Chapitre 1

Concurrence d'accès et reprise après panne

Pré requis

Bases de données (semestre 1)

Objectif général

- Appréhender la gestion de la concurrence dans les SGBD.
- Appréhender la gestion des pannes dans les SGBD.

Objectifs spécifiques

A la fin de ce chapitre, vous serez capable de :

- Comprendre les principes et l'intérêt des transactions.
- Enumérer les problèmes posés par les accès concurrents des transactions aux données.
- Comprendre la réponse apportée par la gestion des transactions.
- Comprendre le mécanisme de reprise après panne.

Eléments de contenu

1. Introduction

Quand on développe un programme *P* accédant à une base de données, on effectue en général plus ou moins explicitement deux hypothèses:

- *P* s'exécutera *indépendamment* de tout autre programme ou utilisateur;
- l'exécution de *P* se déroulera toujours intégralement.

Il est clair que ces deux hypothèses ne se vérifient pas toujours. D'une part les bases de données constituent des ressources accessibles *simultanément* à plusieurs utilisateurs qui peuvent y rechercher, créer, modifier ou détruire des informations: les accès simultanés à une même ressource sont dits *concurrents*, et l'absence de contrôle de cette concurrence peut entraîner de graves problèmes de cohérence dus aux interactions des opérations effectuées par les différents utilisateurs. D'autre part on peut envisager beaucoup de raisons pour qu'un programme ne s'exécute pas jusqu'à son terme. Citons par exemple:

- l'arrêt du serveur de données;
- une erreur de programmation entrainant l'arrêt de l'application;
- la violation d'une contrainte amenant le système à rejeter les opérations demandées;
- une annulation décidée par l'utilisateur.

Une interruption de l'exécution peut laisser la base dans un état transitoire *incohérent*, ce qui nécessite une opération de réparation consistant à ramener la base au dernier état cohérent connu avant l'interruption.
Les SGBD relationnels assurent, par des mécanismes complexes, un partage concurrent des données et une gestion des interruptions qui permettent d'assurer à l'utilisateur que les deux hypothèses adoptées intuitivement sont satisfaites, à savoir:

- son programme se comporte, au moment où il s'exécute, *comme s'il* était seul à accéder à la base de données;
- en cas d'interruption intempestive, les mises à jour effectuées depuis le dernier état cohérent seront annulées par le système.

On désigne respectivement par les termes de *contrôle de concurrence* et de *reprise sur panne* l'ensemble des techniques assurant ce comportement. En théorie le programmeur peut s'appuyer sur ces techniques, intégrées au système, et n'a donc pas à se soucier des interactions avec les autres utilisateurs.
L'objectif est de prendre conscience des principales techniques nécessaires à la préservation de la cohérence dans un système multi-utilisateurs, et d'évaluer leur impact en pratique sur la réalisation d'applications bases de données.

2. Transaction

Une transaction est une unité atomique de traitement (séquence d'opérations qui doit être exécutée dans son intégralité) qui est :

- **Soit complètement exécutée.**
- **Soit complètement abandonnée.**

Elle est définie aussi comme une unité de traitement séquentiel (séquence d'actions cohérente) :

- exécutée pour le compte d'un usager ;
- appliquée à une base de données cohérente ;
- restitue une base de données cohérente.

Une transaction peut être simplement l'exécution d'une requête SQL, ou bien celle d'un programme en langage hôte dans lequel on a des appels au langage de requête).
Les Contraintes d'Intégrité sont donc toujours des invariants pour les transactions ce qui signifie qu'une transaction respecte les Contraintes d'Intégrité à la fin pourvu que celles-ci

soient respectées au début. Le respect des contraintes est à la charge du programmeur de transaction.

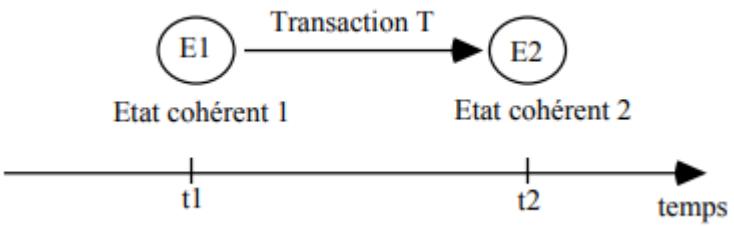


Figure 1 : L'état d'une transaction

Le mécanisme de gestion des transactions doit assurer que :

- Lors d'une exécution d'une transaction toutes ses actions sont exécutées ou bien aucune ne l'est (atomicité). De plus les effets d'une transaction qui s'est exécutée correctement doivent survivre à une panne (permanence),
- Chaque transaction soit isolée, de manière à éviter des incohérences lors d'exécutions concurrentes.

2.1 Les propriétés d'une transaction

Une transaction doit respecter quatre propriétés fondamentales (ACID) :

- **L'atomicité**

Les transactions constituent l'unité logique de travail, toute la transaction est exécutée ou bien rien du tout, mais jamais une partie seulement de la transaction.

- **La cohérence**

Les transactions préservent la cohérence de la BD, c'est à dire qu'elle transforme la BD d'un état cohérent à un autre (sans nécessairement que les états intermédiaires internes de la BD au cours de l'exécution de la transaction respectent cette cohérence)

- **L'isolation**

Les transactions sont isolées les unes des autres, c'est à dire que leur exécution est indépendante des autres transactions en cours. Elles accèdent donc à la BD comme si elles étaient seules à s'exécuter, avec comme corollaire que les résultats intermédiaires d'une transaction ne sont jamais accessibles aux autres transactions.

- **La durabilité**

Les transactions assurent que les modifications qu'elles induisent perdurent, même en cas de défaillance du système.

2.2 Manipulation de transactions en SQL

a) Instructions en SQL

Le langage SQL∗ fournit trois instructions pour gérer les transactions :

- **Début d'une transaction**

Syntaxe :

BEGIN TRANSACTION (ou BEGIN) ;

Cette syntaxe est optionnelle (voire inconnue de certains SGBD), une transaction étant débutée de façon implicite dès qu'instruction est initiée sur la BD.

- **Fin correcte d'une transaction**

Syntaxe :

COMMIT TRANSACTION (ou COMMIT) ;

Cette instruction SQL signale la fin d'une transaction couronnée de succès. Elle indique donc au gestionnaire de transaction que l'unité logique de travail s'est terminée dans un état cohérent est que les données peuvent effectivement être modifiées de façon durable.

- **Fin incorrecte d'une transaction**

Syntaxe :

ROLLBACK TRANSACTION (ou ROLLBACK) ;

Cette instruction SQL signale la fin d'une transaction pour laquelle quelque chose s'est mal passé. Elle indique donc au gestionnaire de transaction que l'unité logique de travail s'est terminée dans un état potentiellement incohérent et donc que les données ne doivent pas être modifiées en annulant les modifications réalisées au cours de la transaction.

N.B :

BEGIN implicite sous Oracle Attention

- La commande BEGIN; ou BEGIN TRANSACTION; ne peut pas être utilisé sous Oracle (la commande BEGIN est réservée à l'ouverture d'un bloc PL/SQL).
- Toute commande SQL LMD (INSERT, UPDATE ou DELETE) démarre par défaut une transaction, la commande **BEGIN TRANSACTION** est donc implicite.

Exemple :

```
BEGIN
   INSERT INTO TAB (a) VALUES (1);
   COMMIT;
END;
```

b) Mode AUTOCOMMIT

La plupart des clients et langages d'accès aux bases de données proposent un mode **AUTOCOMMIT** permettant d'encapsuler chaque instruction dans une transaction. Ce mode revient à avoir un **COMMIT** implicite après chaque instruction.

Ce mode doit être désactivé pour permettre des transactions portant sur plusieurs instructions.

Sous Oracle SQL*Plus :

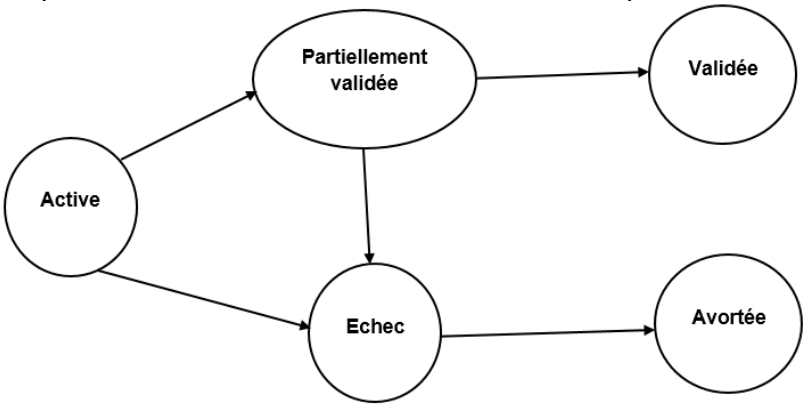
Pour activer commit : SET AUTOCOMMIT ON.

Pour désactiver commit : SET AUTOCOMMIT OFF.

2.3 Etats d'une transaction

- **Active :** la transaction reste dans cet état durant son exécution
- **Partiellement validée :** juste après l'exécution de la dernière opération.
- **Echec :** après avoir découvert qu'une exécution "normale" ne peut pas avoir lieu.
- **Avortée :** Après que toutes les modifications faites par la transaction soient annulées (ROLLBACK).

  Deux options :
  - Réexécuter la transaction
  - Tuer la transaction
- **Validée :** après l'exécution avec succès de la dernière opération.



2.4 Journal des transactions

Le journal est un fichier système qui constitue un espace de stockage redondant avec la BD. Il répertorie l'ensemble des mises à jour faites sur la BD (en particulier les valeurs des enregistrements avant et après mise à jour). Le journal est donc un historique **persistant** (donc en mémoire secondaire) qui mémorise tout ce qui se passe sur la BD.

Le journal est indispensable pour la validation (COMMIT), l'annulation (ROLLBACK) et la reprise après panne de transactions.

Synonymes : Log

3. Concurrence et transactions

3.1 Problèmes soulevés par la concurrence

Nous proposons ci-dessous trois problèmes posés par les accès concurrents des transactions aux données.

- **Perte de mise à jour**

Temps	TR1	TR2
t1	**Lire A**	
t2	**….**	**Lire A**
t3	**UPDATE A**	**….**
t4	**….**	**UPDATE A**
t5	**COMMIT**	**….**
t6		**COMMIT**

Problème de la perte de mise à jour du tuple X par la transaction TR1

Les transactions **TR1** et **TR2** accèdent au même tuple **A** ayant la même valeur respectivement à **t1** et **t2**. Ils modifient chacun la valeur de **A**. Les modifications effectuées par **TR1** seront perdues puisqu'elle avait lu **A** avant sa modification par **TR2**.

Exemple :

Temps	TR1	TR2
t1	**Lire COMPTE C=1000**	
t2	**….**	**Lire COMPTE C=1000**
t3	**UPDATE COMPTE C=C+100=1100**	**….**
t4	**….**	**UPDATE COMPTE C=C+10=1010**
t5	**COMMIT C=1100**	**….**
t6		**COMMIT C=1010**

Dans cet exemple le compte bancaire vaut **1010** à la fin des deux transactions à la place de **1110**.

- **Accès à des données non validées**

Temps	TR1	TR2
t1		**UPDATE A**
t2	**LIRE A**	
t3		**ROLLBACK**

Problème de la lecture impropre du tuple A par la transaction TR1

- La transaction **TR1** accède au **tuple A** qui a été modifié par la transaction **TR2**.
- **TR2** annule sa modification et **TR1** a donc accédé à une valeur qui n'aurait jamais dû exister (virtuelle) de **A**.

- Pire **TR1** pourrait faire une mise à jour de **A** après l'annulation par **TR2**, cette mise à jour inclurait la valeur avant annulation par **TR2** (et donc reviendrait à annuler l'annulation de **TR2**).

Exemple :

Temps	TR1 : Ajouter 10	TR2 : Ajouter 100 (erreur)
t1		**Lire COMPTE C=1000**
t2		**UPDATE COMPTE C=C+100=1100**
t3	**Lire COMPTE C=1100**	**….**
t4	**….**	**ROLLBACK C=1000**
t5	**UPDATE C C=C+10=1110**	**….**
t6	**COMMIT C=1110**	

Annulation de crédit sur le compte bancaire C

Dans cet exemple le compte bancaire vaut **1110** à la fin des deux transactions à la place de **1010**.

- **Lecture incohérente**

Temps	TR1	TR2
t1	**LIRE A**	
t2		**UPDATE A**
t3		**COMMIT**
t4	**LIRE A**	

Problème de la lecture non reproductible du tuple T par la transaction A

Si au cours d'une même transaction, **TR1** accède deux fois à la valeur d'un tuple alors que ce dernier est, entre les deux, modifié par une autre transaction **TR2**, alors la lecture de **TR1** est inconsistante. Ceci peut entraîner des incohérences par exemple si un calcul est en train d'être fait sur des valeurs par ailleurs en train d'être mises à jour par d'autres transactions.

Exemple :

Temps	TR1 : Calcul de S=C1+C2	TR2 : Transfert de 10 de C1 à C2
t1	**Lire COMPTE1 C1=100**	
t2	**…**	**Lire COMPTE1 C1=100**
t3	**…**	**Lire COMPTE2 C2=100**
t4	**…**	**UPDATE COMPTE1 C1=100-10=90**
t5	**…**	**UPDATE COMPTE2 C2=100+10=110**
t6	**…**	**COMMIT**
t7	**LIRE COMPTE2 C2=110**	
t8	**CALCUL S S=C1+C2=210**	

Transfert du compte C1 au compte C2 pendant une opération de calcul C1+C2

Dans cet exemple la somme calculée vaut **210** à la fin du calcul alors qu'elle devrait valoir **200**.

3.2 Le verrouillage

Dans des systèmes multi-utilisateurs, plusieurs utilisateurs peuvent mettre à jour la même information en même temps. Le verrouillage permet à un seul utilisateur de mettre à jour un bloc de données en particulier, empêchant une autre personne de modifier la même donnée.
L'idée de base du verrouillage consiste à bloquer une donnée modifiée dans une transaction jusqu'à ce que la transaction soit validée ou annulée. Le verrou est actif jusqu'à la fin de la transaction, principe mieux connu sous le terme de concurrence d'accès aux données.
Le deuxième objectif du verrouillage est d'assurer que tous les processus peuvent toujours accéder en lecture aux données originales, c'est-à-dire telles qu'elles étaient avant la modification par la transaction non encore validée.

a) Niveaux de verrouillage sous Oracle (row-level et table level locking)

Oracle fournit deux différents niveaux de verrouillage : le verrouillage ligne **(row-level lock)** et le verrouillage table **(table-level lock)**.

- **Verrouillage ligne (row level locking)**

Avec la stratégie de verrouillage ligne, chaque ligne dans une table est verrouillée individuellement. Les lignes verrouillées ne peuvent être mises à jour que par le processus bloquant. Toutes les autres lignes de la table sont toujours disponibles pour mise à jour par d'autres process.

- **Verrouillage table (table level locking)**

Avec le verrouillage table, la table entière est verrouillée intégralement. Une fois qu'un processus a verrouillé la table, seule ce processus peut mettre à jour ou verrouiller une ligne de cette table. Aucune des lignes n'est disponible pour mise à jour par d'autres sessions. Les autres sessions peuvent toujours cependant continuer à lire la table, y compris la ligne qui est en cours de mise à jour.

b) Modes de verrouillage

Oracle utilise 2 modes de verrouillage :

- Le mode de **verrou exclusif** (**X - Exclusive lock mode**) : ce mode empêche la ressource associée d'être partagée. Ce mode de verrou est obtenu pour modifier des données. La première transaction qui verrouille une ressource exclusivement est la seule à pouvoir altérer cette ressource (ligne ou table) jusqu'à ce que le verrou exclusif soit relaxé.

Syntaxes :

Commandes SQL	Mode de verrou
SELECT ... **NOM_TABLE** ...	**Aucun verrou**
INSERT INTO **NOM_TABLE** …	**RX**
UPDATE **NOM_TABLE** ...	**RX**
DELETE FROM **NOM_TABLE**...	**RX**
LOCK TABLE **NOM_TABLE** IN ROW EXCLUSIVE MODE ;	**RX**
LOCK TABLE **NOM_TABLE** IN SHARE ROW EXCLUSIVE MODE;	**SRX**
LOCK TABLE **NOM_TABLE** IN EXCLUSIVE MODE ;	**X**

Le mode de **verrou partagé** (**S - Share lock mode**) : autorise la ressource associée à être partagée, mais tout dépend des opérations engagées. Plusieurs utilisateurs lisant les données peuvent partager les données, laissant des verrous de partage pour empêcher l'accès concurrent à un processus voulant écrire (qui requiert un verrou exclusif). Plusieurs transactions peuvent acquérir des verrous de partage sur la même ressource.

Syntaxes :

Commandes SQL	Mode de verrou
SELECT ... FROM **NOM_TABLE** ... FOR UPDATE OF ...	**RS**
LOCK TABLE **NOM_TABLE** IN ROW SHARE MODE;	**RS**
LOCK TABLE **NOM_TABLE** IN SHARE MODE;	**S**

Explications :

- **Verrou :** Poser un verrou sur un objet (typiquement un tuple) par une transaction signifie rendre cet objet inaccessible aux autres transactions.

Synonymes : Lock

- **Verrou partagé S :** Un verrou partagé, noté S, est posé par une transaction lors d'un accès en **lecture** sur cet objet.
  Un verrou partagé interdit aux autres transactions de poser un verrou exclusif sur cet objet et donc d'y accéder en écriture.

Synonymes : Verrou de lecture, Shared lock, Read lock

- **Verrou exclusif X :** Un verrou exclusif, noté X, est posé par une transaction lors d'un accès en écriture sur cet objet.
  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).

Synonymes : Verrou d'écriture, Exclusive lock, Write lock.

- **Verrous S multiples :** Un même objet peut être verrouillé de façon partagée par plusieurs transactions en même temps. Il sera impossible de poser un verrou exclusif sur cet objet tant **qu'au moins une** transaction disposera d'un verrou S sur cet objet.

N.B :

Opérations interdites

Un verrou X posé par une transaction empêche toute autre transaction de réaliser une quelconque opération **LMD** ou placer tout autre type de verrou sur la table (**Verrou S**).

Exemple :

Session 1	Session 2
LOCK TABLE **ETUDIANT** IN EXCLUSIVE MODE; Table(s) verrouillée(s).	SELECT * FROM **ETUDIANT;** Cette transaction et toutes les autres doivent attendre que la session 1 valide sa transaction. Cette transaction et toutes les autres ne peuvent obtenir aucun verrou sur la table **Etudiant**.
Update **Etudiant** SET ADRESSE_ETU = 'GABES' WHERE CODE_ETU = 'E0002'; 1 ligne mise à jour.	
	LOCK TABLE **ETUDIANT** IN SHARE MODE; En attente ....
	LOCK TABLE **ETUDIANT** IN EXCLUSIVE MODE; En attente ....
	UPDATE **ETUDIANT** SET ADRESSE_ETU = 'NABEUL' WHERE CODE_ETU = 'E0023'; En attente ....

c) Règles de verrouillage

Soit la transaction **TR1** voulant poser un **verrou S** sur un objet O :

1. Si O n'est pas verrouillé alors **TR1** peut poser un verrou S.
2. Si O dispose déjà d'un ou plusieurs verrous S alors **TR1** peut poser un verrou S.
3. Si O dispose déjà d'un verrou X alors **TR1** ne peut pas poser de verrou S.

Soit la transaction **TR1** voulant poser un **verrou X** sur un objet O

1. Si O n'est pas verrouillé alors **TR1** peut poser un verrou X.
2. Si O dispose déjà d'un ou plusieurs verrous S ou d'un verrou X alors **TR1** ne peut pas poser de verrou X.

	Verrou X présent	***Verrou(s) S présent(s)***	***Pas de verrou présent***
Verrou X demandé	***Demande refusée***	***Demande refusée***	***Demande accordée***
Verrou S demandé	***Demande refusée***	***Demande accordée***	***Demande accordée***

Matrice des règles de verrouillage

Remarques :

- Une transaction qui dispose déjà, **elle-même**, d'un verrou **S** sur un objet peut obtenir un verrou X sur cet objet si aucune autre transaction ne détient de verrou S sur l'objet. Le verrou est alors promu du statut partagé au statut exclusif.

- Déverrouillage : lorsqu'une transaction se termine (COMMIT ou ROLLBACK) elle libère tous les verrous qu'elle a posé.
Synonymes : Unlock

3.3 L'inter-blocage

a) Définition

L'**inter-blocage** est le phénomène qui apparaît quand deux transactions (ou plus, mais généralement deux) se bloquent mutuellement par des verrous posés sur les données. Ces verrous empêchent chacune des transactions de se terminer et donc de libérer les verrous qui bloquent l'autre transaction. Un processus d'attente sans fin s'enclenche alors.
Les situations d'inter-blocage sont détectées par les SGBD et gérées, en annulant l'une, l'autre ou les deux transactions, par un ROLLBACK système. Les méthodes utilisées sont la détection de cycle dans un graphe d'attente et la détection de délai d'attente trop long.

b) Cycle dans un graphe d'attente

Principe de détection d'un inter-blocage par détection d'un cycle dans un graphe représentant quelles transactions sont en attente de quelles transactions (par inférence sur les verrous posés et les verrous causes d'attente). Un cycle est l'expression du fait qu'une transaction **TR1** est en attente d'une transaction **TR2** qui est en attente d'une transaction **TR1**.
La détection d'un tel cycle permet de choisir une **victime**, c'est à dire l'une des deux transactions qui sera annulée pour que l'autre puisse se terminer.

c) Délai d'attente

Principe de décision qu'une transaction doit être abandonnée (ROLLBACK) lorsque son délai d'attente est trop long.
Ce principe permet d'éviter les situations d'inter-blocage, en annulant une des deux transactions en cause, et en permettant donc à l'autre de se terminer.

Remarques

- **Risque lié à l'annulation sur délai :** Si le délai est trop court, il y a un risque d'annuler des transactions en situation d'attente longue, mais non bloquées.
  Si le délai est trop long, il y a un risque de chute des performances en situation d'inter-blocage (le temps que le système réagisse).
- **Relance automatique :** Une transaction ayant été annulée suite à un inter-blocage (détection de cycle ou de délai d'attente) n'a pas commis de "faute" justifiant son annulation. Cette dernière est juste due aux contraintes de la gestion de la concurrence. Aussi elle n'aurait pas dû être annulée et devra donc être exécutée à nouveau.
  Certains SGBD se charge de relancer automatiquement les transactions ainsi annulées.

3.4 Solution aux trois problèmes soulevés par la concurrence.

Le principe du verrouillage permet d'apporter une solution aux trois problèmes classiques soulevés par les accès aux concurrents aux données par les transactions.

- **Perte de mise à jour**

Temps	TR1	TR2
t1	**Lire A** **Verrou S**	
t2	**….**	**Lire A** **Verrou S**
t3	**UPDATE A** **En attente…**	**….**
t4	**….** **En attente…**	**UPDATE A** **En attente…**
…	**Inter-blocage**	
t6		**COMMIT**

Problème de la perte de mise à jour du tuple A par la transaction TR1

Le problème de perte de mise à jour est réglé, mais soulève ici un autre problème, celui de **l'inter-blocage**.

- **Accès à des données non validées**

Temps	TR1	TR2
t1		**UPDATE A** **Verrou S**
t2	**LIRE A** **En attente…**	
t3		**ROLLBACK** **Libération de Verrou X…**
t4	**Verrou S**	

Problème de la lecture impropre du tuple A par la transaction TR1

- **Lecture incohérente**

Temps	TR1	TR2
t1	**LIRE A** **Verrou S**	
t2		**UPDATE A** **En attente…**
t3		

t4	**LIRE A** **Verrou S**	
	…libération des verrous	**…reprise de la transaction**

Problème de la lecture non reproductible du tuple A par la transaction A

La lecture reste cohérente car aucune mise à jour ne peut intervenir pendant le processus de lecture d'une même transaction.

3.5 Protocole d'accès aux données

Règles de verrouillage avant les lectures et écritures des données :

- Soit la transaction **TR1** voulant lire des données d'un tuple **A** :
  1. A demande à poser un verrou S sur **TR1**
  2. Si **TR1** obtient de poser le verrou alors **TR1** lit **A**
     Sinon **TR1** attend le droit de poser son verrou (et donc que les verrous qui l'en empêchent soient levés)
- Soit la transaction **TR1** voulant écrire des données d'un tuple **A** :
  1. **TR1** demande à poser un verrou X sur **A.**
  2. Si **TR1** obtient de poser le verrou alors **TR1** écrit **A.**
     Sinon **TR1** attend le droit de poser son verrou (et donc que les verrous qui l'en empêchent soient levés).

Soit la transaction **TR1** se terminant (COMMIT ou ROLLBACK) :

1. **TR1** libère tous les verrous qu'elle avait posé.
2. Certaines transactions en attente obtiennent éventuellement le droit de poser des verrous.

Remarques :

Liste d'attente : Afin de rationaliser les attentes des transactions, des stratégies du type FIFO sont généralement appliquées et donc les transactions sont empilées selon leur ordre de demande.

3.6 Contrôle par verrouillage à deux phases (2PL)

Le 2PL est basé sur le *verrouillage* des nuplets lus ou mis à jour. L'idée est simple : chaque transaction désirant lire ou écrire un nuplet doit auparavant obtenir un verrou sur ce nuplet. Une fois obtenu (sous certaines conditions explicitées ci-dessous), le verrou reste détenu par la transaction qui l'a posé, jusqu'à ce que cette transaction décide de relâcher le verrou.
Le 2PL gère deux types de verrous :

- les ***verrous partagés*** autorisent la pose d'autres verrous partagés sur le même nuplet.
- les ***verrous exclusifs*** interdisent la pose de tout autre verrou, exclusif ou partagé, et donc de toute lecture ou écriture par une autre transaction.

On ne peut poser un **verrou partagé S** que s'il n'y a que des verrous partagés sur le nuplet. On ne peut poser un **verrou exclusif X** que s'il n'y a aucun autre verrou, qu'il soit exclusif ou partagé. Les verrous sont posés par chaque transaction, et ne sont libérés qu'au moment du commit ou **ROLLBACK**.

Exemple :

Soient les deux transactions suivantes :

- **TR1:** r1[x]w1[y]C1
- **TR2:** w2[x]w2[y]C2

et l'exécution concurrente : r1[x]w2[x]w2[y]C2w1[y]C1

N.B : r désigne read (Lecture) et w désigne write (Ecriture)

Solution basée sur le verrouillage à deux phases :

- T1 pose un verrou partagé sur *x*, lit *x* **mais ne relâche pas le verrou ;**
- T2 tente de poser un verrou exclusif sur *x* : **impossible puisque T1 détient un verrou partagé, donc T2 *est mise en attente* ;**
- T1 pose un verrou exclusif sur *y*, modifie *y*, et valide ; **tous les verrous détenus par T1 sont relâchés ;**
- T2 est libérée : elle pose un verrou exclusif sur *x*, et le modifie ;
- T2 pose un verrou exclusif sur *y*, et modifie *y* ;
- T2 valide, ce qui relâche les verrous sur *x* et *y*.

On obtient donc, après réordonnancement, l'exécution suivante :

r1[x]w1[y]w2[x]w2[y]

Exemple :

Voici comment éviter le problème de la mise à jour perdue à l'aide de 2PL :

Temps	TR1	TR2	$Solde_{compte}$
t1		DEBUT_TRANSACTION	100
t2	DEBUT_TRANSACTION	Verrou X ($Solde_{compte}$)	100
t3	Verrou X ($Solde_{compte}$)	Lire ($Solde_{compte}$)	100
t4	En attente…	$Solde_{compte=}$ $Solde_{compte}$+100	100
t5	En attente…	Ecrire ($Solde_{compte}$)	200
t6	En attente…	Validation/Déverrouillage ($Solde_{compte}$)	200
t7	Lire ($Solde_{compte}$)		200
t8	$Solde_{compte=}$ $Solde_{compte}$-10		200
t9	Ecrire ($Solde_{compte}$)		190
t10	Validation/Déverrouillage ($Solde_{compte}$)		190

Remarque :

Un des inconvénients du verrouillage à deux phases est d'autoriser des *interblocages* : deux transactions concurrentes demandent chacune un verrou sur une ressource détenue par l'autre.

4. Reprise après panne

La reprise après panne consiste, comme son nom l'indique, à assurer que le système est capable, après une panne, de récupérer *l'état de la base* au moment où la panne est survenue. Le terme de panne désigne ici tout événement qui affecte le fonctionnement du processeur ou de la mémoire principale. Il peut s'agir par exemple d'une coupure électrique interrompant le serveur de données, d'une défaillance logicielle, ou des pannes affectant les disques Par souci de simplicité on va distinguer deux types de panne (quelle que soit la cause).

- *Panne légère*: affecte la RAM du serveur de données, pas les disques
- *Panne lourde*: affecte un disque.

La problématique de la reprise sur panne est à rapprocher de la garantie de durabilité pour les transactions. Il s'agit d'assurer que même en cas d'interruption à *t+1*, on retrouvera la situation issue des transactions validées.

a) L'état de la base

On définit l'état de la base à un instant *t* comme l'état résultant de l'ensemble des transactions validées à l'instant *t*.

Pour assurer la sécurité des données (face à une panne légère au moins), il est impératif que l'état de la base soit stocké sur support persistant, à tout instant. Une première règle simple est donc:

Règle 1 :

L'état de la base doit toujours être stocké sur disque.

b) Garanties transactionnelles

Les propriétés d'une transaction :

- *Durabilité (et atomicité)*: quand le système rend la main après un commit (*acquittement*), *toutes* les modifications de la transaction intègrent l'état de la base et deviennent *permanentes*.
- *Recouvrabilité (et atomicité)*: tant qu'un commit n'a pas eu lieu, *toutes* les modifications de la transaction doivent pouvoir être *annulées* par un rollback.

c) Image avant et image après

L'image après d'une transaction T désigne la nouvelle valeur des nuplets modifiés par T. L'image avant désigne l'ancienne valeur de ces nuplets (avant modification).
L'exécution d'une transaction peut donc se décrire de la manière suivante, illustrée par la **figure 2 ci-dessous**. Au départ, l'état de la base est stocké sur disque. Une partie de cet état correspond à l'image avant des nuplets qui vont être modifiés par la transaction. Au cours de la transaction, l'image après est constituée, et la transaction a, à chaque instant, deux possibilités:

- Un **COMMIT**, et l'image après remplace l'image avant dans l'état de la base;
- Un **ROLLBACK**, et l'état de la base est restauré comme initialement, avec l'image avant.

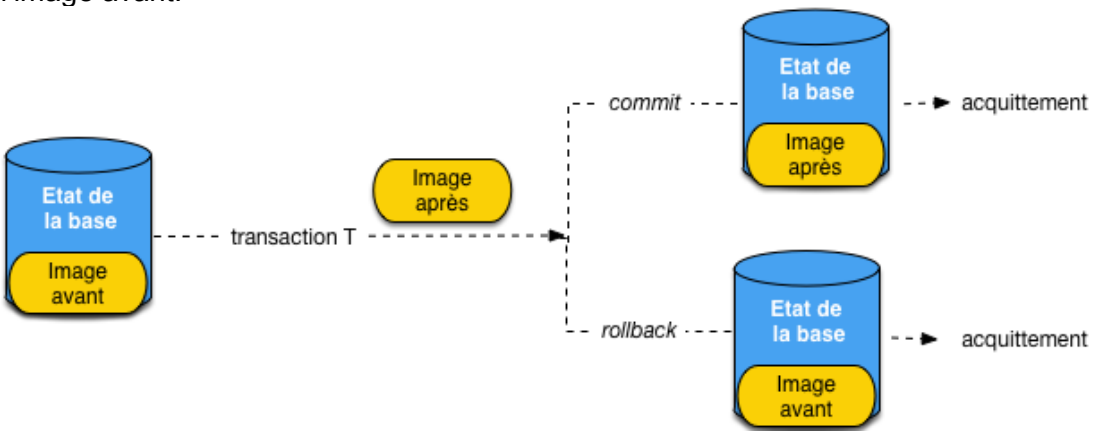


Figure 2 : La reprise sur panne en termes d'état de la base, image avant et image après

Le moment où la transaction effectue un **COMMIT** ou un ROLLBACK est important. Chacune de ces instructions est « acquitté » quand le serveur de données rend la main au processus client. Pour garantir le **COMMIT**, la condition suivante doit être respectée.

Règle 2 :

L'image après doit être sur le disque avant l'acquittement du **COMMIT**.
Si ce n'était pas le cas et qu'une panne survienne juste après l'acquittement, mais avant l'écriture de l'image après sur le disque, cette dernière serait en partie ou totalement perdue, et la garantie de durabilité ne serait pas assurée.

Maintenant, pour garantir le **ROLLBACK**, il faut pouvoir trouver sur le disque, après une panne, les données modifiées par la transaction dans l'état où elles étaient au moment où la transaction a débuté. La condition suivante doit être respectée.

Règle 3 :

L'image avant doit être sur le disque jusqu`à l'acquittement du **COMMIT**.
On peut résumer la difficulté ainsi: jusqu'au **COMMIT**, c'est *l'image avant* qui fait partie de l'état de la base. Au **COMMIT**, *l'image après* remplace *l'image avant* dans l'état de la base. Il faut assurer que ce remplacement s'effectue de manière atomique (« tout ou rien »).