Transactions
Contrôle de concurrence

Mars 2020

igueye@ept.sn

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 : on perd *une partie* des écritures de T1 et *une partie* des écritures de T2 sur des données partagées

[T1: Write a1 → A; T2 : Write b2 →B; T1: Write b1 → B; T2 : Write a2 →A;]

A=a2, B=b1 : on perd une écriture de T1 et une écriture de T2

Non reproductibilité des lectures : une transaction écrit une donnée entre deux lectures d'une autre transaction

[T1 : Read A; T2 : Write b2→A; T1: Read A;]

Si b2 est différente de la valeur initiale de A, alors T1 lit deux valeurs différentes.

Conséquence : *introduction d'incohérence*

Exemple avec une contrainte (A = B) et deux transactions T1 et T2

Avant : A=B

[T1 : A*2→A; T2 : A+1→A; T2 : B+1 → B; T1 : B*2 → B;]

Après : A = 2*A+1, B=2*B+2 → A <> B

Degrés d'isolation SQL-92

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

T1: Write(A); T2: Read(A); T1: abort (abandon en cascade)

- T_1 modifie A qui est lu ensuite par T_2 avant la fin (validation, annulation) de T_1
- Si T_1 annule, T_2 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);

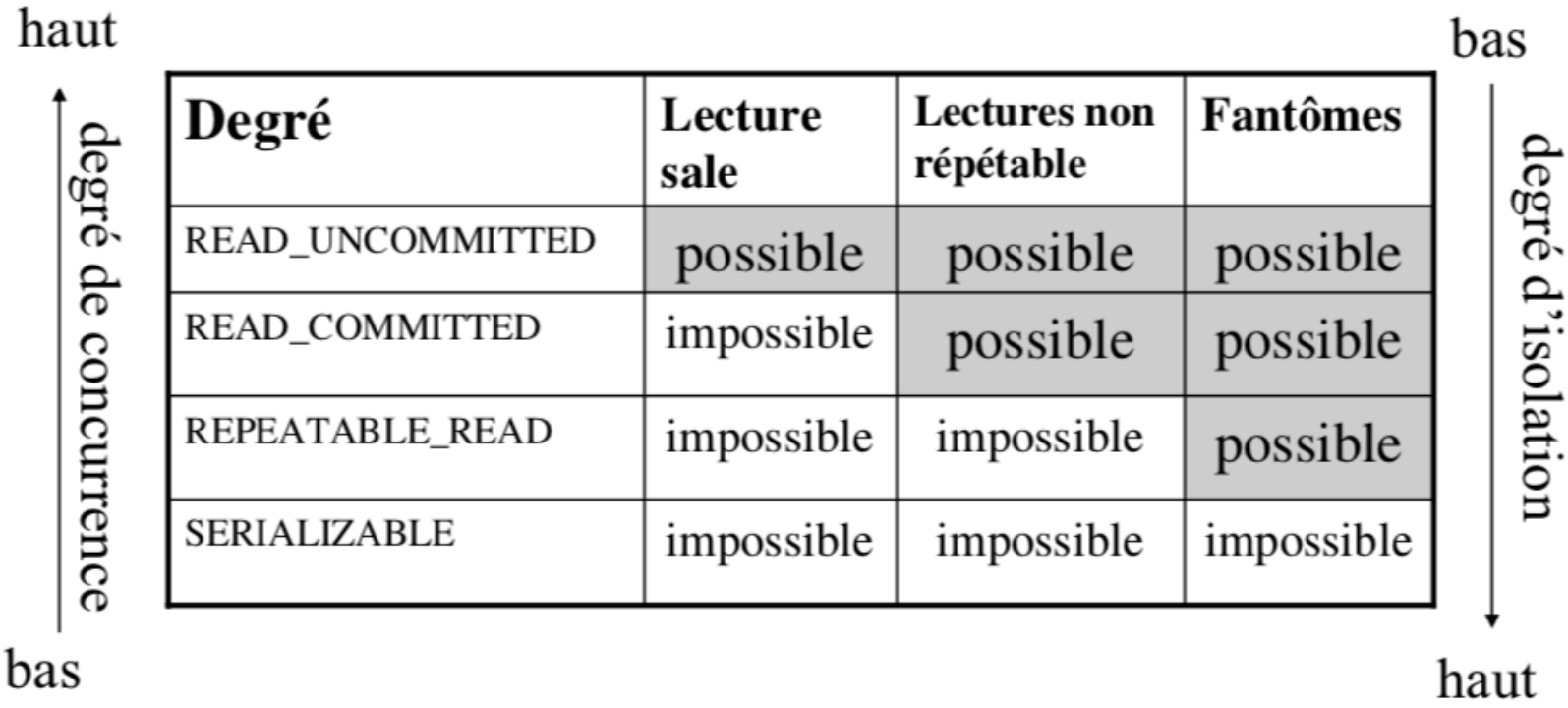
- T_1 lit A; T_2 modifie ou détruit A et valide
- Si T_1 lit A à nouveau et obtient *résultat différent*

Fantômes (requête + insertion) :

T1: Select where R.A=…; T2: Insert Into R(A) Values (…);

- T_1 exécute une requête 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 possible).

Degrés d'isolation SQL-92



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

Exécution (ou Histoire)

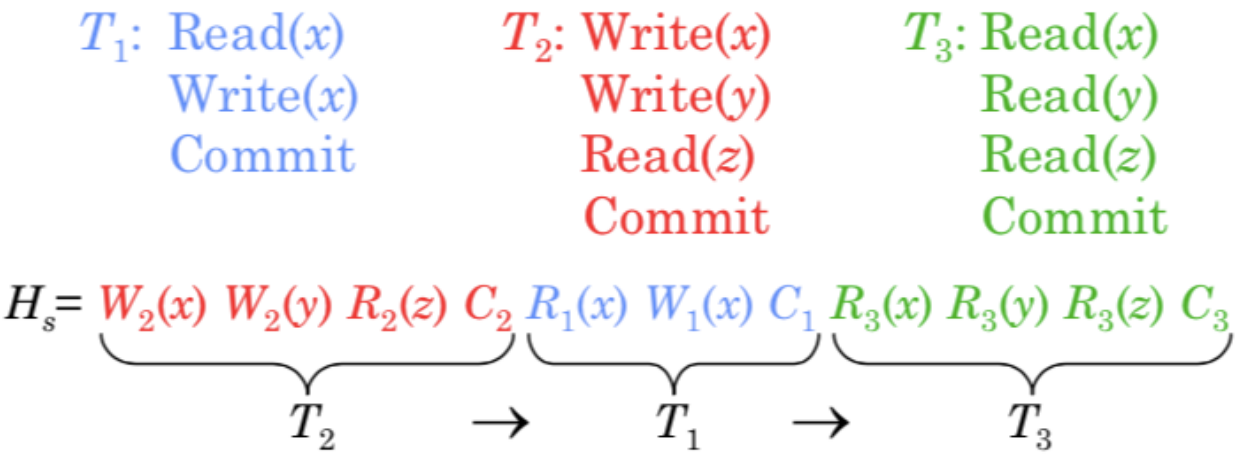
- Exécution : ordonnancement des opérations d'un ensemble de transactions
- Ordre : *total* (séquence = transactions plates)
  ou *partiel* (arbre, modèles avancés))

T_1: Read(x)	T_2: Write(x)	T_3: Read(x)
Write(x)	Write(y)	Read(y)
Commit	Read(z)	Read(z)
	Commit	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:	T_2:	$T3$:
Read(x)	Write(x)	Read(x)
Write(x)	Write(y)	Read(y)
Commit	Read(z)	Read(z)
	Commit	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 :*

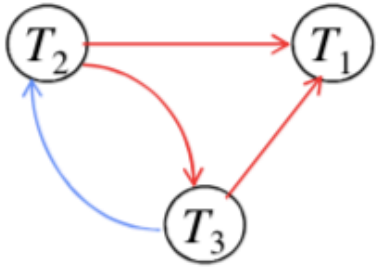
$H_s = W_2(x)\, W_2(y)\, R_2(z)\, C_2\, R_1(x)\, W_1(x)\, C_1\, R_3(x)\, R_3(y)\, R_3(z)\, C_3$

Est ce que H1 est sérialisable ?

Graphe de précédence (GP)

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

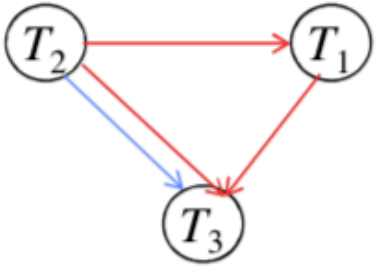
- $V=\{T_i \mid T_i$ est une transaction *validée* dans $H\}$
- $P=\{T_i \rightarrow T_k$ si $o_{ij} \in T_i$ et $o_{kl} \in T_k$ sont en conflit et $o_{ij} <_H o_{kl}\}$



H_1

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

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



H_2

Théorème: l'exécution H est sérialisable ssi GP_H ne contient pas de cycle

H_2 correspond à l'exécution en série : T_2-T_1-T_3

Algorithmes de contrôle de concurrence

- Méthode optimiste
- Verrouillage à deux-phases (2PL)
- Multiversion (snapshot)
- Estampillage

Algorithmes de verrouillage

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

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

Compatibilité (de verrous sur le même granule et deux transactions différentes) :

	vl	*ve*
vl	**Oui**	Non
ve	Non	Non

Algorithmes Lock

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

Exemple Lock

t1 bloqué

t2 bloqué

ti	Gj	V	ti.verrous(Gj)	Gj.attente
t1	a	vl	{vl}	{}
t3	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}	{}

Algorithmes Unlock

```
Procedure Unlock(Transaction t, Granule G) {
    /* t libère tous les verrous sur G et redémarre les transactions en
    attente (si possible) */
     t.verrou(G) := {};
     Pour chaque couple (t', V) dans G.attente faire {
        si Lock(t', G, V) alors {
            G.attente = G.attente - {(t', V)};
            débloquer la transaction t';
        }
     }
}
```

Exemple Unlock

t1 bloqué

t3 bloqué

ti	Gj	V	ti.verrous(Gj)	Gj.attente
…	…	…	…	…
t2	a	ve	{vl,ve}	{(t1,ve)}
t3	b	vl	{vl}	{}
t2	b	vl	{vl}	{}
t3	b	ve	{vl}	{(t3,ve)}
t2	a	unlock	{}	{(t1,ve)}
t1	a	ve	{ve}	{}
t2	b	unlock	{}	{(t3,ve)}
t3	b	ve	{ve,vl}	{}

Exemple Interbloage

ti	Gj	V	ti.verrous(Gj)	Gj.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) }

t1 attend t2, t2 attend t1 : INTERBLOCAGE

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 et Sérialisabilité

Est-ce que le verrouillage permet de garantir la sérialisabilité ???

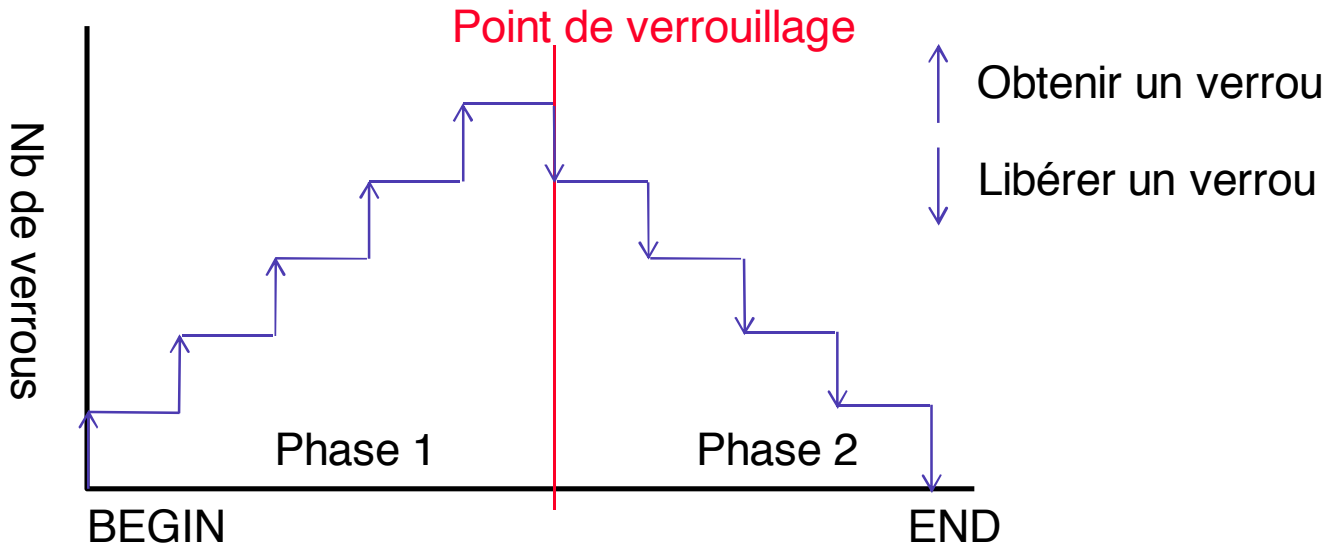
$H_1 = W_2(x)\ R_1(x)\ \ R_3(x)\ W_1(x)\ C_1\ R_3(y)\ R_2(z)\ W_2(y)\ C_2\ R_3(z)\ C_3$

On a vu que H1 n'est pas sérialisable, pourtant…

=> Il ne faut pas libérer les verrous trop tôt

Verrouillage à deux Phases

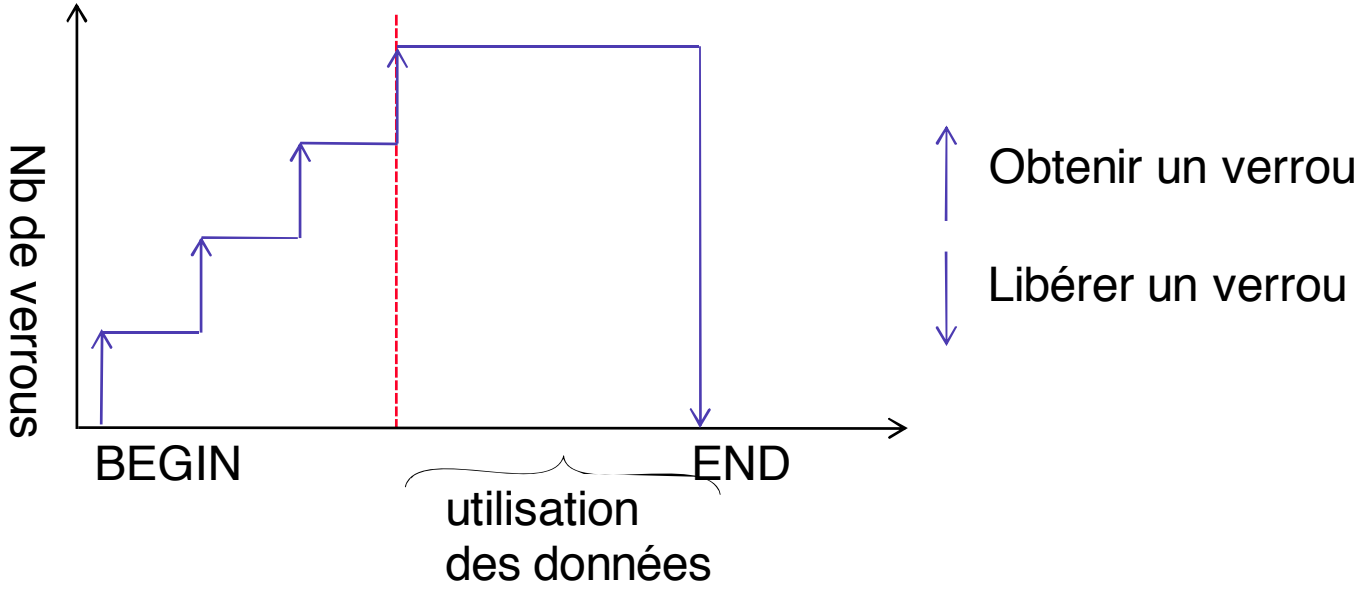
- Chaque transaction verrouille l'objet avant de l'utiliser.
- Quand une demande de verrou est en conflit avec un verrou posé par une autre transaction *en cours*, la transaction qui demande doit attendre.
- **Quand une transaction libère son premier verrou, elle ne peut plus demander d'autres verrous.**



Verrouillage à deux Phases strict

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

Evite les abandons en cascade



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 (Oracle, snapshot isolation):

- 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

Transactions dans Oracle

Une transaction démarre lorsqu'on exécute une instruction SQL qui modifie la base ou le catalogue (DML et DDL).

- Ex : UPDATE, INSERT, CREATE TABLE…

Une transaction se termine dans les cas suivants :

- L'utilisateur valide la transaction (COMMIT)
- L'utilisateur annule la transaction (ROLLBACK sans SAVEPOINT)
- L'utilisateur se déconnecte (la transaction est validée)
- Le processus se termine anormalement (la transaction est défaite)

Amélioration des performances dans Oracle :

- Niveaux d'isolation
- Contrôle de concurrence multiversion

Verrouillage

Commandes transactionnelles

COMMIT

- Termine la transaction courante et écrit les modifications dans la base.
- Efface les points de sauvegarde (SAVEPOINT) de la transaction et relâche les verrous.

ROLLBACK

- Défait les opérations déjà effectuées d'une transaction

SAVEPOINT

- Identifie un point dans la transaction indiquant jusqu'où la transaction doit être défaite en cas de rollback.
- Les points de sauvegarde sont indiqués par une étiquette (les différents points de sauvegarde d'une même transaction doivent avoir des étiquettes différentes).

SET TRANSACTION

SET TRANSACTION

- Spécifie le comportement de la transaction :
  - Lectures seules ou écritures (READ ONLY ou READ WRITE)
  - Établit son niveau d'isolation (ISOLATION LEVEL)
  - Permet de nommer une transaction (NAME)

Cette instruction est facultative. Si elle est utilisée, elle doit être la première instruction de la transaction, et n'affecte que la transaction courante.

READ ONLY & READ WRITE

SET TRANSACTION READ ONLY

- La transaction devient en lecture seule (pas d'INSERT, UPDATE, DELETE)
- Garantit la cohérence en lecture pour toute la transaction. Cette transaction voit seulement les modifications de la base effectuées avant le début de la transaction.
- Utile pour des transactions qui font beaucoup de lectures successives sur des objets modifiés simultanément par d'autres utilisateurs.

SET TRANSACTION READ WRITE

- Option par défaut.

EXEMPLE

```
COMMIT;
```
% assure que l'instruction **set transaction** est la première de la transaction

```
SET TRANSACTION READ ONLY NAME
'Toronto';
SELECT product_id, quantity_on_hand
FROM inventory WHERE warehouse_id=5;
COMMIT;
```
% termine la transaction **READ ONLY**

NIVEAUX D'ISOLATION

Oracle propose deux niveaux d'isolation, pour spécifier comment gérer les mises à jour dans les transactions

- **SERIALIZABLE**
- **READ COMMITTED**

Définition

- Pour une transaction :
  - **SET TRANSATION ISOLATION LEVEL SERIALIZABLE**
  - **SET TRANSATION ISOLATION LEVEL READ COMMITTED**
- Pour toutes les transactions à venir (dans une session)
  - **ALTER SESSION SET ISOLATION LEVEL = SERIALIZABLE;**
  - **ALTER SESSION SET ISOLATION LEVEL = READ COMMITTED;**

Différence read-commited serializable

En mode RC, on lit la dernière valeur validée depuis le début de la commande SQL Select.

En mode SR, on lit la dernière valeur validée depuis le début de la transaction.

En d'autres termes, RC permet des lectures non réparables alors SR ne le permet pas.

Mais E1(a), E2(b), L1(b) L2(a), V1, V2 en mode SR n'est ni équivalent à T1, T2 ni équivalent à T2, T1 car les deux lectures vont lire l'état initial (avant le début de l'exécution).