

# SGBD-R : Les transactions

## Concurrence et anomalies

---

Guillaume Raschia — Nantes Université

Dernière mise-à-jour : 28 février 2023

originaux de Philippe Rigaux, CNAM

Notion de transaction (S9.1)

Les anomalies (S9.3)

- Mise à jour perdue

- Lecture non répétable et lecture fantôme

- Lecture sale

Les niveaux d'isolation (S9.4)

## Notion de transaction (S9.1)

---

# De quoi s'agit-il ?

Deux hypothèses à reconsidérer.

- Hyp. 1 : Un programme SQL s'exécute indépendamment des autres.  
Faux, car de grandes bases de données peuvent gérer des centaines, voire des milliers d'accès concurrents qui peuvent interagir les uns avec les autres.
- Hyp. 2 : Un programme SQL s'exécute sans erreur et intégralement.  
Faux pour des raisons innombrables : plantage de l'application, pb réseau, pb du serveur, panne électrique, etc. L'interruption peut laisser la base dans un état incohérent.

La notion de transaction (tx) est centrale pour traiter les problèmes soulevés.

# Objectifs du cours

Un concepteur d'application doit :

- Maîtriser la notion, très importante, de **transaction**
- Réaliser l'impact des **exécutions** transactionnelles **concurrentes** pour les utilisateurs et les applications
- Choisir un **niveau d'isolation** approprié
- Maîtriser les procédures de **reprise après une panne**

Les techniques ? **Contrôle de concurrence** et **journalisation**.

# Qu'est-ce qu'une transaction ?

## Définition

Une transaction est une séquence d'opérations de **lecture** ou d'**écriture**, se terminant par **commit** (approbation) ou **rollback** (rejet).

Le **commit** est une instruction qui **valide** toutes les mises à jour.

Le **rollback** est une instruction qui **annule** toutes les mises à jour.

## Essentiel

Les opérations d'une tx sont solidaires : elles sont toutes validées, ou toutes annulées (**atomicité**).

## Pour bien comprendre

On parle de **transaction** plutôt que de **programme** : beaucoup plus précis.

Une transaction est le produit d'un échange entre un **processus client** et un **processus serveur** (SGBD).

On peut effectuer une ou plusieurs transactions successives dans un même processus : elles sont dites **sérielles**.

En revanche, deux processus **distincts** engendrent des **transactions concurrentes**.

## Exemple : les données

- Des clients qui réservent des places pour des spectacles

`Client (id_client, nom, nb_places_réservées)`

- Des spectacles qui proposent des places à des clients<sup>1</sup>.

`Spectacle (id_spectacle, jauge, solde, tarif)`

Cette base est **cohérente** si **le nombre de places prises** ( $\text{jauge} - \text{solde}$ ) est égal à **la somme des places réservées**.

---

1. La modélisation n'est pas complète : il manque en particulier l'association client-spectacle.



## Exemple : le programme (lectures)

```
procedure Reservation (v_id_client INT, v_id_spectacle INT, nb_places INT)
-- Variables
v_client Client%ROWTYPE;
v_spectacle Spectacle%ROWTYPE;
v_places_libres INT;
v_places_reservees INT;
BEGIN
-- On recherche le spectacle
SELECT * INTO v_spectacle
FROM Spectacle WHERE id_spectacle=v_id_spectacle;

-- S'il reste assez de places: on effectue la reservation
IF (v_spectacle.solde >= nb_places)
THEN
-- On recherche le client
SELECT * INTO v_client FROM Client WHERE id_client=v_id_client;

-- Calcul du transfert
v_places_libres := v_spectacle.solde - nb_places;
v_places_reservees := v_client.nb_places_reservees + nb_places;
```

## Le programme, suite (écritures)

```
-- On diminue le nombre de places libres
UPDATE Spectacle SET solde = v_places_libres
    WHERE id_spectacle=v_id_spectacle;

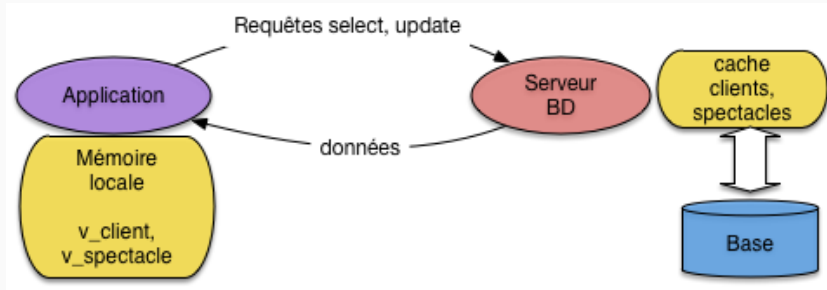
-- On augmente le nombre de places reervees par le client
UPDATE Client SET nb_places_reservees=v_places_reservees
    WHERE id_client = v_id_client;

-- Validation
commit;
ELSE
    rollback;
END IF;
END;
```

NB : le langage (ici PL/SQL) n'a aucun impact sur la modélisation des transactions.

## Ce programme engendre des **transactions**

**Exécution** = séquence de lectures et mises à jour = **transaction**.



Le SGBD ne sait pas ce que fait l'application avec les données transmises. **Il ne voit que la séquence des lectures et des écritures.**

# Représentation d'une transaction

On représente les transactions par ce qu'en connaît le SGBD.

- les transactions, notées  $T_1, T_2, \dots, T_n$ ;
- les « données » (nuplets) échangées sont notées  $x, y, z, \dots$ ;
- pour chaque tx  $T_i$ , une lecture de  $x$  est notée  $r_i(x)$  et une écriture de  $x$  est notée  $w_i(x)$ ;
- $C_i$  et  $R_i$  représentent un **commit** (resp. **rollback**) effectué par la tx  $T_i$ .

Une tx  $T_i$  est donc une séquence de lectures ou d'écritures se terminant par  $C_i$  ou  $R_i$ .

Exemple :  $r_i(x); w_i(y); r_i(y); r_i(z); w_i(z); C_i$

# Les transactions engendrées par Réservation

En s'exécutant, la procédure **Réservation** engendre des transactions.

Exemples :

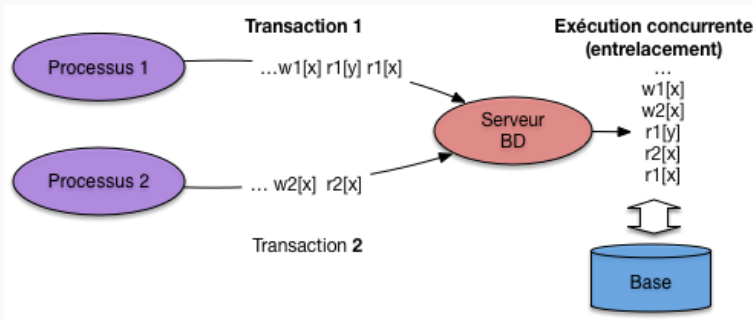
- $r(s_1); r(c_1); w(s_1); w(c_1); C$  : lit le spectacle  $s_1$ , le client  $c_1$ , puis les met à jour;
- $r(s_1); r(c_2); w(s_1); w(c_2); C$  : le client  $c_2$  réserve pour le même spectacle  $s_1$ ;
- $r(s_1); R$  : lit le spectacle  $s_1$ , et rejette la tx (plus assez de places disponibles?)

Un même processus peut effectuer des transactions **en série** :

$r_1(s_1); r_1(c_1); w_1(s_1); w_1(c_1); C_1; r_2(s_1); r_2(c_2); w_2(s_1); w_2(c_2); C_2 \dots$

# Exécutions concurrentes

Quand plusieurs programmes clients sont actifs simultanément, les transactions engendrées s'effectuent **en concurrence**.



L'**entrelacement de requêtes** issues de **transactions concurrentes** peut engendrer des anomalies.

# Propriétés des transactions

Un système relationnel contrôle l'exécution concurrente des transactions et garantit un ensemble de propriétés rassemblées sous l'acronyme **ACID**.

- **A = Atomicité**. Une tx est approuvée complètement ou pas du tout.
- **C = Cohérence**. Une tx mène d'un état cohérent à un autre.
- **I = Isolation**. Une tx s'exécute comme si elle était seule.
- **D = Durabilité**. À l'heure du **commit**, les résultats d'une tx sont définitifs.

## Essentiel

L'isolation **par défaut** est seulement partielle : **meilleures performances** mais **risque d'anomalies**.

## À retenir

Transaction = séquence de lecture et mises à jour soumises par une application client, se terminant par **commit** ou **rollback**.

Important : la séquence des actions d'une tx est **immuable** !

Le SGBD **garantit** que l'exécution des transactions respecte des propriétés dites **ACID**.

Le degré de garantie dépend du **niveau d'isolation** qui est choisi par l'application (son développeur).

**Un niveau d'isolation insuffisant peut entraîner des anomalies très difficiles à comprendre.**



## Les anomalies (S9.3)

---

## Objectifs de la section

Un catalogue des principales anomalies dues à un défaut dans le contrôle de concurrence

- Les **mises à jour perdues** : la plus importante, car autorisée par les systèmes dans la configuration par défaut.
- Les **lectures non répétables** : des données apparaissent, disparaissent, changent de valeur, au cours de l'exécution d'une transaction.
- Les **lectures sales** : on peut lire une donnée modifiée mais non validée.

Nous verrons que les **niveaux d'isolation** servent à éviter tout ou partie de ces anomalies.

## Mises à jour perdues

C'est l'anomalie permise par **tous** les systèmes, en mode transactionnel par défaut.

Retenir : deux transactions **lisent** une même donnée, et **l'écrivent** ensuite, l'une après l'autre : une des écritures est perdue.

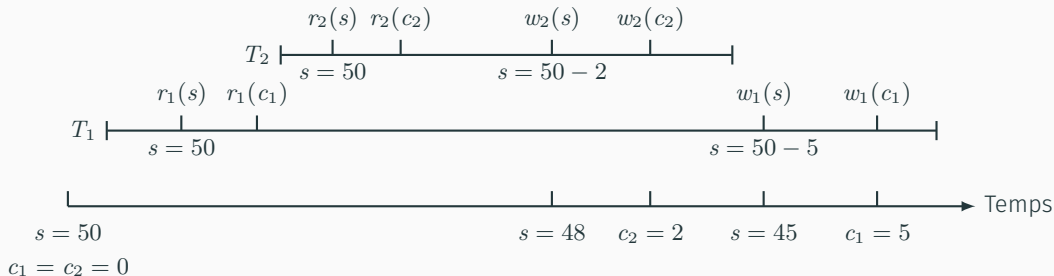
Exemple d'exécution concurrente avec deux transactions de réservation

$$r_1(s); r_1(c_1); r_2(s); r_2(c_2); w_2(s); w_2(c_2); w_1(s); w_1(c_1)$$

On effectue d'abord les lectures pour  $T_1$ , puis les lectures pour  $T_2$  enfin les écritures pour  $T_2$  et  $T_1$  dans cet ordre.

# L'anomalie

- il reste 50 places libres,  $c_1$  et  $c_2$  n'ont encore rien réservé;
- $T_1$  veut réserver 5 places pour  $s$ ;
- $T_2$  veut réserver 2 places pour  $s$ .



# Le déroulement

Pas à pas, voici ce qui se passe.

- $T_1$  lit  $s$  et  $c_1$ . Nb places libres : 50.
- $T_2$  lit  $s$  et  $c_2$ . Nb places libres : 50.
- $T_2$  écrit  $s$  avec nb places =  $50 - 2 = 48$ .
- $T_2$  écrit le nouveau compte de  $c_2$ .
- $T_1$  écrit  $s$  avec nb places =  $50 - 5 = 45$ .
- $T_1$  écrit le nouveau compte de  $c_1$ .

À l'arrivée, 5 places réservées, 7 places payées. Incohérence.

## Pour bien comprendre


À la fin de l'exécution, il reste 45 places vides sur les 50 initiales alors que 7 places ont effectivement été réservées et payées.

Cette anomalie ne survient qu'en cas d'entrelacement défavorable. Dans la plupart des cas, une exécution concurrente ne pose pas de problème.

Le programme est correct. On peut le regarder 1000 fois, le tester 10000 fois sans jamais détecter d'anomalie.

### Essentiel

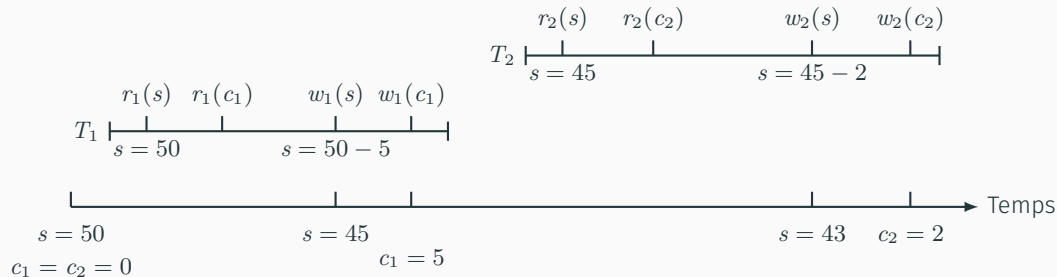
Si on ne sait pas qu'une anomalie de concurrence est possible, l'erreur est incompréhensible.

 Qu'en est-il si l'on permute les écritures de  $T_1$  et  $T_2$  ?

## Solution radicale : exécution en série

Si on force l'exécution en série :  $r_1(s); r_1(c_1); w_1(s); w_1(c_1); r_2(s); r_2(2); w_2(s); w_2(c_2)$

On est sûr qu'il n'y a pas de problème :



Très pénalisant. Je débute une petite transaction, je vais déjeuner, je bloque tout le monde...

✎ Quel est l'état de la base de données si j'inverse  $T_1$  et  $T_2$  dans l'exécution en série ?

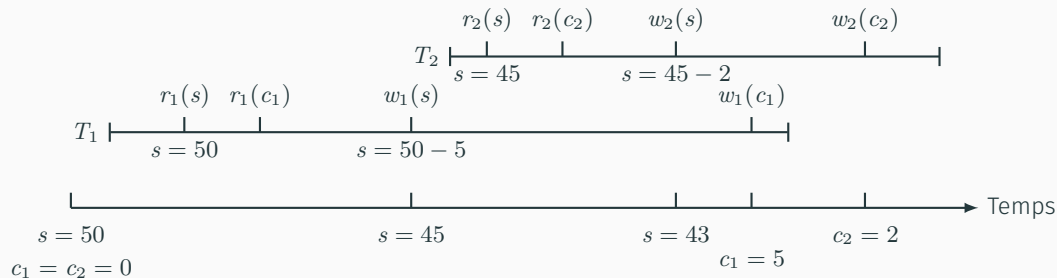
✎ Et si  $s = 6$  dans l'état initial ?



# Les exécutions concurrentes sont possibles

Un exemple qui ne pose pas de problème :

$r_1(s); r_1(c_1); w_1(s); r_2(s); r_2(c_2); w_2(s); w_1(c_1); w_2(c_2)$



Exécution dite **sérialisable** car **équivalente** à une exécution en série. C'est ce que doit assurer le contrôle de concurrence.

## Lectures non répétables et autres fantômes

Deuxième catégorie d'anomalies. Prenons l'exemple du programme **Contrôle**.

```
Procédure Contrôle()
```

```
  Début
```

```
    Lire tous les clients et effectuer la somme des places prises
```

```
    Lire le spectacle
```

```
    SI (Somme(places prises) <> places réservées)
```

```
      Afficher ("Incohérence dans la base")
```

```
  Fin
```

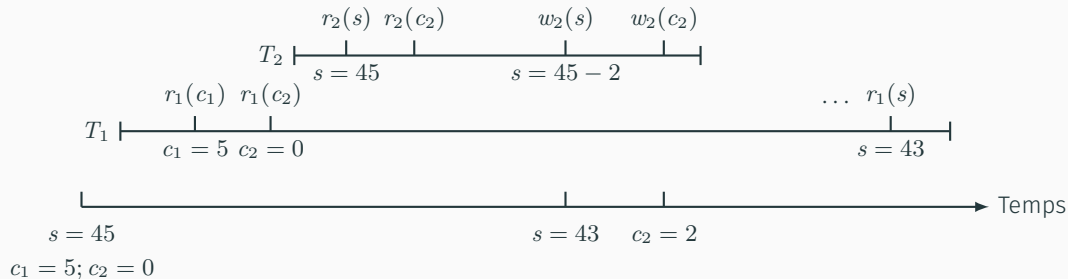
Vérifie la cohérence de la base. Une transaction  $T_c$  a la forme :

$$r_c(c_1); \dots r_c(c_n); r_c(s)$$

# Une exécution concurrente avec réservation

« Entrelacement » de `Contrôle()` et de `Réservation(c2, s, 2)`

$r_1(c_1); r_1(c_2); \text{Réservation}(c_2, s, 2); r_1(c_3); \dots r_1(c_n); r_1(s)$



Le contrôle en déduit (à tort) que la base est incohérente.

## Que s'est-t-il passé ?

Une même transaction (le contrôle) a pu lire deux états **différents** de la base.

On a donc des problèmes liés au manque d'isolation. Deux types :

- La même donnée, lue deux fois de suite, a changé : **lecture non répétable**.
- Des données sont apparues (ou ont disparu) : **lecture fantôme**.

Dans les deux cas, il y a une isolation partielle (on voit les résultats d'une ou plusieurs autres transactions) et donc un risque d'anomalie.

## Une troisième catégorie d'anomalies : les lectures sales

C'est un autre type de problème (dit « de réparabilité »<sup>2</sup>) : l'entrelacement empêche une bonne exécution des `commit` et `rollback`.

Exemple :

$r_1(s); r_1(c_1); w_1(s); r_2(s); r_2(c_2); w_2(s); w_2(c_2); C_2; w_1(c_1); R_1$

Notez :  $T_2$  a lu la donnée  $s$  écrite par  $T_1$ ;  $T_2$  valide,  $T_1$  annule.

Comment gérer cette annulation ?

---

2. *recovery* en anglais, mal traduit par recouvrabilité.

## Problème de dépendances entre Tx

Annuler  $T_1$  suppose que les mises à jour de  $T_1$  n'ont pas eu lieu.

Oui mais,  $T_2$  « a lu »  $T_1$ , et donc une des données qui viennent d'être effacées!

Alors il faudrait annuler  $T_2$  aussi? Mais  $T_2$  a fait un `commit`...

La « réparabilité » suppose qu'une tx ne soit approuvée qu'une fois que les tx qu'elle a lu ont elles-mêmes été approuvées.

La « prévention des annulations en chaîne » impose qu'une tx ne lise que des tx approuvées.

## À retenir

Des anomalies plus ou moins graves apparaissent en cas de défaut d'isolation des tx.

La plus importante est celle des **mises à jour perdues** : elle peut apparaître avec le niveau d'isolation **par défaut** des systèmes.

Les anomalies de concurrence sont **très difficiles** à reproduire et à interpréter.

**Comprendre et utiliser correctement les niveaux d'isolation est impératif pour les applications transactionnelles.**

## Les niveaux d'isolation (S9.4)

---



## Ce qu'on doit savoir

Il existe plusieurs niveaux d'isolation, du plus permissif au plus strict.

Plus le niveau est permissif, plus l'exécution est fluide, mais plus les anomalies sont possibles.

Plus le niveau est strict, plus l'exécution risque de rencontrer des blocages, mais moins les anomalies sont possibles.

Le niveau d'isolation par défaut n'est jamais le plus strict. Car

- inutile dans la grande majorité des cas;
- provoque rejets et blocages difficiles à expliquer à l'utilisateur.

Quand l'isolation totale est nécessaire, il faut l'indiquer explicitement.

# Niveaux d'isolation SQL

Définis en fonction des anomalies : lectures sales, lectures non répétables et lectures fantômes.

1. `read uncommitted` : tout est permis, sauf les écritures sales;
2. `read committed` : garantit en plus l'absence de lecture sale;
3. `repeatable read` : garantit en plus l'absence de lecture non répétable;
4. `serializable` : isolation totale, aucune anomalie.

Niveau par défaut : `read committed` (PostgreSQL, MS SQL Server, Oracle) ou `repeatable read` (MySQL).

Il faut se mettre en mode `serializable` pour les processus transactionnels.

### Énoncé

Une requête accède à l'état de la base **au moment où la requête est exécutée**.

On peut donc lire deux fois le même nuplet dans une tx et obtenir des résultats différents.

Pas de lecture sale, car une donnée en cours de modification ne fait pas partie de **l'état de la base**, l'ensemble des données approuvées (`commit`).

Assez fluide, mais autorise beaucoup d'anomalies.

### Énoncé

Une requête accède à l'état de la base **au moment où la transaction a débuté**.

On peut donc lire  $n$  fois le même nuplet dans une tx et obtenir toujours le même résultat.

Pas de lecture sale, pas de lecture non répétable : les requêtes accèdent toujours au même état (ou « cliché ») de la base.

### Est-ce suffisant ?

Non, car des exécutions **non sérialisables** restent quand même possibles, notamment à cause des **lectures fantômes**.

# Le mode serializable

## Énoncé

Mise en œuvre de l'isolation totale. La cohérence de la base est absolue.

Requiert *a priori* de **poser des verrous** (mutex) sur les nuplets.

## Inconvénient

Risque non négligeable de **blocage** et de **rejet** des transactions.

On se place dans ce mode avec la commande suivante, au début du code de la transaction.

```
| set transaction isolation level serializable;
```

# À propos des mises-à-jour perdues

Les niveaux d'isolation SQL ne tiennent pas compte des mises-à-jour perdues ?!

Libre interprétation du ANSI/ISO SQL dans les SGBD.

L'histoire des tx en SQL fut écrite au temps du « tout 2PL », protocole qui **garantit l'absence de mäj perdues dès le niveau repeatable read**.

## État des lieux

- Dans la plupart des SGBD (PostgreSQL, MS SQL Server, Oracle, IBM DB2), le niveau **repeatable read** prémunit contre les mises-à-jour perdues.
- ...Néanmoins, il existe certaines exceptions, par exemple dans MySQL.

# La clause `for update`

Situation fréquente : contexte d'une mise-à-jour perdue

On **lit** une donnée pour la **modifier** ensuite.

- Le système ne peut pas le deviner!
- Il est obligé de tracer toutes les lectures (coût élevé)
- Il pose des « verrous faibles » qui peuvent entraîner un interblocage

Renforcer l'isolation **à la demande** avec `for update`

- Le programmeur déclare qu'une lecture va être suivie d'une mise à jour
- Le système pose un « verrou fort » pour celle-là, et met les autres en attente

Le mécanisme repose sur le facteur humain, peu fiable...

Savoir repérer les transactions dans une application. Elle doivent respecter la **cohérence applicative** (le système ne peut pas la deviner pour vous).

Pour une cohérence absolue, appliquer le mode d'**isolation sérialisable**.

Savoir qu'en mode sérialisable on risque des **blocages** et des **rejets**.

Comprendre les risques dans les modes dégradés.

La clause **for update** offre une isolation ponctuelle intéressante, mais repose sur le facteur humain : probablement à éviter.