ABD - TP 08

Auteurs

- Anne-Sophie Saint-Omer
- Thomas Bernard

Exercice 1

1. Toutes les modifications sont propagées à la base de données avant l'écriture de Commit sur le fichier log (et donc, avant le commit).

Faux. Principe de Write-Ahead Logging, c'est à dire que les modifications doivent être d'abord ajoutées au fichier de log, avant d'être réellement effectuées sur la base de données.

2. Toutes les modifications sont propagées à la base de données après l'écriture de Commit sur le fichier log (et donc, après le commit).

Vrai. Principe de Write-Ahead Logging, c'est à dire que les modifications doivent être d'abord ajoutées au fichier de log, avant d'être réellement effectuées sur la base de données.

3. Toutes les modifications peuvent être propagées à la base de données soit avant soit après l'écriture de Commit sur le fichier log (et donc, soit avant soit après le commit).

Faux. Principe de Write-Ahead Logging, et d'après les questions précédentes, les modifications ne peuvent être propagées à la base de données qu'une fois le commit écrit dans le fichier de log.

4. Les modifications de données sont propagées à la base de données avant de stocker de façon permanente le valeur précèdent des données dans le fichier log.

Faux. En cas de problème, on peut récupérer les anciennes données dans le fichier log. Si il y a une erreur et qu'elles ne sont pas enregistrées, les données sont définitivement perdues et donc irrécupérables. On converse donc l'état stable de la base de données avant toute modification.

5. Les modifications de données sont propagées à la base de données après le stockage permanent du valeur précèdent des données dans le fichier log.

Vrai. En cas de problème, on peut récupérer les anciennes données dans le fichier log. Si il y a une erreur et qu'elles ne sont pas enregistrées, les données sont définitivement perdues et donc irrécupérables. On converse donc l'état stable de la base de données avant toute modification.

6. Les modifications de données sont propagées à la base de données soit avant soit après le stockage permanent du valeur précèdent des données dans le fichier log.

Faux. En cas de problème, on peut récupérer les anciennes données dans le fichier log. Si il y a une erreur et qu'elles ne sont pas enregistrées, les données sont définitivement perdues et donc irrécupérables. On converse donc l'état stable de la base de données avant toute modification.

Exercice 2

Algorithme undo-redo pour la gestion des échecs de transactions, dans le cas de :

1. Ecriture

- Log de la base de données dans un état stable avant l'écriture (pour UNDO)
- Écriture du fichier (mémoire volatile)
- Log de la base de données dans un état stable après l'écriture (pour REDO)

2. Commit

- Validation de la transaction
- Ècriture effective des données (mémoire persistente)

3. Abort

- Annule la transaction dans son ensemble
- Efface les modifications temporaires et replace la base de données dans un état stable

4. Restart

- Recherche dans journal du dernier point de reprise
- Traitement du journal du point de reprise vers la panne pour connaître les transactions validées et les transactions non validées
- Traitement du journal en arrière pour défaire les transactions non validées (UNDO)
- Traitement du journal en avant pour refaire les transactions validées (REDO)

Exercice 3

Fichier log de transactions gérées par la technique de undo-redo.
Les enregistrements du fichier log ont le format (W, Tid, Variable, Old, New).

```
1.  (BEGIN T1)
2.  (W, T1, A, 25, 50)
3.  (W, T1, B, 25, 250)
4.  (BEGIN T2)
5.  (W, T1, A, 50, 75)
6.  (W, T2, C, 25, 55)
7.  (COMMIT T1)
8.  (BEGIN T3)
9.  (W, T3, E, 25, 65)
10. (W, T2, D, 25, 35)
11. (CKP T2,T3)
12. (W, T2, C, 55, 45)
13. (COMMIT T2)
14. (BEGIN T4)
15. (W, T4, F, 25, 120)
16. (COMMIT T3)
17. (W, T4, F, 120, 150)
18. (COMMIT T4)
```

1. Restart à cause d'un malfonctionnement avant d'écrire l'enregistrement 10 dans le disque.

```
9.  (W, T3, E, 25, 65)
======== CRASH ========
10. (W, T2, D, 25, 35)
```

Variable A : 75
Variable B : 250
Variable C : 25
Variable D : -
Variable E : -
Variable F : -

2. Restart à cause d'un malfonctionnement avant d'écrire l'enregistrement 13 dans le disque.

```
12. (W, T2, C, 55, 45)
======== CRASH ========
13. (COMMIT T2)
```

Variable A : 75
Variable B : 250
Variable C : 25
Variable D : -
Variable E : -
Variable F : -

3. Restart à cause d'un malfonctionnement avant d'écrire l'enregistrement 14 dans le disque.

```
13. (COMMIT T2)
======== CRASH ========
14. (BEGIN T4)
```

Variable A : 75
Variable B : 250
Variable C : 45
Variable D : 35
Variable E : 25
Variable F : -

4. Restart à cause d'un malfonctionnement avant d'écrire l'enregistrement 18 dans le disque.

```
17. (W, T4, F, 120, 150)
======== CRASH ========
18. (COMMIT T4)
```

Variable A : 75
Variable B : 250
Variable C : 45

Variable D : 35
Variable E : 65
Variable F : -

5. Restart à cause d'un malfonctionnement après avoir écrit l'enregistrement 18 dans le disque.

```
18. (COMMIT T4)
======== CRASH ========
```

Variable A : 75
Variable B : 250
Variable C : 45
Variable D : 35
Variable E : 65
Variable F : 150

Exercice 4

Journaux sérialisable : Journaux dont les transactions ont le même output et les mêmes effets sur le base de données stable.

Un graphe de précédence aussi appelé graphe de conflits ou encore graphe sérialisé est utilisé dans le contrôle des bases de données.
Ce graphe contient un noeud pour chaque transaction commitée et un arc entre deux transactions Ti et Tj si une action de la transaction Ti entre en conflit avec une action de la transaction Tj.

Un conflit équivalent correspond à l'invocation d'un même ensemble d'actions sur un même ensemble de transactions.
A schedule is said to be conflict-serializable when the schedule is conflict-equivalent to one or more serial schedules.
Un journal est dit conflit sérialisable quand le journal est conflit équivalent pour plusieurs journaux.
Une autre définition est de dire que le journal est conflit sérializable si et seulement si le graphe de précédence est acyclique c'est à dire si le graphe est défini pour inclure des transactions non commitée puis que les cycles invoqués ses transactions sans violation de conflits sérialisés.

On suppose que les commits des transactions ont tous lieu à la fin du schedule.

```
S1 = {r1 [A] ,w1[B] , r2 [B] ,w2[C] , r3 [C] ,w3[A]}
```

Conflits :
- Transaction 2 : La lecture de B a lieu après l'écriture de B dans la transaction 1
- Transaction 3 : La lecture de C a lieu après l'écriture de C dans la transaction 2

```
S2 = {r1 [A] , r2 [A] ,w1[B] ,w2[B] , r1 [B] , r2 [B] ,w2[C] ,w1[D]}
```

Conflits :
- Transaction 2 : L'écriture de B a lieu après l'écriture de B dans la transaction 1
- Transaction 1 : La lecture de B a lieu après l'écriture de B dans la transaction 2

Suite à ces conflits, T1 et T2 travaillent chacun de leur côté avec une valeur de B potentiellement différente.
La valeur de B qui sera conservée dans la base sera celle de la dernière transaction à commiter ses modifications.
L'autre transaction aura donc travaillé avec des valeurs erronées.