)` effectue en fait l'insertion de $(a, b, a + b)$. Pour cela, vous avez besoin de savoir que l'affectation se fait avec le symbole `:=`.

Reprenez cette insertion, pour que l'insertion ou la modification de n'importe quelle valeur, y compris NULL, pour le triplet $(a, b, somme)$. Donne un resultat coherent. Pour cela vous devez savoir que le langage `plpgsql` vous permet d'utiliser des conditionnelles dans les fonctions. Leur syntaxe est `IF (condition) THEN ...  End IF;` et qu'on peut enchainer des conditions avec `or`,`and`. Faites deux ou trois cas seulement, puis passez à la suite.

```
create table addition(a integer, b integer, somme integer);

create function somme1() returns opaque as
'begin
     new.somme := new.a+new.b;
  return new;
 end;'
language 'plpgsql';

create trigger somme
       before insert or update on addition
       for each row
       execute procedure somme();

insert into addition values (2,2,7);
select * from addition;

insert into addition (a,somme) values (3,45);


drop trigger somme on addition;
drop function somme();
```

A remarquer : le trigger se declenche avant l'insertion.

```
drop trigger somme on addition;
drop function somme();

create function somme() returns opaque as
'begin
  if  new.a !=null and new.b !=null
  then new.somme := new.a+new.b;
  end if;

..... je vous laisse faire les autres cas  ....

  return new;
 end;'
language 'plpgsql';
```