

Contrôle de concurrence

Les problèmes de concurrence

Degrés d'isolation dans SQL

Exécutions et sérialisabilité

Contrôle de concurrence

Améliorations

Contrôle de concurrence

Objectif : *synchroniser les transactions concurrentes* afin de **maintenir la cohérence** de la BD, tout en **maximisant le degré de concurrence**

Principes:

- exécution simultanée des transactions pour des raisons de performance
 - par ex., exécuter les opérations d'une autre transaction quand la première commence à faire des accès disques
- les résultats doivent être équivalents à des exécutions non simultanées (isolation)
 - besoin de *raisonner sur l'ordre d'exécution* des transactions

Problèmes de concurrence

Perte d'écritures :

- $[T1: \text{Write } a1 \rightarrow A; T2: \text{Write } b2 \rightarrow B; T1: \text{Write } b1 \rightarrow B; T2: \text{Write } a2 \rightarrow A;]$
- $A=a2, B=b1$: on perd une écriture de T1 et une écriture de T2

Introduction d'incohérence :

- Contrainte : $A = B$
- $[T1: A*2 \rightarrow A; T2: A+1 \rightarrow A; T2: B+1 \rightarrow B; T1: B*2 \rightarrow B;]$
- $A = (2*A)+1, B=2*(B+1)$

Non reproductibilité des lectures :

- $[T1: \text{Read } A; T2: \text{Write } b2 \rightarrow A; T1: \text{Read } A;]$
- Si $b2$ est différente de la valeur initiale de A , alors T1 lit deux valeurs différentes.

Degrés d'isolation SQL-92

Lecture sale (lecture d'une maj non validées):

- T1: Write(A); T2: Read(A); T1: abort
- T_1 modifie x qui est lu ensuite par T_2 avant la fin (validation, annulation) de T_1
- Si T_2 annule, T_1 a lu des données qui n'existent pas dans la base de données

Lecture non-répétable (maj intercalée) :

- T1: Read(A); T2: Write(A); T2: commit; T1: Read(A);
- T_1 lit x ; T_2 modifie ou détruit x et valide
- Si T_1 lit x à nouveau et obtient *résultat différent*

Fantômes (requête + insertion) :

- T1: Select where R.A=...; T2: Insert Into R(A) Values (...);
- T_1 fait une recherche Q avec un prédicat tandis que T_2 insère de nouveaux n -uplets (fantômes) qui satisfont le prédicat.
- Les *insertions* ne sont pas détectées comme concurrentes pendant l'évaluation de la requête (résultat incohérent).

Degrés d'isolation SQL-92

haut					bas
	↑				↓
degré de concurrence					degré d'isolation
		Degré	Lecture sale	Lectures non répétable	Fantômes
		READ_UNCOMMITTED	possible	possible	possible
		READ_COMMITTED	impossible	possible	possible
		REPEATABLE_READ	impossible	impossible	possible
		SERIALIZABLE	impossible	impossible	impossible
bas					haut

Exécution (où Histoire)

- **Exécution** : ordonnancement des opérations d'un ensemble de transactions
- Ordre *total* (séquence) ou *partiel* (arbre)

T_1 : Read(x)
Write(x)
Commit

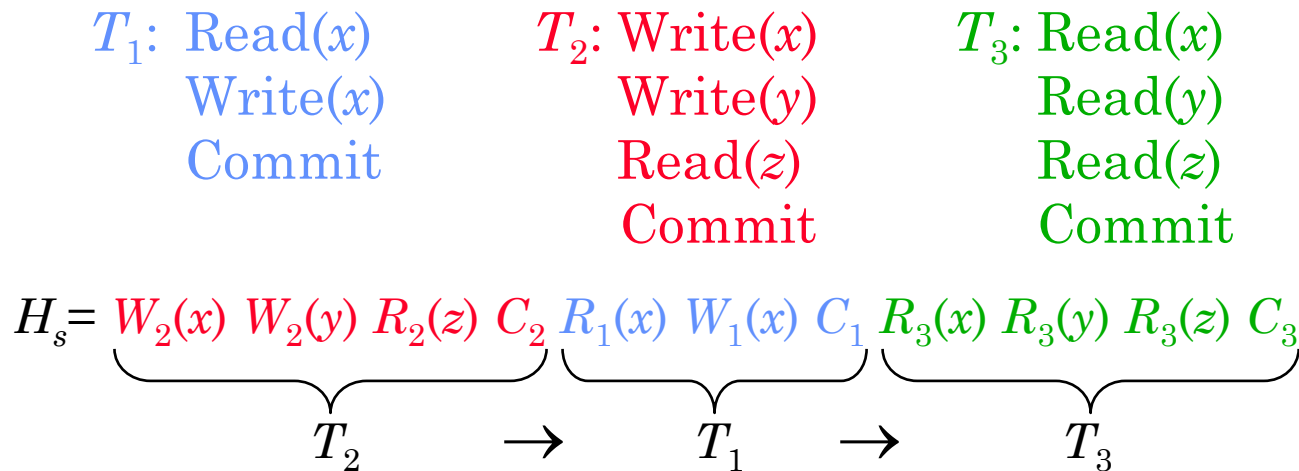
T_2 : Write(x)
Write(y)
Read(z)
Commit

T_3 : Read(x)
Read(y)
Read(z)
Commit

$H_1 = W_2(x) R_1(x) R_3(x) W_1(x) C_1 W_2(y) R_3(y) R_2(z) C_2 R_3(z) C_3$

Exécution en série

- **Exécution en série** : histoire où il n'y a pas d'entrelacement des opérations de transactions
- **Hypothèse** : chaque transaction est *localement* cohérente
- Si la BD est cohérente avant l'exécution des transactions, alors elle sera également cohérente après leur exécution en série.



Exécution sérialisable

Opérations conflictuelles : deux opérations sont en *conflit* si elles accèdent le même granule et *une des deux opérations est une écriture*.

Exécutions équivalentes : deux exécutions H1 et H2 d'un ensemble de transactions sont *équivalentes (de conflit)* si

- l'ordre des opérations de chaque transaction et
- l'ordre des opérations conflictuelles (validées)

sont identiques dans H1 et H2.

Exécution sérialisable : exécution où il existe *au moins une* exécution en série (ou *sérielle*) équivalente (de conflit).

Exécutions équivalentes et sérialisables

T_1 : Read(x)
Write(x)
Commit


T_2 : Write(x)
Write(y)
Read(z)
Commit

T_3 : Read(x)
Read(y)
Read(z)
Commit

Les exécutions suivantes ne sont pas équivalentes :

$H_1 = W_2(x) R_1(x) R_3(x) W_1(x) C_1 R_3(y) W_2(y) R_2(z) C_2 R_3(z) C_3$

$H_2 = W_2(x) R_1(x) W_1(x) C_1 R_3(x) W_2(y) R_3(y) R_2(z) C_2 R_3(z) C_3$



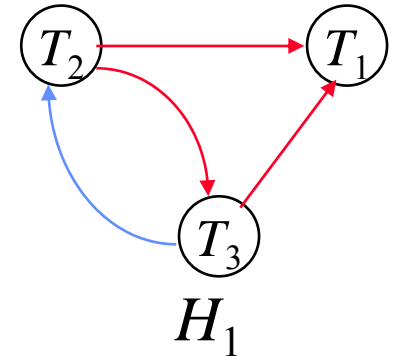
H_2 est équivalente à H_s , qui est sérielle $\Rightarrow H_2$ est *sérialisable* :

$H_s = W_2(x) W_2(y) R_2(z) C_2 R_1(x) W_1(x) C_1 R_3(x) R_3(y) R_3(z) C_3$

Graphe de précédence (GP)

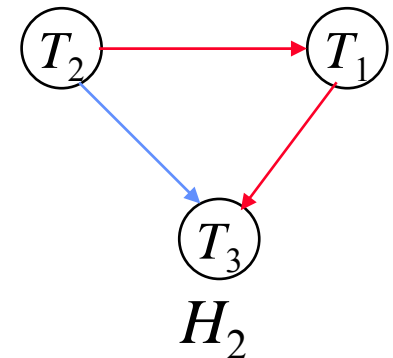
Graphe de précédence $GP_H = \{V, P\}$ pour l'exécution H :

- $V = \{T \mid T \text{ est une transaction validée dans } H\}$
- $P = \{T_i \rightarrow T_k \text{ si } o_{ij} \in T_i \text{ et } o_{kl} \in T_k \text{ sont en conflit et } o_{ij} <_H o_{kl}\}$



$H_1 = W_2(x) \ R_1(x) \ R_3(x) \ W_1(x) \ C_1 \ R_3(y) \ R_2(z) \ W_2(y) \ C_2 \ R_3(z) \ C_3$

$H_2 = W_2(x) \ R_1(x) \ W_1(x) \ C_1 \ R_3(x) \ W_2(y) \ R_3(y) \ R_2(z) \ C_2 \ R_3(z) \ C_3$



Théorème: l'exécution H est sérialisable ssi GP_H ne contient pas de cycle (facile à prouver).

Algorithmes de contrôle de concurrence

- Verrouillage à deux-phases (2PL)
- Estampillage

Algorithmes de verrouillage

Les transactions font des demandes de verrous à un *gérant de verrous* :

- **verrous en lecture** (*vl*), appelés aussi **verrous partagés**
- **verrous en écriture** (*ve*), appelés aussi **verrous exclusifs**

Compatibilité (de verrous sur le même granule) :

	<i>vl</i>	<i>ve</i>
<i>vl</i>	Oui	Non
<i>ve</i>	Non	Non

Algorithme Lock

```
Bool Function Lock (Transaction t, Granule G, Verrou V) {  
  /* retourne vrai si t peut poser le verrou V sur G et faux sinon (t doit attendre) */  
  Cverrous := {};  
  Pour chaque transaction  $i \neq t$  ayant verrouillé le granule G faire {  
    Cverrous := Cverrous  $\cup$  i.verrous(G) } ; // cumuler les verrous sur G  
  }  
  si Compatible(V, Cverrou) alors {  
    t.verrous(G) := t.verrous(G)  $\cup$  V; // marquer l'objet verrouillé  
    return true ;  
  } sinon {  
    insérer le couple (t, V) dans la liste d'attente de G ;  
    bloquer la transaction t ;  
    return false ;  
  }  
}
```

Exemple Lock

t	G	V	t.Verrou(G)	G.attente
t1	a	vl	{ vl }	{ }
t2	a	vl	{ vl }	{ }
t1	b	vl	{ vl }	{ }
t1	a	ve	{ vl }	{ (t1,ve) }
t2	a	ve	{ vl }	{ (t1,ve), (t2, ve) }
t3	b	vl	{ vl }	{ }
t3	b	ve	{ vl }	{ (t3,ve) }

Algorithme Unlock

Procédure **Unlock**(Transaction t , Granule G) {

 /* t libère tous les verrous sur G et redémarre les transactions en attente (si possible) */

$t.\text{verrou}(G) := \{\}$;

Pour chaque couple (t, V) *dans la liste d'attente de G* **faire** {

si **Lock**(t, G, V) **alors** {

enlever (t, V) de la liste d'attente de G ;

débloquer la transaction t ;

 }

 }

}

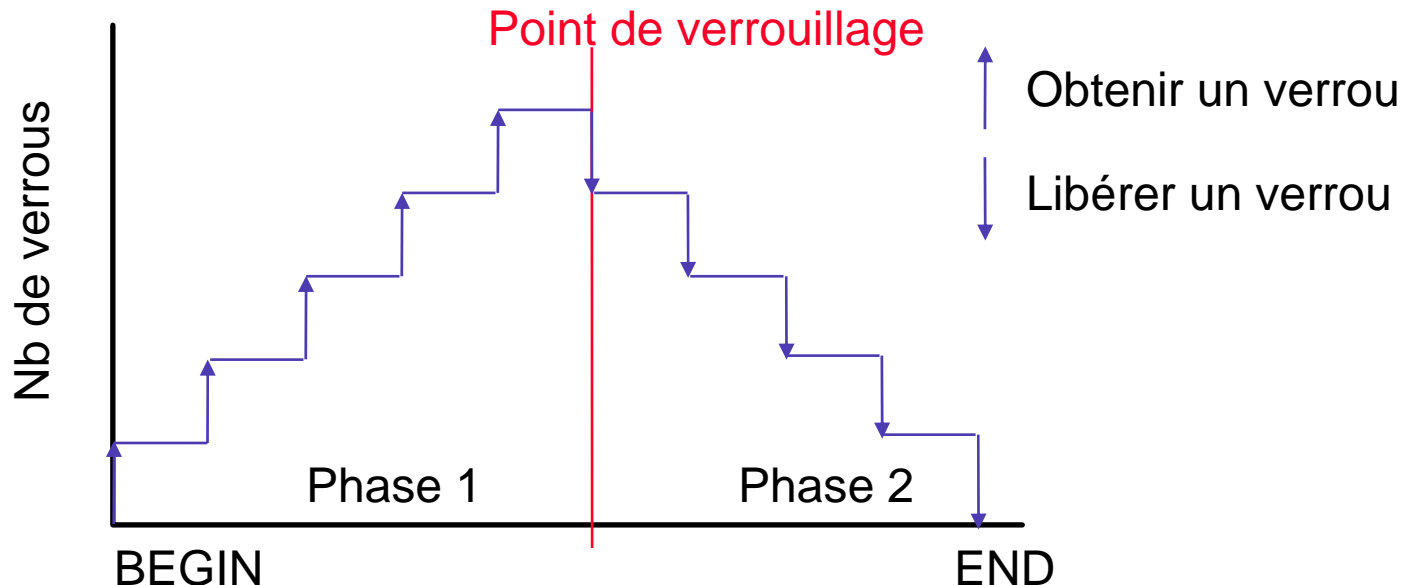
Exemple Unlock

t	G	V	t.Verrou(G)	G.attente
...
t2	a	ve	{vl,ve}	{(t1,ve)}
t3	b	vl	{vl}	{}
t2	b	vl	{vl}	{}
t3	b	ve	{vl}	{(t3,ve)}
t2	b	ve	{vl}	{(t3,ve), (t2,ve)}
t2	a	unlock	{}	{(t1,ve)}
t1	a	ve	{vl,ve}	{}
t3	b	unlock	{}	{(t3,ve), (t2,ve)}
t2	b	ve	{vl,ve}	{}

t3 annulé

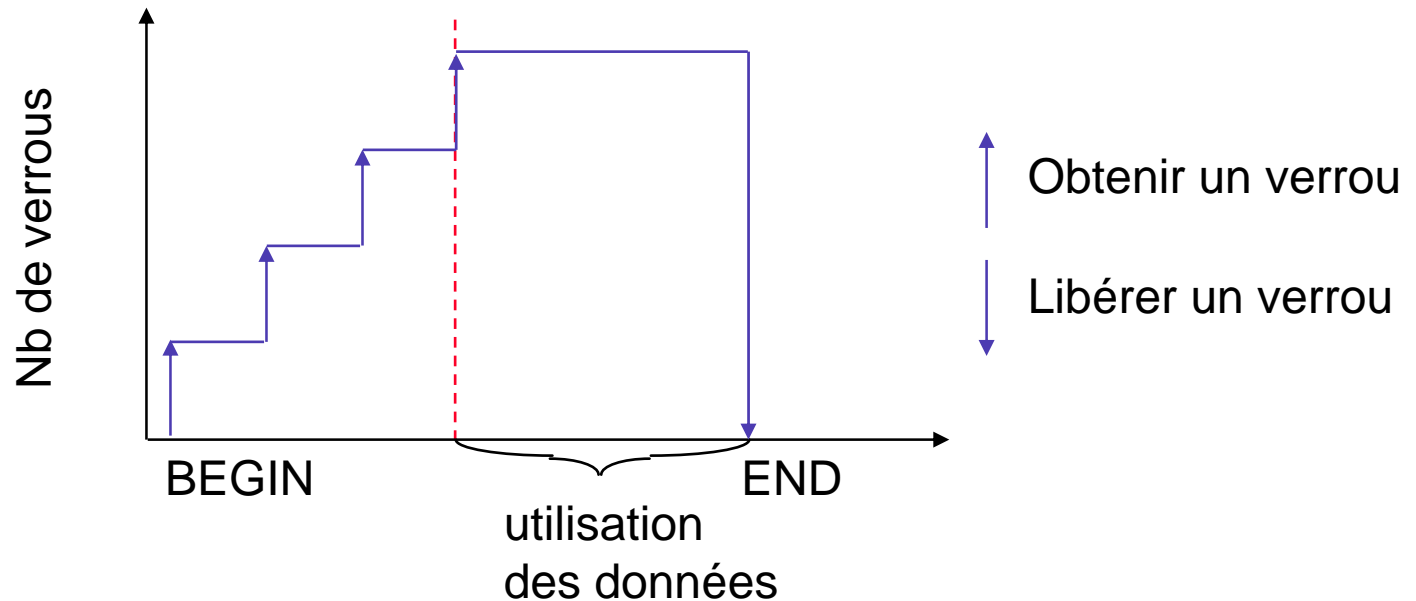
Verrouillage à deux phases

- Chaque transaction verrouille l'objet avant de l'utiliser.
- Quand une demande de verrou est en conflit avec un verrou posé par une autre transaction *en cours*, la transaction qui demande doit attendre.
- **Quand une transaction libère son premier verrou, elle ne peut plus demander d'autres verrous.**



Verrouillage à deux phases strict

On tient les verrous jusqu'à la fin (commit, abort).

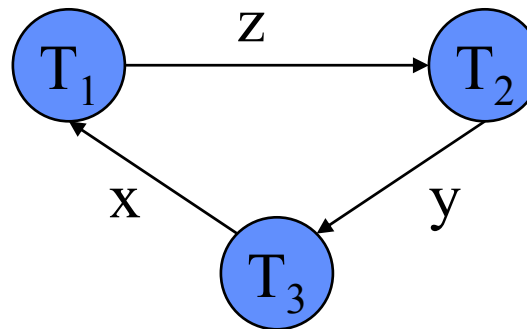


Problème avec le verrouillage : Interblocage

$VP_1(X)$, $VX_2(z)$, $VX_3(y)$, $VP_2(z)$, $VX_3(x)$, ... $D_1(x)$

Granule	Verrou-X	Verrou-P	Attente X	Attente P
x		T_1	T_3	
y	T_3			T_2
z	T_2			T_1

Graphe d'attente :



Cycle \rightarrow Interblocage

Résolution des interblocages

Prévention

- définir des critères de priorité de sorte à ce que le problème ne se pose pas
- exemple : priorité aux transactions les plus anciennes

Détection

- gérer le graphe des attentes
- lancer un algorithme de détection de circuits dès qu'une transaction attend trop longtemps
- choisir une victime qui brise le circuit

Prévention des interblocages

Algorithme : Les transactions sont numérotées par ordre d'arrivée et on suppose que T_i désire un verrou détenu par T_j :

- Choix préemptif (« priorité aux anciens »):
 - $j > i$: T_i prend le verrou et T_j est *abandonnée*.
 - $j < i$: T_i attend.
- Choix non-préemptif (« priorité aux jeunes »):
 - $j > i$: T_i attend.
 - $j < i$: T_j est *abandonnée*.

Théorème : Il ne peut pas avoir d'interblocages, si une transaction abandonnée est toujours relancée avec *le même numéro*.

Verrouillage à deux phases (2PL):

Conclusion

Théorème : Le protocole de verrouillage à deux phases génère des *historiques sérialisables en conflit* (mais n'évite pas les fantômes).

Autres versions de 2PL:

- Relâchement des verrous en lecture après l'opération :
 - - non garantie de la reproductibilité des lectures (READ_COMMITTED)
 - + verrous conservés moins longtemps : plus de parallélisme
- Accès à la version précédente lors d'une lecture bloquante :
 - - nécessité de conserver une version (journaux)
 - + une lecture n'est jamais bloquante

Sérialisation par estampillage (1/3)

On affecte

- à chaque transaction t une **estampille unique** $TS(t)$ dans un domaine ordonné.
- à chaque granule g
 - une **étiquette de lecture** $EL(g)$ et
 - une **étiquette d'écriture** $EE(g)$

qui contient l'estampille de la dernière transaction qui a lu, respectivement, écrit g .

Sérialisation par estampillage (2/3)

La transaction t veut **lire** g :

- Si $TS(t) \geq EE(g)$, la lecture est acceptée et
$$EL(g) := \max(EL(g), TS(t)).$$
- Sinon la lecture est refusée et t est relancée (abort) avec une nouvelle estampille (*plus grande que toutes les autres*).

La transaction t veut **écrire** g :

- Si $TS(t) \geq \max(EE(g), EL(g))$, l'écriture est acceptée et
$$EE(g) := TS(t).$$
- Sinon l'écriture est refusée et t est relancée avec une nouvelle estampille (*plus grande que toutes les autres*).

Sérialisation par estampillage (3/3)

$L_1(b), L_2(b), E_2(b), L_1(a), L_2(a), E_2(a), E_1(b)?$

- $TS(t_1) < TS(t_2)$: T_1 sera relancée avec une nouvelle estampille $TS'(t_1) > TS(t_2)$:

$L_1(b), L_2(b), E_2(b), L_1(a), L_2(a), E_2(a), L_3(b), L_3(a), E_3(b)$

Remarques :

- Il existe des historiques qui ne peuvent pas être *produits* par 2PL mais sont admis par estampillage et vice-versa.
- L'estampillage est une *stratégie optimiste* : on espère qu'une nouvelle transaction ne rentre pas en conflit avec les anciennes

Règle de Thomas

$L_1(b), E_2(b,v), L_1(a), L_2(a), E_2(a), E_1(b,v')?$

$L_1(b), L_1(a), E_1(b,v'), E_2(b,v), L_2(a), E_2(a)$

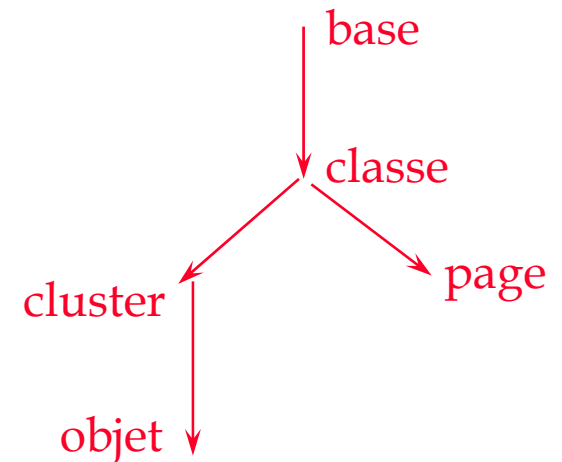
Observation : Aucune transaction avec $TS(t) > TS(t_1)$ a lu b :
L'abandon de T_1 n'est pas nécessaire car v' n'aurait jamais été lue, si l'écriture s'était passée plus tôt.

Nouvelle règle pour l'écriture:

- Si $TS(t) \geq \max(EE(g), EL(g))$, l'écriture est acceptée et $EE(g) := TS(t)$.
- Si $TS(t) < EE(g)$ et $TS(t) \geq EL(g)$, l'écriture est *ignorée*.
- Sinon l'écriture est refusée et T est relancée avec une nouvelle estampille.

Granularité Variable

- Les granules de verrouillage sont organisés dans une hiérarchie d'inclusion.
- Le verrouillage s'effectue du *bas vers le haut*
 - en *mode effectif* (E ou L) sur les granules choisis et
 - en *mode intention* (IE ou IL) sur les granules supérieurs
- Compatibilité :
 - Compatibilité classique entre les modes effectifs
 - Autres conflits : $IL \leftrightarrow E$, $IE \leftrightarrow E$, $IE \leftrightarrow L$



Le problème des annulations

$H = W_2(x) R_1(x) W_1(x) C_1 R_3(x) W_2(y) R_3(y) R_2(z) R_3(z) A_2$

Quand une transaction est abandonnée (A_2) il faut annuler

- les écritures de T_2
- **et** les transactions T_1 et T_3 qui utilisent ces écritures

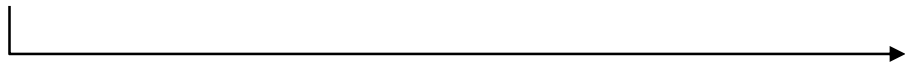
Problème :

- annulation de transactions *en cascade*
- annulation de transactions déjà validées (T_1)

Recouvrabilité

- Exécution *recouvrable* : pas d'annulation de transactions validées
- Solution : retardement du *commit*
 - Si $T \ll \text{lit de} \gg T'$, alors T' doit valider après T

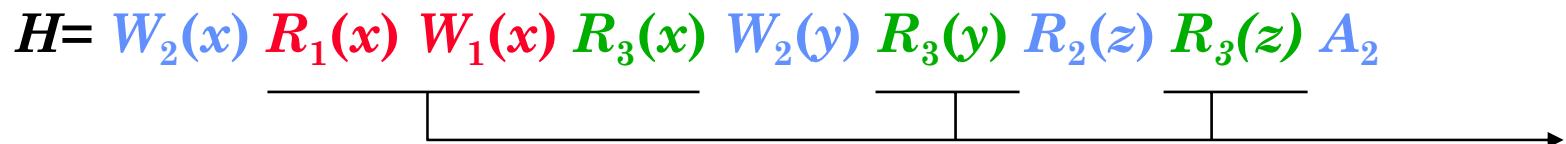
$H = W_2(x) \ R_1(x) \ W_1(x) \ C_1 \ R_3(x) \ W_2(y) \ R_3(y) \ R_2(z) \ R_3(z) \ A_2$



Éviter annulation en cascade

$H = W_2(x) \text{ } R_1(x) \text{ } W_1(x) \text{ } R_3(x) \text{ } W_2(y) \text{ } R_3(y) \text{ } R_2(z) \text{ } R_3(z) \text{ } A_2$

- T2 annule \Rightarrow T1 et T3 doivent annuler
- Solution : retardement des *lectures*
 - Une transaction ne doit lire que de transactions validées

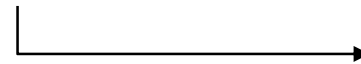


Exécutions strictes

$$H = W_2(x) \ W_1(x) \ A_2 \ A_1$$

- T2 annule \Rightarrow comment restaurer la valeur de x ?
 - A_2 restaure x initial, A_1 restaure x écrit par T₂
- Solution : retardement des *lectures et écritures*
 - *Une transaction ne doit écrire que sur des données validées*

$$H = W_2(x) \ W_1(x) \ A_2$$



Conclusion des transactions

- La gestion des transactions est une tâche importante dans un SGBD :
 - Gestion de pannes
 - Contrôle de concurrence
- Moniteurs transactionnels :
 - Pilotes d'exécution distribuée de transactions globales sur des ressources distribuées
 - Protocoles standards
- Transactions emboîtées
 - Transactions/sous-transactions

Degrés de cohérence

Données sales : Une donnée est **sale** pour une transaction T si elle a été mise-à-jour par une autre transaction T' non-validée.

Degrés de cohérence : Un **ensemble de transactions** $\mathcal{T} = \{T\}$ est cohérent au

- degré 0 si
 - aucune transactions T dans \mathcal{T} *n'écrit* des valeurs obtenues à partir de *données sales* dans la base de données
- degré 1 si \mathcal{T} est cohérent au degré 0 et
 - aucune transaction T dans \mathcal{T} ne *valide* aucune écriture avant sa fin
- degré 2 si \mathcal{T} est cohérent au degré 1 et
 - aucune transaction T ne *lit* de *données sales*
- degré 3 si \mathcal{T} est cohérent au degré 2 et
 - les autres transactions ne salissent aucune donnée lue par T avant la validation de T

Degrés d'isolation SQL-92

Lectures non validées (READ_UNCOMMITTED)

- lectures sales, non répétables et fantômes sont possibles

Lectures validées (READ_COMMITTED)

- lectures non répétables et fantômes sont possibles, mais pas de lectures sales

Lectures répétables (REPEATABLE_READ)

- seuls les fantômes sont possibles

Sérialisable (SERIALIZABLE)

- aucun des phénomènes précédents n'est possible