

Chapitre 4

Transactions et gestion des accès concurrents

Plan

I. INTRODUCTION AUX TRANSACTIONS

1. Notion de transaction
2. Propriétés d'une transaction
3. États d'une transaction
4. Une transaction sous Oracle
5. Exercice d'application

II. ACCÈS CONCURRENTS

1. Problèmes des accès concurrents
2. Contrôle des accès
 - a. Le verrouillage
 - b. L'interblocage
 - c. Résolution de l'interblocage
 - d. Limiter le temps de verrouillage
 - e. Mise en œuvre sous Oracle
3. Exercice d'application

I. Introduction aux transactions

- Dans une BD, plusieurs utilisateurs doivent pouvoir accéder à la BD en même temps ce qui engendre un **problème d'accès concurrents**.

Gestion des transactions

I. Introduction aux transactions

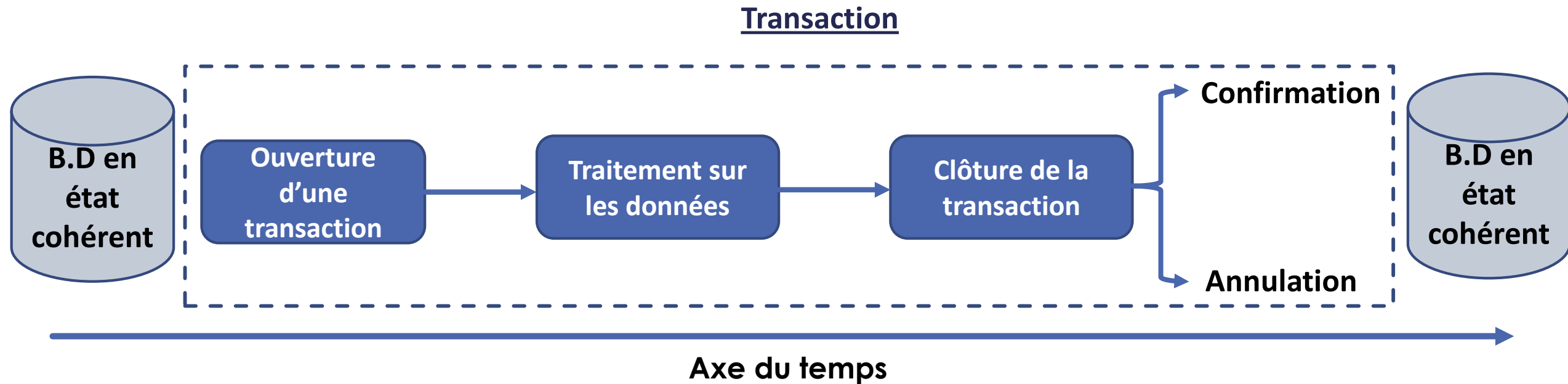
1. Notion de transaction

- Une transaction est une suite/séquence d'opérations (lecture/ écriture) qui doit être exécutée **dans son intégralité**.
- Toute la transaction est réalisée ou rien ne l'est, elle est soit :
 - **Validée** : toute la transaction est prise en compte.
 - **Avortée ou annulée** : la transaction n'a aucun effet.
- Une transaction amène la BD d'un **état cohérent** à un **autre état cohérent**.
 - S'il n'est pas possible d'accéder à un état cohérent, **rien ne doit être fait**.

I. Introduction aux transactions

1. Notion de transaction (2)

- Cycle de vie d'une transaction :



1. Introduction aux transactions

2. Propriétés d'une transaction - ACID

- **Atomicité**

- Toutes **les modifications** effectuées par une transaction **sont enregistrées** dans la BD ou aucune. **En cas d'échec**, le système doit **annuler toutes les modifications** réalisées.

- **Cohérence**

- Une transaction fait passer une BD **d'un état cohérent** à un **autre état cohérent**. En cas d'échec, il faut **restaurer** l'état initial cohérent.
- Un état cohérent est un état dans lequel les **contraintes d'intégrité sont vérifiées**.

1. Introduction aux transactions

2. Propriétés d'une transaction – ACID (2)

- **Isolation**

- Une transaction se déroule **sans être perturbée** par les transactions concurrentes.
- Les **accès concurrents** peuvent mettre en question l'isolation.

- **Durabilité**

- Les **modifications** d'une transaction validée **sont persistantes**.
- Le principal problème de la durabilité survient en cas de panne disque (chapitre suivant).

I. Introduction aux transactions

3. Etats d'une transaction

Etat	Description
Active	La transaction reste dans cet état durant son exécution.
Partiellement validée	Juste après l'exécution de sa dernière opération.
Echec	Après avoir découvert qu'une exécution <i>normale</i> ne peut pas avoir lieu (erreur).
Avortée	<p>Après que toutes les modifications faites soient annulées (Rollback).</p> <p>Dans ce cas deux actions sont possibles :</p> <ul style="list-style-type: none">■ Réexécuter la transaction.■ Tuer la transaction.
Validée	Après l'exécution avec succès de la dernière opération et confirmation (Commit).

I. Introduction aux transactions

4. Une transaction sous Oracle

- **Début / Ouverture d'une transaction :**
 - A la connexion à la BD ou à la fin de la transaction précédente.
- **Fin / Clôture d'une transaction :**
 - Soit avec une **validation** (avec **Commit**) ou une **annulation** (avec **Rollback**).
 - Instructions du **LDD** (CREATE, ALTER, DROP, ..) sont suivies d'un **Commit implicite**, on ne peut donc pas faire d'annulation au niveau du **LDD**.
 - La déconnexion entraîne aussi un **Commit** de la transaction en cours.

I. Introduction aux transactions

4. Une transaction sous Oracle (a)

- Une transaction sous Oracle peut être :
- **Une instruction isolée** : en sauvant les données modifiées à la fin de chaque instruction, tel qu'une instruction est isolée en une seule transaction propre et se termine par un **Commit** ou avec un **Autocommit** qui ajoute automatiquement un **Commit** à la fin de chaque instruction.

```
Sql> INSERT INTO emp VALUES (1111, 'John', 'Clerk');
```

```
Sql> COMMIT;
```

```
Sql> INSERT INTO emp VALUES (2222, 'Smith', 'Analyst');
```

```
Sql> COMMIT;
```

```
Sql> Set Autocommit on;
```

```
Sql> INSERT INTO emp VALUES (1111, 'John', 'Clerk');
```

```
Sql> INSERT INTO emp VALUES (2222, 'Smith', 'Analyst');
```

I. Introduction aux transactions

4. Une transaction sous Oracle (b)

- Une transaction sous Oracle peut être aussi :
 - **Un groupe/bloc d'instructions** : mettre un seul **Commit** à la fin d'un **bloc d'instructions** pour valider toutes les mises à jour.

```
Sql> INSERT INTO emp VALUES (1111, 'Adams', 'Clerk');  
Sql> INSERT INTO emp VALUES (2222, 'Jones', 'Manager');  
...  
Sql> COMMIT;
```

I. Introduction aux transactions

4. Une transaction sous Oracle

- A la fin d'une transaction sous Oracle, on peut annuler **une partie** des mises à jour, en insérant des points de repère, ou **SAVEPOINT**.

- **La création d'un point de repère :**

Sql> SAVEPOINT nom_pointRepère;

- **Pour annuler la partie de la transaction depuis un point de repère :**

Sql> ROLLBACK TO [SAVEPOINT] nom_pointRepère;

1. Introduction aux transactions

5. Exercice d'application

- Les usines **GARVEL** construisent des figurines de super-héros à partir des données présentes dans la BD de l'entreprise. Un gros problème est survenu le mois dernier, lorsque l'usine en charge d'une nouvelle figurine, "*Superchild*", a livré un million d'exemplaires **sans tête**.
- À l'analyse de la base, il a en effet été observé que la base contenait un tuple "*Superchild*" dans la table **Personnage**, et cinq tuples associés dans la table **Membre**, deux pour les bras, deux pour les jambes et un pour le torse, **mais aucun tuple pour la tête**.

I. Introduction aux transactions

5. Exercice d'application (suite)

- Le service qui a opéré la saisie du nouveau personnage assure, sans ambiguïté possible, que la tête a pourtant été saisie dans la base (voir script). En revanche, l'enquête montre des **instabilités de son réseau à cette période.**

I. Introduction aux transactions

5. Exercice d'application (suite)

--Script de l'insertion

SET AUTOCOMMIT ON;

INSERT INTO *Personnage* (designation, prix, identite_secrete, genre) VALUES ('Superchild', '12', 'Jordy', 'superhéros');

INSERT INTO *Membre* (propriétaire, nom, couleur) VALUES ('Superchild', 'bras droit', 'bleu');

INSERT INTO *Membre* (propriétaire, nom, couleur) VALUES ('Superchild', 'bras gauche', 'bleu');

INSERT INTO *Membre* (propriétaire, nom, couleur) VALUES ('Superchild', 'jambe droite', 'bleu');

INSERT INTO *Membre* (propriétaire, nom, couleur) VALUES ('Superchild', 'jambe gauche', 'bleu');

INSERT INTO *Membre* (propriétaire, nom, couleur) VALUES ('Superchild', 'torse', 'bleu');

INSERT INTO *Membre* (propriétaire, nom, couleur) VALUES ('Superchild', 'tête', 'bleu');

I. Introduction aux transactions

5. Exercice d'application (suite)

- **Question 1**
 - Expliquer la nature du problème qui est probablement survenu.
 - Proposer une solution générale pour que le problème ne se renouvelle pas, en expliquant pourquoi.
- **Question 2**
 - Illustrer la solution proposée en corrigeant le code SQL de l'insertion de "Superchild".

I. Introduction aux transactions

5. Exercice d'application – Solution

- Q1 : Expliquer la nature du problème qui est probablement survenu.
 - Le script est en mode **AUTOCOMMIT** ce qui signifie que chaque instruction est isolée dans une transaction propre, un COMMIT implicite étant exécuté après chaque INSERT.
 - Une panne ou une défaillance système (une coupure réseau) est survenue juste avant le dernier INSERT, celui de la tête, ce qui a empêché son exécution et les six premiers INSERT ont donc été exécutées et validées par le COMMIT implicite.

I. Introduction aux transactions

5. Exercice d'application – Solution (suite)

- Q1 : Proposer une solution générale pour que le problème ne se renouvelle pas, en expliquant pourquoi.
 - **Désactiver l'AUTOCOMMIT** et regrouper les instructions en une seule transaction de sorte que si un problème survient lors de l'exécution de l'une des instructions toute la transaction sera avortée.

I. Introduction aux transactions

5. Exercice d'application – Solution (suite)

- Q2 : Illustrer la solution proposée en corrigeant le code SQL de l'insertion de "Superchild".

- Désactiver le AUTOCOMMIT

Sql> **Set Autocommit Off;**

- Regrouper les insertions en une seule transaction.

INSERT INTO *Personnage* (...);

INSERT INTO *Membre* (propriétaire, nom, couleur) VALUES ('Superchild', 'bras droit', 'bleu');

...

INSERT INTO *Membre* (propriétaire, nom, couleur) VALUES ('Superchild', 'tête', 'bleu');

Commit;

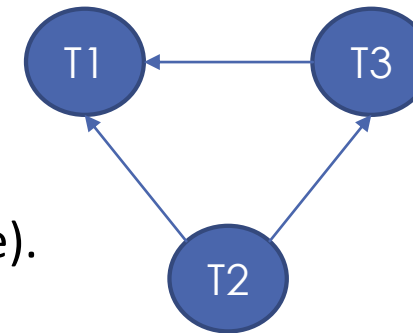
II. Accès concurrents

- On parle **d'accès concurrents** quand plusieurs transactions (plusieurs utilisateurs) tentent **d'accéder à la même donnée en même temps**.
- La théorie de la concurrence permet de garantir la **cohérence** et **l'isolation** des transactions.
- Il faut gérer ces accès concurrents afin de :
 - Garantir que l'exécution **simultanée** des transactions produit le même résultat que leur exécution en **séquentiel** (l'une puis l'autre). C'est la **sérialisabilité**.
 - Rendre invisible aux utilisateurs le partage simultané des données.

II. Accès concurrents Sérialisabilité – Exemple

- Soit l'exécution suivante **E** : **w2[x] w3[z] w2[y] r1[x] w1[z] r3[y]**
- H : Les conflits
 - Sur x : **w2[x] r1[x]** T2, T1
 - Sur y : **w2[y] r3[y]** T2, T3
 - Sur z : **w3[z] w1[z]** T3, T1
- H est sérialisable car le graphe n'admet pas de cycle (acyclique).
- H est équivalente à l'exécution en série de **T2 T3 T1**.

1,2 et 3 : N° transaction
w: Opération d'écriture
r : Opération de lecture
x, y et z : des tuples



Graphe de dépendances

II. 1. Problèmes des accès concurrents

Perte des mises à jour (*Lost Update*)

Axe du temps ↓

Transactions Utilisateur 1	Transactions Utilisateur 2	Valeur de la donnée Initialement $X = 10$
Lire X -- la valeur 10 est lue		
	Lire X -- la valeur 10 est lue	
Mettre à jour $X = X + 20$		
Afficher X		$X = 30$
	Mettre à jour $X = X + 30$	
	Afficher X	$X = 40$



Les modifications effectuées par l'utilisateur 1 sont perdues

II. 1. Problèmes des accès concurrents

Lecture impropre (*dirty read*)

Axe du temps

Transactions Utilisateur 1	Transactions Utilisateur 2	Valeur de la donnée Initialement X = 10
	Mettre à jour X = 20	
	Afficher X	X = 20
Lire X -- la valeur 20 est lue		
	Annuler la mise à jour (rollback)	X = 10



La valeur de X lue par l'utilisateur 1 est impropre car elle est non confirmée

II. 1. Problèmes des accès concurrents

Lecture non reproductible (*Unrepeatable read*)

Axe du temps ↓	Transactions Utilisateur 1	Transactions Utilisateur 2	Valeur de la donnée Initialement X = 10
		Lire x --La valeur 10 est lue	
		Afficher X	X = 10
	Mettre à jour X = X+30		
	Afficher X		X = 40
		Lire x --La valeur 40 est lue	



L'utilisateur 2 lit deux valeurs différentes de X

II. 1. Problèmes des accès concurrents

Lecture fantôme (*phantom read*)

- Exemple :
 - Une transaction de l'utilisateur 1 ajoute un employé travaillant sur le projet X alors que la transaction de l'utilisateur 2 demande la liste des employés du projet X afin de faire un calcul (somme des heures de travail, total des salaires, ...)
 - **La transaction effectuée par l'utilisateur 2 peut ne pas comptabiliser les données du nouvel employé, ajouté par l'utilisateur 1, dans son calcul.**

II. 2. Contrôle des accès

a. Le verrouillage

- Le verrouillage (**Lock**) est la technique la plus populaire pour **résoudre** les problèmes dus aux **accès concurrents**.
- Cette technique permet de **synchroniser** les transactions concurrentielles.
- L'utilisateur peut poser ses propres verrous sur ses propres tables ou toutes les tables s'il est DBA.
- **Principe :**
 - Avant de lire ou écrire une donnée, une transaction peut demander un **verrou** sur cette donnée pour interdire l'accès aux autres transactions → elle **bloque** la donnée.
 - Si le verrou ne peut être obtenu, parce qu'une autre transaction en possède un sur cette donnée, la transaction demandeuse est **mise en attente**.

II. 2. Contrôle des accès

a. Le verrouillage (2)

- 2 phases de verrouillage d'une transaction :
 - **Acquisition** des verrous.
 - **Relâchement** des verrous :
 - **Verrou long** : Pour garantir l'isolation des mises à jour, les verrous sont généralement relâchés à la clôture d'une transaction (avec un **Commit** ou un **Rollback**).
 - **Verrou court** : relâchement après exécution du traitement sur la donnée.

II. 2. Contrôle des accès

a. Le verrouillage (3)

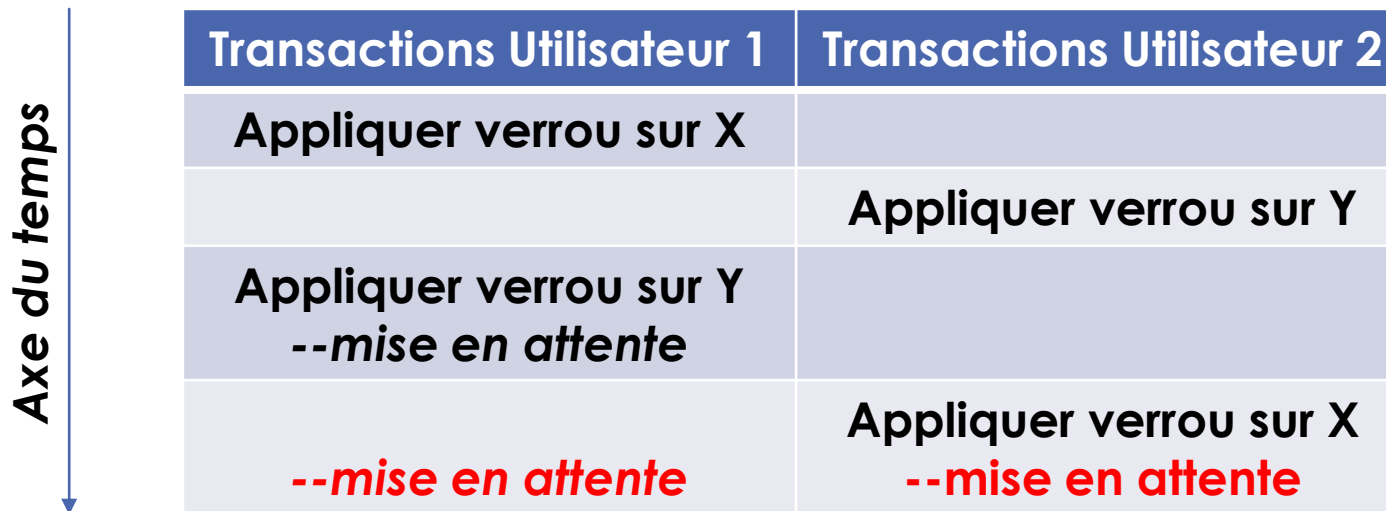
Axe du temps

Transactions Utilisateur 1	Transactions Utilisateur 2	Valeur de la donnée Initialement X = 10
Lire X avec un verrou --La valeur 10 est lue		
Mettre à jour X = X + 20	Lire X avec verrou -- mise en attente	
Afficher X Valider avec Commit		X = 30
	Lire X avec verrou --La valeur 30 est lue	
	Mettre à jour X = X + 30	
	Afficher X Valider avec Commit	X = 60

II. 2. Contrôle des accès

b. L'interblocage

- Le problème d'interblocage le **deadlock** ou le **verrou mortel** :
 - Chaque transaction attend que l'autre transaction libère un élément.



Transactions Utilisateur 1	Transactions Utilisateur 2
Appliquer verrou sur X	
	Appliquer verrou sur Y
Appliquer verrou sur Y --mise en attente	
--mise en attente	Appliquer verrou sur X --mise en attente

- Pour résoudre ce problème, une des deux transactions doit être **avortée** et **ses verrous libérés**.

II. 2. Contrôle des accès

c. Résolution de l'interblocage – Prévention

- L'approche de la prévention

Appliquer une **priorité** aux transactions **des plus vieilles aux moins vieilles** :

- **Estampillage** : attributions des numéros uniques (*timestamp*) aux transactions pour les ordonner \Rightarrow **Estampiller chaque transaction avec l'heure de son lancement.**
- 2 algorithmes sont utilisés :
 - ***Wait-Die*** : Si **T1** a une **priorité supérieure** alors **T1 attend T2**, sinon **T1 est tuée.**
 - ***Wound-Wait (ou Wound-Die)*** : Si **T1** a une **priorité supérieure** alors **T2 est tuée** sinon **T1 attend.**

II. 2. Contrôle des accès

c. Résolution de l'interblocage – Prévention

- On considère deux transactions **T1** et **T2** et une ressource **r** utilisée par **T1** et demandée par **T2**.
- A la création de chaque transaction une estampille lui est associée : soient $e(T1)$ et $e(T2)$ les estampilles des deux transactions.
- **L'algorithme *Wait-Die* :**
 - Si $e(T2) > e(T1)$ // *T1 plus vieille que T2 donc T1 est prioritaire sur T2*
 - Alors **WAIT**(T1) // *T1 est mise en attente*
 - Sinon **ROLLBACK**(T1) // *T1 est tuée et sera relancée avec l'estampille $e(T1)$*
- **L'algorithme *Wound-Wait* :**
 - Si $e(T2) > e(T1)$ // *T1 plus vieille que T2 donc T1 est prioritaire sur T2*
 - Alors **ROLLBACK**(T2) // *T1 tue T2 et T2 sera relancée avec l'estampille $e(T2)$*
 - Sinon **WAIT**(T1) // *T1 est mise en attente*

II. 2. Contrôle des accès

c. Résolution de l'interblocage – Prévention

Explications :

- **Dans W-D :**
 - si la transaction **T1 est la plus ancienne et elle détient la ressource** alors **T1** reste bloquée (en attente) jusqu'à ce que **T2** finit son exécution (Commit ou Rollback) et libère la ressource.
 - sinon si **T2 est la plus ancienne et demande la ressource** alors **T1** (qui détient la ressource mais elle est plus récente) sera abandonnée et sera redémarrée ultérieurement avec le même numéro d'estampille, et **T2** finit son exécution (Commit ou Rollback).
- **Dans W-W :**
 - On privilégie les transactions les plus anciennes et donc d'annuler les plus récentes.
 - Si **T1 est la plus ancienne et elle détient la ressource** alors **T2** sera annulée et **T1** finit son exécution (Commit ou Rollback).
 - Si **T2 est la plus ancienne mais ne détient pas la ressource** (elle la demande) donc **T1** sera mise en attente et **T2** finit son exécution (Commit ou Rollback).

II. 2. Contrôle des accès

c. Résolution de l'interblocage – Détection

- L'approche de la détection

Identifier des **cycles** dans les graphes d'attentes :

- Les **nœuds** correspondent aux **transactions**.
- Les **arcs** représentent les **attentes** entre transactions (pour poser un verrou).
- Si il y a un **cycle** dans un graphe d'attente, il y a une situation de **verrou mortel**.
- Un **cycle** signifie un interblocage entre les transactions représentées par les nœuds du cycle.
- Une transaction dont le **temps d'attente dépasse un certain seuil**, et **annulée (Rollback)** et est **relancée** un peu plus tard.

II. 2. Contrôle des accès

d. Limiter le temps d'attente du verrouillage

- La granularité du verrouillage
 - Pour restreindre la taille de la donnée verrouillée :
 - Verrouillage ligne : *Row Level Locking*
 - Verrouillage table : *Table Level Locking*

II. 2. Contrôle des accès

d. Limiter le temps d'attente du verrouillage (2)

- Le mode de verrouillage
 - Pour restreindre les opérations interdites sur la donnée verrouillée.
 - Le mode de verrou partagé (*S - Share lock mode*) : est posé par une transaction lors d'un accès **en lecture** sur une ressource. Un verrou partagé interdit aux autres transaction de poser un verrou exclusif sur cet objet et donc d'y accéder en écriture.
 - Le mode de verrou exclusif (*X - Exclusive lock mode*) : est posé par une transaction lors d'un accès **en écriture** sur une ressource. Un verrou exclusif interdit aux autres transactions de poser tout autre verrou (partagé ou exclusif) sur cet objet et donc d'y accéder (ni en lecture, ni en écriture).

II. 2. Contrôle des accès

e. Mise en œuvre sous Oracle

	Les verrous de <u>partage</u>	Les verrous <u>exclusifs</u>
Verrouillage <u>ligne</u>	ROW SHARE TABLE LOCKS (RS)	ROW EXCLUSIVE TABLE LOCKS (RX)
Verrouillage <u>table</u>	SHARE TABLE LOCKS (S)	EXCLUSIVE TABLE LOCKS (X)

N.B : Lorsqu'une transaction se termine (COMMIT ou ROLLBACK) elle libère tous les verrous qu'elle a posé.

II. 2. Contrôle des accès

e. Mise en œuvre sous Oracle (2)

Exemple :

Table *emp*

EMPNO	ENAME	JOB

7369	Smith	CLERK
7499	Allen	SALESMAN
7521	Ward	SALESMAN
7566	Jones	MANAGER
7654	Martin	SALESMAN
7698	Blake	MANAGER
7782	Clark	MANAGER
7788	Scott	ANALYST
7839	King	PRESIDENT
7844	Turner	SALESMAN
7876	Adams	CLERK
7900	James	TEST
7902	Ford	ANALYST
7934	Miller	CLERK

II. 2. Contrôle des accès

e. Mise en œuvre sous Oracle (3)

- **Le verrou ROW SHARE TABLE LOCKS (RS) :**

- Indique que la **transaction a verrouillé des lignes** dans une table pour des mises à jour.
- Les commandes :

`SELECT ... FROM ... FOR UPDATE OF...;`

`LOCK TABLE ... IN ROW SHARE MODE;`

😊 Opérations permises : permet aux autres transactions les opérations suivantes : select, insert, update, delete et obtenir un verrou RS ou RX **sur les lignes non verrouillées de la même table**, ou un verrou S sur toute la table.

😞 Opération interdites : empêche les autres transactions d'avoir un accès exclusif X sur les mêmes données de la table.

Session 1



--VERROU : ROW SHARE TABLE LOCKS (RS)

SELECT job FROM emp WHERE job = 'CLERK'
FOR UPDATE OF empno;
LOCK TABLE emp IN Row SHARE MODE;
--OK

Session 2



SELECT job FROM emp WHERE job = 'CLERK' FOR UPDATE OF empno;
--en attente

SELECT job FROM emp WHERE job = 'MANAGER' FOR UPDATE OF empno;
--OK

LOCK TABLE emp IN SHARE MODE;
--OK

LOCK TABLE emp IN EXCLUSIVE MODE;
--en attente : pas d'accès exclusif aux données verrouillées car un verrou RS est placé dans la session1

INSERT INTO emp (empno, ename, job) VALUES (9999,'Smith', 'Test');
--OK

DELETE FROM emp WHERE empno = 7876; --correspond à CLERK
--En attente car bloqué par Session 1

UPDATE emp SET job='CLIMBER' WHERE empno = 7876; --correspond à CLERK
--En attente car bloqué par Session 1

Axe du temps

II. 2. Contrôle des accès

e. Mise en œuvre sous Oracle (4)

- **Le verrou ROW EXCLUSIVE TABLE LOCKS (RX) :**

- Indique qu'une transaction a fait une ou plusieurs **mises à jour** sur la table.

- Les commandes :

`INSERT INTO table .../ UPDATE table .../ DELETE FROM table ...`

`LOCK TABLE ... IN ROW EXCLUSIVE MODE;`

😊 **Opérations permises** : permet aux autres transactions les opérations suivantes : select, insert, update et delete **sur les lignes non verrouillées sur la même table**. Autorise d'autres transactions à obtenir des verrous RS et RX pour la même table **sur les lignes non verrouillées sur la même table**.

😞 **Opérations interdites** : empêche les autres transactions de verrouiller manuellement la table en exclusif ou en partage.

Session 1



--VERROU : ROW EXCLUSIVE TABLE LOCKS (RX)

UPDATE emp SET ename = 'Toto' WHERE empno='7369';

LOCK TABLE emp IN ROW EXCLUSIVE MODE;

--OK

Session 2



LOCK TABLE emp IN EXCLUSIVE MODE;

--en attente : pas de verrouillage manuel exclusif car il y a un verrou RX placé dans la session 1

LOCK TABLE emp IN SHARE MODE;

--en attente : pas de verrouillage manuel de partage car il y a un verrou RX placé dans la session 1

INSERT INTO emp (empno, ename, job) VALUES (9999,'Smith', 'Test');

--OK

DELETE FROM emp WHERE empno = 9999;

--OK

LOCK TABLE emp IN ROW SHARE MODE;

--OK

Axe du temps

II. 2. Contrôle des accès

e. Mise en œuvre sous Oracle (5)

- Le verrou *SHARE TABLE LOCKS (S)* :

- La commande :

LOCK TABLE ... IN SHARE MODE;

- 😊 Opérations permises : permet aux autres transactions les opérations suivantes : select autorise d'autres transactions à obtenir un verrou S pour la même table.
- 😞 Opérations interdites : empêche les mises à jours sur la table, empêche les autres transactions de verrouiller en exclusif RX ou en partage RS des lignes de la même table et de verrouiller en exclusif X la table.

Session 1



--VERROU : SHARE TABLE LOCKS (S)

LOCK TABLE emp IN SHARE MODE;

--OK

UPDATE emp SET ename='Zahn' WHERE empno = 7900;
COMMIT;

--OK

LOCK TABLE emp IN SHARE MODE;

--OK

UPDATE emp SET ename='Muller' WHERE empno = 7900;
--en attente car un verrou S placé dans Session 2 empêche les mises à jour.

Session 2



SELECT * FROM emp;

--OK

LOCK TABLE emp IN SHARE MODE;

--OK car la transaction de Session 1 a été validée avec le Commit

Axe du temps

II. b. Contrôle des accès

e. Mise en œuvre sous Oracle (6)

- **Le verrou EXCLUSIVE TABLE LOCKS (X) :**
 - Il est obtenu automatiquement pour la 1^{ère} transaction sur la table pour mettre à jour les données (update ou delete).
 - Une **seule transaction** à la fois peut acquérir un **verrou X** sur la table.
 - La commande :

`LOCK TABLE ... IN EXCLUSIVE MODE;`
- 😊 Opérations permises : permet aux autres transactions d'interroger la table uniquement (Select).
- 😞 Opérations interdites : empêche les autres transactions de réaliser des opérations de LMD ou de placer d'autres verrous sur la table.

Session 1



--MODE : EXCLUSIVE TABLE LOCKS (X)

LOCK TABLE emp IN EXCLUSIVE MODE;

--OK

UPDATE emp SET ename='Test' WHERE empno = 7900;

--OK

LOCK TABLE emp IN EXCLUSIVE MODE;

--OK

Session 2



SELECT * FROM emp;

--OK

LOCK TABLE emp IN SHARE MODE;

--en attente de la validation avec le Commit de la transaction de Session 1

LOCK TABLE emp IN EXCLUSIVE MODE;

--en attente car le verrou X placé dans la Session 1 empêche le placement d'autres verrous sur la table

UPDATE emp SET ename='Test' WHERE empno = 7900;

--en attente car le verrou X placé dans la Session 1 empêche les mises à jour des données dans la table

Axe du temps

II. b. Contrôle des accès

e. Mise en œuvre sous Oracle (7)

- Sous Oracle, La vue **V\$LOCK** du dictionnaire des données répertorie les verrous actifs de la base de données :

ADDR	Address of lock state object
KADDR	Address of lock
SID	Identifier for session holding or acquiring the lock
TYPE	Type of user or system lock
ID1	Lock identifier #1 (depends on type)
ID2	Lock identifier #2 (depends on type)
LMODE	Lock mode in which the session holds the lock
REQUEST	Lock mode in which the process requests the lock
CTIME	Time since current mode was granted
BLOCK	Indicates whether the lock in question is blocking other processes.

II. 3. Exercice d'application

- Énoncé :
 - Soit la table **Spectacle** suivante, définie en relationnel, permettant d'enregistrer le nombre d'entrées aux spectacles de la BD. Les Spectacles sont identifiés par leurs NumSpec.
Spectacle (NumSpec, entrées)
 - Soit l'exécution concurrente de deux transactions **TR1** et **TR2** visant à ajouter chacune une entrée au spectacle dont le NumSpec = '123' :

	TR1	TR2
t0	CONNECT	
t1		CONNECT
t2	UPDATE Spectacle SET entrées = entrées+1 WHERE NumSpec = '123';	
t3		
t4		UPDATE Spectacle SET entrées = entrées+1 WHERE NumSpec = '123';
t5		
t6	COMMIT;	
t7		
t8		COMMIT;

N.B : On suppose que les instructions sont reportées au moment où elles sont transmises au serveur et aucune autre transaction n'est en cours d'exécution entre **t0** et **t8**.

II. 3. Exercice d'application

Travail demandé :

- De combien les entrées au spectacle '123' ont-t-elles été augmentées :
 1. à **t3** du point de vue de la transaction **TR1** ?
 2. à **t3** du point de vue de la transaction **TR2** ?
 3. à **t5** du point de vue de la transaction **TR1** ?
 4. à **t5** du point de vue de la transaction **TR2** ?
 5. à **t7** du point de vue de la transaction **TR1** ?
 6. à **t7** du point de vue de la transaction **TR2** ?

II. 3. Exercice d'application

Correction

1. à **t3** du point de vue de la transaction **TR1** ?

1 \Rightarrow **TR1 est en cours, de son point de vue isolé les entrées ont été augmentées de 1.**

2. à **t3** du point de vue de la transaction **TR2** ?

0 \Rightarrow **TR1 n'est pas validée à t3, les modifications ne sont pas encore visibles en dehors de la transaction TR1.**

3. à **t5** du point de vue de la transaction **TR1** ?

1 \Rightarrow **TR1 obtient un verrou X à t2 et exécute son UPDATE.**

TR2 est mise en attente avant son UPDATE à t4, car elle ne peut obtenir de verrou X déjà posé par TR1.

À t5 TR1 est toujours en cours d'exécution, et TR2 toujours bloquée, donc les entrées ont été augmentées de 1 pour le moment du point de vue TR1.

II. 3. Exercice d'application

Correction (suite)

4. à **t5** du point de vue de la transaction **TR2** ?

0 \Rightarrow TR1 est en cours d'exécution, et **TR2 est toujours bloquée**, donc l'entrée n'a pas été augmentée de son point de vue.

5. à **t7** du point de vue de la transaction **TR1** ?

1 \Rightarrow Le **commit débloquent TR2** qui exécute son update, mais celui-ci n'est toujours pas visible depuis TR1.

6. à **t7** du point de vue de la transaction **TR2** ?

2 \Rightarrow TR2 a pu exécuter son update à t6 dès que **TR1 a libéré le verrou avec son commit**. TR1 est validée, donc la modification est visible, et du point de vue de TR2, la modification en cours de TR2 est également visible.

Exemples d'interblocage

- Soit le schéma relationnel suivant :

Spectacle (id_spectacle, nb_places_disponibles, nb_places_libres, tarif)

Client (id_client, nom, prenom, nb_places_reservees)

Example 1 :

	Session 1	Session 2
t0	Connect	
t1	SELECT * FROM Spectacle WHERE id_spectacle=1; --un verrou S est pose sur Spectacle --lecture faite	
t2	SELECT * FROM Client WHERE id_client=1; --un verrou S est pose sur Client --lecture faite	Connect
t3		SELECT * FROM Spectacle WHERE id_spectacle=1; --un verrou S est pose sur Spectacle --lecture faite
t4		SELECT * FROM Client WHERE id_client=2; --un verrou S est pose sur Spectacle --lecture faite
t5		UPDATE Spectacle SET nb_places_libres=48 WHERE id_spectacle=1; --en attente car elle ne peut pas appliquer un verrou X sur la table Spectacle qui a un verrou S posé lors de la session 1
t6	UPDATE Spectacle SET nb_places_libres=45 WHERE id_spectacle=1; --en attente car elle ne peut pas appliquer un verrou X sur la table Spectacle qui a un verrou S posé lors de la session 2	

Exemple 1

- ▶ **Interblocage** : Les deux transactions peuvent s'attendre indéfiniment l'une l'autre.
- ▶ La transaction de la session 2 était déjà bloquée par les verrous partagés posé par la transaction de la session 1.
- ▶ En voulant modifier le spectacle, la transaction de la session 1 se trouve bloquée à son tour par les verrous partagés posés par la transaction de la session 2.
- ▶ Le SGBD prend la décision pour débloquent la situation, selon l'algorithme appliqué.

Exemple 1

- ▶ La transaction de la session 1 est la plus ancienne **et elle détient la ressource** (c'est la 1^{ère} transaction à avoir appliqué un verrou sur la table Spectacle sur laquelle elle veut effectuer la mise à jour).
- ➔ **Donc elle sera considérée comme étant la transaction T1**
- ▶ **Algorithme W-D** : La transaction de la session 1 sera mise en attente (de t1 jusqu'à t6) jusqu'à ce que la transaction de la session 2, qui sera débloquée, terminera son exécution (commit ou rollback).
- ▶ **Algorithme W-W** : La transaction de la session 2 sera annulée (rollback de t5 jusqu'à t3) et la transaction de la session 1 sera débloquée et terminera son exécution (commit ou rollback).

Example 2 :

	Session 1	Session 2
t0	Connect	
t1	SELECT * FROM Client WHERE id_client=1; --un verrou S est pose sur Client --lecture faite	
t2		Connect
t3		SELECT * FROM Spectacle WHERE id_spectacle=1; --un verrou S est pose sur Spectacle --lecture faite
t4		UPDATE Client SET nb_places_réservées=5 WHERE id_client=1; --en attente car elle ne peut pas appliquer un verrou X sur la table Client qui a un verrou S posé lors de la session 1
t5	UPDATE Spectacle SET nb_places_libres=45 WHERE id_spectacle=1; --en attente car elle ne peut pas appliquer un verrou X sur la table Spectacle qui a un verrou S posé lors de la session 2	

Exemple 2

- ▶ **Interblocage** : Les deux transactions peuvent s'attendre indéfiniment l'une l'autre.
- ▶ La transaction de la session 2 est bloquée par le verrou partagé posé par la transaction de la session 1.
- ▶ La transaction de la session 1 se trouve bloquée à son tour par le verrou partagé posé par la transaction de la session 2.
- ▶ Le SGBD prend la décision pour débloquer la situation, selon l'algorithme appliqué.

Exemple 2

- ▶ La transaction de **la session 1** est la plus ancienne et **elle ne détient pas la ressource** (la table Spectacle est verrouillée par la transaction de la session2), plutôt elle la demande.
- ➔ **Donc elle sera considérée comme étant la transaction T2**
- ▶ **Algorithme W-D** : La transaction de la session 2 sera annulée (rollback de t4 jusqu'à t3) et la transaction de la session 1 sera débloquée et terminera son exécution (commit ou rollback).
- ▶ **Algorithme W-W** : La transaction de la session 2 sera mise en attente (de t3 jusqu'à t4) jusqu'à ce que la transaction de session 1, qui sera débloquée, terminera son exécution (commit ou rollback).