TP 2021 Fil Rouge « Je code mon SGBD »

4ASTI – INSA CVL – B. Nguyen & P. Clemente

Objectifs généraux du projet

L'objectif du projet est en mettre en application vos connaissances sur les mécanismes internes des SGBD par l'implémentation d'un certain nombre d'opérateurs et de techniques présentées dans le cours, en particulier l'approche « pipeline ».

Les requêtes données en exemple sont en langage SQL. Vos programmes manipulent des opérateurs de l'algèbre relationnelle. Comme nous n'allons pas implémenter de compilateur, ce sera à vous de trouver l'expression algébrique correspondant à ces requêtes.

Compréhension du code existant (fait en cours)

- Ecrivez un programme qui crée une table *table3*, stockée sur disque à l'endroit que vous voulez, qui possède 3 attributs, et 4 lignes avec les valeurs suivantes

ATTRIBUT0	**ATTRIBUT1**	**ATTRIBUT2**
0	1	1
1	1	2
2	2	1
3	2	2

- Ecrivez un programme main qui calcule la requête suivante :

SELECT *

FROM table3

WHERE T.0 = 2

- Ecrivez un programme main qui calcule la requête suivante :

SELECT *

FROM table3

WHERE T.1 = 1 AND T.2 = 2

Travail à faire en TD

Une progression minimale est suggérée, certains d'entre vous peuvent aller plus loin s'ils le souhaitent.

TD 1 : Opérateurs simples et indexation

Opérateur de projection

(*) Codez l'opérateur de projection. Testez le sur la table3 en écrivant le programme correspondant à la requête suivante :

SELECT table3.1

FROM table3

Note : on ne se préoccupe pas de conserver les numéros de colonne initiaux. Par exemple, si on projette un n-uplet avec 7 attributs sur l'ensemble d'attributs {0, 3, 5} alors on aura ensuite un n-uplet avec 3 attributs, les attributs {0, 1, 2}, sachant que 0 <-> 0, 1 <-> 3 et 2 <-> 5. On pourrait aussi réordonner les attributs par exemple en projetant sur {0, 5, 3}.

Opération d'update

(**) Codez l'opération de mise à jour `public void update(byte att1, Object Value, byte att2, Object newValue);`

Cette opération sélectionne les lignes telles que att1 == Value, puis remplace pour ces lignes la valeur de att2 par newValue.

Dans un premier temps vous pourrez ignore les opérations de commit et de rollback.

(***) : implémentez l'insert, le delete et l'update en utilisant un index sous forme de table de hachage, qu'on pourra construire sur n'importe quel attribut de la table. Chaque entrée de la table de hachage contient les positions dans le fichier sur le disque des enregistrements ayant la valeur correspondante pour l'attribut indexé. On pourra également implémenter un opérateur de type Table qui correspondrait à un INDEXSCAN (c'est un dire une récupération de données sur le disque, couplée à une restriction). En comparaison, l'opérateur TableInt déjà implémenté est un FULLSCAN.

*(optionnel ****) : Implémentez l'insert, le delete et l'update en utilisant un arbre B+ qui sera construit sur l'attribut 0. On stockera simplement dans l'arbre B+ la position de l'enregistrement dans le fichier. Il est suggéré de stocker l'arbre B+ au choix dans un fichier annexe, plutôt que dans le même fichier.*

*(optionnel **) : Implémentez un opérateur de scan sur index (arbre B+ ou table de hachage) de la table qui pourra remplacer le FullScan.*

TD 2 : Opérateurs bloquants

Operateur DISTINCT

(**) Codez un opérateur de projection SELECT DISTINCT (ProjectionDistinct) qui implemente l'interface Projection. Vous pouvez coder cet opérateur en mode « pipeline » mais la classe

ProjectionDistinct devra certainement se rappeler d'un certain nombre d'informations. Alternativement, tout en respectant l'interface, vous pouvez coder l'opérateur de mode « bloquant » c'est-à-dire que vous allez calculer l'ensemble du résultat lors de l'opération, le stocker dans une table temporaire (ou toute autre structure de votre choix) en mémoire (ou sur le disque), puis relire cette table lors du next()

Opérateur ORDER BY

(**) Codez un opérateur ORDERBY qui implémente Operateur, et qui ordonne par ordre croissant un flux selon un attibut (passé en paramètre lors de la construction). Cet opérateur doit forcément être implémenté en mode bloquant. On demande dans cette question de stocker le résultat temporaire en mémoire et non sur le disque.

(***) Rajoutez les opérations de commit de rollback à votre opération d'update (et d'insert/delete si vous les avez implémentés)

*(optionnel **) Implémentez les fonctions d'agrégation SUM, COUNT(*) et MAX. Cet opérateur doit prendre en entrée une table avec 1 seule colonne, et retourner une table avec 1 seule colonne et 1 seule ligne.*

*(optionnel ***) Implémentez le GROUP BY (c'est un opérateur bloquant), et illustrez son fonctionnement en utilisant AVG.*

*(optionnel **) Implémentez le HAVING, et illustrez son fonctionnement.*

TD3 : Jointure et optimisation

Opérateurs de jointure

(*) Codez la jointure par double boucle imbriquée (JointureDBI). Cet opérateur peut être implémenté en pipeline.

(***) Codez au choix la jointure par table de hachage ou la jointure « sort merge join ». Ces opérateurs nécessitent un préprocessing (construction de la table de hachage ou bien tri des tables) avant de pouvoir être exécutées en pipeline.

*(optionnel ****) Codez la jointure de hachage hybride (sur disque).*

Pour aller plus loin …

*(optionnel ***) Codez un optimiseur capable de prendre un Operateur (composé de plusieurs opérations), et de le réorganiser en appliquant une heuristique simple : pousser les restrictions et projections en bas, et monter les jointures.*

(optionnel) Simulez la distribution en effectuant un partitionnement horizontal de votre base (i.e. en divisant une table en 2 tables ayant le même nombre de colonnes). (**) Implémentez le fullscan, puis (***) la jointure utilisant un filtre de Bloom.

*(optionnel ****) Implémentez un système de verrouillage à 2 phases respectant le niveau d'isolation SERIALIZABLE. Illustrez son fonctionnement sur plusieurs scenarii, dont un deadlock pour illustrer le problème (il n'est pas demandé de détecter le deadlock).*

*(optionnel ****) Implémentez un journal permettant une reprise sur panne.*