TP 4 - Stockage

Le but de ce TP est d'observer en pratique les notions liées au stockage abordées en cours et en TD. Pour cela et comme dans les TPs précédents, nous allons utiliser une petite table de travail `test(id,a,b)` permettant de stocker des paires d'entiers sans aucune contrainte. L'attribut `id`, de type serial, sera la clé primaire de la table.

Cette table est à utiliser dans **tous** les exercices.

▸ Exercice 1 : Champs de gestion

Nous avons vu en cours que le format (physique) des enregistrements contient des informations qui ne figurent pas dans le schéma (logique) de la table correspondante. Pour Postgres, ces champs supplémentaires sont récapitulées dans la documentation :

`https://www.postgresql.org/docs/9.6/ddl-system-columns.html`

Dans cet exercice, nous allons nous concentrer sur trois champs : `ctid`, `xmin` et `xmax`, et suivre leur évolution au cours de différentes opérations.

1. Expliquez la signification des champs `ctid`, `xmin` et `xmax`.

2. Insérez quelques valeurs dans la table `test` et observez l'état des champs supplémentaires à l'aide de la requête :

```
SELECT *, ctid, xmin, xmax FROM test;
```

3. Observez l'évolution de ces valeurs lors du scénario ci-dessous. Que pouvez-vous en déduire sur l'effet des insertions sur la valeur de `xmin` ?
   (Note : utilisez `SELECT txid_current();` *pour connaître l'id de la transaction courante.)*

Transaction 1	Transaction 2
`BEGIN;`	`BEGIN;`
`INSERT INTO test(a,b) VALUES (1,1);` `INSERT INTO test(a,b) VALUES (2,2);`	
	`INSERT INTO test(a,b) VALUES (3,3);` `INSERT INTO test(a,b) VALUES (4,4);`
`COMMIT;`	`COMMIT;`

4. De même, expliquez l'effet des suppressions sur les valeurs de `xmax` lors du scénario ci-dessous.

Transaction 1	Transaction 2
`BEGIN;`	`BEGIN;`
`DELETE FROM test WHERE a = 1;`	
	`DELETE FROM test WHERE a = 2;`
`ROLLBACK;`	`COMMIT;`

5. Finalement, expliquez ce qu'il se passe lors des mises à jour. Faites bien attention à ce que voit chaque transaction et aux valeurs de la colonne `ctid`.

Transaction 1	Transaction 2
`BEGIN;`	`BEGIN;`
`UPDATE test SET b = 10 WHERE a = 1;`	
	`INSERT INTO test(a,b) VALUES (5,5);`
`COMMIT;`	`COMMIT;`

▸ Exercice 2 : Tailles de fichiers et de blocs

1. Insérez quelques valeurs dans la table `test` puis regardez le champs `ctid` de chaque ligne de la table. Combien de blocs sont utilisés pour stocker la table ?

2. Insérez un grand nombre de valeurs dans la table `test` en choisissant à chaque fois une valeur aléatoire entre 0 et 10 pour `a` et entre 0 et 100 pour `b`.
   Vous pouvez par exemple utiliser la requête ci-dessous :

```
INSERT INTO test(a,b)
  SELECT round(random()*10), round(random()*100)
  FROM generate_series (1,10000);
```

3. Pouvez-vous en déduire combien de blocs sont utilisés pour stocker la table et combien d'enregistrements chaque bloc contient en moyenne ?

4. En supposant que tous les champs de gestion (de l'exercice 1) sont des entiers 32 bits, pouvez-vous en déduire un ordre de grandeur de la taille d'un bloc ?

   Faites une recherche pour confirmer votre estimation.

▸ Exercice 3 : Réorganisation différée

Postgres fournit dans la vue `pg_class` diverses informations sur l'état du système. Nous allons nous en servir pour surveiller le niveau de remplissage de la table `test`.

Utilisez par exemple la requête ci-dessous pour afficher le nombre de blocs et le nombre d'enregistrements présents dans la table :

```
SELECT relpages, reltuples FROM pg_class WHERE relname = 'test';
```

Attention, les informations sur la table ne sont pas systématiquement mises à jour. Utilisez la clause **ANALYZE** pour demander explicitement une mise à jour.

```
ANALYZE test;
-- demande de mise à jour de pg_class pour la table test.
```

1. Vérifiez que les informations de `pg_class` correspondent bien à celles que vous aviez observées dans l'exercice 2.
2. Supprimez tous les enregistrements de la table dont la valeur de `a` est inférieure à 5, et observez l'évolution de `pg_class`. Expliquez le résultat observé.
3. Insérez dans la table 4000 lignes pour lesquelles `a` vaut 99. Observez l'évolution de `pg_class`. Dans quels blocs ont été insérés les nouveaux enregistrements ?
4. Faites une mise à jour de la table pour incrémenter la valeur de `b` quand `a = 99`. Expliquez l'évolution du nombre de blocs de la table.

Les observations précédentes laissent penser que Postgres utilise une stratégie de **réorganisation différée** pour gérer l'espace disponible dans les blocs. Observons le comportement de Postgres lorsque des blocs entiers sont supprimés.

Utilisez la requête ci-dessous vour vider la table et réinitialiser le `serial`.

```
TRUNCATE test RESTART IDENTITY;
```

5. Vérifiez cette fois-ci que tous les blocs ont bien été supprimés.
6. À partir de la table vide, insérez à nouveau 10000 enregistrements, puis supprimez les 5000 **premières** lignes de la table. Observez et expliquez.
7. À partir de la table vide, insérez à nouveau 10000 enregistrements, puis supprimez les 5000 **dernières** lignes de la table. Observez et expliquez.

Nous allons à présent demander une réorganisation explicite de la table.

8. Rejouez les deux questions précédentes en utilisant la commande ci-dessous après chaque suppression, observez et expliquez.

```
VACUUM VERBOSE test;
-- demande de nettoyage de test avec détail des opérations de maintenance.
```

9. Finalement, testez la clause **VACUUM FULL** dans les scénarios des questions 2 à 4, observez les résultats et expliquez.

   Mettez en évidence vos hypothèses sur une petite table contenant peu de lignes.