Université Paris-Dauphine

Cours de bases de données

Aspects système

Philippe Rigaux

Philippe.Rigaux@dauphine.fr

http ://www.lamsade.dauphine.fr/rigaux/bd

Table des matières

Chapitre 1

Introduction à la concurrence d'accès

Sommaire

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

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

Or il est clair que ces deux hypothèses sont 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 :

1. l'arrêt du serveur de données ;
2. une erreur de programmation ;
3. la violation d'une contrainte amenant le système à rejeter les opérations demandées ;
4. 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 mises en œuvre par le système, et n'a donc pas à se soucier des interactions avec les autres utilisateurs. En pratique les choses ne sont pas si simples, et le contrôle de concurrence a pour contreparties certaines conséquences qu'il est souvent important de prendre en compte dans l'écriture des programmes. En voici la liste, chacune étant développée dans le reste de ce chapitre :

– **Définition des points de sauvegardes**. La reprise sur panne garantit le retour au dernier état cohérent de la base précédant l'interruption, mais c'est au programmeur de définir ces points de cohérence (ou « points de sauvegarde ») dans le code des programmes.
– **Blocages des autres utilisateurs**. Le contrôle de concurrence s'appuie sur le verrouillage de certaines resources (tables blocs, n-uplets) qui peut bloquer temporairement d'autres utilisateurs. Dans certains cas des *interblocages* peuvent même apparaître, amenant le système à rejeter l'exécution d'un des programmes en cause.
– **Choix d'un niveau d'isolation**. Une isolation totale des programmes garantit la cohérence, mais entraîne une dégradation des performances due au verrouillage appliqué systématiquement par le SGBD. Or, dans beaucoup de cas, le verrouillage est trop strict et place en attente des programmes dont l'exécution ne met pas en danger la cohérence de la base. Le programmeur peut alors choisir d'obtenir plus de concurrence (autrement dit, plus de *fluidité* dans les exécutions concurrentes), en demandant au système un niveau d'isolation moins strict, et en prenant éventuellement lui-même en charge le verrouillage des ressources critiques.

Ce chapitre est consacré à la concurrence d'accès, vue par le programmeur d'application. Il ne traite pas, ou très superficiellement, des algorithmes implantés par les SGBD. 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. La gestion de la concurrence, du point de vue de l'utilisateur, se ramène en fait à la recherche du bon compromis entre deux solutions extrêmes : une cohérence maximale impliquant un niveau d'interblocage relativement élevé, ou une fluidité concurrentielle totale au prix de risques importants pour l'intégrité de la base. Ce compromis dépend de l'application et de son contexte (niveau de risque acceptable *vs* niveau de performance souhaité) et relève donc du choix du programmeur.

Le chapitre débute par une définition de la notion de *transaction*, et montre ensuite, sur différents exemples, les problèmes qui peuvent survenir en l'absence d'une politique de contrôle de la concurrence. Les différentes mécanismes de contrôle sont ensuite présentés : estampillage, multi-versions et verrouillages. Pour finir nous présentons les niveaux d'isolation définis par la norme SQL.

1.1 Transactions

Une *transaction* est une séquence d'opérations de lecture ou mise à jour sur une base de données, se terminant par l'une des deux instructions suivantes :

1. `commit`, indiquant la validation de toutes les opérations effectuées par la transaction ;
2. `rollback` indiquant l'annulation de toutes les opérations effectuées par la transaction.

Une transaction constitue donc, pour le SGBD, une unité d'exécution. Toutes les opérations de la transaction doivent être validées ou annulées solidairement.

1.1.1 Notions de base

On utilise toujours le terme de transaction, au sens défini ci-dessus, de préférence à « programme », « procédure » ou « fonction » qui sont à la fois inappropriés et quelque peu ambigus. « Programme » peut en effet désigner, selon le contexte, la spécification avec un langage de programmation, ou l'exécution sous

la forme d'un processus client communiquant avec le (programme serveur du) SGBD. C'est toujours la seconde acception qui s'applique pour le contrôle de concurrence. De plus, l'exécution d'un programme consiste en une suite d'ordres SQL adressés au SGBD, cette suite pouvant être découpée en une ou plusieurs transactions en fonction des ordres `commit` ou `rollback` qui s'y trouvent. La première transaction débute avec le premier ordre SQL exécuté [1] ; toutes les autres débutent après le `commit` ou le `rollback` de la transaction précédente.

Dans tout ce chapitre nous allons prendre l'exemple de transactions consistant à réserver des places de spectacle pour un client. On suppose que la base contient les deux tables suivantes :

1. Client (**id_client**, nb_places_réservées)
2. Spectacle (**id_spectacle**, nb_places_offertes, nb_places_libres)

Chaque transaction s'effectue pour un client, un spectacle et un nombre de places à réserver. Elle consiste à vérifier qu'il reste suffisamment de places libres. Si c'est le cas elle augmente le nombre de places réservées par le client, et elle diminue le nombre de places libres pour le spectacle. On peut la coder en n'importe quel langage. Voici, pour être concret, la version PL/SQL.

Exemple 1. *Transaction.sql : Une procédure transactionnelle*

```
/*
  Un programme de reservation
  CREATE TABLE Client (id_client INT NOT NULL,
                       nb_places_reservees INT NOT NULL,
                       solde INT NOT NULL,
                        PRIMARY KEY (id_client));
  CREATE TABLE Spectacle (id_spectacle INT NOT NULL,
                          nb_places_offertes INT NOT NULL,
                          nb_places_libres INT NOT NULL,
                          tarif DECIMAL(10,2) NOT NULL,
                         PRIMARY KEY (id_spectacle));
*/

CREATE OR REPLACE PROCEDURE Reservation (v_id_client INT,
                                         v_id_spectacle INT,
                                          nb_places INT) AS
  -- Déclaration des 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.nb_places_libres >= 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.nb_places_libres - nb_places;
    v_places_reservees := v_client.nb_places_reservees + nb_places;

    -- On diminue le nombre de places libres
    UPDATE Spectacle SET nb_places_libres = v_places_libres
```

1. Il est parfois possible d'indiquer explicitement ce début avec la commande `START TRANSACTION`.

```
       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;
/
```

Chaque *exécution* de ce code correspondra à une transaction. La première remarque, essentielle pour l'étude et la compréhension du contrôle de concurrence, est que l'exécution d'une procédure de ce type correspond à des échanges entre deux processus distincts : le processus *client* qui exécute la procédure, et le processus *serveur* du SGBD qui se charge de satisfaire les requêtes SQL. On prend toujours l'hypothèse que les zones mémoires des deux processus sont distinctes et étanches. Autrement dit le processus client ne peut accéder aux données que par l'intermédiaire du serveur, et le processus serveur, de son côté, est totalement ignorant de l'utilisation des données transmises au processus client. Il s'ensuit que non seulement le langage utilisé pour coder les transactions est totalement indifférent, mais que les variables, les interactions avec un utilisateur ou les structures de programmation –tests et boucles– du processus client sont transparentes pour le programme serveur. Ce dernier ne dispose que des instructions qui lui sont explicitement destinées, autrement dit les ordres de lecture ou d'écriture, les `COMMIT` et les `ROLLBACK`.

D'autre part les remarques suivantes, assez triviales, méritent cependant d'être mentionnées :

- deux exécutions de la procédure ci-dessus peuvent entraîner deux transactions différentes, dans le cas par exemple où le test effectué sur le nombre de places libres est positif pour l'une est négatif pour l'autre ;
- un processus peut exécuter répétitivement une procédure –par exemple pour effectuer plusieurs réservations– ce qui déclenche une *séquence de transactions* ;
- deux processus indépendants peuvent exécuter indépendamment la même procédure, avec des paramètres qui peuvent être identiques ou non.

On fait toujours l'hypothèse que deux processus différents ne communiquent jamais entre eux autrement que par l'intermédiaire des ordres SQL adressés à la base.

En résumé une transaction est une séquence d'instructions de lecture ou de mise à jour effectuées par un processus client.

1.1.2 Exécutions concurrentes

Pour chaque processus il ne peut y avoir qu'une seule transaction en cours à un moment donné [2], mais plusieurs processus peuvent effectuer simultanément des transactions. On suppose que chaque processus est identifié par un numéro unique qui est soumis au système avec chaque ordre SQL effectué par ce processus.

Voici donc comment on représentera une transaction du processus numéro 1 exécutant la procédure de réservation.

$$Read_1(s); Read_1(c); Write_1(s); Write_1(c); C_1$$

Les symboles c et s désignent les tuples –ici un spectacle s et un client c–lus ou mis à jour par les opérations, tandis que le symbole C désigne un `commit` (on utilisera bien entendu R pour le `rollback`). Dans tout ce chapitre on supposera, sauf exception explicitement mentionnée, que l'unité d'accès à la base est le tuple, et que tout verrouillage s'applique à ce niveau.

Voici une autre transaction, effectuée par le processus numéro 2, pour la même procédure.

2. On ignore donc ici les transactions imbriquées, ou les exécutions en parallèle sur plusieurs processeurs.

$$Read_2(s');$$

On a donc lu le spectacle s', et constaté quil n'y a plus assez de places libres. Enfin le dernier exemple est celui d'une réservation effectuée par un troisième processus.

$$Read_3(s); Read_3(c'); Write_3(s); Write_3(c); C_3$$

Le client c' réserve donc ici des places pour le spectacle s. Les trois processus peuvent s'exécuter au même moment, ce qui revient à soumettre à peu près simultanément les opérations au SGBD. Ce dernier pourrait choisir d'effectuer les transactions *en série*, en commençant par exemple par le processus 1, puis en passant au processus 2, enfin au processus 3. Cette stratégie a l'avantage de garantir de manière triviale la vérification de l'hypothèse d'isolation des exécutions, mais elle est potentiellement très pénalisante puisqu'une longue transaction pourrait metre en attente pour un temps indéterminé de nombreuses petites transactions. C'est d'autant plus injustifié que, le plus souvent, l'entrelacement des opérations est sans danger, et qu'il est possible de contrôler les cas où il pourrait poser des problèmes. Tous les SGBD autorisent donc des *exécutions concurrentes* dans lequelles les opérations s'effectuent alternativement pour des processus différents. Voici un exemple d'exécution concurrente pour les trois processus précédents, dans lequel on a abrégé $read$ et $write$ respectivement par r et w.

$$r_1(s); r_3(s); r_1(c); r_2(s'); r_3(c'); w_3(s); w_1(s); w_1(c); w_3(c); C_1; C_3$$

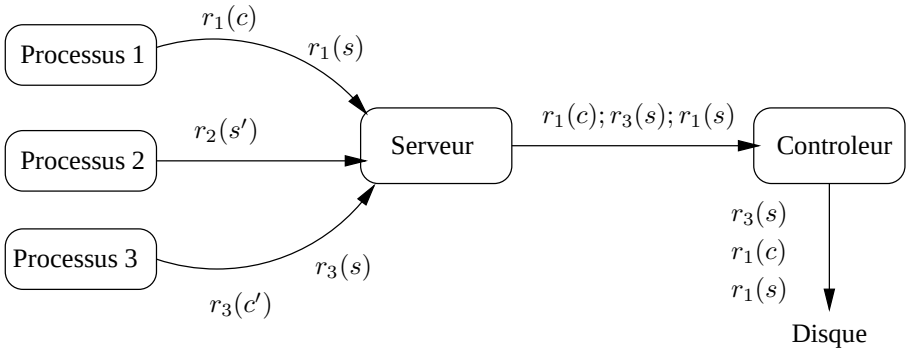


FIGURE 1.1 – Processus soumettant des transactions

Dans un premier temps on peut supposer que l'ordre des opérations dans une exécution concurrente est l'ordre de transmission de ces opérations au système. Comme nous allons le voir sur plusieurs exemples, cette absence de contrôle mène à de nombreuses anomalies qu'il faut absolument éviter. Le SGBD (ou, plus précisément, le module chargé du contrôle de concurrence) effectue donc un ré-ordonnancement si cela s'impose. Cependant, l'entrelacement des opérations ou leur ré-ordonnancement ne signifie *en aucun cas* que l'ordre des opérations internes à une transaction T_i peut être modifié. En « effaçant » d'une exécution concurrente toutes les opérations pour les transactions $T_j, i \neq j$, on doit retrouver exactement les opérations de T_i, dans l'ordre où elles ont été soumises au système. Ce dernier ne change *jamais* cet ordre car cela reviendrait à transformer le programme en cours d'exécution.

La figure 1.1 montre une première ébauche des composants intervenant dans le contrôle de concurrence. On y retrouve les trois processus précédents, chacun soumettant des instructions au serveur. Le processus 1 soumet par exemple $r_1(s)$, puis $r_1(c)$. Le serveur transmet les instructions, dans l'ordre d'arrivée (ici, d'abord $r_1(s)$, puis $r_3(s)$, puis $r_1(c)$), à un module spécialisé, le contrôleur qui, lui, peut réordonnancer l'exécution s'il estime que la cohérence en dépend. Sur l'exemple de la figure 1.1, le contrôleur exécute, dans l'ordre $r_1(s)$, $r_1(c)$ puis $r_3(s)$.

1.1.3 Propriétés ACID des transactions

Les SGBD garantissent que l'exécution des transactions satisfait un ensemble de bonnes propriétés que l'on résume commodément par l'acronyme ACID (Atomicité, Cohérence, Isolation, Durabilité).

Isolation

L'isolation est la propriété qui garantit que l'exécution d'une transaction *semble* totalement indépendante des autres transactions. Le terme « semble » est bien entendu relatif au fait que, comme nous l'avons vu ci-dessus, une transaction s'exécute en fait en concurrence avec d'autres. Du point de vue de l'utilisateur, tout se passe donc comme si son programme, pendant la période de temps où il accède au SGBD, était seul à disposer des ressources du serveur de données.

Le niveau d'isolation totale, telle que défini ci-dessus, est dit *sérialisable* puisqu'il est équivalent du point de vue du résultat obtenu, à une exécution *en série* des transactions. C'est une propriété forte, dont l'inconvénient est d'impliquer un contrôle strict du SGBD qui risque de pénaliser fortement les autres utilisateurs. Les systèmes proposent en fait plusieurs niveaux d'isolation dont chacun représente un compromis entre la sérialisabilité, totalement saine mais pénalisante, et une isolation partielle entraînant moins de blocages mais plus de risques d'interactions perturbatrices.

Le choix du bon niveau d'isolation, pour une transaction donnée, est de la responsabilité du programmeur et implique une bonne compréhension des dangers courus et des options proposées par les SGBD. Le présent chapitre est essentiellement consacré à donner les informations nécessaires à ce choix.

Garantie de la commande `COMMIT` (durabilité)

L'exécution d'un `COMMIT` rend permanentes toutes les mises à jour de la base effectuées durant la transaction. Le système garantit que toute interruption du système survenant après le `COMMIT` ne remettra pas en cause ces mises à jour. Cela signifie également que tout `COMMIT` d'une transaction T rend impossible l'annulation de cette même transaction avec `ROLLBACK`. Les anciennes données sont perdues, et il n'est pas possible de revenir en arrière.

Le `COMMIT` a également pour effet de libérer tous les verrous mis en place durant la transaction pour prévenir les intéractions avec d'autres transactions. Un des effets du `COMMIT` est donc de « libérer » les éventuelles ressources bloquées par la transaction validée. Une bonne pratique, quand la nature de la transaction le permet, est donc d'effectuer les opérations potentiellement bloquantes le plus tard possible, juste avant le `COMMIT`, ce qui diminue d'autant la période pendant laquelle les données en concurrence sont verrouillées.

Garantie de la commande `ROLLBACK` (atomicité)

Le `ROLLBACK` annule toutes les modifications de la base effectuées pendant la transaction. Il relâche également tous les verrous posés sur les données pendant la transaction par le système, et libère donc les éventuels autres processus en attente de ces données.

Un `ROLLBACK` peut être déclenché explicitement par l'utilisateur, ou effectué par le système au moment d'une reprise sur panne ou de tout autre problème empêchant la poursuite normale de la transaction. Dans tout les cas l'état des données modifiées par la transaction revient, après le `ROLLBACK`, à ce qu'il était au début de la transaction. Cette commande garantit donc *l'atomicité* des transactions, puisqu'une transaction est soit effectuée totalement (donc jusqu'au `COMMIT` qui la conclut) soit annulée totalement (par un `ROLLBACK` du système ou de l'utilisateur).

L'annulation par `ROLLBACK` rend évidemment impossible toute validation de la transaction : les mises à jour sont perdues et doivent être resoumises.

Cohérence des transactions

Le maintien de la cohérence peut relever aussi bien du système que de l'utilisateur selon l'interprétation du concept de « cohérence ».

Pour le système, la cohérence d'une base est définie par les contraintes associées au schéma. Ces contraintes sont notamment :

- les contraintes de clé primaire (clause `PRIMARY KEY`) ;
- l'intégrité référentielle (clause `FOREIGN KEY`) ;
- les contraintes `CHECK` ;
- les contraintes implantées par *triggers*.

Toute violation de ces contraintes entraîne non seulement le rejet de la commande SQL fautive, mais également un `rollback` automatique puisqu'il est hors de question de laisser un programme s'exécuter seulement partiellement.

Mais la cohérence désigne également un état de la base considéré comme satisfaisant pour l'application, sans que cet état puisse être toujours spécifié par des contraintes SQL. Dans le cas par exemple de notre programme de réservation, la base est cohérente quand :

1. le nombre de places prises pour un spectacle est le même que la somme des places réservées pour ce spectacle par les clients ;
2. le solde de chaque client est supérieur à 0.

Il n'est pas facile d'exprimer cette contrainte avec les commandes DDL de SQL, mais on peut s'assurer qu'elle est respectée en écrivant soigneusement les procédures de mises à jour pour qu'elle tirent parti des propriétés ACID du système transactionnel.

Reprenons notre procédure de réservation, en supposant que la base est initialement dans un état cohérent (au sens donné ci-dessus de l'équilibre entre le nombre de places prises et le nombres de places réservées). Les propriétés d'atomicité (A) et de durabilité (D) garantissent que :

- la transaction s'effectue totalement, valide les deux opérations de mise à jour qui garantissent l'équilibre, et laisse donc après le `COMMIT` la base dans un état cohérent ;
- la transaction est interrompue pour une raison quelconque, et la base revient alors à l'état initial, cohérent.

De plus toutes les contraintes définies dans le schéma sont respectées si la transaction arrive à terme (sinon le système l'annule).

Tout se passe bien parce que le programmeur a placé le `COMMIT` au bon endroit. Imaginons maintenant qu'un `COMMIT` soit introduit après le premier `UPDATE`. Si le second `UPDATE` soulève une erreur, le système effectue un `ROLLBACK` jusqu'au `COMMIT` qui précède, et laisse donc la base dans un état incohérent –déséquilibre entre les places prises et les places réservées– du point de vue de l'application.

Dans ce cas l'erreur vient du programmeur qui a défini deux transactions au lieu d'une, et entraîné une validation à un moment où la base est dans un état intermédiaire. La leçon est simple : *tous les `COMMIT` et `ROLLBACK` doivent être placés de manière à s'exécuter au moment où la base est dans un état cohérent*. Il faut toujours se souvenir qu'un `COMMIT` ou un `ROLLBACK` marque la fin d'une transaction, et définit donc l'ensemble des opérations qui doivent s'exécuter solidairement (ou « atomiquement »).

Un défaut de cohérence peut enfin résulter d'un mauvais entrelacement des opérations concurrentes de deux transactions, dû à un niveau d'isolation insuffisant. Cet aspect sera illustré dans la prochaine section consacrée aux conséquences d'une absence de contrôle.

1.1.4 Un exemple de concurrence sous ORACLE

Pour conclure cette première présentation par un exemple concret, voici deux sessions effectuées en concurrence sous ORACLE. Ces sessions s'effectuent avec l'utilitaire SQLPLUS qui permet d'entrer directement des requêtes sur la base comprenant les tables *Client* et *Spectacle* décrites en début de chapitre. Les opérations effectuées consistent à réserver, pour le même spectacle, 5 places pour la session 1, et 7 places pour la session 2.

Voici les premières requêtes effectuées par la session 1.

```
Session1>SELECT * FROM Client;

 ID_CLIENT NB_PLACES_RESERVEES      SOLDE
---------- ------------------- ----------
         1                   3       2000
```

```
Session1>SELECT * FROM Spectacle;

ID_SPECTACLE NB_PLACES_OFFERTES NB_PLACES_LIBRES      TARIF
------------ ------------------ ---------------- ----------
           1                250              200         10
```

On a donc un client et un spectacle. La session 1 augmente maintenant le nombre de places réservées.

```
Session1>UPDATE Client SET nb_places_reservees = nb_places_reservees + 5
           WHERE id_client=1;

1 row updated.

Session1>SELECT * FROM Client;

 ID_CLIENT NB_PLACES_RESERVEES      SOLDE
---------- ------------------- ----------
         1                   8       2000

Session1>SELECT * FROM Spectacle;

ID_SPECTACLE NB_PLACES_OFFERTES NB_PLACES_LIBRES      TARIF
------------ ------------------ ---------------- ----------
           1                250              200         10
```

Après l'ordre `UPDATE`, si on regarde le contenu des tables *Client* et *Spectacle*, on voit bien l'effet des mises à jour. Notez que la base est ici dans un état instable puisqu'on n'a pas encore diminué le nombre de places libres. Voyons maintenant les requêtes de lecture pour la session 2.

```
Session2>SELECT * FROM Client;

 ID_CLIENT NB_PLACES_RESERVEES      SOLDE
---------- ------------------- ----------
         1                   3       2000

Session2>SELECT * FROM Spectacle;

ID_SPECTACLE NB_PLACES_OFFERTES NB_PLACES_LIBRES      TARIF
------------ ------------------ ---------------- ----------
           1                250              200         10
```

Pour la session 2, la base est dans l'état initial. L'isolation implique que les mises à jour effectuées par la session 1 sont invisibles puisqu'elles ne sont pas encore validées. Maintenant la session 2 tente d'effectuer la mise à jour du client.

```
Session2>UPDATE Client SET nb_places_reservees = nb_places_reservees + 7
  2              WHERE id_client=1;
```

La transaction est mise en attente car, appliquant des techniques de verrouillage qui seront décrites plus loin, ORACLE a réservé le tuple de la table *Client* pour la session 1. Seule la session 1 peut maintenant progresser. Voici la fin de la transaction pour cette session.

```
Session1>UPDATE Spectacle SET nb_places_libres = nb_places_libres - 5
  2        WHERE id_spectacle=1;

1 row updated.

Session1>COMMIT;
```

Après le `COMMIT`, la session 2 est libérée,

```
Session2>UPDATE Client SET nb_places_reservees = nb_places_reservees + 7
  2             WHERE id_client=1;

1 row updated.
Session2>
Session2>SELECT * FROM Client;

 ID_CLIENT NB_PLACES_RESERVEES      SOLDE
---------- ------------------- ----------
         1                  15       2000

Session2>SELECT * FROM Spectacle;

ID_SPECTACLE NB_PLACES_OFFERTES NB_PLACES_LIBRES      TARIF
------------ ------------------ ---------------- ----------
           1                250              195         10
```

Une sélection montre que les mises à jour de la session 1 sont maintenant visibles, puisque le `COMMIT` les a validé définitivement. De plus on voit également la mise à jour de la session 2. Notez que pour l'instant la base est dans un état incohérent puisque 12 places sont réservées par le client, alors que le nombre de places libres a diminué de 5. La seconde session doit décider, soit d'effectuer la mise à jour de *Spectacle*, soit d'effectuer un `ROLLBACK`. En revanche il et absolument exclu de demander un `COMMIT` à ce stade, même si on envisage de mettre à jour *Spectacle* ultérieurement. Si cette dernière mise à jour échouait, ORACLE ramènerait la base à l'état – incohérent – du dernier `COMMIT`,

Voici ce qui se passe si on effectue le `ROLLBACK`.

```
Session2>ROLLBACK;

Rollback complete.

Session2>SELECT * FROM Client;

 ID_CLIENT NB_PLACES_RESERVEES      SOLDE
---------- ------------------- ----------
         1                   8       2000

Session2>SELECT * FROM Spectacle;

ID_SPECTACLE NB_PLACES_OFFERTES NB_PLACES_LIBRES      TARIF
------------ ------------------ ---------------- ----------
           1                250              195         10
```

Les mises à jour de la session 2 sont annulées : la base se retrouve dans l'état connu à l'issue de la session 1.

Ce court exemple montre les principales conséquences « visibles » du contrôle de concurrence effectué par un SGBD :

1. chaque processus/session dispose d'une vision des données en phase avec les opérations qu'il vient d'effectuer ;
2. les données modifiées mais non validées par un processus ne sont pas visibles pour les autres ;
3. les accès concurrents à une même ressource peuvent amener le système à mettre en attente certains processus.

D'autre part le comportement du contrôleur de concurrence peut varier en fonction du niveau d'isolation choisi qui est, dans l'exemple ci-dessus, celui adopté par défaut dans ORACLE (`READ COMMITTED`). Le choix (ou l'acceptation par défaut) d'un niveau d'isolation inapproprié peut entraîner diverses anomalies que nous allons maintenant étudier.

1.2 Problèmes consécutifs à une concurrence sans contrôle

Pour illustrer les problèmes potentiels en cas d'absence de contrôle de concurrence, ou de techniques insuffisantes, on va considérer un modèle d'exécution très simplifié, dans lequel il n'existe qu'une seule version de chaque tuple (stocké par exemple dans un fichier séquentiel). Chaque instruction est effectuée par le système, dès qu'elle est reçue, de la manière suivante :

1. quand il s'agit d'une instruction de lecture, on lit le tuple dans le fichier et on le transmet au processus ;
2. quand il s'agit d'une instruction de mise à jour, on écrit directement le tuple dans le fichier en écrasant la version précédente.

Les problèmes consécutifs à ce mécanisme simpliste peuvent être classés en deux catégories : défauts d'isolation menant à des incohérences –*défauts de sérialisabilité*–, et difficultés dus à une mauvaise prise en compte des `COMMIT` et `ROLLBACK`, ou *défauts de recouvrabilité*. Les exemples qui suivent ne couvrent pas l'exhaustivité des situations d'anomalies, mais illustrent les principaux types de problèmes.

1.2.1 Défauts de sérialisabilité

Considérons pour commencer que le système n'assure aucune isolation, aucun verrouillage, et ne connaît ni le `COMMIT` ni le `ROLLBACK`. Même en supposant que toutes les exécutions concurrentes s'exécutent intégralement sans jamais rencontrer de panne, on peut trouver des situations où l'entrelacement des instructions conduit à des résultats différents de ceux obtenus par une exécution en série. De telles exécutions sont dites *non sérialisables* et aboutissent à des incohérences dans la base de données.

Les mises à jour perdues

Le problème de mise à jour perdue survient quand deux transactions lisent chacune une même donnée en vue de la modifier par la suite. Prenons à nouveau deux exécutions concurrentes du programme RÉSERVATION, désignées par T_1 et T_2. Chaque exécution consiste à réserver des places pour le même spectacle, mais pour deux clients distincts c_1 et c_2. L'ordre des opérations reçues par le serveur est le suivant :

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

Donc 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. Imaginons maintenant que l'on se trouve dans la situation suivante :

1. il reste 50 places libres pour le spectacle s, c_1 et c_2 n'ont pour l'instant réservé aucune place ;
2. T_1 veut réserver 5 places pour s ;
3. T_2 veut réserver 2 places pour s.

Voici le résultat du déroulement imbriqué des deux exécutions $T_1(s, 5, c_1)$ et $T_2(s, 2, c_2)$, en supposant que la séquence des opérations est celle donnée ci-dessus. On se concentre pour l'instant sur les évolutions du nombre de places vides.

1. T_1 lit s et c_1 et constate qu'il reste 50 places libres ;
2. T_2 lit s et c_2 et constate qu'il reste 50 places libres ;
3. T_2 écrit s avec nb places = $50 - 2 = 48$.
4. T_2 écrit le nouveau compte de c_2.
5. T_1 écrit s avec nb places = $50 - 5 = 45$.
6. T_1 écrit le nouveau compte de c_1.

À la fin de l'exécution, on constate un problème : il reste 45 places vides sur les 50 initiales alors que 7 places ont effectivement été réservées et payées. Le problème est clairement issu d'une mauvaise imbrication des opérations de T_1 et T_2 : T_2 lit et modifie une information que T_1 a déjà lue en vue de la modifier. La figure 1.2 montre la superposition temporelle des deux transactions. On voit que T_1 et T_2 lisent chacun, dans leurs espaces mémoires respectifs, d'une copie de S indiquant 50 places disponibles, et

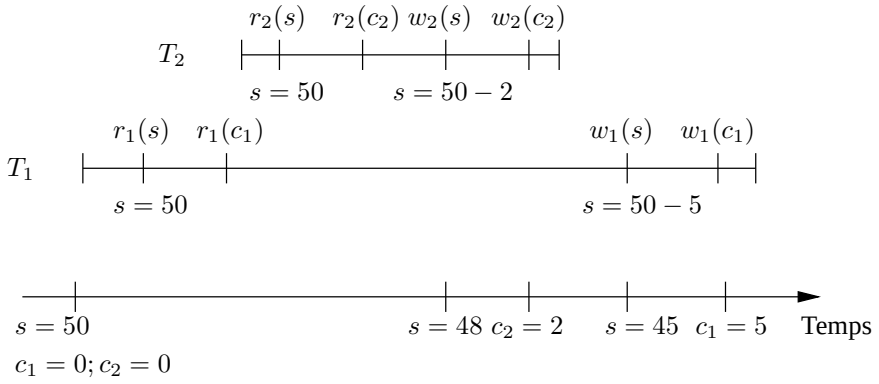


FIGURE 1.2 – Exécution concurrente de T_1 et T_2

que cette valeur sert au calcul du nombre de places restantes sans tenir compte de la mise à jour effectuée par l'autre transaction.

On arrive donc à une base de données incohérente alors que chaque transaction, prise isolément, est correcte, et qu'elles se sont toutes deux exécutées complètement.

Une solution radicale pour éviter le problème est d'exécuter *en série* T_1 et T_2. Il suffit pour cela de bloquer une des deux transactions tant que l'autre n'a pas fini de s'exécuter. On obtient alors l'exécution concurrente suivante :

$$r_1(s)r_1(c)w_1(s)w_1(c)r_2(s)r_2(c)w_2(s)w_2(c)$$

On est assuré dans ce cas qu'il n'y a pas de problème car T_2 lit la donnée écrite par T_1 qui a fini de s'exécuter et ne créera donc plus d'interférence. La figure 1.3 montre que la cohérence est obtenue ici par la lecture de s dans T_2 qui ramène la valeur 50 pour le nombre de places disponibles, ce qui revient bien à tenir compte des mises à jour de T_1.

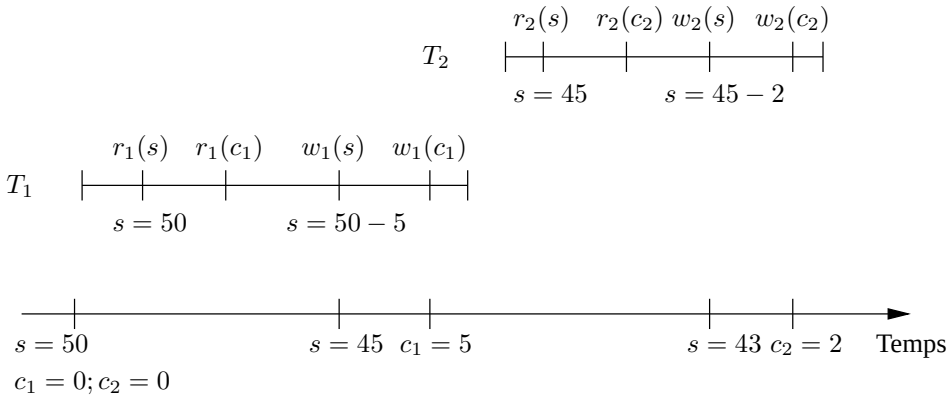


FIGURE 1.3 – Exécution en série de T_1 et T_2

Cette solution de « concurrence zéro » est difficilement acceptable car elle revient à bloquer tous les processus sauf un. Dans un système où de très longues transactions (par exemple l'exécution d'un traitement lourd d'équilibrage de comptes) cohabitent avec de très courtes (des saisies interactives), les utilisateurs seraient extrêmement pénalisés.

Heureusement l'exécution en série est une contrainte trop forte, comme le montre l'exemple suivant.

Exemple 2. *Exécution imbriquée de P_1 et P_2.*

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

Suivons pas à pas l'exécution :

1. T_1 lit s et c_1. Nombre de places libres : 50.
2. T_1 écrit s avec nb places = $50 - 5 = 45$.
3. T_2 lit s. Nombre de places libres : 45.
4. T_2 lit c_2.
5. T_2 écrit s avec nombre de places = $45 - 2 = 43$.
6. T_1 écrit le nouveau compte du client c_1.
7. T_2 écrit le nouveau compte du client c_2.

Cette exécution est correcte : on obtient un résultat strictement semblable à celui issu d'une exécution en série. Il existe donc des exécutions imbriquées qui sont aussi correctes qu'une exécution en série et qui permettent une meilleure concurrence. Le gain, sur notre exemple, peut paraître mineur, mais il il faut imaginer l'intérêt de débloquer rapidement de longues transactions qui ne rentrent en concurrence que sur une petite partie des tuples qu'elles manipulent.

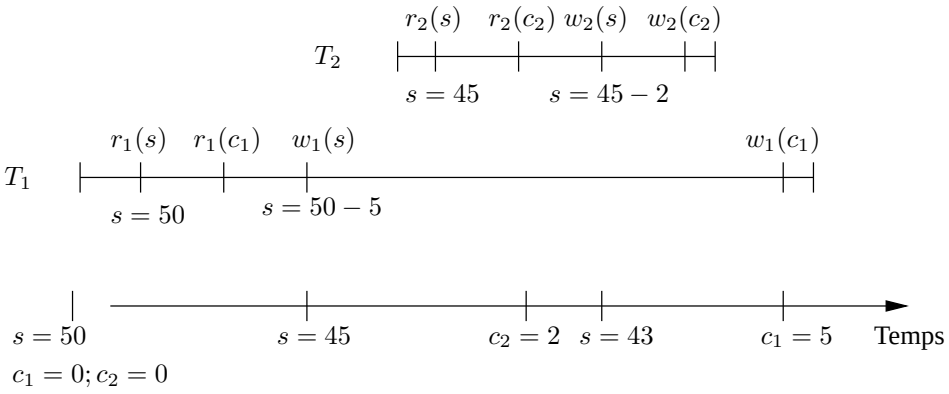


FIGURE 1.4 – Exécution concurrente correcte de T_1 et T_2

On parle d'exécutions *sérialisables* pour désigner des exécutions concurrentes équivalentes à une exécution en série. Un des buts d'un système effectuant un contrôle de concurrence est d'obtenir de telles exécutions. Dans l'exemple qui précède, cela revient à mettre T_2 en attente tant que T_1 n'a pas écrit s. Nous verrons un peu plus loin par quelles techniques on peut automatiser ce genre de mécanisme.

Tuples fantômes

Voici un autre type de problème dû à l'interaction de plusieurs transactions : certaines modifications de la base peuvent devenir visibles *pendant* l'exécution d'une transaction T à cause des mises à jour effectuées *et* validées par d'autres transactions. Ces modifications peuvent rendre le résultat de l'exécution des requêtes en lecture effectuées par T *non répétables* : la première exécution d'une requête q renvoie un ensemble de tuples différent d'une seconde exécution de q effectuée un peu plus tard, parce certain tuples ont disparu ou sont apparus dans l'intervalle (on parle de *tuples fantômes*).

Prenons le cas d'une procédure effectuant un contrôle de cohérence sur notre base de données : elle effectue tout d'abord la somme des places prises par les clients, puis elle compare cette somme au nombre de places réservées pour le spectacle. La base est cohérente si le nombre de places libres est égal au nombre de places réservées

```
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
```

Une exécution de la procédure CONTRÔLE se modélise simplement comme une séquence $r_c(c_1) \dots r_c(c_n) r_c(s)$ d'une lecture des n clients $\{c_1, \dots, c_n\}$ suivie d'une lecture de s (le spectacle). Supposons maintenant qu'on exécute cette procédure sous la forme d'une transaction T_1, en concurrence avec une réservation $Res(c_1, s, 5)$, avec l'entrelacement suivant.

$$r_1(c_1) r_1(c_2) Res(c_2, s, 2) \dots r_1(c_n) r_1(s)$$

Le point important est que l'exécution de $Res(c_2, s, 2)$ va augmenter de 2 le nombre de places réservées par c_2, et diminuer de 2 le nombre de places disponibles dans s. Mais la procédure T_1 a lu le spectacle s *après* la transaction Res, et le client c_2 *avant*. Seule une partie des mises à jour de Res sera donc visible, avec pour résultat la constatation d'une incohérence alors que ce n'est pas le cas.

La figure 1.5 résume le déroulement de l'exécution (en supposant deux clients seulement). Au début de la session 1, la base est dans un état cohérent. On lit 5 pour le client 1, 0 pour le client 2. À ce moment-là intervient la réservation T_2, qui met à jour le client 2 et le spectacle s. C'est cette valeur mise à jour que vient lire T_1.

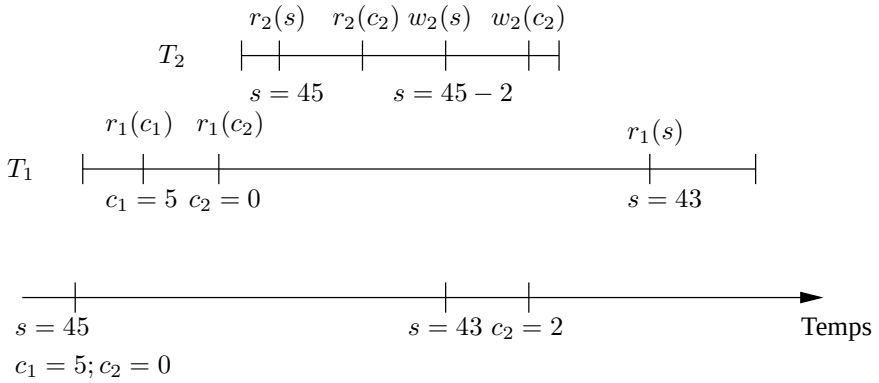


FIGURE 1.5 – Exécution concurrente d'un contrôle et d'une réservation

Il faut noter que T_1 n'effectue pas de mise à jour et ne lit que des données validées. Il est clair que le problème vient du fait que la transaction T_1 accède, durant son exécution, à des versions différentes de la base et qu'un contrôle de cohérence ne peux être fiable dans ces conditions. On peut noter que cette exécution n'est pas sérialisable, puisque le résultat est clairement différent de celui obtenu par l'exécution successive de T_1 et de Res.

On désigne ce type de comportement par le terme de *lectures non répétables* (*non repeatable reads* en anglais). Si T_1 effectue deux lectures de s avant et après l'exécution de Res, elle constatera une modification. De même toute insertion, modification ou suppression d'un tuple dans la table *Client* sera visible ou non selon que T_1 effectuera la lecture avant ou après la mise à jour concurrente.

Les tuples fantômes constituent un des effets désagréables d'une exécution concurrente. On pourrait considérer, à tort, que le risque d'avoir une telle interaction est faible. En fait l'exécution de requêtes sur une période assez longue est très fréquente dans les applications bases de données qui s'appuient sur des curseurs (voir le chapitre consacré aux procédures stockées) permettant de parcourir un résultat tuple à tuple, avec un temps de traitement de chaque tuple qui peut alonger considérablement le temps seulement consacré au parcours du résultat.

1.2.2 Défauts de recouvrabilité

Le fait de garantir une imbrication sérialisable des exécutions concurrentes serait suffisant dans l'hypothèse où tous les programmes terminent normalement en validant les mises à jour effectuées. Malheureusement ce n'est pas le cas puisqu'il arrive que l'on doive annuler les opérations d'entrées sorties effectuées par un programme. Les anomalies d'une exécution concurrente dus au effets non contrôlés des `COMMIT` et `ROLLBACK` constituent une seconde catégorie de problèmes qualifiés collectivement de *défauts de recouvrabilité*.

Nous allons maintenant étendre notre modèle d'exécution simplifié en introduisant les commandes `COMMIT` et `ROLLBACK`. Attention : il ne s'agit que d'une version simplifiée de l'un des algorithmes possibles pour implanter les `COMMIT` et `ROLLBACK` :

1. le contrôleur conserve, chaque fois qu'une transaction T_i modifie un tuple t, l'image de t *avant* la mise à jour, dénotée t_i^{ia} ;
2. quand une transaction T_i effectue un `COMMIT`, les images avant associées à T_i sont effacées ;
3. quand une transaction T_i effectue un `ROLLBACK`, toutes les images avant sont écrites dans la base pour ramener cette dernière dans l'état du début de la transaction.

Imaginons par exemple que le programme de réservation soit interrompu après avoir exécuté les instructions suivantes :

$$r_1(s)r_1(c_1)w_1(s)$$

Au moment d'effectuer $w_1(s)$, notre système a conservé l'image avant modification s_1^{ia} du spectacle s. L'interruption intervient avant le `COMMIT`, et la situation obtenue n'est évidemment pas satisfaisante puisqu'on on a diminué le nombre de places libres sans débiter le compte du client. Le `ROLLBACK` consiste ici à effectuer l'opération $w_1(s_1^{ia})$ pour réécrire l'image avant et revenir à l'état initial de la base.

L'implantation d'un tel mécanisme demande déjà un certain travail, mais cela ne suffit malheureusement toujours pas à garantir des exécutions concurrentes correctes, comme le montrent les exemples qui suivent.

Lectures sales

Revenons à nouveau à l'exécution concurrente de nos deux transactions T_1 et T_2, en considérant maintenant l'impact des validations ou annulations par `COMMIT` ou `ROLLBACK`. Voici un premier exemple :

$$r_1(s)r_1(c_1)w_1(s)r_2(s)r_2(c_2)w_2(s)w_2(c_2)C_2w_1(c_1)R_1$$

Le nombre de places disponibles a donc été diminué par T_1 et repris par T_2, avant que T_1 n'annule ses réservations. On peut noter que cette exécution concurrente est sérialisable, au sens où l'ordre des opérations donne un résultat identique à une exécution en série.

Le problème vient ici du fait que T_1 est annulée *après* que la transaction T_2 a lu une information mise à jour par T_1, manipulé cette information, effectué une écriture, et enfin validé. On parle de « lectures sales » (*dirty read* en anglais) pour désigner l'accès par une transaction à des tuples modifiés *mais non encore validés* par une autre transaction. L'exécution correcte du `ROLLBACK` est ici impossible, puisqu'on se trouve face au dilemne suivant :

1. soit on écrit dans s l'image avant gérée par T_1, s_1^{ia}, mais on écrase du coup la mise à jour de T_2 alors que ce dernier a effectué un `COMMIT` ;
2. soit on conserve le tuple s validé par T_2, et on annule seulement la mise à jour sur c_1, mais la base est alors incohérente.

On se trouve donc face à une exécution concurrente qui rend impossible le respect d'au moins une des deux propriétés transactionnelles requises : la durabilité (garantie du `COMMIT`) ou l'atomicité (garantie du `ROLLBACK`). Une telle exécution est dite *non recouvrable*, et doit absolument être évitée.

La lecture sale transmet un tuple modifié et non validé par une transaction (ici T_1) à une autre transaction (ici T_2). La première transaction, celle qui a effectué la lecture sale, devient donc dépendante du

choix de la seconde, qui peut valider ou annuler. Le problème est agravé irrémédiablement quand la première transaction valide avant la seconde, comme dans l'exemple précédent, ce qui rend impossible une annulation globale.

Une exécution non-recouvrable introduit un conflit insoluble entre les `COMMIT` effectués par une transaction et les `ROLLBACK` d'une autre. On pourrait penser à interdire à une transaction T_2 ayant effectué des lectures sales d'une transaction T_1 de valider avant T_1. On accepterait alors la situation suivante :

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

Ici, le `ROLLBACK` de T_1 intervient sans que T_2 n'ait validé. Il faut alors impérativement que le système effectue également un `ROLLBACK` de T_2 pour assurer la cohérence de la base : on parle *d'annulations en cascade* (noter qu'il peut y avoir plusieurs transactions à annuler).

Quoique acceptable du point de vue de la cohérence de la base, ce comportement est difficilement envisageable du point de vue de l'utilisateur qui voit ses transactions interrompues sans aucune explication liée à ses propres actions. Aucun SGBD ne pratique d'annulation en cascade. La seule solution est donc simplement d'interdire les *dirty read*. Nous verrons qu'il existe deux solutions : soit la lecture lit *l'image avant*, qui par définition est une valeur validée, soit on met en attente les lectures sur des tuples en cours de modification.

Écriture sale

Imaginons qu'une transaction T ait modifé un tuple t, puis qu'un `ROLLBACK` intervienne. Dans ce cas, comme indiqué ci-dessus, il est nécessaire de restaurer la valeur qu'avait t *avant* le début de la transaction (« l'image avant »). Cela soulève des problèmes de concurrence illustrés par l'exemple suivant, toujours basé sur nos deux transactions T_1 et T_2. ci-dessous.

$$r_1(s)r_1(c_1)r_2(s)w_1(s)w_1(c_1)r_2(c_2)w_2(s)R_1w_2(c_2)C_2$$

Ici il n'y a pas de lecture sale, mais une « écriture sale » (*dirty write*) car T_2 écrit s après une mise à jour T_1 sur s, et sans que T_1 ait validé. Puis T_1 annule et T_2 valide. Que se passe-t-il au moment de l'annulation de T_1 ? On doit restaurer l'image avant connue de T_1, mais cela revient clairement à annuler la mise à jour de T_2.

On peut sans doute envisager des techniques plus sophistiquées de gestion des `ROLLBACK`, mais le principe de remplacement par l'image avant a le mérite d'être relativement simple à mettre en place, ce qui implique l'interdiction des écritures sales.

En résumé, on peut avoir des transactions sérialisables et non recouvrables et réciproquement. Le respect des propriétés ACID des transactions impose au SGBD d'assurer :

- la sérialisabilité des transactions ;
- la recouvrabilité dite *stricte*, autrement dit sans lectures ni écritures sales.

Les SGBD s'appuient sur un ensemble de techniques assez sophistiquées dont nous allons donner un aperçu ci-dessous. Il faut noter dès maintenant que le recours à ces techniques peut être pénalisant pour les performances (ou, plus exactement, la « fluidité » des exécutions). Dans certains cas –fréquents– où le programmeur sait qu'il n'existe pas de risque lié à la concurrence, on peut relâcher le niveau de contrôle effectué par le système afin d'éviter des blocages inutiles.

1.3 Les niveaux d'isolation SQL : illustration avec MySQL

Du point du programmeur d'application, l'objectif du contrôle de concurrence est de garantir la cohérence des données et d'assurer la recouvrabilité des transactions. Ces bonnes propriétés sont obtenues en choisissant un niveau d'isolation approprié qui garantit qu'aucune interaction avec un autre utilisateur ne viendra perturber le déroulement d'une transaction, empêcher son annulation ou sa validation.

Une option possible est de toujours choisir un niveau d'isolation maximal, garantissant la sérialisabilité des transactions, mais le mode `SERIALIZABLE` a l'inconvénient de ralentir le débit transactionnel pour

des applications qui n'ont peut-être pas besoin de contrôles aussi stricts. On peut chercher à obtenir de meilleures performances en choisissant explicitement un niveau d'isolation moins élevé, soit parce que l'on sait qu'un programme ne posera jamais de problème de concurrence, soit parce les problèmes éventuels sont considérés comme peu importants par rapport au bénéfice d'une fluidité améliorée.

On considère dans ce qui suit deux exemples. Le premier consiste en deux exécutions concurrentes du programme RÉSERVATION, désignées respectivement par T_1 et T_2.

Exemple 3. [Concurrence entre mises à jour] Chaque exécution consiste à réserver des places pour le même spectacle, mais pour deux clients distincts c_1 et c_2. L'ordre des opérations reçues par le serveur est le suivant :

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

Au départ nous sommes dans la situation suivante :

1. il reste 50 places libres pour le spectacle s, c_1 et c_2 n'ont pour l'instant réservé aucune place ;
2. T_1 veut réserver 5 places pour s ;
3. T_2 veut réserver 2 places pour s.

□

Donc 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. Aucun client n'a réservé de place.

Le second exemple prend le cas de la procédure effectuant un contrôle de cohérence sur notre base de données, uniquement par des lectures.

Exemple 4. [Concurrence entre lectures et mises à jour] La procédure CONTRÔLE s'effectue en même temps que la procédure `Réservation` qui réserve 2 places pour le client c_2. L'ordre des opérations reçues par le serveur est le suivant (T_1 désigne le contrôle, T_2 la réservation) :

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

Au départ le client c_1 a réservé 5 places. Il reste donc 45 places libres pour le spectacle. La base est dans un état cohérent. □

1.3.1 Les modes d'isolation SQL

La norme SQL ANSI (SQL92) définit quatre modes d'isolation correspondant à quatre compromis différents entre le degré de concurrence et le niveau d'interblocage des transactions. Ces modes d'isolation sont définis par rapport aux trois types d'anomalies que nous avons rencontrés dans les exemples qui précèdent :

1. *Lectures sales* : une transaction T_1 lit un tuple mis à jour par une transaction T_2, *avant* que cette dernière ait validé ;
2. *Lectures non répétables* : une transaction T_1 accède, en lecture ou en mise à jour, à un tuple qu'elle avait déjà lu auparavant, alors que ce tuple a été modifié entre temps par une autre transaction T_2 ;
3. *tuples fantômes* : une transaction T_1 lit un tuple qui a été créé par une transaction T_2 em après le début de T_1.

	Lectures sales	Lectures non répétables	Tuples fantômes
`READ UNCOMMITTED`	Possible	Possible	Possible
`READ COMMITTED`	Impossible	Possible	Possible
`REPEATABLE READ`	Impossible	Impossible	Possible
`SERIALIZABLE`	Impossible	Impossible	Impossible

Il existe un mode d'isolation par défaut qui varie d'un système à l'autre, le plus courant semblant être `READ COMMITTED`.

Le premier mode (`READ UNCOMMITTED`) correspond à l'absence de contrôle de concurrence. Ce mode peut convenir pour des applications non transactionnelles qui se contentent d'écrire « en vrac » dans des fichiers sans se soucier ni des interactions avec d'autres utilisateurs.

Avec le mode `READ COMMITTED`, on ne peut lire que les tuples validés, mais il peut arriver que deux lectures successives donnent des résultats différents. Le résultat d'une requête est cohérent par rapport à l'état de la base *au début de la requête*, Il peut arriver que deux lectures successives donnent des résultats différents si une autre transaction a modifié les données lues, et validé ses modifications. C'est le mode par défaut dans ORACLE par exemple. *Il faut bien être conscient que ce mode ne garantit pas l'exécution sérialisable*. Le SGBD garantit par défaut l'exécution correcte des `COMMIT` et `ROLLBACK` (recouvrabilité), mais pas la sérialisabilité. L'hypothèse effectuée implicitement est que le mode `SERIALIZABLE` est inutile dans la plupart des cas, ce qui est sans doute justifié, et que le programmeur saura le choisir explicitement quand c'est nécessaire, ce qui en revanche est loin d'être évident.

Le mode `REPEATABLE READ` (le défaut dans MySQL/InnoDB par exemple) garantit que le résultat d'une requête est cohérent par rapport à l'état de la base *au début de la transaction*. La réexécution de la même requête donne toujours le même résultat. La sérialisabilité n'est pas assurée, et des tuples peuvent apparaître s'ils ont étés insérés par d'autres transactions (les fameux « tuples fantômes »).

Enfin le mode `SERIALIZABLE` assure les bonnes propriétés (sérialisabilité et recouvrabilité) des transactions et une isolation totale. Tout se passe alors comme si on travaillait sur une « image » de la base de données prise au début de la transaction. Bien entendu cela se fait au prix d'un risque assez élevé de blocage des autres transactions.

Le mode est choisi au début d'une transaction par la commande suivante.

```
SET TRANSACTION ISOLATION LEVEL <option>
```

Une autre option parfois disponible, même si elle ne fait pas partie de la norme SQL, est de spécifier qu'une transaction ne fera que des lectures. Dans ces conditions, on peut garantir qu'elle ne soulèvera aucun problème de concurrence et le SGBD peut s'épargner la peine de poser des verrous. La commande est :

```
SET TRANSACTION READ ONLY;
```

Il devient alors interdit d'effectuer des mises à jour jusqu'au prochain *commit* ou *rollback* : le système rejette ces instructions.

1.3.2 Verrouillage explicite

Certains systèmes permettent de poser explicitement des verrous, ce qui permet pour le programmeur averti de choisir un niveau d'isolation relativement permissif, tout en augmentant le niveau de verrouillage quand c'est nécessaire. ORACLE, PostgreSQL et MySQL proposent notamment une clause `FOR UPDATE` qui peut se placer à la fin d'une requête SQL, et dont l'effet est de réserver chaque tuple lu en vue d'une prochaine modification.

Reprenons notre programme de réservation, et réécrivons les deux premières requêtes de la manière suivante :

```
...
SELECT * INTO v_spectacle
FROM Spectacle
WHERE id_spectacle=v_id_spectacle
FOR UPDATE;
...
SELECT * INTO v_client FROM Client
WHERE id_client=v_id_client
FOR UPDATE;
..
```

On annonce donc explicitement, dans le code, que la lecture d'un tuple (le client ou le spectacle) sera suivie par la mise à jour de ce même tuple. Le système pose alors un *verrou exclusif* qui réserve l'accès au tuple, en lecture ou en mise à jour, à la transaction qui a effectué la lecture avec `FOR UPDATE`. Les verrous posés sont libérés au moment du `COMMIT` ou du `ROLLBACK`.

Voici le déroulement de l'exécution pour l'exécution de l'exemple 3 :

1. T_1 lit s, *après l'avoir verrouillé exclusivement* ;
2. T_1 lit c_1, et verrouille exclusivement ;
3. T_2 veut lire s, et se trouve mise en attente ;
4. T_1 continue, écrit s, écrit c_1, valide et libère les verrous ;
5. T_2 est libéré et s'excéute.

On obtient l'exécution en série suivante.

$$r_1(s)r_1(c_1)w_1(s)w_1(c_1)C_1r_2(s)r_2(c_2)w_2(s)w_2(c_2)C_2$$

La déclaration, avec `FOR UPDATE` de l'intention de modifier un tuple revient à le réserver et donc à empêcher un entrelacement avec d'autres transactions menant soit à un rejet, soit à une annulation autoritaire du SGBD.

Les SGBDs fournissent également des commandes de verrouillage explicite. On peut réserver, en lecture ou en écriture, une table entière. Un verrouillage en lecture est *partagé* : plusieurs transactions peuvent détenir un verrou en lecture sur la même table. Un verrouillage en écriture est *exclusif* : il ne peut y avoir aucun autre verrou, partagé ou exclusif, sur la table.

Voici un exemple avec MySQL dont un des moteurs de stockage, MyISAM, ne gère pas la concurrence. Il faut dont appliquer explicitement un verrouillage si on veut obtenir des exécutions concurrentes sérialisables. En reprenant l'exemple 3 avec verrouillage exclusif (`WRITE`), voici ce que cela donne. La session 1 verrouille (en écriture), lit le spectacle puis le client 1.

```
Session 1> LOCK TABLES Client WRITE, Spectacle WRITE;
Query OK, 0 rows affected (0,00 sec)

Session 1> SELECT * FROM Spectacle WHERE id_spectacle=1;
+--------------+--------------------+------------------+-------+
| id_spectacle | nb_places_offertes | nb_places_libres | tarif |
+--------------+--------------------+------------------+-------+
|            1 |                 50 |               50 | 10.00 |
+--------------+--------------------+------------------+-------+
1 row in set (0,00 sec)

Session 1> SELECT * FROM Client WHERE id_client=1;
+-----------+---------------------+-------+
| id_client | nb_places_reservees | solde |
+-----------+---------------------+-------+
|         1 |                   0 |   100 |
+-----------+---------------------+-------+
```

La session 2 tente de verrouiller est est mise en attente.

```
Session 2> LOCK TABLES Client WRITE, Spectacle WRITE;
```

La session 1 peut finir ses mises à jour, et libère les tables avec la commande `UNLOCK TABLES`.

```
Session 1> UPDATE Spectacle SET nb_places_libres=45
            WHERE id_spectacle=1;
Query OK, 1 row affected (0,00 sec)
```

```
Session 1> UPDATE Client SET solde=50, nb_places_reservees=5
            WHERE id_client=1;
Query OK, 1 row affected (0,00 sec)

Session 1> UNLOCK TABLES;
```

La session 2 peut alors prendre le verrou, effectuer ses lectures et mises à jour, et libérer le verrou. Les deux transactions se sont effectuées en série, et le résultat est donc correct.

La granularité du verrouillage explicite avec `LOCK` est la table entière, ce qui est généralement considéré comme mauvais car un verrouillage au niveau de lignes permet à plusieurs transactions d'accéder à différentes lignes de la table.

Le verrouillage des tables est une solution de « concurrence zéro » qui est rarement acceptable car elle revient à bloquer tous les processus sauf un. Dans un système où de très longues transactions (par exemple l'exécution d'un traitement lourd d'équilibrage de comptes) cohabitent avec de très courtes (des saisies interactives), les utilisateurs sont extrêmement pénalisés. Pour ne rien dire du cas où on oublie de relâcher les verrous...

De plus, dans l'exemple 3, il n'existe pas de conflit sur les clients puisque les deux transactions travaillent sur deux lignes différentes c_1 et c_2. quand seules quelques lignes sont mises à jour, un verrouillage total n'est pas justifié.

Le verrouillage de tables peut cependant être envisagé dans le cas de longues transactions qui vont parcourir toute la table et souhaitent en obtenir une image cohérente. C'est par exemple typiquement le cas pour une sauvegarde. De même, si une longue transaction effectuant des mises à jour est en concurrence avec de nombreuses petites transactions, le risque d'interblocage, temporaire ou définitif (voir plus loin) est important, et on peut envisager de précéder la longue transaction par un verrouillage en écriture.

1.3.3 Le mode `READ COMMITTED`

Le mode `READ COMMITTED`, adopté par défaut dans ORACLE par exemple, amène un résultat incorrect pour nos deux exemples ! Ce mode ne pose pas de verrou en lecture, et assure simplement qu'une donnée lue n'est pas en cours de modification par une autre transaction. Voici ce qui se passe pour l'exemple 4.

1. On commence par la procédure de contrôle qui lit le premier client, $r_1[c]$. Ce client a réservé 5 places. La procédure de contrôle lit c_2 qui n'a réservé aucune place. Donc le nombre total de places réservées est de 5.
2. Puis c'est la réservation qui s'exécute, elle lit le spectacle, le client 2 (aucun de ces deux tuples n'est en cours de modification). Le client c_2 réserve 2 places, donc au moment où la réservation effectue une `COMMIT`, il y a 43 places libres pour le spectacle, 2 places réservées pour c_2.
3. La session 1 (le contrôle) reprend son exécution et lit s. Comme s est validée on lit la valeur mise à jour juste auparavant par Res, et on trouve donc 43 places libres. La procédure de contrôle constate donc, à tort, une incohérence.

Le mode `READ COMMITTED` est particulièrement inadapté aux longues transactions pour lesquelles le risque est fort de lire des données modifiées et validées après le début de l'exécution. En contrepartie le niveau de verrouillage est faible, ce qui évite les bloquages.

1.3.4 Le mode `REPEATABLE READ`

Dans le mode `REPEATABLE READ`, chaque lecture effectuée par une transaction lit les données telles qu'elles étaient *au début de la transaction*. Cela donne un résultat correct pour l'exemple 4, comme le montre le déroulement suivant.

1. On commence par la procédure de contrôle qui lit le premier client, $r_1[c]$. Ce client a réservé 5 places. La procédure de contrôle lit c_2 qui n'a réservé aucune place. Donc le nombre total de places réservées est de 5.

2. Puis c'est la réservation qui s'exécute, elle lit le spectacle, le client 2 (aucun de ces deux tuples n'est en cours de modification). Le client c_2 réserve 2 places, donc au moment où la réservation effectue une `COMMIT`, il y a 43 places libres pour le spectacle, 2 places réservées pour c_2.
3. La session 1 (le contrôle) reprend son exécution et lit s. Miracle ! La mise à jour de la réservation n'est pas visible car elle a été effectuée *après* le début de la procédure de contrôle. Cette dernière peut donc conclure justement que la base, *telle qu'elle était au début de la transaction*, est cohérente.

Ce niveau d'isolation est suffisant pour que les mises à jour effectuées par une transaction T' pendant l'exécution d'une transaction T ne soient pas visibles de cette dernière. Cette propriété est extrêmement utile pour les longues transactions, et elle a l'avantage d'être assurée sans aucun verrouillage.

En revanche le mode `REPEATABLE READ` ne suffit toujours pas pour résoudre le problème des mises à jour perdues. Reprenons une nouvelle fois l'exemple 3. Voici un exemple concret d'une session sous MySQL/InnoDB, SGBD dans lequel le mode d'isolation par défaut est `REPEATABLE READ`. C'est la première session qui débute, avec des lectures.

```
Session 1> START TRANSACTION;
Query OK, 0 rows affected (0,00 sec)

Session 1> SELECT * FROM Spectacle WHERE id_spectacle=1;
+--------------+--------------------+------------------+-------+
| id_spectacle | nb_places_offertes | nb_places_libres | tarif |
+--------------+--------------------+------------------+-------+
|            1 |                 50 |               50 | 10.00 |
+--------------+--------------------+------------------+-------+
1 row in set (0,01 sec)

Session 1> SELECT * FROM Client WHERE id_client=1;
+-----------+---------------------+-------+
| id_client | nb_places_reservees | solde |
+-----------+---------------------+-------+
|         1 |                   0 |   100 |
+-----------+---------------------+-------+
```

La session 1 constate donc qu'aucune place n'est réservée. Il reste 50 places libres. La session 2 exécute à son tour les lectures.

```
Session 2> START TRANSACTION;
Query OK, 0 rows affected (0,00 sec)

Session 2> SELECT * FROM Spectacle WHERE id_spectacle=1;
+--------------+--------------------+------------------+-------+
| id_spectacle | nb_places_offertes | nb_places_libres | tarif |
+--------------+--------------------+------------------+-------+
|            1 |                 50 |               50 | 10.00 |
+--------------+--------------------+------------------+-------+
1 row in set (0,00 sec)

Session 2> SELECT * FROM Client WHERE id_client=2;
+-----------+---------------------+-------+
| id_client | nb_places_reservees | solde |
+-----------+---------------------+-------+
|         2 |                   0 |    60 |
+-----------+---------------------+-------+
```

Maintenant la session 2 effectue sa réservation de 2 places. Pensant qu'il en reste 50 avant la mise à jour, elle place le nombre 48 dans la table *Spectacle*.

```
Session 2> UPDATE Spectacle SET nb_places_libres=48
           WHERE id_spectacle=1;
Query OK, 1 row affected (0,00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

Session 2> UPDATE Client SET solde=40, nb_places_reservees=2
             WHERE id_client=2;
Query OK, 1 row affected (0,00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

Session 2> commit;
Query OK, 0 rows affected (0,00 sec)
```

Pour l'instant InnoDB ne dit rien. La session 1 continue alors. Elle aussi pense qu'il reste 50 places libres. La réservation de 5 places aboutit aux requêtes suivantes.

```
Session 1> UPDATE Spectacle SET nb_places_libres=45 WHERE id_spectacle=1;
Query OK, 1 row affected (0,00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

Session 1> UPDATE Client SET solde=50, nb_places_reservees=5 WHERE id_client=1;
Query OK, 1 row affected (0,00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

Session 1> commit;
Query OK, 0 rows affected (0,01 sec)

Session 1> SELECT * FROM Spectacle WHERE id_spectacle=1;
+--------------+--------------------+------------------+-------+
| id_spectacle | nb_places_offertes | nb_places_libres | tarif |
+--------------+--------------------+------------------+-------+
|            1 |                 50 |               45 | 10.00 |
+--------------+--------------------+------------------+-------+
1 row in set (0,00 sec)

Session 1> SELECT * FROM Client;
+-----------+---------------------+-------+
| id_client | nb_places_reservees | solde |
+-----------+---------------------+-------+
|         1 |                   5 |    50 |
|         2 |                   2 |    40 |
+-----------+---------------------+-------+
```

La base est incohérente ! les clients ont réservé (et payé) en tout 7 places, mais le nombre de places libres n'a diminué que de 5. L'utilisation de InnoDB *ne garantit pas* la correction des exécutions concurrentes, du moins avec le niveau d'isolation par défaut.

Ce point est très souvent ignoré, et source de problèmes récurrents chez les organisations qui croient s'appuyer sur un moteur transactionnel assurant une cohérence totale, et constatent de manière semble-t-il aléatoire l'apparition d'incohérences et de déséquilibres dans leurs bases [3]. On soupçonne le plus souvent les programmes, à tort puisque c'est l'exécution concurrente qui, parfois, est fautive, et pas le programme. Il est extrêmement difficile de comprendre, et donc de corriger, ce type d'erreur.

1.3.5 Le mode `SERIALIZABLE`

Si on analyse attentivement l'exécution concurrente de l'exemple 3, on constate que le problème vient du fait que les deux transactions lisent, chacune de leur côté, une information (le nombre de places libres

3. La remarque est valable pour de nombreux autres SGBD, incluant ORACLE, dont le niveau d'isolation par défaut n'est pas maximal.

pour le spectacles) qu'elles s'apprêtent toutes les deux à modifier. Une fois cette information transférée dans l'espace mémoire de chaque processus, il n'existe plus aucun moyen pour ces transactions de savoir que cette information a changé dans la base, et qu'elles s'appuient donc sur une valeur incorrecte.

La seule chose qui reste à faire pour obtenir une isolation maximale est de s'assurer que cette situation ne se produit pas. C'est ce que garantit le mode `SERIALIZABLE`, au prix d'un risque de blocage plus important. On obtient ce niveau avec la commande suivante :

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

Reprenons une dernière fois l'exemple 3, en mode sérialisable, avec MySQL/InoDB. La session 1 commence par ses lectures.

```
Session 1> SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
Query OK, 0 rows affected (0,04 sec)

Session 1> START TRANSACTION;
Query OK, 0 rows affected (0,00 sec)

Session 1> SELECT * FROM Spectacle WHERE id_spectacle=1;
+--------------+--------------------+------------------+-------+
| id_spectacle | nb_places_offertes | nb_places_libres | tarif |
+--------------+--------------------+------------------+-------+
|            1 |                 50 |               50 | 10.00 |
+--------------+--------------------+------------------+-------+
1 row in set (0,00 sec)

Session 1> SELECT * FROM Client WHERE id_client=1;
+-----------+---------------------+-------+
| id_client | nb_places_reservees | solde |
+-----------+---------------------+-------+
|         1 |                   0 |   100 |
+-----------+---------------------+-------+
```

Voici le tour de la session 2. Elle effectue ses lectures, et cherche à effectuer la première mise à jour.

```
Session 2> SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
Query OK, 0 rows affected (0,00 sec)

Session 2> START TRANSACTION;
Query OK, 0 rows affected (0,00 sec)

Session 2> SELECT * FROM Spectacle WHERE id_spectacle=1;
+--------------+--------------------+------------------+-------+
| id_spectacle | nb_places_offertes | nb_places_libres | tarif |
+--------------+--------------------+------------------+-------+
|            1 |                 50 |               48 | 10.00 |
+--------------+--------------------+------------------+-------+
1 row in set (0,00 sec)

Session 2>
Session 2> SELECT * FROM Client WHERE id_client=2;
+-----------+---------------------+-------+
| id_client | nb_places_reservees | solde |
+-----------+---------------------+-------+
|         2 |                   0 |    60 |
+-----------+---------------------+-------+
1 row in set (0,00 sec)

Session 2> UPDATE Spectacle SET nb_places_libres=48 WHERE id_spectacle=1;
```

La transaction 2 est mise en attente car, en mode sérialisable, MySQL/InnoDB pose un verrou en lecture sur les lignes sélectionnées. La transaction 1 a donc verrouillé, en mode partagé, le spectacle et le client. La transaction 2 a pu lire le spectacle, et placer à son tour un verrou partagé, mais elle ne peut pas le modifier car cela implique la pose d'un verrou exclusif.

Que se passe-t-il alors du côté de la transaction 1 ? Elle cherche à faire la mise à jour du spectacle. Voici la réaction de InnoDB.

```
Session 1> UPDATE Spectacle SET nb_places_libres=45 WHERE id_spectacle=1;
ERROR 1213 (40001): Deadlock found when trying to get lock;
            try restarting transaction
```

Un interblocage (*deadlock*) a été détecté. La transaction 2 était déjà bloquée par la transaction 1. En cherchant à modifier le spectacle, la transaction 1 se trouve bloquée à son tour par les verrous partagées posés par la transaction 2.

En cas d'interblocage, les deux transactions peuvent s'attendre indéfiniment l'une l'autre. Le SGBD prend donc la décision d'annuler par `rollback` l'une des deux (ici, la transaction 1), en l'incitant à recommencer. La transaction 2 est libérée (elle garde ses verrous) et peut poursuivre son exécution.

Le mode `SERIALIZABLE` garantit la correction des exécutions concurrentes, au prix d'un risque de blocage et de rejet de certaines transactions. Ce risque, et ses effets désagréables (il faut resoumettre la transaction rejetée) expliquent qu'il ne s'agit pas du mode d'isolation par défaut. Pour les applications transactionnelles, il vaut sans doute mieux voir certaines transactions rejettées que courir un risque d'anomalie.

Verrouillage d'une ligne avec `FOR UPDATE`

Une alternative au mode `SERIALIZABLE` est la pause explicite de verrous sur les lignes que l'on s'apprête à modifier. La clause `FOR UPDATE` place un verrou exclusif sur les tuples sélectionnés par un ordre `SELECT`. Ces tuples sont donc réservés pour une future modification : aucune autre transaction ne peut placer de verrou en lecture ou en écriture. L'intérêt est de ne pas réserver les tuples qui sont simplement lues et non modifiées ensuite. Notez qu'en mode `SERIALIZABLE` toutes les lignes lues sont réservées, car le SGBD, contrairement au programmeur de l'application, ne peut pas deviner ce qui va se passer ensuite.

Voici l'exécution de l'exemple 3, en veillant à verrouiler les lignes que l'on va modifier.

1. C'est la transaction 1 qui commence. Elle lit le spectacle et le client c_1 en posant un verrou exclusif avec la clause `FOR UPDATE`.
2. Ensuite c'est la seconde transaction qui transmet ses commandes au serveur. Elle aussi cherche à placer des verrous (c'est normal, il s'agit de l'exécution du même code). Bien entendu elle est mise en attente puisque la session 1 a déjà posé un verrou exclusif.
3. La session 1 peut continuer de s'exécuter. Le `commit` libère les verrous, et la transaction 2 peut alors conclure.

Au final les deux transactions se sont exécutées en série. La base est dans un état cohérent. L'utilisation de `FOR UPDATE` est un compromis entre l'isolation assurée par le système, et la déclaration explicite, par le programmeur, des données lues en vue d'être modifiées. Elle assure le maximum de fluidité pour une isolation totale, et minimise le risque d'interblocages. Le principal problème est qu'elle demande une grande discipline pendant l'écriture d'une application puisqu'il faut se poser la question, à chaque requête, des lignes que l'on va ou non modifier.

En résumé, il est de la responsabilité du programmeur, sur un SGBD n'adoptant pas le mode `SERIALISABLE` par défaut, de prendre lui-même les mesures nécessaires pour les transactions qui risquent d'aboutir à des incohérences en cas de concurrence sur les mêmes données. Ces mesures peuvent consister soit à passer en mode `SERIALIZABLE` pour ces transactions, soit à poser explicitement des verrous, en début de transaction, sur les données qui vont être modifiées ensuite.

1.4 Exercices

Exercice 1. *On prend l'hypothèse qu'un système sans gestion de concurrence (une seule version des données, les lectures et écritures sont faites directement sur cette version). Donnez deux exécutions concurrente du programme* RÉSERVATION*, pour le même client et deux spectacles différents, telles qu'on aboutisse à une incohérence de la base.*

Exercice 2. *On crée une table* Compteur *pour attribuer des identifiants uniques*

```
CREATE TABLE Compteur (val INT NOT NULL);
INSERT INTO Compteur VALUE (0);
```

Voici le pseudo-code d'une procédure qui génère et renvoie un nouvel identifiant.

```
Procédure génère_id()
début
  my_val INTEGER;
  SELECT val INTO :my_val FROM Compteur;
  my_val = my_val+1;
  UPDATE Compteur SET val=:my_val;
  RETURN my_val;
fin
```

- *Pour chaque niveau d'isolation de la norme SQL, indiquer si deux processus exécutant indépendamment la procédure* `génère_id()` *peuvent obtenir le même numéro d'identifiant.*
- *En mode* `READ UNCOMMITTED`*, donnez un exemple d'exécution concurrente, comprenant des* `commit` *ou* `rollback`*, aboutissant à un "trou" dans la séquence des valeurs dans la table* Compteur.
- *Cette anomalie est-elle possible en* `READ COMMITTED` *?*
- *Comment modifier la procédure pour assurer que chaque numéro généré est unique, et ce quel que soit le niveau d'isolation ?*
- *Quel problème peut quand même survenir ?*

Exercice 3. *Voici le schéma d'une base de données pour un marchand de voitures.*

- `Modèle (nomModèle, marque, prix)`
- `Véhicule (idVéhicule, nomModèle, couleur)`

Donnez le code (en PL/SQL par exemple), des procédures suivantes. À chaque fois indiquez, si besoin est, l'emplacement des `commit` *ou des* `rollback`*.*

1. *Pour un modèle donné (désigné par son nom), afficher une ligne avec la marque et le prix, puis autant de lignes que de véhicules avec leur couleur.*
2. *Détruire tout ce qui concerne un modèle donné ;*
3. *On ajoute les tables* `Option (nomOption, tarif)` *et* `OptionsVéhicule(idVéhicule, nomOption)`*. On veut tenir compte de ces options pour le prix du véhicule.*
   - *Ajouter un attribut* `prix` *à* Véhicule*, et écrire la procédure qui affecte au prix de chaque véhicule le prix du modèle, plus le cumul du tarif de ses options.*
   - *Définir une vue* `VéhiculePrix(idVéhicule, nomModèle, couleur, prix)`*, où* `prix` *est calculé comme précédemment*

   Discutez des avantages et inconvénients des deux solutions.
4. *On reçoit un véhicule, avec une marque, un modèle, un prix et une couleur : écrire la procédure qui insère ce véhicule, ainsi que le modèle s'il n'existe pas déjà.*

Exercice 4. *Soit le programme suivant qui transfère (certes de manière un peu baroque) un montant d'un compte à un autre, en s'assurant qu'aucun ne descend en dessous de 0 :*

```
Transfert (int id_c1, int id_c2, double montant)
Début
  UPDATE Compte SET solde=solde + :montant
   WHERE id=:id_c2;

  SELECT solde INTO :solde_c1
  FROM Compte WHERE id = :id_c1;

  Si (solde_c1 < montant)
    UPDATE Compte SET solde=solde - :montant
     WHERE id=:id_c2;
  Sinon
    UPDATE Compte SET solde=solde - :montant
     WHERE id=:id_c1;
  Fin
Fin
```

On suppose qu'on a trois comptes : A , B, et C, avec respectivement 100, 200 et 300 Euros. On exécute T_1 pour transférer 150 euros de A à B, et T_2 pour transférer 250 euros de B à C.

Décrire le déroulement de l'exécution concurrente suivante, et le résultat final obtenu, en supposant que le système n'applique pas de contrôle de concurrence (mode `READ UNCOMMITTED`) :

$$w_2[C]w_1[B]r_2[B]r_1[A]w_2[B]w_1[B]$$

Exercice 5. *On considère un programme de copie entre deux comptes. Ce programme est donné de manière simplifiée ci-dessous.*

```
Copie (compte1, compte2)
début
  SELECT solde INTO :v_solde
   FROM Compte WHERE nom='compte1';

  UPDATE Compte SET solde = :v_solde
  WHERE nom='compte2';

  Commit;
fin
```

On dispose de deux comptes A et B et on lance à peu près simultanément deux exécutions $Copie(A, B)$ et $Copie(B, A)$.

1. *Quel état final de la base caractérise le mode sérialisable de ces deux exécutions ?*
2. *On se place en mode `READ COMMITTED` (le système ne pose pas de verrou en lecture). Donnez une exécution concurrente aboutissant à un résultat incorrect.*
3. *On se place en mode `SERIALIZABLE`. Donnez une exécution concurrente aboutissant à un interblocage.*
4. *On se place en mode `READ COMMITTED` et on essaie d'éviter les deux problèmes précédents en ajoutant un verrouillage explicite dans le programme, comme suit (la clause `FOR UPDATE` pose un verrou exclusif).*

   ```
   Copie (compte1, compte2)
   début
     SELECT *  FROM Compte WHERE nom='compte2' FOR UPDATE;
     SELECT solde INTO v_solde FROM Compte WHERE nom='compte1';
     UPDATE Compte SET solde = v_solde WHERE nom='compte2';
     Commit;
   fin
   ```

Évite-t-on une exécution incorrecte ? Évite-t-on les interblocages ?

Exercice 6. *On considère maintenant le problème (délicat) de la réservation des places pour un match. Pour cela on dispose des tables* Stade*(id_stade, nb_places),* Match *(id_match, id_stade, nb_places_prises) et* Client *(id_client, nb_places_réservées). Les opérateurs disposent du petit programme suivant pour réserver des places.*

```
Places (id_client, id_match, nb_places)
début
  Lire le match m                  // Donne le nbre de places prises
  Lire le stade s                  // Donne le nbre total de places

  if ( (s.nb_places - m.nb_places_prises) > nb_places)
         // Il reste assez de places
  début
    Lire le client c
    m.nb_places_prises += nb_places;
    c.nb_places_réservées += nb_places;
    Ecrire le match m
    Ecrire le client c
  fin
  commit
fin
```

Les organisateurs s'aperçoivent rapidement qu'il n'y a pas assez de place dans les stades en entreprennent des travaux d'agrandissement. On a le deuxième programme :

```
Augmenter (id_stade, nb_places)
début
  Lire s
  s.places += nb_places
  Ecrire s
  commit
fin
```

1. *On lance simultanément deux exécutions du programme* Augmenter (SF, 1000) *pour augmenter la capacité du Stade de France et on suppose que le système applique un mode d'isolation* `READ UNCOMMITTED`*.*
   (a) *Donnez un exemple d'une exécution concurrente non sérialisable.*
   (b) *Donnez un exemple d'une exécution concurrente non recouvrable (en plaçant un ordre* `commit`*).*

2. *On a maintenant une exécution concurrente de* Places (C, M, 1500) *et de* Augmenter (SF, 1000), M *étant un match qui se déroule au Stade de France. Voici le début de l'exécution*[4] *:*

   $$\mathbf{H} = r_P[M] r_P[SF] r_A[SF] w_A[SF] c_A \ldots$$

   (a) *Donner la fin de l'exécution concurrente en supposant qu'il y a 2 000 places libres au début de l'exécution. Donnez également le nombre de places libres pour le match à la fin de l'exécution.*
   (b) *Cette exécution concurrente est-elle sérialisable ?*
   (c) *Est-il possible d'obtenir une exécution non sérialisable pour ces deux transactions*
   (d) *On suppose qu'on a ajouté des clauses* `FOR UPDATE` *sur chaque lecture. Le* `FOR UPDATE` *pose un verrou exclusif qui est libéré au moment du* `commit` *ou du* `rollback`*.*
   Donner l'exécution concurrente qui en résulte en reprenant l'ordre des opérations donné ci-dessus. Conclusion, les `FOR UPDATE` *étaient-ils nécessaires ? Quels sont les lectures pour lesquelles ils sont vraiment utiles ?*

4. l'indice P désigne $Places$, A $Augmenter$.

3. *Écrire une procédure qui compare le nombre de places réservées pour un match avec la somme des places des clients ayant réservé (pour simplifier on supposera que tous les clients ont réservé pour le même match). Donner un exemple d'exécution concurrente avec la procédure* `Augmenter`*, en mode* `READ UNCOMMITTED`*, qui fausse le résultat obtenu. Quel est le bon niveau d'isolation ?*

4. *Soit l'exécution concurrente du programme* Places *:* T_1 = Places (C, M, 100) *et* T_2 = Places (C, M, 200) *pour le même match et le même client.*

$$\mathbf{H} = r_1[S]r_2[S]r_1[M]r_1[C]w_1[M]r_2[M]r_2[C]w_2[M]w_1[C]c_1w_2[C]c_2$$

   (a) *Montrer que* **H** *n'est pas sérialisable.*

   (b) *On suppose que le système de gère pas la concurrence (mode* `READ UNCOMMITED`*) et qu'il y a 1 000 places prises dans* $Match$ *au début de l'exécution. Combien y en a-t-il à la fin de l'exécution de* **H** *? Quel est le nombre de places réservées par* C*, en supposant qu'il n'avait rien réservé au départ ?*

   (c) *Choisir le niveau d'isolation approprié pour que cette exécution se déroule correctement.*

   (d) *Appliquer un verrouillage à deux phases sur* **H** *et donner l'exécution* **H'** *résultante. Les verrous sont relâchés après le* `commit`*.*

Chapitre 2

Concurrence et reprise sur panne

Sommaire

Un SGBD doit garantir que l'exécution d'un programme effectuant des mises à jour dans un contexte multi-utisateur s'effectue « correctement ». Bien entendu la signification du « correctement » doit être définie précisément, de même que les techniques assurant cette correction : c'est l'objet du *contrôle de concurrence*.

2.1 Techniques du contrôle de concurrence

Le contrôle de concurrence est la partie de l'activité d'un SGBD qui consiste à ordonner l'exécution des transactions de manière à assurer la sérialisabilité et la recouvrabilité. Nous donnons ci-dessous un aperçu des différentes techniques mises en œuvre, toujours dans l'optique de fournir au programmeur d'applications de bases de données une idée au moins intuitive des mécanismes sous-jacents au déroulement d'une application. Nous commençons donc par décrire les mécanismes assurant la recouvrabilité, avant de passer à la gestion de la sérialisabilité.

2.1.1 Versionnement

Il existe toujours deux choix possibles pour une transaction T en cours : effectuer un `commit` pour valider définitivement les opérations effectuées, ou un `rollback` pour les annuler. Pour que ces choix soient toujours disponibles, le SGBD doit maintenir, pendant l'exécution de T, deux versions des données mises à jour :

1. une version des données *après* la mise à jour ;

2. une version des données *avant* la mise à jour.

Ces deux versions sont stockées dans deux espaces de stockage séparés, que nous désignerons respectivement par les termes d'*image après* et d'*image avant* dans ce qui suit. Conceptuellement (les détails pouvant être assez compliqués), le déroulement d'une transaction T, basé sur ces deux images, s'effectue alors comme suit. Chaque fois que T effectue la mise à jour d'un tuple, la version courante est d'abord copiée dans l'image avant, puis remplacée par la nouvelle valeur fournie par T. Dès lors le `commit` et le `rollback` sont implantés comme suit :

- le `commit` consiste à s'assurer que les données de l'image après sont vraiment écrites sur disque, afin de garantir la durabilité ; l'image avant peut alors être effacée ;
- le `rollback` prend les tuples de l'image avant et les écrit dans l'image après pour ramener la base à l'état initial de la transaction.

Un `rollback` effectué à la suite d'une panne fonctionne de la même manière, sous réserve bien entendu que la panne n'ait pas affecté l'image avant elle-même. Il est donc très important de s'assurer que les données de l'image avant sont régulièrement écrites sur disque. Pour des raisons de performance (répartition des entrées/sorties) et de minimisation des risques, on choisit d'ailleurs quand c'est possible de placer l'image avant et l'image après sur des disques différents.

Quand T effectue des lectures sur des données qu'elles vient de modifier, le système doit lire dans l'image après pour assurer une vision cohérente de la base, reflétant les opérations effectuées par une transaction. En revanche, quand c'est une autre transaction T' qui demande la lecture d'un tuple modifié par T, il faut lire dans l'image avant pour éviter les effets de lectures sales.

On peut se poser la question du nombre de versions nécessaires. Que se passe-t-il par exemple quand T' demande la mise à jour d'un tuple déjà modifié par T ? Si cette mise à jour était autorisée, il s'agirait d'une écriture sale (voir la section précédente) et il faudrait prendre garde à créer une troisième version du tuple, l'image avant de T' étant l'image après de T. La multiplication des versions rendrait la gestion des `commit` et `rollback` extrêmement complexe. En pratique les systèmes n'autorisent pas les écritures sales, s'appuyant pour contrôler cette règle sur des mécanismes de verrouillage exclusif qui seront présentés dans ce qui suit. Il s'ensuit que la gestion de la recouvrabilité ne nécessite pas plus de deux versions pour chaque tuple, une dans l'image avant et l'autre dans l'image après.

Le versionnement n'est pas seulement utile à la gestion de la recouvrabilité. Reprenons le problème des *lectures non répétables* (ou tuples fantômes). Elles sont dues au fait qu'une transaction T lit un tuple t qui a été modifié par une transaction T' *après* le début de T. Or, quand T' modifie t, il existe avant la validation deux versions de t : l'image avant et l'image après.

Il suffirait que T lise toujours l'image avant pour que le problème des tuples fantômes soit résolu. En d'autres termes ces images avant peuvent être vues comme un « cliché » de la base pris à un moment donné, et toute transaction ne lit que dans le cliché « valide » au moment où elle a débuté. De nombreux SGBD (dont ORACLE, PostgreSQL, MySQL/InnoDB) proposent un mécanisme de *lecture cohérente* basé sur ce système de versionnement qui s'appuie sur les principes suivants :

1. chaque transaction T se voit attribuer, quand elle débute, une estampille temporelle e_T ; chaque valeur d'estampille est unique et les valeurs sont croissantes : on garantit ainsi un ordre total entre les transactions.
2. chaque version *validée* d'un tuple a est de même estampillée par le moment e_a de sa validation ;
3. quand T doit lire un tuple a, le système regarde dans l'image après. Si t a été modifié par T ou si son estampille est inférieure à e_T, le tuple peut être lu puisqu'il a été validé avant le début de T, sinon T recherche dans l'image avant la version de t validée et immédiatement antérieure à e_T.

La seule extension nécessaire par rapport à l'algorithme de gestion de la recouvrabilité est la non-destruction des images avant, même quand la transaction qui a modifié le tuple valide par `COMMIT`. L'image avant contient alors toutes les versions successives d'un tuple, marquées par leur estampille temporelle. Seule la plus récente de ces versions correspond à une mise à jour en cours. Les autres ne servent qu'à assurer la cohérence des lectures.

La figure 2.1 illustre le mécanisme de lecture cohérente. La transaction T_{23} a débuté à l'instant 23. Elle a modifié à l'instant 25 l'enregistrement $e3$, et la version précédente (dont l'estampille est 14) de $e3$ a été

placée dans l'image avant. Maintenant, si T_{23} effectue une lecture, celle-ci accède à la version de e_3 placée dans l'image après puisque c'est elle qui l'a modifiée. En revanche une transaction T'_{18} dont le moment de début est 18 lira la version de $e3$ dans l'image avant (il y a un pointeur de l'image avant vers l'image après).

En d'autres termes l'image avant peut être vue comme un « cliché » de la base pris à un moment donné, et toute transaction ne lit que dans le cliché « valide » au moment où elle a débuté.

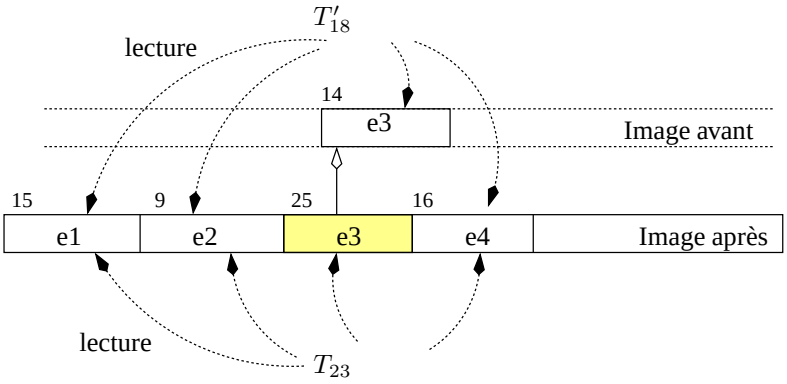


FIGURE 2.1 – Lectures cohérentes avec image avant

L'image avant contient en revanche l'historique de toutes les versions successives d'un enregistrement, marquées par leur estampille temporelle. Seule la plus récente de ces versions correspond à une mise à jour en cours. Les autres ne servent qu'à assurer la cohérence des lectures. Certaines de ces versions n'ont plus aucune chance d'être utilisées : ce sont celles pour lesquelles il existe une version plus récente, antérieure à toutes les requêtes en cours. Cette propriété permet au SGBD d'effectuer un nettoyage (*garbage collection*) des versions devenues inutiles.

2.1.2 Verrouillage

Le mécanisme de versionnement décit ci-dessus est nécessaire mais pas suffisant pour assurer la recouvrabilité et la sérialisabilité des exécutions concurrentes. Le contrôle de concurrence doit parfois effectuer un réordonnancement de l'ordre d'arrivée des opérations pour assurer ces deux propriétés.

La technique la plus fréquente est basée sur le *verrouillage* des tuples lus ou mis à jour. L'idée est simple : chaque transaction désirant lire ou écrire un tuple doit auparavant obtenir un verrou sur ce tuple. 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.

Il existe essentiellement deux types de verrous :

1. les *verrous partagés*, typiquement utilisés pour permettre à plusieurs transactions concurrentes de *lire* le même tuple ;
2. les *verrous exclusifs*, qui interdisent la pose de tout autre verrou, exclusif ou partagé, et donc de toute lecture ou écriture par une autre transaction.

Dans ce qui suit les verrous en lecture seront notés *rl* (comme *read lock*) dans ce qui suit, et ses verrous en écritures seront notés *wl* (comme *write lock*). Donc $rl_i[x]$ indique par exemple que la transaction i a posé un verrou en lecture sur la resource x. On notera de même ru et wu le relâchement des verrous (*read unlock* et *write unlock*).

Il ne peut y avoir qu'un seul verrou exclusif sur un tuple. Son obtention par une transaction T suppose donc qu'il n'y ait aucun verrou déjà posé par une autre transaction T'. En revanche il peut y avoir plusieurs verrous partagés : l'obtention d'un verrou partagé est possible sur un tuple tant que ce tuple n'est pas verrouillé exclusivement par une autre transaction. Enfin, si une transaction est la seule à détenir un verrou partagé sur un tuple, elle peut poser un verrou exclusif.

Si une transaction ne parvient pas à obtenir un verrou, elle est mise en attente, *ce qui signifie que la transaction s'arrête complètement jusqu'à ce que le verrou soit obtenu*. Rappelons qu'une transaction est une séquence d'opérations, et qu'il n'est évidemment pas question de changer l'ordre, ce qui reviendrait à modifier la sémantique du programme. Quand une opération n'obtient pas de verrou, elle est mise en attente ainsi que toutes celles qui la suivent pour la même transaction.

Les verrous sont posés de manière automatique par le SGBD en fonction des opérations effectuées par les transactions/utilisateurs. Il est également possible de demander explicitement le verrouillage de certaines ressources (tuple ou même table) (cf. chapitre d'introduction à la concurrence).

Tous les algorithmes de gestion de la concurrence (voir les deux présentés ci-dessous) utilisent le verrouillage. L'idée général est de bloquer l'accès à une donnée dès qu'elle est lue ou écrite par une transaction (« pose de verrou ») et de libèrer cet accès quand la transaction termine par *commit* ou *rollback* (« libération du verrou »).

En reprenant l'exemple de la réservation, et en supposant que tout accès en lecture ou en écriture pose un verrou bloquant les autres transactions, les transactions T_1 et T_2 s'exécuteront clairement en série et les anomalies disparaîtront. Le bloquage systématique des transactions est cependant une contrainte trop forte, comme le montre l'exemple 2. L'exécution est correcte, mais le verrouillage total bloquerait pourtant sans nécessité la transaction T_2.

Un bon algorithme doit garantir la sérialisabilité et la recouvrabilité au prix d'un minimum de blocages. On peut réduire ce blocage en jouant sur deux critères :

1. le *degré de restriction* (verrou partagé ou verrou exclusif) ;
2. la *granularité* du verrouillage (i.e. le niveau de la *ressource* à laquelle il s'applique : tuple, page, table, etc).

Tous les SGBD proposent un verrouillage au niveau du tuple, et privilégient les verrous partagés tant que cela ne remet pas en cause la correction des exécutions concurrentes. Un verrouillage au niveau du tuple est considéré comme moins pénalisant pour la fluidité, puisqu'il laisse libres d'autres transactions d'accéder à tous les autres tuples non verrouillés. Il existe cependant des cas où cette méthode est inappropriée. Si par exemple un programme parcourt une table avec un curseur pour modifier chaque tuple, et valide à la fin, on va poser un verrou sur chaque alors qu'on aurait obtenu un résultat équivalent avec un seul verrou au niveau de la table. Certains SGBD pratiquent également *l'escalade des verrous* : quand plus d'une certaine fraction des tuples d'une table est verrouillée, le SGBD passe automatiquement à un verrouillage au niveau de la table. Sinon on peut envisager, en tant que programmeur, la pose explicite d'un verrou exclusif sur la table à modifier au début du programme.

2.2 Algorithmes de contrôle de concurrence

Nous présentons dans cette section deux algorithmes de contrôle de concurrence, les plus utilisés en pratique. Ils s'appuient tous deux sur une condition qui permet au SGBD de *décider* si une exécution concurrente est sérialisable ou non. La notion de base est celle de *conflits* entre deux transactions sur la même ressource.

Définition 1. *Deux opérations $p_i[x]$ et $q_j[y]$ sont* en conflit *si $x = y$, $i \neq j$, et p ou q est une écriture.*

On étend facilement cette définition aux exécutions concurrentes : deux transactions dans un exécution sont en *conflit* si elles accèdent à la même donnée et si un de ces accès au moins est une écriture.

Exemple 5. Reprenons une nouvelle fois l'exemple 3, page 20. L'exécution correspond à deux transactions T_1 et T_2, accédant aux données s, c_1 et c_2. Les conflits sont les suivants :

1. $r_1(s)$ et $w_2(s)$ sont en conflit ;
2. $w_2(s)$ et $w_1(s)$ sont en conflit.

Noter que $r_1(s)$ et $r_2(s)$ *ne sont pas* en conflit, puisque ce sont deux lectures. Il n'y a pas de conflit sur c_1 et c_2.

De même, dans l'exécution de l'exemple 4 les conflits sont les suivants :

1. $r_1(c_2)$ et $w_2(c_2)$ sont en conflit ;
2. $w_2(s)$ et $r_1(s)$ sont en conflit.

□

Les conflits permettent de définir un *ordre* sur les transactions d'une exécution concurrente.

Définition 2. *Soit H une exécution concurrente d'un ensemble de transactions $T = \{T_1, T_2, \cdots, T_n\}$. Alors il existe une relation d'ordre partiel $\lhd$ sur cet ensemble, définie par :*

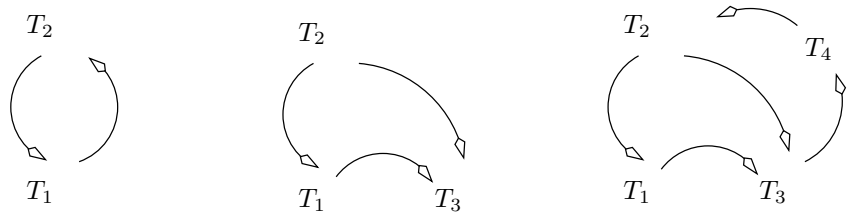
$$T_i \lhd T_j \Leftrightarrow \exists p \in T_i, q \in T_j, p \; est \; en \; conflit \; avec \; q \; et \; p <_H q$$

où $p <_H q$ indique que p apparaît avant q dans H.

Dans les deux exemples qui précèdent, on a donc $T_1 \lhd T_2$, ainsi que $T_2 \lhd T_1$. Dans le cas général, $\lhd$ est un ordre partiel. Une relation T_i peut ne pas être en relation (directe) avec une transaction T_j. La condition sur la sérialisabilité s'exprime sur le graphe de la relation $(T, \lhd)$, dit *graphe de sérialisation* :

Théorème 1. *Soit H une exécution concurrente d'un ensemble de transactions T. Alors H est sérialisable si et seulement si le graphe de $(T, \lhd)$ est acyclique.*

La figure 2.2 montre quelques exemples de graphes de sérialisation. Le premier (2.2.a) correspond aux exemples données ci-dessus : il est clairement cyclique. Le second n'est pas cyclique et correspond donc à une exécution sérialisable. Le troisième (2.2.c) est cyclique.



a. Non sérialisable b. Sérialisable c. Non sérialisable

FIGURE 2.2 – Exemples de graphes de sérialisation

Les algorithmes de contrôle de concurrence ont donc pour objectifs d'éviter la formation d'un cycle dans le graphe de sérialisation. Nous présentons les deux principaux algorithmes dans ce qui suit.

2.2.1 Contrôle par verrouillage à deux phases

Le verrouillage à deux phases est le protocole le plus ancien pour assurer des exécutions concurrentes correctes. Le respect du protocole est assuré par un module dit *scheduler* qui reçoit les opérations émises par les transactions et les traite selon l'algorithme suivant :

1. Le *scheduler* reçoit $p_i[x]$ et consulte le verrou déjà posé sur x, $ql_j[x]$, s'il existe.
   - si $pl_i[x]$ est en conflit avec $ql_j[x]$, $p_i[x]$ est retardée et la transaction T_i est mise en attente.
   - sinon, T_i obtient le verrou $pl_i[x]$ et l'opération $p_i[x]$ est exécutée.
2. les verrous ne sont relâchés qu'au moment du `commit` ou du `rollback`.

Le terme « verrouillage à deux phases » s'explique par le processus détaillé ci-dessus : il y a d'abord *accumulation* de verrous pour une transaction T, puis *libération* des verrous à la fin de la transaction. Les transactions obtenues par application de cet algorithme satisfont les propriétés ACID. Il est assez facile de voir que les lectures ou écritures sales sont interdites, puisque toutes deux reviennent à tenter de lire ou d'écrire un tuple déjà écrit par une autre, et donc verrouillé exclusivement par l'algorithme.

Le protocole garantit que, en présence de deux transactions en conflit T_1 et T_2, la dernière arrivée sera mise en attente de la première ressource conflictuelle et sera bloquée jusqu'à ce que la première commence à relâcher ses verrous (règle 1). À ce moment là il n'y a plus de conflit possible puisque T_1 ne demandera plus de verrou.

Pour illustrer l'intérêt de ces règles, on peut prendre l'exemple des deux transactions suivantes :

1. $T_1 : r_1[x]w_1[y]C_1$
2. $T_2 : w_2[x]w_2[y]C_2$

et l'exécution concurrente :

$$r_1[x]w_2[x]w_2[y]C_2w_1[y]C_1$$

Maintenant supposons que l'exécution avec pose et relâchement de verrous se passe de la manière suivante :

1. T_1 pose un verrou partagé sur x, lit x puis relâche le verrou ;
2. T_2 pose un verrou exclusif sur x, et modifie x ;
3. T_2 pose un verrou exclusif sur y, et modifie y ;
4. T_2 relâche les verrous sur x et y, puis valide ;
5. T_1 pose un verrou exclusif sur y, modifie y, relâche le verrou et valide.

On a violé la règle 3 : T_1 a relâché le verrou sur x puis en a repris un sur y. Une « fenêtre » s'est ouverte qui a permis a T_2 de poser des verrous sur x et y. Conséquence : l'exécution n'est plus sérialisable car T_2 a écrit sur T_1 pour x, et T_1 a écrit sur T_2 pour y. Reprenons le même exemple, avec un verrouillage à deux phases :

1. T_1 pose un verrou partagé sur x, lit x mais ne relâche pas le verrou ;
2. T_2 tente de poser un verrou exclusif sur x : impossible puisque T_1 détient un verrou partagé, *donc T_2 est mise en attente* ;
3. T_1 pose un verrou exclusif sur y, modifie y, et valide ; tous les verrous détenus par T_1 sont relâchés ;
4. T_2 est libérée : elle pose un verrou exclusif sur x, et le modifie ;
5. T_2 pose un verrou exclusif sur y, et modifie y ;
6. T_2 valide, ce qui relâche les verrous sur x et y.

On obtient donc, après réordonnancement, l'exécution suivante, qui est évidemment sérialisable :

$$r_1[x]w_1[y]w_2[x]w_2[y]$$

En général, le verrouillage permet une certaine imbrication des opérations tout en garantissant sérialisabilité et recouvrabilité. Notons cependant qu'il est un peu trop strict dans certains cas : voici l'exemple d'une exécution sérialisable impossible à obtenir avec un verrouillage à deux phases.

$$r_1[x]w_2[x]C_2w_3[y]C_3r_1[y]w_1[z]C_1$$

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. Reprenons notre exemple de base : deux exécutions concurrentes de la procédure RÉSERVATION, désignées par T_1 et T_2, consistant à réserver des places pour le même spectacle, mais pour deux clients distincts c_1 et c_2. L'ordre des opérations reçues par le serveur est le suivant :

$$r_1(s)r_1(c_1)r_2(s)r_2(c_2)w_2(s)w_2(c_2)C_2w_1(s)w_1(c_1)C_1$$

On effectue des lectures pour T_1 puis T_2, ensuite les écritures pour T_2 puis T_1. Cette exécution n'est pas sérialisable, et le verrouillage à deux phases doit empêcher qu'elle se déroule dans cet ordre. Malheureusement il ne peut le faire qu'en rejettant une des deux transactions. Suivons l'algorithme pas à pas :

1. T_1 pose un verrou partagé sur s et lit s ;

2. T_1 pose un verrou partagé sur c_1 et lit c_1 ;
3. T_2 pose un verrou partagé sur s, *ce qui est autorisé car T_1 n'a elle-même qu'un verrou partagé* et lit s ;
4. T_1 pose un verrou partagé sur c_2 et lit c_2 ;
5. T_2 veut poser un verrou exclusif sur s : impossible à cause du verrou partagé de T_1 : donc T_2 est mise en attente ;
6. T_1 veut à son tour poser un verrou exclusif sur s : impossible à cause du verrou partagé de T_2 : donc T_1 est à son tour mise en attente.

T_1 et T_2 sont en attente l'une de l'autre : il y a *interblocage* (*deadlock* en anglais). Cette situation ne peut pas être évitée et doit donc être gérée par le SGBD : en général ce dernier maintient un *graphe d'attente des transactions* et teste l'existence de cycles dans ce graphe. Si c'est le cas, c'est qu'il y a interblocage et une des transactions doit être annulée autoritairement, ce qui est à la fois déconcertant pour un utilisateur non averti, et désagréable puisqu'il faut resoumettre la transaction annulée. Cela reste bien entendu encore préférable à un algorithme qui autoriserait un résultat incorrect.

Notons que le problème vient d'un accès aux mêmes ressources, mais dans un ordre différent : il est donc bon, au moment où l'on écrit des programmes, d'essayer de normaliser l'ordre d'accès aux données.

À titre d'exercice, on peut reprendre le programme de réservation donné initialement, mais dans une version légèrement différente :

Programme RESERVATION2
Entrée : Une séance s
Le nombre de places souhaité $NbPlaces$
Le client c
debut
Lire la séance s
si (nombre de places libres $> NbPlaces$)
Lire le compte du spectateur c
Débiter le compte du client
Soustraire $NbPlaces$ au nombre de places vides
Ecrire le compte du client c
Ecrire la séance s
finsi
fin

Exercice : donner une exécution concurrente de RESERVATION et de RESERVATION2 qui aboutisse à un interblocage.

Dès que 2 transactions lisent la même donnée avec pour objectif d'effectuer une mise à jour ultérieurement, il y a potentiellement interblocage. D'où l'intérêt de pouvoir demander dès la lecture un verrouillage exclusif (écriture). C'est la commande `SELECT ... FOR UPDATE` que l'on trouve dans certains SGBD.

2.2.2 Contrôle de concurrence multi-versions

Les systèmes qui s'appuient sur des lectures cohérentes et gèrent donc des bases multi-versions, peuvent tirer parti du fait que les lectures s'appuient toujours sur une version cohérente (le « cliché ») de la base. Tout se passe comme si les lectures effectuées par une transaction $T(t_0)$ débutant à l'instant t_0 lisaient la base, dès le début de la transaction, donc dans l'état t_0.

Cette remarque réduit considérablement les cas possibles de conflits, comme le montre le raisonnement suivant. Prenons une lecture $r_1[d]$ effectuée par la transaction $T_1(t_0)$. Cette lecture accède à la version *validée* la plus récente de d qui précède t_0, par définition de l'état de la base à t_0. Deux cas de conflits sont envisageables :

1. $r_1[d]$ est en conflit avec une écriture $w_2[d]$ qui a eu lieu *avant* t_0 ;
2. $r_1[d]$ est en conflit avec une écriture $w_2[d]$ qui a eu lieu *après* t_0.

Dans le premier cas, T_2 a forcément effectué son `commit` avant t_0, puisque T_1 lit l'état de la base à t_0 : tous les conflits de T_1 avec T_2 sont dans le même sens, et il n'y a pas de risque de cycle (Figure 2.3).

Le second cas est celui qui peut poser problème. Si T_1 cherche à écrire d après l'écriture $w_2[d]$, alors un conflit cyclique apparaît (Figure 2.3). Notez qu'une nouvelle lecture de d par T_1 n'introduit pas de cycle puisque toute lecture s'effectue à t_0.

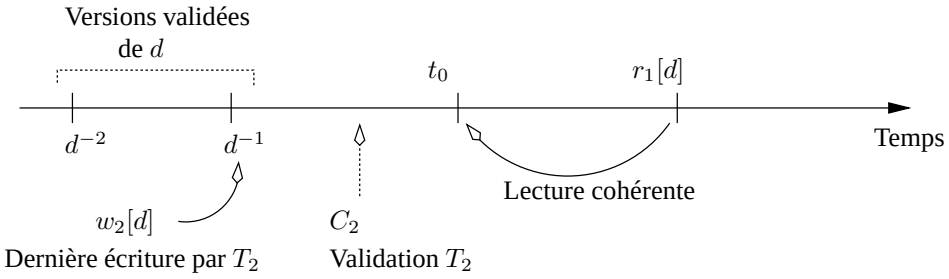


FIGURE 2.3 – Contrôle de concurrence multi-versions : conflit avec les écritures précédentes

En résumé le contrôle de concurrence peut alors se limiter à vérifier, au moment de l'écriture d'un tuple a par une transaction T, qu'aucune transaction T' n'a modifié a entre le début de T et l'instant présent. Si on autorisait la modification de a par T, des risques de conflits cycliques apparaîtraient avec T'. Autrement dit une mise à jour n'est possible que si la partie de la base à modifier n'a pas changé depuis que T a commencé à s'exécuter.

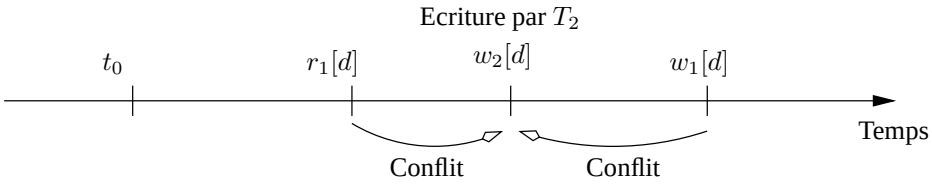


FIGURE 2.4 – Contrôle de concurrence multi-versions : conflit avec une écriture d'une autre transaction.

Le contrôle de concurrence multi-versions s'appuie sur les règles suivantes. Rappelons que pour chaque transaction T on connaît son estampille temporelle de début d'exécution e_T ; et pour chaque version d'un tuple a son estampille de validation e_a.

1. toute lecture $r_T[a]$ lit la plus récente version de a telle que $e_a \leq e_T$; aucun verrou n'est posé ;
2. en cas d'écriture $w_T[a]$,
   (a) si $e_a \leq e_T$ et aucun verrou n'est posé sur a : T pose un verrou exclusif sur a, et effectue la mise à jour ;
   (b) si $e_a \leq e_T$ et un verrou n'est posé sur a : T est mise en attente ;
   (c) si $e_a > e_T$, T est rejetée.
3. au moment du `commit` d'une transaction T, tous les enregistrements modifiés par T obtiennent une nouvelle version avec pour estampille l'instant du `commit`.

Avec cette technique, on peut ne pas poser de verrou en lecture. En revanche les verrous exclusifs sont toujours indispensables pour les écritures, afin d'éviter lectures ou écritures sales.

Voici un déroulé de cette technique, toujours sur notre exemple d'une exécution concurrente du programme de réservation avec l'ordre suivant :

$$r_1(s)r_1(c_1)r_2(s)r_2(c_2)w_2(s)w_2(c_2)C_2w_1(s)w_1(c_1)C_1$$

On suppose que $e_{T_1} = 100$, $e_{T_2} = 120$. On va supposer qu'une opération est effectuée toutes les 10 unités de temps, même si seul l'ordre compte, et pas le délai entre deux opérations. Le déroulement de l'exécution est donc le suivant :

1. T_1 lit s, sans verrouiller ;
2. T_1 lit c_1, sans verrouiller ;
3. T_2 lit s, sans verrouiller ;
4. T_2 lit c_2, sans verrouiller ;
5. T_2 veut modifier s : l'estampille de s est inférieure à $e_{T_2} = 120$, ce qui signifie que s n'a pas été modifié par une autre transaction depuis que T_2 a commencé à s'exécuter ; on pose un verrou exclusif sur s et on effectue la modification :
6. T_2 modifie c_2, avec pose d'un verrou exclusif ;
7. T_2 valide et relâche les verrous ; deux nouvelles versions de s et c_2 sont créées avec l'estampille 150 ;
8. T_1 veut à son tour modifier s, mais cette fois le contrôleur détecte qu'il existe une version de s avec $e_s > e_{T_1}$, donc que s a été modifié après le début de T_1. Le contrôleur doit donc rejeter T_1 sous peine d'obtenir une exécution non sérialisable.

Comme dans le verrouillage à deux phases, on obtient le rejet de l'une des deux transactions, mais le SGBD effectue dans ce cas un contrôle *à postériori* au lieu d'effectuer un blocage *à priori* comme le verrouillage à deux phases. On parle parfois d'approche « pessimiste » pour le verrouillage à deux phases et « optimiste » pour le contrôle multi-versions, exprimant ainsi l'idée que la première technique tente de prévenir les problèmes, alors que la seconde choisit de laisser faire et d'intervenir seulement quand ils surviennent réellement.

L'absence de verrouillage en lecture favorise la fluidité des exécutions concurrentes par rapport au verrouillage à deux phases, et limite le coût de la pose de verrous. On peut le vérifier par exemple sur l'exécution concurrente de l'exemple 4, page 20. Bien entendu le coût en contrepartie est la nécessité de gérer les versions et de maintenir une vue cohérente de la base pour chaque transaction.

2.3 Reprise sur panne

La reprise sur 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, ou d'une défaillance logicielle. Les pannes affectant les disques sont plus graves, mais nous verrons cependant qu'avec une stratégie appropriée de sauvegarde, il est relativement facile de garantir que l'état de la base peut être récupéré même en cas de panne de l'un des disques.

On définit *l'état de la base à un instant t* comme *c'est l'état résultant de l'ensemble des transactions validées à l'instant t*. La problématique de la reprise sur panne est donc 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.

Pour bien comprendre les mécanismes utilisés, il faut bien avoir en tête l'architecture générale d'un serveur de données en cours de fonctionnement, et se souvenir que la performance d'un système est fortement liée au nombre de lectures/écritures qui doivent être effectuées. La reprise sur panne, comme les autres techniques mises en œuvre dans un SGBD, vise à minimiser ces entrées/sorties. La première section discute de l'impact de l'architecture sur les techniques de reprise sur panne.

Ces techniques sont ensuite développées dans les sections suivantes.

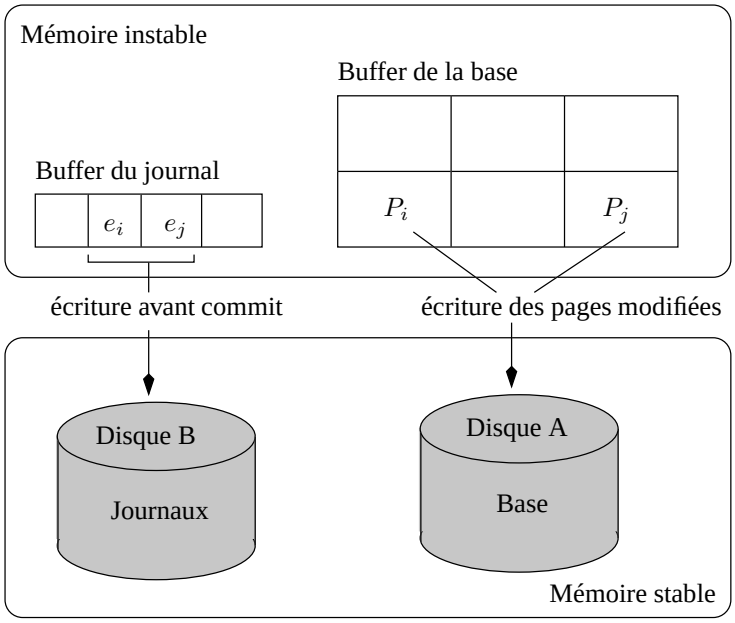


FIGURE 2.5 – Architecture d'un SGBD et reprise sur panne

2.3.1 Rappels sur le fonctionnement des caches

La figure 2.5 illustre les composants d'un SGBD qui interviennent dans la reprise sur panne. On distingue la *mémoire stable* (les disques) qui survit à une panne « légère » de type électrique ou logicielle, et la *mémoire instable* qui est irrémédiablement perdue en cas, par exemple, de panne électrique. Pour assurer la reprise sur panne, une première règle simple est donc :

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

Ce qui revient à dire que toute donnée modifiée par une transacion qui valide par `commit` doit être écrite sur disque *avant* l'acquitement de l'opération de `commit`. Or, pour des raisons de performance, le serveur de données cherche à limiter les accès aux disques, et s'appuie sur un *cache* qui stocke, en mémoire principale (donc instable) les blocs de données provenant des fichiers stockés sur disque. Que ce soit en lecture ou en écriture, le serveur va chercher à s'appuyer sur le *cache*.

Pour les écritures (cas qui nous intéresse ici), le serveur recherche dans le cache la donnée [1] à modifier. Si elle est présente la modification a lieu en mémoire. Sinon le bloc contenant la donnée est chargé du disque vers le cache, ce qui ramène au cas précédent. *Dans tous les cas, la modification dans le cache n'entraîne pas une écriture sur le disque*. Une écriture systématique après une modification entraînerait des perfomances désastreuses. Heureusement ce n'est pas nécessaire, car si une panne survient avant l'écriture sur disque, la donnée perdue est une donnée non validée par un `commit`, et elle ne fait donc pas partie de l'état de la base.

Un bloc placé dans le cache et non modifié est l'image exacte du bloc correspondant sur le disque. Quand une transaction vient modifier un enregistrement dans un bloc, son image en mémoire (l'image *après*) devient différente de celle sur le disque (l'image *avant*). La figure 2.5 illustre la situation pour deux enregistrements modifiés, e_i et e_j, situés respectivement dans les pages P_i et P_j.

Quand un enregistrement est modifié dans le cache, le bloc qui le contient est marqué. Le système tient alors compte de cette marque. En particulier, si le cache est plein et que de l'espace doit être libéré dans le cache, un bloc qui n'est pas marqué comme étant modifié est simplement supprimé du cache, alors qu'un bloc modifié doit être écrit sur le disque. Si le système applique une stratégie classique de remplacement

1. Comme pour les transactions, le terme de « donnée » peut correspondre à plusieurs granularités. On suppose ici qu'il s'agit d'un enregistrement.

des blocs du cache (par exemple la stratégie dite *LRU*, pour *Least Recently Used*), un bloc modifié sera tôt ou tard la cible d'un remplacement, et les modifications qu'il contient seront écrites sur le disque. Une autre possibilité (peu employée) consiste à « épingler » (*pin* en anglais) une page dans le cache pour interdire son remplacement et donc son écriture. Cette stratégie paresseuse vise à n'effectuer une écriture que quand elle devient nécessaire. Avec un cache suffisamment, la partie de la base la plus utilisée n'est en fait jamais écrite sur le disque tant que le serveur n'est pas arrêté. Toutes les lectures et écritures ont lieu en mémoire.

Si on veut concilier à la fois de bonnes performances par limitation des entrées/sorties et la garantie de reprise sur panne, on réalise rapidement que le problème est plus compliqué qu'il n'y paraît. Voici par exemple un premier algorithme, simpliste, qui ne fonctionne pas. L'idée est d'utiliser le cache pour les données modifiées (donc « l'image après », cf. le chapitre sur les transactions), et le disque pour les données validées (« l'image avant »).
Algorithme simpliste :

1. ne jamais écrire une donnée modifiée par une transaction T avant que le `commit` n'arrive,
2. au moment du `commit` de T, forcer l'écriture de tous les blocs modifiés par T.

Pourquoi cela ne marche-t-il pas ? Pour des raisons de performance *et* des raisons de correction (la reprise n'est pas garantie).

Surcharge du cache. Si on interdit l'écriture des blocs modifiés, qui peut dire que le cache ne va pas, au bout d'un certain temps, contenir uniquement des blocs modifiés et donc « épinglés » en mémoire Aucune remplacement ne devient alors possible, et le système est bloqué. Entretemps il est probable que l'on aura assisté à une lente diminution des performances due à la réduction de la capacité effective du cache.

Écritures aléatoires. Si on décide, au moment du `commit`, d'écrire tous les blocs modifiés, on risque de déclencher des écritures, à des emplacements éloignés, de blocs donc seule une petite partie est modifiée. Or un principe essentiel de la performance d'un SGBD est de privilégier les *écritures séquentielles de blocs pleins*.

Risque sur la durabilité. Que faire si une panne survient *après* des écritures mais *avant* l'enregistrement du `commit` ?

Dans le dernier cas, on ne peut simplement plus assurer une reprise. Donc cette solution est inefficace et incorrecte. On en conclut que les fichiers de la base ne peuvent pas, à eux seuls, servir de support à la reprise sur panne car on est dans l'incapacité d'assurer, à chaque instant, que *l'état de la base* est l'état des *fichiers* de la base. Nous avons besoin d'une structure auxiliaire, le journal des transactions.

2.3.2 Le journal des transactions

Un journal des transactions (*log* en anglais) est un ensemble de fichiers complémentaires à ceux de la base de données, servant à stocker sur un support non volatile les informations nécessaires à la reprise sur panne. L'idée de base est exprimée par l'équation suivante :

état de la base = journaux de transactions + fichiers de la base

Le journal contient les types d'enregistrements suivants :

1. `start(T)`
2. `write(T, x, old_val, new_val)`
3. `commit`
4. `rollback`
5. `checkpoint`

L'enregistrement dans le journal des opérations de lectures n'est pas nécessaire, sauf pour de l'audit éventuellement. Le journal est un fichier *séquentiel*, avec un cache dédié, qui fonctionne selon la technique classique. Quand le cache est plein, on écrit dans le fichier et on vide le cache. Les écritures sont séquentielles et maximisent la rentabilité des entrées/sorties. On doit écrire dans le journal (physiquement) à deux occasions.

Règle du point de commit. Au moment d'un `commit` le cache du journal doit être écrit sur le disque (écriture forcée). On satisfait donc l'équation : l'état de la base est sur le disque au moment où l'enregistrement `commit` est écrit dans le fichier journal.

Règle de recouvrabilité. Si un bloc du fichier de données, marqué comme modifié mais non validé, est écrit sur le disque, il va écraser l'image avant. Le risque est alors de ne plus respecter l'équation, et il faut donc écrire dans le journal pour être en mesure d'effectuer un rollback éventuel.

La figure 2.5 explique ce choix qui peut sembler inutilement complexe. Elle montre la structure des mémoires impliquées dans la gestion du journal des transactions. Nous avons donc sur mémoire stable (c'est-à-dire non volatile, résistante aux coupures électriques) les fichiers de la base d'une part, le fichier journal de l'autre. Si possible ces fichiers sont sur des disques différents. En mémoire centrale nous avons un *cache* principal stockant une image partielle des fichiers de la base, et un *cache* pour le fichier journal. Une donnée modifiée et validée est *toujours* dans le fichier journal. Elle peut être dans les fichiers de la base, mais seulement une fois que le bloc modifié est écrit, ce qui finit toujours par arriver sur la durée du fonctionnement normal d'un système. Si tout allait toujours bien (pas de panne, pas de `rollback`), on n'aurait jamais besoin du journal.

2.3.3 Que faire en cas de panne ?

Si une panne légère (pas de perte de disque) survient, il faut effectuer deux types d'opérations :

1. *refaire* (REDO) les transactions validées avant la panne qui ne seraient par correctement écrites dans les fichiers de la base ;
2. *défaire* (UNDO) les transactions en cours au moment de la panne, qui avaient déjà effectué des mises à jour dans les fichiers de la base.

Ces deux opérations sont basées sur le journal. On doit faire un REDO pour les transactions validées (celle pour lesquelles on trouve un `commit` dans le journal) et un UNDO pour les transactions actives (celles qui n'ont ni `commit`, ni `rollback` dans le journal).

Qu'est-ce qu'un « checkpoint »

En cas de panne, il faudrait en principe refaire toutes les transactions du journal, *depuis l'origine de la création de la base*, et défaire celles qui étaient en cours. Au moment d'un *checkpoint*, le SGBD écrit sur disque tous les blocs modifiés, ce qui garantit que les données validées par `commit` sont dans la base. Il devient inutile de faire un REDO pour les transactions validées avant le checkpoint.

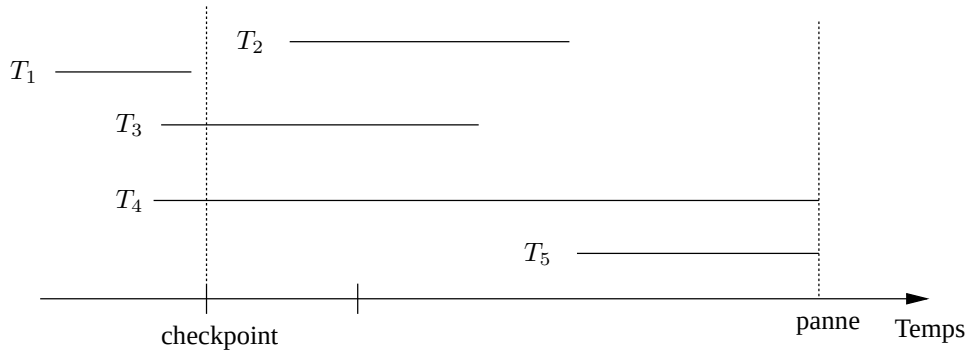


FIGURE 2.6 – Reprise sur panne après un *checkpoint*

La figure 2.6 montre un exemple, avec un *checkpoint* survenant après la validation de T_1. Toutes les mises à jour de T_1 ont été écrites dans les fichiers de la base au moment du checkpoint, et il est donc inutile d'effectuer un REDO pour cette transaction. Pour toutes les autres le REDO est indispensable. Les mises à

jour de T_2 par exemple, bien que validées, peuvent rester dans le *cache* sans être écrites dans les fichiers de la base au moment de la panne. Elles sont alors perdues et n'existent que dans le journal.

Un checkpoint prend du temps : il faut écrire sur le disque tous les blocs modifiés. Sa fréquence est généralement paramétrable par l'administrateur. Il est indispensable de maîtriser la taille des journaux de transactions qui, en l'absence de mesures de maintenance (comme le *checkpoint*) ne font que grossir et peuvent atteindre des volumes considérables.

2.3.4 Algorithmes avec mise à jour différée

Voici maintenant un premier algorithme correct, qui s'appuie sur l'interdiction de toute écriture d'un bloc contenant des mises à jour non validées. L'intérêt est d'éviter d'avoir à effectuer des UNDO à partir du journal. L'inconvénient est de devoir « épingler » des pages en mémoire, avec un risque de se retrouver à court d'espace disponible.

Au moment d'un `commit`, on écrit d'abord dans le journal, puis on retire l'épingle des données du cache pour qu'elles soient écrites. Il n'y a jamais besoin de faire un UNDO. C'est un algorithme NO-UNDO/REDO :

1. on constitue la liste des transactions validées depuis le dernier *checkpoint* ;
2. on prend les entrées `write` de ces transactions *dans l'ordre de leur exécution*, et on s'assure que chaque donnée x a bien la valeur `new_val`.

On peut aussi refaire les opérations *dans l'ordre inverse*, en s'assurant qu'on ne refait que la dernière mise à jour validée.

L'opération de REDO est *idempotente* : on peut la réexécuter autant de fois qu'on veut sans changer le résultat de la première exécution. C'est une propriété nécessaire, car la reprise sur panne elle-même peut échouer !

2.3.5 Algorithmes avec mise à jour immédiate

Dans ce second algorithme (de loin le plus répandu), on autorise l'écriture de blocs modifiés. Dans ce cas il faut *défaire* les mises à jours de transactions annulées. Il existe deux variantes :

1. Avant un commit, on force les écritures dans la base : il n'y a jamais jamais besoin de faire un REDO (UNDO/NO-REDO)
2. Si on ne force pas les écritures dans la base, c'est un algorithme UNDO/REDO, le plus souvent rencontré car il évite le flot d'écriture aléatoires à déclencher sur chaque `commit`.

L'algorithme se décrit simplement comme suit :

– on constitue la liste des transactions actives L_A et la liste des transactions validées L_V au moment de la panne ;
– on annule les écritures de L_A avec le journal : attention les annulations se font dans l'ordre inverse de l'exécution initiale ;
– on refait les écritures de L_V avec le journal.

Notez qu'avec cette technique on ne force jamais l'écriture des données modifiées (sauf aux *checkpoints*) donc on attend qu'un *flush* (mise sur disque des blocs modifiées) intervienne naturellement pour qu'elles soient placées sur le disque.

2.3.6 Journaux et sauvegardes

Le journal peut également servir à la reprise en cas de perte d'un disque. Il est cependant essentiel d'utiliser deux disques séparés. Les sauvegardes binaires (les fichiers de la base), associées aux journaux des mises à jour, vérifient en effet l'équation suivante :

état de la base = sauvegarde binaire + journaux des mises à jour

En ré-exécutant ces modifications à partir d'une sauvegarde, on récupère l'état de la base au moment de la panne d'un disque. Deux cas se présentent : panne du disque contenant le journal (appelons-le D_l) et panne du disque contenant les fichiers de la base (appelons-le D_b).

Panne du disque du *log*

Si D_l tombe en panne, l'état de la base peut être reconstitué en effectuant toutes les écritures des blocs modifiés du *cache*. On obtient alors des fichiers sur le disque D_b contenant toutes les mises à jour. Le bon reflexe est donc d'arrêter proprement le système, ou d'utiliser une commande *flush* si elle existe. Il n'y a plus ensuite qu'à effectuer des sauvergardes et les réparations matérielles nécessaires.

Panne du disque contenant les fichiers de la base

Le cas est un peu plus délicat. Il faut en fait effectuer une reprise sur panne à partir des journaux, en appliquant les REDO et UNDO à la dernière sauvegarde disponible.

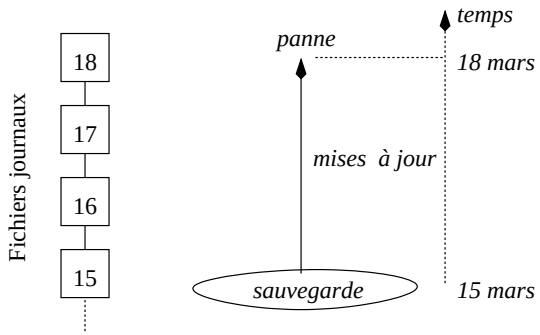


FIGURE 2.7 – Reprise à froid avec une sauvegarde et des fichiers *log*

La figure 2.7 montre une situation classique, avec un sauvegarde effectuée le 15 mars, des fichiers journaux avec un *checkpoint* quotidien, chaque *checkpoint* entraînant la création d'un fichier physique supplémentaire. En théorie seul le dernier fichier journal est utile (puisque seules les opérations depuis le dernier *checkpoint* doit être refaites). C'est vrai seulement pour des reprises à chaud, après coupure de courant. En cas de perte d'une disque tous les fichiers journaux depuis la dernière sauvegarde sont nécessaires.

Il faut donc que l'administrateur réinstalle un disque neuf et y place la sauvegarde du 15 mars. Il demande ensuite au système une reprise sur panne depuis le 15 mars, en s'assurant que les fichiers journaux sont bien disponibles depuis cette date. Sinon l'état de la base au 18 mars ne peut être récupéré, et il faut repartir de la sauvegarde.

On réalise l'importance des journaux et de leur rôle pour le maintien des données. Un soin tout particulier (sauvegardes fréquentes, disques en miroir) doit être consacré à ces fichiers sur une base « sensible ». Autant la reprise peut s'effectuer automatiquement après une panne légère, de type coupure d'électricité, autant elle demande des interventions de l'administrateur, parfois délicates, en cas de perte d'un disque. On parle respectivement de *reprise à chaud* et de *reprise à froid*. Bien entendu les procédures de reprise doivent être testées et validées *avant* qu'un vrai problème survienne, sinon on est sûr de faire face dans la panique à des difficultés imprévues.

2.4 Exercices

Exercice 7. *Soit T_1 et T_2 deux transactions et x, y deux items de la base. L'ordonnancement suivant est-il sérialisable ? Est-il accepté par un verrouillage en deux phases ?*

Exercice 8 (Verrouillage à 2 phases)**.** *Un contrôleur avec verrouillage à 2 phases reçoit la séquence d'opérations ci-dessous.*

$$r_1[x]\ r_2[y]\ w_3[x]\ w_1[y]\ w_1[x]\ w_2[y]\ c_2\ r_3[y]\ r_1[y]\ c_1\ w_3[y]\ c_3$$

Opération	T_1	T_2
(1)	$r_1(x)$	
(2)	$w_1(x)$	
(3)		$r_2(x)$
(4)	$w_1(y)$	
(5)	$r_1(y)$	
(6)		$r2(y)$

Indiquez l'ordre d'exécution établi par le contrôleur, en considérant qu'une opération bloquée en attente d'un verrou est exécutée en priorité dès que le verrou devient disponible. On suppose que les verrous d'une transaction sont relâchés au moment du `Commit`.

Exercice 9 (Graphe de sérialisabilité et équivalence des exécutions)**.** *Construisez les graphes de sérialisabilité pour les exécutions (histoires) suivantes. Indiquez les exécutions sérialisables et vérifiez s'il y a des exécutions équivalentes.*

1. $H_1 : w_2[x]\ w_3[z]\ w_2[y]\ c_2\ r_1[x]\ w_1[z]\ c_1\ r_3[y]\ c_3$
2. $H_2 : r_1[x]\ w_2[y]\ r_3[y]\ w_3[z]\ c_3\ w_1[z]\ c_1\ w_2[x]\ c_2$
3. $H_3 : w_3[z]\ w_1[z]\ w_2[y]\ w_2[x]\ c_2\ r_3[y]\ c_3\ r_1[x]\ c_1$

Exercice 10 (Recouvrabilité)**.** *Parmi les exécutions concurrentes suivantes, lesquelles sont recouvrables (impossibilité d'assurer correctement un* `commit` *ou un* `rollback`*), lesquelles évitent les annulations en cascade (l'annulation d'une transaction entraîne l'annulation d'une ou plusieurs autres) ?*

Indiquez s'il y a des exécutions sérialisables.

1. $H_1 : r_1[x]\ w_2[y]\ r_1[y]\ w_1[x]\ c_1\ r_2[x]\ w_2[x]\ c_2$
2. $H_2 : r_1[x]\ w_1[y]\ r_2[y]\ c_1\ w_2[x]\ c_2$
3. $H_3 : r_1[y]\ w_2[x]\ r_2[y]\ w_1[x]\ c_2\ r_1[x]\ c_1$

Exercice 11. *Le programme suivant s'exécute dans un système de gestion de commandes pour les produits d'une entreprise. Il permet de commander une quantité donnée d'un produit qui se trouve en stock. Les paramètres du programme représentent respectivement la référence de la commande (c), la référence du produit (p) et la quantité commandée (q).*

```
Commander (c, p, q)
début
  Lecture prix produit p
  Lecture du stock de p
  si (q > stock de p) alors
    rollback
  sinon
     Mise à jour stock de p;
     Enregistrement dans c du total de facturation
     commit
  fin si
fin
```

Notez que le prix et la quantité de produit en stock sont gardés dans des enregistrements différents.

1. *Lesquelles des transactions suivantes peuvent être obtenues par l'exécution du programme ci-dessus ? Justifiez votre réponse.*

(a) $T_1 : r[x]r[y]R$

(b) $T_2 : r[x]R$

(c) $T_3 : r[x]w[y]w[z]C$

(d) $T_4 : r[x]r[y]w[y]w[z]C$

2. *Dans le système s'exécutent en même temps trois transactions : deux commandes d'un même produit et le changement du prix de ce même produit. Montrez que l'histoire ci-dessous est une exécution concurrente de ces trois transactions et expliquez la signification des enregistrements qui y interviennent.*
   H *:* $r_1[x]$ $r_1[y]$ $w_2[x]$ $w_1[y]$ c_2 $r_3[x]$ $r_3[y]$ $w_1[z]$ c_1 $w_3[y]$ $w_3[u]$ c_3
3. *Vérifiez si* **H** *est sérialisable en identifiant les conflits et en construisant le graphe de sérialisation.*
4. *Quelle est l'exécution obtenue par verrouillage à deux phases à partir de* **H** *? Quel prix sera appliqué pour la seconde commande, le même que pour la première ou le prix modifié ?*
   On considère que le relâchement des verrous d'une transaction se fait au Commit *et qu'à ce moment on exécute en priorité les opérations bloquées en attente de verrou, dans l'ordre de leur blocage.*

Exercice 12 (Concurrence : Gestion Bancaire)**.** *Les trois programmes suivants peuvent s'exécuter dans un système de gestion bancaire.* Débit *diminue le solde d'un compte* `c` *avec un montant donné* `m`*. Pour simplifier, tout débit est permis (on accepte des découverts).* Crédit *augmente le solde d'un compte* `c` *avec un montant donné* `m`*.* Transfert *transfère un montant* `m` *à partir d'un compte source* `s` *vers un compte destination* `d`*. L'exécution de chaque programme démarre par un* **Start** *et se termine par un* **Commit** *(non montrés ci-dessous).*

```
Débit (c:Compte;   |  Crédit (c:Compte;   |  Transfert (s,d:Compte;
       m:Montant)  |          m:Montant)  |              m:Montant)
begin              |  begin               |  begin
  t := Read(c);    |    t := Read(c);     |    Débit(s,m);
  Write(c,t-m);    |    Write(c,t+m);     |    Crédit(d,m);
end                |  end                 |  end
```

Le système exécute en même temps les trois opérations suivantes :
(1) un transfert de montant 100 du compte A vers le compte B
(2) un crédit de 200 pour le compte A
(3) un débit de 50 pour le compte B

1. *Écrire les transactions* T_1*,* T_2 *et* T_3 *qui correspondent à ces opérations. Montrer que l'histoire* **H** *:* $r_1[A]$ $r_3[B]$ $w_1[A]$ $r_2[A]$ $w_3[B]$ $r_1[B]$ c_3 $w_2[A]$ c_2 $w_1[B]$ c_1 *est une exécution concurrente de* T_1*,* T_2 *et* T_3*.*

2. *Mettre en évidence les conflits dans* **H** *et construire le graphe de sérialisation de cette histoire.* **H** *est-elle sérialisable ?* **H** *est-elle recouvrable ?*

3. *Quelle est l'exécution* **H'** *obtenue à partir de* **H** *par verrouillage à deux phases ? On suppose que les verrous d'une transaction sont relâchés après le Commit de celle-ci. Une opération bloquée en attente d'un verrou bloque le reste de sa transaction. Au moment du relâchement des verrous, les opérations en attente sont exécutées en priorité.*
   Si au début le compte A avait un solde de 100 et B de 50, quel sera le solde des deux comptes après la reprise si une panne intervient après l'exécution de $w_1[B]$ *?*

Exercice 13. *Dans un contrôleur multi-versions, on met en attente une transaction* T *qui essaye d'effectuer une mise à jour sur un tuple* a *qui a verrouillé en écriture. Pourquoi ne pas rejeter directement cette transaction ? Donner un cas où cette transaction finit par s'exécuter.*

Exercice 14. *Soit* T_1*,* T_2*,* T_3 *et* T_4 *quatre transactions et* x*,* y*,* z *trois enregistrements. On considère l'ordonnancement suivant :*

Instant	T_1	T_2	T_3	T_4
(15)		$r_2[x]$		
(16)			$r_3[x]$	
(17)		$w_2[y]$		
(18)			$w_3[x]$	
(19)		$r_2[z]$		
(20)	$r_1[y]$			
(21)				$r_4[y]$
(22)	$r_1[x]$			
(23)	$w_1[z]$			
(24)				$w_4[x]$

1. *Cet ordonnancement est-il sérialisable ? Si oui, donner l'ordre séquentiel des transactions équivalent.*

2. *Donnez l'ordonnancement obtenu par un verrouillage à deux phases ?*

3. *On suppose qu'avant le début des transactions les estampilles de lecture et d'écriture de x, y et z sont toutes égales à 10. Les opérations s'exécutent dans l'ordre indiqué, l'instant de la demande d'exécution étant indiqué en première colonne.*
   Appliquer l'algorithme de contrôle multi-version sur cette exécution concurrente. Que se passe-t-il si T_3 choisit finalement de valider ? Et si T_3 choisit d'annuler ?

Exercice 15. *L'exécution suivante est reçue par un SGBD :*

$$H : r_1[x]r_2[z]r_1[y]w_1[x]r_3[x]r_2[y]w_2[z]w_2[y]c_2r_3[y]r_3[z]c_3w_1[y]c_1$$

1. *Parmi les transactions T_1, T_2, T_3, certaines correspondent au virement d'un compte vers un autre : on lit les valeurs des deux comptes, puis on les modifie. De quelles transactions s'agit-il ?*

2. *Existe-t-il des "lectures sales" dans H ? Indiquez-les, ainsi que les conséquences possibles.*
3. *Vérifiez si H est sérialisable en identifiant les conflits et en construisant le graphe de sérialisation.*

4. *Qu'obtient-on en appliquant un verrouillage à deux phases à partir de H (indiquer le déroulement de l'exécution, et les blocages éventuels) ?*

5. *On suppose maintenant que toutes les transactions placent un* `FOR UPDATE` *pour toute lecture d'une donnée* qui va être modifiée ensuite. *Qu'obient-on si on applique à H une variante du verrouillage à deux phases qui pose un verrou exclusif (au lieu d'un verrou partagé) pour les lectures comprenant la clause* `FOR UPDATE` *?*

6. *Quelle est l'exécution obtenue par l'algorithme de contrôle de concurrence avec versionnement ? Indiquez la (ou les) transaction(s) rejetée(s) et expliquez pourquoi. Quel danger a-t-on évité ?*
   On suppose que toutes les transactions débutent au même moment.

Exercice 16. *Soit le fichier journal suivant (les écritures les plus anciennes sont en haut).*

```
start(T1)
write (T1, x, 10, 20)
commit(T1)
checkpoint
start(T2)
write(T2, y, 5, 10)
```

```
start(T4)
write(T4, x, 20, 40)
start(T3)
write(T3, z, 15, 30)
write(T4, u, 100, 101)
commit(T4)
write (T2, x, 40, 60)
..................... panne!
```

1. *Indiquer la reprise sur panne avec l'algorithme* UNDO/REDO
2. *Donner le comportement de la variante avec mise à jour différée (*NO-UNDO/REDO*). Y a-t-il des informations inutiles dans le journal ?*

Exercice 17. *Spécifiez un algorithme de* REDO *qui effectue le parcours du log dans l'ordre inverse des insertions, et s'arrête dès que possible.*

Exercice 18. *Indiquez la bonne réponse aux questions suivantes (en les justifiant).*

1. *Pendant une reprise sur panne les opérations doivent être :*
   (a) *commutatives ;*
   (b) *associatives ;*
   (c) *idempotentes ;*
   (d) *distributives.*
2. *Dans un protocole de reprise sur panne avec mise à jour différée, quelles sont les opérations nécessaires :*
   (a) UNDO *;*
   (b) REDO *;*
   (c) UNDO *et* REDO *;*
   (d) *aucune des deux.*

Dans le cas d'un algorithme avec mises à jour différées, que doit-on conserver dans le log ?

1. *la valeur avant mise à jour*
2. *la valeur après mise à jour*
3. *les valeurs avant et après mise à jour*
4. *uniquement les* `start(T)` *et* `commit(T)`

Exercice 19. *Donner un exemple illustrant la nécessité d'effectuer un* UNDO *dans l'ordre inverse de l'exécution (donner les entrées du fichier journal, et expliquer le déroulement de l'annulation).*

Annexe A

Annexe A

L'exemple 6 donne le schéma de la base de données illustrant la plupart des exemples.

Exemple 6. *SchemaFilms.sql : Le schéma de la base Films*

```
/*
   Commandes de création de la base Films,  testé avec MySQL et PostgreSQL.
   Pour Oracle, il suffit de remplacer le type TEXT par le type LONG dans
   la table Film.
   Philippe Rigaux, 2004
*/

/* Destruction eventuelle des tables existantes */

DROP TABLE  Notation;
DROP TABLE  Role;
DROP TABLE  Film;
DROP TABLE Artiste;
DROP TABLE  Internaute;
DROP TABLE  Pays;
DROP TABLE  Genre;

/* Creation des tables                          */

CREATE TABLE Internaute (email VARCHAR (40) NOT NULL,
                         nom VARCHAR (30) NOT NULL ,
                         prenom VARCHAR (30) NOT NULL,
                         region VARCHAR (30),
                         CONSTRAINT PKInternaute PRIMARY KEY (email));

CREATE TABLE Pays (code    VARCHAR(4) NOT NULL,
                   nom  VARCHAR (30) DEFAULT 'Inconnu' NOT NULL,
                   langue VARCHAR (30) NOT NULL,
                   CONSTRAINT PKPays PRIMARY KEY (code));

CREATE TABLE Artiste  (idArtiste INTEGER NOT NULL,
                       nom VARCHAR (30) NOT NULL,
                       prenom VARCHAR (30) NOT NULL,
                       anneeNaiss INTEGER,
                       CONSTRAINT PKArtiste PRIMARY KEY (idArtiste),
                       CONSTRAINT UniqueNomArtiste UNIQUE (nom, prenom));

CREATE TABLE Film  (idFilm INTEGER NOT NULL,
                    titre    VARCHAR (50) NOT NULL,
```

```
                    annee    INTEGER NOT NULL,
                    idMES    INTEGER,
                    genre VARCHAR (20) NOT NULL,
                    /* Remplacer TEXT par LONG pour ORACLE */
                    resume      TEXT,
                    codePays    VARCHAR (4),
                    CONSTRAINT PKFilm PRIMARY KEY (idFilm),
                    FOREIGN KEY (idMES) REFERENCES Artiste,
                    FOREIGN KEY (codePays) REFERENCES Pays);

CREATE TABLE Notation (idFilm INTEGER NOT NULL,
                       email  VARCHAR (40) NOT NULL,
                       note  INTEGER NOT NULL,
                       CONSTRAINT PKNotation PRIMARY KEY (idFilm, email));

CREATE TABLE Role (idFilm  INTEGER NOT NULL,
                   idActeur INTEGER NOT NULL,
                   nomRole  VARCHAR(30),
                   CONSTRAINT PKRole PRIMARY KEY (idActeur,idFilm),
                   FOREIGN KEY (idFilm) REFERENCES Film,
                   FOREIGN KEY (idActeur) REFERENCES Artiste);

CREATE TABLE Genre (code    VARCHAR (20) NOT NULL,
                    CONSTRAINT PKGenre PRIMARY KEY (code));
```