École Doctorale – 2012/2013

Systèmes de Bases de Données Avancés

3. Contrôle de concurrence

École nationale Supérieure d'Informatique

Plan

1. Notion de transaction

2. Théorie de la « sérialisabilité »

3. Les différents types de contrôleurs de concurrence

3. Contrôle de concurrence

3.1. Notion de transaction

3.1. Architecture

- Qu'est-ce qu'une Transaction ? -

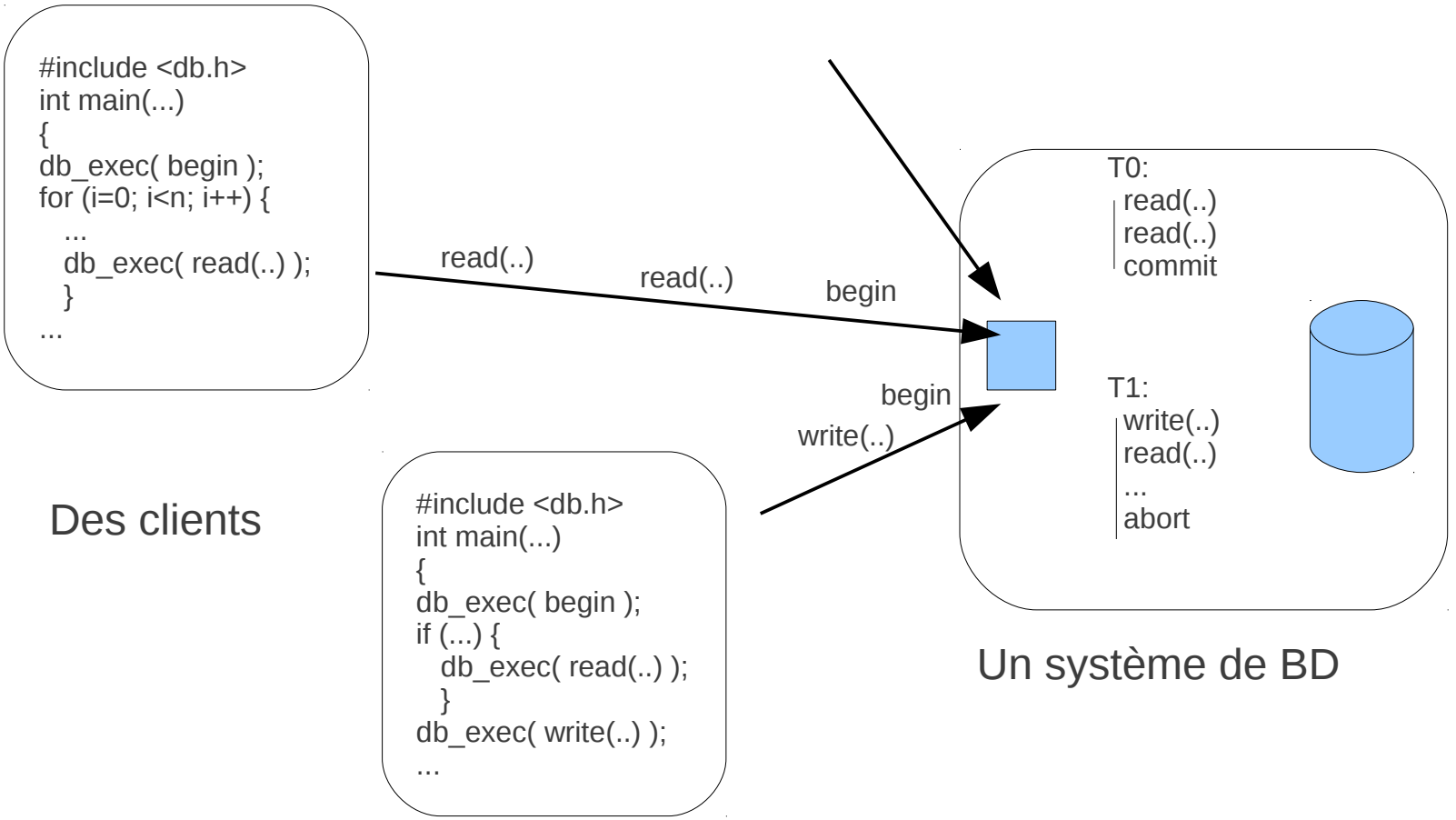
- Un ensemble d'opérations d'accès (à la base) issues d'un programme d'application (utilisateur), consultant et/ou modifiant l'état de la BD.
- Elle se termine de 2 manières possibles :

  Par **validation** : COMMIT (utilisateur) – terminaison normale
  toutes les m-a-j effectuées deviennent persistantes et les lectures sont correctes.

  Par **annulation** : ABORT (utilisateur ou système) – terminaison anormale
  toutes les m-a-j effectuées sont annulées et les valeurs lues ne sont pas sûres
- Ex: Begin ... Read(a) ... Read(b) ... Write(c) ... Read(d) ... COMMIT

3.1. Architecture

- Qu'est-ce qu'une Transaction ? -



Exemples

- Programme pour la réservation de places (transports, ...)
- Programme de virement entre deux comptes
- Programme de mise à jour d'un stock
- Programme d'inscription en-ligne des nouveaux bacheliers
- Programme de prévisions météorologiques
- ...
- Et en général, tout programme d'accès à un ensemble de données partagées

Propriétés A.C.I.D

Pour avoir un comportement « **correct** » lors de l'exécution d'un ensemble de transactions, les propriétés suivantes doivent être vérifiées

- **A**tomicité : Chaque transaction doit s'exécuter entièrement ou alors tous ses effets partielles (sur la BD) seront annulés
- **C**ohérence : Aucune transaction ne doit laisser la BD (lors de sa terminaison) dans un état incohérent (relativement aux contraintes d'intégrités)
- **I**solation : L'état intermédiaire (de la BD) généré par une transaction ne doit pas être visible par les autres transactions concurrentes
- **D**urabilité : Une transaction validée ne peut plus être annulée.

Système transactionnel

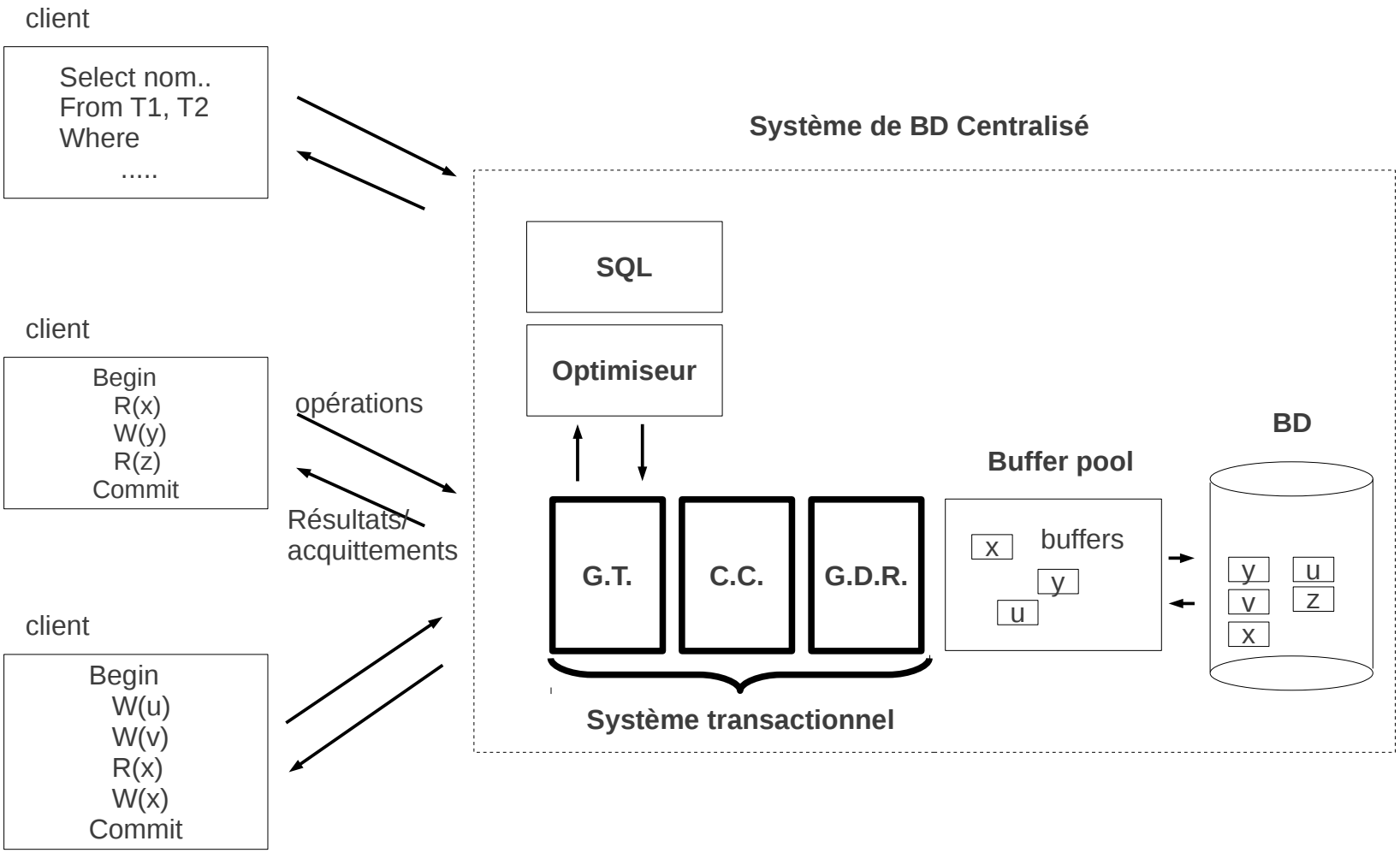
Le système transactionnel, doit garantir les propriétés : d'atomicité, d'isolation, de durabilité et une partie de la cohérence (celle en relation avec les 3 autres propriétés)

Gestionnaire de **T**ransactions ⟺
- identifie les transactions
- oriente les opérations vers le CC cible
- coordonne la validation atomique

Contrôleur de **C**oncurrence ⟺
- Contrôle l'ordre d'exécution des opérations afin de maintenir la propriété d'isolation
- éventuellement gère l'interblocage

Gestionnaire de **D**onnées et de **R**ecouvrement ⟺
- exécute les opérations d'accès
- implémente les techniques de reprise et de points de contrôle

Architecture globale



3.2. Théorie de la sérialisabilité

Contrôle de Concurrence

Les opérations d'accès (read, write, commit, abort) des différentes transactions actives, arrivent au niveau du contrôleur de concurrence (CC) dans un ordre quelconque

Cette exécution « entrelacée » de ces différentes opérations peut provoquer des incohérences dans la BD, même si chacune des transactions impliquées, exécutée seule ne provoque aucune incohérence

Le CC doit alors « ordonnancer » les différentes opérations de telle sorte à éviter ces problèmes

Les règles permettant d'ordonnancer les opérations d'accès sont appelées Protocole de contrôle de concurrence

Problèmes de l'exécution concurrente

- Perte d'opération
  - Mises à jour perdues
- Lectures non reproductibles
  - Une transaction lit deux fois la même donnée et trouve des valeurs différentes
- Introduction d'incohérences
  - Dans la BD ou dans la vue que la transaction donne à l'utilisateur

Exemple 1

```
procedure Ajouter (numCompte, somme)
   BEGIN
      temp := READ( numCompte)
      temp := temp + somme
      WRITE( numCompte, temp)
      COMMIT
   END
```

posons alors :
ri = read de Ti
wi = write de Ti
ci = commit de Ti
ai = abort de Ti

Chaque exécution de cette procédure génère une transaction

Soient les 2 transactions suivantes :
T1 : rajoute 1000 au compte X = (r1(X), w1(X), c1)
T2 : rajoute 200 au compte X = (r2(X), w2(X), c2)

Une exécution 'entrelacée' de T1 et T2 pourrait être par exemple :
$(r1(X)_{[X=5000]}, r2(X)_{[X=5000]}, w1(X)_{[X=6000]}, w2(X)_{[X=5200]}, c1, c2)$
==> incorrecte : la mise à jour de T1 est perdue
Par contre l'exécution suivante :
$(r1(X)_{[X=5000]}, w1(X)_{[X=6000]}, r2(X)_{[X=6000]}, w2(X)_{[X=6200]}, c1, c2)$
==> produit une exécution correcte

Exemple 2

```
procedure virement(compteDebite, compteCredite, S)
    BEGIN                                                    T1:
        t := READ(compteDebite); t := t – S                        R1(X)
        WRITE(compteDebite, t)                                     W1(X)
        t := READ(compteCredite); t := t + S                       R1(Y)
        WRITE(compteCredite, t)                                    W1(Y)
        COMMIT                                                 C1
    END

procedure Afficher_cumul(compte1, compte2)
    BEGIN                                                    T2:
        temp1 := READ(compte1)                                 R2(X)
        temp2 := READ(compte2)                                 R2(Y)
        ecrire(temp1+temp2)
        COMMIT                                                 C2
    END
```

L'exécution de ces 2 procédures peut générer les entrelacements suivant :

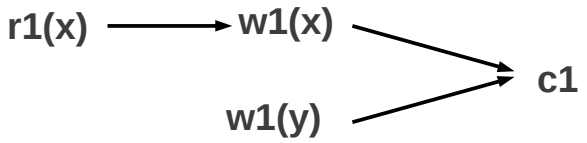
(R1(X),W1(X),R2(X),R2(Y),R1(Y),W1(Y),C1,C2) ==> incorrect

(R1(X),W1(X),R2(X),R1(Y),W1(Y), R2(Y),C1,C2) ==> correct

Formalisation

Une Transaction peut être modélisée par un **ordre partiel** ($<_i$) se terminant soit par ai soit par ci. les sommets sont les opérations {r,w}, les arcs modélisent l'ordre d'exécution (relation de précédence)

Ex: T_1 = ({r1(x), w1(x), w1(y), c1} , { r1(x)$<_1$w1(x) , w1(x)$<_1$c1 , w1(y)$<_1$c1 })



Dans une transaction, deux opérations (p et q) sur la même donnée (x) doivent toujours être en relation : p(x) < q(x) ou bien q(x) < p(x)

Un **historique** est une structure qui modélise une exécution (pouvant être entrelacée), d'un certain nombre de transactions ==> Ordre partiel

Deux opérations p et q sont dites **conflictuelles** si elles appartiennent à 2 transactions différentes et portent sur la même donnée et l'une au moins est un write
==> 2 opérations conflictuelles ne sont pas commutatives

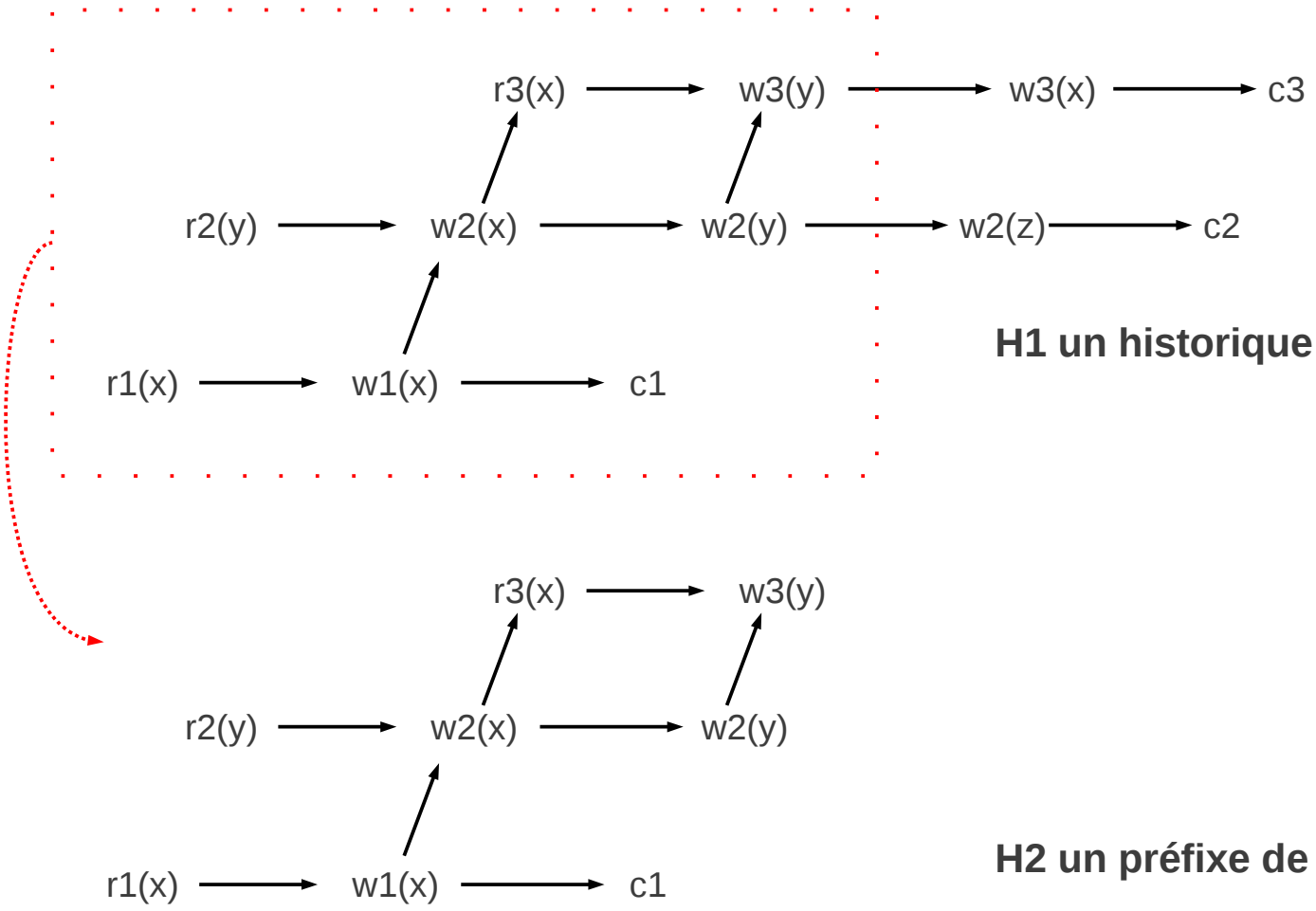
Un **historique complet** (H,$<_H$) est formellement défini :

a) $H = \cup_{i=1..n} T_i$ b) $<_H \supseteq \cup_{i=1..n} <_i$

c) pour chaque couple d'opérations conflictuelles p et q, on a soit $p <_H q$ soit $q <_H p$

Formalisation

Un historique est donc un **préfixe** d'un historique complet



Formalisation

- Une transaction T_i dans un historique H est dite :

  Validée si (ci ∈ H) **Annulée** si (ai ∈H) **Active** si (ci ∉H et ai ∉H)

- La **projection validée** d'un historique H (notée C(H)) est un historique obtenu à partir de H en supprimant toutes les opérations des transactions actives ou annulées
  ==> C(H) est donc un historique complet

- Un historique H1 est **équivalent** à un historique H2 (H1 ≡ H2) ssi:
  a- H1 et H2 sont formés du même ensemble de transactions et des mêmes opérations
  b- L'ordre des opérations conflictuelles des transactions non annulées est le même dans les deux historiques

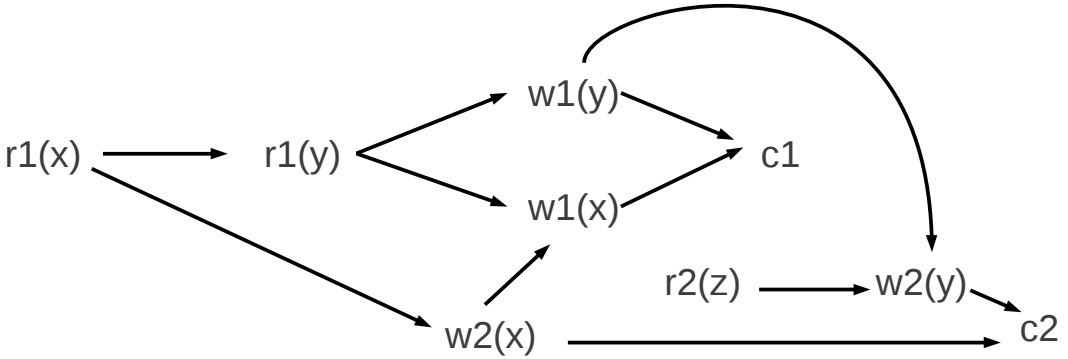
  ∀ pi ∈ Ti et pj ∈ Tj, 2 op conflictuelles, et ai ∉ Ti et aj ∉ Tj

  alors $pj <_{H1} qj \Rightarrow pj <_{H2} qj$

  - **Les 2 exécutions produisent le même effet sur la BD et sur les transactions**

Exemple

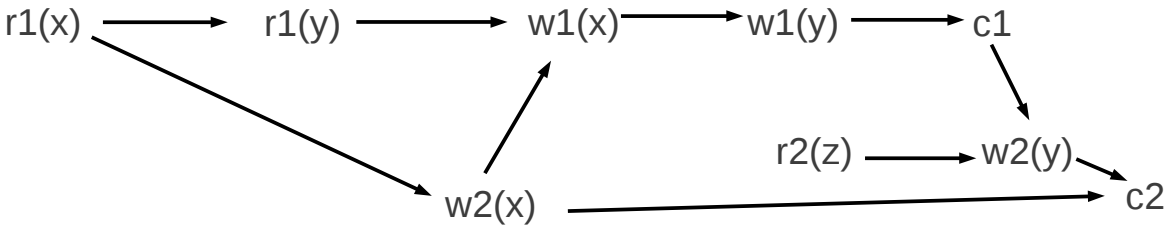
H1



H1 ≡ H2

les op conflictuelles de H1 et H2 sont ordonnées de la même manière

H2

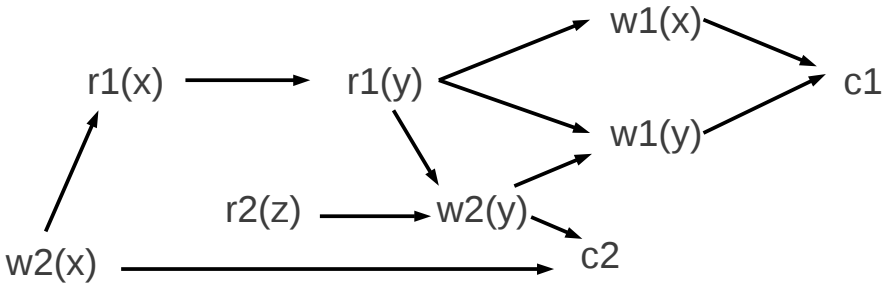


H1 ≠ H3

contre exemple :

$r1(x) <_{H1} w2(x)$ alors que $w2(x) <_{H3} r1(x)$

H3



H2 ≠ H3

contre exemple :

$r1(x) <_{H2} w2(x)$ alors que $w2(x) <_{H3} r1(x)$

La sérialisabilité (1/4)

Un historique complet H est « **sériel** » ssi pour chaque couple de transactions Ti et Tj dans H, soit toutes les opérations de Ti précèdent celles de Tj soit toutes les opérations de Tj précèdent celles de Ti
C'est une exécution en série des transactions.
==> **il n'y a pas d'entrelacement des opérations**

Remarque : un historique sériel est forcément complet

Un historique H est « **sérialisable** » si sa projection validée C(H) est un historique équivalent à un historique sériel formé par les mêmes transactions

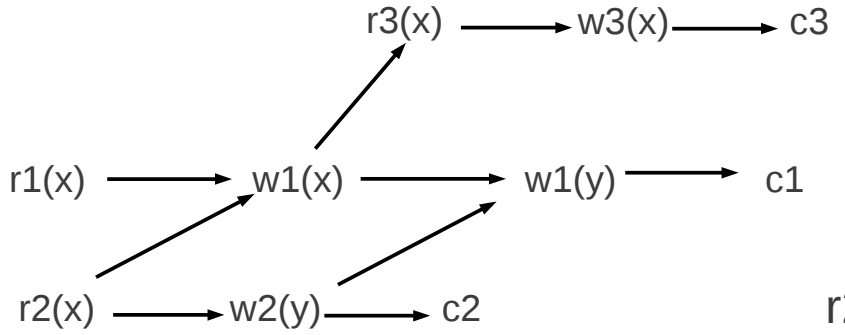
Exemples: T1 = r1(x);w1(x); r1(y); w1(y); c1 T2 = r2(x); r2(y); c2

H1 = (r1(x), w1(x), r2(x), r2(y), r1(y), w1(y), c1, c2)
non sérialisable car H1 n'est équivalent ni à T1;T2 ni à T2;T1

H2 = (r1(x),w1(x),r2(x),r1(y),w1(y), r2(y), c1, c2)
sérialisable car H2 est équivalent à T1;T2

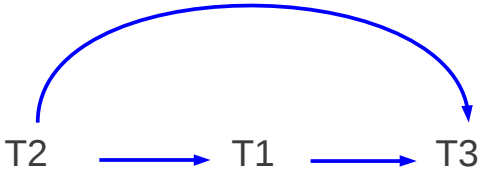
La sérialisabilité (2/4)

Le **graphe de sérialisation** d'un historique H (noté **SG(H)**) est un graphe orienté sans circuit (GOSC) où les noeuds sont les transactions validées de H et les arcs de type Ti→Tj indiquent l'existence d'une opération de Ti qui précède et est en conflit avec une opération de Tj dans l'historique H



H

SG(H)



r2(x) < w1(x) ou w2(y) < w1(y)
donne **T2→T1**
w1(x) < r3(x) ou w1(x) < w3(x) ou r1(x) < w3(x)
donne **T1→T3**
r2(x) < w3(x)
donne **T2→T3**

si à la place de w3(x) on avait w3(z),
il n'y aurait pas eu d'arc entre T2 et T3

La sérialisabilité (3/4)

Théorème :

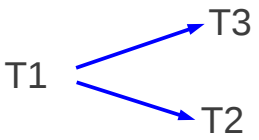
Un historique H est sérialisable ssi SG(H) est sans circuit

Conséquence :

Si un historique complet H possède un graphe de sérialisation SG(H) sans circuit, alors H est équivalent à tout historique sériel obtenu par un tri topologique de SG(H)

Exemple :

H = (w1(x); w1(y); c1; r2(x); r3(y); w2(x); c2; w3(y); c3)

SG(H) = 

il existe alors deux tri topologiques :

T1; T2; T3 et T1; T3; T2

et donc H est équivalent à une exécution en série T1;T2;T3 ou T1;T3;T2 (les deux donneront le même résultat)

La sérialisabilité (4/4)

Démonstration du théorème

SG(H) sans circuit => H Sérialisable

SG(H) est sans circuit => on peut générer un tri topologique de ses sommets.
Soit (T1,T2, Tm) un tel tri. L'historique sériel Hs = (T1,T2, Tm) est forcément équivalent à C(H) (proj. Validée de H) et donc que H est SR, car :
1- Hs et C(H) sont formés des mêmes transactions (T1, T2 ... Tm) – trivial.
2- Chaque couple d'opérations conflictuelles pi et qj sont ordonnées de la même manière dans les 2 historiques.

Si pi $<_{C(H)}$ qj alors il existe un arc Ti→Tj dans SG(H)

donc Ti apparait avant Tj dans tout tri topologique, donc dans Hs
donc toutes les op de Ti précèdent celles de Tj, en particulier pi$<_{Hs}$qj

H Sérialisable => SG(H) sans circuit

H sérialisable => C(H) équivalent à un certain historique sériel Hs
de plus s'il existe un arc Ti→Tj dans SG(H) cela veut dire qu'il existe un couple d'op conflictuelles pi et qj tel que pi$<_{C(H)}$qj et donc pi$<_{Hs}$qj donc Ti apparaît avant Tj dans Hs
donc s'il existait un circuit dans SG(H) tel que T1→T2→T3→...T1 on aurait eu une contradiction dans l'ordre d'apparition des transactions de Hs (T1 avant T2 avant ...T1)
donc il ne peut pas exister de circuits dans SG(H).

Autres problèmes causés par les annulations

Quand une transaction est annulée, ses écritures dans la BD seront défaites

Pb : il se peut que d'autres transactions ont déjà lues ces données
Ex: H = w1(x), r2(x), w2(x), c2, a1 à ce niveau les m-a-j de T1 seront défaites (w1(x)) et la valeur lue par T2 (r2(x)) n'est plus valide et comme T2 a déjà validé, on ne peut plus l'annuler.

Solution : retarder c2 jusqu'à la terminaison de T1 (a1 ou c1)

Formellement, une transaction *Ti lit une donnée x à partir de Tj* dans H si:

- $w_j(x) <_H r_i(x)$
- aj ne précède pas ri(x) dans H
- s'il existe un wk(x) tel que $w_j(x) <_H w_k(x) <_H r_i(x)$, alors $a_k <_H r_i(x)$

Un historique est dit « **Recouvrable** » si à chaque fois que Ti lit une donnée à partir de Tj ($i \neq j$) on a $c_j <_H c_i$

autrement dit : H est recouvrable si chaque transaction ne peut valider qu'après la validation des transactions à partir desquelles elle a lu des données.

Si Tj est annulée il en sera de même pour Ti (**annulations en cascade**)

Autres options désirables

Pour éviter les annulations en cascade (qui dégradent les performances), on n'autorise pas la lecture de données écrites par des transactions encore actives.

→ On dit alors que H **évite les annulations en cascade**

A chaque fois qu'une transaction Ti lit une donnée x à partir de Tj (i ≠ j) dans H, on doit avoir $c_j <_H r_i(x)$

Pour faciliter l'implémentation de la procédure d'annulation (à l'aide des images avant), on impose une contrainte supplémentaire stipulant qu'une transaction ne peut lire ou écrire sur une donnée déjà écrite par une transaction active.

→ On dit alors que H est **strict**

A chaque fois qu'on a dans un historique H: $w_j(x) <_H p(x)$ (avec p = ri ou wi et i≠j) on doit aussi avoir : $a_j <_H p(x)$ ou bien $c_j <_H p(x)$

Exemple (x=10 au départ)

H = { w1(x,20) , w2(x,30), a1, a2 }

img-av=10 img-av=20 on aura X=20 au lieu de 10

Exemples

Soient T1 = w1(x), w1(y), w1(z), c1 et T2 = r2(u), w2(x), r2(y), w2(y), c2

L'historique H1 = w1(x), w1(y), r2(u), w2(x), r2(y), w2(y), c2, w1(z), c1
n'est pas recouvrable, car T2 a validé avant T1 (c2 < c1), alors qu'elle a lu une donnée produite par T1 (w1(y) < r2(y)).

L'historique H2 = w1(x), w1(y), r2(u), w2(x), r2(y), w2(y), w1(z), c1, c2
est recouvrable, mais n'évite pas les cascades d'annulations, car T2 a lu une donnée produite par T1 avant que celle-ci ne valide (w1(y) < r2(y) < c1).

L'historique H3 = w1(x), w1(y), r2(u), w2(x), w1(z), c1, r2(y), w2(y), c2
est recouvrable et évite les annulations en cascade, mais n'est pas strict, car T2 a écrit sur x alors qu'il contenait une valeur non encore validée par T1 (w1(x) < w2(x) < c1).

L'historique H4 = w1(x), w1(y), r2(u), w1(z), c1, w2(x), r2(y), w2(y), c2
est recouvrable, évite les annulations en cascade, et est strict.

Résumé

- Sérialisable (critère important)

  Produire les mêmes effets qu'une exécution en série

- Recouvrable (critère important)

  Une transaction T ne peut valider que si toutes les transactions qui ont écrits des données lues par T se sont terminées (annulées ou validées)
  – retarde la validation

- Évite les annulations en cascades (optionnel ==> performances)

  Une transaction ne peut lire que des données écrites par des transactions validées – retarde les lectures

- Stricte (optionnel ==> performances)

  Même chose que précédemment, en plus T ne doit pas écraser des données écrites par des transaction non validées
  – retarde les lectures et écritures

3.3. Les différents types de contrôleurs de concurrence

Comportement agressif / conservateur

- Quand une opération arrive au niveau du CC, 3 possibilités sont offertes
  - L'exécuter immédiatement => stratégie agressive
  - La faire patienter (en l'insérant dans une file d'attente) => stratégie conservatrice
  - Être obligé de la rejeter (ce qui provoque l'annulation de la transaction)
- Il y a des contrôleurs de concurrence qui favorisent la première approche
  - Ce qui les amènent à rejeter plus tard d'autres opérations
  - Approche bien adaptée s'il y a peu de conflits
- Il y a des contrôleurs de concurrence qui favorisent la deuxième approche
  - Ce qui permet de minimiser le nombre de rejets ultérieurs
  - Approche bien adaptée s'il y a beaucoup de conflits

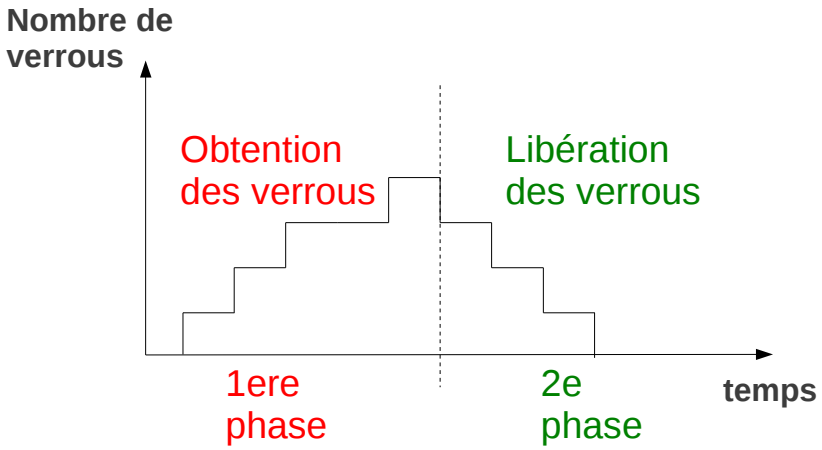
2PL de base

- Two Phases Locking protocol -

- Avant d'exécuter une opération d'accès, la transaction doit d'abord obtenir un verrou sur la donnée manipulée
- Si une transaction commence à libérer des verrous, elle ne pourra plus en demander d'autres

Généralement il y a au moins 2 types de verrous:
lr(x): verrou partagé (plusieurs lectures en même temps) – en conflit avec lw(x)
lw(x): verrou exclusif (1 seule écriture à la fois) – en conflit avec lw(x) et avec lr(x)

Pour déverrouiller une donnée x:
ur(x) : libère le verrou partagé sur x
uw(x): libère le verrou exclusif sur x



Exemple d'historique 2PL:
lr1(x), r1(x), lr2(y), r2(y), lr2(z), r2(z), ur2(y), lw1(y), w1(y), ur2(z), c2, ur1(x), uw1(y), c1

2PL de base
- Propriétés -

Soit H un historique vérifiant le protocole 2PL, et soient p , q deux opérations dans C(H), on a alors les propriétés suivantes :

a) ∀ p une opération (r ou w) et ∀ i une transaction, on a: **$lp_i(x) < p_i(x) < up_i(x)$**
le verrouillage (lock) d'une donnée précède toujours l'opération d'accès qui précède toujours son déverrouillage (unlock)

b) si $p_i(x)$ et $q_j(x)$ sont conflictuelles, alors on a soit: **$up_i(x) < lq_j(x)$ soit $uq_j(x) < lp_i(x)$**
(par définition des opérations conflictuelles)

c) si $p_i(x)$ et $q_i(y)$ appartiennent à la même transaction i, alors : **$lp_i(x) < uq_i(y)$**
la phase 1 (verrouillage) précède toujours la phase 2 (déverrouillage)

2PL de base

- Preuve de sérialisabilité -

Soit H un historique conforme avec les règles du 2PL

Montrons qu'il n'existe pas de circuit dans SG(H):

1) A partir de la propriété b), on déduit que s'il existe une arc Ti → Tj dans SG(H) alors il existe 2 opérations conflictuelles pi et qj sur la même donnée x, tel que **$up_i(x) < lq_j(x)$**

2) On peut alors (par récurrence) généraliser dans le cas où il existe un chemin T1 → T2 → T3 ... → Tn dans SG(H), il doit alors forcément exister 2 données x et y et 2 opérations $p_1(x)$ et $q_n(y)$ tel que **$up_1(x) < lq_n(y)$**

3) S'il existait un circuit T1 → T2 → T3 ... → T1 dans SG(H) alors d'après le point 2), il existe 2 données x et y et 2 opérations $p_1(x)$ et $q_1(y)$ tel que $up_1(x) < lq_1(y)$

c-a-d la phase 2 (unlock) précède la phase 1 (lock) dans la même transaction
ce qui contredit une des propriétés du 2PL => Impossible
Donc il ne peut pas exister de circuit dans SG(H)

Donc H est forcément sérialisable.

2PL de base

- Exemples -

Il existe des historiques conformes au protocole 2PL et sérialisables :

H1 = { r1(x), r2(y), r2(z), w1(y), w3(z), c2, w3(x), c1, c3 }

On peut placer les opérations sur les verrous en gardant le même ordre des opérations d'accès :

H1 = { lr1x, r1(x), lr2y, r2(y), lr2z, r2(z), ur2y, lw1y, w1(y), ur2z, lw3z, w3(z), c2, ur1x, lw3x, w3(x), uw3x, uw1y, c1, uw3z c3 }

Il existe par contre, des historiques sérialisables qui ne peuvent pas être produits par un contrôleur 2PL :

H2 = { r1(x), w2(x), w1(y), w2(y), c1, c2 } => Non 2PL

SG(H2) : T1 → T2

H3 = { r1(x), w2(x), c2, r3(y), w1(y), c1, c3 } => Non 2PL

SG(H3) : T3 → T1 → T2

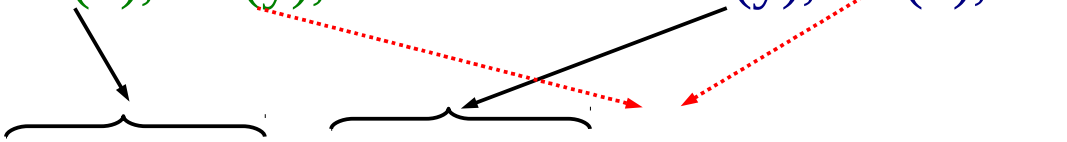
2PL de base
- Interblocage (Deadlock) -

Un ensemble de transactions T1, T2, ... Tn sont en situation d'interblocage, si chacune d'entre elles est bloquée en attendant la libération de ressources détenues par d'autres transactions de l'ensemble.
=> aucune transaction ne peut avancer dans son exécution

Puisque 2PL fait attendre des transactions pouvant déjà avoir acquit des verrous au préalable, une situation d'interblocage peut alors apparaître

Exemple:

T1 = r1(x); w1(y); c1 T2 = r2(y); w2(x); c2



H = lr1(x), r1(x), lr2(y), r2(y), interblocage ...

la prochaine op de T1 (w1(y)) est bloquée et celle de T2 (w2(x)) l'est aussi

2PL de base

- Interblocage : « Détection/Guérison » -

On maintient un **graphe des attentes** (wait-for-graph : WFG) :

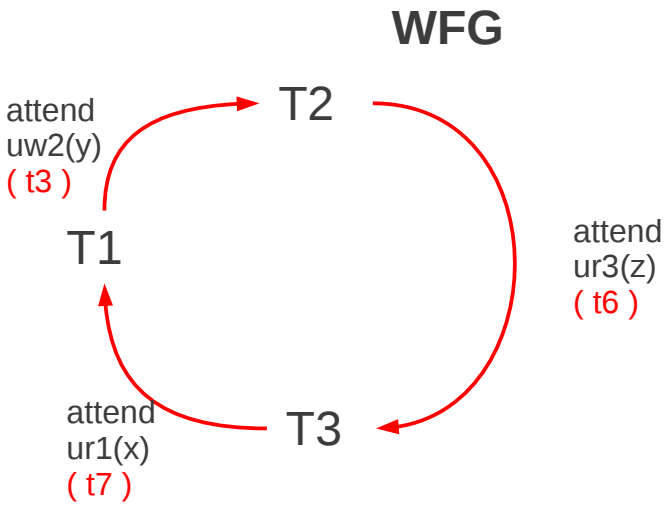
les sommets sont les **transactions actives** et les arcs représentent les relations d'attentes de libération de verrous

Ti → Tj veut dire que Ti a demandé un verrou conflictuel avec celui détenu par Tj et doit donc attendre sa libération pour continuer son exécution.

Si le WFG contient un **circuit** alors il y a un **interblocage**

Solution : annuler une transaction (victime) appartenant au circuit

	T1	T2	T3
t0	r1(x)		
t1		w2(y)	
t2			r3(z)
t3	r1(y)		
t4	---	r2(z)	
t5	---		r3(x)
t6	---	w2(z)	
t7	---	---	w3(x)
t8	---	---	---



2PL de base

- Interblocage : « Prévention » -

a) Par « timeout » : dès qu'une transaction bloquée par un verrou, dépasse un certain délai d'attente, elle est annulée

b) Ne pas permettre à une transaction d'attendre, s'il y a un risque d'interblocage potentiel
à chaque transaction Ti est associé une estampille (timestamp) e(Ti)
quand Ti demande un verrou en conflit avec celui détenu par Tj, le CC applique l'une des 2 règles suivantes:

Wait-Die : si e(Ti) < e(Tj) /* Ti est plus ancienne que Tj */
alors Ti est autorisée à attendre
sinon Ti est annulée, puis relancée en gardant la même estampille

Wound-Wait : si e(Ti) < e(Tj)
alors Tj est annulée, puis relancée en gardant la même estampille
sinon Ti est autorisée à attendre

2PL de base
- Non recouvrable -

Par un contre-exemple, on montre que 2PL n'est pas recouvrable
une transaction Tj peut valider avant la terminaison d'une transaction Ti à partir de laquelle Tj a lu une donnée

=> Si Ti est par la suite annulée, Tj aura alors lu une donnée incorrecte

Soit H = lw1(x), w1(x), uw1(x), lr2(x), r2(x), ur2(x), c2, ... a1

Validation précoce

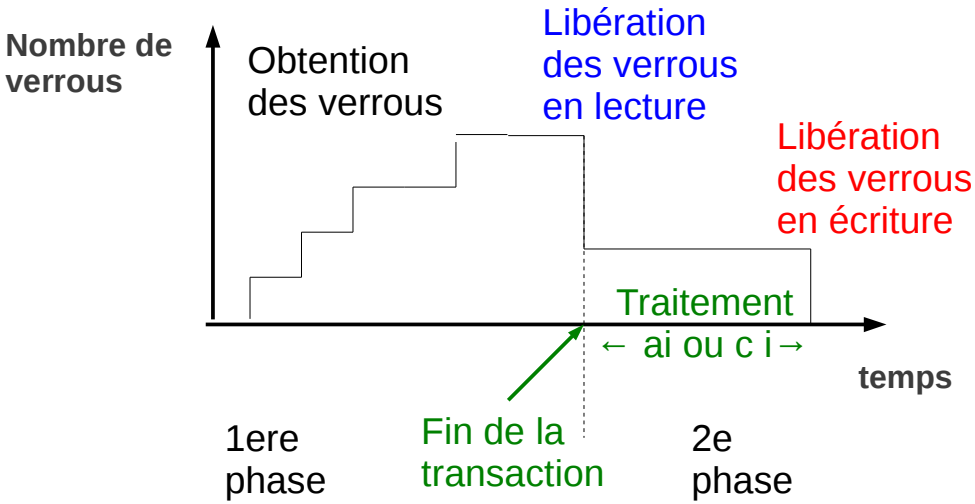
Cet historique est conforme aux propriétés du 2PL mais n'est pas recouvrable (w1(x) < r2(x) et c2 < a1)

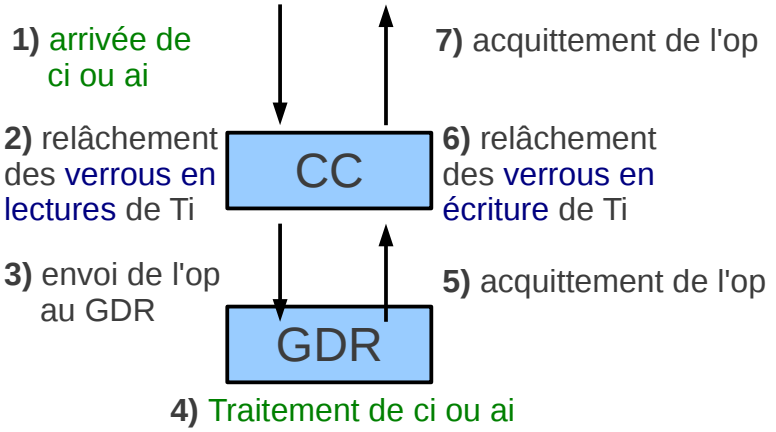
2PL Strict

C'est le même principe que 2PL de base, sauf que le relâchement des **verrous en écriture** est retardé après la terminaison de la transaction

En théorie, les verrous en lectures peuvent être relâchés avant, tout en respectant le protocole à deux phases.

En pratique, comme le CC ne peut pas prévoir les opérations qui vont arriver, les **verrous en lectures** d'une transaction Ti restent positionnés jusqu'à la réception d'une opération de terminaison (ci ou ai) => principe du **precommit**





2PL Strict

- Recouvrable, évite les annulations en cascade et strict -

Recouvrable : Toute transaction Ti qui lit une donnée (x) produite par une autre transaction Tj, ne peut valider qui si Tj valide avant
c-a-d si wj(x) < ri(x) alors cj < ci
avec 2PL Strict cette propriété est vérifiée car :
wj(x) < cj < uwj(x) et uwj(x) < lri(x) < ri(x) < ci

Sans annulations en cascade : aucune transaction ne peut lire une donnée non validée. c-a-d si wj(x) < ri(x) alors cj < ri(x)
avec 2PL Strict cette propriété est vérifiée car :
wj(x) < cj < uwj(x) et uwj(x) < lri(x) < ri(x)

Strict : aucune transaction ne peut lire ou écrire sur une donnée non validée. c-a-d si wj(x) < pi(x) alors cj < pi(x) avec p =(r ou w)
avec 2PL Strict cette propriété est vérifiée car :
wj(x) < cj < uwj(x) et uwj(x) < lpi(x) < pi(x) avec p =(r ou w)

Autres variantes de 2PL

- **2PL Statique**
  - Si les transactions « pré-déclarent » les données qu'elles vont manipuler, le CC peut faire patienter les transactions durant les débuts de leurs exécutions, jusqu'à ce que tous les verrous nécessaires soient disponibles
  - Si au moins un des verrous demandés n'est pas disponible, aucun verrou ne sera finalement positionné tant que la transaction reste en attente
  - Il ne peut donc y avoir d'interblocage
- **2PL sans attente**
  - Toute transaction qui ne peut obtenir un verrou est annulée. Elle sera relancée lorsque le verrou demandé sera à nouveau libre
  - Evite les interblocage et limite la congestion sur les données.

Problème des fantômes

Tab

x	y	z
4	1	10
2	1	15

3	1	20

Soient 2 transactions T1 et T2 :

T1:1 select sum(x) from Tab where y = 1
T1:2 select sum(x) from Tab where y = 1

T2:1 insert into Tab values (3, 1, 20)

Supposons que les opérations entrelacées de T1 et T2 s'exécutent dans cet ordre : H = { T1:1 , T2:1 , c2 , T1:2 , c1 }

Bien que les opérations T1:1 et T1:2 soient identiques, la dernière (T1:2) ne retournera pas le même résultat que T1:1, car un nouveau tuple (celui inséré par T2:1) a été inséré entre les exécutions de T1:1 et T1:2

==> C'est le problème des fantômes !

peut être résolu par le verrouillage multiniveau ou par le verrouillage d'index

Problème des fantômes

- Exemple avec 2 tables-

Soient 2 tables relationnelles :

Comptes(numcpt, numcli, numagc, solde, ...)
Agences(numagc, capital, …)

Soit T1 une transaction qui vérifie que la somme des soldes de tous les comptes appartenant à une agence 'a' dans la table Comptes, soit égale au capital de 'a' dans la table Agences :

1) X ← select sum(solde) from Comptes where numagc = a ;
2) Y ← select capital from Agences where numagc = a ;
3) Tester si (X=Y)

Soit T2 une transaction qui rajoute un nouveau compte (avec un solde initial 's') au niveau de l'agence numéro 'a' :

1') insert into Comptes values(nnn, ccc, a, s) ;
2') update Agences set capital = capital + s where numagc = a ;

L'entrelacement suivant, peut provoquer un problème de fantôme :

1 , 1' , 2' , 2 , 3

Degré d'isolation et verrouillage

On considère deux types de transactions :
En consultation seulement (**readers**) ou En lecture/écriture (**updaters**)

- Read uncommitted (+lectures sales, +Non reproductibles, +fantômes)
  - Pas de verrouillage pour les readers / 2PL strict pour les updaters
- Read Committed (-lectures sales, +Non reproductibles, +fantômes)
  - Verrouillage court pour les readers / 2PL strict pour les updaters
- Repeatable Read (-lectures sales, -Non reproductibles, +fantômes)
  - 2PL strict (sur tuples) pour les readers et les updaters
- Serializable (-lectures sales, -Non reproductibles, -fantômes)
  - 2PL strict pour readers et updaters + {verrouillage multiniveaux ou index}

2PL avec verrouillage multi-niveaux

- La taille du granule influence sur les performances
  - Petits granules => surcharge dans la gestion des verrous
  - Grands granules => augmentent le nombre de conflits
- Offrir la possibilité de verrouiller des objets à différents niveaux
  - Par exemple : BD – Segment – Fichier – Page – Tuple – Attribut
- Verrouillage hiérarchique
  - Le verrouillage d'un objet O => le verrouillage implicite de tous ses descendants
  - Ex: un verrou sur un fichier F représente le verrouillage de toutes ses pages, de tous ses tuples et de tous ses attributs
- Permet entre autre de résoudre le problème des « fantômes »
  - Ex: lors de l'insertion de nouvelles données par une transaction

2PL avec verrouillage multi-niveaux
- Les verrous d'intention -

- En plus des verrous de lecture et d'écriture (r et w), on définit les verrous intentionnels suivants:
  - ir(O) : indique que la transaction a l'intention de lire des sous-objet de O
  - iw(O): indique que la transaction a l'intention de modifier des sous-objet de O
  - riw(O) : est la combinaison des verrous r et iw
- Si une transaction pose un verrou r(O) sur un objet O (une table par exemple), elle pourra alors la lire dans sa globalité sans avoir à demander des verrous r sur ses composants (les tuples de la table par exemple)
- Si une transaction pose un verrou w(O) sur un objet O (une table par exemple), elle pourra alors la modifier dans sa globalité sans avoir à demander des verrous w sur ses composants.

2PL avec verrouillage multi-niveaux
- Les verrous d'intention -

- Matrice de compatibilité

	ir	iw	r	riw	w
ir	1	1	1	1	0
iw	1	1	0	0	0
r	1	0	1	0	0
riw	1	0	0	0	0
w	0	0	0	0	0

- Matrice de conversion

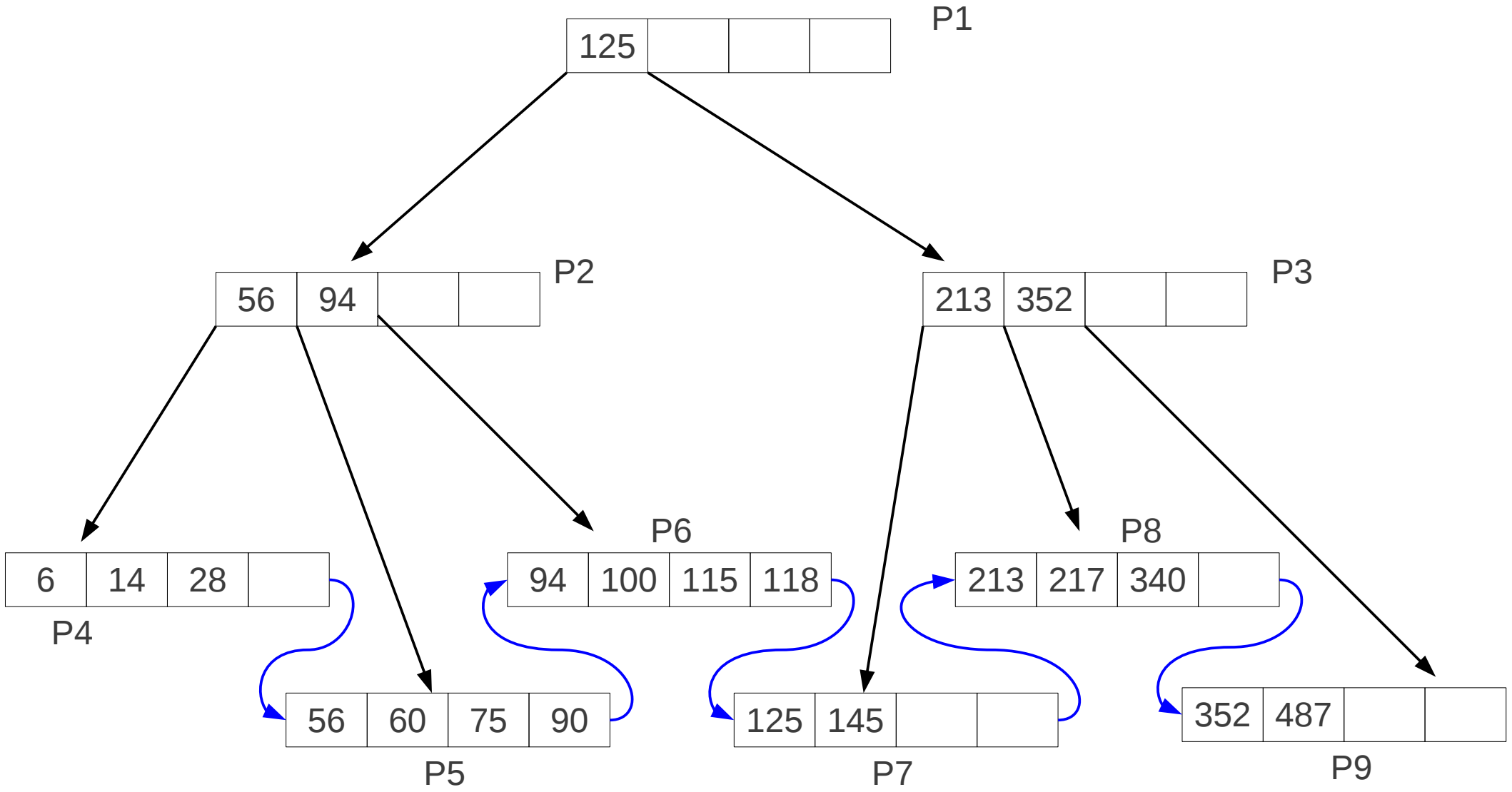
ir < iw
ir < r
iw < riw
r < riw
riw < w

Verrous demandés / Verrous acquits

	ir	iw	r	riw	w
ir	ir	iw	r	riw	w
iw	iw	iw	riw	riw	w
r	r	riw	r	riw	w
riw	riw	riw	riw	riw	w
w	w	w	w	w	w

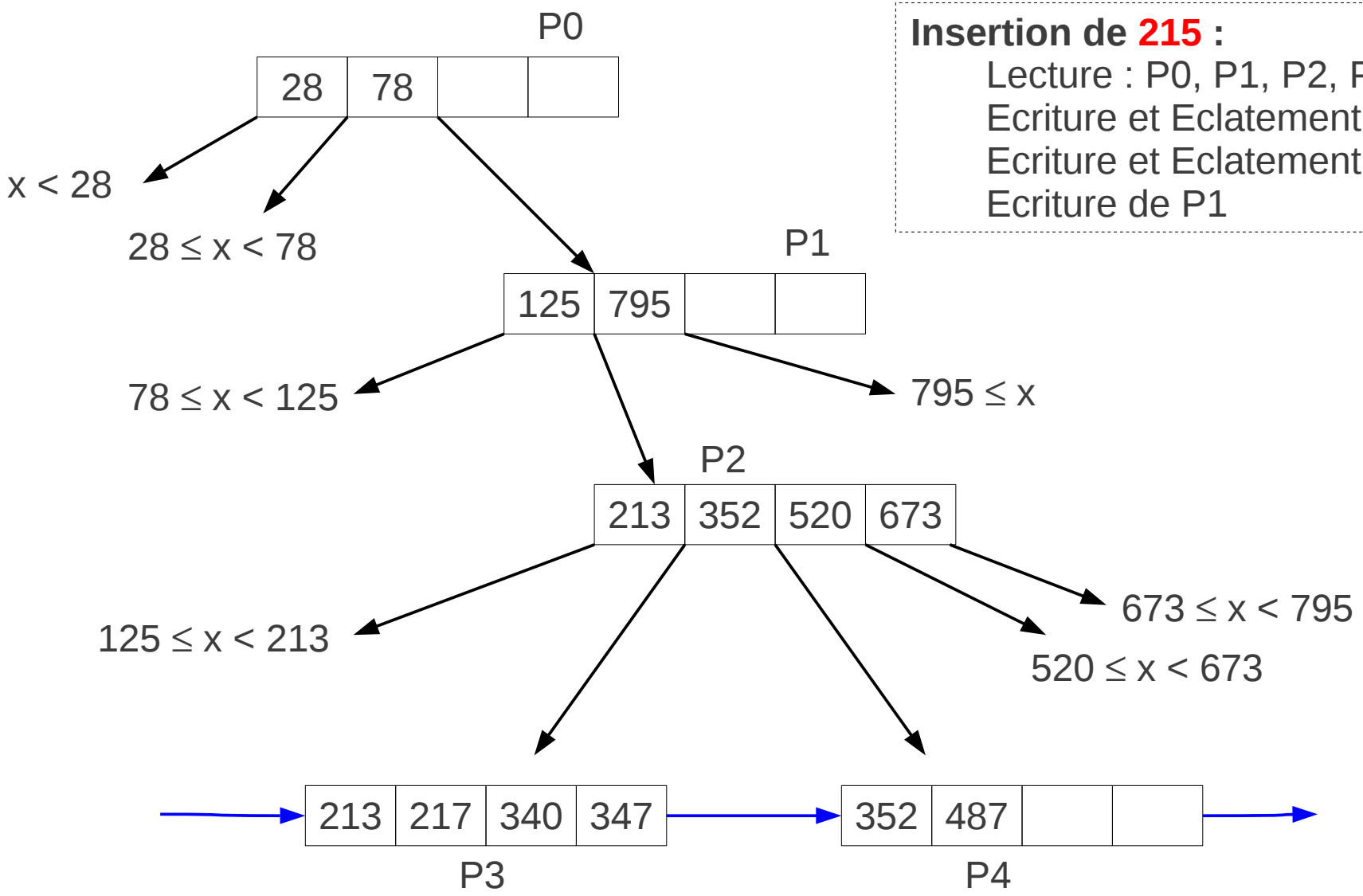
Verrouillage d'index

- Rappel : B^+-Arbres -



Verrouillage d'index

- insertion dans un B$^+$-Arbre -

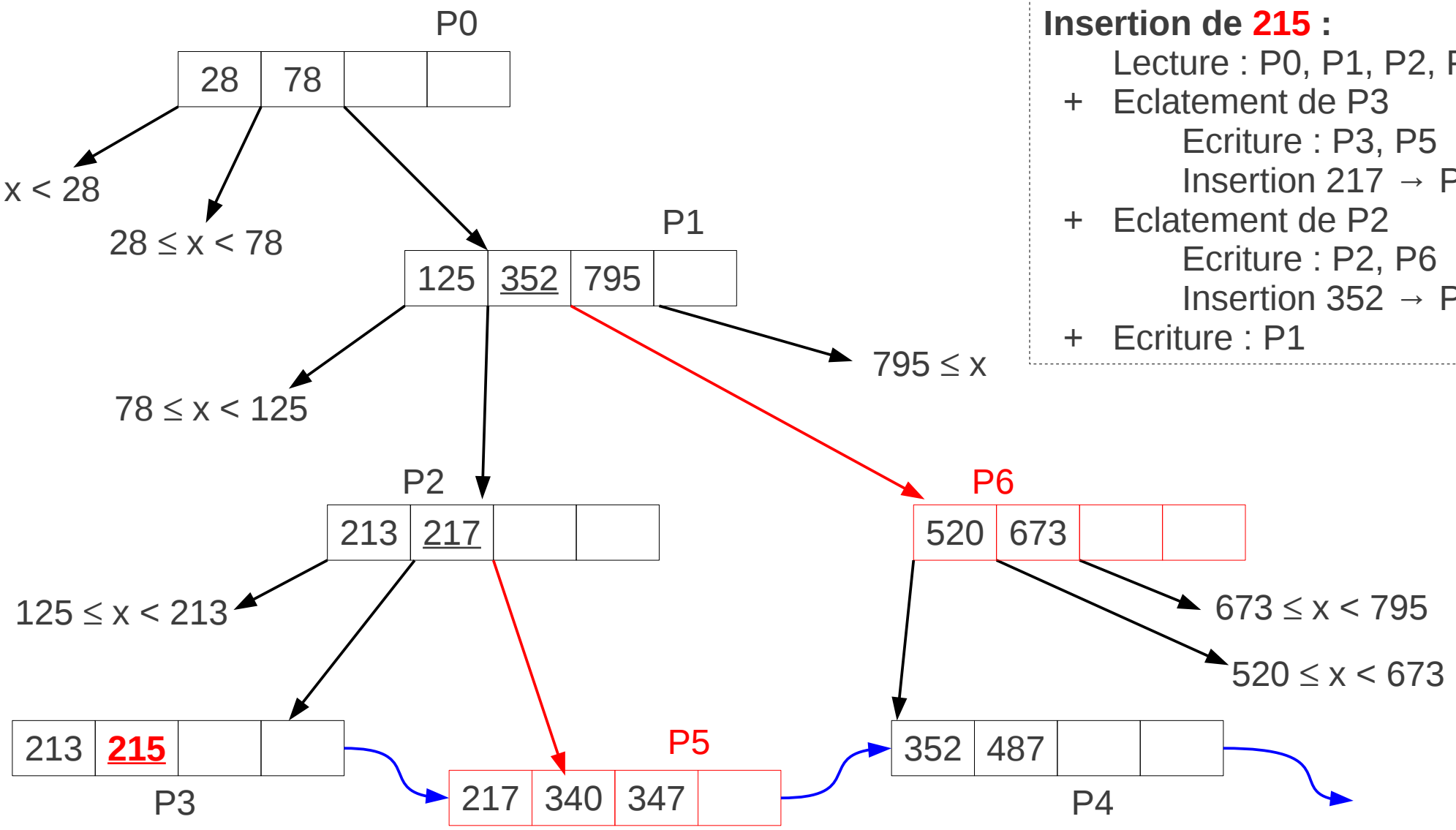


Insertion de 215 :

Lecture : P0, P1, P2, P3
Ecriture et Eclatement de P3
Ecriture et Eclatement de P2
Ecriture de P1

Verrouillage d'index

- insertion (suite) -



Insertion de 215 :

Lecture : P0, P1, P2, P3

\+ Eclatement de P3

Ecriture : P3, P5

Insertion 217 → P2

\+ Eclatement de P2

Ecriture : P2, P6

Insertion 352 → P1

\+ Ecriture : P1

Verrouillage d'index

- Crabbing -

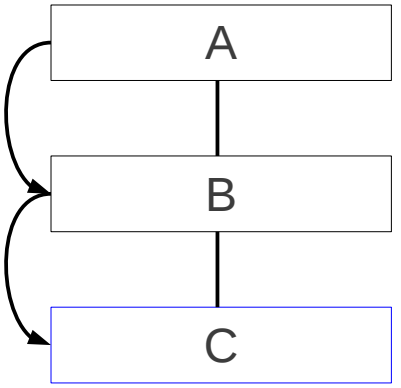
- **Recherche d'une clé**

Lecture des pages d'une branche de l'arbre
Racine → Feuille (avec verrous partagés)

Ne relâcher le verrou sur la page mère, qu'après avoir obtenu un verrou sur la page fille

T pose un verrou **Ir** sur B, avant de relâcher celui sur A

T pose un verrou **Ir** sur C, avant de relâcher celui sur B



- **Insertion d'une clé**

1ere phase : descente en « Crabbing »
Racine → Feuille (avec verrous exclusifs)

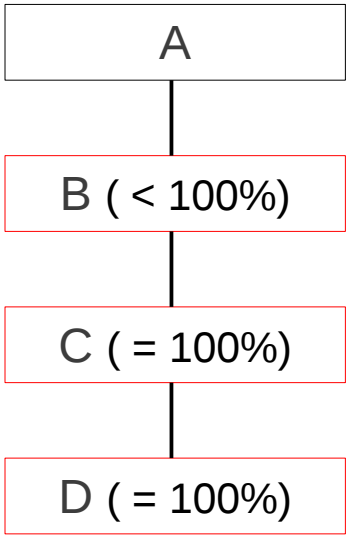
2e phase : remontée dans l'arbre possible, à cause des éclatements (à chaque éclatement, il y a insertion de la clé du milieu dans la page mère)

Donc nécessité de garder les verrous sur toutes les pages, en fin de branche, remplis à 100% plus celle qui précède la première page remplie à 100%

T pose un verrou **Iw** sur B, avant de relâcher celui sur A

T pose un verrou **Iw** sur C, en gardant celui sur B

T pose un verrou **Iw** sur D, en gardant celui sur C



Garder les verrous sur la feuille jusqu'à la fin de la transaction => évite Pb des fantômes

Verrouillage d'index
- Optimisation B-link -

- **Principe**

Permettre aux recherches de pouvoir relâcher les verrous sur une page avant d'obtenir un verrou sur la page fille (=> une recherche peut donc être dépassée par une insertion lors de la traversée d'une branche)

Les pages d'un même niveau dans l'arbre, sont chaînées entre elles (B-link)

Si une insertion dépasse une recherche et provoque un éclatement qui fait migrer l'intervalle cherché par la 1ere transaction dans une nouvelle page, le lien B-link permettra à la transaction qui fait la recherche de pouvoir localiser la bonne page.

Lors de la recherche d'une clé C,

SI C > à la plus grande clé de la page courante P,

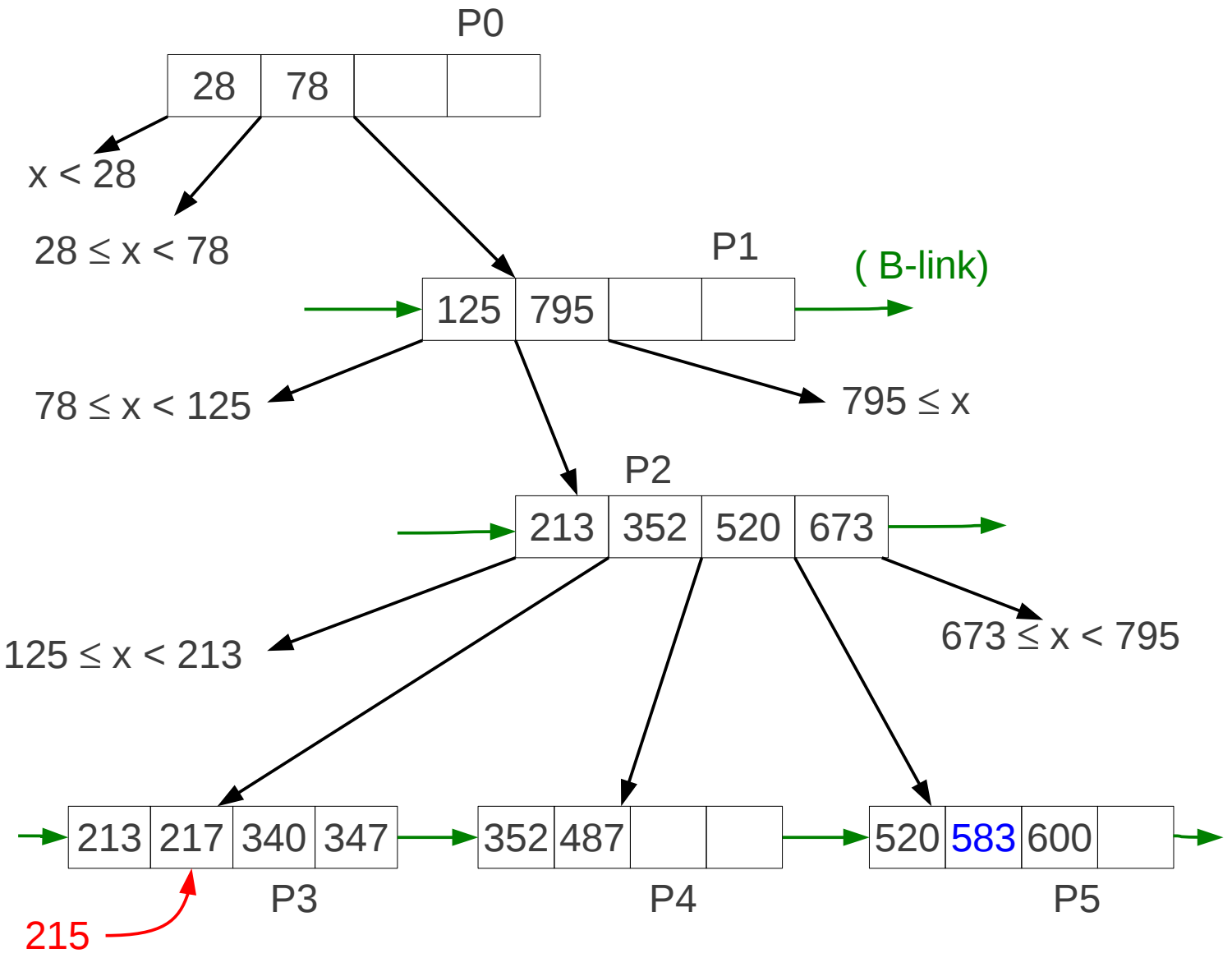
suivre le lien B-link (vers la prochaine page Q dans le même niveau)
pour vérifier si C >= à la 1ere clé de Q,
auquel cas il faut continuer la recherche dans la page Q.

SINON

l'intervalle cherché n'a pas été déplacé (par une transaction d'insertion concurrente)
donc la recherche continue dans la page courante P.

Verrouillage d'index

- Exemple : Optimisation B-link -

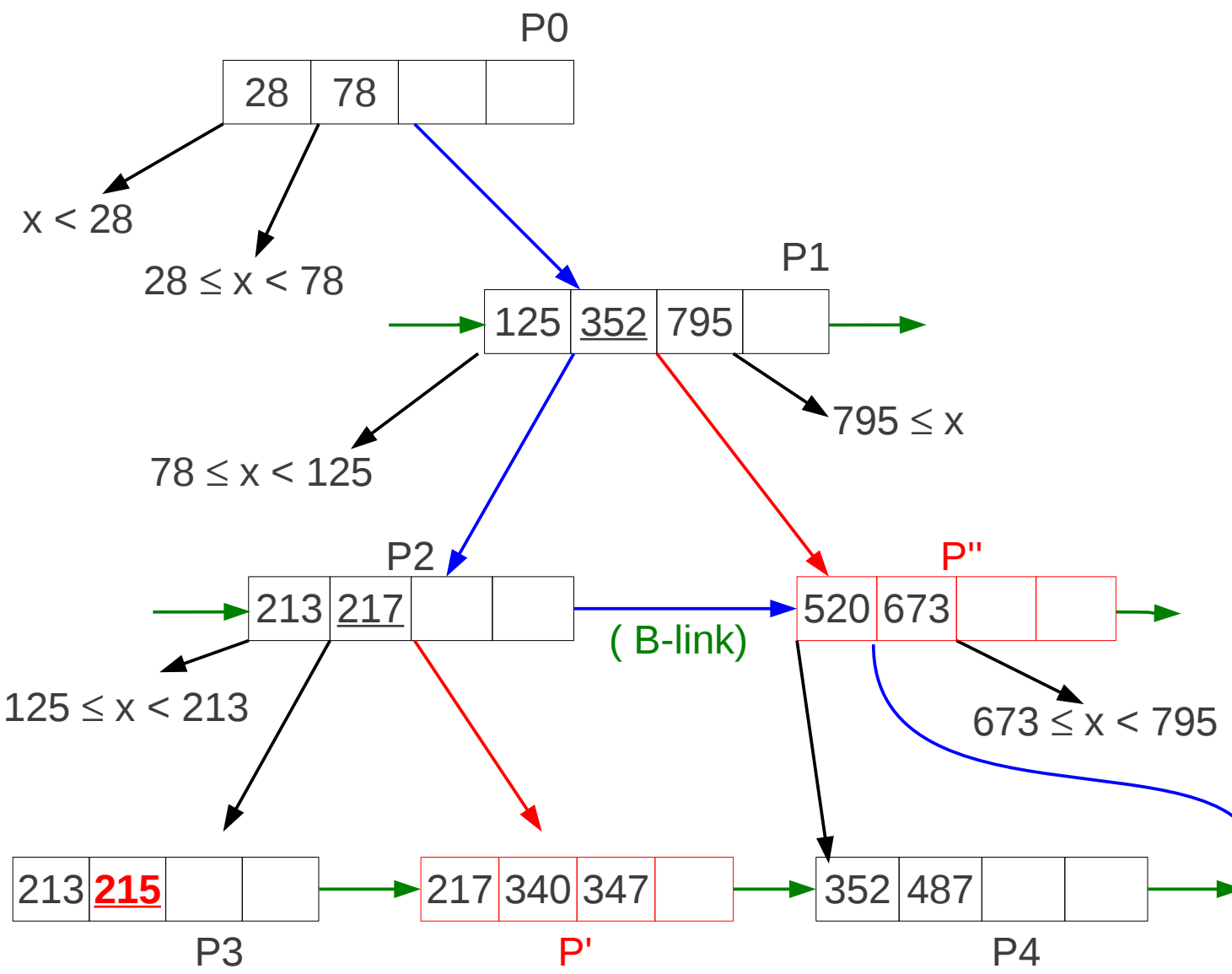


T1 : recherche 583
T2 : insère 215
Soit H un début d'historique possible :

T1	T2
lr1(P0),	
r1(P0),	
ur1(P0),	
	lw2(P0),
	r2(P0),
lr1(P1),	
r1(P1),	
ur1(P1),	
	lw2(P1),
	r2(P1),
	lw2(P2),
	r2(P2),
	lw2(P3),
	r2(P3),
	…

Verrouillage d'index

- Exemple (suite) : optimisation B-link -



T1 (Suite de H) **T2**

w2(P3),
w2(P'),
w2(P2),
w2(P''),
w2(P1),
uw2(all),

lr1(P2),
r1(P2),
// 583 > 217 donc Blink : P''
ur1(P2),
lr1(p''),
r1(P''),
ur1(P''),
lr1(P5),
r1(P5),
ur1(P5)

Verrouillage et Hotspots

- Définition

  Certaines données peuvent être très sollicitées (en MAJ) et deviennent des goulots d'étranglement pour le verrouillage => HOTSPOTS

  Ex : - Les compteurs utilisés dans la gestion des 'serials'
  - La marque de fin de fichier pour l'opération d'insertion

- Quelques techniques utiles pour relativiser les effets des hotspots

  - Garder les données sollicitées en MC
  - Retarder les opérations sur les hotspots juste avant le commit
  - Utiliser des opérations de haut niveau qui ne retournent pas de valeur et n'entrent pas en conflit entre elles
  - Le batching

    ex1 : chaque transaction insère les nouvelles données dans une file privée, qui est périodiquement vidée dans le fichier principal

    ex2 : chaque transaction lui est alloué un intervalle de 'serials'
  - Le partitionnement

    ex : Si x est un hotspots (par exemple le nombre de places dispo dans un avion), on peut maintenir plusieurs données x1, x2, ...xk tel que leur somme = x
    les transactions travaillent de manière aléatoire sur l'un des xi

Verrouillage et Hotspots

- exemple : opérations de haut niveau -

- Dans une BD de gestion de compte, on utilise souvent des opérations d'ajout et de retrait de sommes d'argent sur des comptes
  On définit alors 2 opérations non conflictuelles (car commutatives) :
  Ajouter(x, somme) : ajoute 'somme' au compte x
  Retirer(x, somme) : retire 'somme' du compte x
  => implémentées à l'aides des reads et des writes mais sont exécutée de manière atomique (indivisible)
  Matrice de compatibilité :

	r	w	Aj	Rt
r	V	F	F	F
w	F	F	F	F
Aj	F	F	V	V
Rt	F	F	V	V

Si les transactions n'utilisent que les opérations Aj et Rt, il y aura plus d'opportunité de parallélisme car pas de conflits

TO : Timestamp Ordering

- Estampillage -

- A chaque fois qu'une transaction Ti commence son exécution, le Gestionnaire de Transaction (GT) lui associe une estampille e(Ti) qui va marquer toutes ses opérations
- L'idée est de d'ordonnancer les opérations conflictuelles suivant l'ordre d'entrées des transactions dans le système

  Règle TO:

  Si pi(x) et qj(x) sont conflictuelles, alors l'exécution (par le Gestionnaire de Données - GDR) de pi(x) doit se faire avant celle de qj(x) **ssi** e(Ti) < e(Tj)

- Preuve de serialisabilité d'un historique H vérifiant la règle TO

  1) Si $Ti \rightarrow Tj$ appartient à SG(H) alors il existe pi et qj, 2 op conflictuelles tel que $pi <_H qj$ et d'après la règle du TO on aura $e(Ti) < e(Tj)$

  2) S'il existait un circuit $T1 \rightarrow T2 \rightarrow T3 \ldots \rightarrow T1$ dans SG(H), il en découlerait alors que $e(T1) < e(T1)$ ce qui est une contradiction. Donc SG(H) ne peut pas contenir de circuit et **donc H est sérialisable**.

TO : Timestamp Ordering

- version de base -

A chaque donnée x, le CC garde l'estampille (**max_r(x)**) de la plus jeune transaction l'ayant lu et celle (**max_w(x)**) de la plus jeune transaction l'ayant écrite

Quand une opération $\mathbf{p_i(x)}$ arrive, on teste $e(T_i)$ avec les max_q(x) (pour chaque opération q conflictuelle avec p) pour savoir si $p_i(x)$ est arrivée en retard:

Si $e(T_i) < max\_q(x)$, l'opération est rejetée => T_i annulée

Sinon, $p_i(x)$ est exécutée et max_p(x) reçoit $MAX(max\_p(x) , e(T_i))$

Dans le cas des opérations read et write, on peut utiliser le protocole suivant :

1. cas d'une lecture : $\mathbf{r_i(x)}$

Si $e(T_i) < max\_w(x)$ Alors annuler T_i

Sinon exécuter $r_i(x)$; $max\_r(x) \leftarrow max(e(T_i), max\_r(x))$

2. cas d'une écriture : $\mathbf{w_i(x)}$

Si $e(T_i) < max\_r(x)$ Alors annuler T_i

Sinon Si $e(T_i) < max\_w(x)$ Alors ignorer $w_i(x)$ (règle de Thomas)

Sinon exécuter $w_i(x)$; $max\_w(x) \leftarrow e(T_i)$

TO : Timestamp Ordering

- Exemples -

- Supposons e(T1) < e(T2)

  H1 = {r1(x), r2(x), w2(x), r1(y), r2(y), c1, w2(y), c2}

  H1 est conforme à TO et est sérialisable

- Supposons e(T1) > e(T2)

  H2 = {r2(x), w2(x), r1(x), r1(y), c1, r2(y), a2(car w2(y) est arrivé)}

  H2 est aussi conforme à TO mais n'est pas recouvrable.

TO : Timestamp Ordering

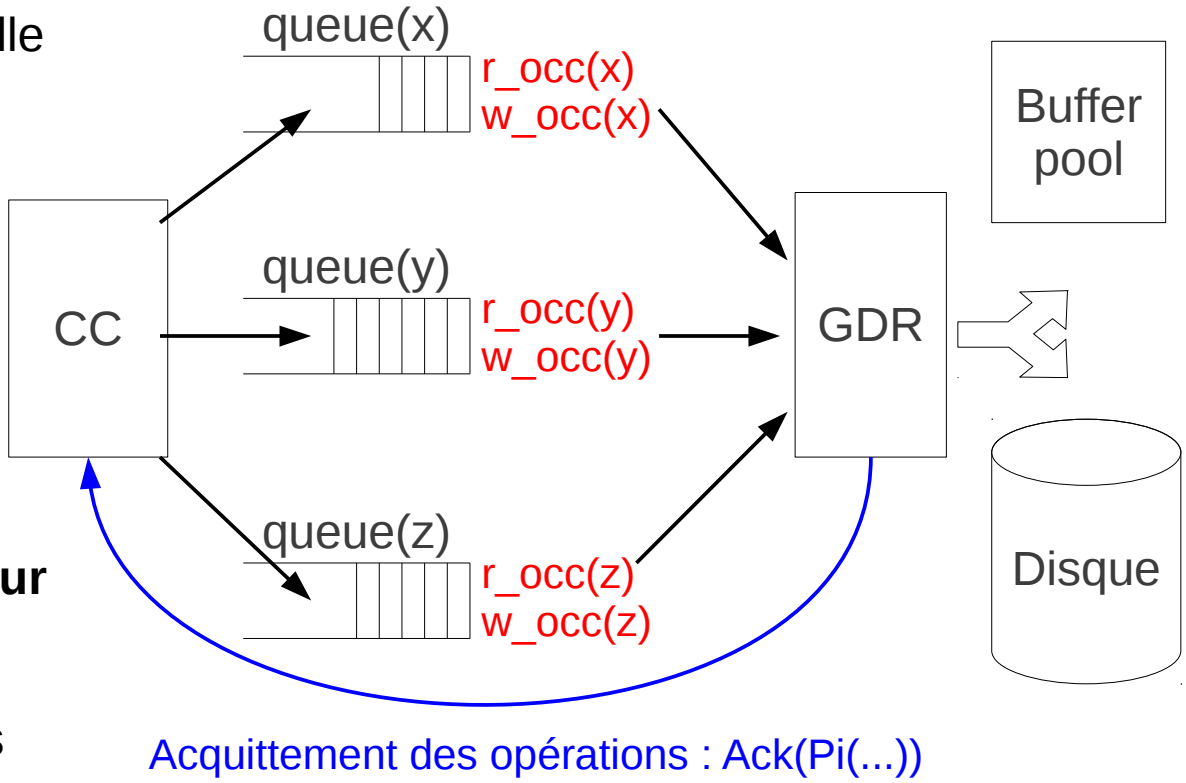
- Détails internes -

- Des files d'attentes (**queue(x)**) sont maintenues entre le CC et GDR pour faire patienter les opérations (acceptées) devant attendre la terminaison de leurs (éventuelles) opérations conflictuelles déjà en cours d'exécution par le GDR.

Quand le CC veut exécuter Pi(x), il l'envoie directement au GDR si tous les **compteurs** q_occ(x) (pour q conflictuelle avec p) sont à 0.
Sinon Pi(x) est enfilée dans queue(x)

A chaque fois qu'une opération Pi(x) est envoyée au GDR, on incrémente le **compteur** p_occ(x)

Quand Ack(Pi(x)) est envoyé du GDR vers le CC : on décrémente le **compteur** p_occ(x)
S'il devient nul : on défile toutes les opérations qj(x) qui peuvent l'être dans queue(x), pour les envoyer au GDR.



TO : Timestamp Ordering
- Version Stricte -

- Pour avoir une exécution **stricte** (donc recouvrable et évitant les annulations en cascade aussi), il suffit de retarder les lectures (ri(x)) et les écritures (wi(x)) dans queue(x), jusqu'à l'acquittement de la terminaison (Ack(cj) ou Ack(aj)) de la transaction Tj, encore active, ayant une opération d'écriture (wj(x)) en cours d'exécution au niveau du GDR ou alors déjà acquittée .
- Implémentation :
  - Lorsqu'une opération Wj(x) est envoyée au GDR, on positionne w_occ(x) à 1 et on garde la trace de l'identifiant Tj de la transaction et le nom de la file d'attente (x)
  - Lors de la réception d'un acquittement d'écriture (Ack(Wj(x)) on ne décrémente pas le compteur w_occ(x)
  - Lors de la réception d'un acquittement de terminaison (Ack(Cj) ou Ack(Aj)), on décrémente w_occ(x)

SGT : Serialization Graph Testing

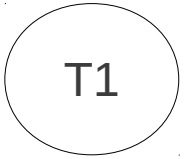
- Le CC construit un graphe de sérialisation particulier (où les nœuds peuvent être des transactions encore actives)
- Lors de la réception d'une opération Pi(x), le CC ajoute un nœud Ti (s'il n'existe pas déjà) et rajoute des arcs Tj → Ti pour chaque opération Qj(x) déjà exécutée et conflictuelle avec Pi(x)
  - Si le nouveau graphe contient un circuit, l'opération est rejetée et Ti est annulée (le nœud est supprimé du graphe)
  - Sinon, Pi(x) est acceptée et envoyée au GDR à travers la file queue(x), pour être exécutée

Exemple : r1(x), w3(y), r2(y), r2(z), w3(x), w1(z)

SGT : Serialization Graph Testing

- Le CC construit un graphe de sérialisation particulier (où les nœuds peuvent être des transactions encore actives)
- Lors de la réception d'une opération Pi(x), le CC ajoute un nœud Ti (s'il n'existe pas déjà) et rajoute des arcs Tj → Ti pour chaque opération Qj(x) déjà exécutée et conflictuelle avec Pi(x)
  - Si le nouveau graphe contient un circuit, l'opération est rejetée et Ti est annulée (le nœud est supprimé du graphe)
  - Sinon, Pi(x) est acceptée et envoyée au GDR à travers la file queue(x), pour être exécutée

Exemple : r1(x), w3(y), r2(y), r2(z), w3(x), w1(z)



SGT : Serialization Graph Testing

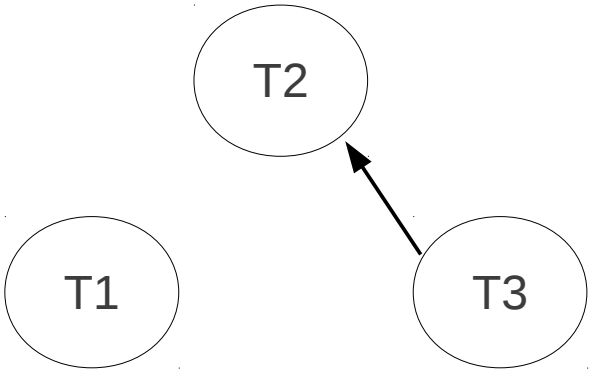
- Le CC construit un graphe de sérialisation particulier (où les nœuds peuvent être des transactions encore actives)
- Lors de la réception d'une opération Pi(x), le CC ajoute un nœud Ti (s'il n'existe pas déjà) et rajoute des arcs Tj → Ti pour chaque opération Qj(x) déjà exécutée et conflictuelle avec Pi(x)
  - Si le nouveau graphe contient un circuit, l'opération est rejetée et Ti est annulée (le nœud est supprimé du graphe)
  - Sinon, Pi(x) est acceptée et envoyée au GDR à travers la file queue(x), pour être exécutée

Exemple : r1(x), w3(y), r2(y), r2(z), w3(x), w1(z)

SGT : Serialization Graph Testing

- Le CC construit un graphe de sérialisation particulier (où les nœuds peuvent être des transactions encore actives)
- Lors de la réception d'une opération Pi(x), le CC ajoute un nœud Ti (s'il n'existe pas déjà) et rajoute des arcs Tj → Ti pour chaque opération Qj(x) déjà exécutée et conflictuelle avec Pi(x)
  - Si le nouveau graphe contient un circuit, l'opération est rejetée et Ti est annulée (le nœud est supprimé du graphe)
  - Sinon, Pi(x) est acceptée et envoyée au GDR à travers la file queue(x), pour être exécutée

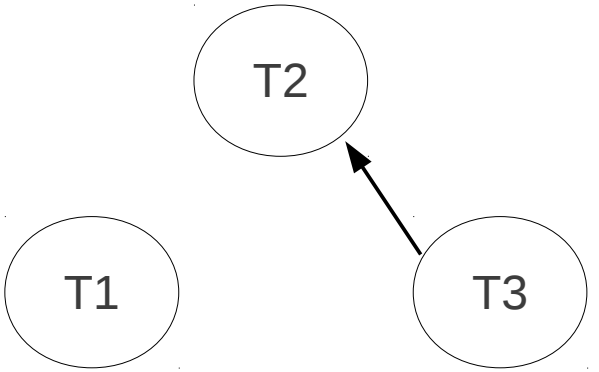
Exemple : r1(x), w3(y), r2(y), r2(z), w3(x), w1(z)



SGT : Serialization Graph Testing

- Le CC construit un graphe de sérialisation particulier (où les nœuds peuvent être des transactions encore actives)
- Lors de la réception d'une opération Pi(x), le CC ajoute un nœud Ti (s'il n'existe pas déjà) et rajoute des arcs Tj → Ti pour chaque opération Qj(x) déjà exécutée et conflictuelle avec Pi(x)
  - Si le nouveau graphe contient un circuit, l'opération est rejetée et Ti est annulée (le nœud est supprimé du graphe)
  - Sinon, Pi(x) est acceptée et envoyée au GDR à travers la file queue(x), pour être exécutée

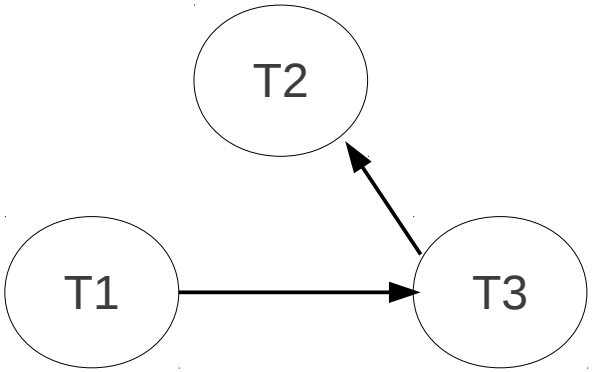
Exemple : r1(x), w3(y), r2(y), r2(z), w3(x), w1(z)



SGT : Serialization Graph Testing

- Le CC construit un graphe de sérialisation particulier (où les nœuds peuvent être des transactions encore actives)
- Lors de la réception d'une opération Pi(x), le CC ajoute un nœud Ti (s'il n'existe pas déjà) et rajoute des arcs Tj → Ti pour chaque opération Qj(x) déjà exécutée et conflictuelle avec Pi(x)
  - Si le nouveau graphe contient un circuit, l'opération est rejetée et Ti est annulée (le nœud est supprimé du graphe)
  - Sinon, Pi(x) est acceptée et envoyée au GDR à travers la file queue(x), pour être exécutée

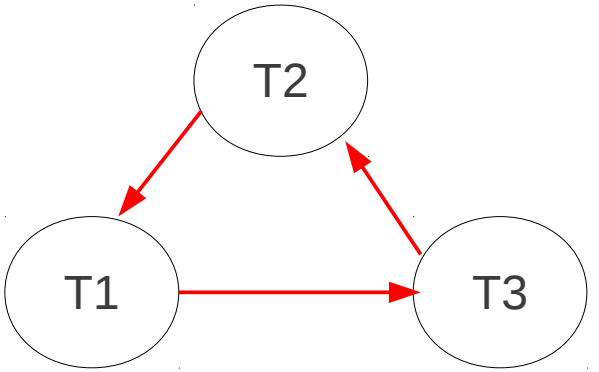
Exemple : r1(x), w3(y), r2(y), r2(z), w3(x), w1(z)



SGT : Serialization Graph Testing

- Le CC construit un graphe de sérialisation particulier (où les nœuds peuvent être des transactions encore actives)
- Lors de la réception d'une opération Pi(x), le CC ajoute un nœud Ti (s'il n'existe pas déjà) et rajoute des arcs Tj → Ti pour chaque opération Qj(x) déjà exécutée et conflictuelle avec Pi(x)
  - Si le nouveau graphe contient un circuit, l'opération est rejetée et Ti est annulée (le nœud est supprimé du graphe)
  - Sinon, Pi(x) est acceptée et envoyée au GDR à travers la file queue(x), pour être exécutée

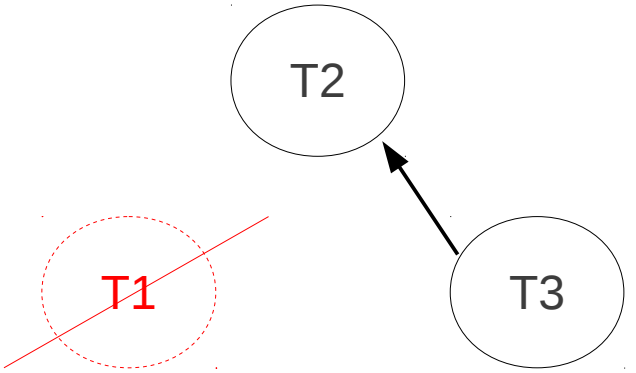
Exemple : r1(x), w3(y), r2(y), r2(z), w3(x), w1(z)



SGT : Serialization Graph Testing

- Le CC construit un graphe de sérialisation particulier (où les nœuds peuvent être des transactions encore actives)
- Lors de la réception d'une opération Pi(x), le CC ajoute un nœud Ti (s'il n'existe pas déjà) et rajoute des arcs Tj → Ti pour chaque opération Qj(x) déjà exécutée et conflictuelle avec Pi(x)
  - Si le nouveau graphe contient un circuit, l'opération est rejetée et Ti est annulée (le nœud est supprimé du graphe)
  - Sinon, Pi(x) est acceptée et envoyée au GDR à travers la file queue(x), pour être exécutée

Exemple : ~~r1(x)~~, w3(y), r2(y), r2(z), w3(x), a1



SGT : Serialization Graph Testing

- Détails internes -

- Pour **chaque transaction Tj présente dans le graphe**, le CC maintient deux ensembles de données (x, y, z, ...)
  - **r[j]** : l'ensemble de toutes les données lues par Tj (rj(x) déjà exécutée)
  - **w[j]** : l'ensemble de toutes les données écrites par Tj (wj(x) déjà exécutée)

- Pour **rajouter de nouveaux arcs Tj → Ti**,
  (lors de la réception d'une opération Pi(x)),
  le CC vérifie si x ∈ à l'un des ensembles **Q[j]** (Q conflictuelle avec P)
  de chaque transaction Tj du graphe

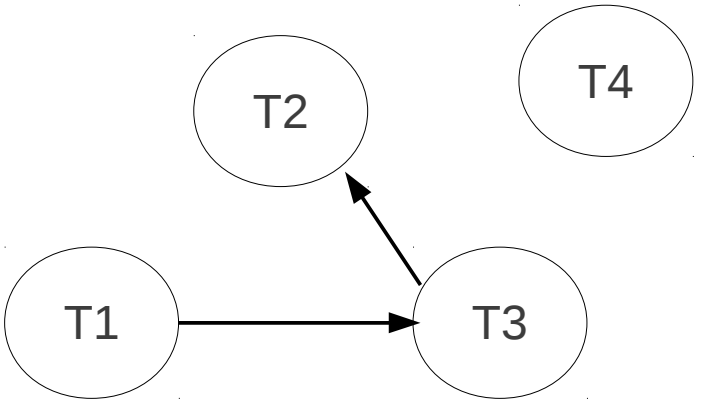
SGT : Serialization Graph Testing

- Nettoyage du graphe -

- Tant qu'une transaction Tj est active, de nouveaux arcs entrants peuvent être rajoutés
- Pour une transaction validée, il ne peut plus y avoir de nouveaux arcs entrants (car elle n'envoie plus de nouvelles opérations)
  - Par contre de nouveaux arcs sortants sont toujours possibles
- Dans un circuit, tout sommet à au moins un arc entrant et un arc sortant, donc :
  - On peut enlever (supprimer) un sommet Tj uniquement lorsque la transaction a validé et n'a plus d'arcs entrants

Exemple : r1(x), w3(x), r4(y), r2(x), c3, w2(x), r4(x), c1, ...

Le sommet T3 n'est pas supprimé à ce moment

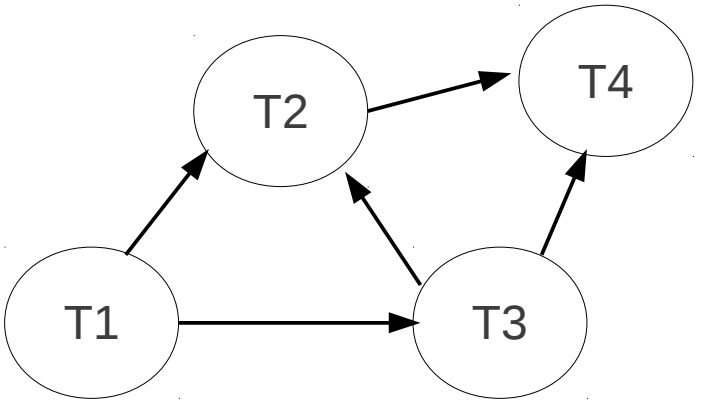


SGT : Serialization Graph Testing

- Nettoyage du graphe -

- Tant qu'une transaction Tj est active, de nouveaux arcs entrants peuvent être rajoutés
- Pour une transaction validée, il ne peut plus y avoir de nouveaux arcs entrants (car elle n'envoie plus de nouvelles opérations)
  - Par contre de nouveaux arcs sortants sont toujours possibles
- Dans un circuit, tout sommet à au moins un arc entrant et un arc sortant, donc :
  - On peut enlever (supprimer) un sommet Tj uniquement lorsque la transaction a validé et n'a plus d'arcs entrants

Exemple : r1(x), w3(x), r4(y), r2(x), c3, w2(x), r4(x), c1, ...



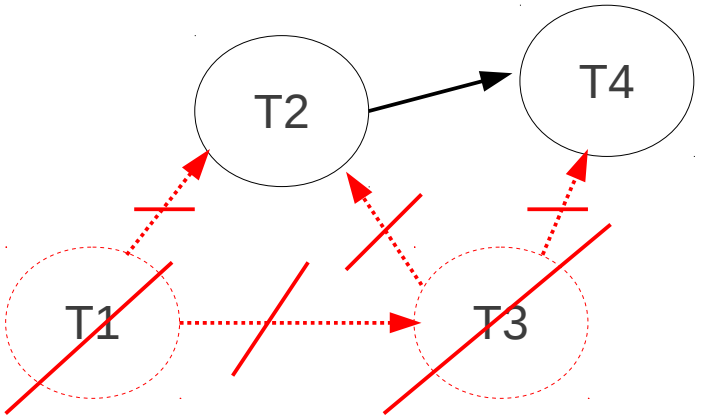
SGT : Serialization Graph Testing

- Nettoyage du graphe -

- Tant qu'une transaction Tj est active, de nouveaux arcs entrants peuvent être rajoutés
- Pour une transaction validée, il ne peut plus y avoir de nouveaux arcs entrants (car elle n'envoie plus de nouvelles opérations)
  - Par contre de nouveaux arcs sortants sont toujours possibles
- Dans un circuit, tout sommet à au moins un arc entrant et un arc sortant, donc :
  - On peut enlever (supprimer) un sommet Tj uniquement lorsque la transaction a validé et n'a plus d'arcs entrants

Exemple : r1(x), w3(x), r4(y), r2(x), c3, w2(x), r4(x), c1, ...

Suppression de T1 puis de T3



Protocoles Mutliversions

- **Idée de base :**

  Maintenir plusieurs versions (x_1, x_2, … x_n) d'une même donnée x

  $w_i(x)$ est traduite en $w_i(x_i)$

  → l'écriture produit une nouvelle version x_i

  $r_i(x)$ est traduite en $r_i(x_j)$

  → la lecture retourne une des versions existantes x_j

- **Les versions ne sont pas visibles aux clients**
  - Le système se comporte comme si la BD ne contenait qu'une seule version (mono-version)
- **Généralisation**
  - On peut construire des contrôleurs de concurrence multiversions à base des protocoles mono-versions

    MVTO, MV2PL, MVSGT, MVTO+2PL, …

TO Mutliversion (MVTO)

A chaque version xk, est associé un intervalle [ew, er]

ew : estampille de la transaction ayant générée xk,

er : estampille de la plus jeune transaction ayant lu xk

Lors d'une opération de lecture **ri(x)**, le CC :

- choisit la version xk ayant la plus grande estampille er telle que **ew <= e(Ti)**
- envoie ri(xk) au GDR (pour son traitement) et
  met à jour er[xk] ← max(er[xk] , e(Ti))

Lors d'une opération d'écriture **wi(x)** :

Si le CC a déjà exécuté un rj(xk) tel que : **e(Tk) < e(Ti) < e(Tj)**

Alors **Ti est annulée** (wi est arrivée trop en retard)

Sinon wi(xi) est envoyée au GDR

une nouvelle version xi est produite avec l'intervalle initial [et(Ti), e(Ti)]

MVTO peut être recouvrable

Si le CC retarde l'exécution du ci jusqu'à ce que tous les cj soient acquittés, pour chaque Tj ayant produit une version (xj) lu par Ti (c-a-d ri(xj))

2PL Mutliversion (MV2PL)

On maintient **au maximum 2 versions** (une ancienne version validée et une nouvelle version en cours de production)

wi(x) : pose un verrou lw(x) et génère une nouvelle version xi

ri(x) : pose un verrou lr(x) et accède à l'unique (ancienne) version validée

A la fin de la transaction une phase de « **certification** » est nécessaire pour remplacer la version validée

Table de compatibilité :

	r	w	cert
r	V	V	F
w	V	F	F
cert	F	F	F

Phase de certification : Cela consiste à transformer les verrous lw en lcert, donc d'attendre la terminaison de toutes les transactions ayant lu la donnée, avant de la remplacer par la nouvelle version

Après la phase de certification, l'ancienne version est supprimée.

Combinaison MVTO+2PL

Pour chaque donnée x, il peut exister plusieurs versions (x_1, x_2, … x_n)

Le CC gère une **liste de transactions validées** L

On considère **2 types de transactions** (readers et updaters)

Les readers n'utilisent pas de verrou, elles se basent uniquement sur la liste L pour connaître la bonne version à lire (la version la plus récente produite par une transaction de cette liste)

Les updaters utilisent 2PL

Quand une transaction T (readers) commence, le CC lui associe la liste des transactions validées à ce moment. T garde la même liste inchangée jusqu'à sa terminaison.
Pour **lire x**, on utilise cette liste pour récupérer **la version la plus récente** de x produite par une transaction de cette liste

==> Souvent appelé '**Snapshot Mode**'

Snapshot Isolation

Les readers n'utilisent pas de verrou, elles se basent uniquement sur la liste L des transactions déjà validées, pour connaître la bonne version à lire

Les updaters utilisent le principe du First Committer Win

Les READs utilisent la liste L pour choisir la bonne version à lire
Les WRITEs génèrent de nouvelles versions (wi(xi))
Au moment de la validation, on vérifie s'il y a eu un conflit (existence d'un wj(xj) déjà validé), auquel cas, Ti est annulée.

Ce protocole peut générer des historiques non sérialisables :

T1 = {r1(x), w1(y), c1 } T2 = {r2(y), w2(x), c2 }

Si au départ x=10 et y=20,
l'exécution en série T1;T2 produira x=10 et y=10
l'exécution en série T2;T1 produira x=20 et y=20

L'historique multiversion H={ r1(x0), r2(y0), w1(y1), w2(x2), c1, c2 } est conforme à Snapshot Isolation, mais l'état produit (x=20, y=10) est incorrect !

Snapshot Isolation

Les readers n'utilisent pas de verrou, elles se basent uniquement sur la liste L des transactions déjà validées, pour connaître la bonne version à lire

Les updaters utilisent le principe du First Committer Win

Les READs utilisent la liste L pour choisir la bonne version à lire
Les WRITEs génèrent de nouvelles versions (wi(xi))
Au moment de la validation, on vérifie s'il y a eu un conflit (existence d'un wj(xj) déjà validé), auquel cas, Ti est annulée.

Snapshot Isolation

Les readers n'utilisent pas de verrou, elles se basent uniquement sur la liste L des transactions déjà validées, pour connaître la bonne version à lire

Les updaters utilisent le principe du First Committer Win

Les READs utilisent la liste L pour choisir la bonne version à lire
Les WRITEs génèrent de nouvelles versions (wi(xi))
Au moment de la validation, on vérifie s'il y a eu un conflit (existence d'un wj(xj) déjà validé), auquel cas, Ti est annulée.