Bases de Données Avancées : TP7

Ceci est le dernier TP, et il est un peu … dense ! : / Vous allez y retrouver des commandes pour insérer, supprimer et modifier des records, ainsi que pour sélectionner des tuples (le fameux SQL SELECT, restreint à une seule table).

A. Information : commandes à gérer pour ce TP. A LIRE ATTENTIVEMENT !!!

A1. Insertion d'un tuple dans une table

Le format de cette commande est le suivant :

INSERT INTO nomRelation VALUES (val1,val2, … ,valn)

Cette commande demande l'insertion, dans la relation *nomRelation*, d'un record / tuple dont les valeurs sont *val1*,*val2*, … ,*val*.

Précisions de format :

- Il y a un espace entre les mots INSERT et INTO, un espace entre le mot INTO et le nom de la relation, un espace entre le nom de la relation et le mot VALUES, un espace entre le mot VALUES et la parenthèse ouverte.
- Le nom de la relation ne contient pas d'espaces, et il est censé correspondre à une relation existante
- Il n'y a pas d'espace avant / après les virgules, ni avant la parenthèse fermée.
- Les valeurs type chaîne de caractères sont encadrées entre guillemets, il n'y a pas d'espace entre les guillemets.
- On suppose que les valeurs de type chaîne de caractères ne contiendront pas de caractère autre que lettres minuscules et majuscules et chiffres.
- Les valeurs insérées sont censées être compatibles avec les types des colonnes de la relation, et leur nombre correspondre au nombre de colonnes de la relation.

Exemple :

INSERT INTO Profs VALUES (”Ileana”,”BDDA”,13)

A2. Rajout « en bloc » d'un ensemble de tuples dans une table, à partir d'un fichier .csv

Le format de cette commande est le suivant :

APPEND INTO nomRelation ALLRECORDS (nomFichier.csv)

Cette commande demande le rajout dans la relation *nomRelation* de tous les records / tuples dont les valeurs apparaissent dans le fichier *nomFichier.csv*,

Le fichier .csv décrit un record sur chaque ligne, avec les valeurs du record séparées par le caractère **virgule**. Par exemple, un csv contenant des tuples à insérer dans la relation **Profs** (comme dans l'exemple ci-dessus) pourrait contenir les deux lignes suivantes:

”Ileana”,”BDDA”,13
”Soto”,”BDD”,1

Les valeurs de type chaîne de caractères respecteront les mêmes contraintes que pour l'insertion normale.

Les valeurs correspondant aux colonnes de type FLOAT sont écrites avec un point décimal, de la même manière que nous écrivons les float dans le code.
Notez que si jamais une telle valeur est entière, on peut l'écrire sans point décimal (pour la valeur 3 on peut avoir 3 ou alors 3.0).
Vous pouvez supposer que les fichiers csv seront toujours importés dans des relations dont le schéma (nombre et type des colonnes) est compatible avec le contenu du fichier !

Vous pouvez consulter le contenu d'un tel fichier, S.csv, sur Moodle.
Ce fichier est « importable » dans une relation à 5 colonnes dont les types sont respectivement INT, FLOAT, INT, INT, INT.

Attention : on suppose que le fichier .csv se trouve à la racine de votre dossier projet. Dans la commande il n'y a pas de chemin pour ce fichier !

Exemple :

APPEND INTO NombresMysterieux ALLRECORDS (S.csv)

A3. Commande SELECT style SQL, quoique simplifiée ;)

Cette commande doit sélectionner tous les records d'une relation qui respectent une conjonction de plusieurs conditions (y inclus aucune condition:)), puis en afficher certaines valeurs.
Notez qu'il s'agit d'une simplification du SELECT standard SQL notamment sur deux points :

- on opère sur une seule relation
- on considère uniquement une *conjonction* de conditions, sans se pencher sur la disjonction ou négation.

Le format de l'affichage du résultat sera :
1 record par ligne,

puis à la fin, sur une nouvelle ligne,
la phrase : ***Total selected records=x*** , où *x* est le nombre de records affichés.

Pour afficher un record, on affiche les valeurs demandées par la commande séparées par des « ; » (espace, point virgule, espace) et on rajoute un point à la fin.
Attention : veillez à respecter à la lettre le format d'affichage demandé.

Le format de la commande est le suivant :
(attention, tout doit être sur la même ligne !!! la gestion des retours à la ligne est optionnelle, la commande est présentée en mode « segmenté sur plusieurs lignes » ci-dessous simplement pour la lisibilite!!!)

SELECT aliasRel.col_{p1}, aliasRel.col_{p2},..., aliasRel.col_{pm}
FROM nomRelation aliasRel
[WHERE C_1 AND C_2 ... AND C_n]

avec chaque **col_{pi}** désignant un nom de colonne valide de la relation **nomRelation**

ou alternativement (suivant toujours la syntaxe SQL) :

SELECT *
FROM nomRelation aliasRel
[WHERE C_1 AND C_2 ... AND C_n]

pour signaler qu'il faut afficher l'ensemble des colonnes de la relation.

Dans la partie **WHERE,** *qui peut ne pas être présente !!!*, chaque C_i est une condition dont le format est :

Terme1*OP*Terme2

où:

- OP est un des opérateurs suivants : =,<,>,<=,>=,<>
- **Au maximum un parmi les termes Terme1 et Terme2 est une valeur (constante)**
- **L'autre (ou les autres) terme(s) sont de la forme aliasRel.nomColonne**

Précisions de format :

- Il y a un espace après SELECT
- Il n'y a pas d'espace avant / après les virgules
- Il y a un espace avant et après FROM
- Il y a un espace avant et après aliasRel
- Il y a un espace après WHERE
- Les critères sont séparés par le mot clé **AND** (et il y a un espace avant et après ce mot clé)
- Il n'y a pas d'espace avant / après OP
- Le même nom de colonne peut figurer dans plusieurs critères !

Exemples :

```
SELECT t.nom,t.prenom
FROM ToutesLesNotes t
WHERE t.Cours="IF3BDDA" AND 8<=t.NoteCT AND t.NoteProjet>13
AND t.NoteProjet>=t.NoteCT

SELECT *
FROM ToutesLesNotes x
```

A4. Commande DELETE style SQL, quoique simplifiée ;)

Cette commande vise la suppression, dans une relation donnée, de tous les tuples qui respectent des conditions spécifiés par le mot clé **WHERE**. La manière de spécifier la partie **WHERE** est strictement identique à celle de la commande **SELECT** décrite ci-dessus.

Le format de la commande **DELETE** est le suivant (attention, tout doit être sur la même ligne!):

DELETE nomRelation aliasRel
[WHERE C_1 AND C_2 ... AND C_n]

La commande affichera comme résultat, sur une seule ligne,
la phrase : ***Total deleted records=x*** , où x est le nombre de records supprimés.

Exemple :

```
DELETE ToutesLesNotes t
WHERE t.StatutCours= "Annulé"
```

A5. Commande UPDATE style SQL, quoique simplifiée ;)

Cette commande vise la mise à jour de certains champs (valeurs) dans une relation donnée, pour les tuples qui respectent des condition spécifiés par le mot clé **WHERE**. La manière de spécifier la partie **WHERE** est strictement identique à celle de la commande **SELECT** décrite ci-dessus.

Le format de la commande **UPDATE** est le suivant (attention, tout doit être sur la même ligne!):

UPDATE nomRelation aliasRel SET aliasRel.col$_{p1}$=val$_1$,..., aliasRel.col$_{pm}$ =val$_m$ [WHERE C$_1$ AND C$_2$ ... AND C$_n$]

La commande affichera comme résultat, sur une seule ligne,
la phrase : ***Total updated records=x*** , où x est le nombre de records modifiés.

Exemple :
UPDATE ToutesLesNotes t SET t.NoteProjet=20
WHERE t.Cours="IF3BDDA"

B.Code : Implémentation des commandes

Sans surprise, il faudra coder (et tester!) la gestion des commandes décrites ci-dessus. Pour cela, vous allez procéder comme lors du TP précédent, en créant des méthodes **ProcessCommand** sur la classe **SGBD**. Vous allez également utiliser le code du TP5.

Ci-dessous quelques suggestions d'implémentation.

B1. La classe Condition

Pour gérer les conditions présentes dans une commande de type **SELECT**, nous vous conseillons vivement de créer une classe **Condition**.

Un terme non-constante, tel que décrit dans la présentation de la commande **SELECT** ci-dessus, pourra être simplement représenté en tant qu'indice de colonne. Pour la valeur / constante éventuelle, vous pouvez utiliser le même type que celui que vous avez employé pour les valeurs dans la classe **Record**.

Attention toutefois : si vous utilisez des chaînes de caractères, il faudra pouvoir avoir accès au type des colonnes concernées au moment de l'évaluation de la condition !!!!
En effet, 10>3 (si on a des entiers)
Mais '10 '< '3' si on a des chaînes de caractères (car ordre lexicographique).
Il est donc essentiel de savoir le type exact des valeurs !!!

B2. L'interface IRecordIterator

Un grand nombre de briques logiques dans les requêtes peuvent être implémentées de manière efficace en utilisant le concept des *itérateurs*. Cela permet d'évaluer les requêtes « en flux tendu », d'être plus économique en termes de mémoire, et est aussi une manière classique d'« articuler » entre eux plusieurs opérateurs relationnels.

Pour gérer tout ceci de manière uniforme, vous allez créer une interface **IRecordIterator**, qui sert à parcourir « petit à petit » un ensemble de tuples, et qui contient les méthodes suivantes :

- une méthode
  record **GetNextRecord(),** avec *record* un **Record**.
  qui retourne le record courant et « avance d'un cran le curseur de l'itérateur ».
  La méthode retournera *null* lorsqu'il ne reste plus de record dans l'ensemble de tuples.
- une méthode
  void **Close()**
  qui signale qu'on n'utilise plus cet itérateur.
- une méthode
  void **Reset**()
  qui met / remet le curseur au début de l'ensemble des records à parcourir.

Vous allez pouvoir, si vous le souhaitez, décliner cette interface en plein d'implémentations différentes ! x)

B3. Les classes SelectOperator et ProjectOperator

La commande **SELECT** combine en réalité deux opérateurs d'algèbre relationnelle : la sélection et la projection. Pour gérer ces opérateurs de manière plus générale, en permettant également d'implémenter de manière efficace la commande **SELECT**, vous pouvez créer deux classes, **SelectOperator** et **ProjectOperator**.

Ces deux classes :

- Implémenteront elles-mêmes l'interface **IRecordIterator**
- Tiendront également une référence vers un autre **IRecordIterator** (qui sera « l'opérateur fils »).

Pour l'opérateur de projection, son opérateur fils sera justement un opérateur de sélection ; lorsqu'on appelle **GetNextRecord** sur l'opérateur de projection, cela fera un appel à **GetNextRecord** sur l'opérateur de sélection, puis on enlève les valeurs superflues (correspondant aux colonnes qu'on ne retient pas) !

Pour l'opérateur de sélection, son opérateur fils sera un « scanner » de relation, c'est à dire un itérateur qui parcourt la relation. Plus d'informations concernant ce scanner ci-dessous.

B4. La classe RelationScanner

Pour énumérer les tuples d'une relation (et donc faire le fameux « scan séquentiel »x)), on peut créer une classe **RelationScanner**, qui implémente elle aussi l'interface **IRecordIterator.**

Une manière simple / rapide d'implémenter cette classe (sachant que cette manière n'est absolument pas optimale en usage mémoire !!!) est d'appeler, dans le constructeur de **RelationScanner**, la méthode **GetAllRecords** de la classe **Relation**. Une fois cette liste de tuples récupérée, la méthode **GetNextRecord** ne fera qu'avancer un indice dans la liste.

Le gros désavantage de cette manière « simple et rapide » de faire c'est qu'on va devoir garder tous les tuples d'une relation en mémoire, ce qui n'est absolument pas réaliste ; par ailleurs, ça casse le paradigme central du SGBD qui est celui que l'essentiel de la mémoire consommée par le SGBD doir être gérée par le BufferManager.

Accessoirement, certains scénarios d'évaluation contiendront une quantité importante de données, et il sera pas possible de garder ces données en mémoire simultanément.

Pour être efficace (et avoir tous les points lors de l'éval), il est donc souhaitable de faire un vrai itérateur.

Cet itérateur gardera en mémoire qu'un tuple à la fois, et il « avancera petit à petit dans les pages de données et dans le contenu de chaque telle page », en faisant l'« extraction » du tuple courant.
Challenge supplémentaire : on peut réaliser cet itérateur parcimonieux en mémoire *en n'utilisant qu'un seul buffer dans le BufferManager*!!!

B5. La classe RelationScannerWithSelect

On peut envisager de « fusionner » le **SelectOperator** et le **RelationScanner** pour créer un scanner (qui implémente, encore et toujours, **IRecordIterator**) qui filtre « en direct » les tuples suivant la liste des conditions qu'on lui fournit.
Ce filtrage peut d'ailleurs être « redescendu » au niveau de la lecture des valeurs du Heap File, pour écarter « le plus tôt possible » les tuples qui ne respectent pas les conditions.

Si vous implémentez un tel scanner, qui correspond de manière formelle à un *chemin d'accès (« access path »)* de la relation, alors il n'y a plus besoin de **SelectOperator**, et vous pouvez utiliser le **RelationScannerWithSelect** directement dans le **ProjectOperator**.

B6. La classe RecordPrinter

Puisque, pour la commande **SELECT**, on vous demande d'afficher les tuples, il vous sera utile de créer une classe **RecordPrinter** qui prend en argument de son constructeur un **IRecordIterator** et qui affiche petit à petit les tuples fournis par cet itérateur.

B7. Un petit scénario de test

Vous trouverez ci-après un scénario de test de votre application. N'hésitez pas, bien sûr, à créer vos propres scénarios de test !
Et n'oubliez pas que les scénarios de test de l'application ne remplacent pas les tests à faire au niveau de chaque classe !

CREATE TABLE Pomme (C1:INT,C2:VARCHAR(3),C3:INT)
INSERT INTO Pomme VALUES (1,"aab",2)
INSERT INTO Pomme VALUES (2,"ab",2)
INSERT INTO Pomme VALUES (1,"agh",1)
SELECT * FROM Pomme p
résultat attendu :
1 ; aab ; 2
2 ; ab ; 2
1 ; agh ; 1
Total selected records = 3
SELECT pp.C2 FROM Pomme pp WHERE pp.C1=1
résultat attendu :
aab
agh
Total selected records = 2
SELECT pote.C1,pote.C1 FROM Pomme pote WHERE pote.C3=1
résultat attendu :
1 ; 1
Total selected records = 1
DELETE Pomme c WHERE c.C1=1 AND c.C3=2
résultat attendu :
Total deleted records = 1
UPDATE Pomme p SET p.C1=3 WHERE p.C2<"ac"
résultat attendu :
Total updated records = 1
CREATE TABLE S (C1:INT,C2:REAL,C3:INT,C4:INT,C5:INT)
APPEND INTO S ALLRECORDS(S.csv)
SELECT * FROM S s WHERE s.C3=12
(résultat attendu : 10 tuples)