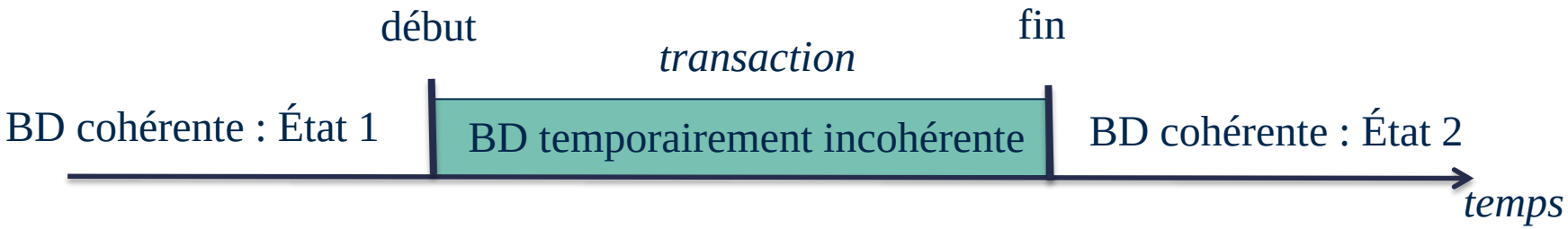
6 Transactions (sécurité des données, gestion des accès concurrents)

Définition d'une transaction

- Définition : suite d'actions (lectures/écritures) qui manipule/modifie le contenu d'une BD en maintenant la cohérence des données



- Une transaction idéale doit satisfaire quatre critères :

Atomicité : ses actions ne peuvent pas être séparées et ne doivent donc pas être interrompues (tout ou rien)

Cohérence : les changements dûs à la transaction ne doivent pas altérer la cohérence de la BD

Isolation : les transactions sont indépendantes les unes des autres

Durabilité : les effets d'une transaction validée doivent perdurer même en cas de panne

Problèmes liés à la non atomicité

Virement bancaire d'un montant *m* d'un compte C_1 vers un compte C_2 :

Début
- $x \leftarrow Lire(C_1)$
- $x \leftarrow x - m$ *//calcul du nouveau crédit*
- $C_1 \leftarrow Ecrire(x)$ *//écriture du crédit de* C_1
- $y \leftarrow Lire(C_2)$
- $y \leftarrow y + m$ *//calcul du nouveau crédit*
- $C_2 \leftarrow Ecrire(y)$ *//écriture du crédit de* C_2

Fin

Pb : Si le programme s'arrête avant l'écriture de y dans C_2, la base entre dans un état incohérent

→ Transaction = **unité atomique d'actions**

Si la transaction n'a pas pu exécuter toutes ses actions, la base est remise dans l'état où elle se trouvait avant le début de la transaction

Problèmes liés à la non isolation

- Lectures impropres

Ex: T1 inscrit un débit *N* sur un compte *C* et *T2* inscrit un crédit *M* sur *C* :

T1
$x \leftarrow Lire(C)$
$x \leftarrow x - N$
$C \leftarrow Ecrire(x)$

T2
$y \leftarrow Lire(C)$
$y \leftarrow y + M$
$C \leftarrow Ecrire(y)$

Ex : $C = 100$, $N = 10$, $M = 50$
Résultat attendu : 100-10+50 = **140**

Actions	*C*	*x*	*y*
T1: $x \leftarrow Lire(C)$	*100*	*100*	
T1: $x \leftarrow x - N$	*100*	*90*	
T2: $y \leftarrow$ ***Lire(C)***	*100*	*90*	*100*
T2: $y \leftarrow y + M$	*100*	*90*	*150*
T1: $C \leftarrow Ecrire(x)$	*90*	*90*	*150*
T2: $C \leftarrow Ecrire(y)$	***150***	*90*	*150*

→ Il faut que T1 travaille en **isolation**, c'est-à-dire sans interférence avec d'autres transactions, jusqu'à sa terminaison

Problèmes liés à la non isolation

- Non répétabilité des lectures (lectures non reproductibles)

Transaction T1	*Transaction T2*
a ← Lire(A)	
	b ← Lire(A)
	Imprimer(b)
a ← a + 100	
A ← Ecrire(a)	
	b ← Lire(A)
	Imprimer(b)

Pb : les impressions de T2 produiront des valeurs différentes

T1 ne devrait pas pouvoir modifier les données lues par T2

Problèmes liés à la non isolation

- Tuples "fantômes" (apparition/disparition de tuples)

Transaction T1	Transaction T2
Supprimer A = {a | a < 4000}	V ← Lire(A), A={a | a < 4000} Imprimer(V) V ← Lire(A) , A={a | a < 4000} Imprimer(V)

Pb : La deuxième impression de V ne donne pas le même nombre de tuples (tuples «fantômes »)

T1 ne devrait pas pouvoir modifier les données lues par T2

Plusieurs outils pour empêcher ces problèmes

- *Le sous-système d'intégrité* : permet de détecter les incohérences prévues par le programmeur (CI)
- *Le sous-système de reprise* : permet d'assurer l'atomicité des transactions et la cohérence de la base en cas d'accident
- *Le sous-système de contrôle de la concurrence* : permet de contrôler les accès concurrents aux données par la technique des "verrous"

Atomicité des transactions : politique du "tout ou rien"

Début transaction (BEGIN TRAN)
Action1
...
Action n
Si toutes les actions se sont bien passées
Alors **valider la transaction (COMMIT)**
Sinon **annuler les effets de la transaction (ROLLBACK)**

Une transaction se termine lorsque l'une des deux commandes COMMIT ou ROLLBACK est exécutée

Démarrage des transactions : Cas du SGBD Oracle

- Pas de démarrage explicite de transaction : pas de commande BEGIN TRAN

- Une transaction démarre implicitement
  - à l'ouverture d'une session de travail
  - ou après toute commande COMMIT ou ROLLBACK

- Les commandes du LDD (CREATE, DROP, ALTER) sont validées automatiquement
  - COMMIT implicite avant et après toute commande du LDD

```
--ouverture session Oracle
Action1
Action 2
COMMIT;
Action 3  -- commande du LDD
ROLLBACK;
```

Utilisation de points de retour (*savepoints*)

```
UPDATE employees SET salary = 7000 WHERE last_name = 'Banda';
SAVEPOINT banda_sal;
UPDATE employees SET salary = 12000 WHERE last_name = 'Greene';
SAVEPOINT greene_sal;
SELECT SUM(salary) FROM employees;      -- salaire total trop important
ROLLBACK TO SAVEPOINT banda_sal;
UPDATE employees SET salary = 11000 WHERE last_name = 'Greene'; --réajuster
COMMIT;
```

si UPDATE à la place de ROLLBACK, il faut vérifier que la modification du salaire n'a pas entrainé des modifications en cascade et corriger si besoin

avec ROLLBACK : retour à l'état précédent

Le journal des transactions

- Pour pouvoir procéder à la reprise en cas d'accident, le système garde une trace (*journal* ou *log*) des actions (transactions) effectuées sur la base

- Le journal est conservé sur un support non volatile (disque) et doit être archivé périodiquement

Le journal des transactions

- Pour chaque transaction T, le journal comporte :
  - *<début_transaction, identification de T>*
  - *<écriture, identification de T, donnée concernée, ancienne valeur, nouvelle valeur>*
  - *<lecture, identification de T, donnée concernée>*
  - *<COMMIT, identification de T>*
- Ainsi si un accident se produit, le système peut
  - annuler (UNDO) les effets des transactions non validées
  - refaire (REDO) les actions des transactions validées

Reprise en cas d'accident

- Reprise (*recovery*) : reconstruire un état cohérent de la base à partir du passé, cet état devant être le plus proche possible de l'instant où s'est produit l'incident → utilisation du journal des transactions

- La stratégie de reprise dépend de la gravité de l'incident qui la provoque :
  - *Reprise à froid* : pour des dégâts importants, on recharge une sauvegarde de la base et on ré-exécute (REDO) les transactions marquées valides dans le journal, depuis la date de la sauvegarde
  - *Reprise à chaud* : pour des dégâts moins importants, on défait (UNDO) les actions des transactions non validées

Contrôle de la concurrence : la technique du verrouillage

Granule = unité d'accès (BD, relation, tuple, attribut ...)

- **Verrou binaire** : un granule est accessible ou inaccessible
- **Verrou partagé et verrou exclusif** : en *lecture* un verrouillage en mode partagé (*share*) est possible, il permet des accès multiples à un granule. Par contre, en *écriture*, il est nécessaire de verrouiller le granule en mode exclusif (*exclusive*)
- **Verrou d'intention** ou verrou de mise à jour (*update*) : verrou de lecture avec *intention* d'écriture

Compatibilité des verrous :

verrou demandé sur le même granule

verrou apposé sur un granule	Share	Exclusive	Update
Share	oui	non	non
Exclusive	non	non	non
Update	oui	non	non

Les quatre niveaux d'isolation SQL-92 (1/2)

- Niveau 0 (**`READ UNCOMMITTED`**)

  Une transaction T peut lire des objets modifiés par une autre transaction, pas de verrou en mode lecture

  → risque de lectures impropres, lectures non reproductibles et tuples fantômes

- Niveau 1 (**`READ COMMITED`**)

  = niveau 0, sauf que : T lit uniquement les mises-à-jour des transactions validées (verrou *exclusif* en écriture)

  → Il n'y a plus de risque de lectures impropres

Les quatre niveaux d'isolation SQL-92 (2/2)

- Niveau 2 (**`REPEATABLE READ`**)

  = niveau 1, sauf que : aucun objet lu par une transaction T ne peut être modifié par une autre transaction (verrou *partagé* en lecture)

  → il n'y a plus de risque de lectures non reproductibles

- Niveau 3 (**`SERIALIZABLE`**)

  = niveau 2, sauf : possibilité de poser des verrous sur un ensemble d'objets

  → il n'y a plus de risque de tuples fantômes

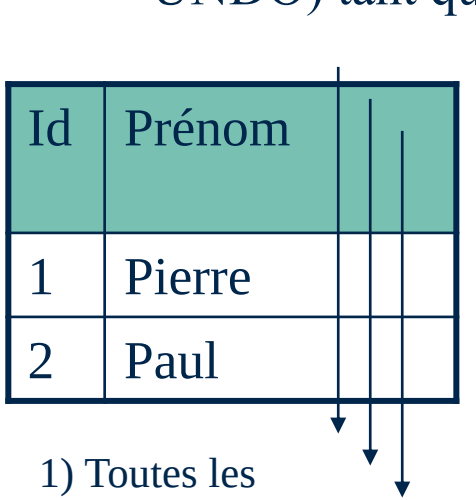
Application au SGBD Oracle

- Oracle supporte les niveaux d'isolation 1 et 3 (lectures impropres impossibles)
  - Niveau 1 par défaut (`read committed`)
- Pour une transaction, un utilisateur peut choisir un mode d'accès et un niveau d'isolation

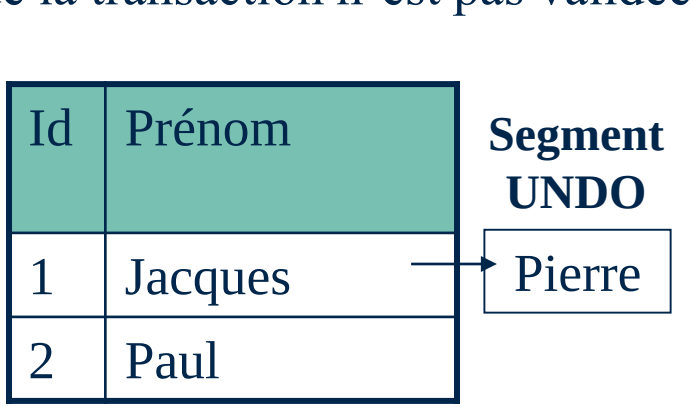
```
SET TRANSACTION [{READ ONLY | READ WRITE} ]
[ISOLATION LEVEL {READ COMMITTED | SERIALIZABLE}]
[NAME NOM_TRANSACTION]
```

Application au SGBD Oracle

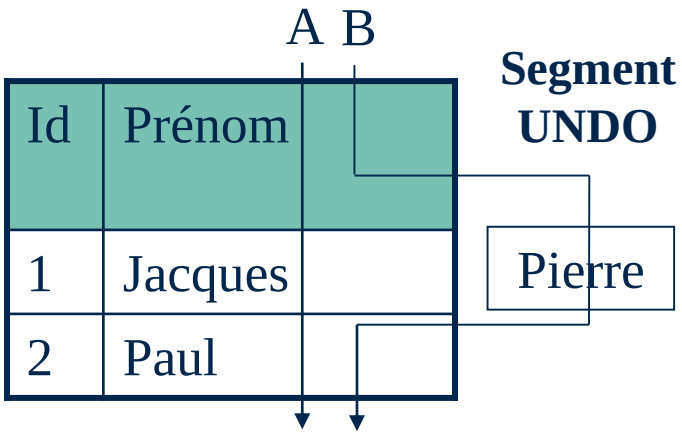
- Niveau 1 : Quand un utilisateur modifie des valeurs, tous les autres utilisateurs ont accès aux données non modifiées (utilisation des segments UNDO) tant que la transaction n'est pas validée



1) Toutes les transactions lisent les données de la table



2) La transaction A modifie un enregistrement. L'ancienne valeur est placée dans un segment d'annulation UNDO



3) La transaction A lit la donnée modifiée, la transaction B lit l'ancienne valeur placée dans le segment UNDO tant que A n'est pas validée

Types de verrous utilisés sous Oracle

- **SHARE (S)**

  Permet l'accès concurrent à la table mais interdit toute modification de la table

- **EXCLUSIVE (X)**

  Permet l'accès concurrent à la table mais interdit toute modification de la table ou toute pose **explicite** de verrou

- **ROW SHARE (RS) ,** *verrou d'intention*

  Permet l'accès concurrent à la table et empêche toute pose de verrou **EXCLUSIVE** sur la table

- **ROW EXCLUSIVE (RX)**

  Idem ROW SHARE et empêche aussi les verrous **SHARE**

Pose implicite ou explicite (Oracle)

- Pose implicite
  - Les instructions INSERT , UPDATE et DELETE posent automatiquement un verrou RX sur la table en cours de modification
  - Une instruction SELECT FROM FOR UPDATE pose un verrou S sur la table
- Pose explicite : l'instruction LOCK

```
LOCK TABLE {table / view}   IN   {EXCLUSIVE /
 SHARE / ROW SHARE / ROW EXCLUSIVE }  MODE ;
```

Les verrous : problèmes

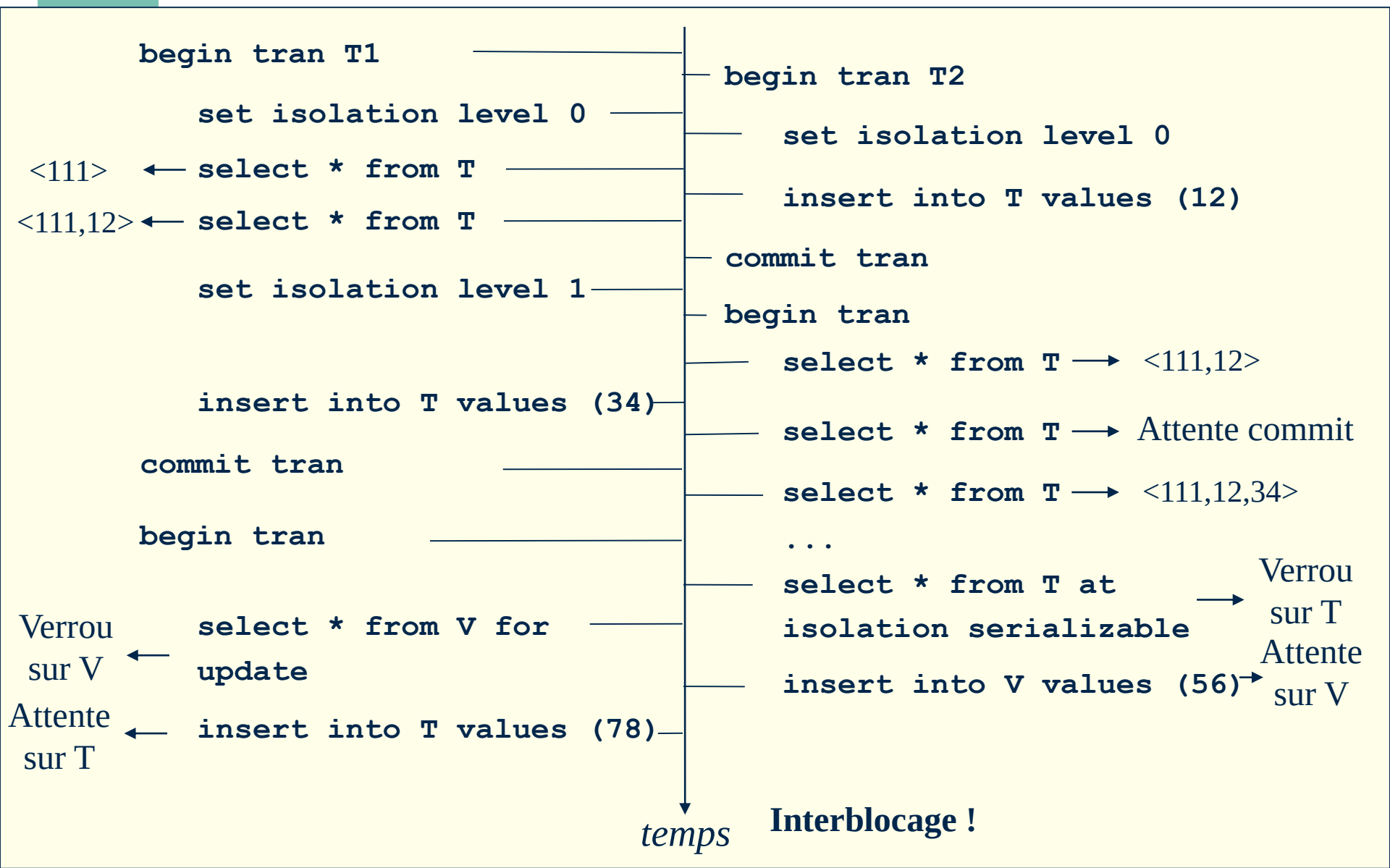
- **Inter-blocage**

T1 détient le granule G1 et attend que T2 libère G2 pendant que T2 attend la libération de G1

- **Famine**

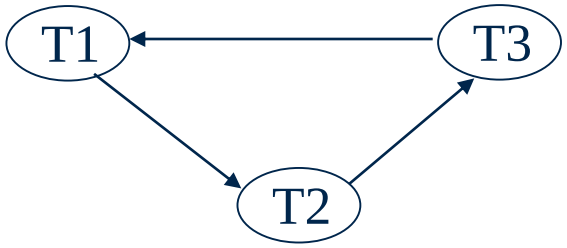
Une transaction est perpétuellement en attente alors que d'autres continuent à s'exécuter

Exemple (SGBD Sybase)



Solutions pour l'inter-blocage

- La *prévention* : imposer à toute transaction de verrouiller en avance tous les éléments dont elle a besoin (n'apposer aucun verrou si un de ces éléments n'est pas libre)
- La *détection* : tester périodiquement si une situation d'inter-blocage s'est produite (solution implémentée dans Sybase et Oracle) :
  - détection de cycles dans un *graphe d'attente*, dont les nœuds représentent les transactions et les arcs la relation *est_en_attente_de*
  - si une telle situation se produit, une des transactions impliquée est avortée



- *T1 détient R1 et attend R2*
- *T2 détient R2 et attend R3*
- *T3 détient R3 et attend R1*

Solutions pour la famine

Cette situation se produit si la priorité est toujours donnée aux mêmes transactions .

Solutions possibles :

- avoir un mécanisme de type "premier arrivé, premier servi" (*First In First Out*)
- adopter une stratégie avec priorité dynamique
- Cas de Sybase : après trois tentatives infructueuses de satisfaction d'une demande d'apposition d'un verrou exclusif, toute nouvelle demande de verrou partagé est refusée