SGBD-R : Les transactions

Contrôle de concurrence

Guillaume Raschia — Nantes Université

originaux de Philippe Rigaux, CNAM

Dernière mise-à-jour : 30 mars 2022

Plan de la session

Isolation par versionnement (S10.1)

Sérialisabilité (S10.2)

Contrôle de concurrence multi-versions (S10.3)

2PL (S10.4)

Isolation par versionnement (S10.1)

Versionnement

Toute transaction T en cours a deux choix à chaque instant :

- valider les maj effectuées avec `commit`
- les annuler avec `rollback`.

Le SGBD maintient, pendant l'exécution de T, deux versions des nuplets mis à jour :

- une version du nuplet après la mise à jour;
- une version du nuplet avant la mise à jour.

On parle d'image après et d'image avant pour ces deux versions.

Isolation fondée sur les versions

Soit deux transactions T et T'.

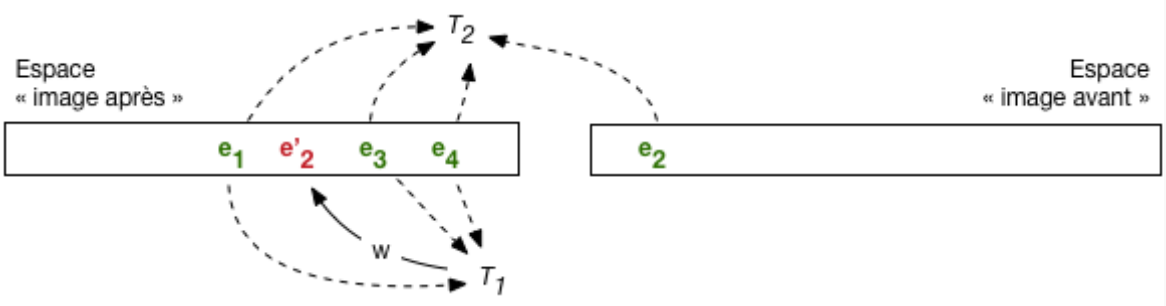
- Chaque fois que T met à jour un nuplet, la version courante est copiée dans l'image avant, puis remplacée par la valeur de l'image après fournie par T.
- Quand T lit des nuplets, le système doit lire dans l'image après pour assurer une vision cohérente de la base
- En revanche, une autre transaction T' doit lire dans l'image avant pour éviter les effets de lectures sales.

✎ Combien d'images avant / après ? Une seule mise à jour est autorisée à la fois [1], donc une seule paire (image avant, image après).

1. pas d'écriture sale.

Illustration

Le nuplet e_2 est en cours de modification.



T_1 lit e'_2 après sa maj, et T_2 lit e_2 : elles ne lisent pas la même version.

L'image après peut être interprétée comme l'état courant de la base, éventuellement non approuvé.

Les lectures ne sont pas répétables

Le mode précédent est correct pour le niveau `read committed`.

Mais les lectures ne sont pas répétables : une fois que T_1 est approuvée, T_2 lit la dernière version validée de e_2.

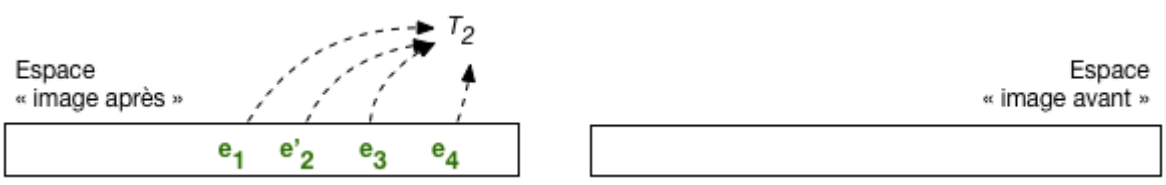


Image avant et `repeatable read`

Les images avant constituent un « cliché » de la base pris à un moment donné.

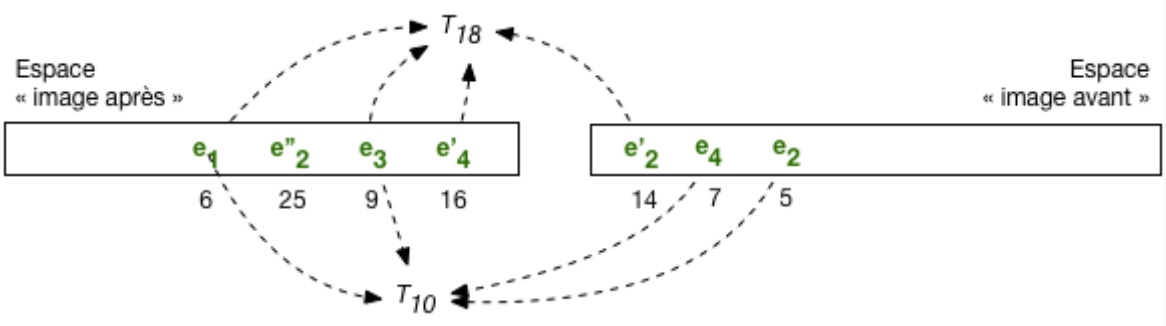
Principe des lectures répétables : toute transaction ne lit que dans le cliché valide au moment où elle a débuté.

Horodatage :

- Toute transaction T_i est « estampillée » par le moment τ_i où elle a commencé.
- Chaque nuplet e est marqué par le moment τ_e de sa validation (`commit`).
- T_i ne lit que les nuplets dont l'estampille $\tau_e < \tau_i$ est immédiatement antérieure à la sienne.

Ce mécanisme de versionnement assure la cohérence de toutes les lectures.

Le niveau `repeatable read`



Combien de versions ? Il faut maintenir le « cliché » de la base pris pour la transaction la plus ancienne en cours d'exécution.

À retenir

Notion d'image avant et image après

- Image après : les dernières versions de chaque nuplet ; certaines ne sont pas encore validées.
- Image avant : toutes les versions antérieures

Le multi-versions assure les niveaux `read committed` et `repeatable read`.

- `read committed` : il suffit de conserver une version dans l'image avant
- `repeatable read` : il faut conserver les versions immédiatement antérieures au début de la transaction (« cliché » = *snapshot*)

Niveau d'isolation plus élevé = performances affectées

Sérialisabilité (S10.2)

Une définition

Sérialisabilité (à connaître et—surtout—comprendre)

Une exécution concurrente H de n transactions $T_1, \cdots, T_n$ est sérialisable si et seulement si le résultat de l'exécution de H est le même que celui d'une exécution en série de $T_1, \cdots, T_n$.

Bien comprendre : si j'ai deux transactions T_1 et T_2, leur exécution imbriquée est sérialisable ssi équivalente à T_1 puis T_2, ou à T_2 puis T_1 (en série).

En d'autres termes, on compare H à une exécution en isolation complète.

Caractérisation de la sérialisabilité : les conflits

La définition précédente est déclarative : elle dit ce qu'est la sérialisabilité, pas la manière de la vérifier en pratique.

Nous avons besoin d'une caractérisation plus opérationnelle.

Opérations conflictuelles (à connaître)

Deux opérations $p_i(x)$ et $q_j(y)$ sont en conflit ssi $x = y$, $i \neq j$, p ou q est une écriture.

En clair : deux transactions distinctes accèdent au même nuplet, et—au moins—une veut le modifier.

Exemple

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

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

- $r_1(s)$ et $w_2(s)$ sont en conflit ;
- $r_2(s)$ et $w_1(s)$ 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 (lectures) ; pas de conflit sur c_1 et c_2 ; pas de conflit entre $r_1(s)$ et $w_1(s)$.

✎ Quid des conflits ici ?

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

Ordonnancement des transactions

Relation $\lhd$ de préséance (à connaître)

Soit H une exécution concurrente des transactions $\mathcal{T} = \{T_1, T_2, \cdots, T_n\}$;

Il existe une relation $\lhd$ sur cet ensemble, définie par :

$$T_i \lhd T_j \Leftrightarrow \exists p_i \in T_i, q_j \in T_j, p_i \text{ est en conflit avec } q_j \text{ et } p_i <_H q_j$$

où $p_i <_H q_j$ indique que p_i apparaît avant q_j dans H.

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

Condition de sérialisabilité

Une définition opérationelle de la sérialisabilité s'exprime à partir du graphe de la relation $(\mathcal{T}, \lhd)$, dit graphe de sérialisation.

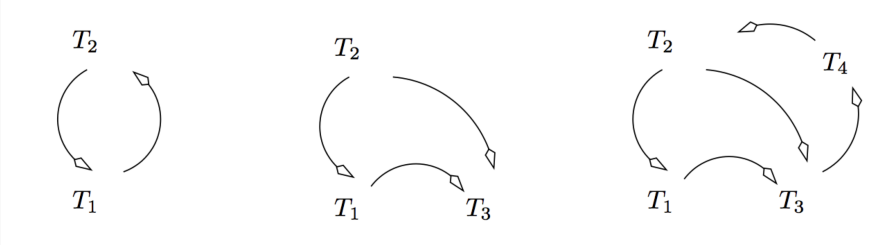
Sérialisabilité par conflit : une condition suffisante de sérialisabilité

Soit H une exécution concurrente d'un ensemble de transactions $\mathcal{T}$. Si le graphe de $(\mathcal{T}, \lhd)$ est acyclique, alors H est sérialisable.

Autrement dit, la sérialisabilité par conflit impose que $(\mathcal{T}, \lhd)$ soit une relation d'ordre partiel (antisymétrie).

Graphes de sérialisabilité

✎ Lesquels correspondent à une exécution sérialisable ?



Important : un algorithme de contrôle de concurrence doit s'assurer qu'aucun cycle ne puisse apparaître dans ce graphe.

✎ Trouver une exécution à la fois sérialisable et non sérialisable par conflit !

À retenir

Caractérisation « sémantique » de la sérialisabilité : équivalence avec une exécution en série, pour au moins un ordonnancement des transactions impliquées.

Caractérisation « syntaxique » de la sérialisabilité : pas de cycle dans le graphe de sérialisation construit sur les conflits entre opérations.

Objectif d'un algorithme de contrôle s'assurer que toutes les exécutions sont sérialisables.

- en surveillant le graphe de sérialisation et en rejetant une transaction si un cycle apparaît (variante « optimiste »)
- en tentant de prévenir l'apparition de cycles (variante « pessimiste »)

Contrôle de concurrence multi-versions (S10.3)

Un algorithme de contrôle de concurrence multi-versions

Connu sous le nom de *Snapshot Isolation*. Très utilisé.

Tire parti du versionnement, qui limite les opportunités de conflits.

Très léger : aucun verrouillage en lecture, un contrôle sur les écritures.

Mais : ne garantit pas la sérialisabilité stricte…

Souvenons-nous des versions

Un système qui fournit un niveau `repeatable read` s'appuie sur du versionnement par horodatage.

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

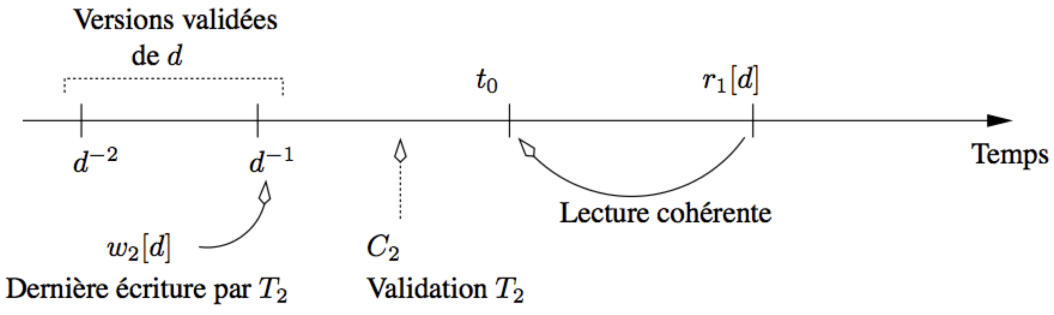
Deux possibilités de conflit entre une lecture de T_1 et une tx T_2.

- $r_1(d)$ est en conflit avec une écriture $w_2(d)$ qui a eu lieu avant τ_1 ;
- $r_1(d)$ est en conflit avec une écriture $w_2(d)$ qui a eu lieu après τ_1.

Approfondissons.

Premier cas de conflit

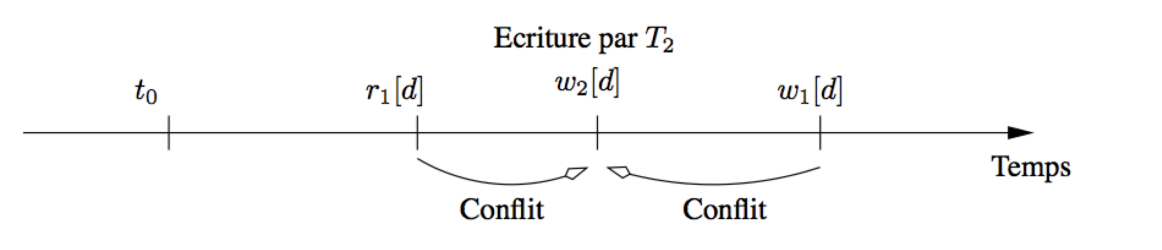
Cas 1 : $r_1(d)$ en conflit avec $w_2(d)$ avant τ_1. Alors T_2 est approuvée. Pas de risque.



En fait, cela traduit une exécution sérielle de T_2 puis T_1.

Second cas de conflit

Cas 2 : $r_1(d)$ en conflit avec $w_2(d)$ après τ_1.



Si T_1 cherche à écrire d après l'écriture $w_2(d)$, un conflit cyclique apparaît.
L'algorithme doit surveiller ce cas.

Le contrôle de concurrence multi-versions

Le principe

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

En cas d'écriture $w_1(e)$ [2],

- si $\tau_e \leq \tau_1$ et e non verrouillé : T_1 verrouille (exclusif) e, et effectue $w_1(e)$;
- si $\tau_e \leq \tau_1$ et e verrouillé : T_1 est mise en attente ;
- si $\tau_e > \tau_1$, T_1 est rejetée.

Estampillage. Au moment de C_1 tous les nuplets modifiés par T_1 obtiennent une nouvelle version avec pour estampille l'instant du `commit`.

2. Règle du **First-Updater-Wins** (alt. : **First-Commiter-Wins**).

Exemple

Maj perdues : $r_1(s); r_1(c_1); r_2(s); r_2(c_2); w_2(s); w_2(c_2); C_2; w_1(s); w_1(c_1); C_1$.

On prend $\tau_1 = 100$, $\tau_2 = 120$.

- T_1 lit s, T_1 lit c_1, T_2 lit s, T_2 lit c_2 ; aucun verrou.
- T_2 veut modifier s : l'estampille de s est inférieure à $\tau_2 = 120$: 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, relâche les verrous ; versions de s et c_2 avec l'estampille 150 ;
- T_1 veut à son tour modifier s : version de s avec $\tau_s > \tau_1 = 100$. T_1 est rejetée.

Écriture biaisée : un cas de non-sérialisabilité

L'algorithme *Snapshot Isolation* dans certaines exécutions non sérialisables.

Exemple : une simple copie d'une ligne à une autre :

```
function copie (id1, id2)
begin
  val := select valeur from T where id = id1
  update T set valeur = :val where id = id2
  commit
end
```

Exécution concurrente de $copie(a, b)$ et $copie(b, a)$:

$$r_1(a); r_2(b); w_1(b); w_2(a)$$

L'anomalie *Write Skew* se produit, mais autorisée par le contrôle multi-versions.

À retenir

Snapshot Isolation est un algorithme très simple à mettre en œuvre, utilisé par exemple dans des contexte distribués (applications web).

Très fluide, très peu de verrouillage, contrôle bien le cas des mises à jour perdues.

Ne garantit pas la sérialisabilité stricte (des améliorations ont été proposées).

2PL (S10.4)

Verrouillage à deux phases

Le principal (et le plus ancien) algorithme pour assurer la sérialisabilité.

Est réputé engendrer beaucoup de blocages et de rejets.

Repose sur le verrouillage (surprise !) des nuplets.

Les verrous

Pose de verrous sur les nuplets.

- Le verrou partagé (*shared lock* ou s) autorise la pose d'autres verrous partagés sur le même nuplet.
- Le verrou exclusif (*exclusive lock* ou x) interdit la pose de tout autre verrou, exclusif ou partagé, et donc de toute lecture ou écriture par une autre transaction.

Verrous = blocages. Il faut donc en poser :

- le moins possible, surtout pour les verrous exclusifs ;
- pour la durée la plus courte possible.

Transactions et verrous

Le contrôleur pose des verrous pour une transaction T.

- On ne peut poser un verrou partagé sur un nuplet a que s'il n'y a que des verrous partagés sur ce nuplet.
- On ne peut poser un verrou exclusif que si
  - il n'y a aucun autre verrou, ou,
  - il y a un verrou partagé déjà posé par T elle-même (*upgrade* ou promotion)

Une transaction qui n'obtient pas un verrou, est mise en attente.

L'algorithme 2PL

On pose des verrous pour empêcher l'apparition de cycle dans le graphe de sérialisation.

- Lecture sur a, on regarde s'il y a un verrou exclusif sur a ;
  - si oui la transaction T_i est mise en attente.
  - si non, la transaction pose un verrou en lecture et l'opération est exécutée.
- Ecriture sur a, on regarde s'il y a un verrou quelconque sur a ;
  - si oui la transaction T_i est mise en attente.
  - si non, la transaction pose un verrou en écriture et l'opération est exécutée.

La règle 2PL

Chaque tx pose tous ses verrous d'abord (phase 1), et les libère ensuite (phase 2).

2 variantes de 2PL

2PL strict (ou S2PL)

Les verrous exclusifs ne sont relâchés qu'au `commit` ou `rollback` de la tx.

2PL rigoureux (ou SS2PL)

Tous les verrous (exclusifs et partagés) ne sont relâchés qu'au `commit` ou `rollback` de la tx.

En pratique : C'est le protocole 2PL rigoureux (*Strong Strict 2PL* ou *Rigorous 2PL*) qui est mis en œuvre.

Exemple d'exécution avec SS2PL

Prenons l'exécution concurrente : $H = r_1(a); w_2(a); w_2(b); C_2; w_1(b); C_1$

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

Exécution après réordonnancement : $H' = r_1(a); w_1(b); C_1; w_2(a); w_2(b); C_2$

Un gros problème : les `deadlock`

Reprenons notre exemple des mises à jour perdues.

$$r_1(s); r_1(c_1); r_2(s); r_2(c_2); w_2(s); w_2(c_2); C_2; w_1(s); w_1(c_1); C_1$$

- T_1 lit s et c_1, qui sont verrouillés en lecture.
- T_2 lit s et c_2, qui sont verrouillés en lecture ; s partage deux verrous en lecture
- T_2 veut écrire s : conflit, donc blocage de T_2.
- T_1 veut écrire s : conflit, donc blocage de T_1.

C'est l'étreinte fatale ou interblocage, ou encore verrou mortel (*deadlock*) !!

Le système va rejeter une des transactions. Très regrettable, mais toujours mieux que d'introduire des anomalies (?)

Détection des interblocages

Construction du graphe d'attentes :

- les nœuds sont les tx ;
- les arcs matérialisent l'attente par la tx source d'une ressource détenue par la tx cible.

deadlock

Détection de cycle dans le graphe d'attentes !

Quelle tx rejeter après un dealdock ?

Stratégie fondée sur l'estampillage des tx.

Il en existe 2 variantes :

1. *wound-wait* : la plus ancienne tx est autorisée à préemter le nuplet, la plus récente est mise en attente.
2. *wait-die* : la plus ancienne tx est autorisée à attendre, la plus récente est immédiatement rejetée.

✎ Cas de famine ? évité en rejouant les tx avec leur estampille d'origine.

Extensions de 2PL

- verrous de mise-à-jour ; clause `for update`
- verrous d'incrément, etc.
- protocole arborescent et verrous de prédicats
- multi-granularité et verrous d'intention

Un point sur les plans

- 2PL : protocole de verrouillage à 2 phases
- S2PL : 2PL strict
- SS2PL : 2PL rigoureux
- SI : isolation par cliché (famille MVCC - optimiste)
- REP : réparable
- PAC : prévention des annulations en chaîne
- CSER : sérialisable par conflit
- SER : sérialisable
- SERIE : sériel

✎ Construire un plan SER et non 2PL (à l'aide d'« écritures aveugles »).

À retenir

Le SS2PL est le seul algorithme actuellement utilisé pour garantir la sérialisabilité.

Il repose intensivement sur la pose de verrous qui ne sont relâchés qu'à la fin de la transaction.

Il entraîne des interblocages, et donc le rejet de certaines transactions : difficilement explicable pour un utilisateur, pose le problème de resoumettre la transaction.