Optimisation SQL

Rendu TP2

Ricardo Rodríguez

Optimisation SQL
M1 ILSEN
Université d'Avignon et des Pays de Vaucluse

1. Requête R1

Ecrivez une requête SQL (R1) qui affiche le nombre de déclarations annuelles effectuées par les chefs-lieux de canton dont le nombre des naissances est compris entre 100 et 500. Cette requête devra afficher la valeur de 745 et être écrite avec :

1. *une jointure interne entre les tables commune et naissance*
2. *une restriction sur le nombre de naissances entre [100, 500]*
3. *une restriction sur les communes « chef-lieu » de canton*

Analysez le plan d'exécution de la requête R1

Version du serveur

Je travaille sur un serveur local dans ma machine personnelle, version de PostgreSQL 9.6.1. Système d'exploitation Windows 10 64 bits.

Requête

```
SELECT COUNT(*)
FROM naissance
      INNER JOIN commune USING (code_departement, code_commune)
WHERE naissance.nb BETWEEN 100 AND 500
      AND commune.chef_lieu = 1;
```

Résultat

```
 count
-------
      745
(1 row)
```

Plan d'exécution

Explain

```
QUERY PLAN
------------------------------------------------------------------------------------------
--
  Aggregate  (cost=949.94..949.95 rows=1 width=8)
   ->  Hash Join  (cost=181.60..949.73 rows=84 width=0)
       Hash Cond: (((naissance.code_departement)::text = (commune.code_departement)::text)
AND ((naissance.code_commune)::text = (commune.code_commune)::text))
       ->  Seq Scan on naissance  (cost=0.00..662.13 rows=1402 width=7)
             Filter: ((nb >= 100) AND (nb <= 500))
       ->  Hash  (cost=169.59..169.59 rows=801 width=7)
             ->  Seq Scan on commune  (cost=0.00..169.59 rows=801 width=7)
                   Filter: (chef_lieu = 1)
```

Le plan crée la table de hash avec les lignes de la table "commune" en réalisant un parcours séquentiel et un filtre sur le champ chef_lieu. Il n'y a pas d'indice, il faut donc faire un filtre classique.
Le plan fait aussi un parcours séquentiel sur "naissance" et applique le filtre sur la colonne "nb"de chaque ligne.

Finalement il fait la jointure des résultats à l'aide de la table de hash, dont la clé est la pair (code_departement, code_commune).

2. Requête R2

Transformez la requête SQL (R1) en une requête équivalente (R2) écrite avec :

1. une projection simple du nombre de déclarations annuelles sur la table naissance

2. une restriction sur le nombre de naissances entre [100, 500]

3. une restriction avec l'opérateur IN appliqué sur une sous-requête sélectionnant les communes « chef-lieu » de canton

Analysez le plan d'exécution de la requête R2.

Requête

```
SELECT COUNT(naissance.nb)
FROM naissance
WHERE naissance.nb BETWEEN 100 AND 500
      AND (naissance.code_departement, naissance.code_commune) IN
      (SELECT code_departement, code_commune
       FROM commune
       WHERE chef_lieu = 1);
```

Résultat

```
 count
-------
     745
(1 row)
```

Plan d'exécution

Explain

```
QUERY PLAN
-----------------------------------------------------------------------------------------
--
 Aggregate  (cost=863.81..863.82 rows=1 width=8)
   ->  Hash Join  (cost=187.09..860.30 rows=1402 width=4)
       Hash Cond: (((naissance.code_departement)::text = (commune.code_departement)::text)
AND ((naissance.code_commune)::text = (commune.code_commune)::text))
       ->  Seq Scan on naissance  (cost=0.00..662.13 rows=1402 width=11)
             Filter: ((nb >= 100) AND (nb <= 500))
       ->  Hash  (cost=178.99..178.99 rows=540 width=7)
             ->  HashAggregate  (cost=173.59..178.99 rows=540 width=7)
                   Group Key: (commune.code_departement)::text,
(commune.code_commune)::text
                   ->  Seq Scan on commune  (cost=0.00..169.59 rows=801 width=7)
                         Filter: (chef_lieu = 1)
```

Le plan commence par créer une table de hash avec les lignes de la table "commune" qui passent le filtre sur "chef_lieu". Le parcours de cette table est fait de manière séquentielle. La table de

hash est créée avec la fonction HashAggregate, qui va générer une clé en combinant les colonnes “code_departement” et “code_commune”.
Le plan fait aussi un parcours séquentiel sur “naissance” et applique le filtre sur la colonne “nb” de chaque ligne. Pour chaque enregistrement il fera ensuite la jointure avec la table de hachage.

L’utilisation de HashAggregate ne nécessite pas de données pré-triées. Par contre, on a besoin de plus de mémoire pour stocker la table. La table de hachage résultante contient moins d’éléments que dans le premier cas (540 vs 801), ce qui accélère la jointure avec “naissance” et donne comme résultat un temps d'exécution plus bas.

3. Désactiver HashAggregate

Désactivez le booléen enable_hashagg et générez à nouveau le plan d’exécution de la requête R2. Comparez le plan obtenu avec celui de la question [2]. Que pouvez-vous en déduire ?

Réactivez le booléen enable_hashagg.

Requête

```
SET enable_hashagg = false;

EXPLAIN SELECT COUNT(naissance.nb)
FROM naissance
WHERE naissance.nb BETWEEN 100 AND 500
      AND (naissance.code_departement, naissance.code_commune) IN
      (SELECT code_departement, code_commune
      FROM commune
      WHERE chef_lieu = 1);

SET enable_hashagg = true;
```

Plan d’exécution

Explain

```
QUERY PLAN
------------------------------------------------------------------------------------------
--
 Aggregate  (cost=899.04..899.05 rows=1 width=8)
   ->  Hash Join  (cost=222.33..895.54 rows=1402 width=4)
       Hash Cond: (((naissance.code_departement)::text =
((commune.code_departement)::text)) AND ((naissance.code_commune)::text =
((commune.code_commune)::text)))
       ->  Seq Scan on naissance  (cost=0.00..662.13 rows=1402 width=11)
             Filter: ((nb >= 100) AND (nb <= 500))
       ->  Hash  (cost=214.23..214.23 rows=540 width=7)
             ->  Unique  (cost=208.22..214.23 rows=540 width=7)
                   ->  Sort  (cost=208.22..210.22 rows=801 width=7)
                         Sort Key: ((commune.code_departement)::text),
((commune.code_commune)::text)
                         ->  Seq Scan on commune  (cost=0.00..169.59 rows=801 width=7)
                               Filter: (chef_lieu = 1)
```

Vu que l'on a désactivé HashAggregate, le nouveau plan a besoin de trier les éléments de la table "commune". Cette procédure ralenti l'exécution de la requête et donne comme résultat un temps total plus élevé.

Le plan fait un parcours séquentiel dans "commune" et filtre les données par rapport à la valeur du champ "chef_lieu". Il fait ensuite le trie de ces données et puis il les insère dans une table de hash.
Le plan fait toujours un parcours séquentiel sur "naissance", applique le filtre sur la colonne "nb" et fait la jointure avec la table de hachage.

4. Requêtes avec constraints (PK et FK)

Exécutez le script tp_dbconstraints.pgsql qui complète la mise en place des tables avec l'ajout des clés primaires et des contraintes référentielles. Puis générez à nouveau les plans d'exécution de vos requêtes (R1 et R2).
Expliquez les différences observées entre les plans d'exécution c-a-d avant et après la mise en uvre des contraintes PK et FK.

Plan d'exécution R1

Explain

```
QUERY PLAN
------------------------------------------------------------------------------------------
--
 Aggregate  (cost=950.05..950.06 rows=1 width=8)
   ->  Hash Join  (cost=181.60..949.73 rows=128 width=0)
       Hash Cond: (((naissance.code_departement)::text = (commune.code_departement)::text)
AND ((naissance.code_commune)::text = (commune.code_commune)::text))
       ->  Seq Scan on naissance  (cost=0.00..662.13 rows=1402 width=7)
             Filter: ((nb >= 100) AND (nb <= 500))
       ->  Hash  (cost=169.59..169.59 rows=801 width=7)
             ->  Seq Scan on commune  (cost=0.00..169.59 rows=801 width=7)
                   Filter: (chef_lieu = 1)
```

Pour cette première requête, l'ajout des clés primaires et foraines ne change pas ni le plan d'exécution ni le temps total. Le résultat de la jointure avec la table de hachage (Hash Join) a même plus de lignes que dans le cas sans PK et FK.

Plan d'exécution R2

Explain

```
QUERY PLAN
------------------------------------------------------------------------------------------
--
 Aggregate  (cost=858.60..858.61 rows=1 width=8)
   ->  Hash Join  (cost=181.60..855.09 rows=1402 width=4)
       Hash Cond: (((naissance.code_departement)::text = (commune.code_departement)::text)
AND ((naissance.code_commune)::text = (commune.code_commune)::text))
```

```
->  Seq Scan on naissance  (cost=0.00..662.13 rows=1402 width=11)
      Filter: ((nb >= 100) AND (nb <= 500))
->  Hash  (cost=169.59..169.59 rows=801 width=7)
      ->  Seq Scan on commune  (cost=0.00..169.59 rows=801 width=7)
            Filter: (chef_lieu = 1)
```

Pour la deuxième requête le plan résultant est le même que pour R1.
Le temps d'exécution de ce plan est à peine plus court que dans le cas sans PK et FK (858.61 vs 863.82), mais cette différence est dépréciable.
La seule différence de ce plan avec celui de R1 est la quantité de lignes dans la table finale (1402 dans ce cas vs 128 pour R1).

5. Requêtes avec indices

Exécutez le script tp_dbindex.pgsql qui complète la mise en place des tables avec l'ajout d'index sur les contraintes référentielles. Puis générez à nouveau les plans d'exécution de vos requêtes (R1 et R2).
Expliquez les différences observées entre les plans d'exécution c-a-d avant et après la mise en uvre des index sur les FK.

J'ai aussi fait DROP des contraintes PKcommune et FK1naissance.

Plan d'exécution R1

Explain

```
QUERY PLAN
------------------------------------------------------------------------------------------
--
 Aggregate  (cost=950.31..950.32 rows=1 width=8)
   ->  Hash Join  (cost=181.60..950.10 rows=84 width=0)
       Hash Cond: (((naissance.code_departement)::text = (commune.code_departement)::text)
AND ((naissance.code_commune)::text = (commune.code_commune)::text))
       ->  Seq Scan on naissance  (cost=0.00..662.13 rows=1407 width=7)
             Filter: ((nb >= 100) AND (nb <= 500))
       ->  Hash  (cost=169.59..169.59 rows=801 width=7)
             ->  Seq Scan on commune  (cost=0.00..169.59 rows=801 width=7)
                   Filter: (chef_lieu = 1)
```

Pour cette première requête, l'ajout des indices ne change pas ni le plan d'exécution ni le temps total. Par contre, le résultat de la jointure avec la table de hachage (Hash Join) a seulement 84 lignes. Ceci peut être grâce à l'ajout de l'indice sur la paire de colonnes (code_departement, code_commune) dans "naissance", ce qui trie les données dans ces colonnes.

Il est possible que la quantité de données affecte le résultat attendue et l'optimiseur décide de ne pas accéder à l'indice.

Comme commentaire, j'utilise la version 9.6.2 de PostgreSQL. Il est de même possible que dans cette version les stratégies aient été modifiées afin d'améliorer leur performances et cela peut affecter les résultats.

Plan d'exécution R2

Explain

```
QUERY PLAN
------------------------------------------------------------------------------------------
--
 Aggregate  (cost=863.87..863.88 rows=1 width=8)
   ->  Hash Join  (cost=187.09..860.35 rows=1407 width=4)
       Hash Cond: (((naissance.code_departement)::text = (commune.code_departement)::text)
AND ((naissance.code_commune)::text = (commune.code_commune)::text))
       ->  Seq Scan on naissance  (cost=0.00..662.13 rows=1407 width=11)
             Filter: ((nb >= 100) AND (nb <= 500))
       ->  Hash  (cost=178.99..178.99 rows=540 width=7)
             ->  HashAggregate  (cost=173.59..178.99 rows=540 width=7)
                   Group Key: (commune.code_departement)::text,
(commune.code_commune)::text
                   ->  Seq Scan on commune  (cost=0.00..169.59 rows=801 width=7)
                         Filter: (chef_lieu = 1)
```

Pour la deuxième requête le plan résultant est à nouveau celui qui utilise HashAggregate et le temps d'exécution est exactement le même que dans ce cas-là.

6. Améliorations possibles

Proposez et justifiez une utilisation d'index susceptible d'améliorer les performances des plans d'exécution. Si cela n'est pas possible, expliquez en la raison.

Indice sur "nb"

```
CREATE INDEX naissance_nb_idx ON naissance USING btree(nb);
```

Explain R1

```
QUERY PLAN
------------------------------------------------------------------------------------------
--
 Aggregate  (cost=515.49..515.50 rows=1 width=8)
   ->  Hash Join  (cost=212.26..515.28 rows=84 width=0)
       Hash Cond: (((naissance.code_departement)::text = (commune.code_departement)::text)
AND ((naissance.code_commune)::text = (commune.code_commune)::text))
       ->  Bitmap Heap Scan on naissance  (cost=30.66..227.69 rows=1402 width=7)
             Recheck Cond: ((nb >= 100) AND (nb <= 500))
             ->  Bitmap Index Scan on naissance_nb_idx  (cost=0.00..30.31 rows=1402
width=0)
                   Index Cond: ((nb >= 100) AND (nb <= 500))
       ->  Hash  (cost=169.59..169.59 rows=801 width=7)
             ->  Seq Scan on commune  (cost=0.00..169.59 rows=801 width=7)
                   Filter: (chef_lieu = 1)
```

Explain R2

```
QUERY PLAN
------------------------------------------------------------------------------------------
--
 Aggregate  (cost=429.36..429.37 rows=1 width=8)
   ->  Hash Join  (cost=217.75..425.86 rows=1402 width=4)
       Hash Cond: (((naissance.code_departement)::text = (commune.code_departement)::text)
AND ((naissance.code_commune)::text = (commune.code_commune)::text))
       ->  Bitmap Heap Scan on naissance  (cost=30.66..227.69 rows=1402 width=11)
             Recheck Cond: ((nb >= 100) AND (nb <= 500))
             ->  Bitmap Index Scan on naissance_nb_idx  (cost=0.00..30.31 rows=1402
width=0)
                   Index Cond: ((nb >= 100) AND (nb <= 500))
       ->  Hash  (cost=178.99..178.99 rows=540 width=7)
             ->  HashAggregate  (cost=173.59..178.99 rows=540 width=7)
                   Group Key: (commune.code_departement)::text,
(commune.code_commune)::text
                   ->  Seq Scan on commune  (cost=0.00..169.59 rows=801 width=7)
                         Filter: (chef_lieu = 1)
```

On pourrait utiliser un indice sur la colonne “nb” de la table “naissance”. Cet indice est utilisé pour appliquer le filtre sur “nb”, ce qui accélère la recherche. Dans le plan on peut voir l’utilisation de l’index dans “Bitmap Index Scan”.

On peut voir que le temps d’exécution de la requête est diminué pour R1 et R2.

Indice sur “chef_lieu”

```
CREATE INDEX commune_chef_lieu_idx ON commune USING btree(chef_lieu);
```

Explain R1

```
QUERY PLAN
------------------------------------------------------------------------------------------
--
 Aggregate  (cost=868.86..868.87 rows=1 width=8)
   ->  Hash Join  (cost=100.52..868.65 rows=84 width=0)
       Hash Cond: (((naissance.code_departement)::text = (commune.code_departement)::text)
AND ((naissance.code_commune)::text = (commune.code_commune)::text))
       ->  Seq Scan on naissance  (cost=0.00..662.13 rows=1402 width=7)
             Filter: ((nb >= 100) AND (nb <= 500))
       ->  Hash  (cost=88.51..88.51 rows=801 width=7)
             ->  Bitmap Heap Scan on commune  (cost=18.49..88.51 rows=801 width=7)
                   Recheck Cond: (chef_lieu = 1)
                   ->  Bitmap Index Scan on commune_chef_lieu_idx  (cost=0.00..18.29
rows=801 width=0)
                         Index Cond: (chef_lieu = 1)
```

Explain R2

```
QUERY PLAN
------------------------------------------------------------------------------------------
--
 Aggregate  (cost=782.73..782.74 rows=1 width=8)
```

```
   ->  Hash Join  (cost=106.01..779.22 rows=1402 width=4)
      Hash Cond: (((naissance.code_departement)::text = (commune.code_departement)::text)
AND ((naissance.code_commune)::text = (commune.code_commune)::text))
       ->  Seq Scan on naissance  (cost=0.00..662.13 rows=1402 width=11)
             Filter: ((nb >= 100) AND (nb <= 500))
       ->  Hash  (cost=97.91..97.91 rows=540 width=7)
             ->  HashAggregate  (cost=92.51..97.91 rows=540 width=7)
                   Group Key: (commune.code_departement)::text,
(commune.code_commune)::text
                   ->  Bitmap Heap Scan on commune  (cost=18.49..88.51 rows=801 width=7)
                         Recheck Cond: (chef_lieu = 1)
                         ->  Bitmap Index Scan on commune_chef_lieu_idx
(cost=0.00..18.29 rows=801 width=0)
                               Index Cond: (chef_lieu = 1)
```

Une autre option serait d'utiliser seulement un indice sur la colonne "chef_lieu" de la table "commune". Cet indice est bien utilisé par le plan mais il ne change pas beaucoup le temps d'exécution.

Même dans le cas de la mise en place des deux indices, le temps final pour les requêtes n'est pas supérieur à celui où l'on appliquait seulement un filtre sur "nb". Les indices utilisent du space disque additionnel. Dans ce cas, l'index sur "chef_lieu" n'est pas vraiment utile, on va donc pas le mettre en place.

7. Conclusions

Avec les différentes observations que vous avez relevé au cours des questions posées, faîtes une conclusion en précisant ce que vous avez retenu de ces travaux pratiques.

Normalement, on pourrait s'attendre au fait que l'utilisation d'une jointure de type "INNER JOIN" sera meilleure en termes de temps d'exécution. C'est quand même quelque chose qui nous est arrivé dans mon travail. En utilisant les JOIN, le moteur de BDD sait exactement quel type de recherche nous voulons faire grâce au fait d'expliciter que l'on veut une jointure entre deux tables. Néanmoins, dans ce TP on a vu que le plan optimal dépend beaucoup aussi de la quantité d'information et de la structure et distribution des données.

En outre, l'utilisation d'indices peut améliorer de manière importante les temps d'exécution, mais il faut d'abord réfléchir aux colonnes qui seront utilisées pour les index. Pour cette analyse il est important de prendre en compte l'ensemble de requêtes utilisées par l'application et mettre en place les indices adéquats. Par exemple, dans ce TP (et dans mon environnement particulier) la mise en place de l'indice sur la FK dans "naissance" n'a pas changé les plans. Par contre, l'indice sur "nb" a beaucoup amélioré les recherches.

En ce qui concerne l'indice sur "chef_lieu", personnellement je m'attendais à que cet index baisse considérablement le temps d'exécution des requêtes. Par contre, cela n'a pas été le cas. La cause doit être que les valeurs possibles pour "chef_lieu" sont trop peu. Ceci fait qu'il y ait beaucoup de lignes par chaque valeur possible de cet attribut. On obtient comme résultat un indice qui ne fait qu'occuper de la place dans le disque mais qui n'optimise pas vraiment l'exécution.