Systèmes de gestion de bases de données

Algorithmes de concurrence

P. Rigaux

Cnam, dépt. informatique

May 21, 2015

Verrouillage à deux phases

Un des deux principaux algorithmes pour assurer la sérialisabilité.

Est réputé engendrer beaucoup de blocages et de rejets.

Repose sur la détection des **conflits**.

Définition (à connaître)

Deux opérations $p_i[x]$ et $q_j[y]$ sont **en conflit** si $x = y$, $i \neq j$, p ou q est une écriture.

En clair: deux transactions accèdent au même tuple, et (au moins) une veut le modifier.

Exemple

Reprenons une nouvelle fois l'exemple des mises à jour perdues:

$$r_1(s)r_1(c_1)r_2(s)r_2(c_2)w_2(s)w_2(c_2)w_1(s)w_1(c_1)$$

Les conflits sont les suivants :

- $r_1(s)$ et $w_2(s)$ sont en conflit ;
- $w_2(s)$ et $w_1(s)$ sont en conflit.

$r_1(s)$ et $r_2(s)$ **ne sont pas** en conflit, puisque ce sont deux lectures.

Il n'y a pas de conflit sur c_1 et c_2.

Qui des conflits ici?

$$r_1(c_1)r_1(c_2)r_2(s)r_2(c_2)w_2(s)w_2(c_2)r_1(s)$$

L'ordre sur les transactions

Le verrouillage à deux phases ordonne les transactions en fonction des conflits.

Définition (à connaître)

H exécution concurrente des transactions $T = \{T_1, T_2, \cdots, T_n\}$.
Il existe une relation $\lhd$ sur cet ensemble, définie par :

$$T_i \lhd T_j \Leftrightarrow \exists p \in T_i, q \in T_j, p \textit{ est en conflit avec } q \textit{ et } p <_H q$$

où $p <_H q$ indique que p apparaît avant q dans H.

Dans l'exemple précédent: on a $T_1 \lhd T_2$, ainsi que $T_2 \lhd T_1$.

Condition de sérialisabilité

La condition sur la sérialisabilité s'exprime sur le graphe de la relation $(T, \lhd)$, dit **graphe de sérialisation**.

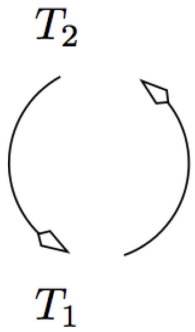
Condition de sérialisabilité

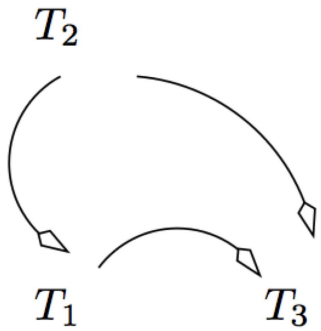
Soit H une exécution concurrente d'un ensemble de transactions T. Alors H est sérialisable si et seulement si le graphe de $(T, \lhd)$ est acyclique.

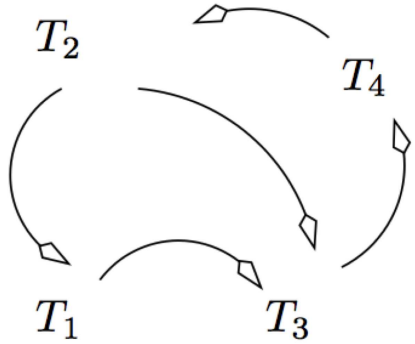
Autrement dit, $(T, \lhd)$ est une relation d'ordre partiel (antisymétrie).

Graphes de sérialisabilité

Lesquels correspondent à une exécution sérialisable?







Bilan: le verrouillage à deux phases doit vérifier qu'aucun cycle ne peut intervenir dans ce graphe.

L'algorithme

Simple: on pose des verrous, et on les utilise pour empêcher l'apparition de cycles.

- Pour chaque **lecture**, sur x, on regarde s'il y a un verrou **exclusif** sur x ;
  - si oui la transaction T_i est mise en attente.
  - si non, la transaction pose un verrou en lecture et l'opération est exécutée.
- Pour chaque **écriture** sur x, on regarde s'il y a un verrou **quelconque** (exclusif ou partagé) sur x ;
  - si oui la transaction T_i est mise en attente.
  - si non, la transaction pose un verrou en écriture et l'opération est exécutée.

Important: ne fonctionne que si les verrous sont relâchés au `commit` ou au `rollback`.

Exemple

Prenons les deux transactions $T_1 : r_1[x]w_1[y]C_1$ et $T_2 : w_2[x]w_2[y]C_2$

Et l'exécution concurrente : $r_1[x]w_2[x]w_2[y]C_2w_1[y]C_1$

- T_1 pose un verrou partagé sur x, lit x mais ne relâche pas le verrou ;
- T_2 tente de poser un verrou exclusif sur x : impossible puisque T_1 détient un verrou partagé, *donc T_2 est mise en attente* ;
- T_1 pose un verrou exclusif sur y, modifie y, et valide ; tous les verrous détenus par T_1 sont relâchés ;
- T_2 est libérée : elle pose un verrou exclusif sur x, et le modifie ;
- T_2 pose un verrou exclusif sur y, et modifie y ;
- T_2 valide, ce qui relâche les verrous sur x et y.

Exécution après réordonnancement: $r_1[x]w_1[y]w_2[x]w_2[y]$

Un gros problème : les deadlock

Reprenons notre exemple des mises à jour perdues.

$$r_1(s)r_1(c_1)r_2(s)r_2(c_2)w_2(s)w_2(c_2)C_2w_1(s)w_1(c_1)C_1$$

- P_1 lit s et c_1, qui sont verrouillés en lecture.
- P_2 lit s et c_2, qui sont verrouillés en lecture.
  s **partage deux verrous en lecture**
- P_2 veut écrire s : conflit, donc arrêt de P_2.
- P_1 veut écrire s : conflit, donc arrêt de P_1.

C'est **l'étreinte fatale** (*deadlock*) !! Le système va rejeter une des transactions.

Très regrettable, mais encore mieux que d'introduire des anomalies (?)

Souvenons-nous des versions

Un système qui fournit un niveau `repeatable read` doit s'appuyer sur un système de versions estampillées.

Dans ce cas les lectures se font sur l'état de la base **avant** le début de la transaction.

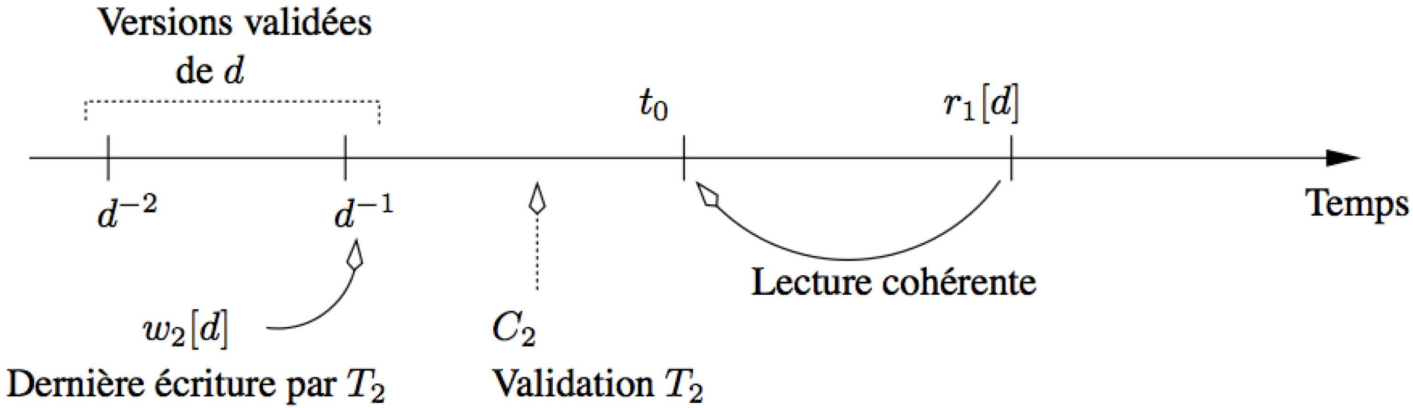
Deux possibilités de conflit entre une lecture de T_1 et T_2.

- $r_1[d]$ est en conflit avec une écriture $w_2[d]$ qui a eu lieu **avant** t_0 ;
- $r_1[d]$ est en conflit avec une écriture $w_2[d]$ qui a eu lieu **après** t_0.

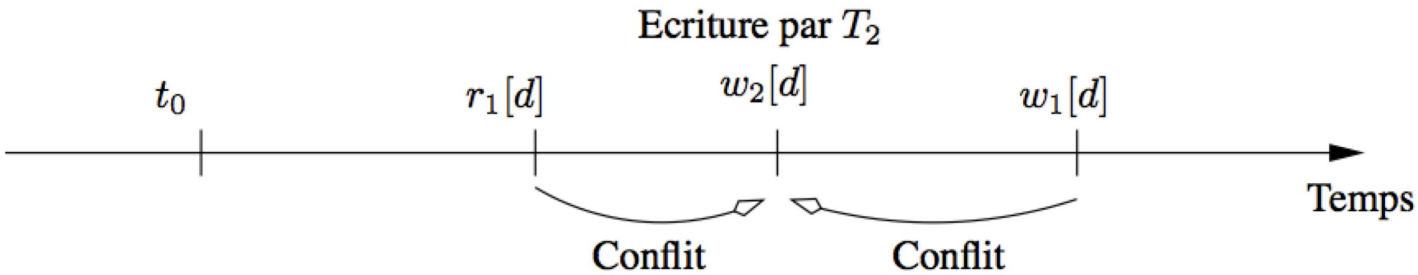
Approfondissons.

Les deux cas de conflit

Cas 1: $r_1[d]$ en conflit avec $w_2[d]$ **avant** t_0. Alors T_2 a validé. Pas de risque.



Cas 2: $r_1[d]$ en conflit avec $w_2[d]$ **après** t_0.



Si T_1 cherche à écrire d après l'écriture $w_2[d]$, un conflit cyclique apparaît

Le contrôle de concurrence multi-versions

Le principe

On vérifie, au moment d'exécuter $w_1[a]$, qu'aucune transaction T_2 n'a modifié a entre le début de T_1 et l'instant présent.

La méthode:

- $r_1[a]$ lit la première version de a avec $e_a \leq e_{T_1}$; pas de verrou;
- en cas d'écriture $w_1[a]$,
  - si $e_a \leq e_{T_1}$ et a non verrouillé : T_1 verrouille (exclusif) a, et effectue $w_1[a]$;
  - si $e_a \leq e_{T_1}$ et a verrouillé : T est mise en attente ;
  - si $e_a > e_T$, T est rejetée.
- au moment de C_1 tous les tuples modifiés par T_1 obtiennent une nouvelle version avec pour estampille l'instant du `commit`.

Exemple

Nos maj perdues: $r_1(s)r_1(c_1)r_2(s)r_2(c_2)w_2(s)w_2(c_2)C_2w_1(s)w_1(c_1)C_1$
On suppose que $e_{T_1} = 100$, $e_{T_2} = 120$.

- T_1 lit s, T_1 lit c_1, T_2 lit s, T_2 lit c_2, sans verrouiller ; **aucun verrou**.
- T_2 veut modifier s : l'estampille de s est inférieure à $e_{T_2} = 120$: s n'a pas été modifié; on pose un verrou exclusif sur s et on effectue $w_2[s]$:
- T_2 modifie c_2, avec pose d'un verrou exclusif ;
- T_2 valide et relâche les verrous ; deux nouvelles versions de s et c_2 sont créées avec l'estampille 150 ;
- T_1 veut à son tour modifier s, mais cette fois le contrôleur détecte qu'il existe une version de s avec $e_s > e_{T_1}$. T_1 est rejetée.

Bilan

Deux algorithmes principaux:

- **Verrouillage à deux phases.**. Préventif (”pessimiste”); pose beaucoup de verrous, entraîne des interblocages.
- **Multi-versions**. A posteriori (”optimiste”); pose peu de verrou, entraîne des rejets.

Le second est réputé plus fluide (prouvé?); utilisé dans d’autres contextes (applications distribuées / Web).