

BDR - Indexation

semestre 2 - 2024

1 Présentation de PostgreSQL

PostgreSQL est un SGBD relationnel open source qui utilise une architecture client-serveur. Nous utiliserons le serveur du département : `webtp.fil.univ-lille.fr`

Ce semestre, nous utiliserons divers clients : `phppgadmin` client web d'administration, `psql` client en mode texte, ou `dbeaver` client graphique non lié à postgresQL et se présentant comme "gestionnaire universel de bases de données".

Voici quelques notions sur le modèle de données :

1. Un serveur postgresql héberge plusieurs **bases**. Sur le serveur du FIL, il y a une base `<nom_user>` pour chaque utilisateur de login `<nom_user>`, mais ça n'est pas du tout généralisable à tous les serveurs postgresql.
2. Une base contient des **schémas**. Lorsqu'on crée une base sur postgresql, cela entraîne la création d'un schéma appelé **public** au sein de cette base. Sur le serveur du FIL vous n'avez pas le droit de créer de nouvelle base mais vous pouvez créer des schémas au sein de votre base.
3. Un schéma contient des objets : tables, vues, indexes, fonctions stockées etc ...

Exercice 1.1 : Le schéma que nous manipulons aujourd'hui représente des ventes de produits chez un disquaire. Un produit est un disque (par exemple un CD) correspondant à un Album. La facture (ou ticket de caisse) contient des lignes, chacune indiquant qu'on a acheté une certaine quantité d'un produit à un prix unitaire de vente. Ce schéma contient les tables :

```
ALBUM(al_id, al_titre, al_sortie)
PRODUIT(prod_id, prod_al, prod_code_barre_ean, prod_type_support)
FACTURE(fac_num, fac_date, fac_montant)
LIGNE_FACTURE(lig_facture, lig_produit, lig_prix_vente, lig_quantite)
```

Question 1 : Exécutez le script `TD_music_data.sql` disponible sur Moodle.

Question 2 : En fonction du client postgresql que vous ou votre enseignant.e a choisi, vérifiez que vous êtes capable de trouver les informations suivantes :

1. la liste des schémas de votre base ;
2. la liste des tables du schéma `music` ;
3. le schéma relationnel (i.e. les colonnes avec leur type) de la table `facture` ;
4. le contenu de la table `facture` ;

2 Indexation

Nous allons étudier les plans d'exécution générés par le moteur SQL. Un plan d'exécution est la description des opérations d'accès aux données nécessaires à l'exécution d'une requête. C'est un arbre, et les opérations qui sont exécutées les premières sont aux feuilles. L'instruction `explain <requete>` donne le plan d'exécution de la requête.

Si on veut également obtenir des informations sur la performance de l'exécution, on utilise l'instruction `explain analyze <requete>`

- indexes : <https://www.postgresql.org/docs/14/indexes.html>
- explain : <https://www.postgresql.org/docs/14/using-explain.html>

Pour approfondir le sujet :

- Le site <https://use-the-index-luke.com/> pour l'indexation, et des comparaisons entre différents SGBD.
- Le site <https://explain.depesz.com/> pour les plans d'exécution de PostgreSQL.

Important : Pour que le moteur SQL puisse avoir accès aux informations de volumétrie avant de construire le plan d'exécution, il faut analyser les tables par la commande **Analyze**. Avant de commencer les exercices, exécutez donc l'instruction :

```
ANALYZE album, produit, facture, ligne_facture ;
```

Exercice 2.1 : INDEX SUR LA CLÉ PRIMAIRE

Question 1 : La table **FACTURE** comporte une clé primaire **fac_num**. Vérifiez que cette table est bien indexée selon cette colonne. De quel type est cet index (table de hashage, Btree, etc) ?

Question 2 : L'index est utilisé lorsque la requête porte sur la colonne indexée. Vérifiez cela en examinant le plan d'exécution de la requête suivante :

```
SELECT fac_num, fac_date, fac_montant FROM facture  
WHERE fac_num = 500;
```

Question 3 : Est-ce que l'index est utilisé pour la requête suivante ? Pourquoi ?

```
SELECT fac_num, fac_date, fac_montant FROM facture  
WHERE floor(fac_num/100) = 5 ;
```

Sinon, trouvez une requête équivalente qui permet d'utiliser l'index.

Remarque : La problématique de l'utilisation de fonctions posée en question 1.3 est un grand classique en optimisation de requêtes. Il se pose parfois de manière insidieuse, prenons par exemple le cas de certains ORM qui utilisent **upper** ou **lower** sur les chaînes de caractères, sans que le développeur en soit averti. Cela a des conséquences sur les performances. Heureusement on peut définir un index basé sur un appel de fonction.

Question 4 : L'index n'est bien sûr d'aucune utilité lorsque la clause `where` ne porte pas sur la colonne indexée. Mais qu'en est-il si on combine plusieurs conditions dont une sur la colonne indexée ? Répondez à cette question en recherchant les lignes qui ont un numéro de facture entre 500 et 550, et dont la date est entre le 1er juin et le 31 août 2017.

Exercice 2.2 : CRÉATION DE NOUVEAUX INDEXES

Dans l'exercice précédent, on a utilisé les indexes créés par défaut, indexes sur la clé primaire. La question 1.1 vous a permis par exemple de retrouver l'instruction SQL de définition de l'index `facture_pkey`. Il est aussi possible de définir d'autres indexes, voici une version très simplifiée de la syntaxe :

```
CREATE [ UNIQUE ] INDEX nom ON nom_table
    [ USING méthode ] ( [ nom_colonne | ( expression ) ] [, ...] )
-- méthodes possibles : btree ou hash
-- note : un index unique est forcément un BTree
```

Question 1 : En imaginant qu'on a souvent besoin d'accéder aux factures par paquet de 100 (cf question 1.3) créez un index sur l'expression `floor(fac_num/100)`. Quelle est la sélectivité de cet index ?

Question 2 : Comparez le plan et les temps de préparation et d'exécution entre :

1. la requête de la question 1.3, qui utilise `floor(...)` mais sans index adhoc.
2. la même requête mais lorsque l'index de la question précédente est créé
3. une requête équivalente, *range query* sur `fac_num`

Question 3 : On suppose maintenant qu'on a souvent besoin d'accéder aux factures selon leur date d'émission. Créez un index `btree` sur cette date.

Question 4 : Le temps est un critère énormément utilisé dans les requêtes. On veut par exemple accéder aux factures par mois (e.g. janvier 2021, février 2021, etc), ou par trimestre. Postgres offre plusieurs solutions pour extraire le mois d'une date (`date_trunc`, `to_char`, etc). Essayez de définir un index à partir d'une expression qui extrait le mois et l'année de la date d'émission. Quelle erreur se produit ?

Explication : Lorsqu'on définit un index basé sur une expression ou une fonction, il est évidemment très important que celle-ci soit déterministe. Postgres appelle une telle fonction *immutable*, on retrouve ce critère dans d'autres SGBD, sous d'autres termes (Oracle appelle ces fonctions *deterministic*). C'est le cas de `floor(fac_num/100)` utilisé précédemment. Le problème des dates est qu'on peut avoir un décalage d'une journée pour un même instant : certaines îles du Pacifique sont à GMT-12 et d'autres à GMT+14. Donc, selon la configuration du serveur, les fonctions sur les dates ne donneront pas le même résultat.

Question 5 : Est-ce qu'on peut utiliser l'index de la question 2.3 pour rechercher les factures pour un mois ou pour un trimestre d'une année ? Si oui, donnez un exemple d'une telle requête dont le plan utilise l'index.

Exercice 2.3 : INDEX MULTI-COLONNES

La table `LIGNE_FACTURE` est indexée par `(lig_facture, lig_produit)`.

Question 1 : Parmi les requêtes suivantes, quelles sont celles qui peuvent utiliser l'index ? Examinez les plans d'exécution pour comprendre les différentes stratégies utilisées pour accéder aux données.

```
SELECT * FROM ligne_facture
WHERE lig_facture = 250 AND lig_produit = 44 ;
```

```
SELECT * FROM ligne_facture
WHERE lig_facture BETWEEN 1000 AND 1005 ;
```

```
SELECT * FROM ligne_facture
WHERE lig_produit = 44 ;
```

Question 2 : Il faut parfois choisir entre créer un index sur le couple `(col1, col2)` ou deux indexes, i.e. un sur chacune des colonnes. Cela dépend du type de requêtes d'interrogation (`select`) et de mises à jour (`insert`, `delete`, `update`) que l'on fait le plus fréquemment. Quels avantages et inconvénients voyez-vous pour chacune de ces possibilités ?

Pour résoudre la question des combinaisons de critères, les indexes multi-colonnes ne sont pas envisageables si le nombre de colonnes est supérieur à 2. Les entrepôts de données disposent d'indexes Bitmap : ce sont des tableaux de bits, avec autant d'éléments qu'il y a de lignes dans la table à indexer. Une combinaison de critères se traduit en une opération booléenne sur ces tableaux de bits et permet rapidement d'accéder aux lignes qui satisfont tous les critères. Postgres ne propose pas ces bitmap indexes, mais on a vu dans les exemples précédents qu'il construit en mémoire des tableaux de bits pour combiner les critères.

Exercice 2.4 : VOLUMÉTRIE

Certaines requêtes peuvent être calculées uniquement avec l'index. Quand il est nécessaire d'accéder à la table, l'utilisation ou non d'un index dépend de la volumétrie des données et du résultat (taille de la table, de l'index, sélectivité de l'index, sélectivité de la requête, etc.).

Question 1 : La requête suivante utilise l'index avec un plan d'exécution légèrement différent. Que signifie-t-il ?

```
SELECT COUNT(*) FROM facture
WHERE fac_num BETWEEN 500 AND 550;
```

Pour les questions suivantes, il peut être utile de connaître la volumétrie des tables et des indexes. Postgres propose des fonctions calculant la taille des tables et des indexes associés. Par exemple, vous obtenez les tailles concernant la table `ALBUM` avec les requêtes :

```
-- taille de la table
SELECT pg_size_pretty(pg_table_size('music.album'));

-- taille de l'index
SELECT pg_size_pretty(pg_indexes_size('music.album'));
```

Question 2 : Toutes les tables sont indexées selon leur clé primaire. Posez une requête sur la table `ALBUM` pour retrouver la ligne telle que `al_id = 20`. Est-ce que l'index sur la clé primaire est utilisé ? Pourquoi ?

Question 3 : Examinez le plan d'exécution de la requête suivante :

```
SELECT lig_facture,
       SUM(lig_quantite) AS nb_produits,
       COUNT(*) AS nb_produits_distincts
FROM ligne_facture
GROUP BY lig_facture;
```

Comparez avec ceux obtenus lorsqu'on ajoute à la précédente requête une clause `where` qui a pour effet de :

1. sélectionner les factures des numéros 500 à 550
2. sélectionner les factures des numéros 200 à 1000

Comment expliquer les différences entre ces deux derniers plans ?

Exercice 2.5 : INDEX PARTIEL

Un index partiel permet de définir un index sur un sous-ensemble des lignes d'une table. Supposons que l'on s'intéresse aux produits achetés en plusieurs exemplaires. On pourra définir l'index suivant :

```
CREATE INDEX ligne_facture_par_prod
ON ligne_facture(lig_produit)
WHERE lig_quantite > 1 ;
```

Question 1 : Vérifier que l'index précédent est utilisé dans la requête suivante :

```
SELECT lig_facture, lig_produit, lig_quantite FROM ligne_facture
WHERE lig_quantite > 1 ;
```

Est-ce que c'est toujours le cas si on remplace la condition par `lig_quantite > 2` ?

Exercice 2.6 : INDEX COUVRANT

Observez le plan d'exécution pour la requête suivante :

```
SELECT fac_num, fac_montant FROM facture
WHERE fac_num between 100 and 150 ;
```

Maintenant, créez l'index suivant :

```
CREATE INDEX facture_num_include_montant ON facture(fac_num) INCLUDE(fac_montant) ;
```

puis observez si le plan pour la requête précédente a changé.

Question 1 : Quelle est la particularité de l'index que nous avons créé ? Quels en sont les avantages et inconvénients ?

Exercice 2.7 : JOINTURE

Nous avons étudié la semaine dernière plusieurs algorithmes de jointures. Un index peut être utilisé pour calculer une jointure. Parfois une structure de type index (par exemple une Map lors d'un Hash Join) est construite en mémoire pour éviter une double boucle i.e éviter de lire de nombreuses fois l'une des deux tables.

Observez le plan d'exécution pour la requête suivante :

```
SELECT prod_al, prod_type_support, lig_quantite, lig_prix_vente
FROM ligne_facture
Join produit ON lig_produit = prod_id ;
```

Question 1 : Comment se fait le calcul de cette jointure ? Combien de fois est lue chaque table ? Est-ce que le SGBD utilise un index ?

Indication : Hash Join est une opération de jointure qui se fait en 2 sous-opérations. Tout d'abord, la sous opération Hash appelle une autre opération qui fabrique en mémoire (ou sur disque) une Map avec les lignes d'une table selon un attribut utilisé dans la condition de jointure. Ensuite cette Map est utilisée lors de la lecture de l'autre table (seconde sous-opération, par exemple Seq Scan).

Question 2 : Même question pour cette seconde requête, où on a ajouté une clause Where :

```
SELECT prod_al, prod_type_support, lig_quantite, lig_prix_vente
FROM ligne_facture JOIN produit ON lig_produit = prod_id
WHERE lig_facture = 500;
```

Indication : BitMap Index Scan crée un index bitmap avec 1 bit par bloc de la table à traiter. Ce bit est à 1 si le bloc est susceptible de contenir des lignes utiles à la requête (c'est une estimation). Ensuite, BitMap Head Scan utilise cet index bitmap pour lire la table partiellement. C'est donc une amélioration du Seq Scan qui lit la table dans sa totalité.