Plan : Gestion de Transaction

1. Transaction ?
2. Résistance aux pannes.
3. Gestion de la concurrence.
4. Les commandes en SQL ?

1. Transaction ?

Transaction ?

Ensemble d'accès à la base de données concernant une seule opération logique.

Exemple :

- VOL (NVol, NbPlacesLibres, CapMax)
- BILLET (NVol, NClient)
- CLIENT (NClient, Nom, Prénom)

Contrainte : Pour tout Vol, ***NbPlacesLibres +Nb_Billets_vendus = CapMax***

1. Transaction ?

Exemple : Achat d'un billet ?

AchatBillet(*NVol, nom, prénom***)**
début
rechercher *NbPlacesLibres* **pour le vol** *NVol* **;**
Si *NbPlacesLibres* **> 0 alors**
Si la recherche du client (*nom, prénom***) échoue alors**
insérer le client (*nom, prenom***) ;**
finSi ;
insérer le billet (*client, vol* **) ;**
mettreAJour *NbPlacesLibres* **=** *NbPlacesLibres***-1 pour le vol** *NVol* **;**
Si la mise à jour du *NbPlacesLibres* **pour le** *NVol* **à échouée alors**
ABORT -- annule les insertions billet et client ;
finSi ;
finSi ;
fin ;

1. Transaction ?

Remarques :

- Besoin de pouvoir écrire du code => Langage de programmation.
  - API dans d'autres langages: JDBC en java, Php
  - « natif » : PL/SQL pour oracle. , etc…
- Contraintes d'intégrité à respecter tout le temps ***SAUF*** à l'intérieur d'une transaction.
- Il faut pouvoir revenir en arrière et annuler ce qui a été fait => Instructions spécialisées : COMMIT, ABORT, SAVE/ROLLBACK, REDO

1. Transaction ?

Remarques :

- Etat de la base donné par :
  - Etat de la mémoire stable (disque dur),
  - Etat du cache en mémoire vive.

  Différences possibles entre les 2 : synchronisation des données non effectuée en permanence.

- Si deux achats de billets simultanés ?

- Si « crash » système au milieu d'une transaction?

1. Transaction ?

Propriétés ***ACID*** d'une transaction :

- ***A***tomicité :
- ***C***ohérence :
- ***I***solation :
- ***D***urabilité :

1. Transaction ?

Problèmes principaux :

- Comment traiter les pannes systèmes ?
- Comment traiter les ABORT, SAVE/ROLLBACK et REDO ?
- Comment traiter les accès concurrents ?

2. Résistance aux pannes. 1. Typologie des erreurs

Les erreurs par gravité croissante :

1. Panne d'une action : une commande du SGBD s'est mal passée.

1. Panne d'une transaction : une transaction s'est mal passée (erreur de programmation, verrou mortel, etc.)

1. Panne de système : crash du SGBD ou du système , mémoire vive perdue, mais disques durs intègres.

1. Panne de mémoire secondaire.

2. Résistance aux pannes. 2. Récupération des erreurs système.

Comment savoir quelles transactions relancer ?

Lesquelles s'étaient terminées correctement ?

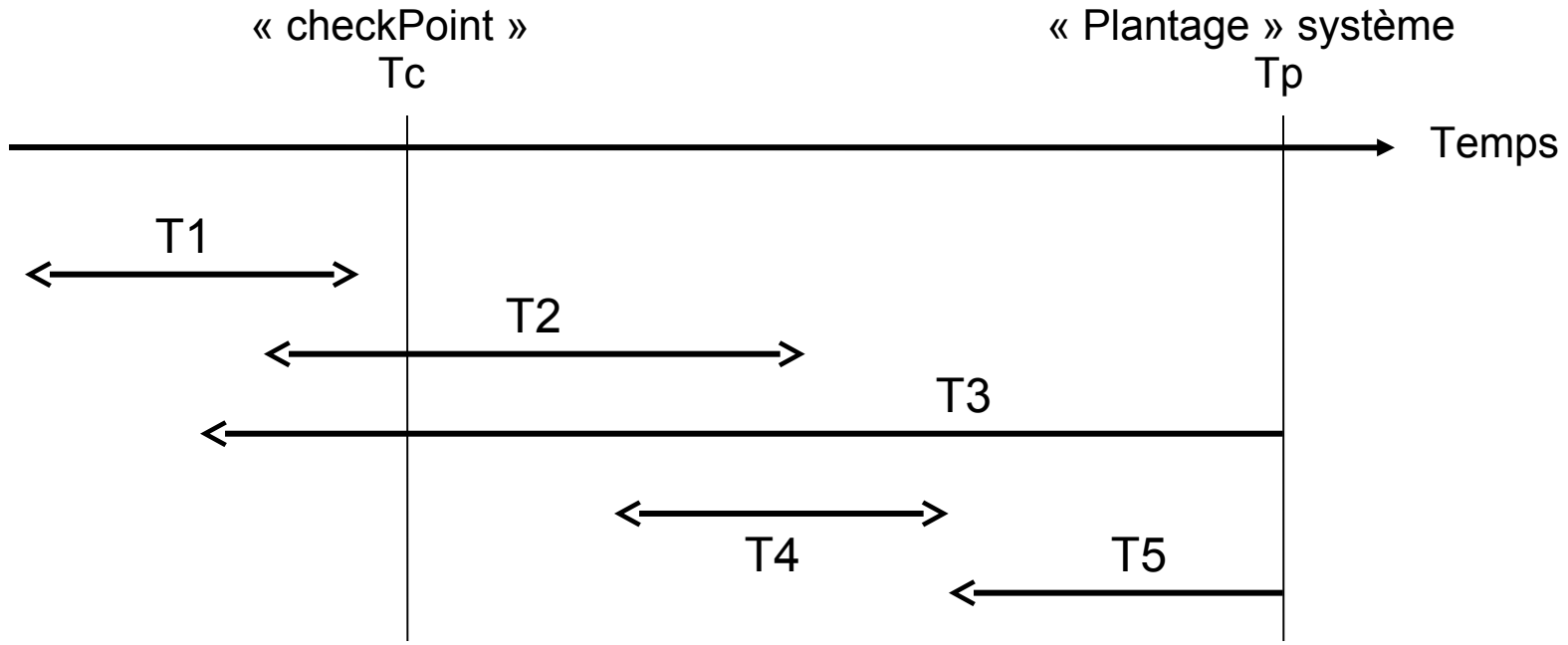
Lesquelles avaient été correctement sauvegardées sur le DD ?

⇒ Mécanisme particulier.

1. Gestion de l'historique des transactions. Journal « log » gardant la trace de toutes les actions du SGBD. Sauvegardé en permanence sur le DD.
2. Créations de « checkpoint » à intervalles réguliers : synchronisation de la mémoire vive avec le DD ; écriture sur le journal de la liste de toutes les transactions en cours.

Remarque : *Pour le moment seul le COMMIT est traité, extension pour les autres instructions spécifiques {SAVE/ROLLBACK, ABORT, REDO}*

2. Résistance aux pannes. 2. Récupération des erreurs système.



Au moment du « restart » :

T3 et T5 doivent être défaites : normalement rien à défaire pour T5.

T2 et T4 doivent être refaites.

Rien à faire pour T1, sauvegardée sur le disque.

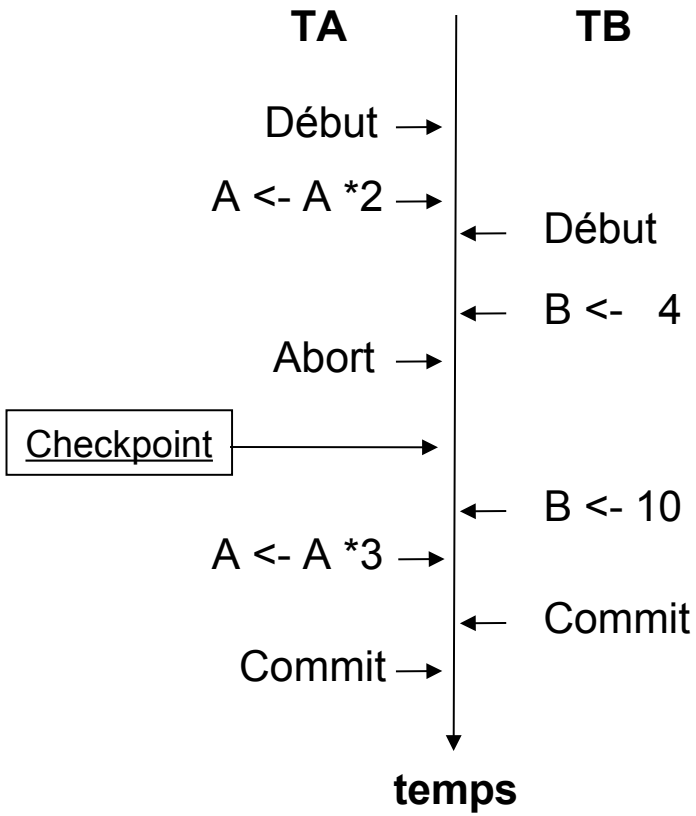
2. Résistance aux pannes. 2. Récupération des erreurs système.

a) Les journal(aux) log(s) :

- Journal des images avant : fichier contenant dans l'ordre chronologique, avec leur date :
  - Pour chaque requête modifiant la base de données :
    - La transaction origine de la modification.
    - L'état de la donnée **AVANT** sa modification.
  - Actions de transactions : Débuts, validations, reprise et annulation de transactions ; points de « checkpoint », etc.
- Journal des images après : fichier contenant dans l'ordre chronologique, avec leur date :
  - Pour chaque requête modifiant la base de données :
    - La transaction origine de la modification.
    - L'état de la donnée **APRES** sa modification.
  - Actions de transactions :…

Remarque : Souvent les deux journaux sont fusionnés en un seul.

Exemple de transactions : deux variables A = 4 et B = 5 initialement ;
TA modifie A, TB modifie B.



Contenu du journal :

- TA Début
- TA avant, A = 4
- TA après, A = 8
- TB Début
- TB avant, B = 5
- TB après, B = 4
- TA Abort
- Ckpt {TA, TB}
- TB avant, B = 4
- TB après, B = 10
- TA avant, A = 4
- TA après, A = 12
- TB Commit
- TA Commit

2. Résistance aux pannes. 2. Récupération des erreurs système.

b) Algorithme de reprise système (Restart) après panne au temps *Tp* :

1) Identification des transactions à défaire/refaire :

1. On reprend le journal au dernier « checkpoint » **Tc**
2. Initialiser deux listes de transactions ***UNDO*** et ***REDO***.
   - UNDO <= {toutes les transactions en cours au temps **Tc**}
   - REDO <= ∅
3. On consulte le journal des log depuis **Tc** jusqu'à **Tp**
   1. Si (Begin **Ti**) alors ajouter **Ti** à ***UNDO***
   2. Si (Commit **Ti)** alors déplacer **Ti** de ***UNDO*** à ***REDO***.

2. Résistance aux pannes. 2. Récupération des erreurs système.

b) Algorithme de reprise système (Restart) après panne au temps *Tp* :

2) défaire/refaire les transactions des listes *UNDO/REDO*:

1. **Avec le journal des images AVANT, défaire les transactions ∈ *UNDO*.**

   Répéter depuis **Tc** vers l'origine des temps
   si la trace concerne une transaction Ti à defaire alors
   si la trace est **(BEGIN Ti)**, alors sortir **Ti** de ***UNDO*** finSi ;
   si la trace a une image avant, alors restaurer la valeur finSi ;
   finSi ;
   jusqu'à ce que ***UNDO*** soit vide.

2. **Avec le journal des images APRES, refaire les transactions ∈ *REDO*.**

   Répéter depuis **Tc** vers le temps **Tp** de la panne
   si la trace concerne une transaction Ti à refaire alors
   si la trace est **(COMMIT Ti)**, alors sortir **Ti** de ***REDO*** **;** finSi ;
   si la trace a une image après, alors restaurer la valeur ; finSi ;
   finSi ;
   jusqu'à ce que ***REDO*** soit vide.

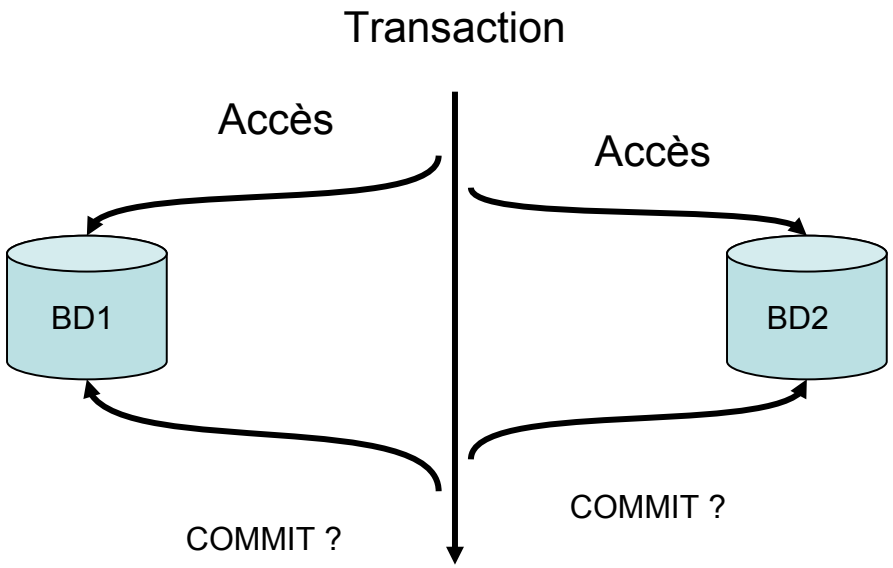
2. Résistance aux pannes. 2. Récupération des erreurs système.

c) Prise en compte des SAVE/ROLLBACK, ABORT, REDO ?

- Inscriptions dans le(s) journal(aux).
- Modifier l'algo de reprise système.
- Algorithmes spécifiques pour ROLLBACK/ABORT et REDO.

2. Résistance aux pannes. 2. Récupération des erreurs système.

d) Cas des bases réparties : une même transaction fait appel à deux bases distantes.



Atomicité : La transaction doit valider sur les deux bases ou sur aucune !

2. Résistance aux pannes. 2. Récupération des erreurs système.

d) Cas des bases réparties : une même transaction fait appel à deux bases distantes.

⇒ **COMMIT ou ROLLBACK global géré par un coordinateur (avec un journal propre) :**

1. La transaction demande au coordinateur le commit général
2. Le coordinateur demande aux deux bases de sauvegarder le résultat (pré-commit).
3. Il demande un compte-rendu d'exécution aux deux bases.
4. Si tous les deux répondent OK,
   - Sauvegarde de la décision OK
   - Demande aux bases du commit définitif.

   Sinon (une base ne répond pas ou répond échec)
   - Sauvegarde de la décision d'annulation
   - Demande aux bases du ROLLBACK.

3. Gestion de la concurrence

Objectif :

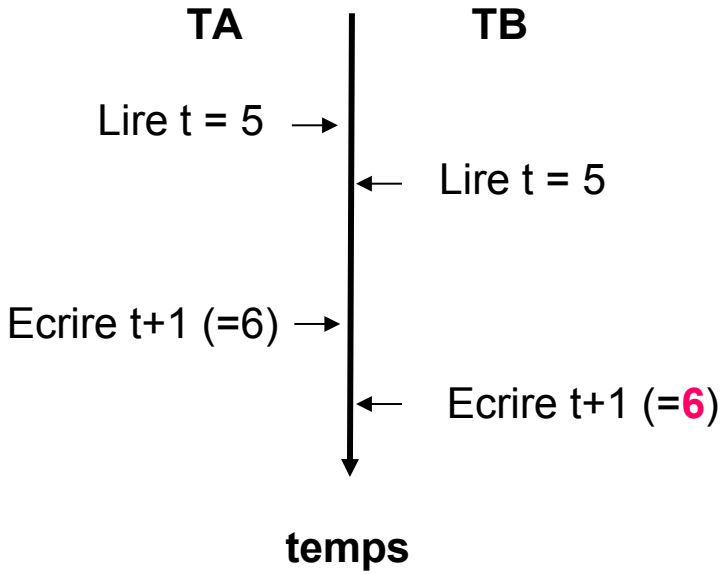
rendre invisible aux clients le partage des données.

=> Contrôle des accès pour éviter perte de mises à jour et incohérences.

3. Gestion de la concurrence. 1. Types de problèmes.

Perte de mise à jour :

TA et **TB** incrémentent une variable **t** initialisée à 5.

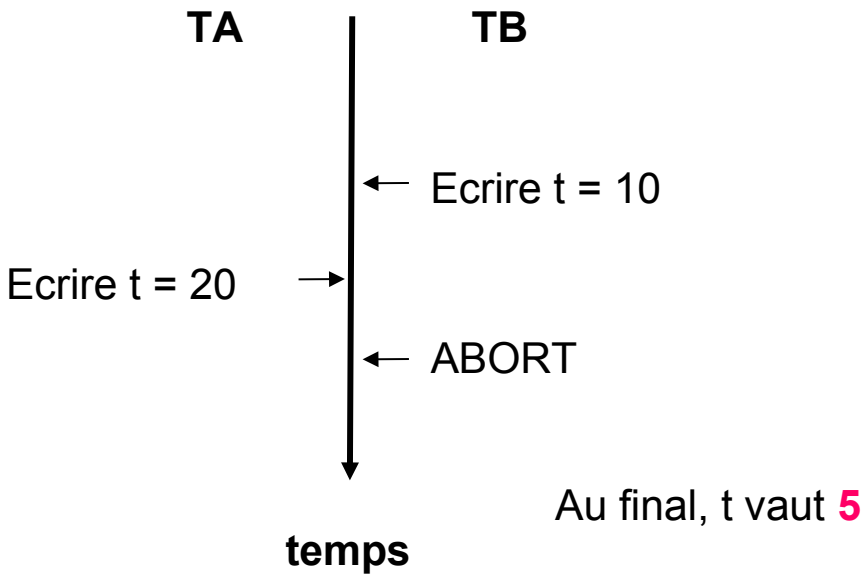


La mise à jour de TA n'est pas prise en compte.

3. Gestion de la concurrence. 1. Types de problèmes.

Perte de mise à jour :

TA et **TB** modifient une variable **t** initialisée à 5.

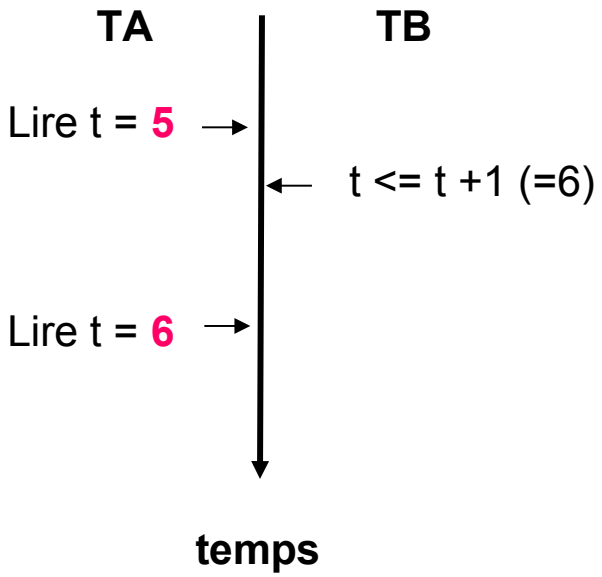


La mise à jour de TA n'est pas prise en compte.

3. Gestion de la concurrence. 1. Types de problèmes.

Lecture non répétable :

TA lit **t, TB** incrémente une variable **t** initialisée à 5.

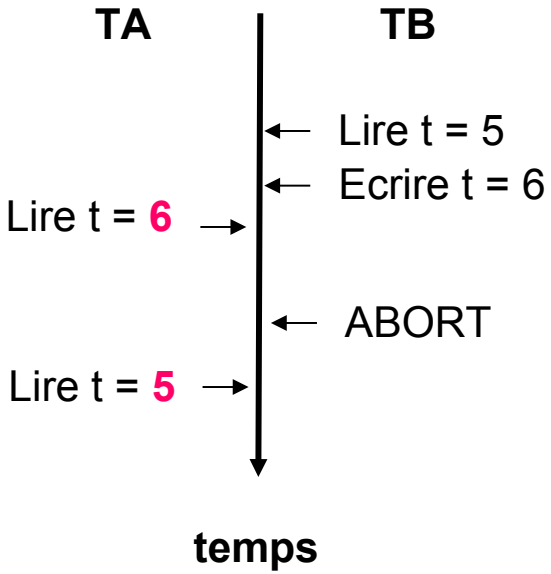


La valeur lue change entre deux lectures !

3. Gestion de la concurrence. 1. Types de problèmes.

Lecture sale :

TA lit **t, TB** incrémente une variable **t** initialisée à 5.



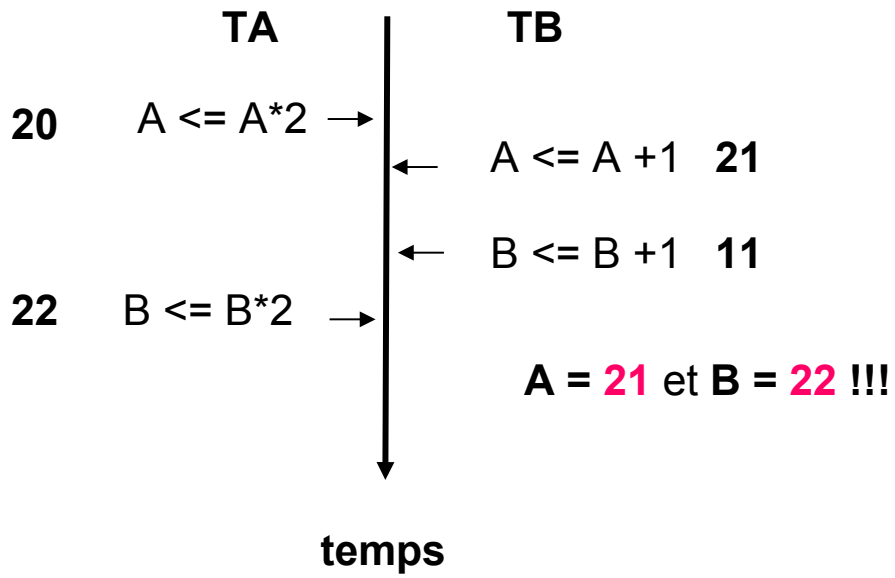
La transaction lit une valeur qui est sensé n'avoir jamais existé !

3. Gestion de la concurrence. 1. Types de problèmes.

Incohérence :

Deux variables **A** et **B** devant rester égales et initialisées à 10

TA multiplie **A** et **B** par 2. **TB** incrémente A et B.



Si séparément chaque transaction est respectueuse de la contrainte, ce n'est pas le cas des deux simultanément

3. Gestion de la concurrence. 2. le verrouillage

Solution : le verrouillage.

la transaction pose un verrou en :

- **Ecriture.** Elle est la seule à avoir le droit d'écrire (et de lire) et interdit aux autres transactions tout accès.
- **Lecture.** Elle lit, les autres peuvent lire, interdit aux autres d'écrire.

Quand on n'a pas le verrou, on est bloqué jusqu'à ce qu'il se libère.

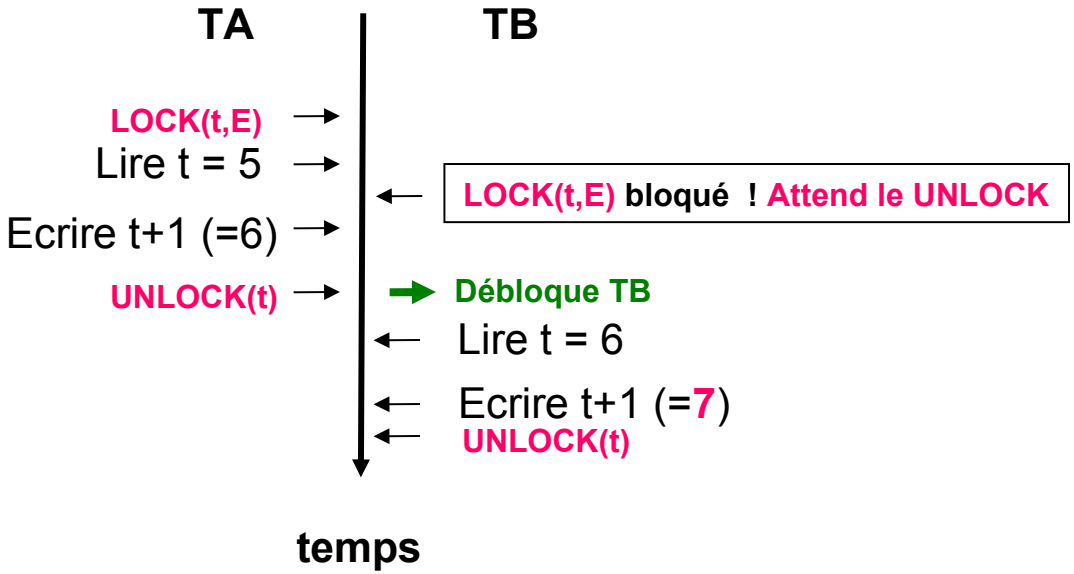
deux procédures LOCK(G,M) et UNLOCK(G)

- **G** : objet bloqué, **M** : mode du verrou (E/L)
- Granule (dimension de l'objet) de taille variable ? (champ, enregistrement, table ?)

3. Gestion de la concurrence. 2. le verrouillage.

Perte de mise à jour :

TA et **TB** incrémentent une variable **t** initialisée à 5.

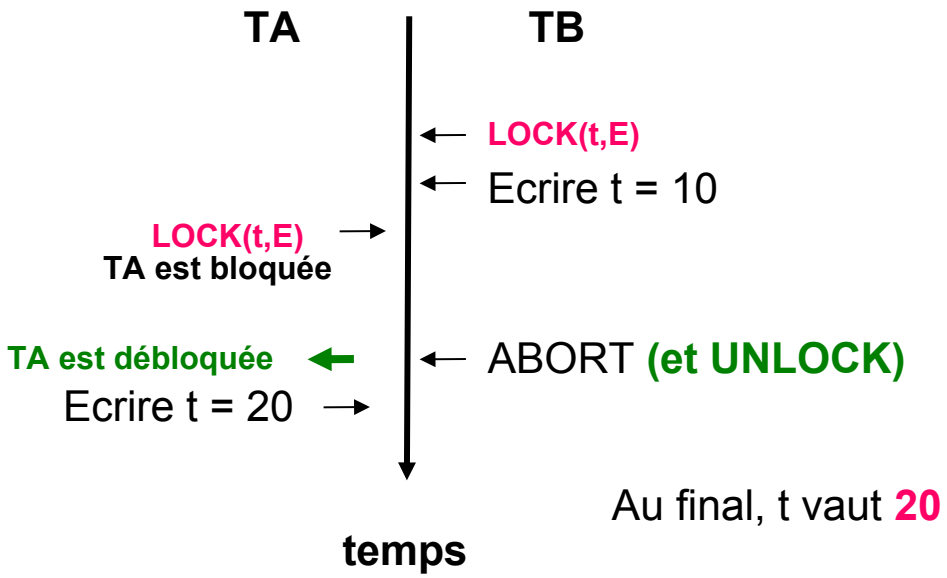


- Les deux mises à jours sont prises en compte.
- Poser d'abord un verrou en lecture mène à un autre problème.

3. Gestion de la concurrence. 2. le verrouillage.

Perte de mise à jour :

TA et **TB** modifient une variable **t** initialisée à 5.

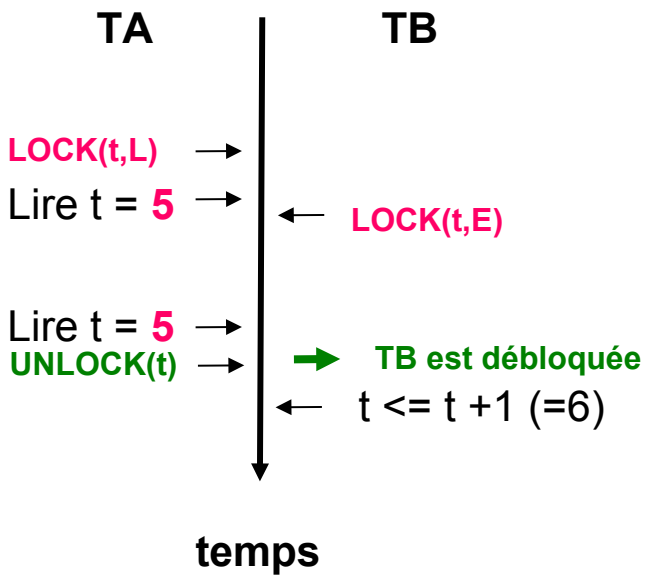


La mise à jour de TA n'est pas perdue.

3. Gestion de la concurrence. 2. le verrouillage.

Lecture non répétable :

TA lit **t, TB** incrémente une variable **t** initialisée à 5.

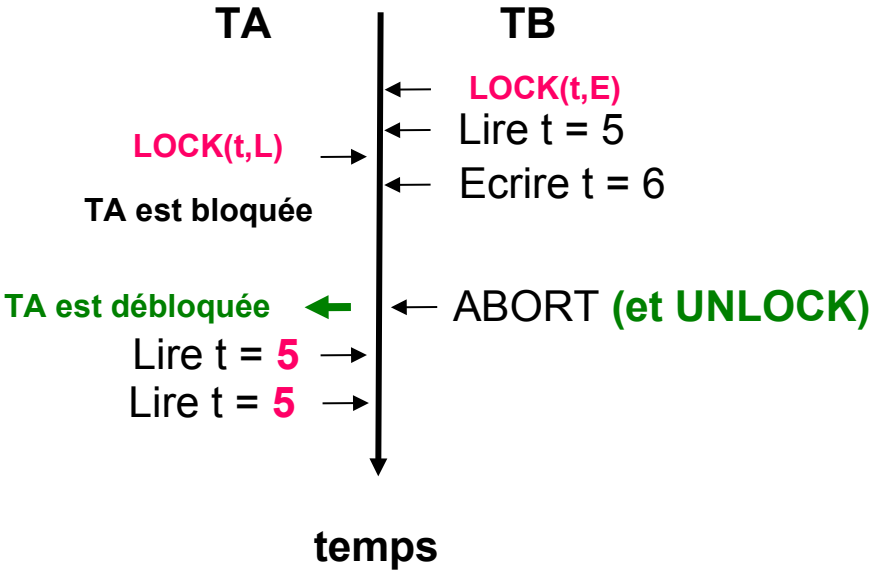


La valeur lue ne change pas entre deux lectures

3. Gestion de la concurrence. 2. Le verrouillage.

Lecture sale :

TA lit **t, TB** incrémente une variable **t** initialisée à 5.



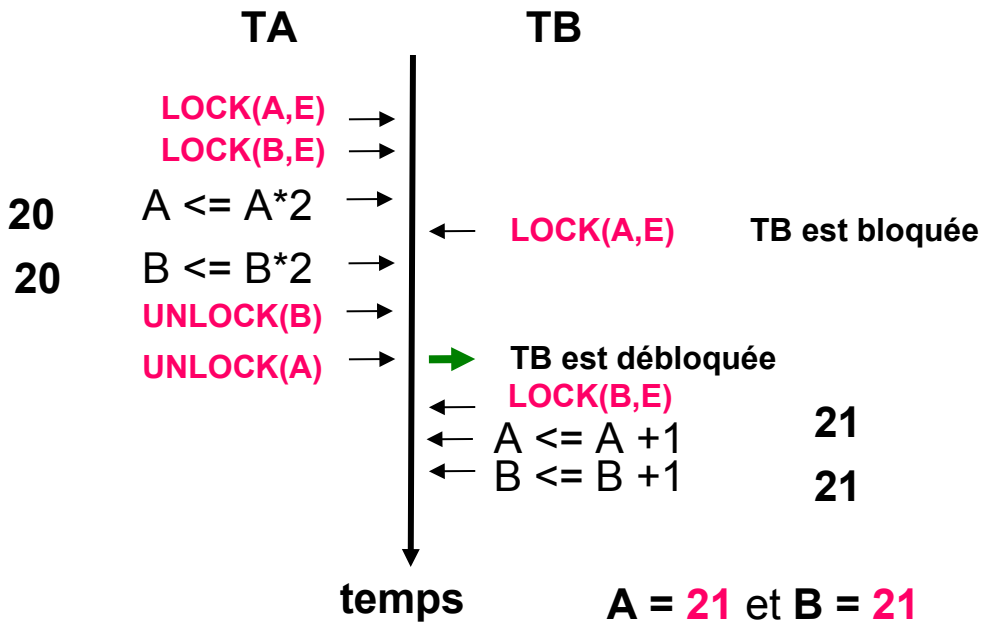
Impossible de lire la valeur n'ayant jamais existé.

3. Gestion de la concurrence. 2. Le verrouillage.

Incohérence :

Deux variables **A** et **B** devant rester égales et initialisées à 10

TA multiplie **A** et **B** par 2. **TB** incrémente **A** et **B**.

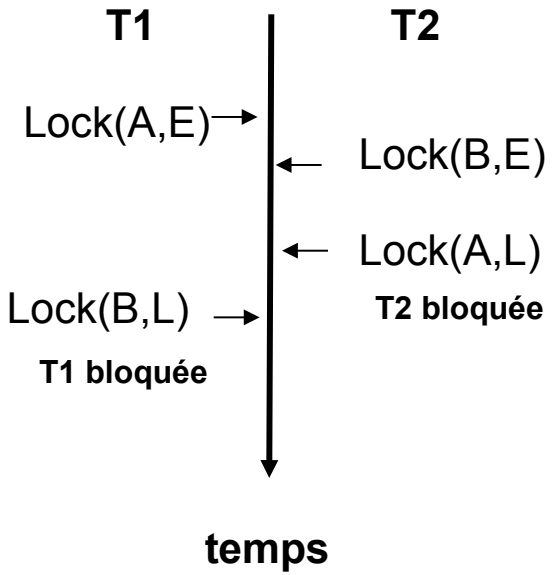


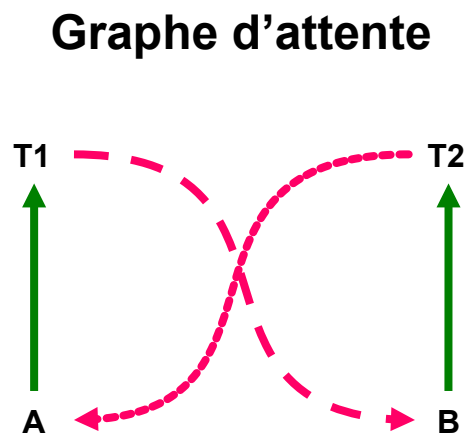
- La contrainte est respectée après TA et TB.
- Poser les verrous dans un autre ordre pourrait poser problème.

3. Gestion de la concurrence. 2. Le verrouillage

Inconvénient : risque de verrou mortel.

verrou mortel : ensemble de transactions bloquées, chacune attendant qu'une autre libère un verrou pour pouvoir continuer.





Verrou Mortel = circuit dans le graphe d'attente.

3. Gestion de la concurrence. 2. Le verrouillage

Traitement du verrou mortel :

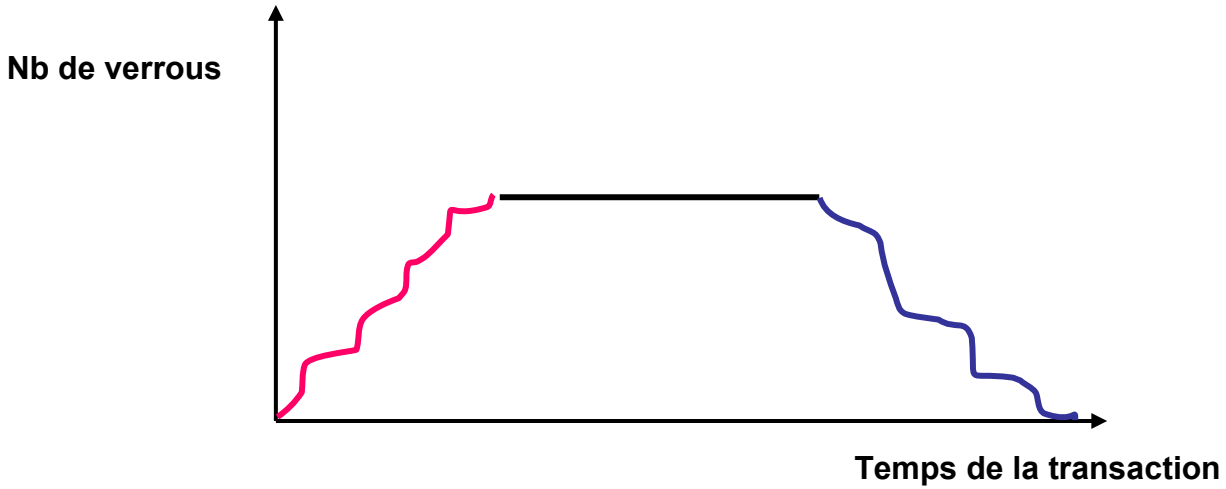
- Méthode brute :
  - On ne fait rien tout en gérant le graphe d'attente,
  - Lorsqu'un verrou mortel est détecté, on annule une transaction du cycle pour la reprendre plus tard.

⇒ Beaucoup de perte de temps => autres méthodes.

3. Gestion de la concurrence. 3. Prévention du verrou mortel

a) Verrouillage 2 phases.

Un phase où l'on pose les verrous, une phase où les verrous sont relâchés. Quand on relâche un verrou, on n'en prend plus jamais.



Remarques :

- N'évite pas à 100% le verrou mortel !
- En pratique tous les verrous sont relâchés au Commit ou Abort.

3. Gestion de la concurrence. 3. Prévention du verrou mortel

b) Une méthode de prévention.

Principe : annuler les transactions demandant des ressources tenues par les plus anciennes. *(La politesse demande que l'on s'efface devant les anciens).*

1. **Estampiller les transactions :**

1. **Traitement du Lock :** la transaction **T1** veut faire un **Lock(G).**
   - Si **G** est libre alors **T1** le bloque et continue.
   - Si **G** est tenu par une transaction **T2** plus récente que **T1**, alors **T1** attend.
   - Si **G** est tenu par une transaction **T2** plus ancienne que **T1**, alors **T1** s'annule, et repart après la fin de **T2**, en gardant son estampille

3. Gestion de la concurrence. 3. Prévention du verrou mortel

Propriété : ***Impossible qu'une transaction jeune attende une plus ancienne.***

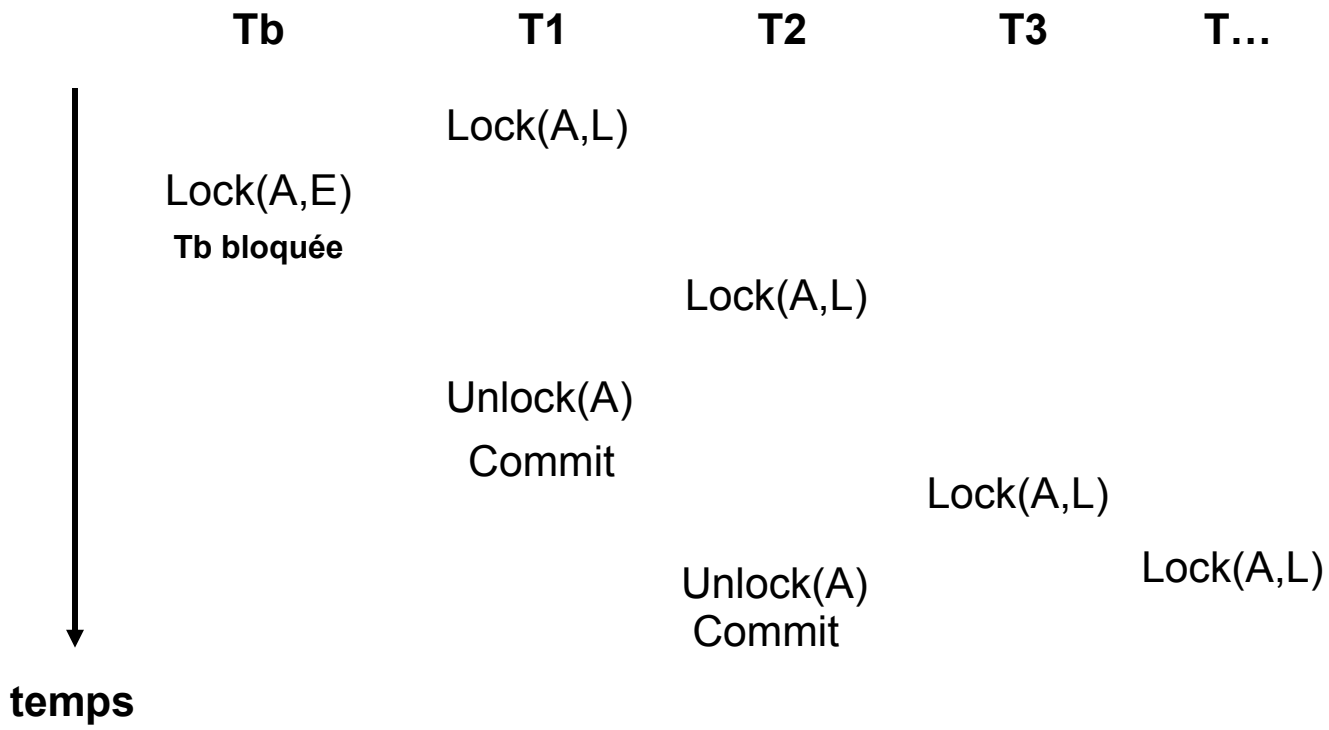
⇒ **Il ne peut y avoir de circuit dans le graphe d'attente.**

⇒ **Il ne peut y avoir de verrou mortel.**

Inconvénient : ***Il faut défaire et reprendre de nombreuses transactions, parfois inutilement.***

⇒ **Autres méthodes plus subtiles.**

Famine : Une transaction attend indéfiniment.



Remarque : La méthode de prévention évite la famine (ce n'est pas le cas de toutes les méthodes).

3. Gestion de la concurrence. 4. Autres problèmes du verrouillage.

Fantômes : des enregistrements apparaissent…

Exemple :

- Une table billets, les transactions verrouillent, et le granule est l'enregistrement.
- Deux transactions :
  - **T1** affiche deux fois de suite les billets du vol 100 ;
  - **T2** crée un billet **(100, Nono)**

initialement

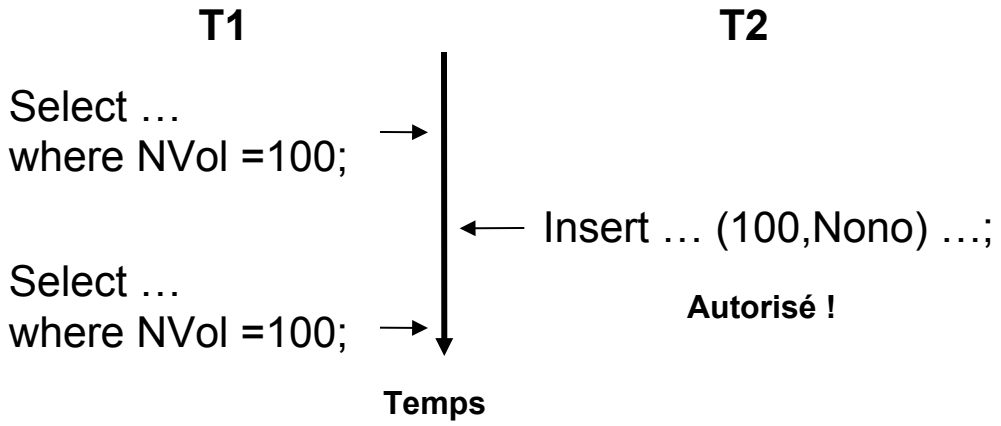
NVOL	CLIENT
100	Titi
103	Tata
100	Toto
100	Tutu
103	Titi

Le 1er select de T1 affiche et verrouille les enregistrements :

100	**Titi**
100	**Toto**
100	**Tutu**

Le 2eme select affiche :

100	**Titi**
100	**Toto**
100	**Tutu**
100	***Nono***



4. Les commandes en SQL.

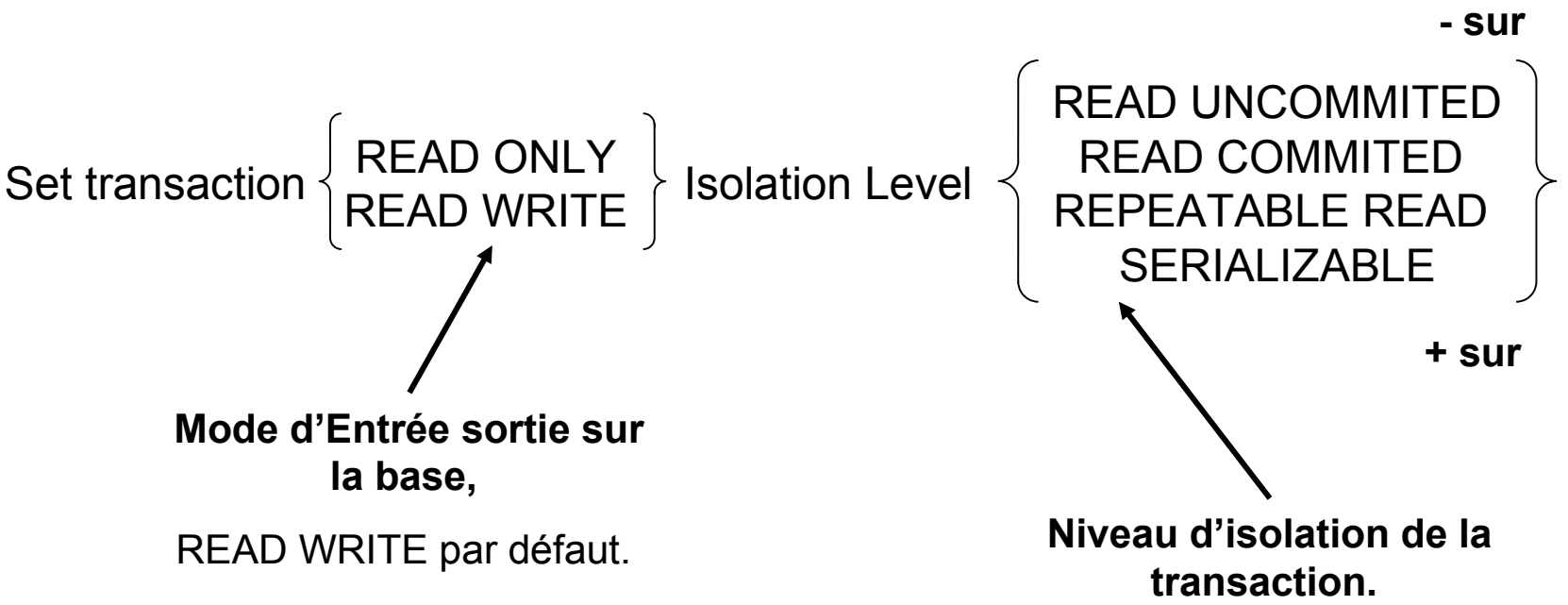
Très dépendant du SGBD.

4. Les commandes en SQL. 1. Commit Abort et autres.

Voir TP.

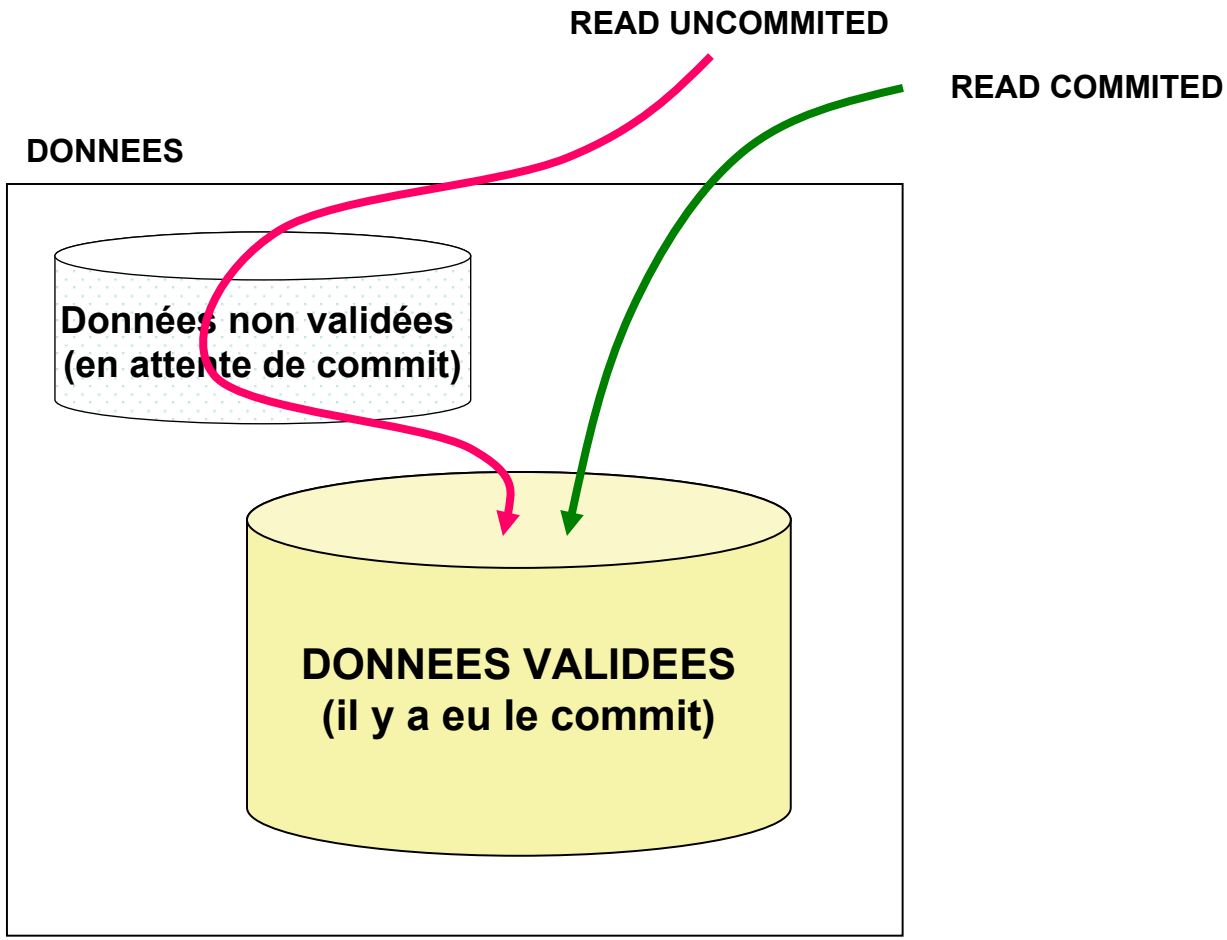
4. Les commandes en SQL. 2. Concurrence

Une commande définissant le comportement de la transaction vis-à-vis de la base et des autres transactions.



4. Les commandes en SQL. 2. Concurrence

Commited/Uncommited ?



4. Les commandes en SQL. 2. Concurrence

Problèmes garantis par les différents niveaux d'isolation :

- Lecture sale
- Lecture non répétable
- Fantômes

	Lecture Sale	Lecture non répétable	fantômes
SERIALIZABLE	N	N	N
REPEATABLE READ	N	N	O
READ COMMITED	N	O	O
READ UNCOMMITED	O	O	O

4. Les commandes en SQL. 2. Concurrence

Attention :

Niveau d’isolation + élevé = risque d’attente élevé....