GESTION DE TRANSACTIONS

1. INTRODUCTION

En bases de données, le concept de transaction a été introduit depuis près de trente ans [Bjork72]. Ce concept est aussi d'actualité dans les systèmes d'exploitation. Dans ce chapitre, nous étudions les problèmes liés à la gestion de transactions dans les systèmes de bases de données. Le concept de transaction est fondamental pour maintenir la cohérence des données. Une transaction est une unité de mise à jour composée de plusieurs opérations successives qui doivent être toutes exécutées ou pas du tout. En gestion, les transactions sont courtes (quelques secondes). En conception, elles peuvent être beaucoup plus longues (quelques minutes voire quelques heures). Le SGBD doit garantir les fameuses propriétés ACID des transactions. Outre l'atomicité mentionnée, les transactions effectuent des accès concurrents aux données qui doivent être contrôlés afin d'éviter les conflits entre lecteurs et écrivains. Cela nécessite d'isoler les mises à jour dont les effets doivent par ailleurs être durables. En conséquence, la gestion de transactions mélange intimement les problèmes de fiabilité et de reprise après panne avec les problèmes de concurrence d'accès. Les problèmes de sécurité qui recouvrent la confidentialité sont aussi connexes.

Ce chapitre essaie de faire le point sur tous les aspects de la gestion de transactions dans les SGBD centralisés. Après quelques rappels de base, nous traitons d'abord les problèmes de concurrence. Nous introduisons la théorie de la concurrence, qui s'appuie sur le concept de sérialisation, conduisant à n'accepter que les exécutions simultanées de transactions produisant les mêmes résultats qu'une exécution séquentielle des transactions, c'est-à-dire l'une après l'autre. Nous exposons la technique de verrouillage qui est une méthode préventive de conflits plutôt pessimiste. Bien qu'entraînant des difficultés telles que le verrou mortel, cette méthode est de loin la plus appliquée. Cependant, nous analysons aussi les méthodes optimistes qui laissent les conflits se produire et les corrigent après, telles que l'ordonnancement par estampille, la certification et les contrôles avec versions multiples.

Nous étudions ensuite les principes de la gestion de transactions. Tout repose sur les algorithmes de validation et de reprise après panne. Nous approfondissons les outils nécessaires à ces algorithmes puis les différents algorithmes de validation et de reprise, en incluant la validation en deux étapes, généralement appliquée même dans les systèmes centralisés. Comme exemple de méthode de reprise intégrée, nous décrivons la méthode ARIES implémentée à IBM, la référence en matière de reprise. Nous terminons la partie sur les transactions proprement dites en présentant les principaux modèles de transactions étendus introduits dans la littérature. Ces modèles sont destinés à permettre un meilleur support des transactions longues, en particulier pour les applications de conception qui nécessitent de longs travaux sur des objets complexes. Ces modèles sophistiqués commencent à être introduits dans les SGBD, notamment dans les SGBD objet ou objet-relationnel.

Pour terminer ce chapitre, nous traitons du problème un peu orthogonal de confidentialité. Beaucoup peut être dit sur la confidentialité. Nous résumons l'essentiel des méthodes DAC (*Discretionary Access Control*) et MAC (*Mandatory Access Control*). La première permet d'attribuer des autorisations aux utilisateurs et de les contrôler, la seconde fonctionne avec des niveaux de confidentialité. Nous concluons par quelques mots sur le cryptage des données.

2. NOTION DE TRANSACTION

Un modèle simplifié de SGBD se compose de programmes utilisateurs, d'un système et de mémoires secondaires. Les programmes accèdent aux données au moyen d'opérations du SGBD (*Select, Insert, Update, Delete*), généralement en SQL. Ces opérations déclenchent des actions de lecture et écriture sur disques (*Read, Write*), qui sont exécutées par le système, et qui conduisent à des entrées-sorties sur les mémoires secondaires. Afin de pouvoir déterminer les parties de programmes correctement exécutées, le concept de **transaction** a été proposé depuis longtemps.

2.1 Exemple de transaction

Les opérations classiques réalisées par une transaction sont les mises à jour ponctuelles de lignes par des écrans prédéfinis. Les transactions sont souvent répétitives. Elles s'appliquent toujours sur les données les plus récentes. Un exemple typique de transaction est fourni par le banc d'essais TPC A [Gray91]. Il s'agit de réaliser le débit/crédit sur une base de données bancaire. Le benchmark a pour objectif la mesure des performances du SGBD en nombre de transactions exécutées par seconde.

La base se compose des tables Agences, Comptes, Caissiers et Historique (voir figure XVI.1). Une agence gère 100.000 comptes, possède 100 caissiers et comporte un système avec 10 terminaux.

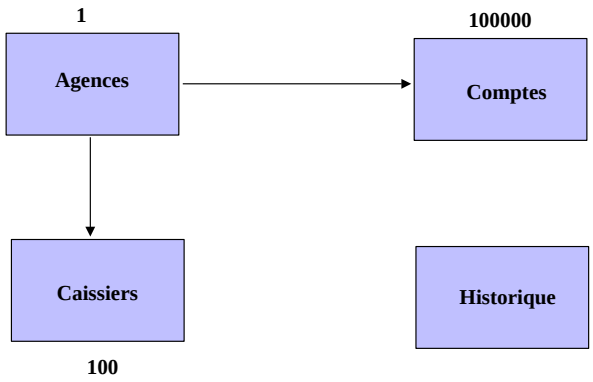

Figure XVI.1 — Schéma de la base TPC A

Le code de la transaction TPC A apparaît figure XVI.2. Il réalise le crédit ou le débit d'un compte d'un montant Delta et répercute cette mise à jour dans les autres tables. On voit qu'il s'agit bien d'un groupe de mises à jour ponctuelles mais liées. Le benchmark suppose que chaque terminal lance l'exécution d'une transaction toute les 10 secondes. 90% des transactions doivent avoir un temps de réponse inférieur à 2 secondes. La performance est obtenue en divisant le nombre de transactions exécutées par le temps d'exécution moyen. On obtient ainsi un nombre de transactions par seconde. Ces vingt dernières années, l'objectif de tous les SGBD a été d'exécuter un maximum de telles transactions par seconde.

```
Begin-Transaction
        Update Account Set Balance = Balance + Delta
                Where AccountId = Aid ;
        Insert into History (Aid, Tid, Bid, Delta, TimeStamp) ;
        Update Teller Set Balance = Balance + Delta
                Where TellerId = Tid ;
        Update Branch Set Balance = Balance + Delta
                Where TellerId = Tid ;
End-Transaction.
```

Figure XVI.2 — La transaction TPC

En pratique, les transactions doivent souvent cohabiter avec des requêtes décisionnelles, traitant un grand nombre de tuples en lecture, souvent pour calculer des agrégats. Par exemple, une requête décisionnelle pourra calculer la moyenne des avoir des comptes par agence comme suit :

```
    SELECT B.BRANCHID, AVG(C.BALANCE)
FROM BRANCH B, ACCOUNT C
WHERE B.BRACHID = C.BRANCHID
GROUP BY B.BRANCHID ;
```

Une telle requête parcourt tous les comptes et peut en conséquence empêcher les transactions débit/crédit de s'exécuter. Elle peut aussi être bloquée par les transactions débit/crédit. Nous allons étudier les solutions aux problèmes des transactions ci-dessous.

2.2 Propriété des transactions

Une **transaction** est donc composée d'une suite de requêtes dépendantes à la base qui doivent vérifier les propriétés d'**atomicité**, de **cohérence**, d'**isolation** et de **durabilité**, résumées par le vocable ACID.

> **Notion XVI.1 : Transaction (*Transaction*)**
>
> Séquence d'opérations liées comportant des mises à jour ponctuelles d'une base de données devant vérifier les propriétés d'atomicité, cohérence, isolation et durabilité (ACID).

Nous analysons brièvement les propriétés ACID ci-dessous.

2.2.1 Atomicité

Une **transaction** doit effectuer toutes ses mises à jour ou ne rien faire du tout. En cas d'échec, le système doit annuler toutes les modifications qu'elle a engagées. L'atomicité est menacée par les pannes de programme, du système ou du matériel, et plus généralement par tout événement susceptible d'interrompre une transaction en cours.

2.2.2 Cohérence

La transaction doit faire passer la base de données d'un état cohérent à un autre. En cas d'échec, l'état cohérent initial doit être restauré. La cohérence de la base peut être violée par un programme erroné ou un conflit d'accès concurrent entre transactions.

2.2.3 Isolation

Les résultats d'une transaction ne doivent être visibles aux autres transactions qu'une fois la transaction validée, afin d'éviter les interférences avec les autres transactions. Les accès concurrents peuvent mettre en question l'isolation.

2.2.4 Durabilité

Dès qu'une transaction valide ses modifications, le système doit garantir qu'elles seront conservées en cas de panne. Le problème essentiel survient en cas de panne, notamment lors des pannes disques.

Ces propriétés doivent être garanties dans le cadre d'un système SGBD centralisé, mais aussi dans les systèmes répartis. Elles nécessitent deux types de contrôle, qui sont intégrés dans un SGBD : contrôle de concurrence, résistance aux pannes avec validation et reprise. Nous allons les étudier successivement, puis nous étudierons la méthode intégrée ARIES qui est une des plus efficaces parmi celles implémentées dans les SGBD.

3. THEORIE DE LA CONCURRENCE

Cette section aborde les problèmes de gestion des accès concurrents. Les solutions proposées permettent de garantir la cohérence et l'isolation des mises à jour des transactions (le C et le I de ACID). Elles sont basées sur la théorie de la sérialisabilité des transactions, que nous examinons maintenant.

3.1 Objectifs

L'objectif général est de rendre invisible aux clients le partage simultané des données. Cette transparence nécessite des contrôles des accès concurrents au sein du SGBD. Ceux-ci s'effectuent au moyen de protocoles spéciaux permettant de synchroniser les mises à jour afin d'éviter les **pertes de mises à jour** et l'**apparitions d'incohérences.**

Une **perte de mise à jour** survient lorsqu'une transaction T1 exécute une mise à jour calculée à partir d'une valeur périmée de donnée, c'est-à-dire d'une valeur modifiée par une autre transaction T2 depuis la lecture par la transaction T1. La mise à jour de T2 est donc écrasée par celle de T1. Une perte de mise à jour est illustrée figure XVI.3. La mise à jour de la transaction T2 est perdue.

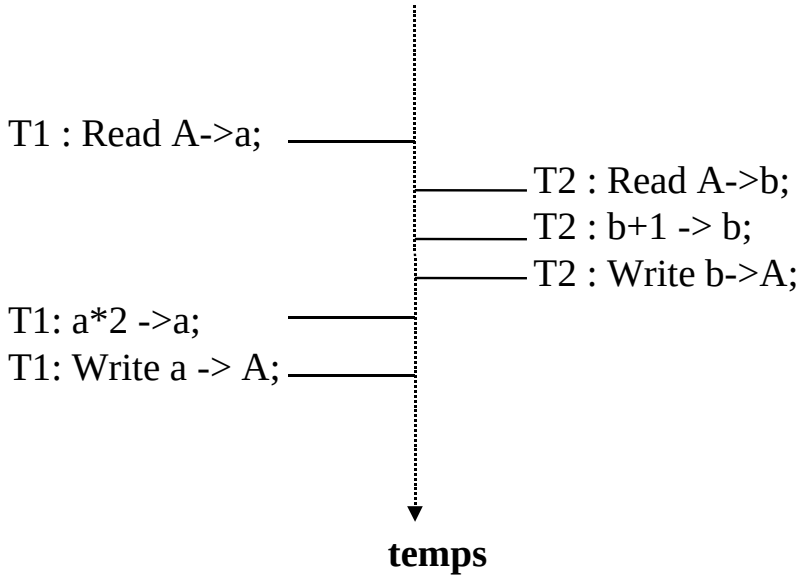

Figure XVI.3 — Exemple de perte de mise à jour

Une **incohérence** apparaît lorsque des données liées par une contrainte d'intégrité sont mises à jour par deux transactions dans des ordres différents, de sorte à enfreindre la contrainte. Par

exemple, soient deux données A et B devant rester égales. L'exécution de la séquence d'opérations {T1 : A = A+1 ; T2 : B = B*2 ; T2 : A = A*2; T1: B=B+1} rend en général A différent de B, du fait de la non-commutativité de l'addition et de la multiplication. Elle provoque donc l'apparition d'une incohérence. Cette situation est illustrée figure XVI.4.

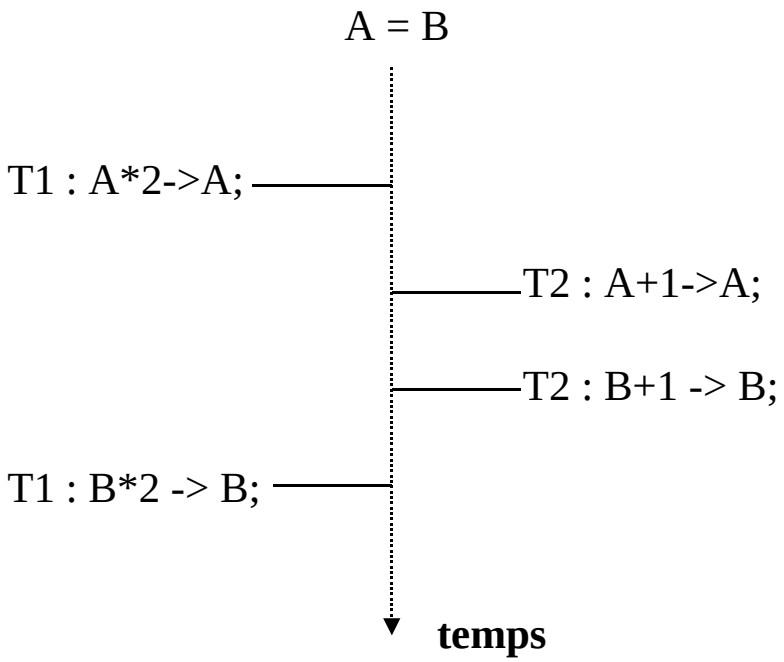

Figure XVI.4 — Exemple d'introduction d'incohérence

Un autre problème lié aux accès concurrents est la **non-reproductibilité des lectures** : deux lectures d'une même donnée dans une même transaction peuvent conduire à des valeurs différentes si la donnée est modifiée par une autre transaction entre les deux lectures (voir figure XVI.5). Le problème ne survient pas si les mises à jour sont isolées, c'est-à-dire non visibles par une autre transaction avant la fin de la transaction. Il en va de même de l'apparition d'incohérences. Pour les pertes de mise à jour, l'isolation des mises à jour n'est pas suffisante : il faut aussi ne pas laisser deux transactions modifier simultanément une même donnée.

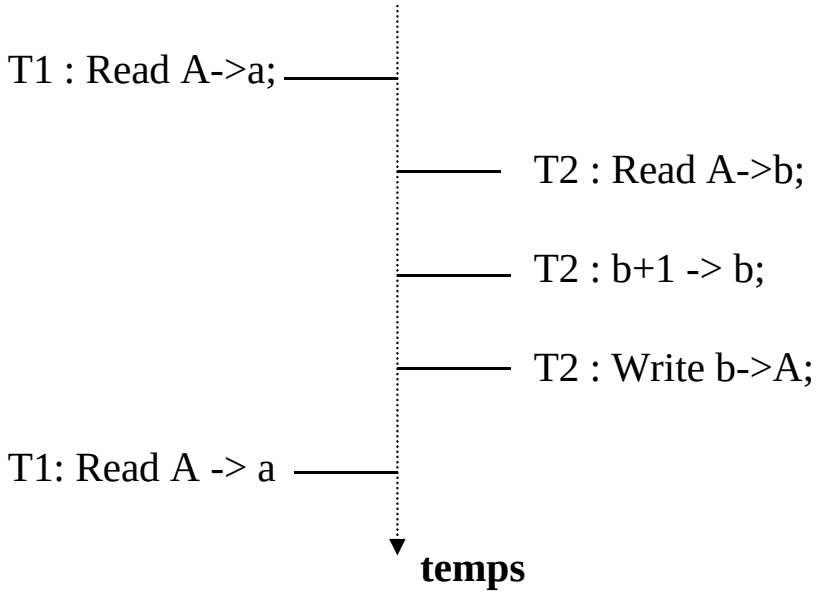

Figure XVI.5 — Exemple de non-reproductibilité des lectures

La résolution dans un système des problèmes évoqués nécessite la mise en place d'algorithmes de contrôle de concurrence spécialisés. Ces algorithmes s'appuient sur la théorie de la concurrence, que nous examinons ci-dessous.

3.2 Quelques définitions de base

Pour éviter perte d'opérations, incohérences et non reproductibilité des lectures, le SGBD doit contrôler l'accès aux données. L'unité de données contrôlée dépend du SGBD. De plus en plus de SGBD permettent des contrôles variables selon le type de transactions. Nous appellerons cette unité **granule de concurrence ;** le terme objet est parfois aussi employé.

> **Notion XVI.2 : Granule de concurrence (*Concurrency granule)***
>
> Unité de données dont les accès sont contrôlés individuellement par le SGBD.

Un granule au sens de la concurrence peut être une ligne, une page ou une table dans un système relationnel. Ce peut être un objet ou une page dans un SGBD objet. Nous discuterons de la taille du granule de concurrence plus loin.

Les granules de concurrence sont lus et écrits par les utilisateurs, éventuellement par parties. On appelle **action** un accès élémentaire à un granule.

Notion XVI.3 : Action (*Action*)

Unité indivisible exécutée par le SGBD sur un granule pour un utilisateur, constituée généralement par une lecture ou une écriture.

Un système de bases de données exécute donc une suite d'actions résultant de transactions concurrentes. Après complétude d'un ensemble de transactions (T1, T2,…Tn), une histoire du système peut être représentée par la suite des actions exécutées. Plus généralement, toute suite d'actions pouvant représenter une histoire possible sera appelée simplement **exécution**.

Notion XVI.4 : Exécution de transactions (*Schedule*, ou *Log*, ou *History*)

Séquence d'actions obtenues en intercalant les diverses actions des transactions T1, T2 …Tn tout en respectant l'ordre interne des actions de chaque transaction.

Une exécution respecte donc l'ordre des actions de chaque transaction participante et est, par définition, séquentielle. Par exemple, considérons les transactions T1 et T2 figure XVI.6, modifiant les données A et B reliées par la contrainte d'intégrité A = B ; A et B appartiennent à deux granules distincts, maximisant ainsi les possibilités de concurrence. Une exécution correcte de ces deux transactions est représentée figure XVI.7 (a). Une autre exécution est représentée figure XVI.7 (b), mais celle-là est inacceptable car elle conduit à une perte d'opérations.

T1	**T2**
Read A → a1	Read A → a2
a1+1 → a1	a2*2 → a2
Write a1 → A	Write a2 → A
Read B → b1	Read B → b2
b1+1 → b1	b2*2 → b2
Write b1 → B	Write b2 → B

Figure XVI.6 — Deux transactions T1 et T2

T1 : Read A → a1	T2 : Read A → a2
T1 : a1+1 → a1	T2 : a2*2 → a2
T1 : Write a1 → A	T1 : Read A → a1
T2 : Read A → a2	T1 : a1+1 → a1
T2 : a2*2 → a2	T2 : Write a2 → A
T2 : Write a2 → A	T2 : Read B → b2
T1 : Read B → b1	T2 : b2*2 → b2
T1 : b1+1 → b1	T1 : Write a1 → A
T1 : Write b1 → B	T1 : Read B → b1
T2 : Read B → b2	T1 : b1+1 → b1
T2 : b2*2 → b2	T1 : Write b1 → B
T2 : Write b2 → B	T2 : Write b2 → B
(a)	(b)

Figure XVI.7 — Deux exécutions des transactions T1 et T2

3.3 Propriétés des opérations sur granule

Un granule accédé concurremment obéit à des contraintes d'intégrité internes. Lors des modifications de la base de données, les granules sont modifiés par des suites d'actions constituant des unités fonctionnelles appelées **opérations**. Les opérations respectent la cohérence interne du granule, c'est-à-dire les contraintes d'intégrité qui relient les données appartenant au granule.

> **Notion XVI.5 : Opération (*Operation*)**
>
> Suite d'actions accomplissant une fonction sur un granule en respectant sa cohérence interne.

Par exemple, si le granule est la page, les opérations de base sont souvent LIRE (page) et ECRIRE (page), qui sont également dans bien des systèmes des actions indivisibles. Si le granule est l'article, des opérations plus globales nécessitant plusieurs actions indivisibles sont LIRE (article) et ECRIRE (article), mais aussi MODIFIER (article) et INSERER (article). Avec ces opérations de base, il est possible d'en construire d'autres plus globales encore. Sur un objet typé, tel un compte en banque, on peut distinguer des opérations, créer, créditer, débiter, détruire, etc.

L'application d'opérations à des granules conduit à des **résultats**. Le résultat d'une opération est constitué par l'état du granule concerné après l'application de l'opération considérée et par les effets de bords qu'elle provoque. Par exemple, le résultat d'une opération LIRE est représenté par la valeur du tampon récepteur après exécution, alors que le résultat d'une transaction modifiant une base de données est l'état des granules modifiés après exécution ainsi que la valeur des messages édités.

Les opérations sont enchevêtrées au niveau des actions lors de l'exécution simultanée de transactions. Deux opérations qui ne modifient aucun granule et qui appartiennent à deux transactions différentes peuvent être enchevêtrées de manière quelconque sans modifier les résultats de leur exécution. Autrement dit, toute intercalation d'opérations n'effectuant que des lectures conduit à des résultats identiques à une exécution successive de ces opérations. Plus généralement, il est possible de définir la notion **d'opérations compatibles**.

> **Notion XVI.6 : Opérations compatibles (*Compatible operations*)**
>
> Opérations 0i et 0j dont toute exécution simultanée donne le même résultat qu'une exécution séquentielle 0i suivie de 0j ou de 0j suivie de 0i (à noter que les résultats 0i puis 0j, et 0j puis 0i peuvent être différents).

Considérons par exemple les opérations représentées figure XVI.8. Les opérations O11 et O21 sont compatibles ; O11 et O12 ne le sont pas.

O11

{ Read A → a1
a1+1 → a1
Write a1 → A }

O12

{ Read A → a2
a2*2 → a2
Write a2 → A

O21

{ Read B → b1
b1+1 → b1
Write b1 → B }

O22

{ Read B → b2
b2*2 → b2
Write b2 → B }

O31

{ Read A → a1
a1+10 → a1
Write a1 → A }

Figure XVI.8 — Exemple d'opérations

Il est important de remarquer que deux opérations travaillent sur deux granules différents sont toujours compatibles. En effet, dans ce cas aucune perte d'opérations ne peut survenir si l'on intercale les opérations. Or il est simple de voir que deux opérations sont incompatibles lorsque qu'il existe une possibilité d'intercalation générant une perte d'opérations.

Les problèmes surviennent avec les opérations incompatibles, lorsqu'une au moins modifie un granule auquel l'autre a accédé. L'ordre d'exécution des deux opérations peut alors changer les résultats. Dans d'autres cas, il peut être indifférent. Plus généralement, nous définirons la notion d'**opérations permutables**, qu'il faut bien distinguer de celle d'opérations compatibles (la première est une notion indépendante de l'ordre d'exécution, alors que la seconde est définie à partir de la comparaison des ordres d'exécution).

> **Notion XVI.7 : Opérations permutables (*Permutable operations*)**
>
> Opérations 0i et 0j telles que toute exécution de 0i suivie par 0j donne le même résultat que celle de 0j suivie par 0i.

Par exemple, les opérations O11 et O31 représentées figure XVI.8 sont permutables alors que les opérations O11 et O12 ne le sont pas. Soulignons que deux opérations travaillant sur des granules différents sont toujours permutables. En effet, dans ce cas, l'exécution de la première ne peut modifier le résultat de la seconde et réciproquement. Par exemple, O11 et O12 sont permutables. Plus généralement, deux opérations compatibles sont permutables, mais la réciproque n'est pas vraie.

3.4 Caractérisation des exécutions correctes

Certaines exécutions introduisent des pertes d'opérations ou des inconsistances, comme nous l'avons vu ci-dessus. L'objectif du contrôle de concurrence consiste à ne laisser s'exécuter que des exécutions sans pertes d'opérations ou inconsistances. Il est bien connu que l'exécution successive de transactions (sans simultanéité entre transactions) est un cas particulier d'exécution sans perte d'opérations ni inconsistances. Une telle exécution est appelée **succession** et peut être définie plus formellement comme suit :

> **Notion XVI.8 : Succession (*Serial Schedule*)**
>
> Exécution E d'un ensemble de transactions {T1, T2,…Tn} telle qu'il existe une permutation π de (1,2…n) telle que :
>
> $$E = < T\,\pi(1) ; T\pi(2) ; \ldots T\,\pi(n) >$$

Afin d'assurer l'absence de conflit, il est simple de ne tolérer que les exécutions qui donnent le même résultat qu'une succession pour chaque transaction. De telles exécutions sont dites **sérialisables**.

Notion XVI.9 : Exécution sérialisable (*Serialisable Schedule*)

Exécution E d'un ensemble de transactions {T1, T2,…Tn} donnant globalement et pour chaque transaction participante le même résultat qu'une succession de (T1, T2…Tn).

Le problème du contrôle de concurrence est donc d'assurer qu'un système centralisé (ou réparti) ne peut générer que des exécutions sérialisables. C'est là une condition suffisante pour assurer l'absence de conflit dont la nécessité peut être discutée [Gardarin77]. En fait, la condition est nécessaire si le système n'a pas de connaissances sur la sémantique des opérations.

Afin de caractériser les exécutions sérialisables, nous introduisons deux transformations de base d'une exécution de transactions. Tout d'abord, la séparation d'opérations compatibles Oi et Oj exécutées par des transactions différentes consiste à remplacer une exécution simultanée des opérations E (Oi, Oj) par la séquence donnant le même résultat, soit < Oi ; Oj > ou < Oj ; Oi >. La séparation d'opérations permet donc de mettre en succession des opérations compatibles exécutées par des transactions différentes. Ensuite, la permutation d'opérations permutables Oi et Oj exécutées par des transactions différentes consiste à changer l'ordre d'exécution de ces opérations ; par exemple la séquence < Oi ; Oj > est remplacée par la séquence < Oj ; Oi >.

Une condition suffisante pour qu'une exécution soit sérialisable est qu'elle puisse être transformée par séparation des opérations compatibles et permutations des opérations permutables en une succession des transactions composantes. En effet, par définition, séparations et permutations conservent les résultats. Par suite, si l'exécution peut être transformée en une succession, elle donne le même résultat que cette succession pour chaque transaction et est donc sérialisable. La condition n'est pas nécessaire car, au moins pour certaines valeurs des données, des opérations incompatibles ou non permutables peuvent être exécutées simultanément sans conflits.

A titre d'exemple, considérons l'exécution représentée figure XVI.7(a). En représentant seulement globalement les opérations, cette exécution s'écrit :

T1 : A + 1→ A

T2 : A * 2 → A

T1 : B + 1 → B

T2 : B * 2 → B

Les opérations A *2 → A et B + 1 → B sont permutables car elles agissent sur des granules différents. Par suite, cette exécution peut être transformée en :

T1 : A + 1 → A

T1 : B + 1 → B

T2 : A * 2 → A

T2 : B * 2 → B

qui est une succession de T1 puis T2. Par suite, l'exécution figure XVI.7(a) est sérialisable.

3.5 Graphe de précédence

Une exécution sérialisable est correcte car elle donne un résultat que l'on obtiendrait en exécutant les transactions l'une après l'autre. Lorsqu'on examine une séquence d'opérations résultant d'une exécution simultanée d'un ensemble de transactions, il apparaît que l'ordre de certaines opérations ne peut être changé sans changer le résultat, du fait de la non-commutativité des opérateurs exécutés (par exemple, addition et multiplication).

Les chercheurs ont ainsi abouti à définir la notion de **précédence** de transactions dans une exécution simultanée.

> **Notion XVI.10 : Précédence (*Precedence*)**
>
> Propriété stipulant qu'une transaction a accompli une opération Oi sur une donnée avant qu'une autre transaction accomplisse une opération Oj, Oi et Oj n'étant pas commutatives ({Oi; Oj} ≠ {Oj; Oi}).

La notion de précédence est générale et s'applique à tout type d'opération. En pratique, les systèmes ne considèrent d'ordinaire que les opérations de lecture et d'écriture. Les précédences sont alors créées par les séquences d'actions de base lecture et écriture. Les séquences non commutatives lecture puis écriture, écriture puis lecture, écriture puis écriture, d'une même donnée introduisent des précédences. Plus précisément, l'une des séquences :

- Ti : lire(D) … Tj : écrire(D) ;
- Ti : écrire(D) … Tj : écrire(D) ;
- Ti : écrire(D) … Tj : lire(D) ;

implique que Ti précède Tj.

Considérons une exécution simultanée de transactions. La relation de précédence entre transactions peut être représentée par un graphe :

> **Notion XV.11 : Graphe de précédence (*Precendency graph*)**
>
> Graphe dont les nœuds représentent les transactions et dans lequel il existe un arc de Ti vers Tj si Ti précède Tj dans l'exécution analysée.

Une exécution simultanée des transactions T1, T2 et T3 et le graphe de précédence associé sont illustrés figure XVI.9.

Il est simple de montrer qu'une condition suffisante de sérialisabilité est que le graphe de précédence soit sans circuit. En effet, dans ce cas, il est toujours possible de transformer l'exécution simultanée en une succession en séparant puis permutant les opérations. L'ordre des transactions dans la succession est induit par le graphe sans circuit. Par exemple, l'exécution simultanée représentée figure XVI.9 n'est pas sérialisable puisque le graphe de précédence possède un circuit.

{ T1 : Lire A ;
T2 : Ecrire A ;
T2 : Lire B ;
T2 : Ecrire B ;
T3 : Lire A ;
T1 : Ecrire B }

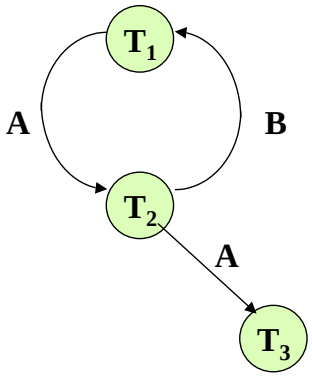

Figure XVI.9 — Exemple de graphe de précédence

4. CONTROLE DE CONCURRENCE PESSIMISTE

Deux types de techniques ont été développées pour garantir la sérialisabilité des transactions : les techniques de prévention des conflits qui empêchent leur apparition et les techniques de détection qui laissent les conflits se produire mais les détectent et annulent leurs effets. Les premières sont dites pessimistes car elles préviennent des conflits qui ne surviennent en général pas. Elles sont basées sur le verrouillage. Les secondes sont dites optimistes. Dans cette partie, nous étudions le verrouillage qui est de loin la technique la plus appliquée.

4.1 Le Verrouillage deux phases

Le verrouillage deux phases est une technique de prévention des conflits basée sur le blocage des objets par des verrous en lecture ou écriture avant d'effectuer une opération de sélection ou de mise à jour. En théorie, une transaction ne peut relâcher de verrous avant d'avoir obtenu tous ceux qui lui sont nécessaires, afin de garantir la correction du mécanisme [Eswaran76].

> **Notion XVI.12 : Verrouillage deux phases (*Two Phase Locking*)**
>
> Technique de contrôle des accès concurrents consistant à verrouiller les objets au fur et à mesure des accès par une transaction et à relâcher les verrous seulement après obtention de tous les verrous.

Une transaction comporte donc deux phases (voir figure XVI.10) : une phase d'acquisition de verrous et une phase de relâchement. Cette condition garantit un ordre identique des transactions sur les objets accédés en mode incompatible. Cet ordre est celui d'exécution des points de

verrouillage maximal φ. En pratique, afin de garantir l'isolation des mises à jour, les verrous sont seulement relâchés en fin de transaction, lors de la validation.

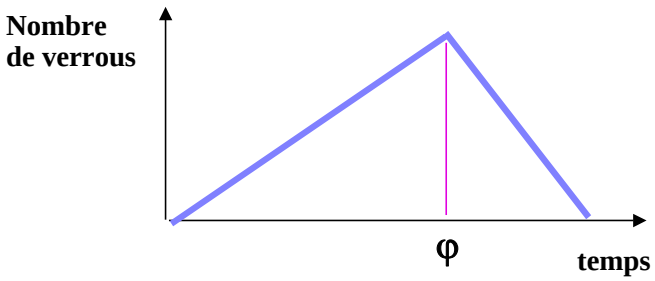

Figure XVI.10 — Comportement des transactions deux phases

Les verrous sont demandés au moyen de l'opération `Lock(G,M)` et relâchés au moyen de l'opération `Unlock(G)`, G étant le granule à verrouiller/déverrouiller et M le mode de verrouillage. Les compatibilités entre opérations découlent des précédences ; elles sont décrites par la matrice représentée figure XVI.11. Les algorithmes `Lock` et `Unlock` sont détaillés figure XVI.12. Lors d'une demande de verrouillage, si l'objet demandé est verrouillé, la transaction demandante est mise en attente jusqu'à libération de l'objet. Ainsi, toute transaction attend la fin des transactions incompatibles, ce qui garantit un graphe de précédence sans circuit. Une analyse fine montre que les circuits sont transformés en verrous mortels.

	L	E
L	V	F
E	F	F

Figure XVI.11 — Compatibilité des opérations de lecture/écriture

```
Bool Function Lock(Transaction t, Granule G, Mode M){
 Cverrou := 0 ;
 Pour chaque transaction i ≠ t ayant verrouillé l'objet G faire {
        Cverrou := Cverrou ∪ t.verrou(G) }; // cumuler verrous sur G
        si Compatible (Mode, Cverrou) alors {
                t.verrou(G) = t.verrou(G) ∪ M; // marquer l'objet verrouillé
                Lock := true ; }
        sinon {
                insérer (t, Mode) dans queue de G ; // mise en attente de t
                bloquer la transaction t ;
                Lock := false ; } ;
} ;
```

```
Procédure Unlock(Transaction t, Granule G){
 t.verrou(G) := 0 ; // Remise à 0 du verrou de la transaction sur G
 Pour chaque transaction i dans la queue de G faire {
        si Lock(i, G,M) alors { // Tentative de verrouillage pour Ti
                enlever (i,M) de la queue de G ;
                débloquer i ; } ; };
} ;
```

Figure XVI.12 — Algorithmes de verrouillage et déverrouillage

L'application du verrouillage dans un système pose le problème du choix du granule de verrouillage. Dans une base de données relationnelle, les objets à verrouiller peuvent être des tables, des pages ou des tuples. Une granularité variable des verrous est souhaitable, les transactions manipulant beaucoup de tuples pouvant verrouiller au niveau table ou page, celles accédant ponctuellement à quelques tuples ayant la capacité de verrouiller au niveau tuple. Nous examinerons ci-dessous le problème des granules de taille variable. Le choix d'une unité de verrouillage fine (par exemple le tuple) minimise bien sûr les risques de conflits. Elle maximise cependant la complexité et le coût du verrouillage.

4.2 Degré d'isolation en SQL2

Le verrouillage, tel que présenté ci-dessus, est très limitatif du point de vue des exécutions simultanées possibles. Afin de proposer une approche plus permissive et de laisser s'exécuter simultanément des transactions présentant des dangers limités de corruption des données, le groupe de normalisation de SQL2 a défini des **degrés d'isolation** emboîtés, du moins contraignant au plus contraignant, ce dernier correspondant au verrouillage deux phases. Le groupe distingue

les **verrous courts** relâchés après exécution de l'opération et les **verrous longs** relâchés en fin de transaction. Le degré de verrouillage souhaité est choisi par le développeur de la transaction parmi les suivants :

- Le degré 0 garantit les non pertes des mises à jour ; il correspond à la pose de verrous courts exclusifs lors des écritures.

- Le degré 1 garantit la cohérence des mises à jour ; il génère la pose de verrous longs exclusifs en écriture par le système.

- Le degré 2 assure la cohérence des lectures individuelles ; il ajoute la pose de verrous courts partagés en lecture à ceux du degré 1.

- Le degré 3 atteste de la reproductibilité des lectures ; il complète le niveau 2 avec la pose de verrous longs partagés en lecture.

Ainsi, le développeur peut contrôler la pose des verrous. Un choix autre que le degré 3 doit être effectué avec précaution dans les transactions de mises à jour, car il implique des risques d'incohérence. Seul en effet le degré 3 assure la sérialisabilité des transactions.

4.3 Le problème du verrou mortel

4.3.1 Définition

Le verrouillage soulève quelques problèmes. Le problème essentiel est le risque de **verrous mortels** entre transactions.

> **Notion XVI.13 : Verrou mortel (*Deadlock*)**
>
> Situation dans laquelle un groupe de transactions est bloqué, chaque transaction du groupe attendant qu'une autre transaction du groupe relâche un verrou pour pouvoir continuer.

Une transaction Ti attend une transaction Tj si Ti a demandé l'obtention d'un verrou sur un objet verrouillé par Tj en mode incompatible. La figure XVI.13 donne un exemple de verrou mortel.

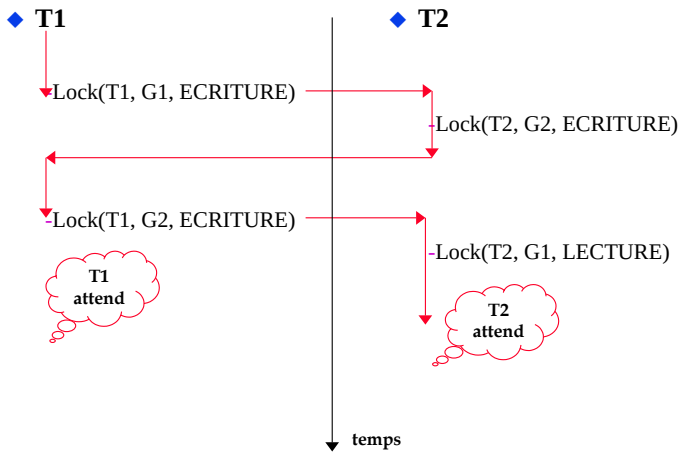

Figure XVI.13 — Exemple de verrou mortel

Deux classes de solutions sont possibles dans les SGBD afin de résoudre le problème du verrou mortel : la première, appelée **prévention**, empêche les situations de verrous mortels de survenir ; la seconde, appelée **détection**, est une solution curative qui consiste à supprimer les verrous mortels par reprise de transactions.

4.3.2 Représentation du verrou mortel

Nous présentons ci-dessous deux types de graphes représentant les verrous mortels : le graphe des attentes et le graphe des allocations.

4.3.2.1 Graphe des attentes

Le **graphe des attentes** [Murphy68] est un graphe G (T,Γ) où T est l'ensemble des transactions concurrentes $\{T_1, T_2 \ldots T_n\}$ se partageant les granules $G_1, G_2 \ldots G_m$ et Γ est la relation « attend » définie par : T_p « attend » T_q si et seulement si T_p attend le verrouillage d'un objet G_i alors que cette requête ne peut pas être acceptée parce que G_i a déjà été verrouillé par T_q.

> **Notion XVI.14 : Graphe des attentes (*Waiting graph*)**
>
> Graphe dont les nœuds correspondent aux transactions et les arcs représentent les attentes entre transactions.

Le théorème suivant a été introduit dès 1968 [Murphy68] : il existe une situation de verrou mortel si et seulement si le graphe des attentes possède un circuit. La figure XVI.14 illustre ce théorème sur l'exemple introduit ci-dessus. La preuve est simple. En effet, si le graphe des attentes possède un circuit, c'est qu'il existe un groupe de transactions tel que : T_1 attend T_2, T_2 attend $T_3, \ldots, T_k$ attend T_1. Chaque transaction du groupe est donc bloquée en attente d'un objet du fait de l'utilisation de cet objet par une autre transaction du groupe. La fin d'exécution de toutes les transactions n'appartenant pas au groupe ne permet donc pas de débloquer une transaction du groupe.

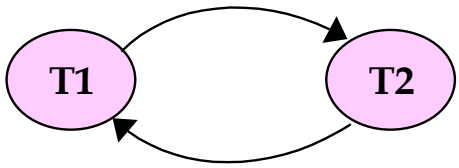

Figure XVI.14 — Exemple de graphe d'attente avec circuit

Réciproquement, l'existence d'une situation de verrou mortel implique l'existence d'au moins un circuit. S'il n'en était pas ainsi, tout groupe de transaction serait tel que le sous-graphe des attentes qu'il engendre ne possèderait pas de circuit. Après exécution de toutes les transactions n'appartenant pas au groupe, il serait donc possible de débloquer une transaction du groupe puisqu'un graphe sans circuit possède au moins un sommet pendant. Toutes les transactions appartenant à un circuit sont en situation de verrou mortel ; de plus, une transaction attendant une transaction en situation de verrou mortel est elle-même en situation de verrou mortel (voir figure XVI.15).

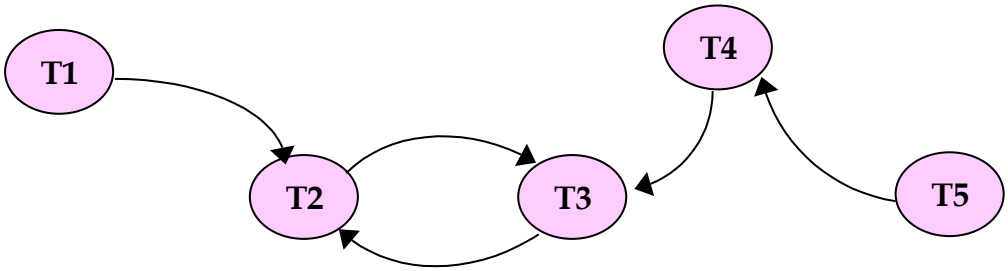

Figure XVI.15 — Transactions en situation de verrou mortel

Il est intéressant d'établir le rapport entre graphes des attentes et graphe de précédence. Par définition, si une transaction T_i `attend` une transaction T_j, alors T_j a verrouillé un objet O dont le verrouillage est demandé par T_i dans un mode incompatible. Ainsi, l'opération pour laquelle Tj a verrouillé O sera exécutée avant celle demandée par Tj car les deux opérations sont incompatibles et donc non permutables. Donc Tj `précède` Ti. Toutefois, la relation de précédence n'implique généralement pas la relation d'attente. Donc, en changeant l'orientation des arcs du graphe des attentes, on obtient un sous-graphe du graphe de précédence. Cela implique que si le graphe des attentes a un circuit, il en sera de même pour le graphe de précédence. En conséquence, une situation de verrou mortel ne peut pas donner lieu à une exécution sérialisable même s'il était possible de terminer les transactions interbloquées.

4.3.2.2 Graphe des allocations

Le **graphe des allocations** [Holt72] est composé de deux ensembles de sommets :

1. l'ensemble T des transactions

2. l'ensemble O des objets.

Un arc relie l'objet Oi à la transaction Tp si et seulement si Tp a obtenu le verrouillage de Oi dans au moins un mode d'opération ; l'arc est valué par les modes d'opérations alloués. Un arc relie la transaction Tp à l'objet Oi si et seulement si Tp a demandé et n'a pas encore obtenu l'allocation de ce granule ; l'arc est valué par les modes d'opérations demandés. La figure XVI.16 représente le graphe des allocations de l'exemple de la figure XVI.13.

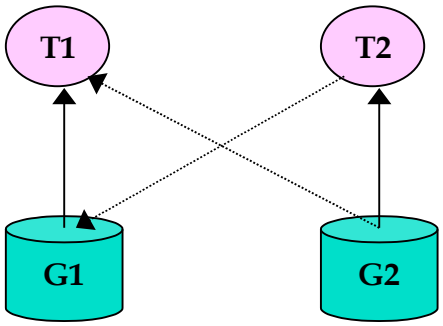

Figure XVI.16 — Exemple de graphe des allocations

Il est simple de démontrer le théorème suivant : une condition nécessaire d'existence de verrou mortel est la présence d'un circuit sur le graphe des allocations. Cette condition n'est en général pas suffisante. La preuve s'effectue par l'absurde. En effet, il est possible de prouver que s'il n'existe pas de circuits sur le graphe des allocations, il ne peut y avoir d'interblocage. En effet, soit T un groupe quelconque de transactions. Du fait que le graphe des allocations est sans circuit, le sous-graphe obtenu après exécution des transactions n'appartenant pas à T est sans circuit. Il possède donc un sommet pendant. Ce sommet ne peut être un granule car un granule non verrouillé ne peut être attendu. Dans tout groupe de transactions T, l'exécution supposée de toutes les transactions n'appartenant pas au groupe T conduit donc à débloquer une transaction du groupe. Il n'y a donc pas situation de verrou mortel.

Il faut remarquer que la condition est suffisante dans le cas où les seuls modes de lecture et d'écriture sont distingués. Ceci permet en général de détecter les situations de verrou mortel par détection de circuits dans le graphe des allocations dans la plupart des systèmes classiques. Nous pouvons noter qu'en général il n'y a pas de rapport direct entre graphe de précédence et graphe des allocations. Cependant, si les seuls modes existants sont lecture et écriture, la présence d'un circuit dans le graphe des allocations est équivalente à l'existence d'une situation de verrou mortel et donc à celle d'un circuit dans le graphe des attentes. Sous cette condition, la présence d'un circuit dans le graphe des allocations entraîne ainsi celle d'un circuit dans le graphe de précédence.

4.3.3 Prévention du verrou mortel

La prévention consiste à appliquer une stratégie de verrouillage garantissant que le problème ne survient pas. Il existe classiquement deux approches, l'une basée sur l'ordonnancement des ressources, l'autre sur celui des transactions. L'ordonnancement des ressources (tables, pages, objets, tuples) pour les allouer dans un ordre fixé aux transactions est impraticable vu le grand nombre d'objets distribués. L'ordonnancement des transactions est possible à partir d'une **estampille**.

Notion XVI.15 : Estampille de transaction (*Transaction Timestamp*)

Numéro unique attribué à une transaction permettant de l'ordonner strictement par rapport aux autres transactions.

En général, l'estampille attribuée à une transaction est son horodate de lancement concaténée avec le numéro de processeur sur lequel elle est lancée, ceci afin d'empêcher l'égalité des estampilles pour deux transactions lancées au même instant : celles-ci diffèrent alors par le numéro de processeur en poids faibles. Le numéro de processeur n'est utile que dans les architectures parallèles.

A partir des estampilles, deux algorithmes ont été proposés [Rosenkrantz77] pour prévenir les verrous mortels. Tous deux consistent à défaire plus ou moins directement une transaction dans le cas d'attente, de sorte à ne permettre que des attentes sans risque de circuit. L'algorithme WAIT-DIE consiste à annuler les transactions qui demandent des ressources tenues par des transactions plus anciennes. La transaction la plus récente est alors reprise avec la même estampille ; elle finit ainsi par devenir ancienne et par passer. Il ne peut y avoir de verrou mortel, les seules attentes possibles étant dans l'ordre où une transaction ancienne attend une transaction récente. Le contrôle des attentes imposé par l'algorithme est précisé figure XVI.17.

```
// Algorithm WAIT-DIE
Procédure Attendre (Ti,Tj) {
        // Ti réclame un verrou tenu par Tj
        si ts(Ti) < ts(Tj) alors Ti waits sinon Ti dies ; }
```

Figure XVI.17 — Contrôle des attentes dans l'algorithme WAIT-DIE

L'algorithme WOUND-WAIT est un peu plus subtil. Tout type d'attente est permis. Mais si une transaction plus ancienne attend une plus récente, la récente est blessée (*wounded*), ce qui signifie qu'elle ne peut plus attendre : si elle réclame un verrou tenu par une autre transaction, elle est automatiquement défaite et reprise. Le contrôle des attentes imposé par l'algorithme est représenté figure XVI.18 ; une transaction blessée ne peut donc attendre.

```
// Algorithm WOUND-WAIT
Procédure Attendre (Ti,Tj) {
        // Ti réclame un verrou tenu par Tj
        si ts(Ti) < ts(Tj) alors Tj is wounded sinon Ti waits ; }
```

Figure XVI.18 — Contrôle des attentes dans l'algorithme WOUND-WAIT

Les deux algorithmes empêchent les situations de verrous mortels en donnant priorité aux transactions les plus anciennes. L'algorithme WOUND-WAIT provoque en principe moins de reprises de transactions et sera en général préféré.

4.3.4 Détection du verrou mortel

La prévention provoque en général trop de reprises de transactions, car les méthodes défont des transactions alors que les verrous mortels ne sont pas sûrs d'apparaître. Au contraire, la détection laisse le problème se produire, détecte les circuits d'attente et annule certaines transactions afin de rompre les circuits d'attente.

Un algorithme de détection de l'interblocage peut se déduire d'un algorithme de détection de circuits appliqué au graphe des attentes ou des allocations. Nous présentons ici une mise en œuvre de l'algorithme qui consiste à tester si un graphe est sans circuit par élimination successive des sommets pendants.

Sur le graphe des attentes, un sommet est pendant si la transaction qu'il représente n'attend le verrouillage d'aucun granule. Soit N(k) le nombre de granules dont la transaction T_k attend le verrouillage. Une première réduction du graphe peut être obtenue par élimination des sommets pendants, donc tels que N(k) = 0 . Le problème est alors de recalculer les nombres de granules attendus N(k) après réduction pour pouvoir effectuer la réduction suivante. Ceci peut être fait en comptant les demandes qui peuvent être satisfaites après chaque réduction, et en décrémentant N(k) chaque fois que l'on compte une demande de la transaction T_k. L'application de la méthode nécessite deux précautions :

1. marquer les demandes comptées pour ne pas les compter deux fois ;
2. disposer d'une procédure permettant de tester si une demande peut être satisfaite compte tenu de l'état des verrouillages des transactions non encore éliminées du graphe des attentes.

Soit donc `SLOCK (k, G, M)` une procédure booléenne permettant de tester si la demande du granule G en mode M de la transaction Tk peut être satisfaite compte tenu de l'état d'allocation des granules aux transactions présentes dans le graphe des attentes. Cette procédure répond `VRAI` si la demande peut être satisfaite et `FAUX` sinon. Le code de cette procédure est analogue à celui de l'algorithme `LOCK` vu ci-dessus, à ceci près que seule les transactions de T sont prises en compte (les autres sont supposées exécutées et terminées) et que l'état de verrouillage n'est pas modifié. La figure XVI.19 présente une procédure `DETECTER` répondant `VRAI` s'il y a situation de verrou mortel et `FAUX` sinon. Cette procédure élimine donc progressivement les transactions pendantes du graphe des attentes.

```
Bool Procedure Detecter {
T = {Liste des transactions telles que N(k) ≠ 0 }
G = {liste des granules allouées aux transactions dans T}
Pour chaque entrée g de G faire
        Pour chaque demande non marquée M de Tk en attente de g faire {
                Si SLOCK(k, g, Q) = VRAI alors {
                        Marquer Q ;
                        N(k) = N(k) –1 ;
                        Si N(k) = 0 alors {
                                Eliminer Tk de T ;
                                Ajouter les granules verrouillés par Tk à G ;
                                } ;
                        } ;
                } ;
        Si T = ∅ alors DETECTER = FAUX
        Sinon DETECTER = VRAI ;
} ;
```

Figure XVI.19 — Algorithme de détection du verrou mortel

Quand une situation d'interblocage est détectée, le problème qui se pose est de choisir une transaction à recycler de façon à briser les circuits du graphe des attentes. L'algorithme de détection présenté ci-dessus fournit la liste des transactions en situation d'interblocage. Il faut donc choisir une transaction de cette liste. Cependant, tous les choix ne sont pas judicieux, comme le montre la figure XVI.20. Une solution à ce problème peut être de recycler la transaction qui bloque le plus grand nombre d'autres transactions, c'est-à-dire qui correspond au sommet de demi-degré intérieur le plus élevé sur le graphe des attentes. Le choix de la transaction à reprendre doit aussi chercher à minimiser le coût de reprise.

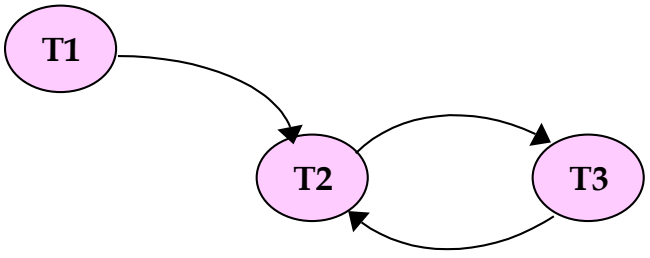

Figure XVI.20 — Exemple de choix difficile de transaction à reprendre

Le coût d'une solution de type détection avec reprise peut être réduit. En effet, il est possible de déclencher un algorithme détection seulement quand une transaction attend un verrouillage depuis un temps important (par exemple, quelques secondes), plutôt qu'à chaque début d'attente.

D'autres algorithmes de détection sont possibles. Le graphe d'allocation est souvent utilisé dans les systèmes répartis. Lors d'une attente qui dure, un algorithme envoie une enquête le long des arcs du graphe des allocations. Cette enquête est transmise au granule attendu, puis aux transactions bloquant ce granule, puis aux granules attendus s'il en existe, etc. Si l'enquête revient à la transaction initiale, c'est qu'il y a un verrou mortel. Cet algorithme est moins efficace en centralisé.

4.4 Autres problèmes soulevés par le verrouillage

Un autre problème soulevé par le verrouillage est le problème de famine, encore appelé **blocage permanent**. Ce problème survient dès qu'un groupe de transactions se coalise, en effectuant des opérations compatibles entre elles (par exemple des lectures), contre une transaction individuelle qui désire effectuer une opération incompatible avec les précédentes (par exemple une écriture). La transaction individuelle peut alors attendre indéfiniment. Les solutions à ce problème consistent en général à mettre en file d'attente les demandes de verrouillage dans leur ordre d'arrivée et à n'accepter une requête de verrouillage que si elle est compatible avec les verrouillages en cours et ceux demandés par les requêtes les plus prioritaires en attente. Il faut noter que les algorithmes de prévention DIE-WAIT et WOUND-WAIT ne conduisent jamais une transaction à l'attente infinie. En effet, une transaction qui meurt garde son ancienne estampille lorsqu'elle est relancée. Elle devient ainsi plus vieille et finit toujours par passer, le principe étant d'avorter les transactions les plus jeunes.

Le **problème des fantômes** a également été soulevé [Eswaran76]. Il survient lorsqu'un objet est introduit dans la base de données et ne peut être pris en compte par une transaction en cours qui devrait logiquement le traiter. Par exemple, soit la base de données de réservation de places d'avions, représentée figure XVI.21, composée de deux relations : PASSAGER (nom, numéro de vol, numéro de siège) et OCCUPATION (numéro de vol, nombre de passagers). Considérons maintenant les transactions suivantes :

- T1 (1^e partie) : lister la relation PASSAGER en lisant tuple à tuple ;
- T1 (2^e partie) : lister la relation OCCUPATION d'un seul tuple ;
- T2 : insérer dans PASSAGER le tuple (Fantomas, 100, 13) et incrémenter le nombre de passagers du vol numéro 100.

Les transactions sont supposées verrouiller les tuples. Supposons que les transactions s'enchevêtrent dans l'ordre : T1 (1^e partie), T2, T1 (2^e partie). C'est une exécution valide puisque T2 accède à un granule non verrouillé qui n'existe même pas lorsque T1 exécute sa 1^e partie : le tuple « Fantomas ». Toutefois, le résultat de T1 est une liste de 4 noms alors que le nombre de passagers est 5. « Fantomas » est ici un fantôme pour T1.

Passagers	Nom	N°Vol	N°Siège
	Dubois	100	3
	Durand	100	5
	Dupont	100	10
	Martin	100	15

Occupation	N°Vol	NbrePass
	100	4

Figure XVI.21 — Illustration du problème des fantômes

Ce problème, ainsi que la difficulté de citer les granules à verrouiller, peuvent être résolus par la définition de granules logiques (dans l'exemple, les passagers du vol 100) au moyen de prédicats [Eswaran76]. Le verrouillage par prédicat permet également de définir des granules de tailles variables, ajustées aux besoins des transactions. Malheureusement, il nécessite des algorithmes pour déterminer si deux prédicats sont disjoints et ce problème de logique n'a pas de solution suffisamment efficace pour être appliqué dynamiquement lors du verrouillage des objets. De plus, les prédicats sont définis sur des domaines dont les extensions ne sont pas consultables dans la base pour des raisons évidentes de performance. Donc, il est très difficile de déterminer si deux prédicats sont disjoints ; par exemple, PROFESSION = " Ingénieur" et SALAIRE < 7000 seront déterminés logiquement non disjoints, alors qu'ils le sont dans la plupart des bases de données. Le verrouillage par prédicat est donc en pratique source d'attentes inutiles et finalement inapplicable.

4.5 Les améliorations du verrouillage

Malgré le grand nombre de solutions proposées par les chercheurs, les systèmes continuent à appliquer le verrouillage deux phases avec prévention ou détection des verrous mortels. Les degrés d'isolation choisis par les transactions permettent de maximiser le partage des données en limitant le contrôle. Le verrouillage est cependant très limitatif. Un premier problème qui se pose est le choix de la granularité des objets à verrouiller. Au-delà, la recherche sur l'amélioration du verrouillage continue et des solutions parfois applicables ont été proposées. Nous analysons les plus connues ci-dessous.

4.5.1 Verrouillage à granularité variable

Une granularité variable est possible. La technique consiste à définir un graphe acyclique d'objets emboîtés et à verrouiller à partir de la racine dans un mode d'intention jusqu'aux feuilles désirées qui sont verrouillées en mode explicite. Par exemple, une transaction désirant verrouiller un tuple en mode écriture verrouillera la table en intention d'écriture, puis la page en intention d'écriture, et enfin le tuple en mode écriture. Les modes d'intentions obéissent aux mêmes règles de compatibilités que les modes explicites, mais sont compatibles entre eux. Le verrouillage en intention permet simplement d'éviter les conflits avec les modes explicites. Sur un même objet, les modes explicites règlent les conflits. La figure XVI.22 donne la matrice de compatibilité entre les modes lecture (L), écriture (E), intention de lecture (IL) et intention d'écriture (IE).

	L	E	IL	IE
L	V	F	V	F
E	F	F	F	F
IL	V	F	V	V
IE	F	F	V	V

Figure XVI.22 — Compatibilités entre les modes normaux et d'intention

Une telle méthode peut être appliquée dans les bases relationnelles, mais aussi objet. Le graphe d'inclusion pour une base de données objet peut être base, extension de classe, page ou groupe (*cluster*) et objet. Il est représenté figure XVI.23.

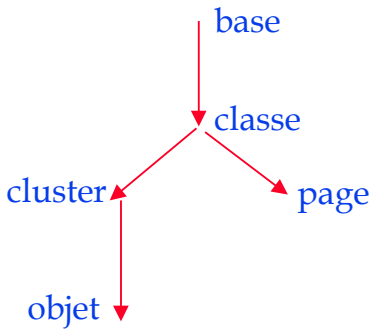

Figure XVI.23 — Granules variables pour une BD objet

4.5.2 Verrouillage multi-versions

Le **verrouillage multi-versions** suppose l'existence d'au moins une version précédente d'un objet en cours de modification. C'est généralement le cas dans les systèmes puisque, comme nous le verrons ci-dessous, un journal des images avant mise à jour est géré en mémoire. Le principe est simple : lors d'un verrouillage en lecture, si le granule est occupé par une transaction en mode incompatible (donc en écriture en pratique), la version précédente du granule est délivrée à l'utilisateur. Une telle technique est viable lorsque les granules verrouillés sont des pages ou des tuples. Au-delà, il est difficile de constituer une version cohérente du granule rapidement.

Avec le verrouillage multi-versions, tout se passe comme si la transaction qui lit avait été lancée avant la transaction qui écrit. Malheureusement, la sérialisabilité n'est pas garantie si la transaction qui accède à la version ancienne écrit par ailleurs. En effet, la mise à jour de la transaction en quelque sorte sautée n'est pas prise en compte par la transaction lisant, qui risque d'obtenir des résultats dépassés ne pouvant servir aux mises à jour. Seule les transactions n'effectuant que des lectures peuvent utiliser ce mécanisme, aussi appelé lecture dans le passé. Si l'on veut de plus garantir la reproductibilité des lectures, il faut gérer au niveau du système un cache des lectures effectuées dans le passé, afin de les retrouver lors de la deuxième lecture. Ce type de verrouillage est réservé au décisionnel qui ainsi n'est pas perturbé par les mises à jour.

4.5.3 Verrouillage altruiste

Le **verrouillage altruiste** suppose connu les patterns d'accès aux granules des transactions, c'est-à-dire au moins l'ordre d'accès aux granules et les granules non accédés. Il devient alors possible de relâcher les verrous tenus par une transaction longue lorsqu'on sait qu'elle n'utilisera plus un granule. Ce granule est alors ajouté à l'ensemble des granules utilisés par la transaction, appelée la **traînée** (*wake*) de la transaction. En cas de reprise, toutes les transactions dans la traînée d'une transaction sont aussi reprises (c'est-à-dire celles ayant accédé à des objets dans la traînée). C'est l'**effet domino**, selon lequel une transaction implique la reprise d'autres pour compenser la non isolation (voir ci-dessous). Par exemple, figure XVI.24, on voit que T2 peut s'exécuter alors que T1 n'est pas terminée car on sait que T1 ne reviendra pas sur c et que T2 n'accédera pas a. Si T3 est reprise, T4 doit l'être aussi car elle a modifié c, lui-même modifié par T3.

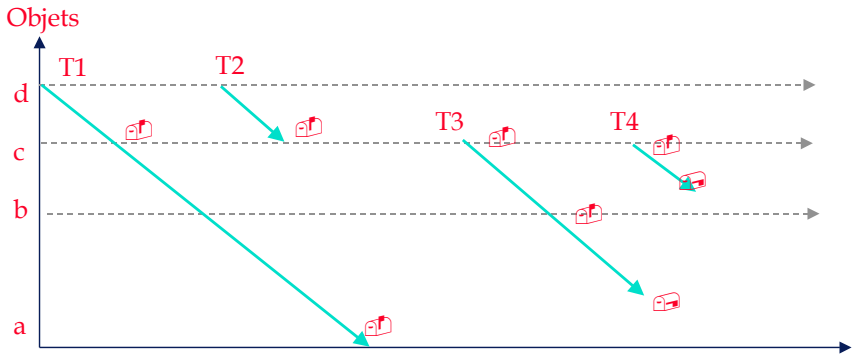

Figure XVI.24 — Exemple de verrouillage altruiste

Le verrouillage altruiste est difficile à appliquer en pratique car on ne connaît pas les patterns d'accès des transactions. De plus, l'effet domino reste mal maîtrisé. Cette technique pourrait être intéressante pour faire cohabiter des transactions longues avec des transactions courtes [Barghouti91].

4.5.4 Commutativité sémantique d'opérations

Il est possible d'exploiter la **sémantique des opérations**, notamment dans les systèmes objet ou objet-relationnel où il existe des types (ou classes) [Gardarin76, Weihl88, Cart90]. Chaque type est caractérisé par une liste d'opérations (méthodes). Comme nous l'avons vu ci-dessus, les opérations commutatives sont permutables et n'entraînent pas de conflits de précédence. Il est donc intéressant de distinguer des modes de verrouillage plus fins que lecture et écriture, permettant de prendre en compte les opérations effectives sur les objets typés, et non pas les actions de base lecture et écriture.

Introduire la commutativité entre opérations est utile si les opérations de mises à jour, a priori incompatibles, sont souvent commutatives. Si l'on regarde seulement le nom de l'opération, la commutativité est rare car elle dépend souvent des paramètres, notamment de la réponse. Les chercheurs ont donc introduit des modes d'opérations incluant les réponses. Par exemple, avec des ensembles, il est intéressant de distinguer insérer avec succès `[Ins,ok]`, supprimer avec succès

`[Del,ok]`, tester l'appartenance avec succès `[In, true]` et avec échec `[In, False]`. La matrice de commutativité est représentée figure XVI.25.

	[Ins,ok]	[Del,ok]	[In,true]	[In,False]
[Ins,ok]	1	0	0	0
[Del,ok]	0	1	0	1
[In,true]	0	0	1	1
[In,False]	0	1	1	1

Figure XVI.25 — Commutativité d'opérations sur ensemble

Dans un système typé, chaque objet peut posséder en option un contrôle de concurrence défini au niveau de la classe. Les verrouillages sont alors délégués au contrôleur du type d'objet. Celui-ci laisse passer simultanément les verrouillages en modes d'opérations commutatives. Par exemple, un ensemble pourra être verrouillé simultanément par deux transactions en mode `[Ins,ok]`, ou en `[In,False]` et `[Del,ok]`. Le contrôleur bloque seulement les opérations non commutatives (ordonnancement).

Les reprises en cas de panne ou d'obtention d'un résultat invalide (non verrouillé par exemple) sont cependant difficiles. En effet, comme pour le verrouillage altruiste, le modèle est ouvert et permet à des transactions de voir des données modifiées par des transactions non encore terminées. Il faut donc gérer la portée des transactions, par exemple sous forme de listes de transactions dépendantes. L'effet domino introduit ci-dessus survient : lors de la reprise d'une transaction, toutes les transactions dépendantes doivent être reprises.

Certains auteurs [Weihl88] ont aussi considéré la commutativité en avant et la commutativité en arrière. Par exemple `[In,true]` et `[Insert,ok]` commutent en avant mais pas en arrière : si l'on exécute ces deux opérations à partir d'un état s sur lequel elles sont définies, on obtient bien le même résultat quel que soit l'ordre. Mais si l'on a l'exécution `[Insert, ok] [In, true]`, on n'est pas sur que `[In,true]` soit définie sur l'état initial (l'objet inséré peut être celui qui a permis le succès de l'opération `In`). Donc, on ne peut pas commuter lorsqu'on défait et refait une exécution. Tout cela complique les procédures de reprises et l'exploitation de la commutativité des opérations.

5. CONTROLES DE CONCURRENCE OPTIMISTE

Le verrouillage est une solution pessimiste : il empêche les conflits de se produire, ou plutôt les transforme en verrou mortel. Analysons maintenant une autre gamme de solutions qualifiées d'optimistes qui laissent se produire les conflits et les résout ensuite.

5.1 Ordonnancement par estampillage

Bien que le verrouillage avec prévention ou détection du verrou mortel soit la technique généralement appliquée dans les SGBD, de nombreuses autres techniques ont été proposées. En particulier, l'**ordonnancement par estampille** peut être utilisé non seulement pour résoudre les verrous mortels comme vu ci-dessus, mais plus complètement pour garantir la sérialisabilité des transactions.

Une méthode simple consiste à conserver pour chaque objet accédé (tuple ou page), l'estampille du dernier écrivain W et celle du plus jeune lecteur R. Le contrôleur de concurrence vérifie alors :

1. que les accès en écriture s'effectuent dans l'ordre croissant des estampilles de transactions par rapport aux opérations créant une précédence, donc l'écrivain W et le lecteur R.

2. que les accès en lecture s'effectuent dans l'ordre croissant des estampilles de transactions par rapport aux opérations créant une précédence, donc par rapport à l'écrivain W.

On aboutit donc à un contrôle très simple d'ordonnancement des accès conformément à l'ordre de lancement des transactions [Gardarin78, Bernstein80]. En cas de désordre, il suffit de reprendre la transaction ayant créé le désordre. Les contrôles nécessaires en lecture et écriture sont résumés figure XVI.26.

```
// Contrôle d'ordonnancement des transactions
Fonction Ecrire(Ti, O) { // la transaction Ti demande l'écriture de l'objet O;
        si ts(Ti) < W(O) ou ts(Ti) < R(O) alors abort(Ti)
        sinon executer_ecrire(Ti,O) } ;

Fonction Lire(Ti,O) {// la transaction Ti demande la lecture de l'objet O;
        si ts(Ti) < W(O) alors abort(Ti)
        sinon executer_lire(Ti,O) } ;
```

Figure XVI.26 — Algorithme d'ordonnancement des accès par estampillage

L'algorithme d'ordonnancement par estampillage soulève plusieurs problèmes. De fait, les estampilles W et R associées à chaque objet remplacent les verrous. Il n'y a pas d'attente, celles-ci étant remplacées par des reprises de transaction en cas d'accès ne respectant pas l'ordre de lancement des transactions. Ceci conduit en général à beaucoup trop de reprises. Une amélioration possible consiste à garder d'anciennes versions des objets. Si l'estampille du lecteur ne dépasse pas celle du dernier écrivain, on peut délivrer une ancienne version, plus exactement, la première inférieure à l'estampille du lecteur. Ainsi, il n'y a plus de reprise lors des lectures. La méthode est cependant difficile à mettre en œuvre et n'est guère utilisée aujourd'hui.

5.2 Certification optimiste

La certification optimiste est une méthode de type curative, qui laisse les transactions s'exécuter et effectue un contrôle garantissant la sérialisabilité en fin de transaction. Une transaction est divisée en trois phases : phase d'accès, phase de certification et phase d'écriture. Pendant la phase d'accès, chaque contrôleur de concurrence garde les références des objets lus/écrits par la transaction. Pendant la phase de certification, le contrôleur vérifie l'absence de conflits avec les transactions certifiées pendant la phase d'accès. S'il y a conflit, la certification est refusée et la transaction défaite puis reprise. La phase d'écriture permet l'enregistrement des mises à jour dans la base pour les seules transactions certifiées.

En résumé, nous introduirons ainsi la notion de certification qui peut être effectuée de différentes manières :

> **Notion XVI.16 : Certification de transaction (*Transaction certification*)**
>
> Action consistant à vérifier et garantir que l'intégration dans la base de donnés des mises à jour préparées en mémoire par une transaction préservera la sérialisabilité des transactions.

Vérifier l'absence de conflits pourrait s'effectuer en testant la non-introduction de circuits dans le graphe de précédence. L'algorithme commun [Kung81] de certification est plus simple. Il consiste à mémoriser les ensembles d'objets lus (*Read Set RS*) et écrits (*Write Set WS*) par une transaction. La certification de la transaction Ti consiste à tester que RS(Ti) n'intersecte pas avec WS(Tj) et que WS(Ti) n'intersecte pas avec WS(Tj) ou RS(Tj) pour toutes les transactions Tj lancées après Ti. On vérifie donc que les transactions n'agissent pas en modes incompatibles avec les transactions concurrentes avant de les valider. L'algorithme est représenté figure XVI.27.

```
Bool Fonction Certifier(Ti) {
        Certifier = VRAI ;
        Pour chaque transaction t concurrente faire {
                Si RS(Ti) ∩WS(t) ≠ ∅ ou WS(Ti) ∩ RS(t) ≠ ∅ ou WS(Ti) ∩WS(t) ≠ ∅
                Alors {
                                Certifier = FAUX ;
                                Abort(Ti) ;
                        } ;
        } ;
} ;
```

Figure XVI.27 — Algorithme de certification optimiste

En résumé, cette méthode optimiste est analogue au verrouillage, mais tous les verrous sont laissés passants et les conflits ne sont détectés que lors de la validation des transactions. L'avantage est la simplicité du contrôleur de concurrence qui se résume à mémoriser les objets accédés et à un test simple d'intersection d'ensembles de références lors de la validation. L'inconvénient majeur est la

tendance à reprendre beaucoup de transactions en cas de conflits fréquents. La méthode optimiste est donc seulement valable pour les cas où les conflits sont rares.

5.3 Estampillage multi-versions

Comme pour le verrouillage deux phases et même mieux, la stratégie d'ordonnancement par estampillage vue ci-dessus peut être améliorée en gardant plusieurs versions d'un même granule [Reed79]. Pour chaque objet O, le système peut maintenir :

1. un ensemble d'estampilles en écriture {EEi(O)} avec les valeurs associées { Oi }, chacune d'elles correspondant à une version i ;

2. un ensemble d'estampilles en lecture {ELi(O)}.

Il est alors possible d'assurer l'ordonnancement des lectures par rapport aux écritures sans jamais reprendre une transaction lisant. Pour cela, il suffit de délivrer à une transaction Ti demandant à lire l'objet O la version ayant une estampille en écriture immédiatement inférieure à i. Ainsi, Ti précédera toutes les créations d'estampilles supérieures écrivant l'objet considéré et suivra celles d'estampilles inférieures. Ti sera donc correctement séquencée. Tout se passe comme si Tj avait demandé la lecture juste après l'écriture de la version d'estampille immédiatement inférieure. L'algorithme de contrôle de l'opération LIRE avec un dispositif d'ordonnancement partiel multi-versions est représenté figure XVI.28.

```
// Lecture de la bonne version dans le passé
Fonction Lire(Ti,O) {// la transaction Ti demande la lecture de l'objet O;
        j = index de la dernière version de O ;
        Tant que ts(Ti) < W(O) faire j = j-1 ; // chercher la version précédent Ti
        executer_lire(Ti,Oj) } ; // lire la bonne version
}
```

Figure XVI.28 — Algorithme de lecture avec ordonnancement multi-versions

Il est en général très difficile de refaire le passé. Cependant, il est parfois possible de forcer l'ordonnancement des écritures de Ti en insérant une nouvelle version créée par Ti juste après celle d'estampille immédiatement inférieure, soit Oj. Malheureusement, si une transaction Tk (k >i) a lu la version Oj, alors cette lecture doit aussi être reséquencée. Ce n'est possible que si la transaction Tk pouvait être reprise. Afin d'éviter la reprise de transactions terminées, on préférera reprendre l'écrivain Tj avec une nouvelle estampille i' supérieure à k.

L'algorithme de contrôle de l'opération WRITE correspondant est représenté figure XVI.29. Les notations sont identiques à celles utilisées ci-dessus, les indices désignant les numéros de versions d'objets.

```
// Réordonnancement des écritures dans le passé
Fonction Ecrire(Ti, O) { // la transaction Ti demande l'écriture de l'objet O;
        j = index de la dernière version de O ;
        Tant que ts(Ti) < W(Oj) faire j = j-1 ; // chercher la version précédent Ti
        Si ts(Ti) < R(Oj) alors abort(Ti) // abort si lecture non dans l'ordre
        sinon executer_ecrire(Ti,Oj) } ; // écrire en bonne place
}
```

Figure XVI.29 — Algorithme d'écriture avec ordonnancement multi-versions

La figure XVI.30 illustre l'algorithme. Les transactions entrent en conflit sur un objet O unique dont les versions successives sont représentées par des rectangles. La situation originale est représentée en haut de la figure. Trois versions de l'objet existent, successivement créées par les transactions 1, 5 et 7. La version 1 a été lue par la transaction 1, la version 5 par la transaction 7 et la version 7 par la transaction 10. Nous supposons que T3 accomplit une écriture sur l'objet O après l'avoir lu. La nouvelle version 3 créée est insérée en bonne place. Nous supposons ensuite que T6 procède à une écriture sur O. L'objet ayant été lu par T7, il faudrait refaire le passé. On préférera annuler T6 et la relancer plus tard.

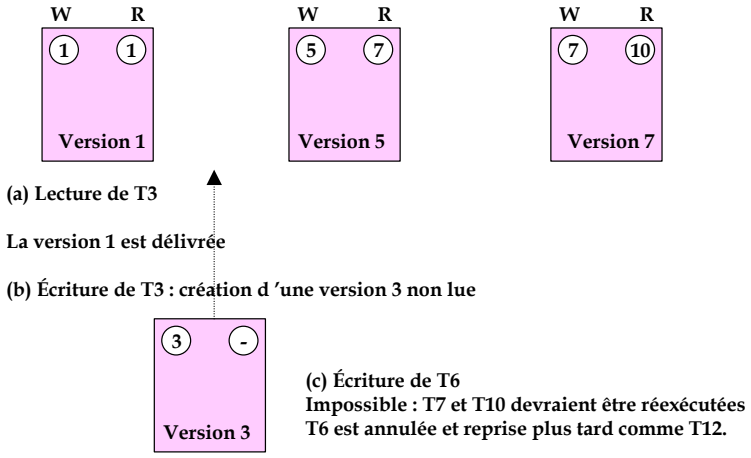

Figure XVI.30 — Exemple de reséquencement

En résumé, beaucoup d'algorithmes basés sur des estampilles peuvent être inventés pour contrôler les accès concurrents. Il est même possible de mixer estampilles et verrouillage, comme déjà vu au niveau des algorithmes DIE-WAIT et WOUND-WAIT. Cependant, les performances de ces algorithmes restent faibles car ils provoquent tous des reprises qui deviennent de plus en plus fréquentes lorsqu'il y a un plus grand nombre de conflits, donc lorsque le système est chargé. Voilà sans doute pourquoi la plupart des SGBD utilisent le verrouillage deux phases.

6. LES PRINCIPES DE LA RESISTANCE AUX PANNES

Nous avons étudié ci-dessus les mécanismes de contrôle de concurrence permettant de limiter les interférences entre transactions. Abordons maintenant les techniques de résistance aux pannes, qui permettent aussi d'assurer les propriétés ACID des transactions, particulièrement l'atomicité et la durabilité.

6.1 Principaux types de pannes

Ils existe différentes sources de pannes dans un SGBD. Celles-ci peuvent être causées par une erreur humaine, une erreur de programmation ou le dysfonctionnement d'un composant matériel. On peut distinguer [Gray78, Fernandez80] :

1. **La panne d'une action** survient quand une commande au SGBD est mal exécutée. En général, elle est détectée par le système qui retourne un code erreur au programme d'application. Ce dernier peut alors tenter de corriger l'erreur et continuer la transaction.

2. **La panne d'une transaction** survient quand une transaction ne peut continuer par suite d'une erreur de programmation, d'un mauvais ordonnancement des accès concurrents, d'un verrou mortel ou d'une panne d'action non corrigeable. Il faut alors défaire les mises à jour effectuées par la transaction avant de la relancer.

3. **La panne du système** nécessite l'arrêt du système et son redémarrage. La mémoire secondaire n'est pas affectée par ce type de panne ; en revanche, la mémoire centrale est perdue par suite du rechargement du système.

4. **La panne de mémoire secondaire** peut survenir soit suite à une défaillance matérielle, soit suite à une défaillance logicielle impliquant de mauvaises écritures. Alors, une partie de la mémoire secondaire est perdue. Il s'agit du type de panne le plus catastrophique.

Les différents types de panne sont de fréquence très différente. Par exemple, les deux premiers peuvent survenir plusieurs fois par minute alors qu'une panne système apparaît en général plusieurs fois par mois et qu'une panne mémoire secondaire n'arrive que quelquefois par an, voire moins. Aussi, seul le dernier type de panne conduit à faire appel aux archives et peut s'avérer, dans certains cas très rares, non récupérable.

6.2 Objectifs de la résistance aux pannes

L'objectif essentiel est de minimiser le travail perdu tout en assurant un retour à des données cohérentes après pannes. Compte tenu de l'aspect non instantané de l'apparition d'une panne et de sa détection, nous considérerons généralement que la transaction est l'unité de traitement atomique, ou si l'on préfère l'unité de reprise. Cependant, ceci n'est pas toujours vrai et une unité plus faible a été retenue dans les systèmes basés sur SQL à l'aide de la notion de point de reprise de transaction (*savepoint*). Une transaction est divisée en étapes, encore appelées **unités d'œuvre** (voir figure XVI.31). L'atomicité de chaque unité d'œuvre doit être garantie par le système transactionnel. Une panne de transaction provoque le retour au dernier point de reprise de la

transaction. L'exécution d'un point de reprise permet de conserver les variables en mémoire de la transaction, bien que la reproductibilité des lectures ne soit en général pas garantie entre les unités d'œuvre. Dans la suite, pour simplifier, nous ne considérerons en général que des transactions composées d'une seule unité d'œuvre.

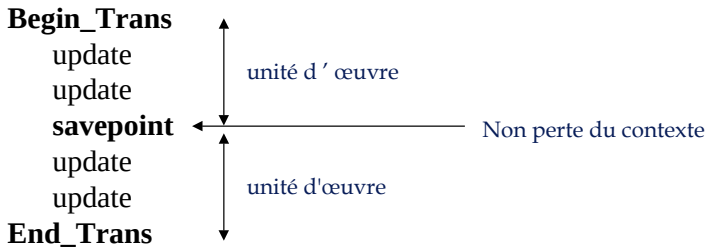

Figure XVI.31 — Transaction composée d'unités d'œuvre multiples

Les objectifs premiers de la résistance aux pannes sont de fournir un protocole aux applications permettant d'assurer l'atomicité. Pour ce faire, une application doit pouvoir commencer l'exécution d'une transaction et la terminer avec succès ou par un échec. Des actions atomiques sont ainsi fournies par le système de gestion de transactions aux applications. En plus de la création d'une transaction, ces actions correspondent aux trois notions de **validation** (encore appelée *commitment*, consolidation ou confirmation), **d'annulation** (encore appelée *abort* ou abandon ou retour arrière) et de **reprise** (encore appelée restauration ou *redo*). Nous définissons ci-dessous ces trois notions.

> **Notion XVI.17 : Validation de transaction (*Transaction commitment*)**
>
> Action atomique spéciale (appelée Commettre ou *Commit*), exécutée en fin de transaction, provoquant l'intégration définitive de toutes les mises à jour non encore commise de la transaction exécutante dans la base de données.

La validation est donc la terminaison avec succès d'une transaction. Dans le cas de transactions composées de plusieurs unités d'œuvre, une validation est effectuée à la fin de chaque unité d'œuvre. L'opposé de la validation est **l'annulation**.

> **Notion XVI.18 : Annulation de transaction (*Transaction abort*)**
>
> Action atomique spéciale (appelée Annuler ou Défaire ou *Abort* ou *Undo*), généralement exécutée après une défaillance, provoquant l'annulation de toutes les mises à jour de la base effectuées par la transaction et non encore commises.

Notez que seules les transactions non validées peuvent être annulées. Défaire une transaction validée est une opération impossible sauf à utiliser des versions antérieures de la base. Par contre, une transaction défaite peut être refaite (on dit aussi rejouée) : c'est l'objet de la **reprise**.

Notion XVI.19 : Reprise de transaction (*Transaction redo*)

Exécution d'une action spéciale (appelée Refaire ou *Redo*) qui refait les mises à jour d'une transaction précédemment annulée dans la base de données.

La reprise peut s'effectuer à partir de journaux des mises à jour, comme nous le verrons ci-dessous. Elle peut aussi nécessiter une nouvelle exécution de la transaction.

6.3 Interface applicative transactionnelle

La mise à disposition des fonctionnalités de validation, annulation et reprise a nécessité le développement d'une interface entre les applications et le système transactionnel. Cette interface a été standardisée par l'X/OPEN dans le cadre de l'architecture DTP (*Distributed Transaction Processing*). Pour des transactions simples, elle se résume à trois actions de base :

- `Trid` **`Begin`** `(context)` permet de débuter une transaction en fournissant un contexte utilisateur ; elle retourne un identifiant de transaction Trid ;
- **`Commit`** `(TrId)` valide la transaction dont l'identifiant est passé en paramètre ;
- **`Abort`**`(TrId)` annule la transaction dont l'identifiant est passé en paramètre.

Des points de sauvegardes peuvent être introduits, comme vu ci-dessus avec les opérations :

- `SaveId` **`Save`**`(TrId)` déclare un point de sauvegarde pour la transaction et demande la validation de l'unité d'œuvre en cours ; elle retourne un identifiant de point de sauvegarde ;
- **`Rollback`** `(Trid, SaveId)` permet de revenir au point de sauvegarde référencé, en général le dernier.

Quelques opérations de service supplémentaires sont proposées, telles par exemple :

- **`ChainWork`** `(context)` valide la transaction en cours et ouvre une nouvelle transaction ;
- `TrId` **`MyTrId`**`()` retourne l'identifiant de la transaction qui l'exécute ;
- **`Status`**`(TrId)` permet de savoir quel est l'état de la transaction référencée en paramètre ; elle peut être active, annulée, commise, en cours d'annulation ou de validation.

En résumé, l'objectif d'un système transactionnel au sein d'un SGBD est de réaliser efficacement les opérations précédentes. En plus, celui-ci doit bien sûr intégrer un contrôle de concurrence correct et efficace.

6.4 Eléments utilisés pour la résistance aux pannes

Nous décrivons maintenant les différents éléments utilisés pour la validation et la reprise de transactions.

6.4.1 Mémoire stable

Avant tout, il est nécessaire de disposer de mémoires secondaires fiables et sûres. Plus précisément, la notion de **mémoire stable** recouvre l'espace disque qui n'est ni perdu ni endommagé lors d'une panne simple, d'action, de transaction ou de système. La mémoire stable est organisée en pages. Une écriture de page dans la mémoire stable est atomique : une page est soit correctement écrite sur mémoire secondaire, soit pas du tout ; elle ne peut être douteuse ou partiellement écrite. De plus, si elle est écrite, elle ne peut être détruite que par une panne catastrophique explicite ou par une réécriture.

> **Notion XVI.20 : Mémoire stable (*Stable store*)**
>
> Mémoire découpée en pages dans laquelle une écriture de page est soit correctement exécutée, soit non exécutée, garantissant la mémorisation de la page jusqu'à réécriture, donc sa non perte suite à des pannes simples.

Dans les SGBD, la mémoire stable est le disque ; il permet de mémoriser les données persistantes. La réalisation d'une mémoire sûre garantissant l'atomicité des écritures n'est pas triviale. Les techniques utilisées sont en général les codes de redondances, ainsi que les doubles écritures. Dans la suite, nous considérons les mémoires stables comme sûres.

6.4.2 Cache volatile

Les SGBD utilisent des caches des bases de données en mémoire afin d'améliorer les performances. Comme vu ci-dessus, la mémoire centrale peut être perdue en cas de panne système : le **cache** est donc une mémoire **volatile**.

> **Notion XVI.21 : Cache volatile (*Transient cache*)**
>
> Zone mémoire centrale contenant un cache de la base, considérée comme perdue après une panne système.

Les transactions actives exécutent des mises à jour dont l'effet apparaît dans le cache et n'est pas instantanément reporté sur disque. En théorie, l'effet d'une transaction devrait être reporté de manière atomique lors de la validation. La figure XVI.32 illustre les mouvements entre mémoire volatile (le cache) et mémoire stable souhaitables lors de la validation ou de l'annulation. Ceci est la vue logique donnée à l'utilisateur. En pratique, les mécanismes pour assurer un tel fonctionnement logique sont plus complexes.

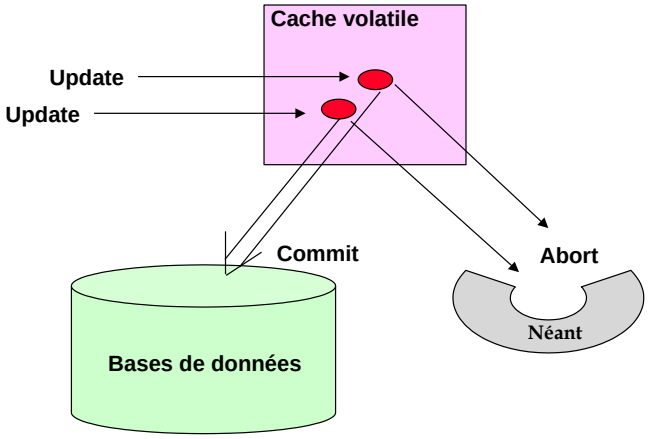

Figure XVI.32 — Mouvements de données entre cache et mémoire stable

6.4.3 Journal des mises à jour

A un instant donné, l'état de la base est déterminé par l'état de la mémoire stable et l'état du cache. En effet, des mises à jour ont été effectuées et ne sont pas encore reportées sur disque. Certaines effectuées par des transactions venant juste d'être validée peuvent être en cours de report. Il faut cependant garantir la non-perte de mise à jour des transactions commises en cas de panne. Si le système reporte des pages dans la mémoire stable avant validation d'une transaction, par exemple pour libérer de la place dans le cache, il faut être capable de défaire les reports de pages contenant des mises à jour de transactions annulées.

La méthode la plus classique pour permettre la validation atomique, l'annulation et la reprise de transaction consiste à utiliser des journaux [Verhofstad78]. On distingue le **journal des images avant** et le **journal des images après**, bien que les deux puissent être confondus dans un même journal.

> **Notion XVI.22 : Journal des images avant (*Before image log*)**
>
> Fichier système contenant d'une part les valeurs (images) avant modification des pages mises à jour, dans l'ordre des modifications avec les identifiants des transactions modifiantes, ainsi que des enregistrements indiquant les débuts, validation et annulation de transactions.

Le journal des images avant est utilisé pour **défaire** les mises à jour d'une transaction (*undo*). Pour cela, il doit être organisé pour permettre d'accéder rapidement aux enregistrements correspondant à une transaction. Un fichier haché sur l'identifiant de transaction (TrId) sera donc opportun.

Notion XVI.23 : Journal des images après (*After image log*)

Fichier système contenant d'une part les valeurs (images) après modifications des pages mises à jour, dans l'ordre des modifications avec les identifiants des transactions modifiantes, ainsi que des enregistrements indiquant les débuts, validation et annulation de transactions.

Le journal des images après est utilisé pour **refaire** les mises à jour d'une transaction (*redo*). Comme le journal des images après, il doit être organisé pour permettre d'accéder rapidement aux enregistrements correspondant à une transaction. Un fichier haché sur l'identifiant de transaction (TrId) sera donc opportun.

En guise d'illustration, la figure XVI.33 représente un enregistrement d'un journal contenant à la fois les images avant et après. Les enregistrements sont précédés d'une lettre R (Redo) pour les images après, U (Undo) pour les images avant, et T (Transaction) pour les changements d'états des transactions.

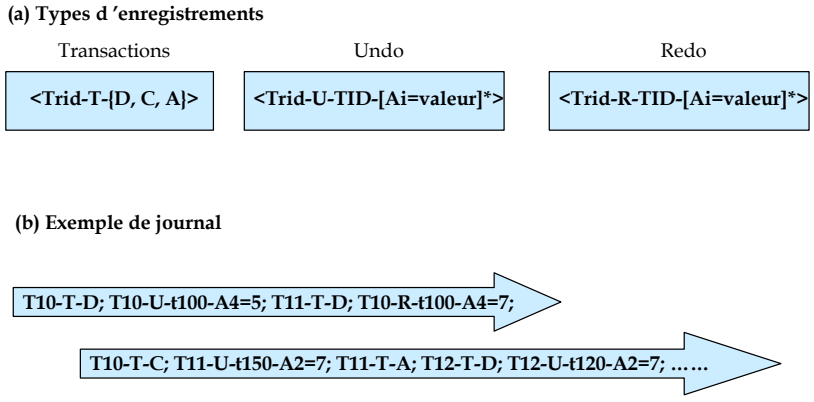

Figure XVI.33 — Enregistrements dans le journal global

Comme indiqué ci-dessus, les modifications sont tout d'abord exécutées dans des caches en mémoire. Ces caches sont **volatils**, c'est-à-dire perdus lors d'une panne. Ce n'est bien souvent que lors de la validation que les mises à jour sont enregistrées dans le journal et dans la base. Afin d'être capable d'annuler une transaction dans tous les cas, il faut écrire les enregistrements dans le journal avant de reporter le cache dans la base, comme illustré figure XVI.34. L'ordre dans lequel les opérations doivent être accomplies est indiqué sur la figure. Les règles suivantes sont souvent conseillées [Bernstein87] :

1. Avant d'écrire une page modifiée en mémoire stable, il faut enregistrer son image avant dans le journal (pour pouvoir défaire) ainsi que son image après (pour pouvoir refaire). Cette règle est connue sous le nom de **journalisation avant écriture** (*log ahead rule*).

2. Toutes les pages modifiées en mémoire volatile par une transaction doivent être écrites en mémoire stable, donc sur disque, avant la validation de la transaction. Cette dernière règle est connue sous le nom de **validation après écriture** (*commit after rule*).

L'application de ces deux règles conduit naturellement à enregistrer journal puis page modifiée sur disques soit à chaque mise à jour, soit en fin de transaction avant le commit effectif. Elles sont donc très limitatives. En conséquence, la première est généralement suivie pour éviter de ne pouvoir défaire des transactions non validées ou refaire des validées. La seconde peut être relaxée avec quelques précautions. Dans certains cas, par exemple si les journaux sont doublés ou si la base est doublée par une base miroir, ces deux règles peuvent être remplacées par des règles plus souples.

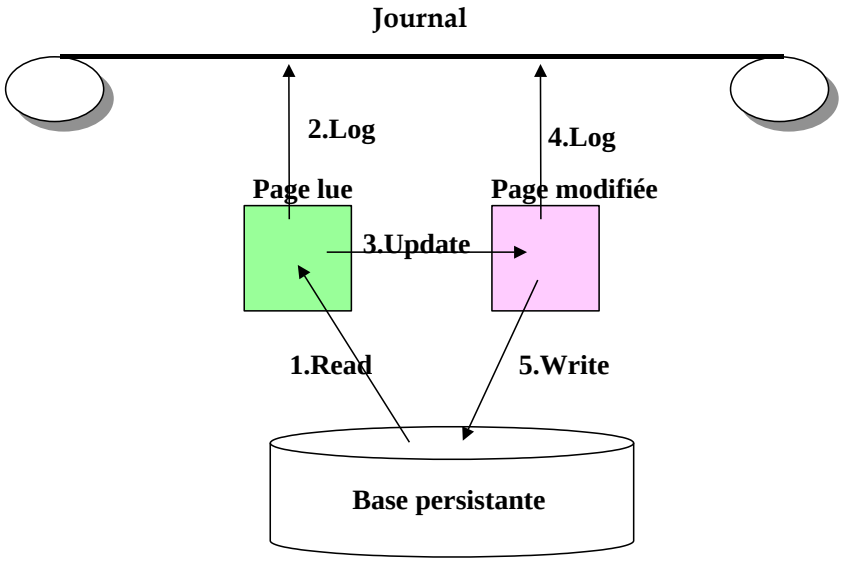

Figure XVI.34 — Ordre d'enregistrement des mises à jour

L'utilisation d'un journal coûte très cher : chaque mise à jour nécessite a priori trois entrées-sorties. Afin d'améliorer les performances, les enregistrements du journal sont généralement gardés dans un tampon en mémoire et vidés sur disques lorsque le tampon est plein. Malheureusement, il faut écrire le journal avant d'écrire dans la base pour pouvoir défaire ou refaire les mises à jour de la mémoire stable. La technique utilisée pour résoudre ce problème consiste à bloquer ensemble plusieurs validations de transactions.

> **Notion XVI.24 : Validation bloquée (*Blocked commit*)**
>
> Technique consistant à valider plusieurs transactions ensemble pour pouvoir écrire ensemble les enregistrements dans le journal.

Ainsi, le système peut attendre qu'un tampon du journal soit plein avant de le vider. Lorsqu'une transaction commet, si le tampon n'est pas plein, elle attend que d'autres transactions effectuent des mises à jour pour remplir le tampon. Si le système est peu actif, un délai maximal (*time out*) permet de forcer le vidage du tampon journal. Soulignons que le journal est en général compressé, pour réduire sa taille au maximum et aussi maximiser le nombre de transactions commises simultanément.

Un dernier problème concernant le journal est celui de sa purge. En effet, on ne peut enregistrer indéfiniment les enregistrements étudiés sur un disque dans un fichier haché. Même avec du hachage extensible, le fichier risque de devenir important et difficile à gérer. En conséquence, les systèmes changent périodiquement de fichier journal. Il est possible par exemple de tourner sur N fichiers hachés, en passant au suivant chaque fois que l'un est plein ou au bout d'une période donnée. Un fichier journal qui n'est plus actif est vidé dans une archive. Il sera réutilisé plus tard pour une nouvelle période de journalisation.

6.4.4 Sauvegarde des bases

En cas de perte de la mémoire stable, donc d'une panne des disques en général, il faut pouvoir retrouver une archive de la base détruite. Pour cela, des sauvegardes doivent être effectuées périodiquement, par exemple sur un disque séparé ou sur bandes magnétiques.

> **Notion XVI.25 : Sauvegarde (*Backup copy*)**
>
> Copie cohérente d'une base locale effectuée périodiquement alors que cette base est dans un état cohérent.

Une sauvegarde peut par exemple être effectuée en début de chaque semaine, de chaque journée, de chaque heure. La prise de sauvegarde pendant le fonctionnement normal du système est un problème difficile. Pour cela, un mécanisme de verrouillage spécifique — ou mieux, un mécanisme de multi-versions [Reed79] — peut être utilisé.

6.4.5 Point de reprise système

Après un arrêt normal ou anormal du système, il est nécessaire de repartir à l'aide d'un état machine correct. Pour cela, on utilise en général des points de reprise système.

> **Notion XVI.26 : Point de reprise système (*System checkpoint*)**
>
> Etat d'avancement du système sauvegardé sur mémoires secondaires à partir duquel il est possible de repartir après un arrêt.

Les informations sauvegardées sur disques comportent en général l'image de la mémoire, l'état des travaux en cours et les journaux. Un enregistrement « point de reprise système » est écrit dans le journal. Celui-ci est recopié à partir des fichiers le contenant. Lors d'une reprise, on repart en général du dernier point de reprise système. Plus ce point de reprise est récent, moins le démarrage est coûteux, comme nous le verrons ci-dessous.

7. VALIDATION DE TRANSACTION

Comme indiqué ci-dessus, la validation de transaction doit permettre d'intégrer toutes les mises à jour d'une transaction dans une base de données de manière atomique, c'est-à-dire que toutes les mises à jour doivent être intégrées ou qu'aucune ne doit l'être. L'atomicité de la validation de transaction rend les procédures d'annulation de transactions non validées simples. Le problème est

donc de réaliser cette atomicité. Plusieurs techniques ont été introduites dans les systèmes afin de réaliser une validation atomique. Elles peuvent être combinées afin d'améliorer la fiabilité [Gray81]. La plupart des SGBD combinent d'ailleurs les écritures en place et la validation en deux étapes.

7.1 Ecritures en place dans la base

Avec cette approche, les écritures sont effectuées directement dans les pages de la base contenant les enregistrements modifiés. Elles sont reportées dans la base au fur et à mesure de l'exécution des commandes Insert, Update et Delete des transactions. Ce report peut être différé mais doit être effectué avant la validation de la transaction. Comme vu ci-dessus, les mises à jour d'une transaction sont écrites dans le journal avant d'être mises en place dans la base de données. L'atomicité de la validation d'une transaction est réalisée par écriture d'un enregistrement dans le journal [Verghofstad78]. Les écritures restent invisibles aux autres transactions tant qu'elles ne sont pas validées ; pour cela, les pages ou les enregistrements sont verrouillés au fur et à mesure des écritures. La validation proprement dite consiste à écrire dans le journal un enregistrement « transaction validée ». Ensuite, les mises à jour sont rendues visibles aux autres transactions.

Dans tous les cas, la validation d'une transaction ayant mis à jour la base de données génère un nouvel état, ainsi que des enregistrements dans le journal (images des pages modifiées et enregistrement transaction validée), comme indiqué figure XVI.35. L'annulation d'une transaction ayant mis en place des mises à jour dans la base est une procédure difficile. Elle s'effectue à partir du journal des images avant. L'annulation nécessite le parcours du journal à l'envers, c'est-à-dire en reculant dans le temps.

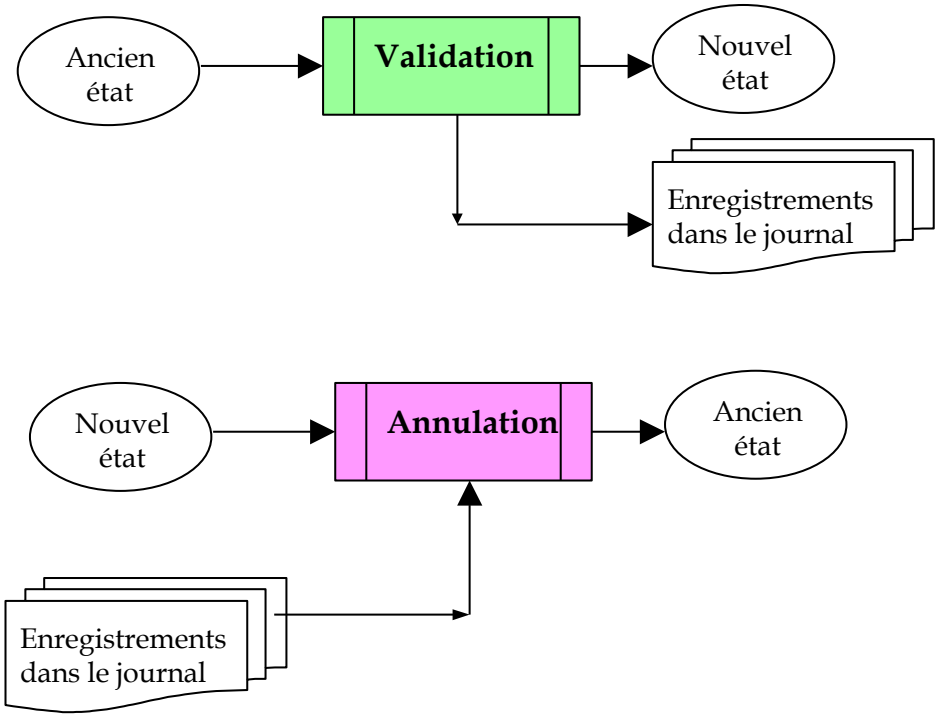

Figure XVI.35 — Principes de la validation et de l'annulation avec journal

7.2 Ecritures non en place

Avec cette approche, les pages modifiées sont recopiées dans de nouvelles pages en mémoire et écrites dans de nouveaux emplacements dans la base. L'atomicité de la validation d'une transaction peut être réalisée par la **technique des pages ombre** [Lorie77]. Les pages nouvelles séparées et attachées à la transaction modifiante sont appelées pages différentielles. Les pages anciennes constituent les pages ombre. Avant toute lecture, il faut alors que le SGBD consulte les pages différentielles validées et non encore physiquement intégrées à la base. Cela conduit à alourdir les procédures de consultation et accroître les temps de réponse.

Une solution plus efficace a été proposées dans [Lampson76]. Elle consiste à réaliser une intégration physique atomique par **basculement des tables des pages** (voir figure XVI.34). Pour cela, chaque fichier de la base de données possède un descripteur pointant sur la table des pages du fichier. Quand une transaction désire modifier un fichier, une copie de la table des pages est effectuée et les adresses des pages modifiées sont positionnées à leurs nouvelles valeurs, de sorte que la copie de la table des pages pointe sur la nouvelle version du fichier (anciennes pages non modifiées et nouvelles pages modifiées). La validation consiste alors simplement à mettre à jour le descripteur du fichier en changeant l'adresse de la table des pages.

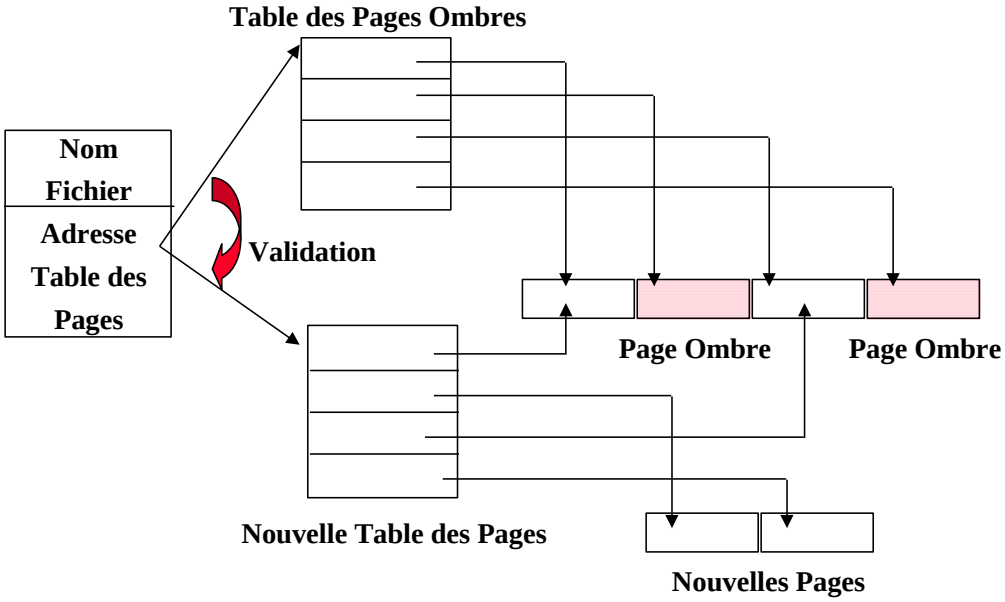

Figure XVI.36 — Validation par basculement de la table des pages

7.3 Validation en deux étapes

Dans la plupart des systèmes, un ensemble de processus collabore à la vie d'une transaction. Afin de coordonner la décision de valider une transaction, un **protocole de validation en deux étapes** est généralement utilisé. Ce protocole a été proposé dans un contexte de système réparti [Lampson76, Gray78] mais est aussi utile dans un contexte centralisé multi-processus. La validation en deux étapes peut être perçue comme une méthode de gestion des journaux. Lors de la première étape, les images avant et après sont enregistrées dans le journal si bien qu'il devient

ensuite possible de valider ou d'annuler la transaction quoi qu'il advienne (excepté une perte du journal) ; cette étape est appelée préparation à la validation. La seconde étape est la réalisation de la validation ou de l'annulation atomique, selon que la première étape s'est bien ou mal passée. La preuve de correction d'un tel protocole déborde le cadre de cet ouvrage et pourra être trouvée dans [Baer81].

Le protocole de validation en deux phases coordonne l'exécution des commandes `COMMIT` par tous les processus coopérant à l'exécution d'une transaction. Le principe consiste à diviser la validation en deux phases. La phase 1 réalise la préparation de l'écriture des résultats des mises à jour dans la BD et la centralisation du contrôle. La phase 2, réalisée seulement en cas de succès de la phase 1, intègre effectivement les résultats des mises à jour dans la BD répartie. Le contrôle du système est centralisé sous la direction d'un processus appelé **coordinateur**, les autres étant des **participants**. Nous résumons ces concepts ci-dessous.

> **Notion XVI.27 : Protocole de validation en deux étapes (*Two Phase Commit*)**
>
> Protocole permettant de garantir l'atomicité des transactions dans un système multi-processus ou réparti, composé d'une préparation de la validation et de centralisation du contrôle, et d'une étape d'exécution.

> **Notion XVI.28 : Coordinateur de validation (*Commit Coordinator*)**
>
> Processus d'un système multi-processus ou réparti qui dirige le protocole en centralisant le contrôle.

> **Notion XVI.29 : Participant à validation (*Commit Participant*)**
>
> Processus d'un système multi-processus ou réparti qui exécute des mises à jour de la transaction et obéit aux commandes de préparation, validation ou annulation du coordinateur.

Le protocole de validation en deux étapes découle des concepts précédents. Lors de l'étape 1, le coordinateur demande aux autres sites s'ils sont prêts à commettre leurs mises à jour par l'intermédiaire du message PREPARER (en anglais *PREPARE*). Si tous les participants répondent positivement (prêt), le message VALIDER (en anglais *COMMIT*) est diffusé : tous les participants effectuent leur validation sur ordre du coordinateur. Si un participant n'est pas prêt et répond négativement (KO) ou ne répond pas (*time out*), le coordinateur demande à tous les participants de défaire la transaction (message ANNULER, en anglais *ABORT*).

Le protocole nécessite la journalisation des mises à jour préparées et des états des transactions dans un journal local à chaque participant, ceci afin d'être capable de retrouver l'état d'une transaction après une éventuelle panne et de continuer la validation éventuelle. Le protocole est illustré figure XVI.37 dans le cas favorable où un site client demande la validation à deux sites

serveurs avec succès. Notez qu'après exécution de la demande de validation (VALIDER), les participants envoient un acquittement (FINI) au coordinateur afin de lui signaler que la transaction est maintenant terminée.

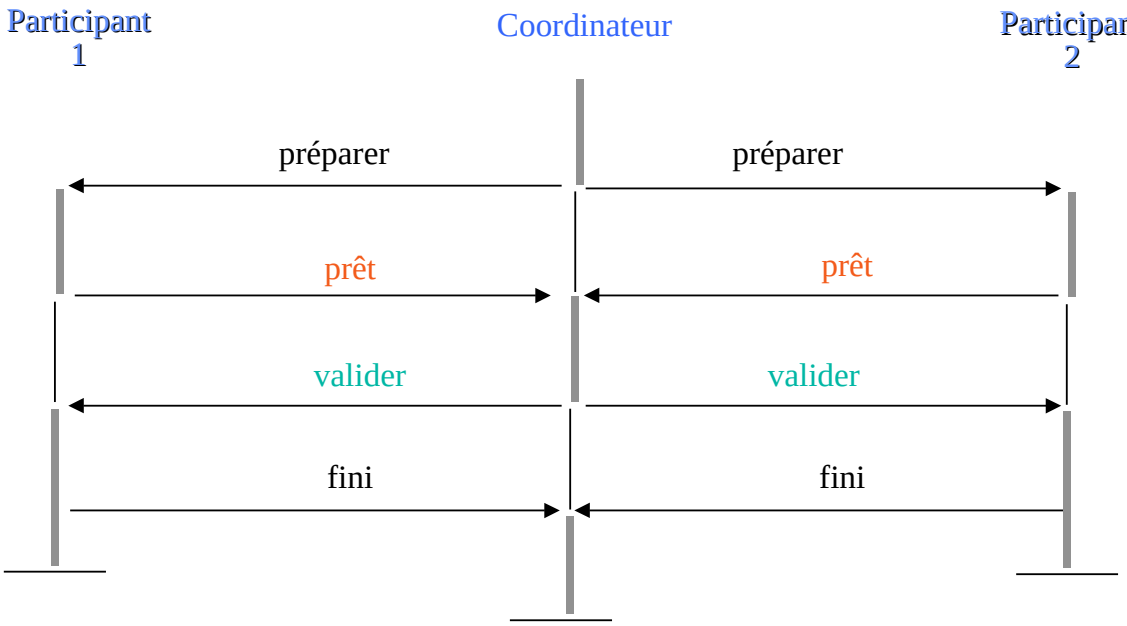

Figure XVI.37 — Validation en deux étapes avec succès

Le cas de la figure XVI.38 est plus difficile : le participant 2 est en panne et ne peut donc répondre au message de demande de préparation. Une absence de réponse est assimilée à un refus. Le coordinateur annule donc la transaction et envoie le message ANNULER. Le participant 1 annule la transaction. Le participant 2 l'annulera lorsqu'il repartira après la panne, une transaction non prête étant toujours annulée.

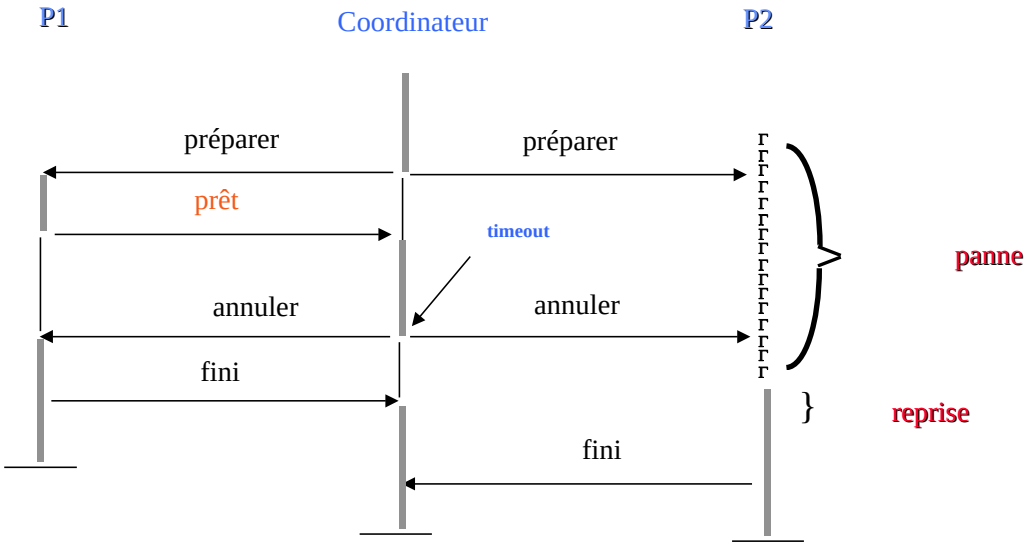

Figure XVI.38 — Validation en deux étapes avec panne totale du participant 2

Le cas de la figure XVI.39 est encore plus difficile : le participant 2 tombe en panne après avoir répondu positivement à la demande de préparation. Le coordinateur envoie donc le message COMMIT qui n'est pas reçu par le participant 2. Heureusement, celui-ci a dû sauvegarder l'état de la sous-transaction et ses mises à jour dans un journal fiable sur disque. Lors de la procédure de reprise, le journal sera examiné. La transaction étant détectée prête à commettre, son état sera demandé au coordinateur (ou a un participant quelconque) par un message appelé STATUS. En réponse à ce message, la validation sera confirmée et les mises à jour seront appliquées à partir du journal. Les deux sous-transactions seront donc finalement validées.

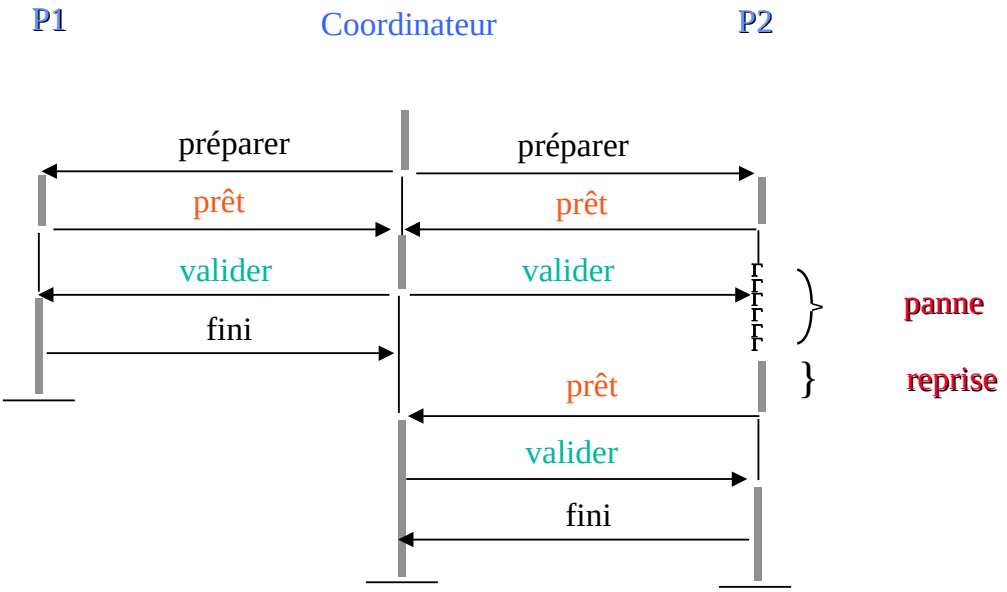

Figure XVI.39 — Validation en deux étapes avec panne partielle du participant 2

Au-delà du protocole en deux étapes, il existe des protocoles en trois étapes qui évitent les blocages du protocole précédent en cas de panne du coordinateur. Le plus courant est le protocole dit d'*abort* supposé, qui en cas de panne du coordinateur suppose l'abandon de la transaction. Ceci est même possible en deux phases [Mohan86].

Le protocole en deux étapes a été standardisé par l'ISO. TP est le protocole standard proposé par l'ISO dans le cadre de l'architecture OSI afin d'assurer une validation cohérente des transactions dans un système distribué. Le protocole est arborescent en ce sens que tout participant peut déclencher une sous-transaction distribuée. Un site responsable de la validation de la sous-transaction est choisi. Un coordinateur est responsable de ses participants pour la phase 1 et collecte les PREPARE. Il demande ensuite la validation ou l'*abort* selon la décision globale à un site appelé ***point de validation*** qui est responsable de la phase 2 : c'est le point de validation qui envoie les COMMIT aux participants. L'intérêt du protocole, outre l'aspect hiérarchique, est que le point de validation peut être différent du coordinateur : ce peut être un nœud critique dans le réseau dont la présence à la validation est nécessaire. La figure XVI.40 illustre ce type de protocole hiérarchique avec point de validation.

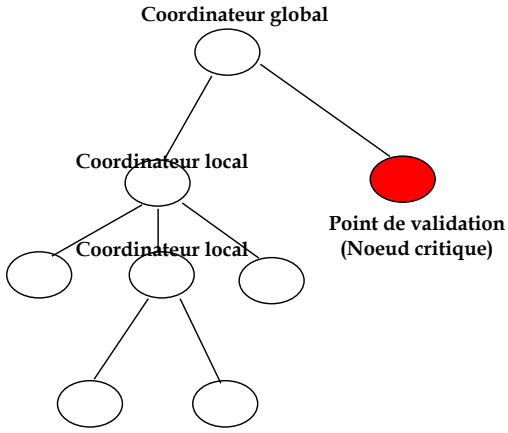

Figure XVI.40 — Le protocole hiérarchique avec point de validation TP

8. LES PROCEDURES DE REPRISE

Dans cette section, nous examinons les procédures de reprises. Après quelques considérations générales, nous introduisons divers protocoles de reprise possibles [Bernstein87].

8.1 Procédure de reprise

On appelle **procédure de reprise** la procédure exécutée lors du redémarrage du système après un arrêt ou une panne. Plusieurs types de reprises sont distingués et un schéma de principe possible pour chaque type est étudié ci-dessous. Ces procédures consistent à repartir à partir du journal et éventuellement de sauvegardes, pour reconstituer une base cohérente en perdant le minimum du travail exécuté avant l'arrêt. Afin de préciser les idées, nous définissons la notion de procédure de reprise.

> **Notion XVI.30 : Procédure de reprise (*Recovery procedure*)**
>
> Procédure système exécutée lors du redémarrage du système ayant pour objectif de reconstruire une base cohérente aussi proche que possible de l'état atteint lors de la panne ou de l'arrêt du système.

La reprise normale ne pose pas de problème. Elle a lieu après un arrêt normal de la machine. Lors de cet arrêt, un point de reprise système est écrit comme dernier enregistrement du journal. La reprise normale consiste simplement à restaurer le contexte d'exécution sauvegardé lors de ce point de reprise. Une telle procédure est exécutée lorsque le dernier enregistrement du journal est un point de reprise système.

8.2 Reprise à chaud

La reprise après une panne du système est appelée **reprise à chaud**. Rappelons qu'une panne système entraîne la perte de la mémoire centrale sans perte de données sur mémoires secondaires. Dans ce cas, le système doit rechercher dans le journal le dernier point de reprise système et

restaurer l'état machine associé. Le journal est alors traité en avant à partir du point de reprise afin de déterminer les transactions validées et celles non encore validées (appelées respectivement gagnantes et perdantes dans [Gray81]). Puis le journal est traité en sens inverse afin de défaire les mises à jour des transactions non validées (les perdantes). Une telle procédure est illustrée figure figure XVI.41. Elle est exécutée lors des redémarrages du système quand le dernier enregistrement du journal n'est pas un point de reprise et que l'opérateur ne signale pas une perte de mémoire secondaire.

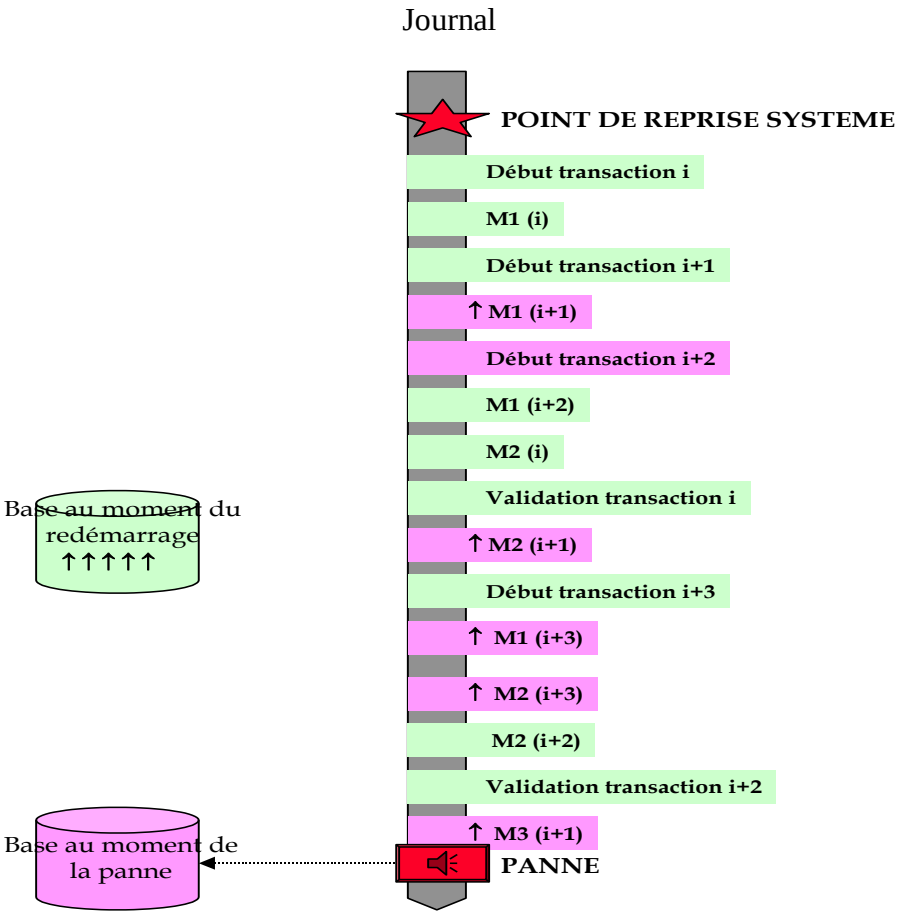

Figure XVI.41 — Exemple de reprise à chaud

Les transactions i et (i+2) sont gagnantes alors que (i+1) et (i+3) sont perdantes. Mj(i) désigne la j^e^ mise à jour de la transaction i. Les mises à jour défaites sont marquées par une flèche.

8.3 Protocoles de reprise à chaud

Lors d'une reprise à chaud, il est généralement nécessaire de défaire (*Undo*) des transactions non commises et refaire (*Redo*) des transactions commises. Cependant, différents protocoles sont applicables selon les cas indiqués figure XVI.42. Les conditions indiquées en colonne portent sur les mises à jour des transactions commises, celles en ligne portent sur les mises à jour des transactions non commises au moment de la panne. Les actions nécessaires sur les transactions

exécutées depuis le dernier point de reprise système sont indiquées au carrefour de la ligne et de la colonne, quand les deux conditions sont vérifiées.

Non commises → Commises ↓	Pas de mise à jour en base	Certaines mises à jour en base
Certaines mise à jour en base	No Undo, Redo (1)	Undo, Redo (2)
Toute mise à jour en base	No Undo, No Redo (3)	Undo, No Redo (4)

Figure XVI.42 — Protocoles de reprise à chaud possibles

Les renvois numériques du tableau ci-dessus correspondent aux protocoles suivants :

1. Le **protocole Refaire sans Défaire (*No Undo, Redo*)** est applicable dans le cas où l'on est sûr que toute transaction non commise n'a aucune mise à jour installée en base permanente. L'application d'un tel protocole nécessite donc le report des mises à jour d'une transaction simultanément ou après l'écriture de l'enregistrement COMMIT pour cette transaction dans le journal. La simultanéité nécessite un matériel spécial capable d'assurer l'atomicité de plusieurs mises à jour simultanées de disques différents. Différer les mises à jour en base après la validation relâche la règle de la validation après écritures vue ci-dessus.

2. Le **protocole Défaire et Refaire (*Undo, Redo*)** est le plus classique. Il est appliqué dans le cas où certaines mises à jour de transactions non commises peuvent être enregistrées dans la base et certaines mises à jour de transactions commises peuvent être perdues dans la base. Il est général et est le plus fréquemment utilisé. Il ne nécessite pas de différer les mises à jour en base permanente.

3. Le **protocole Défaire sans Refaire (*Undo, No Redo*)** est applicable dans le cas où l'on est sûr que toute transaction commise a toutes ses mises à jour reportées en base permanente. Ce peut être vrai si le système respecte la règle de validation après report des écritures vue ci-dessus.

4. Le **protocole Ni Refaire Ni Défaire (*No Undo, No Redo*)** est rarement applicable. Il nécessite que toute transaction commise ait toutes ses mises à jour en base permanentes et que toute transaction non commise n'ait aucune de ses mises à jour en base permanentes. Il faut pour cela un matériel spécifique capable d'exécuter toutes les écritures à la fois et de garantir l'atomicité de ces écritures multiples.

8.4 Reprise à froid et catastrophe

La reprise après une panne de mémoire secondaire est plus difficile. Ce type de reprise est souvent appelé **reprise à froid**. Une telle reprise est exécutée lorsqu'une partie des données est perdue ou lorsque la base est devenue incohérente. Dans ce cas, une sauvegarde cohérente de la base ainsi que le journal des activités qui ont suivi sont utilisés afin de reconstruire la base actuelle. Il suffit pour cela d'appliquer les images après à partir de la sauvegarde et du point de reprise associé. Lorsque le journal a été parcouru en avant jusqu'à la fin, la reprise à froid enchaîne en général sur une reprise à chaud.

Une **panne catastrophique** survient quand tout ou partie du journal sur mémoire secondaire est perdu. Certains systèmes, tel système R, permettent de gérer deux copies du journal sur des mémoires secondaires indépendantes afin de rendre très peu probable un tel type de panne. Dans le cas où cependant une telle panne survient, il n'existe guère d'autre solution que d'écrire des transactions spéciales chargées de tester la cohérence de la base en dialoguant avec les administrateurs et de compenser les effets des mises à jour malheureuses.

9. LA MÉTHODE ARIES

ARIES (*Algorithm for Recovery and Isolation Exploiting Semantics*) [Mohan92] est une des méthodes de reprise de transactions des plus efficaces. Elle est à la base des algorithmes de reprises implémentés dans de nombreux systèmes, dont ceux des SGBD d'IBM. Réalisée dans le cadre du projet de SGBD extensible d'IBM Starburst, cette méthode s'est imposée comme une des meilleures méthodes intégrées.

9.1 Objectifs

Les objectifs essentiels initiaux de la méthode étaient les suivants [Mohan92] :

- **Simplicité**. Vu la complexité du sujet, il est essentiel de viser à la simplicité de sorte à être applicable en vraie grandeur et à garantir des algorithmes robustes. C'est d'ailleurs vrai pour tous les algorithmes, au moins en système.

- **Journalisation de valeur et d'opération**. Classiquement, ARIES permet de défaire et refaire une mise à jour avec les images avant et après. Au-delà, la méthode intègre la possibilité de journaliser des opérations logiques, comme incrément et décrément d'un compte. Cela permet le verrouillage sémantique avec des modes d'opérations sophistiqués, pour supporter des exécutions à opérations commutatives non sérialisables au niveau des lectures et écritures. Ceci est important pour permettre une meilleure concurrence, comme vu ci-dessus. Une des difficultés dans ce type de journalisation par opération est de bien associer un enregistrement du journal avec l'état de la page qui lui correspond : ceci est accompli astucieusement avec un numéro de séquence d'enregistrement journal (*Log Sequence Number*, *LSN*) mémorisé dans la page. Une autre difficulté est due aux pannes pendant la reprise : il faut pouvoir savoir si l'on a refait ou non une opération. Ceci est accompli en journalisant aussi les mises à jour effectuées

pendant la reprise par des enregistrements de compensation (*Compensation Log Records, CRLs*).

- **Gestion de mémoire stable et du cache flexible**. La mémoire stable est gérée efficacement. Différentes stratégies de report du cache sont utilisables. Reprise et verrouillage sont logiques par nature ; cela signifie que la reprise reconstruit une base cohérente qui n'est pas forcément physiquement identique à la base perdue (pages différentes), mais aussi que les verrouillages sont effectués à des niveaux de granularités variables, en utilisant le verrouillage d'intention vu ci-dessus.

- **Reprise partielle de transactions**. Un des objectifs de la méthode est de pouvoir défaire une transaction jusqu'au dernier point de sauvegarde. C'est important pour pouvoir gérer efficacement les violations de contraintes d'intégrité et l'utilisation de données périmées dans le cache.

- **Reprise granulaire orientée page**. La méthode permet de reprendre seulement une partie de la base, la reprise d'un objet (une table par exemple) ne devant pas impliquer la reprise d'autres objets. La reprise est orientée page en ce sens que la granularité de reprise la plus fine est la page : si une seule page est endommagée, il doit être possible de la reconstruire seule.

- **Reprise efficace et rapide**. Il s'agit bien sûr d'avoir de bonnes performances, à la fois pendant l'exécution normale de transactions et pendant la reprise. L'idée est de réduire le nombre de pages à réécrire sur disques et le temps unité central nécessaire à la reprise.

Outre ces quelques objectifs, ARIES en a atteint de nombreux autres tels que l'espace disque minimal nécessaire en dehors du journal, le support d'objets multi-pages, la prise de points de reprise système efficaces, l'absence de verrouillage pendant la reprise, le journal limité en cas de pannes multiples lors de la reprise, l'exploitation du parallélisme possible, etc. [Mohan92].

9.2 Les structures de données

ARIES gère **un journal unique**. Le format d'un enregistrement est représenté figure XVI.43.

LSN	*Log Sequence Number* : Adresse relative du premier octet de l'enregistrement dans le log, utilisé comme identifiant de l'enregistrement (non mémorisé).
TYPE	*Record type* : Indique s'il s'agit d'un enregistrement de compensation, d'un enregistrement de mise à jour normal, d'un enregistrement de validation à deux étapes (prêt, commise) ou d'un enregistrement système (*checkpoint*).
PrevLSN	*Previous LSN* : LSN de l'enregistrement précédent

	pour la transaction s'il y en a un.
TransID	*Transaction ID* : Identifiant de la transaction qui a écrit l'enregistrement.
PageID	*Page ID :* Identifiant de la page à laquelle les mises à jour de l'enregistrement s'appliquent, si elle existe. Présent seulement pour les enregistrements de mise à jour ou de compensation.
UndoNxtLSN	*Undo next LSN* : LSN de l'enregistrement suivant à traiter pendant la reprise arrière. Présent seulement pour les enregistrements de compensation.
Data	*Data :* Données décrivant la mise à jour qui a été effectuée, soit l'image avant et l'image après, soit l'opération et son inverse si elle ne s'en déduit pas. Présent seulement pour les enregistrements de mise à jour ou de compensation qui n'ont que les informations pour refaire.

Figure XVI.43 — Format des enregistrements du journal

Pour chaque page de données, ARIES nécessite **un seul champ par page nommé page_LSN** qui mémorise le numéro LSN du dernier enregistrement du journal qui concerne cette page.

En plus, ARIES gère une **table des transactions** utilisée pendant la reprise pour mémoriser l'état de validation des transactions actives (Prête ou Non dans le protocole à deux phases), et enfin une **table des pages sales**, c'est-à-dire modifiée en cache et ne correspondant plus à la version sur mémoire stable.

9.3 Aperçu de la méthode

ARIES garantit l'atomicité et la durabilité des transactions en cas de panne de processus, de transaction, du système ou de mémoire secondaire. Pour cela, ARIES utilise un journal et applique les règles de **journalisation avant écriture** (*Write-Ahead Loging, WAL*) et de **validation après écriture** (*Write Before Commit, WBC*) vues ci-dessus. ARIES journalise les mises à jour effectuées par les transactions sur chaque page. Il journalise aussi les états prêts et validés des transactions gérées par un protocole de validation à deux étapes, et les points de reprise système (*system checkpoints*).

En plus, ARIES journalise les mises à jour effectuées pour défaire les transactions, à la fois pendant le fonctionnement normal du système (par exemple, suite à une violation d'intégrité) et

pendant la reprise. Ces enregistrements spéciaux de restauration sont appelés des **enregistrements de compensation** (*Compensation Log Records CLR*). Comme tous les enregistrements, chacun pointe sur l'enregistrement précédent du journal pour la même transaction (PrevLSN). Chaque enregistrement de compensation pointe aussi par UndoNxtLSN sur le prédécesseur de l'enregistrement juste défait. Par exemple, si une transaction a écrit les enregistrements normaux 1, 2, 3, puis 3' et 2' pour compenser respectivement 3 et 2, alors 3' pointe sur 2 et 2' sur 1. Cela permet de retrouver ce qui a déjà été défait. Si une panne survient pendant une reprise arrière d'une transaction, par exemple après exécution de 2', grâce au journal on retrouvera 2' puis 1 et l'on saura qu'il suffit de défaire 1. Ainsi, les enregistrements de compensation CLR gardent trace des progrès des reprises de transaction et permettent d'éviter de défaire ce qui a déjà été défait. Ils permettent aussi de refaire des opérations logiques grâce à la trace gardée.

ARIES marque chaque page avec le dernier LSN lui correspondant dans le journal. Ainsi, il est possible de retrouver très rapidement le dernier enregistrement du journal concernant cette page. Cela permet de connaître précisément l'état de la page, particulièrement de savoir que toutes les mises à jour ayant un LSN plus petit ont été correctement enregistrées. Cela est un point clé pour les performances de la reprise : on évite ainsi de ré-appliquer des mises à jour non perdues.

ARIES utilise des **points de reprise système périodique** marquant des points de bon fonctionnement dans le journal. Un enregistrement point de reprise identifie les transactions actives, leurs états, le LSN de leur plus récent enregistrement dans le journal, et aussi les pages modifiées et non reportées sur disques dans le cache (pages sales).

La procédure de reprise détermine le dernier point de reprise et, dans **une première passe d'analyse**, balaye le journal. Grâce aux LSN des pages sales figurant dans l'enregistrement point de reprise, le point de reprise exact du journal pour la passe suivante de réfaction est déterminé. L'analyse détermine aussi les transactions à défaire. Pour chaque transaction en cours, le LSN de l'enregistrement le plus récent du journal est déterminé. Il est mémorisé dans la table des transactions introduites ci-dessus.

Une deuxième passe de reconstruction est ensuite effectuée, durant laquelle ARIES répète l'histoire mémorisée dans le journal et non enregistrée sur mémoire stable. Ceci est fait pour toutes les transactions, y compris celles qui n'avaient pas été validées au moment de la panne. L'objectif est de rétablir l'état de la base au moment de la panne. Un enregistrement du journal est appliqué en avant (refait) sur une page si son LSN est supérieur à celui de la page, ce qui réduit le nombre de réfactions nécessaires. Durant cette phase, les verrous pour les transactions prêtes à valider au moment de la panne sont restaurés, afin de pouvoir les relancer.

Une troisième passe de réparation consiste à défaire les transactions perdantes, c'est-à-dire celles non validées ou non prêtes à valider au moment de la panne. Ceci s'effectue par parcours du journal en arrière depuis le plus grand LSN restant à traiter pour les transactions perdantes. Si l'enregistrement du journal est un enregistrement normal, les données d'annulation (image avant ou opération inverse) sont appliquées. L'enregistrement suivant à traiter pour la transaction est alors déterminé en regardant le champ PrevLSN. S'il s'agit d'un enregistrement de compensation, il est simplement utilisé pour déterminer l'enregistrement suivant à traiter mémorisé dans UndoNxtLSN, puisque les enregistrements de compensation ne sont jamais compensés.

En résumé, la méthode comporte donc trois passes : analyse, reconstruction en avant et réparation en arrière. Grâce aux chaînages intelligents des enregistrements du journal et à la mémorisation du numéro de séquence du dernier enregistrement pertinent au niveau de chaque page, la méthode évite de refaire et défaire des mises à jour inutilement. Elle intègre aussi la possibilité d'annulation logique ou physique d'opérations, ce qui permet des verrouillages en modes variés exploitant la commutativité. La méthode a été implémentée dans de nombreux systèmes industriels.

10. LES MODELES DE TRANSACTIONS ETENDUS

Nous avons ci-dessus étudié les techniques implémentées dans les SGBD pour assurer l'atomicité des transactions. Ces techniques sont aujourd'hui bien connues et opérationnelles pour les transactions en gestion. Cependant, les techniques de gestion de transactions atomiques sont trop limitées dans le cas de transactions longues que l'on rencontre en conception (CAO, CFAO), en production de documents (PAO) et plus généralement dans les domaines techniques nécessitant des objets complexes. Aussi différents modèles de transactions ont-ils été proposés, cherchant à permettre une plus grande flexibilité que le tout ou rien. Ces modèles étendent plus ou moins les propriétés ACID vues ci-dessus.

10.1 Les transactions imbriquées

Le modèle des **transactions imbriquées** est dû à [Moss85]. Il étend le modèle plat, à un seul niveau, à une structure de transaction à niveaux multiples. Chaque transaction peut être découpée en sous-transactions, et ce récursivement.

> **Notion XVI.31 : Transaction imbriquée (*Nested transaction*)**
>
> Transaction récursivement décomposée en sous-transactions, chaque sous-transaction et la transaction globale étant bien formée, c'est-à-dire commençant par BEGIN et se terminant par COMMIT.

On obtient ainsi un arbre de transactions (voir figure XVI.44). Une transaction fille est lancée par sa mère par exécution d'une commande *BEGIN* ; elle rend un compte-rendu à sa mère qui se termine après toutes ses filles. De manière classique, chaque (sous-)transaction se termine par *COMMIT* ou *ABORT* et est une unité atomique totalement exécutée ou pas du tout. Chaque (sous-)transaction peut être refaite (*REDO*) ou défaite (*UNDO*) par un journal hiérarchisé.

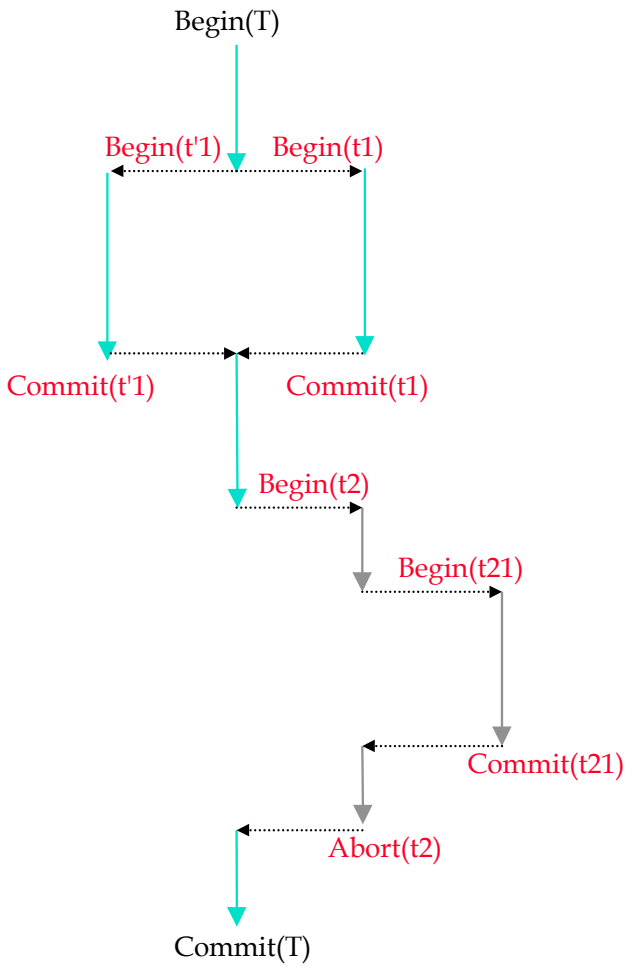

Figure XVI.44 — Exemple de transactions imbriquées

Les règles de validation et d'annulation résultent du découpage logique d'un travail en sous-tâche :

- L'annulation d'une transaction mère implique l'annulation de toutes ses filles, et ce récursivement.

- La validation d'une transaction fille est conditionnée par celle de ses ancêtres, et ce récursivement.

En conséquence, bien que chaque transaction se termine par *COMMIT* ou *ABORT*, la validation n'est confirmée que lorsque tous les ancêtres ont validé. Le **modèle fermé** [Moss85] garantit l'atomicité de la transaction globale. Donc un échec d'une sous-transaction implique une annulation de la mère. Ceci est très limitatif, mais conserve les propriétés ACID entre transactions globales. Au contraire, le **modèle ouvert** relâche l'atomicité de la transaction globale.

Du point de vue de la concurrence, chaque (sous-)transaction applique le verrouillage deux phases. Les verrous sont hérités dans les deux sens : lorsqu'une transaction fille réclame un verrou

tenu par un de ses ancêtres, elle n'est pas bloquée ; lorsqu'une transaction fille valide, les verrous qu'elle a acquis sont transmis à sa mère. Deux (sous-)transactions peuvent donc se bloquer voir s'interbloquer si le verrou n'est pas tenu par un ancêtre commun. Ceci complique les détections de verrous mortels.

Le **modèle ouvert** relâche donc la nécessité d'atomicité des transactions globales. Alors, une (sous-)transaction peut être annulée alors que sa mère est validée. Le travail est donc seulement partiellement réalisé. Ceci nécessite l'introduction de **transactions de compensation** [Korth90]. Par exemple, la compensation d'une transaction achetant de la marchandise consiste à rendre cette marchandise. Une (sous-)transaction de compensation défait donc une (sous-)transaction précédemment exécutée. Une variante est une **transaction de complément**. Ces transactions de complément viendront plus tard refaire les portions de travail non validées. Par exemple, le complément d'une transaction de mise à jour d'un ouvrage qui a échoué sur le chapitre II est une transaction qui met à jour le chapitre II. Finalement, dès que l'on relâche l'atomicité, il est nécessaire d'introduire de telles transactions dépendant de la sémantique de l'application.

Ces principes ont conduit à définir un modèle de transactions imbriquées ouvert très flexible [Weikum92]. Outre l'atomicité, ce modèle relâche en plus le principe d'isolation en laissant les résultats des sous-transactions commises visibles aux transactions concurrentes. La reprise nécessite alors de gérer des sphères d'influence de transactions, généralisation de la notion de traînée déjà vue. La sémantique des opérations est aussi intégrée pour gérer les commutativités d'opérations typées. De tels modèles commencent à être implémentés dans les SGBD classiques. Leurs limites et intérêts restent encore mal connus.

10.2 Les sagas

Les sagas [Garcia-Molina87] ont été introduites pour supporter des transactions longues sans trop de blocages. Une saga est une séquence de sous-transaction ACID {T1,, …, Tn} devant s'exécuter dans cet ordre, associée à une séquence de transactions de compensation {CT1,, …, CTn}, Cti étant la compensation de Ti. La saga elle-même est une transaction longue découpée en sous-transactions. Tout abandon d'une sous-transaction provoque l'abandon de la saga dans sa totalité. La figure XVI.45 représente deux exécutions possibles d'une saga.

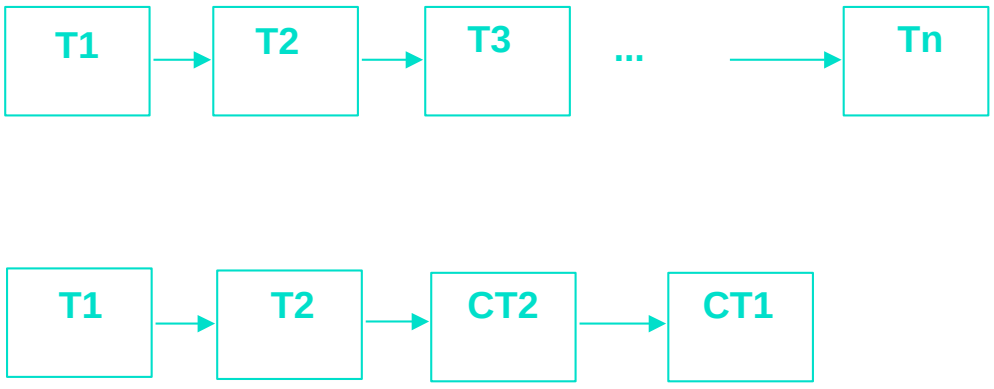

Figure XVI.45 — Exemples d'exécution de sagas

L'intérêt des sagas est de pouvoir relâcher le principe d'isolation. En effet, chaque transaction composante relâche ses verrous dès qu'elle est terminée. Une autre saga peut alors voir les résultats. L'annulation de la saga doit donc provoquer l'annulation de l'autre saga. Pour cela, il suffit d'enchaîner les transactions de compensation.

10.3 Les activités

La généralisation des sagas conduit au **modèle d'activités**, proche des modèles de *workflow*. L'idée est d'introduire un langage de contrôle de transactions permettant de définir des travaux sous la forme d'une collection d'étapes avec enchaînements conditionnels, les étapes étant des transactions et les travaux des activités. Les propriétés souhaitables d'une activité sont de pouvoir garder un contexte d'exécution persistant, progresser par le biais de transactions ou reculer par le biais de compensation, essayer des alternatives et plus généralement introduire un flot de contrôle dépendant des succès et échecs des transactions composantes. Il faut aussi pouvoir être capable de différencier les échecs système des échecs de programmes, les premiers étant corrigés par le système, les seconds par l'activité.

Un exemple typique d'activité est la réservation de vacances. Cette activité est par exemple composée de trois transactions, la première T1 ayant une alternative T1' :

1. T1 : réservation avion alternative T1' : location voiture
2. T2 : réservation hôtel
3. T3 : location voiture

De plus, chaque transaction est associée à une transaction de compensation CT1, CT2 et CT3 permettant d'annuler son effet.

Cette idée a conduit à concevoir des modèles et des langages de contrôle d'activités [Dayal91, Wachter92] et à implémenter des moniteurs d'activités. Un tel langage permet de contrôler les transactions par des ordres du type *If abort, If commit, Run alternative, Run compensation*, etc. Le moniteur contrôle les transactions et exécute les programmes de ce langage. On obtient ainsi des systèmes de gestion de la coopération très flexibles, proches des *workflows* [Rusinkiewicz95].

10.4 Autres modèles

Un modèle basé sur des **transactions multi-niveaux** a été introduit par [Weikum91, Beeri88]. Ce modèle est une variante du modèle des transactions imbriquées dans lequel les sous-transactions correspondent à des opérations à des niveaux différents d'une architecture en couches. Le modèle permet de considérer des opérations au niveau N qui se décompose en séquences d'opérations au niveau N - 1. Par exemple, le crédit d'un compte en banque se décompose au niveau inférieur (SQL) en une sélection (`SELECT`) et une mise à jour (`UPDATE`). Ceux-ci se décomposent à leur tour en lectures et écritures de pages au niveau système. Il est alors possible de décomposer une exécution simultanée de transactions en exécution (ou histoire) au niveau N, N - 1, etc. La méthode traditionnelle de gestion de concurrence introduite ci-dessus ne considère que le niveau le

plus bas, des lectures et écritures de pages. La sérialisabilité d'une exécution simultanée multi-niveaux est alors définie comme la sérialisabilité à chaque niveau avec des ordres de sérialisation compatibles entre eux. En considérant la sémantique des opérations à chaque niveau, il devient possible d'exploiter la commutativité pour tolérer des exécutions simultanées qui ne seraient pas valides si l'on ne considérait que le niveau le plus bas. Des contrôleurs de transactions peuvent être définis aux différents niveaux. Il en résulte moins de reprise de transactions.

Le modèle *split-join* [Pu88] est une autre variante des transactions imbriquées. A l'image des *split* et *join* de processus sur Unix, il est possible avec ce modèle de diviser une transaction dynamiquement en deux sous-transactions ou de réunir deux sous-transactions en une. Des règles strictes sont imposées sur les sous-transactions pour garantir la sérialisabilité globale.

L'introduction d'un **arbre de versions d'objet** introduit une autre dimension aux modèles de transaction (voir figure XVI.46). La gestion de versions autorise les mises à jour simultanées d'un objet. Pour manipuler les versions, deux opérations sont introduites : *Check-out* pour sortir un objet de la base dans un espace privé et *Check-in* pour le réintroduire dans la base. Les objets versionnables sont verrouillés pendant tout le temps de la sortie de la base dans des modes spécifiques **dérivation partagée** ou **dérivation exclusive**. La dérivation exclusive ne permet que la sortie d'une version à un instant donné alors que la dérivation partagée permet plusieurs sorties simultanées. La figure XVI.47 donne les compatibilités de ces modes.

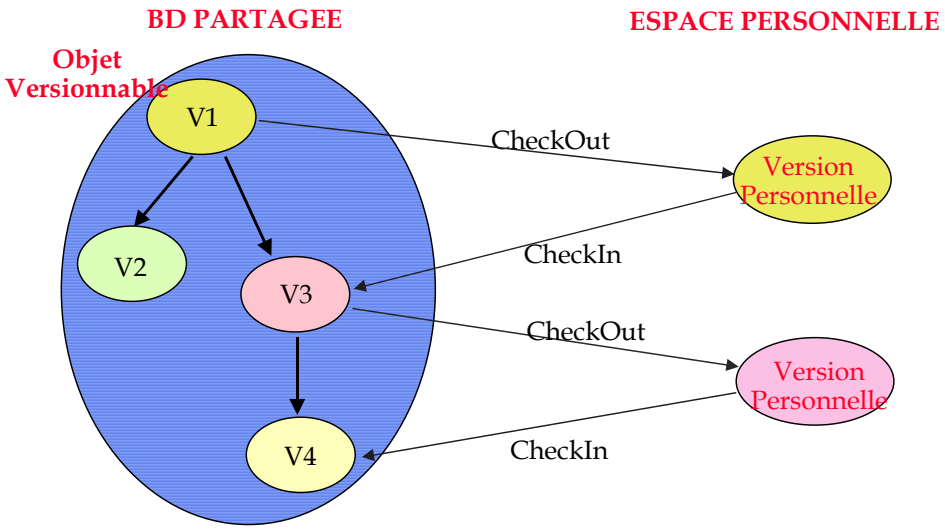

Figure XVI.46 — Exemple d'objet versionnable et d'évolution

	Lecture	Ecriture	Dérivation Partagée	Dérivation Exclusive
Lecture	1	0	1	1
Ecriture	0	0	0	0
Dérivation Partagée	1	0	1	0
Dérivation Exclusive	1	0	0	0

Figure XVI.47 — Compatibilité des opérations de lecture, écriture et dérivation

Lors de la réinsertion d'une version dans la base (*Check-in*), le graphe de versions est mis à jour et la nouvelle version ajoutée comme une fille de celle dont elle dérive. Les versions sont généralement maintenues en différentiel, c'est-à-dire que pour un nœud donné de l'arbre, seuls les attributs ou pages modifiés par rapport au nœud père sont maintenus. Le problème qui se pose est de faire converger les versions. Si deux versions dérivant d'un ancêtre commun n'ont pas de données communes modifiées, une fusion automatique peut être réalisée par intégration de toutes les mises à jour effectuées à l'ancêtre. Sinon, il y a nécessité de définir des procédures de réconciliation de versions, voire d'une intervention manuelle effectuant des choix.

Au-delà de ces modèles fort nombreux, des formalismes de compréhension et description ont été proposés. ACTA [Chrysanthis90] est un cadre pour spécifier et raisonner sur la structure des transactions et leur comportement. Il permet d'exprimer la sémantique des interactions entre transactions en termes d'effets sur la validation ou l'annulation des autres transactions, et sur l'état des objets et des synchronisations nécessaires. Il permet ainsi de décrire et de raisonner sur un modèle d'activité ou de transaction.

11. SECURITE DES DONNEES

Dans cette section, nous abordons brièvement les problèmes de sécurité, de plus en plus importants avec, par exemple, l'utilisation des bases de données pour le commerce électronique. Soulignons que les techniques de sécurité [Russell91] ne doivent cesser de progresser pour ne pas s'exposer à être dépassées par les escrocs.

11.1 Identification des sujets

Un système de bases de données doit assurer la sécurité des données qu'il gère, c'est-à-dire que les opérations non autorisées ou mal intentionnées doivent être rejetées. Assurer la sécurité nécessite tout d'abord une identification des utilisateurs, encore appelés **sujets**. Les sujets effectuent des **opérations** sur les **objets** protégés qui les subissent. Dans les systèmes de base de données, les sujets sont les utilisateurs devant les terminaux et les objets les données protégées avec différents niveaux de granularité possibles. Les sujets doivent tout d'abord être **identifiés**.

> **Notion XVI.32 : Identification (*Identification*)**
>
> Procédé consistant à associer à un sujet un nom ou un numéro qui le désigne de manière unique.

Un premier moyen de violer la sécurité des données est pour un sujet de se faire passer pour un autre. Afin d'éviter cela, un procédé d'**authentification** des sujets est introduit.

> **Notion XVI.33 : Authentification (*Authentification*)**
>
> Procédé permettant de vérifier qu'un sujet est bien qui il prétend être.

Le procédé d'authentification le plus courant est l'utilisation de **mots de passe**. Un mot de passe est une chaîne de caractères en principe connue du sujet et de l'objet seuls, que le sujet doit fournir pour être authentifié. Les mots de passe ne doivent pas être facilement retrouvables, par exemple à partir d'un dictionnaire par essais exhaustifs. Il est donc souhaitable de choisir des mots de passe longs (plus de 7 caractères), comportant un mixage de caractères numériques, alphanumériques et de contrôle. D'autres procédés plus sophistiqués et plus sûrs sont l'utilisation de questionnaires, l'exécution d'algorithmes connus seulement par l'objet et le sujet, l'utilisation de badges, de cartes à puces, d'empreintes digitales ou d'empreintes de la rétine. Ces derniers procédés nécessitent un périphérique spécialisé.

La définition des sujets ne pose a priori pas de problème : tout utilisateur est un sujet. Cependant, il est utile de considérer des **groupes** d'utilisateurs.

> **Notion XVI.34 : Groupe d'utilisateurs (*User group*)**
>
> Ensemble d'utilisateurs ayant chacun nom et mot de passe, désigné globalement par un nom, pouvant intervenir comme sujet dans le mécanisme d'autorisation.

La notion de groupe peut être étendue de manière hiérarchique, avec des sous-groupes. Dans ce cas, un groupe hérite de toutes les autorisations de ses antécédents hiérarchiques. Il est aussi possible de superposer plusieurs hiérarchies [Fernandez80]. Egalement, un sujet peut être un ensemble de transactions cataloguées ; l'accès à ces transactions doit alors être aussi protégé.

Un utilisateur peut joindre ou quitter un groupe. La dynamique des groupes lors des départs de personnes dans une entreprise peut être un problème : il faut par exemple enlever un utilisateur et le remplacer par un autre. Attribuer ou enlever de multiples droits à des groupes devient

rapidement difficile lors des changements de fonctions. Pour éviter ces problèmes, les SGBD ont introduit la notion de **rôle**.

> **Notion XVI.35 : Rôle (*Role*)**
>
> Ensemble de droits sur les objets caractérisé par un nom, pouvant intervenir comme sujet dans le mécanisme d'autorisation.

Ainsi, des droits sur les objets sont attribués aux rôles. Ceux-ci peuvent être ensuite affectés aux utilisateurs ou aux groupes d'utilisateurs. Les SGBD modernes sont capables de gérer à la fois des utilisateurs, des groupes et des rôles qui sont donc les sujets des mécanismes de contrôle d'autorisations.

11.2 La définition des objets

Le choix des objets à protéger est un des problèmes importants lors de la conception d'un module de protection pour un SGBD. Les systèmes anciens tels CODASYL et IMS restreignent les objets protégeables aux types d'objets : CODASYL permet de protéger par des clés n'importe quel niveau de type d'objet décrit dans le schéma (donnée élémentaire, article, fichier), alors qu'IMS permet de protéger des sous-ensembles de champs d'un segment. Au contraire, les systèmes relationnels permettent de protéger des ensembles d'occurrences de tuples définis par des prédicats, donc dépendants du contenu. Les systèmes objet permettent souvent de protéger à la fois des types et des instances.

Dans les systèmes relationnels, les **vues** jouent un rôle essentiel dans la définition des objets à protéger [Chamberlin75]. Tout d'abord, lorsqu'une vue est un objet autorisé à un usager, celui-ci ne peut accéder qu'aux tuples de cette vue. Ensuite, il est possible avec la notion de vue de définir des objets protégés à granularité variable. Une vue peut être une table de la base : un droit d'accès est alors attribué à un usager sur une relation entière. Une vue peut aussi être un ou plusieurs tuples extraits dynamiquement d'une ou de plusieurs tables : un droit d'accès est alors attribué sur une relation virtuelle résultant de l'évaluation d'une question. La technique de modification de questions [Stonebraker75] étudiée au chapitre IX traitant des vues permet alors de protéger efficacement les tuples ou attributs de tuples composant la vue. Une autre approche est d'utiliser le même langage pour définir les objets sur lesquels on autorise des opérations que celui pour exprimer les questions. Cette approche est celle de QBE où les questions et les autorisations sont exprimées de la même manière. Cela revient au même que l'utilisation de vues comme objets de protection, puisqu'une vue est définie comme une question.

Dans les systèmes objet, il est souvent souhaitable de protéger les types et même individuellement les opérations des types, mais aussi les instances. Les objets sont organisés en hiérarchie d'héritage : les instances d'une classe peuvent être perçues comme le dernier niveau de la hiérarchie. Il doit être possible d'ajouter des prédicats pour partitionner plus finement l'extension d'une classe en objets autorisés ou non. Généralement, un utilisateur d'une sous-classe hérite des droits de la classe de base. Dans le cas où des autorisations positives (droits d'accès) et négatives (interdictions d'accès) seraient attribuées, déterminer les autorisations d'une sous-classe devient

difficile, notamment en cas d'héritage multiple [Rabitti91]. Aujourd'hui, les SGBD objet restent pour la plupart encore assez faibles du point de vue de la sécurité.

11.3 Attribution et contrôle des autorisations : la méthode DAC

La méthode DAC (*Discretionary Access Control*) est la plus utilisée dans les SGBD. Elle consiste à attribuer aux sujets des droits d'opérations sur les objets et à les vérifier lors des accès. Un sujet authentifié peut exécuter certaines opérations sur certains objets selon les droits d'exécution accordés par les administrateurs du système ou plus généralement par d'autres sujets. Plus précisément, nous définirons le concept **d'autorisation** déjà utilisé informellement.

> **Notion XVI.36 : Autorisaton (*Authorization*)**
>
> Droit d'exécution d'une opération par un sujet sur un objet.

Les autorisations considérées sont en général positives : ce sont des accords de droits. Cependant, la défense américaine a aussi introduit dans ses standards des possibilités d'autorisations négatives, qui sont des interdictions d'accès. Cela pose des problèmes de conflits, les interdictions dominant en général les autorisations. Dans la suite, nous ne considérons que les autorisations positives.

L'attribution d'une autorisation peut dépendre :

- du sujet, par exemple de ses privilèges d'accès ou du terminal qu'il utilise,
- de l'objet, par exemple de son nom, son état actuel, son contenu ou sa valeur,
- de l'opération effectuée, par exemple lecture ou mise à jour,
- d'autres facteurs, par exemple de l'heure du jour ou de la date.

Il existe plusieurs manières de mémoriser les autorisations. Une première est l'utilisation de **matrices d'autorisations**. Une matrice d'autorisations est une matrice dont les lignes correspondent aux sujets et les colonnes aux objets, définissant pour chaque couple sujet-objet les opérations autorisées.

Considérons par exemple deux relations décrivant les objets NOM et RESULTAT d'étudiants et des sujets étudiants, secrétaires et professeurs. Les opérations que peuvent accomplir les sujets sur les objets nom et résultat sont lecture et écriture. Les autorisations peuvent être codées par deux bits, droit d'écriture puis droit de lecture. La figure XVI.48 représente la matrice correspondant à cet exemple, 0 signifiant accès interdit et 1 accès autorisé.

	NOM	RÉSULTAT
ÉTUDIANT	0 1	0 1
SECRÉTAIRE	1 1	0 1
PROFESSEUR	1 1	1 1

Figure XVI.48 — Exemple de matrice d'autorisations

Dans les SGBD, les définitions des sujets, objets et autorisations sont mémorisées dans la méta-base ou catalogue du système. En pratique, la matrice d'autorisation peut être stockée :

- par ligne — à chaque sujet est alors associée la liste des objets auxquels il peut accéder ainsi que les droits d'accès qu'il possède ;
- par colonne — à chaque objet associée la liste des sujets pouvant y accéder avec les droits d'accès associés ;
- par élément — à chaque couple sujet-objet sont associés les droits d'accès du sujet sur l'objet.

C'est cette dernière technique qui est retenue dans les SGBD relationnels. Les autorisations sont mémorisées dans une table DROITS(<Sujet>, <Objet>, <Droit>, <Donneur>), le donneur permettant de retrouver la provenance des droits.

L'**attribution de droits** aux sujets sur les objets est un autre problème important. Plus précisément, il est nécessaire de définir qui attribue les droits. Deux approches sont possibles. Soit, comme dans SQL2, le créateur d'un objet (une relation ou une vue par exemple) devient son propriétaire et reçoit tous les droits. Afin de passer et retirer les droits qu'il possède, il doit alors disposer de commandes du type [Griffiths76] :

- **GRANT** < types opérations > **ON** < objet >

  TO < sujet >

  [**WITH GRANT OPTION**]

  avec :

  <type opération> ::= **ALL PRIVILEGES** | <action>$^+$

  <action> ::= **SELECT** | **INSERT**

  | **UPDATE** [(<nom de colonne>$^+$)]

  | **REFERENCE** [(<nom de colonne>$^+$)]

<sujet> ::=**PUBLIC** | <identifiant d'autorisation>

- **REVOKE** < types opérations > **FROM** < objet > **TO** < sujet >.

Soulignons que PUBLIC désigne l'ensemble des utilisateurs, alors qu'un identifiant d'autorisation peut désigner un utilisateur, un groupe ou un rôle. L'option WITH GRANT OPTION permet de passer aussi le droit de donner le droit.

Soit un groupe de sujets particuliers (les administrateurs des bases de données) possèdent tous les droits et les allouent aux autres par des primitives du même type. La première approche est décentralisée alors que la seconde est centralisée. Des approches intermédiaires sont possibles si l'on distingue plusieurs groupes de sujets avec des droits a priori [Chamberlin78].

11.4 Attribution et contrôle des autorisations : la méthode MAC

Un moyen plus simple de contrôler les autorisations consiste à associer un **niveau d'autorisation** aux sujets et aux objets.

> **Notion XVI.37 : Niveau d'autorisation (*Authorization level*)**
>
> Fonction mappant l'ensemble des sujets et utilisateurs sur les entiers [0-N] permettant à un sujet d'accéder à un objet si et seulement si : Niveau(Sujet) ≥ Niveau(Objet).

Un niveau d'autorisation est un nombre associé à chaque objet ou à chaque sujet. Il caractérise des niveaux de sensibilité des informations (Top secret, secret, confidentiel, non classé) et des niveaux de permissions des sujets vis à vis des objets. Un accès est autorisé si et seulement si le niveau du sujet accédant est supérieur ou égal au niveau de l'objet accédé. Le sujet domine alors le type d'information auquel il accède.

Considérons par exemple les sujets et objets introduits ci-dessus avec les niveaux d'autorisation : Etudiant (1), Secrétaire (2), Enseignant (3), Nom (1), Résultat (3). La matrice d'autorisation équivalente est représentée figure XVI.49.

	NOM	RÉSULTAT
ÉTUDIANT	1 1	0 1
SECRÉTAIRE	1 1	0 0
PROFESSEUR	1 1	1 1

Figure XVI.49 — Matrice d'autorisation équivalente aux niveaux indiqués

L'inconvénient de la solution niveau d'autorisation est que l'on perd la notion d'opération. Cependant, cette solution est simple à implanter et de plus elle permet de combiner les niveaux. En effet, si un sujet de niveau S_1 accède à travers une procédure ou un équipement de niveau S_2, on associera au sujet le niveau $S = \min(S_1, S_2)$. Par exemple, s'il existe un terminal en libre accès de niveau 1 et un terminal situé dans un bureau privé de niveau 3, un enseignant ne conservera ses privilèges que s'il travaille à partir du terminal situé dans le bureau. De plus, la méthode peut être étendue, avec des classes de niveaux partiellement ordonnées, vers un modèle plus complet de contrôle de flots d'informations. Elle peut aussi être combinée avec la méthode MAC.

11.5 Quelques mots sur le cryptage

Les données sensibles dans les bases peuvent être cryptées. Le cryptage est fondé sur des algorithmes mathématiques qui permettent de transformer les messages en rendant très difficile la découverte des transformations inverses. Plus précisément, on utilise en général des fonctions non facilement réversibles basées sur des divisions par des produits de grands nombres premiers. La recherche de la fonction inverse nécessite des décompositions en facteurs premiers très longues à réaliser. Les algorithmes de codage et décodage demandent donc l'attribution de clés de cryptage et de décryptage.

Les algorithmes symétriques utilisent **une seule clé secrète** pour le cryptage et le décryptage. La clé est par exemple un nombre de 128 bits. Un décodage d'un message crypté sans connaître la clé nécessite en moyenne quelques années avec des machines parallèles les plus puissantes. Le mécanisme est donc sûr, mais le problème est la gestion des clés qui doivent être échangées. L'application d'un tel algorithme aux bases de données nécessite son implémentation dans le SGBD et au niveau de l'utilisateur. Il faut en effet pouvoir décrypter au niveau du SGBD pour effectuer les recherches. Rechercher sur des données cryptées est très difficile puisque les chaînes de données codées dépassent en général un attribut. Le SGBD devra donc garder un catalogue des clés de cryptage/décryptage. Les utilisateurs doivent pouvoir accéder au catalogue pour retrouver leur clé secrète. Celui-ci devient un point faible du système.

Les algorithmes à **clés publiques et privées** proposent deux clés différentes, inverses l'une de l'autre. L'une permet de coder, l'autre de décoder. Les clés doivent être plus longues qu'avec les algorithmes à clés secrètes, car elles sont plus facilement cassables, c'est-à-dire qu'à partir d'une clé publique de 40 bits, on peut déduire la clé privée en l'essayant sur un message compréhensible en quelques heures de calcul d'une machine puissante. Il faut donc des clés de 512 bits et plus

pour obtenir une sécurité totale, ce qui peut poser des problèmes de législation. Avec de telles clés, le SGBD peut garder ses clés privées et publier ses clés publiques. L'utilisateur code les données avec une clé publique et les insère dans la base. Le SGBD décode avec la clé privée correspondante. Les résultats peuvent être envoyés codés avec une clé publique de l'utilisateur qui décode alors avec sa clé privée. Ce schéma est très sûr si les clés sont assez longues. Il est illustré figure XVI.50. Les algorithmes asymétriques tels que Diffie-Hellmann et RSA permettent ainsi des solutions très sûres, où SGBD et utilisateurs gardent leurs clés privées.

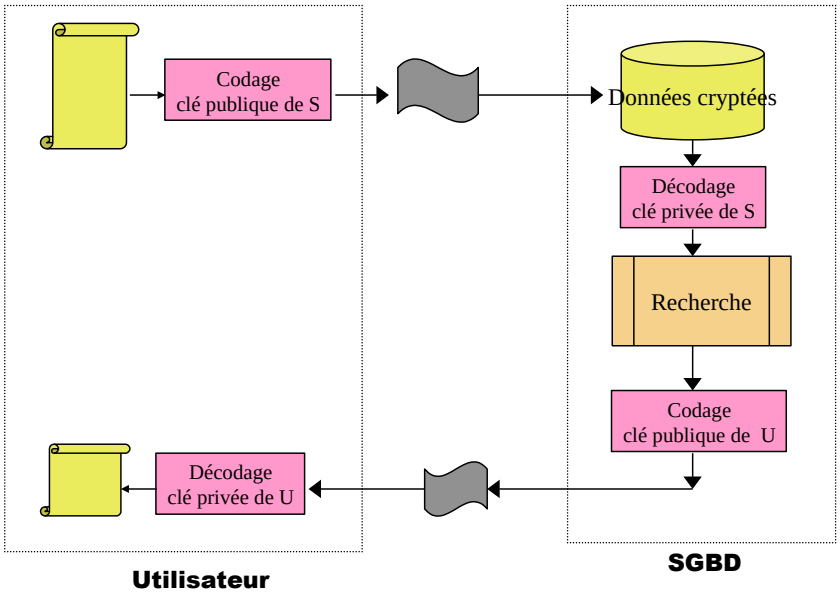

Figure XVI.50 — Fonctionnement avec clés asymétriques

(U désigne l'utilisateur et S le SGBD)

12. CONCLUSION ET PERSPECTIVES

Dans ce chapitre, nous avons abordé les problèmes de gestion de transactions. Nous avons étudié la théorie de la concurrence puis les algorithmes de contrôle. Ensuite, nous avons abordé la résistance aux pannes avec ses aspects validation et reprise. La méthode ARIES couplée au verrouillage deux phases et à la validation en deux étapes constitue une bonne référence que nous avons présentée.

Nous avons aussi abordé les modèles de transactions étendues, tels que les transactions imbriquées. Sur le contrôle de concurrence et sur les transactions étendues, de nombreux articles ont été publiés. Il est difficile d'y voir clair, mais nous avons synthétisé l'essentiel. Finalement, au-delà du verrouillage deux phases, de la validation en deux étapes et des procédures de reprise type ARIES, peu de choses sont implémentées dans les systèmes industriels. Une des raisons est sans doute la complexité des algorithmes très sensibles pour l'exploitation réelle des bases de données.

Plusieurs problèmes n'ont cependant pas été abordés, en particulier celui de la concurrence dans les index et celui de la sécurité dans les bases de données statistiques.

Les algorithmes de concurrence au niveau des index nécessitent des approches spécifiques du fait que l'index est un point de passage obligé pour toute les transactions accédant sur clé. On ne peut donc bloquer l'index pendant la durée de vie d'une transaction. La prise en compte des commutativités d'opérations telles que l'insertion et l'insertion avec reprise par des opérations logique comme la suppression est une direction de solution. Une autre voie repose sur une granularité de verrouillage très fine correspondant à une entrée d'index. Avec une telle granularité, dans un arbre B, il est possible de relâcher les verrous sur les entrées supérieures dès qu'une entrée inférieure est verrouillée et que l'on est sûr de ne pas devoir modifier l'entrée supérieure. Verrouiller efficacement des index nécessite à la fois une granularité fine et la prise en compte des commutativités d'opérations. Ceci complique la méthode de reprise. Notez qu'ARIES supporte ces points. Pour une introduction plus complète sur le contrôle de concurrence au niveau des index, vous pouvez consulter [Besancenot97].

L'objectif des bases de données statistiques est de fournir des résumés sur une population sans permettre à l'utilisateur de déduire des informations sur des individus particuliers. Ce type de protection est important, notamment dans le domaine médical et plus généralement pour le décisionnel. Il apparaît très difficile d'éviter les possibilités de déduction d'informations spécifiques à un individu en corrélant plusieurs résumés. Ainsi le chasseur de secrets reste-t-il une menace pour les bases de données statistiques [Denning80]. La seule manière de le perturber est sans doute de lui mentir un peu. Une vue d'ensemble du domaine est présentée dans un ouvrage assez ancien [Demillo78]. Le sujet n'est plus guère d'actualité, mais il peut le redevenir.

13. BIBLIOGRAPHIE

[Baer81] Baer J.-L., Gardarin G., Girault C., Roucairol G., « The Two-Step Commitment Protocol : Modeling, Specification and Proof Methodology », *5th Intl. Conf. on Software Engineering*, IEE Ed., San Diego, 1981.

Cet article modélise le protocole de validation en deux étapes par des réseaux de Petri. Il prouve partiellement la correction du protocole, c'est-à-dire que tous les sites prennent la même décision pour une transaction.

[Barghouti91] Barghouti N. S., Kaiser G. E., « Concurrency Control in Advance Database Applications » *ACM Computing Survey*, Vol. 23, N°3, pp. 270-317, Sept. 1991

Cet article fait le point sur les techniques de contrôle de concurrence avancées. Il rappelle les techniques traditionnelles vues ci-dessus et résume le verrouillage altruiste, la validation par clichés, les transactions multi-niveaux, le verrouillage sémantique, les sagas, etc.

[Bancilhon85] Bancilhon F., Korth H., Won Kim, « A Model of CAD Transactions », *11th Int. Conf. on Very Large data Bases*, Stockholm, Suède, Août 1985.

Cet article propose un modèle de transactions longues pour les bases de données en CAO. Il fut l'un des précurseurs des modèles de transactions imbriqués développés par la suite.

[Beeri88] Beeri C., Scheck H-J., Weikum G., « Multi-Level Transaction Management, Theoretical Art or Practical Need ? », *Proc. Intl. Conf. on Extending Database Technology* (EDBT), pp. 134-155, Venise, Mars 1988.

Cet article introduit la sérialisabilité à niveaux multiples présentée ci-dessus.

[Bernstein80] Bernstein P.A., Goodman N., « Timestamp-based Algorithm for Concurrency Control in Distributed Database Systems », *5th Intl. Conf. On Very Large data Bases*, Montreal, Oct. 1980.

Cet article introduit différents algorithmes d'estampillage de transactions dans le contexte de systèmes répartis.

[Bernstein87] Bernstein P.A., Hadzilacos V., Goodman N., *Concurrency Control and Recovery in Database Systems*, Addison-Weley, 1987.

Cet excellent livre de 370 pages fait le tour des problèmes de gestion de transactions dans les SGBD centralisés et répartis. Il présente notamment la théorie de base, le verrouillage, la certification, le multi-version, les protocoles de validation à deux et trois phases, la gestion de données répliquées.

[Bernstein90] Bernstein P.A., Hsu M., Mann B., « Implementing Recoverable Requests Using Queues », *ACM SIGMOD Int. Conf.*, ACM Ed., SIGMOD Record V. 19, N° 2, June 1990.

Cet article propose un protocole résistant aux pannes pour gérer les flots de requêtes de transactions entre clients et serveurs. Il discute une implémentation en utilisant des files d'attente persistantes récupérables après panne.

[Besancenot97] Besancenot J., Cart M., Ferrié J., Guerraoui R., Pucheral Ph., Traverson B., *Les Systèmes Transactionnels : Concepts, Normes et Produits*, Ed. Hermès, Paris, 1997.

Ce remarquable livre sur les systèmes transactionnels couvre tous les aspects du sujet : concepts de base, algorithmes de reprise, transactions réparties, duplication, modèles de transactions étendus, normes et standards, transactions dans les SGBD relationnels et objet.

[Bjork72] Bjork L.A., Davies C.T., « The Semantics of the Preservation and Recovery of Integrity in a Data System », *Technical Report TR-02.540,* IBM, Dec. 1972.

Un des tout premiers articles introduisant un concept proche de celui de transaction.

[Cart90] Cart M., Ferrié J., « Integrating Concurrency Control into an Object-Oriented Database System », *Proc. Intl. Conf. on Extending Database Technology*, EDBT, LCNS N° 416, pp. 367-377, Venise, Mars 1990.

Cet article présente l'intégration d'algorithmes de contrôle de concurrence avec prise en compte de la commutativité des opérations dans un SGBD objet.

[Chamberlin75] Chamberlin D.D., Gray J.N., Traiger I.L., « Views, Authorizations and Locking in a Relational Data Base System », *Proc. of ACM National Computer Conf.*, pp. 425-430, 1975

Cet article discute de l'utilisation des vues comme mécanisme d'autorisation et de verrouillage par prédicat dans un SGBD.

[Chamberlin78] Chamberlin D.D., Gray J.N., Griffiths P.P., Mresse M., Traiger I.L., Wade B.W., « Data Base System Authorization », in *Foundations of Secure Computation* [Demillo78], pp. 39-56.

Cet article discute des mécanismes d'autorisation dans un SGBD relationnel.

[Chrysanthis90] Chrystandis P.K., Ramamritham K., « ACTA : A Framework for Specifying and reasoning about Transaction Structure and Behavior », *ACM SIGMOD Intl. Conf. on Management of Data, SIGMOD Record* Vol. 19, N°2, pp. 194-203, Atlantic City, NJ, Juin 1990.

Cet article présente ACTA, un formalisme permettant de spécifier structure et comportement des transactions, en exprimant notamment la sémantique des interactions.

[Dayal91] Dayal U., Hsu M., Ladin R., « A Transactional Model for Long-Running Activities », Proc. of the 17th Intl. *Conf. on Very Large Data Bases*, Morgan Kaufmann Ed., pp. 113-122, Bacelonne, Sept. 1991.

Cet article décrit un modèle d'activités composées récursivement de sous-activités et de transactions. Le modèle définit la sémantique des activités et décrit une implémentation en utilisant des files résistantes aux pannes pour chaîner les activités.

[Denning80] Denning D.E., Schlörer J., « A Fast Procedure for Finding a Tracker in a Statistical Database », *ACM Transactions on Database Systems*, Vol. 5, N°1, pp. 88-102, Mars 1980.

Cet article présente une procédure rapide pour découvrir les caractéristiques d'un individu à partir de requêtes sur des ensembles d'individus.

[Demillo78] DeMillo R.A., Dobkin D.P., Jones A.K., Lipton R.J., *Foundations of Secure Computation*, Academic Press, 1978.

Ce livre présente un ensemble d'articles sur la sécurité dans les BD statistiques et les systèmes opératoires.

[Eswaran76] Eswaran K.P., Gray J.N., Lorie R.A., Traiger L.L., « The Notion of Consistency and Predicates Locks in a Database System », *Comm. of the ACM*, Vol. 19, N° 11, pp. 624-633, Nov. 1976.

Un des articles de base sur la notion de transaction et le principe du verrouillage deux phases. La notion de sérialisabilité et le verrouillage par prédicat sont aussi introduits par les auteurs qui réalisent alors la gestion de transactions dans le fameux système R.

[Fernandez80] Fernandez E.B., Summers R.C., Wood C., *Database Security and Integrity*, The Systems Programming Series, 1980.

Ce livre de 320 pages fait le tour des techniques d'integrité et de sécurité au sens large dans les bases de données.

[Garcia-Molina87] Garcia-Molina H., Salem K., « Sagas » *Proc. ACM SIMOD Intl. Conf. on Management of Data*, pp. 249-259, San Fransisco, 1987.

Cet article présente le modèle des sagas introduit ci-dessus.

[Gardarin76] Gardarin G., Spaccapietra S., « Integrity of Databases : A General Locking Algorithm with Deadlock Detection », *IFIP Intl. Conf. on Modelling in DBMS*, Freudenstadt, January 1976.

Cet article présente un algorithme de verrouillage multi-mode et un algorithme de détection du verrou mortel dans ce contexte.

[Gardarin77] Gardarin G., Lebeux P., « Scheduling Algorithms for Avoiding Inconsistency in Large Data Bases », *3rd Intl. Conf. on Very Large Data Bases*, IEEE Ed., Tokyo, 1977.

Cet article introduit pour la première fois la commutativité des opérations comme solution pour tolérer plus d'exécution sérialisable. Il présente un algorithme de verrouillage à modes multiples prenant en compte les commutativités possibles.

[Gardarin78] Gardarin G., « Résolution des Conflits d'Accès simultanés à un Ensemble d'Informations – Applications aux Bases de Données Réparties », Thèse d'État, Paris VI, 1978.

Cette thèse introduit différents algorithmes de verrouillage et de détection du verrou mortel, dont certains avec prise en compte de la commutativité des opérations, d'autres basés sur l'ordonnancement total ou partiel par estampillage. Ces deux résultats étaient nouveaux à cette époque. La méthode basée sur la commutativité des opérations avait été publiée auparavant au VLDB 1977 [Gardarin77]. Les algorithmes d'estampillage ont été un peu plus tard plus clairement présentés avec M. Melkanoff dans le rapport INRIA N° 113 et en parallèle par P. Bernstein [Bernstein80].

[Gray78] Gray J.N., « Notes on Database Operating Systems », in *Operating Systems – An Advanced Course*, R. Bayer Ed., Springer-Verlag, 1978.

Cet article est un autre tutorial sur la gestion de transactions, basé sur la réalisation du gestionnaire de transaction du système R.

[Gray81] Gray J., « The Transaction Concept : Virtues and Limitations », *Proc. of the 7th Intl. Conf. on Very Large Data Bases*, IEEE Ed., pp. 144-154, 1981.

Un autre tutorial sur la notion de transaction, discutant verrouillage, validation et procédures de reprise.

[Gray91] Gray J. Ed., *The Benchmark Handbook* , Morgan & Kaufman Pub., San Mateo, 1991.

Le livre de base sur les mesures de performances des SGBD. Composé de différents articles, il présente les principaux benchmarks de SGBD, en particulier le fameux benchmark TPC qui permet d'échantillonner les performances des SGBD en transactions par seconde. Les conditions exactes du benchmark définies par le « Transaction Processing Council *» sont précisées. Les benchmarks de l'université du Madisson, AS3AP et Catell pour les bases de données objets, sont aussi présentés.*

[Gray93] Gray J.N., Reuter A., *Transaction Processing : Concepts and Techniques*, Morgan Kaufman Ed., 1993.

Une bible de 1070 pages qui traite en détail tous les aspects des transactions.

[Griffiths76] Griffiths P.P., Wade B.W., « An Authorization Mechanism for a Relational Database System », *ACM Transactions on Database Systems*, Vol. 1, N°3, pp. 242-255, Sept. 1996.

Cet article décrit le mécanismes des droits d'accès avec opérations GRANT et REVOKE pour la première fois. L'implémentation du système R réalisée par les auteurs fut la première.

[Holt72] Holt R.C., « Some Deadlock Properties of Computer Systems », *ACM Computing Surveys*, Vol. 4, N°3, pp. 179-196, Sept. 1972.

Cet article discute des problèmes de deadlock *et introduit notamment le graphe d'allocation des ressources avec des algorithmes de détection associés.*

[Kaiser95] Kaiser G.E., « Cooperative Transactions for Multiuser Environments », *in Modern Database Systems*, Won Kim Ed., ACM Press, pp. 409-433, 1995.

Cet article est un tutorial sur les transactions coopératives de longue durée. Il passe en revue le modèle des versions, split-join*, de transactions imbriquées et de groupes.*

[Korth90] Korth H.F., Levy E., Silberschatz A., « A Formal Aproach to Recovery by Compensating Transactions », Proc. of 16th *Intl. Conf. on Very Large Data Bases*, Morgan Kaufman Ed., pp. 95-106, Brisbane, Australia, August 1990.

[Kung81] Kung H.T., Robinson J.T., « On Optimistic Method for Concurrency Control », *ACM Transaction on Database Systems* (TODS), Vol. 6, N°2, pp. 213-226, Juin 1981.

Cet article présente deux familles d'algorithmes optimistes de contrôle de concurrence basés sur la certification.

[Lampson76] Lampson B., Sturgis H., Crash Recovery in Distributed Data Storage System, *Xerox Technical Report*, Palo Alto, Xerox Research Center, 1976.

Cet article de base présente la première implémentation du protocole de validation en deux étapes. Lampson et Sturgis sont connus comme étant les inventeurs de ce fameux protocole.

[Lorie77] Lorie R.A., « Physical Integrity in a Large Segmented Database », ACM Transactions on Database Systems, Vol. 2, N°1, pp.91-104, Mars 1977.

Cet article décrit le système de stockage du système R et sa gestion de transaction et reprise.

[Mohan86] Mohan C., Lindsay B., Obermark R., « Transaction Management in the R* Distributed Database Management System, *ACM Trans. On Database Syst*., Vol. 11, N°4, Dec. 1986.

R est le prototype de SGBD réparti développé par IBM au début des années 80. Cet article présente la gestion de transactions distribuées, notamment les protocoles de validation à deux phases.*

[Mohan92] Mohan C., Haderle D., Lindsay B., Pirahesh H., Schwarz P., « ARIES : A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging », *ACM Transactions on Database Systems*, Vol. 17, N°1, pp. 94-162, Mars 1992.

Cet article présente la méthode ARIES décrite ci-dessus.

[Moss82] Moss J.E.B., « Nested transactions and Reliable Distributed Computing », *2nd Symposium on Reliability in Distributed Software and Database Systems*, pp. 33-39, IEEE Ed., Pittsburgh, 1982.

Cet article introduit les transactions imbriquées, inventées par Moss dans son PhD. Les transactions imbriquées, au départ fermées, ont été ouvertes (non-atomicité globale) et son implémentées dans des systèmes de plus en plus nombreux.

[Moss85] Moss J.E.B., *Nested Transactions : An Approach to Reliable Distributed Computing*, The MIT Press, Cambridge, Mass., 1985.

Ce livre, version améliorée du PhD de Moss, introduit et étudie en détail la théorie et la pratique des transactions imbriquées.

[Murphy68] Murphy J.E., « Ressource Allocation with Interlock Detection in a Multi-Task system », *Proc of AFIPS-FJCC Conf.*, Vol. 33, N° 2, pp. 1169-1176, 1968.

Cet article introduisit les graphes d'attente et l'un des premiers algorithmes de détection de *deadlock.*

[Özsu91] Özsu M.T., Valduriez P., *Principles of Distributed Database Systems,* Prentice Hall, Englewood Cliffs, New Jersey, 562 p., 1991.

Cet ouvrage est le livre de référence en anglais sur les bases de données réparties. Il couvre en particulier les aspects architecture, conception, contrôle sémantique, optimisation de requêtes, gestion de transactions, fiabilité et concurrence, bases de données fédérées. Chaque aspect est traité de manière très complète. Les algorithmes sont esquissés et une formalisation minimale est souvent introduite.

[Ozsu94] Ozsu T., « Transaction Models and Transaction Management in Object-Oriented Database Management Systems », in *Advances in Object-Oriented Database Systems,* pp. 147-184, A. Dogac et. Al. Ed., NATO ASI Series, Springer Verlag, Computer and System Sciences, 1994.

Ce remarquable article est un autre tutorial sur la gestion de transactions. Il couvre particulièrement bien les modèles de transactions étendus et l'influence de la commutativité des opérations.

[Pu88] Pu C., Kaiser G.E., Hutchinson N., « Split-Transactions for Open-Ended Activities » 14th *Intl. Conf. on Very Large data Bases*, pp. 26-37, Morgan Kaufman Ed., Los Angelès, 1998.

Cet article présente le modèle dynamique split et join évoqué ci-dessus pour les transactions imbriquées.

[Rabitti91] Rabitti F., Bertino E., Kim W., Woelk D., « A Model of Authorization for Next Generation Database Systems », *ACM Trans. On Database Systems,* Vol. 16, N°1, pp. 88-131, Mars 1991.

Cet article développe un modèle formel de sécurité de type DAC pour un SGBD objet. Il s'appuie sur l'expérience conduite avec le SGBD ORION.

[Reed79] Reed D.P., « Implementing Atomic Actions », *Proc. of the 7th ACM SIGOPS Symposium on Operating Systems Principles,* ACM Ed., Dec. 1979.

L'auteur a proposé un algorithme d'ordonnancement multi-version qu'il utilise pour implémenter des actions atomiques que l'on appelle aujourd'hui des transactions.

[Rosenkrantz78] Rosenkrantz D., Stearns R., Lewis P., « System Level Concurrency Control for Distributed Database Systems », *ACM Transactions on Database Systems,* Vol. 3, N° 2, pp. 178-198, Juin 1978.

Cet article présente les techniques de contrôle de concurrence WAIT-DIE et WOUND-WAIT décrites ci-dessus.

[Rusinkiewicz95] Rusinkiewicz M., Sheth A., « Specification and Execution of Transactional Workflows », in *Modern Database Systems*, Won Kim Ed., ACM Press, pp. 592-620, 1995.

Cet article discute les applications du concept de transaction aux activités de type workflow *qui nécessitent la coordination de tâches multiples.*

[Russell91] Russell D., Gangemi Sr. G.T., *Computer Security Basics*, O'Reilly & Associates, Inc., 1991.

*Un livre de base sur les techniques de sécurité, couvrant notamment les contrôles d'accès, les virus, le livre orange de l'armée américaine (*Orange Book*) et la sécurité dans les réseaux.*

[Skeen81] Skeen D., « Nonblocking Commit Protocols », *ACM SIGMOD Intl. Conf. on Management of Data*, Ann Arbor, pp. 133-142, ACM Ed., Mai 1981.

Des protocoles non bloquants, en particulier le protocole en trois étapes décrit ci-dessus, sont introduits dans cet article.

[Stonebraker75] Stonebraker M., « Implémentation of Integrity Constraints and Views by Query Modification », *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, San José, CA 1975.

Cet article propose de modifier les questions au niveau du source par la définition de vues pour répondre aux requêtes. La technique est formalisée avec le langage QUEL d'INGRES, où elle a été implémentée.

[Verhofstad78] Verhofstad J.S.M, « Recovery Techniques for database Systems, *ACM Computing Surveys*, Vol. 10, N°2, pp. 167-195, Juin 1978.

Un état de l'art sur les techniques de reprise dans les bases de données en réseaux et relationnelles.

[Wachter92] Wachter H., Reuter A., « The Contract Model », *in Database Transaction Models for Advanced Applications*, Morgan Kaufman Pub., 1992.

Cet article introduit les contrats comme un ensemble de transactions dirigées par un script. Il fut l'un des précusseurs des workflows.

[Weikum91] G. Weikum, « Principles and Realization Strategies of Multi-Level Transaction Management », *ACM Transactions on Database Systems*, Vol. 16, N°1, 1991.

Cet article décrit un protocole de gestion de transactions imbriquées et distribuées, avec des sphères d'influence définies pour chaque transaction : sphère de validation des sous-transactions devant être validées, sphère d'annulation des sous-transaction devant être annulées, ceci en cas de validation ou d'annulation de la transaction maître.

[Weihl88] Weihl W.E., « Commutativity-based Concurrency Control for Abstract Data Types », *IEEE Transactions on Computers*, Vol. 37, N°12, pp. 1488-1505, 1988.

Cet article propose des méthodes et algorithmes de contrôle de concurrence prenant en compte la commutativité des opérations pour gérer des types abstraits de donnés.