M1–Master Informatique – 2020/2021

Bases de données relationnelles

mars-avril 2021

Dénormalisation

Exercice 1 :
Nous reprenons le schéma relationnel sur les emprunts bancaires utilisé au TD précédent :

```
create table agence(
  nag integer constraint agence_pkey primary key,
  nomag varchar(70) not null,
  villeag varchar(30)
);
create table client(
  ncli integer constraint client_pkey primary key,
  nomcli varchar(30) not null,
  prenomcli varchar(30) not null,
  villecli varchar(30)
);
create table compte(
  ncompte integer constraint compte_pkey primary key,
  nag integer references agence,
  solde float default 0.0 not null,
  typecpte varchar(15)
);
create table compte_client(
  ncompte integer references compte,
  ncli integer references client,
  constraint compte_client_pkey primary key(ncompte, ncli)
);
create table emprunt(
  nemprunt integer constraint emprunt_pkey primary key,
  ncompte integer not null references compte,
  montant float not null
);
```

On a vu comment vérifier des contraintes avec des triggers. Les triggers servent aussi à maintenir la cohérence de la base lorsqu'on y introduit de la redondance.
Question 1.1 : Ecrire la requête SQL qui donne pour chaque compte le nombre total de propriétaires et le montant total des emprunts.

Question 1.2 : On suppose que la requête précédente est faite très souvent, sur un volume de données important. Afin de l'optimiser, on va stocker le résultat des calculs. L'idée est de ne pas être obligé d'aller lire les tables `COMPTE_CLIENT` et `EMPRUNT`.
Ajouter des colonnes `nb_proprietaires` et `montant_emprunts` à la table `COMPTE`.

Question 1.3 : Ecrire et exécuter la requête SQL `update` qui met à jour ces nouvelles colonnes en fonction des données déjà présentes dans la base.

Question 1.4 : Définir des triggers pour maintenir ces colonnes à jour lorsqu'on modifie (par `insert`, `delete`, `update`) les données des tables `COMPTE_CLIENT` ou `EMPRUNT`. Pour éviter de refaire toujours les mêmes calculs, ces triggers calculent les nouvelles valeurs de `nb_proprietaires` et `montant_emprunts` en fonction des anciennes – contrairement à l'instruction `update` de la question précédente.

Question 1.5 : En quoi la définition de ces nouveaux attributs peut modifier la solution implémentée au TD précédent pour vérifier que seuls les comptes courants peuvent être partagés.

Question 1.6 : De même, en quoi la définition de ces nouveaux attributs simplifie la vérification du montant total des emprunts liés à un compte ?

Exercice 2 : Nous allons maintenant nous intéresser au partitionnement de table. Postgres propose un mécanisme de partitionnement de table qu'il faut maintenir à la main. Il est probable qu'une prochaine version de Postgres permette de maintenir ce partitionnement automatiquement (on en parle pour la prochaine version). En attendant, nous allons découvrir le mécanisme de partitionnement offert par la version actuelle de Postgres et utiliser des fonctions stockées pour simplifier la maintenance des partitions.
Nous allons pour cela travailler avec **les données du TD sur l'indexation** qui vous sont distribuées via deux fichiers :

- `music-schema.sql` qui contient le schema de la base de données,
- `music-data.sql` qui contient les données.

Nous allons nous intéresser à la table `facture`. Cette table sert essentiellement à faire des bilans d'activité mensuels, aussi, nous décidons de la découper en plusieurs tables, chaque table représentant les factures d'un mois particulier.
Postgres propose la commande suivante :

```
CREATE TABLE tbl (col1 tp1, ..., coln tpn) PARTITION BY RANGE(col);
```

Cela signifie que la table `tbl` est en quelque sorte virtuelle et est composée de plusieurs tables qui la découpe en fonction des valeurs de la colonne `col`.
Question 2.1 : Le fonctionnement des tables partitionnées est un peu limité dans Postgres. En particulier, il n'est pas possible qu'elles contiennent de clé primaire. Modifier le schéma donné dans le TD sur l'indexation pour que la table `facture` puisse être partionnée en fonction des dates des factures.

Question 2.2 : Afin de créer les tables qui composent la partition de `facture`, nous allons utiliser la commande suivante :

```
CREATE TABLE tbl_part PARTITION OF tbl
FOR VALUES FROM (low) TO (high);
```

Avec cette instruction, `tbl_part` entre dans la partition de `tbl` en prenant en charge les valeurs allant de `low` (inclus)à `high` (exclus).
Dès que nous avons plusieurs mois à gérer, la création des tables qui composent la partition devient pénible. Tous les mois, il faut créer une nouvelle table pour la partition. Au lieu de cela, nous allons utiliser une fonction stockée qui prend en argument une date et vérifie que la table de la partition correspondant à ce mois existe :

- si c'est le cas, elle renvoie `TRUE`,
- sinon crée la table, affiche un message d'information, puis renvoie `TRUE`.

Nous souhaitons également que lors de la création des tables de la partition un index sur la colonne `fac_date` soit créé.
Pour réaliser cette fonction il faut penser à plusieurs choses :

- utiliser une convention de nommage simple des partitions en fonction de leur mois,
- utiliser les fonctions sur les dates proposées par Postgres (`date_trunc`, `to_char`, …),
- utiliser les méta-données liées au schéma pour tester l'existence d'une table (c.f. la documentation de Postgres),
- utiliser la commande `EXECUTE` pour lancer les requêtes de création de table (**attention :** `EXECUTE` s'applique à une chaîne de caractères. Il s'agit donc de composer les requêtes passées à `EXECUTE` sous forme de chaîne de caractères.)

Question 2.3 : Pour pouvoir rentrer les données dans la table, il faut au préalable que les tables correspondant à leur date dans la partition soient déjà créées. Ainsi, la commande :

```
INSERT INTO facture VALUES(1503, '2021-01-15', 5.15);
```

renvoie l'erreur suivante :

```
ERROR:  no partition of relation "facture" found for row
DETAIL:  Partition key of the failing row contains (fac_date) = (2021-01-15).
```

Toutes les factures contenues dans le fichier sql sont datées de 2020. Même avec la fonction de la question précédente, créer toutes les tables pour 2020 est fastidieux.
Écrivez une fonction stockée qui prend en argument une date et crée les tables pour l'année de cette date.
Vous pourrez utiliser avec profit la fonction `generate_series` de Postgres qui permet de générer des intervalles pour des types donnés. Par exemple :

```
SELECT * FROM generate_series('2017-01-01'::DATE, '2017-02-28'::DATE, '1 day');
```

renvoie la séquence des jours du mois de janvier et février 2017 (c.f. la documentation de Postgres).
Utilisez la fonction que vous venez d'écrire pour créer les tables de la partition pour l'année 2020. Testez votre configuration à l'aide des entrées dans la suite du fichier. Si vous n'avez pas d'erreur à l'insertion, vérifiez tout de même que les données sont bien présentes avec quelques requêtes.

Question 2.4 : Bien que le partitionnement de la table `facture` ne l'autorise pas à avoir de clé primaire, nous souhaitons que `fac_num` soit une clé primaire pour chaque table composant la partition de `facture`. Pour cela, nous allons créer chaque partition en deux temps :

— On crée la table en utilisant un schéma similaire à celui de `facture`, avec `fac_num` comme clé primaire,
— on rattache ensuite cette table à la partition de `facture` en utilisant la commande :

```
ALTER TABLE tbl ATTACH PARTITION tbl_part
    FOR VALUES FROM (low) TO (high);
```

qui rajoute la table `tbl_part` à la table partitionnée `tbl`.
Modifiez la fonction stockée de la première question pour qu'elle crée les tables de cette façon.

Question 2.5 : Afin de conserver une bonne performance de la base de données, on souhaite ne conserver que les factures des six derniers mois. Vous allez pour cela écrire une fonction stockée qui réalise les opérations suivantes :
— détache les partitions de plus de six mois qui composent la table `facture`.
Pour détacher une partition d'une table partitionnée, utilisez la commande suivante :

```
ALTER TABLE tbl_part DETACH PARTITION tbl;
```

qui élimine la composante `tbl_part` de la table découpée `tbl`. On souhaite que si cette fonction est appelée une seconde fois, aucune erreur ne se produise. Pour cela, il convient de vérifier qu'une table appartient bien à une partition, ce qui revient à vérifier que l'attribut `relispartition` est vrai pour la table concernée dans les métadonnées de `pg_class` (c.f. la documentation de Postgres).
Testez votre fonction sur les données fournies.
NB : pour rechercher les tables de la partition en fonction de leur date, envisagez de changer de convention de nommage pour les tables de votre partition pour vous faciliter la tâche.

Question 2.6 : Nous souhaiterions automatiser plus avant le processus de maintenance des partitions. Malheureusement, nous avons atteint les limites de ce que l'on pouvait faire à partir de `pgplsql`. Nous souhaiterions avoir les fonctionnalités suivantes :
— lancer la mise à jours de la partition tous les mois automatiquement,
— sauvegarder les tables détachées (avec `pg_dump` par exemple), puis les effacer (toujours automatiquement).
Comment vous y prendriez vous pour réaliser ce type d'opération ?