M1 Informatique

Bases de données avancées

Examen de session 1 (durée 2h)

16 mai 2019

Documents autorisés : dix feuilles A4 recto-verso manuscrites ou imprimées.

Indexation

Question 1 Ci-dessous est donné le psuedo-code (vu en cours) de l'algorithme de recherche d'une clef c dans un arbre B^+ sans doublons. On considère qu'à chaque itération $K_1, ..K_n$ dénotent les clefs stockées dans le noeud courant l'arbre, dans l'ordre. Pour simplifier le code on suppose aussi l'existence de clefs fictives $K_0 = -\infty$, $K_{n+1} = +\infty$ sur chaque noeud. De cette facon l'enfant i d'un noeud de l'arbre se trouve toujours entre la clef K_{i-1} et la clef K_i.

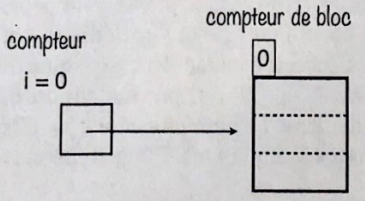
```
B := le noeud racine;
while le noeud courant B n'est pas une feuille do
    trouver l'enfant i de B tel que K_{i-1} =< c < K_i;
    B := i-eme enfant de B;
end
chercher c dans B
```

a. Proposer un algorithme (en adaptant le pseudo-code ci-dessus) pour chercher toutes les clefs entre c_1 et c_2 ;

b. Supposer maintenant que les clefs de recherche sont de la forme $< a, b >$. Proposer un algorithme pour chercher, donné c, toutes les clefs $< a, b >$ telles que $a = c$.

Question 2 Nous considérons ici un index hash sur une clef de type entier. Faire l'hypothèse (irréaliste) qu'un bloc du disque puisse stocker jusqu'à trois entrées de l'index (i.e. trois couples <clef, pointeur à enregistrement>). La fonction de hachage est $h(x) = x \bmod 7$. Supposer que les clefs suivantes sont insérées dans l'index hash initialement vide, dans l'ordre : 2, 3, 5, 7, 11, 17, 19, 23, 29, 31.

Montrer l'état de l'index hash après chaque insertion. En particulier montrer, après chaque insertion : les blocs utilisés par l'index, leur contenu, la structure des pointeurs et la valeur de tous les compteurs (compteur global et compteurs de bloc). Se rappeler que dans un index hash vide il y a un seul pointeur à un bloc du disque vide, et tous les compteurs ont valeur 0, comme le montre la figure ci-dessous.



Évaluation de requêtes

Question 3 Ecrire un algorithme (pseudo-code) qui implémente un itérateur pour le *index join* entre `R` et `S` sur l'attribut `A`. On supposera l'index disponible sur `S.A`. Le code doit définir les fonctions standard `open()`, `next()` et `close()` de l'itérateur, ainsi que spécifier quel est son état interne. Se rappeler que `open()` initialise et `close()` finalise l'état interne de l'itérateur, alors que `next()` renvoie le prochain tuple du résultat de la jointure.

On supposera qu'un *linear scan* de `R` ainsi qu'un *index scan* de `S` sont disponibles en pipeline. C'est-à-dire, on supposera que `R.open()` et `R.close()` s'occupent d'ouvrir et fermer le parcours séquentiel de `R`, et `R.next()` renvoie le prochain tuple de `R`. De façon similaire `S.open()` et `S.close()` ouvrent et ferment l'itérateur sur `S`, alors que `S.next(c)` retourne le prochain tuple de `S` dont l'attribut `A` vaut `c`; on dispose également de `S.rewind()` pour remettre l'itérateur de `S` au début. Toutes les opérations de `next()` renvoient `null` s'il ne reste plus de tuples à retourner. (Notez qu'il n'est pas demandé d'écrire ces fonctions pour `R` et `S`.)

Formes normales

Question 4 Soit $R(A, B, C, D, E, F, G)$ un schéma de relation avec dépendances fonctionnelles :

$\mathcal{F} = \{$
$A \to FD$
$G \to B$
$GAF \to C$
$E \to D$
$AB \to C$
$FE \to A$
$CB \to G$
$FEB \to G$
$GFE \to C\}$

- Donner toutes les clefs de R. E(B/G) (A/F)
- Trouver une décomposition de R en *forme normale de Boyce-Codd* sans perte d'information. Est-ce que cette décomposition préserve toutes les dépendances fonctionnelles ?
- Trouver une décomposition de R en *troisième forme normale* sans perte d'information et sans perte de dépendances fonctionnelles. (Se rappeler que il est nécessaire d'abord de minimiser $\mathcal{F}$, c'est-à dire d'abord minimiser toutes les parties gauches, et ensuite éliminer toutes le dépendances fonctionnelles redondantes.)

Transactions

Question 5 Soit les deux transactions suivantes :

```
T1:
read(A);
read(B);
if A = 0 then B:= B+1;
write(B).

T2:
read(B);
read(A);
if B = 0 then A:=A+1;
write(A).
```

Supposer que la contrainte d'intégrité $A = 0 \vee B = 0$ doit être respectée par la base de données, et que initialement $A = B = 0$.

- Est-ce que chaque transaction exécutée en isolation préserve la cohérence de la base de données ?
- Y a-t-il une exécution concurrente de T1 et T2 qui produit un ordonnancement non-sérialisable ? Si non, expliquer pourquoi. Si oui, montrer l'ordonnancement et expliquer pourquoi il est non-sérialisable.
- Y a-t-il une exécution concurrente de T1 et T2 qui produit un ordonnancement sérialisable, mais non sériel ? Si non, expliquer pourquoi. Si oui, montrer l'ordonnancement et dire pourquoi il est sérialisable.
- Ajouter des opérations de *lock* et *unlock* sur T1 et T2 de telle sorte qu'elles soient conformes au 2PL (verrouillage à deux phases).
  - Y a-t-il un ordonnancement de T1 et T2 compatible avec les verrous, où T2 commence après T1, mais avant que T1 effectue l'opération d'écriture ? Si oui montrer l'ordonnancement, si non dire pourquoi.
  - Y a-t-il une exécution de T1 et T2 compatible avec les verrous qui génère un deadlock ? Si oui montrer l'ordonnancement jusqu'au deadlock. Si non, expliquer pourquoi.