ENSSAT LANNION

UNIVERSITÉ DE RENNES 1

Requêtes flexibles et stratégies d'exécution (4h de TP)

Travail à rendre

Il vous sera demandé de déposer sur l'ENT à la fin de la séance un rapport de TP. Ce rapport de TP doit décrire les expérimentations menées et les interprétations des résultats obtenus.

Préparation : Connexion au serveur PostgreSQL et accès aux données

Connectez-vous sur barn-e-01 avec ssh. La première fois, vous allez installer la base de données et y insérer toutes les données du TP. C'est une base appelée *SecondHandCars*. Voici les commandes pour commencer :

```
$ ssh votrelogin@barn-e-01
barn-e-01> application install postgresql
barn-e-01>
```

Il faut configurer le numéro de port du service. En effet, tout le monde va travailler sur la même machine, mais chacun(e) avec son serveur Postgres. Il va y avoir conflit de port. Donc, éditez le fichier **~/postgreSQL/data/postgresql.conf** et sur la ligne **#port=5432** , enlevez le **#** et changez le numéro en autre chose, à débattre entre vous (ex : numéro de table + 15432). Ensuite continuez :

```
barn-e-01> PG='-h localhost -p numérodeport'
barn-e-01> application start postgresql
barn-e-01> createdb $PG
barn-e-01> curl https://perso.univ-rennes1.fr/pierre.nerzic/BDDA/ (même ligne)
SecondHandCars.dump | psql $PG
barn-e-01>
```

Maintenant, la base est prête pour le TP. Voici comment on s'y connecte :

```
barn-e-01> psql $PG
votrelogin=#
```

Ce prompt est celui du SGBD qui attend des requêtes SQL. Par exemple pour obtenir la liste des tables, tapez ceci (**\d** est une commande de psql, comme **\h**) :

```
votrelogin=# \d
votrelogin=# \d secondhandcars
votrelogin=#
```

Essayez quelques requêtes triviales pour afficher quelques informations : le nombre d'annonces, le nombre d'annonces de moins de 1000€, le prix le plus grand (probablement une erreur de saisie).

IMPORTANT : à la fin de la séance de TP, vous devrez stopper votre SGBD.

Voici les commandes : d'abord quitter Postgresql, puis arrêter le service :

```
votrelogin=# \q
barn-e-01> application stop postgresql
barn-e-01> ^D
```

Il faudra relancer l'application au début de chaque séance, et penser à l'arrêter à la fin.

Partie 1 : Estimation de cardinalités à l'aide des métadonnées

Au cours de cette première partie du TP vous allez effectuer quelques expérimentations sur le planificateur et l'optimiseur de requêtes de *PostgreSQL*.

Gestion des index

Dans la table *SecondHandCars* l'attribut *prix* est associé à une structure d'indexation permettant au moteur d'exécution de PostgreSQL d'effectuer efficacement des opérations de sélection sur cet attribut. Les valeurs des autres attributs ne sont pas indexées.

Question 1 Écrire les requêtes permettant d'obtenir les informations suivantes :

1. le nombre de tuples dans la table *SecondHandCars*,
2. les valeurs distinctes de l'attribut *annee*,
3. le nombre d'années différentes, c'est à dire le nombre de valeurs de la question précédente (requête imbriquée).
4. le nombre de tuples pour chaque valeur d'*annee* (groupement), avec deux colonnes : année, nombre, triées par année. Cela vous permettra par la suite de comprendre les différentes stratégies d'indexation en fonction du type des données.

Dans votre compte-rendu, vous mettrez les requêtes et ce qu'elles affichent. Certains de ces résultats seront utilisés plus loin.

Question 2 L'instruction **`explain` *`requête`*** affiche le plan d'exécution d'une requête, c'est à dire la stratégie suivie par le SGBD pour évaluer la requête. On peut ajouter le mot clé **`analyze`** après **`explain`** pour obtenir des estimations sur le temps de calcul, c'est à dire : **`explain analyze` *`requête`***

Par exemple :

```
votrelogin=# explain select * from secondhandcars where puissance_ch < 200;
votrelogin=# explain select * from secondhandcars where puissance_ch >= 200;
votrelogin=#
```

Attention, les requêtes à expliquer ne sont pas celles de comptage, mais celles qui retournent des n-uplets (**`explain select * from`**… et non pas **`explain select count(*) from`**…).

D'autre part, cela n'est intéressant que s'il y a un index sur les colonnes concernées par la clause where. Ajoutez un index sur la colonne **`puissance_ch`**, puis relancez les deux requêtes précédentes.

Sans rentrer dans tous les détails techniques, vous allez observer deux stratégies d'exécution. Soit « Seq Scan », c'est à dire un parcours exhaustif des n-uplets, soit « Bitmap Heap Scan » qui est un parcours à deux niveaux. L'explication est à lire de bas en haut. Ça commence d'abord en bas par un « Bitmap Index Scan » qui consiste en une sélection des « pages » contenant les n-uplets susceptibles d'être sélectionnés par la condition « Index Cond » tout en bas. C'est à dire que les n-uplets sont groupés dans le stockage interne (disque dur) pour former des pages. Une page fait 8 Ko : **`SELECT current_settings('block_size');`** et contient donc un certain nombre de n-uplets. Certaines pages contiennent des n-uplets utiles, d'autres aucun. L'index contient un « bitmap » qui indique lesquelles sont pertinentes. C'est comme un tableau de booléens, un par valeur remarquable présente dans l'index, qui indiquent dans quelles pages on trouve cette valeur. Par exemple qu'on trouve des annonces pour des voitures de 80 chevaux dans telles et telles pages sur le disque. Vous comprenez que ce mécanisme occupe de la place mais il accélère considérablement certaines recherches.

Donc il y a d'abord cette sélection des pages, puis en remontant, il y a un « Recheck Cond » pour vérifier la condition individuellement sur les n-uplets des pages choisies.

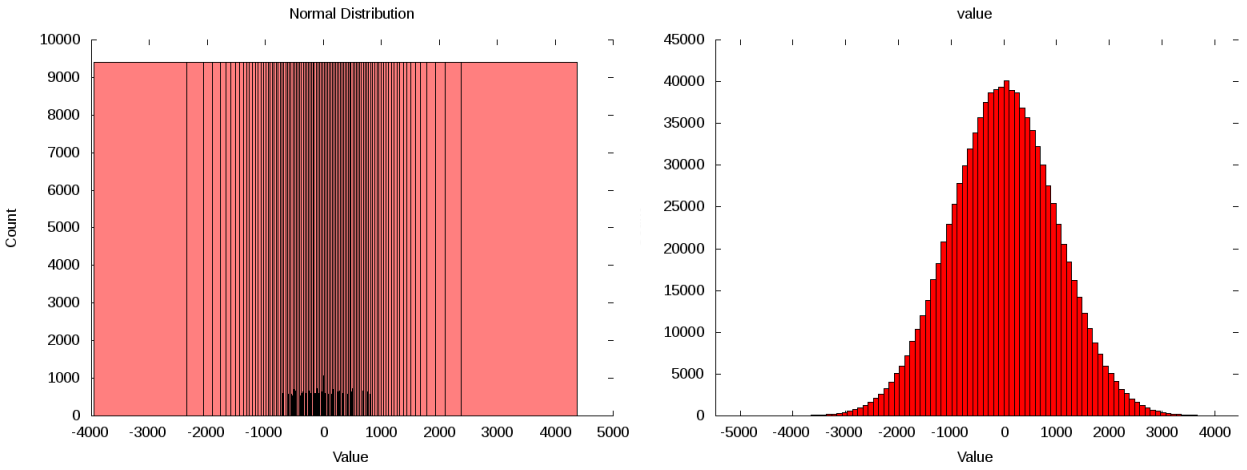
La bascule entre les deux stratégies, « Seq Scan » ou « Bitmap Scan » dépend du nombre attendu de n-uplets et leur répartition dans les pages. Il vaut mieux faire un parcours exhaustif si beaucoup de n-uplets satisfont la condition *where*, et utiliser le bitmap s'il y en a peu, répartis sur peu de pages différentes, et évidemment, s'il y a un bitmap pour les valeurs recherchées.

Sur chacune des requêtes suivantes, d'abord, demandez au SGBD combien il y aura de n-uplets, rapportez-le au nombre total de n-uplets : certaines requêtes récoltent une grande proportion des données, d'autres assez peu. On pourrait penser que le SGBD se base sur cela, mais son choix dépend de la difficulté à récupérer les n-uplets ou de l'absence d'information dans l'index. Puis analysez le plan d'exécution de ces requêtes et signalez quand la décision semble surprenante (ex : parcours séquentiel alors qu'il y a peu de n-uplets).

- Les voitures dont le prix est compris entre 3000 et 7500 euros.
- Les voitures dont le kilométrage est compris entre 50 000 et 65 000 km. Ensuite, ajoutez un index sur le kilométrage et redemandez l'explication.
- Les voitures dont le prix est strictement inférieur à 8900 euros.
- Les voitures dont le prix est inférieur ou égal à 8900 euros.

Métadonnées

Le planificateur de requêtes estime donc, sans accéder aux données, combien de tuples seront retournés par la requête. Pour cela, PostgreSQL maintient des statistiques sur chaque colonne. Ce sont des estimations sur la distribution unidimensionnelle sous la forme de trois information : un histogramme, la liste des valeurs les plus fréquentes, ainsi que la proportion de valeurs « NULL ».



L'histogramme de gauche est de type *equi-depth*, c'est-à-dire que le domaine est découpé en un certain nombre de barres (100 par défaut) qui représentent toutes le même nombre de tuples. Donc, en fonction de la distribution des données, les barres seront plus ou moins larges, plus ou moins serrées, contrairement à un histogramme *equi-height* comme à droite, où la largeur est identique et la hauteur représente la cardinalité. Dans un *equi-depth*, seule la largeur varie et la hauteur n'a pas d'importance. C'est ce qui est utilisé dans plusieurs SGBD.
Ces statistiques (histogramme equi-depth, valeurs fréquentes et proportion de NULL) sont rassemblées dans une table spéciale (méta-données) appelée `pg_stats`. Attention, ce sont des valeurs estimées, approximatives. Cependant le taux d'erreur est relativement faible, quelques %.

On commence par les valeurs les plus fréquentes. Utilisez la requête suivante pour récupérer le nombre de valeurs distinctes, la liste des valeurs fréquentes et leur fréquence estimée concernant l'attribut prix de la table secondhandcars :

```
SELECT n_distinct,
       array_to_string(most_common_vals, E'\n'),
       array_to_string(most_common_freqs, E'\n')
FROM pg_stats WHERE tablename='secondhandcars' AND attname='prix';
```

Ci-dessous, voici une variante pour la 3e colonne, pour afficher les cardinalités absolues estimées, remplacez ***NN*** par le nombre total de n-uplets de la table :

```
array_to_string(array(select round(NN*unnest(most_common_freqs))), E'\n')
```

Voici encore une variante, pour la première requête. Elle crée une sorte de table temporaire qui peut vous être utile pour la suite, contrairement aux deux requêtes précédentes qui affichent des textes :

```
SELECT * FROM (
  SELECT unnest(most_common_vals::text::int[]) AS val,
         unnest(most_common_freqs) AS freq
  FROM pg_stats WHERE tablename = 'secondhandcars' AND attname = 'prix'
) AS data
ORDER BY val;
```

Question 3 Interrogez la table *SecondHandCars* pour vérifier la précision de ces fréquences : écrivez d'abord quelques requêtes qui comptent effectivement certaines de ces valeurs sur les vraies données.

Vous pouvez ensuite comparer les estimations avec la valeur réelle, soit par la cardinalité relative, soit par la cardinalité absolue. Quelle est la précision relative $|card_{est}\text{-}card_{réelle}|/card_{réelle}$?

NB : vous n'obtiendrez pas toutes et tous les mêmes précisions, car cela dépend du fonctionnement interne. Les estimations peuvent être refaites en tapant la commande SQL **`ANALYZE;`**

Pour vous faciliter la comparaison entre vraies cardinalités et estimations, voici une requête qui affiche l'estimation relative pour un prix de 7500€ :

```
SELECT most_common_freqs[array_position(most_common_vals::text::int[], 7500)]
FROM pg_stats WHERE tablename='secondhandcars' AND attname='prix';
```

Explications : *most_common_vals* et *most_common_freqs* sont deux tableaux parallèles (de 100 cases). Le premier contient les valeurs fréquentes, le second contient les cardinalités estimées. Le principe est de chercher la position de la valeur fréquente qui nous intéresse puis de récupérer sa cardinalité.

Vérifiez la requête suivante qui calcule la somme des cardinalités de toutes les valeurs fréquentes de l'attribut prix. Notez son résultat.

```
SELECT (SELECT SUM(mcf) FROM unnest(most_common_freqs) AS mcf)
FROM pg_stats WHERE tablename='secondhandcars' AND attname='prix';
```

Voilà pour les valeurs isolées et les conditions comme *where colonne = valeur*. Ce n'est pas tout. Pour les inégalités et les intervalles (*between*), il faut aussi prendre en compte la répartition des tuples décrite dans l'histogramme. Utilisez la requête suivante pour connaître les bornes de l'histogramme construit sur l'attribut prix. C'est une liste de valeurs, ce sont les [min, max=min du suivant[… des barres de l'histogramme *equidepth*, comme des piquets dans une clôture autour d'un champ.

```
SELECT array_to_string(histogram_bounds, E'\n')
FROM pg_stats WHERE tablename='secondhandcars' AND attname='prix';
```

Notez qu'on ne peut pas essayer de vérifier les cardinalités de ces barres parce qu'on ne connaît pas la cardinalité totale de l'histogramme. On sait seulement que chaque barre représente environ 1/N de

cette cardinalité totale, avec N = nombre de barres. Ce sont les deux questions suivantes qui vont la préciser.

Déjà, voici comment compter les barres. Le -1, c'est parce qu'il y a une barre de moins que les bornes :

```
SELECT (SELECT COUNT(*) FROM unnest(histogram_bounds::text::int[]))-1
FROM pg_stats WHERE tablename='secondhandcars' AND attname='prix';
```

Enfin, il faut aussi connaître le nombre de valeurs « NULL » dans la colonne prix :

```
SELECT null_frac
FROM pg_stats WHERE tablename='secondhandcars' AND attname='prix';
```

Question 4 Comparez cette valeur à la véritable proportion de valeurs NULL.

Question 5 À partir de ces statistiques, vous allez estimer le nombre de tuples de la table dont le prix est inférieur ou égal à 2000. Pour cela il faut calculer la sélectivité [0,1] de l'intervalle [0, 2000].

Le travail consiste à :

1. Déterminer la cardinalité relative de chaque intervalle de l'histogramme. Il faut cumuler la cardinalité de toutes les valeurs fréquentes, et y ajouter la proportion de valeurs NULL. Ainsi, la cardinalité totale des barres = 1.0 – somme des cardinalités relatives des valeurs fréquentes – proportion de valeurs NULL. Il reste à diviser cette cardinalité totale des barres par le nombre de barres pour avoir la cardinalité relative de chaque barre.
2. Compter manuellement les barres qui sont intégralement incluses dans [0, 2000]. Indiquez les bornes dans le rapport, ex : 1, 200, 450, etc.
3. Pour les deux intervalles du début et de la fin, éventuellement partiellement inclus dans [0, 2000], il faut utiliser une hypothèse de distribution uniforme des données au sein de chaque barre, et donc compter au *pro-rata*, c'est à dire la proportion de la barre qui est couverte par l'intervalle [0, 2000].
4. Ajouter les cardinalités de toutes les valeurs fréquentes incluses dans l'intervalle [0, 2000] (notez-les dans le rapport, avec leur cardinalité). Au lieu de les chercher à la main, essayez d'adapter une précédente requête.
5. Comparez le résultat à la cardinalité réelle (taux d'erreur).

Mettez toutes ces informations un peu disparates en forme lisible dans le rapport.

Partie 2 : Requêtes flexibles

Vous allez maintenant écrire vos premières requêtes flexibles. Vous allez considérer que vous cherchez la voiture la moins chère possible, ayant le moins de kilomètres possible et la plus récente possible. Vous allez définir les fonctions caractéristiques des ensembles flous correspondant à ces trois notions en utilisant des fonctions écrites en pl/pgsql. Voici un exemple :

```
CREATE OR REPLACE FUNCTION faible(km integer) RETURNS float AS
$$
BEGIN
    IF (km >= 15000)
    THEN
        RETURN 0;
    ELSE
        RETURN (15000 - CAST(km AS FLOAT))/15000;
    END IF;
END;
$$
LANGUAGE plpgsql;
```

que vous pouvez utiliser sur des n-uplets ou isolément, comme ceci :

```
SELECT km, faible(km) AS mu FROM secondhandcars WHERE faible(km) > 0.9;

SELECT faible(8000);
```

Question 6 Définissez trois fonctions permettant de récupérer respectivement les voitures pas chères, les voitures ayant un kilométrage modéré et les voitures récentes. À vous de choisir les définitions de ces termes flous, mais pour le prix, ne dépassez pas 8000€. La dernière année de la base est 2011. Cette dernière colonne étant représentée par une chaîne, il faut sans doute définir une fonction d'appartenance discrète, comme ceci :

```
CREATE OR REPLACE FUNCTION recente(annee varchar) RETURNS float AS
$$
BEGIN
    RETURN CASE
        WHEN annee='2011' THEN 1.0
        WHEN annee='2010' THEN ...
        ...
        ELSE 0.0
    END;
END;
$$
LANGUAGE plpgsql;
```

Question 7 Analysez le plan d'exécution d'une requête retournant les voitures de faible kilométrage puis d'une seconde requête retournant les voitures pas chères, en comparant avec une requête non floue. Qu'en pensez-vous ?

Question 8 Analysez le plan d'exécution d'une requête utilisant une conjonction des trois conditions de sélection floue. Pour cette conjonction floue, utilisez la t-norme « minimum ». Attention au piège, en SQL, le minimum de valeurs d'attributs d'un même n-uplet est obtenu par LEAST.

Vous verrez peut-être des réponses ayant un prix valant NULL. Le problème est ouvert : doit-on accepter des valeurs NULL pour des colonnes soumises à des tests ? Ici, est-ce qu'un prix ou un kilométrage valant NULL peuvent être considérés comme tout à fait satisfaisants, peu importe le critère ? Dans la négative, il vaudrait mieux que les fonctions retournent 0 face à des NULL. Reprenez donc toutes les fonctions et transformez les conditionnelles IF en CASE avec un cas pour NULL.

Requêtes flexibles dérivées

Pour rappel, la dérivation d'une requête flexible consiste à construire la requête « booléenne » retournant l'ensemble des tuples pouvant satisfaire la requête flexible, ceux du support, puis à calculer le degré de satisfaction pour ces tuples uniquement.

Question 9 Intégrer dans les deux requêtes flexibles définies lors de la question 7 une étape de dérivation et analysez leurs plans d'exécution. Y-a-t-il une amélioration ?

Question 10 Faites de même pour la requête conjonctive de la question 8. Constatez également qu'il n'est pas forcément besoin de dériver chacune des conditions floues quand certaines d'entre elles sont très sélectives. D'autre part, la dérivation d'une condition basée sur une appartenance discrète peut se révéler difficile à écrire. Peut-on en déduire une stratégie pour dériver une requête floue ?

Partie 3 : Une approche qualitative de gestion des préférences : « Skyline »

En considérant un ensemble de préférences définies sous la forme de fonctions croissantes (e.g. maximiser la puissance moteur, l'année, etc.) ou décroissantes (e.g. minimiser la consommation, le

prix, etc.), l'approche « Skyline » consiste à déterminer l'ensemble des tuples non dominés par un autre tuple selon les préférences exprimées.

Question 11 Rappelez la définition d'une relation de dominance entre deux tuples.

Question 12 Vous allez implémenter l'algorithme de construction du skyline sur les tuples de la table *SecondHandCars* en cherchant à minimiser km et prix, et à maximiser l'année. Vous avez le choix du langage, Java, JavaScript ou Python. Les données se trouvent au format CSV :

https://perso.univ-rennes1.fr/pierre.nerzic/BDDA/SecondHandCars.csv

pour les charger dans votre programme (utilisez une bibliothèque pour lire le fichier, ex : import csv en Python).

Bilan

Sur la base de vos expérimentations et observations, dressez un bilan comparatif des deux approches de gestion des préférences que vous avez étudiées (floue vs. Skyline).