République **A**lgérienne **D**émocratique et **P**opulaire
Ministère de l'**E**nseignement **S**upérieur et de la **R**echerche **S**cientifique
Université **M**'hamed **B**ougara - **B**oumerdès

Faculté des Sciences
Département d'Informatique

MEMOIRE

Pour l'obtention du diplôme de MAGISTER

Spécialité : Systèmes Informatiques et Génie des Logiciels

Option : Spécification des Logiciels et Traitement de l'Information

Thème:

Proposition d'une solution de journalisation pour les structures de données distribuées et scalables(SDDS) de type hachage linéaire distribué (LH*)

Présenté et soutenu par

BOUCETTA Mohammed

Soutenu le : 09/10/2011.

Devant le jury composé de:

Mr MEZGHICE Mohamed	**MCA**	**UMBB**	**Président**
Mr BOULIF Menouar	**MCA**	**UMBB**	**Examinateur**
Mr RAHMOUNE Faiçal	**MCA**	**UMBB**	**Examinateur**
Mr HIDOUCI Khaled Walid	**MCA**	**ESI**	**Promoteur**

Année universitaire: 2010/2011

Sommaire

Liste des figures

Liste des algorithmes

Liste des tableaux

Liste des abréviations

2PL **:** Protocole de verrouillage à deux phases (two phases locking protocol).

ACID : Propriétés d'atomicité, de cohérence, d'isolation et de durabilité des transactions.

CC : Contrôleur de la concurrence.

GD : Gestionnaire de données.

GT : Gestionnaire de transactions.

LH : Hachage linéaire.

Oi(x) : Wi(x) ou Ri(x).

Redo : Journal des opérations à refaire.

RE(x) : La plus grande estampille des transactions ayant lu la donnée x.

Ri(x) : Opération de lecture de la donnée x par la transaction i.

SDDS: Scalable distributed data structures.

TP: Transaction processing.

Undo: journal des opérations à défaire.

WE(x) : La plus grande estampille des transactions ayant écrit dans la donnée x.

Wi(x) : Opération d'écriture de la donnée x par la transaction i.

Dédicaces

A ma chère mère pour son affection et son soutien indéfectible, qui ne cesse de prier pour nous,

A mon cher père qui m'a toujours encouragé et soutenu et qui fait tout pour nous rendre heureux,

A mes chers frères Madani, Elkhier, Abdelaziz et mes chères sœurs M'barka, Naanaa, Zahia, Fatima, Khadidja, Roumaissa et ma femme nadjiba et mes chers enfants Abdennour et Hinde.

A tous les membres de ma famille,

A tous mes amis,

A tous mes enseignants du primaire jusqu'à l'université,

Je dédie ce modeste travail.

Remerciements

Je tiens à remercier tout particulièrement mon encadreur de Mémoire, Monsieur W.K.Hidouci, Maître de conférences à l'ESI, pour sa grande disponibilité, son aide et ses conseils qu'ils m'ont apportés tout au long de ce travail.

Je suis profondément redevable à Pr. M.Mezghiche pour l'honneur qu'il m'a fait de présider le jury, ainsi qu'à Mr M.Boulif et Mr F.Rahmoune, pour avoir accepté d'examiner mon modeste travail.

Je voudrais aussi remercier Monsieur K.Badari Professeur de (U.Boumerdès), pour son aide et ses encouragements pour l'accomplissement de ce travail.

Mes remerciements s'adressent aussi à, Melle A.Badaoui, Houcine, Malik, pour leurs aides précieuses.

Finalement, je tiens à remercier tous ceux qui ont contribué de près ou de loin à l'élaboration de ce travail.

مُلخص

اقترحنا في هذا العمل المنجز حلاً يعتمد على نسخ العمليات المُنَفّذة والتي تخص الطريقة التوزيعية للبيانات *LH (التقسيم الخطي الموزع).

*LH عبارة عن صيغة من صيغ الـ SDDS (النظام البياني التوزيعي المتنامي)، بالتالي فهي تتضمن معالجة البيانات على الذاكرة الحية، فهي جد معرّضة لضياع ما تحويه من معلومات.

لهذا فقد اقترحنا حلاً يعتمد على إنشاء نسخة Undo/Redo لكل كتلة من العمليات الغير التامّة، وهذا لكل خادم (خانة) معني بإحدى هذه الكتل.

كما اقترحنا حلّين لتسيير نُسخ العمليات الخاصة بالتامة منها. الأول يستخدم نُسخة Redo عامّة على قرص صلب مركزي، و الثاني يستخدم نُسخة Redo على قرص كل خادم معني أيضًا.

طريقتنا المعتمِدة تقنية نسخ العمليات، ستسمح بردّ آخر حالة متجانسة للمعطيات بعد أي عطب، حتى ذاك الذي يكون أثناء العمليات المُركبة ونعني بهذا، الإنقسام والتوحد لخانات*LH .

كلمات مفتاحية :

نسخ العمليات، التقسيم الخطي الموزع، رد البيانات، حالة متجانسة للمعطيات، توحد الخانات، إنقسام الخانات، كتلة من العمليات.

Résumé

Le travail consiste à proposer une solution de journalisation adaptée à la méthode de distribution de données LH* (Distributed Linear Hashing).

LH* fait partie de la classe de SDDS (Scalable distributed data structures), et possède alors comme caractéristique la gestion des données en mémoire centrale. Donc elle très sensible aux pannes systèmes provoquant la perte de la mémoire centrale.

Nous avons proposé une solution où les journaux Undo/Redo des transactions actives sont gérés en mémoire centrale de chaque serveur.

Nous avons aussi proposé deux solutions pour la gestion de la journalisation des transactions validées, l'une utilisant un journal Redo global sur disque centralisé et l'autre utilisant des journaux Redo sur disques au niveau de chaque serveur.

Notre méthode de journalisation, permet le recouvrement du dernier état cohérent des données après n'importe quelle panne, même celles qui se produisant au milieu des opérations complexe telle que l'éclatement et la fusion de cases LH*.

MOTS-CLES

Journalisation, Hachage Linéaire Distribué (LH*), SDDS, Undo/Redo, Recouvrement, Etat cohérent, Eclatement de case, Fusion de cases, Transaction.

Abstract

This work consists in proposing a solution of logging adapted to the data distribution method LH* (Distributed Linear Hashing).

LH* is a part of SDDS class (Scalable distributed data structures), thus it has the characteristic of managing data in main memory. So, it is very sensitive to crash systems which cause the loss of main memory content.

We have proposed a solution where the Undo/Redo logs of the active transactions are managed in main memory of each server.

We have also proposed two solutions to log management of the committed transactions.

The first one consists of using a global Redo log on a centralized disc, and in the second solution we have used Redo log on discs in each LH* server.

Our method of logging, allows the recovery of the last coherent state of the data after any crash, even those occurring during the complex operations such as the splitting and the merging processes in insertions and deletions.

KEYWORDS

Logging, Distributed Linear Hashing (LH*), SDDS, Undo/Redo, Recovery, coherent State, Bucket Splitting, Bucket merging, Transaction.

Introduction générale

Pour supporter une grande charge ainsi que des tailles importantes de données, les systèmes de gestion de bases de données (SGBD) utilisent le parallélisme et la répartition. Les multi-ordinateurs (ou cluster) sont l'une des architectures utilisées pour implémenter les SGBD parallèles.

Parmi les méthodes de stockage qui sont utilisées dans les systèmes de multi-ordinateur, on a les Structures de Données Distribuées et Scalables (**SDDS**). Ces structures stockent les données dans les mémoires centrales des nœuds du multi-ordinateur. Ils assurent des temps d'accès beaucoup plus courts que pour des données stockées sur les disques, car avec les nouveaux réseaux (gigabits) l'accès à des mémoires distantes devient plus rapide que l'accès à un disque local. Les SDDS disposent aussi, en plus du traitement parallèle, une capacité de stockage potentiellement illimitée, et en plus de ça, elles offrent une disponibilité plus élevée pour exécuter des transactions d'une manière concurrente. Plusieurs **SDDSs** ont été proposées notamment ceux basées sur le hachage ; citons par exemple les familles : **LH*** (Linear Hashing) **[LIT99]**, **DDH** (Distributed Dynamic Hashing), **CTH** (Compact Trie Hashing) **[ZEG04,CHE06]**.

L'accès aux données d'un fichier **SDDS** passe par des opérations des transactions, ces transactions terminent leurs effets correctement (validation ou commit) ou les annulent carrément (Abort). L'exécution concurrente de plusieurs transactions peut générer de l'incohérence dans les données. Dans [**DON02**] beaucoup d'exemples d'incohérence sont présentés. Une transaction a quatre propriétés (**A.C.I.D**), tel que A c'est (l'Atomicité ; principe de tout ou rien), C (la Consistance ; transformation d'un état cohérent de la base de données en un autre état cohérent), I (l'Isolation ; chacune des transactions s'exécute isolément des autres), D (la Durabilité ; Les mises à jour d'une transaction seront permanentes une fois qu'elle est validée). Le système de stockage qui exécute les opérations des transactions doit assurer une partie de ces propriétés (A, I et D). Les techniques utilisées pour cela, sont le contrôle de concurrence (l'Isolation et une partie de la Consistance) et le recouvrement (l'Atomicité et la Durabilité et une partie de la Consistance).

Dans ce travail on s'intéresse à la partie recouvrement. Il existe plusieurs techniques pour recouvrer les données d'un serveur après une panne, parmi elles, on a les pages d'ombre, les listes d'intentions et le journal incrémental (journalisation), ces dernières permettent de retrouver un état cohérent de la base de données. Nous allons étendre la SDDS LH* pour

supporter la journalisation. Ainsi on pourra alors l'utiliser comme système de stockage avec recouvrement pour construire un SGBD parallèle.

Plus en détails, il s'agit de proposer une solution pour permettre aux serveurs LH* de journaliser les opérations des transactions. Cette solution permettra de défaire les effets des transactions à annuler durant le déroulement normal du système et de refaire ceux des transactions validées lors des reprises après pannes. A notre connaissance, c'est la première fois qu'une technique de journalisation a été proposée pour la SDDS LH*.

Notre mémoire s'articule en deux parties, et chacune des deux composée de deux chapitres comme suit:

- La première partie : la première partie représente l'état de l'art, elle contient deux chapitres.

Chapitre 1 : dénommé « **Gestion des transactions** » et qui contient les notions générales liées à la gestion des transactions dans les systèmes de stockage classiques.

Chapitre 2 : dénommé « **Les Structures de Données Distribuées et Scalables SDDS** », et aborde une définition sur les **SDDSs** et le principe de leur fonctionnement, on va voir aussi leurs caractéristiques, et aussi leur différents types, on mettra l'accent sur les **SDDSs** de type **LH*** parce que celles-ci sont l'objet de notre travail. Nous verrons aussi leurs architectures et composants (clients, serveurs, coordinateur) avec des exemples explicatifs.

- La deuxième partie : elle représente nos propositions sur l'extension de LH* pour supporter la journalisation. Elle est formée aussi de deux chapitres comme suit.

Chapitre 3 : dénommé « **journalisation des opérations de mises à jour dans LH*** », on va voir dans ce chapitre la présentation des différents composants, et des différents protocoles du verrouillage …etc. Et on verra aussi dans ce chapitre une proposition sur le journal **Undo/Redo** (contient les traces des transactions actives) pour les différentes opérations (ajout, suppression, modification) et deux propositions pour la gestion du journal Redo (contient les traces des transactions validées). La journalisation des éclatements et des fusions de cases est aussi prise en compte par notre proposition ainsi que les méthodes de reprise après une panne appropriées.

Chapitre 4 : dénommé « **Mise en œuvre** », on va voir dans ce chapitre la description d'un prototype qui nous a permis de tester nos propositions concernant la journalisation dans LH*.

Nous terminerons notre mémoire par une conclusion générale qui comportera le bilan général du travail ainsi que les éventuelles perspectives que nous prévoyons pour l'extension du système.

Chapitre

I

Gestion des transactions

I.1.Introduction

La gestion des transactions permet à un système de base de données d'exécuter correctement, Les opérations d'accès aux données appartenant à plusieurs programmes clients, de manière entrelacés (mélangés).

Nous présentons dans ce chapitre les notions de base de la gestion des transactions ainsi que la théorie sous-jacente (la sérialisabilité).

I.2.Notions générales sur les transactions

I. 2. 1. Définition d'une transaction

Une transaction est une séquence d'actions (de requêtes consultatives et/ou modificatives) sur les données de la base de données, lancées par un même utilisateur et considérées par lui comme un tout indivisible. Cette séquence est à effectuer totalement ou pas du tout et prenant la base de données dans un état cohérent, elle doit la rendre dans un nouvel état cohérent.

Les transactions se caractérisent par un seul début (mot clé Start) et un seul point de validation(Commit) si elles n'ont pas fait la validation, alors elles seront annulées immédiatement (Abort). Chaque transaction accède à des données partagées sans interférer avec les autres transactions, et si elle se termine normalement alors ses effets sont permanents, autrement, elle n'aura pas d'effet **[BER87]**.

I.2.2.Les propriétés ACID des transactions

L'acronyme ACID est couramment utilisé pour désigner les propriétés que possède toute transaction : Atomicité Consistance Isolation Durabilité.

Atomicité :

D'abord, une transaction doit être atomique (tout ou rien), signifiant qu'elle s'exécute complètement ou pas du tout. Il ne doit pas y avoir aucune possibilité que seulement une partie d'un programme de transaction soit exécutée.

Le système de **TP** (*Transaction Processing*) garantit l'atomicité par les mécanismes de base de données qui dépistent l'exécution de la transaction. Si le programme de transaction échoue pour quelque raison avant qu'il termine ses opérations, le système de **TP** défera les effets de toutes les mises à jour que le programme de transaction a déjà faites, sauf s'il obtient à la fin et exécute toutes ses mises à jour, le système de **TP** permet aux mises à jour de devenir une partie permanente de la base de données.

Si le système de **TP** échoue, alors en tant qu'élément de ses actions de rétablissement il défait les effets de toutes les mises à jour par toutes les transactions qui s'exécutaient à l'heure de l'échec. Ceci s'assure que la base de données est renvoyée à un état incohérent suivant un échec, réduisant la condition pour l'intervention manuelle pendant la relance.

En employant la propriété d'atomicité, nous pouvons écrire un programme de transaction qui émule une transaction atomique, comme un retrait de compte bancaire, une réservation de vol, ou une vente des parts courantes. Chacune de ces actions d'affaires exige de mettre à jour les données élémentaires multiples. En mettant en application l'action d'affaires par une transaction, nous nous assurons que toutes les mises à jour sont exécutées ou absentes. L'accomplissement réussi d'une transaction s'appelle ***Commit***. L'échec d'une transaction s'appelle ***Abort***.

Consistance :

Cette propriété assure qu'une transaction fait passer la base d'un état consistant. Autrement dit, les transformations apportées par une transaction préservent certains invariants.

Isolation :

Cette propriété assure que les exécutions concurrentes de transactions sont isolées les unes des autres : il n'y a aucune interférence entre ces exécutions. En d'autres termes, cela signifie que les effets d'une transaction ne sont visibles de l'extérieur que si elle peut être totalement exécutée et cela avec succès. Aucune des étapes intermédiaires ne doit donc être visible avant cette validation. Si pour une raison ou une autre, une transaction ne peut s'exécuter dans sa totalité, aucun des

effets des opérations qu'elle a pu réaliser avant cet abandon ne doit être perceptible de l'extérieur.

Durabilité :

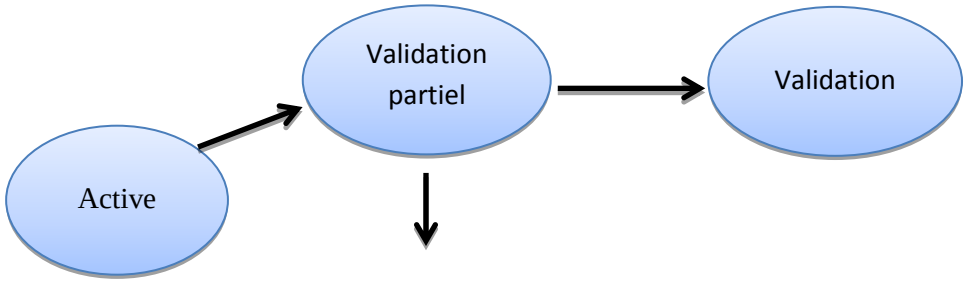
Cette propriété correspond au fait qu'une fois une transaction validée, quoiqu'il puisse arriver, ses effets deviennent permanents et visibles de l'extérieur. Cela doit rester le cas même en cas de défaillance de l'un des composants du système (disque, site, lignes de communication).

Par conséquent, une transaction est un ensemble d'opérations exécutées sur un ou plusieurs serveurs de données qui sont publiées par un programme d'application et sont garanties pour avoir les propriétés **ACID** par le système d'exécution des serveurs impliqués. Le contrat **ACID** entre le programme d'application et les serveurs de données exige au programme de délimiter les frontières de la transaction aussi bien que la fin réussie ou anormale désirée des résultats de la transaction, toutes les deux d'une façon dynamique. Il y a deux manières qu'une transaction peut finir : elle peut commettre, ou elle peut annuler. Si elle commet, tous ses changements à la base de données sont installés, et ils demeureront dans la base de données jusqu'à ce qu'une autre application apporte d'autres modifications. En outre, les changements sembleront à d'autres programmes d'intervenir ensemble. Si la transaction est annulée, aucun de ses changements n'entrera en effectué, et le système de gestion de bases de données va retrouver en reconstituant des valeurs précédentes à toutes les données qui ont été mises à jour par le programme d'application. Une interface de programmation d'un système transactionnel doit par conséquent offrir trois types d'appels **[LEU09]**:

(i) « *début d'une transaction* » : pour spécifier le commencement d'une transaction.

(ii) « *fin d'une transaction* » : pour spécifier la fin réussie d'une transaction.

(iii) « *arrêt d'une transaction* » : pour spécifier la fin non réussie d'une transaction avec la demande d'annuler la transaction.



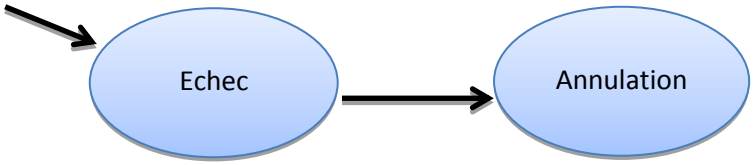


Figure 0-1 : Etats d'une transaction[LEU09]

I.2.3.Types de transactions

Les transactions se divisent en quatre types, transactions centralisées et transactions réparties comme on va le voir.

I.2.3.1.**Transaction centralisée**

Une transaction centralisée veut dire que les données dans un site doivent être toujours stables, c'est-à-dire chaque fois nous aurons cherché d'une donnée quelconque, si dans un instant donné, cette donnée existe dans un site donné alors toujours cette donnée doit exister dans le même site précédent, par contre, la transaction répartie ne sera jamais ainsi.

I.2.3.2.**Transaction répartie**

Une transaction répartie veut dire que les données dans un site ne doivent pas être toujours stables, c'est-à-dire chaque fois nous allons chercher une donnée quelconque, peut-être nous avons vu précédemment cette donnée dans un site, et dans un autre instant nous allons voir la même donnée existée dans un autre site.

I.2.3.3.**Les transactions plates**

Représentent les transactions qui sont effectuées à seulement un niveau d'exécution, c.-à-d., chaque transaction représente seulement un indépendant de bloc de n'importe quelle autre transaction. Elles sont mieux adaptées aux transactions en ligne (On Line Transaction Processing : **OLTP**) qui sont de courte durée, traitent peu de données et permettent l'exécution de plusieurs transactions simultanées.

I.2.3.4.**Les transactions emboîtées (imbriquées)**

Une transaction emboîtée (ou sous transaction) est une transaction exécutée à l'intérieur d'une autre transaction (transaction englobant). Il existe deux types d'imbrication :

- **Imbrication fermée**

Une sous transaction lance ces opérations après la transaction mère et se termine avant elle.

- **Imbrication ouverte**

Une sous transaction peut s'exécuter et valider indépendamment de la transaction comportant.

Une sous transaction peut s'exécuter et valider indépendamment de la transaction comportant.

I.2.4.Schéma général d'un système transactionnel centralisé

Le système transactionnel permet de superviser l'exécution concurrente de tâches clientes (transactions) dans un système informatique, tout en garantissant un comportement correct et cohérent de la base même en présence de pannes **[HID07].**

On peut diviser le système transactionnel en trois modules principaux : le gestionnaire de transactions (**TM** : transaction manager), le contrôleur de concurrence (**CC** : Concurrency Controler) et le gestionnaire de données (**DM** : data manager). Ce dernier peut être divisé en deux parties : gestionnaire de recouvrement et gestionnaire du cache **[BER87].**

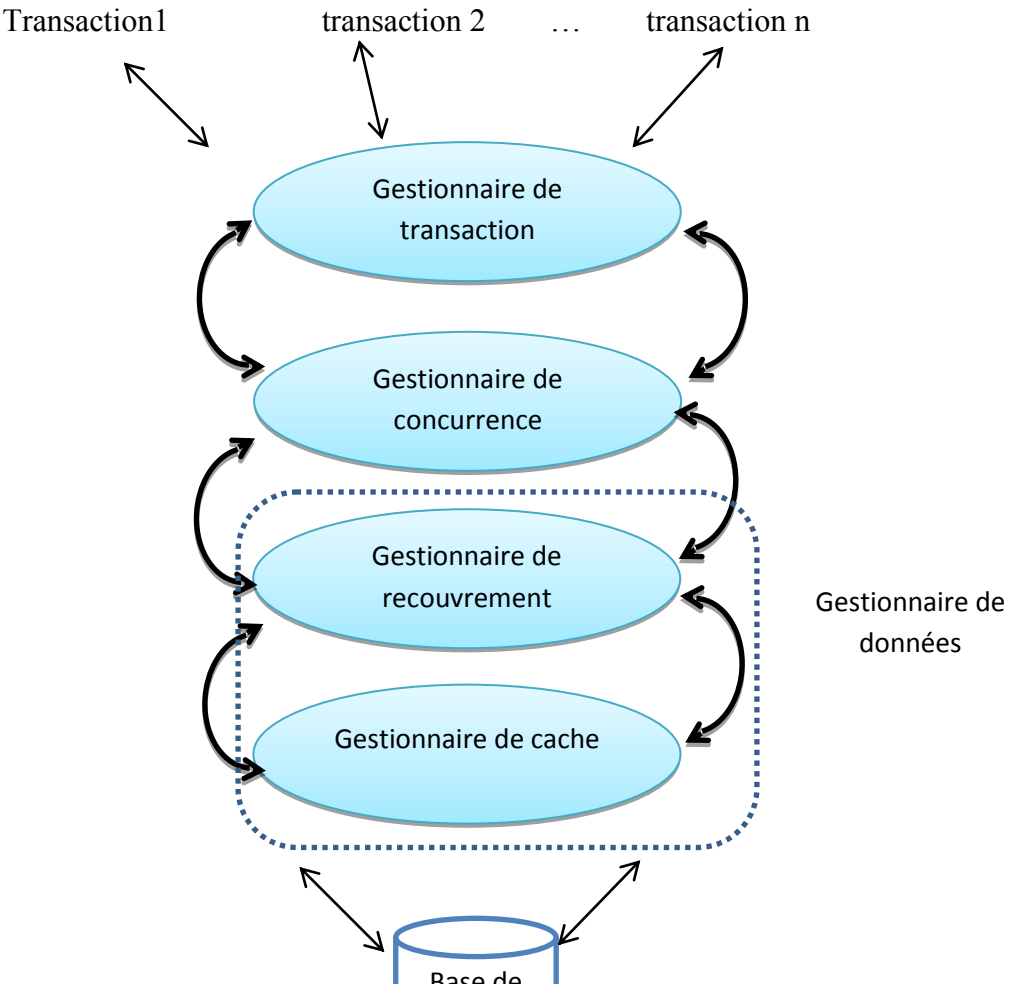


Figure 0-2 : Schéma général d'un système transactionnel centralisé[BER87].

Dans ce schéma, le travail s'effectue selon un modèle de couches i.e. la communication se fait avec le niveau immédiatement supérieur ou inférieur. Voici, de bas en haut dans le schéma, le rôle de chaque module du système transactionnel.

I.2.4.1.**Le gestionnaire de données**

Ce module est lui-même divisé en deux parties **[BER87][HID07]**:

1. **Le gestionnaire du cache**

Responsable des accès au disque dur. Il effectue deux opérations de base :

- **Fetch (x) :** transfert de l'élément de donnée x de la mémoire stable vers la mémoire volatile.
- **Flush (x) :** l'opération inverse de fetch(x). Peut-être enclenchée par requête d'écriture, ou dans le cas de remplissage du cache, afin de libérer de l'espace.

2. **Le gestionnaire de recouvrement**

Ce module assure la propriété d'atomicité des transactions, en éliminant les effets des transactions annulées avant la terminaison de leur exécution (dans le cas d'une panne système, par exemple), et assure leur durabilité, en rendant permanents les modifications des transactions validées.

En utilisant un journal des images avant pour défaire les mises à jour d'une transaction (Undo), et un journal des images après pour refaire les mises à jour d'une transaction (Redo).

Pour atteindre cet objectif, on utilise généralement une structure de journal, et la communication avec le gestionnaire du cache, immédiatement inférieur, se fait à l'aide des deux actions fetch et flush uniquement.

I.2.4.2.**Le contrôleur de concurrence**

Ce module est responsable de la gestion de l'exécution concurrente des transactions i.e. contrôle l'ordre d'exécution des opérations par le gestionnaire de données, et ceci en faisant attendre certaines opérations ou, carrément, en les annulant **[BER87].**

Le but est de produire une exécution respectant la propriété d'isolation **[HID07]**, et ceci grâce à l'implémentation d'algorithmes de gestion de la concurrence dont les plus utilisés sont le verrouillage et l'estampillage **[LOU05]**.

Pour exécuter une opération de base de données, une transaction passe cette opération au **CC**. Après réception de l'opération, ce dernier peut prendre une des trois possibilités:**[BER87]**

1. Exécuter : Il peut passer l'opération au **DM**. Quand le **DM** finit d'exécuter l'opération, il informe le **CC**. D'ailleurs, si l'opération est lire, le **DM** renvoie les valeurs qu'il a lues, que le **CC** transmet par relais de nouveau à la transaction.

2. Rejeter : Il peut refuser de traiter l'opération, dans ce cas il indique à la transaction que son opération a été rejetée. Ceci fait annuler la transaction. L'annulation peut être publiée par la transaction ou par le Gestionnaire des transactions.

3. Retarder : Il peut retarder l'opération en la plaçant dans une file d'attente interne au **CC**. Plus tard, il peut enlever l'opération de la file d'attente pour l'exécuter ou pour la rejeter. Dans le remplacement (tandis que l'opération est retardée), le **CC** est libre pour choisir d'autres opérations.

I.2.4.3.**Le gestionnaire de transactions**

Il reçoit toute les instructions et requêtes de type accès disque (Read, Write), transaction (Start, Commit, Abort) ou base de données (Insert, Delete), les estampille d'un numéro unique, représentant l'identifiant de la transaction **[HID07]** et les envoie à la couche immédiatement inférieure : le contrôleur de concurrence.

Selon le type de l'opération envoyée, une réponse sous forme de résultat (dans le cas d'une lecture) ou d'un acquittement (dans les autres cas) est attendue avant de poursuivre l'exécution. Ce module est aussi responsable de l'affectation des opérations à des sites distants dans le cas d'une base de données distribuée **[BER87]**.

I.2.5.Schéma général d'un système transactionnel réparti

On garde le schéma d'un système transactionnel centralisé au niveau de chaque site, en apportant quelques modifications au gestionnaire de transactions pour prendre en compte les transactions globales (s'exécutant sur plusieurs sites). En effet, le gestionnaire de transactions aura pour rôle, en plus des tâches définies précédemment, de découper la transaction en sous transactions s'exécutant sur les différents sites et de coordonner les actions de la transaction globale**.**

Pour ce faire, le gestionnaire de transactions au niveau du site local est relié aux contrôleurs de concurrence de chaque site du système distribué, pour envoyer des requêtes et recevoir par la suite des résultats ou des acquittements.

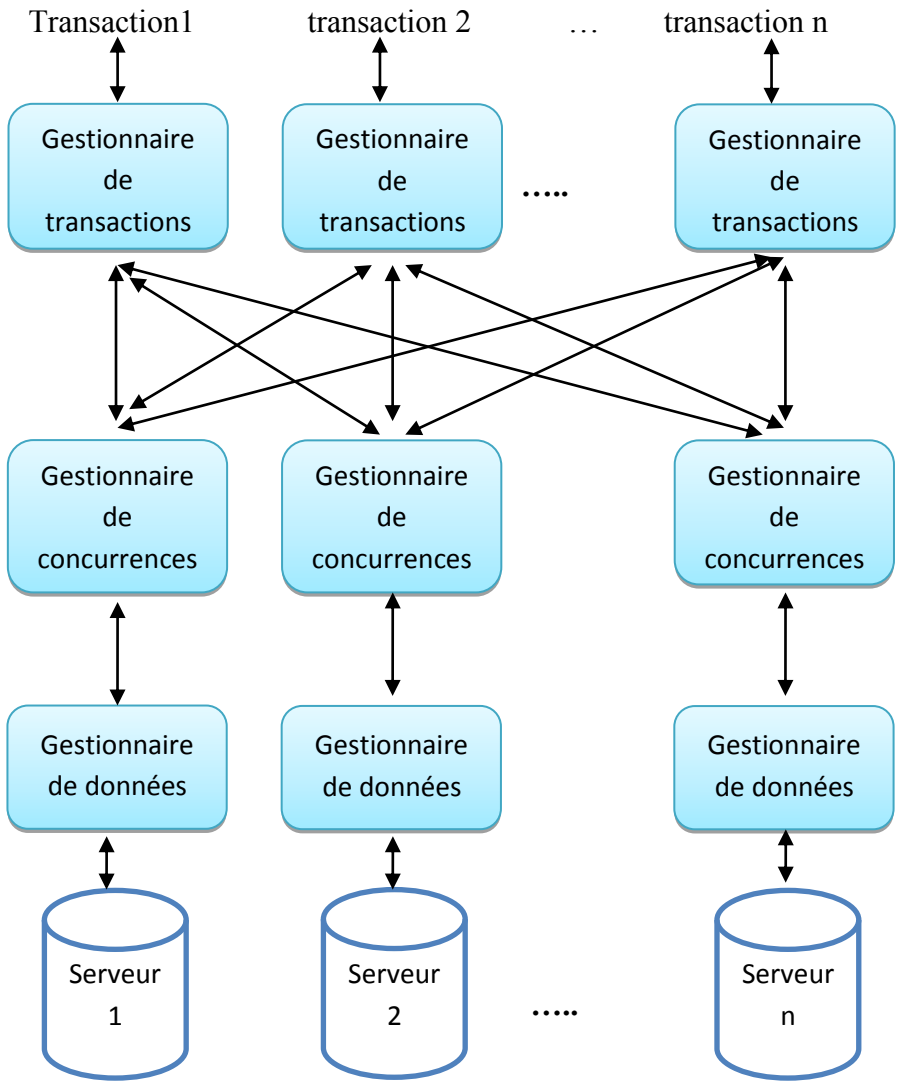


Figure 0-3 : Système transactionnel réparti[BER87].

I.3.Théorie de la sérialisabilité

Pour simplifier la formalisation des opérations d'une transaction, posons alors :

R_i = Read de T_i

W_i= Write de T_i

C_i= Commit de T_i

A_i= Abort de T_i

I.3.1.Ordre partiel

Une Transaction peut être modélisée par un **ordre partiel** ($<_i$) se terminant soit par A_i soit par C_i. Les sommets sont les opérations $\{r, w\}$, les arcs modélisent l'ordre d'exécution.
Exemple : T1 = ($\{R_1(x), W_1(x), W_1(y), C_1\}$, $\{R_1(x) <_1 W_1(x), W_1(x) <_1 C_1, W_1(y) <_1 C_1\}$)

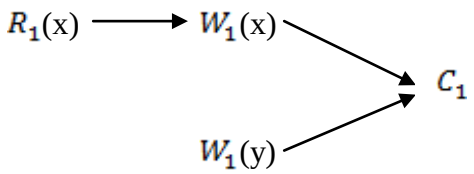


Figure 0-4:Ordre entre opérations

Dans une transaction, deux opérations (p et q) sur la même donnée (x) doivent toujours être en relation : p(x) < q(x) ou bien q(x) < p(x)

I.3.2.Notion d'histoire

I.3.2.1.**Définition**

Une histoire (H) est une structure qui modélise une exécution (pouvant être entrelacée), d'un certain nombre de transactions ordonnées partiellement.

Deux opérations p et q sont dites conflictuelles si elles appartiennent à deux transactions différentes et portent sur la même donnée et l'une au moins est un write c.-à-d. deux opérations conflictuelles ne sont pas commutatives.

Une transaction T_i dans une histoire H est dite :

Validée si ($C_i \in$ H) **Annulée** si ($A_i \in$H) **Active** si ($C_i \notin$H et $A_i \notin$ H)

I.3.2.2.**Histoire complète**

Une histoire complet (H, $<_H$) est formellement défini :

a) H =$U_{i=1..n}\ T_i$

b) $<_H \supseteq U_{i=1..n}\ <_i$

c) pour chaque couple d'opérations conflictuelles p et q, on a soit p $<_H$ q soit q $<_H$ p

I.3.2.3.**Projection validée d'une histoire**

La **projection validée** d'une histoire H (notée C(H)) est une histoire obtenu à partir de H en supprimant toutes les opérations des transactions actives ou annulées, donc C(H) est une histoire complète.

I.3.2.4.**Définition de l'équivalence de deux histoires**

Nous voulons définir l'équivalence de sorte que deux histoires soient équivalentes si elles ont les mêmes effets. Les effets d'une histoire sont les valeurs produites par la sauvegarde des opérations des transactions non annulées.

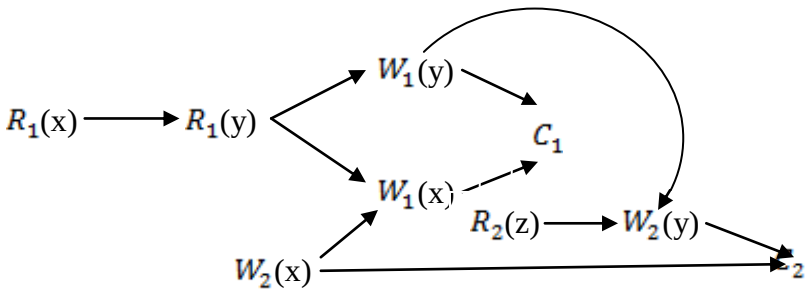
Par conséquent un histoire H est **équivalent** à une histoire H' (H ≡ H') Si et seulement si [**BER87**]:

- H et H' sont formés du même ensemble de transactions et des mêmes opérations.
- L'ordre des opérations conflictuelles des transactions non annulées est le même dans les deux histoires.

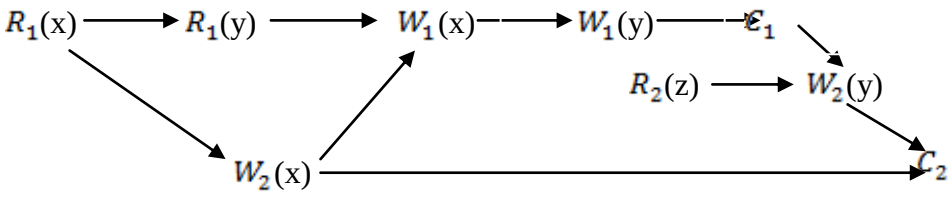
$\forall\ p_i \in T_i$ Et $p_j \in T_j$, deux opérations conflictuelles, et $A_i \notin T_i$ et $A_j \notin T_j$ alors $p_i <_H p_j$ $\Rightarrow p_i <_{H'} p_j$, Autrement dit Les deux exécutions produisent le même effet sur la base de données et sur les transactions.

Exemple :

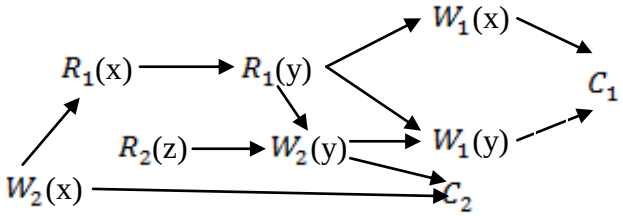
(H_1)



(H_2)



(H_3)



- $H_1 \equiv H_2$

  Car les opérations conflictuelles de H_1 et H_2 sont ordonnées de la même manière

- $H_1 \neq H_3$

  Contre-exemple :

$R_1(x) <_{H1} W_2(x)$ alors que $W_2(x) <_{H3} R_1(x)$

- $H_2 \neq H_3$

  Contre-exemple :

$R_1(x) <_{H2} W_2(x)$ alors que $W_2(x) <_{H3} R_1(x)$.

Alors chaque deux transactions équivalentes lisent les mêmes données alors elles auront sauvé les mêmes données. Pour cela on trouve:

1. si chaque transaction lit chacune de ses données élémentaires de la même chose sauve dans les deux histoires, alors toutes écrire sauvent les mêmes valeurs dans les deux histoires.

2. si pour chaque donnée élémentaire x, la sauvegarde finale de x est la même en les deux histoires, alors la valeur finale de toutes les données élémentaires sera la même dans les deux histoires.

3. si tout écrire sauve les mêmes valeurs dans les deux histoires et laisse la base de données dans le même état final, alors les histoires doivent être équivalentes.

I.3.3.Sérialisabilité

I.3.3.1.**Histoire sérielle**

Une histoire complète H est dite **sérielle** si, pour chaque deux transactions T_i et T_j qui apparaissent dans H, toutes les opérations de T_i apparaissent avant de T_j, ou vice versa. Ainsi, une histoire séquentielle représente une exécution dans laquelle il y a non interfoliage des exécutions de différentes transactions. Chaque transaction exécute du commencement jusqu'à la fin avant que le prochain puisse commencer.

Remarque :

Une histoire sérielle est forcément complète **[BER87]**.

I.3.3.2.**Histoire sérialisable**

Lorsque deux ou plusieurs transactions s'exécutent en concurrence, leurs opérations peuvent être entrelacées (une opération de la transaction T_i se trouve entre deux opérations de la transaction T_i). Cet entrelacement peut amener des incohérences à la base de données même si les transactions en elles-mêmes sont cohérentes.

Deux opérations sont dites conflictuelles si elles s'exécutent sur le même élément de données, et au moins une des deux est une écriture **[BER87].** Les opérations conflictuelles ne sont pas commutables entre elles. Si p et q sont deux opérations conflictuelles alors l'exécution de p suivi par q ne donne pas le même résultat que l'exécution de q suivi par p.

L'exécution entrelacée d'un ensemble de transactions ne donne pas forcément un résultat correct, même si ces transactions sont correctes.

Considérons, par exemple, une gestion simple d'un compte bancaire. Dans cet exemple, deux transactions T_1 et T_2 ont pour but d'ajouter 100 et 200 DA, respectivement, à un compte bancaire x. Ces transactions peuvent être schématisées ainsi :

T_1 : Start, R_1 (x, t), W_1 (x, t+100), C_1. T_1 lit la somme contenue dans le compte et y ajoute 100 DA.

T_2 : Start, R_2 (x, a), W_2 (x, a+200), C_2. T_2 lit la somme contenue dans le compte et y ajoute 200 DA.

Supposons que le contrôleur de concurrence choisisse pour ordre d'exécution: $R_1(x)$, $R_2(x)$, W_2 (x), $C_2, W_1(x), C_1$. Il est clair que cette exécution fait perdre la somme d'argent déposée par la deuxième transaction, car la dernière opération exécutée, $W_1(x)$, n'a pas pris en compte cette modification de T_2, apportée entre la lecture et l'écriture de la somme par T_1.

Définition :

Une histoire H est « **sérialisable (SR)** » si sa projection validée C(H) est une histoire équivalente à une histoire sérielle formée par les mêmes transactions.

Exemples:

T_1 = $R_1(x)$; $W_1(x)$; $R_1(y)$; $W_1(y)$;C_1 T_2 = $R_2(x)$; $R_2(y)$; C_2

H_1 = ($R_1(x)$, $W_1(x)$, $R_2(x)$, $R_2(y)$, $R_1(y)$, $W_1(y)$, C_1 , C_2) non sérialisable car il n'est équivalent ni à T_1 ; T_2 ni à T_2 ; T_1

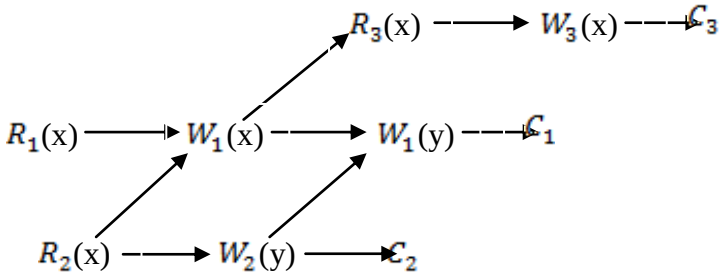
H2 = ($R_1(x)$, $W_1(x)$, $R_2(x)$, $R_1(y)$, $W_1(y)$, $R_2(y)$, C_1 , C_2) sérialisable car il est équivalent à T_1 ; T_2

I.3.3.3.**Graphe de sérialisation**

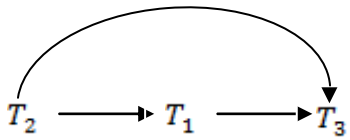
Le graphe de sérialisation d'une histoire SG(H) est un graphe orienté sans circuit (GOSC) où les nœuds sont les transactions validées de H et les arcs de type $T_i \rightarrow T_j$ indiquent l'existence d'une opération de T_i qui précède et est en conflit avec une opération de T_j dans l'histoire H.

Exemple :

H



Donc cette histoire implique le graphe de sérialisation suivant :



$T_2 \longrightarrow T_1$: Car $W_2(y) < W_1(y)$ ou $R_2(x) < W_1(x)$.

$T_1 \longrightarrow T_3$: Car $W_1(x) < R_3(x)$ ou $W_1(x) < W_3(x)$ ou $R_1(x) < W_3(x)$.

$T_2 \longrightarrow T_3$: Car $R_2(x) < W_3(x)$.

Remarque : si à la place de $W_3(x)$ on avait $W_3(z)$ par exemple, il n'y aurait pas eu d'arc entre T_2 et T_3.

I.3.3.4.**Théorème d'équivalence entre le SG sans circuit et la SR d'une histoire H:**

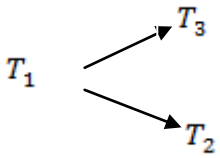
Une histoire H est sérialisable si et seulement si son SG(H) est sans circuit.

Donc par conséquent si une histoire complète H possède un graphe de sérialisation SG(H) sans circuit, alors H est équivalent à toute histoire sérielle obtenue par un tri topologique de SG (H).

$R_1(x), W_1(x), R_2(x), R_1(y), W_1(y), R_2(y), C_1, C_2$

Exemple :

H = $(W_1(x) ; W_1(y) ; C_1 ; R_2(x) ; R_3(y) ; W_2(x) ; C_2; W_3(y) ; C_3)$

SG(H) = 

Il existe alors deux tri topologiques : $T_1 ; T_2; T_3$ ET $T_1 ; T_3; T_2$ et donc H est équivalent à une exécution en série T1;T2;T3 ou T1;T3;T2 (les deux donneront le même résultat).

Démonstration :

Pour démontrer l'équivalence **SG(H) sans circuit <=> H Sérialisable,** il faudra démontrer l'implication dans les deux sens.

1. **SG(H) sans circuit => H Sérialisable ?**

SG(H) est sans circuit => on peut générer un tri topologique de ses sommets.
Soit $(T_1, T_2,... T_m)$ un tel tri. L'histoire sérielle H_S,= $(T_1, T_2,... T_m)$ est forcément équivalente à C(H) (projection Validée de H) et donc que H est SR, car :

1- H_S et C(H) sont formés des mêmes transactions (T_1, T_2,... T_m) par construction.

2- Chaque couple d'opérations conflictuelles p_i et q_j sont ordonnées de la même manière dans les deux histoires.

Si $p_i <_{C(H)} q_j$ alors il existe un arc $T_i \longrightarrow T_j$ dans SG(H) donc T_i apparait avant T_j dans tout tri topologique, donc dans H_S aussi et donc toutes les opérations de T_i précèdent celles de T_j, en particulier $p_i <_{Hs} q_j$.

2. H Sérialisable => SG(H) sans circuit ?

H sérialisable => C(H) équivalent à une certaine histoire sérielle H_S (par définition de **SR).**

De plus s'il existe un arc $T_i \rightarrow T_j$ dans SG(H) cela veut dire qu'il existe un couple d'opérations conflictuelles $p_i$et q_j tel que $p_i <_{C(H)} q_j$ et donc $p_i <_{Hs} q_j$ (car H_S est équivalente à C(H)) donc T_i apparaît avant T_j dans H_S .

Donc s'il existait un circuit dans SG(H) tel que $T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_1$ on aurait eu une contradiction dans l'ordre d'apparition des transactions de H_S (T_1 avant T_2 avant ... T_1) donc il ne peut pas exister de circuits dans SG(H).

I.4.Recouvrabilité

Quand une transaction est annulée, ses écritures dans la base de données seront défaites. Mais c'est possible qu'il existe des transactions validées qui aient déjà lues ces données avant qu'elles soient défaites. Donc ces transactions ont lu des données erronées.

Exemple: H = W_1(x), R_2(x), W_1(y), C_2 , A_1

A ce niveau les mises à jour de T_1 seront défaites (W_1(x)) et la valeur lue par T_2 (R_2(x)) n'est plus valide et comme T_2 a déjà validé, on ne peut plus l'annuler.

La solution consiste à retarder C_2 jusqu'à la terminaison de T_1.

Formellement, une transaction T_i lit une donnée x à partir de T_j dans H si:

- W_j(x) < R_i(x)
- A_j ne précède pas R_i(x) dans H
- s'il existe un W_k(x) tel que W_j (x) < W_k(x) < R_i(x), alors A_k< R_i(x).

I.4.1.Histoire recouvrable

Une histoire est dite « **Recouvrable (RC)** » si à chaque fois que $T_i$lit une donnée à partir de T_j (i ≠ j) on a $C_j <_H C_i$. Autrement dit : H est recouvrable si chaque transaction ne peut valider qu'après la validation des transactions à partir desquelles elle a lue des données.
Si T_j est annulée il en sera de même pour T_i (**annulations en cascade**).

I.4.2.Annulations en cascade CA (Cascading Aborts)

Pour éviter les annulations en cascade (qui dégradent les performances), on n'autorise pas la lecture de données écrites par des transactions encore actives.
- A chaque fois qu'une transaction T_i lit une donnée x à partir de T_j (i ≠ j) dans H, on doit avoir $C_j < R_i$ (x), « C'est-à-dire retarder les lectures $R_i(x)$ »
- On dit alors que H **évite les annulations en cascade**

Exemple:
H = W_1(x), R_2(x), W_1(y), A_1 , A_2

Dans cet exemple l'annulation de la transaction T_1 conduit à l'annulation de la transaction T_2 , car cette dernière a utilisé une donnée (x) n'est pas valable.

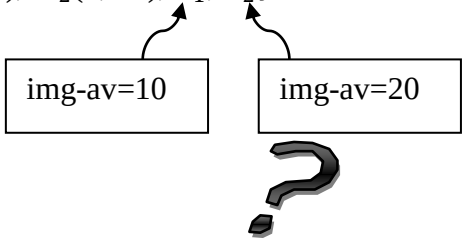
I.4.3. Histoire strict (ST)

Pour faciliter l'implémentation de la procédure d'annulation (à l'aide des images avant), on impose une contrainte supplémentaire stipulant qu'une transaction ne peut lire ou écrire sur une donnée déjà écrite par une transaction active. On dit alors que H **strict** à chaque fois qu'on a dans une histoire W_j(x) < op_i (x) (avec op_i = R_i ou W_i et i ≠ j) on doit aussi avoir : A_j < op_i (x) ou bien C_j < op_i (x).

Exemple:

(X=10 au départ)

H = {W_1(x, 20), W_2(x, 30), A_1, A_2}



On aura X=20 au lieu de 10 alors il faut utiliser la **ST**

I.4.4.Classification entre (RC ACA ST)

Il a été démontré dans [**BER87**] que toutes les histoires ST sont aussi ACA et toutes les histoires ACA sont aussi RC. Les exécutions correctes sont celles qui sont RC et SR en même temps. Il existe des histoires SR qui ne sont pas RC.

Les propriétés ACA et ST ont un intérêt d'ordre pratique seulement car elles permettent d'obtenir des exécutions performantes.

ST ⊂ACA⊂ RC

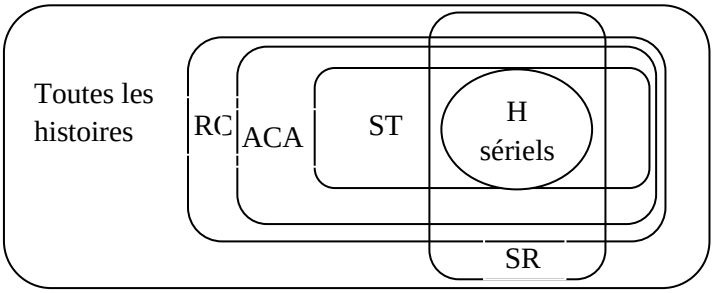


Figure 0-5 : relations entre les propriétés des histoires[LAR87]

I.5.Méthodes de contrôle de concurrence :

I.5.1.Contrôle de concurrence par graphe de sérialisation

Dans cette méthode, on construit le graphe de sérialisation au fur et à mesure de l'exécution des transactions et à chaque fois on vérifie s'il y a un cycle dans le graphe. Dans le cas où il y aurait un cycle alors la transaction est annulée **[BER82]**. L'un des problèmes de cette méthode est la gestion coûteuse du graphe **[AMR04]**.

I.5.2.Contrôle de concurrence par certification

Cette méthode est une méthode optimiste, c'est-à-dire, elle laisse les transactions s'exécuter sans aucun contrôle et à la fin, vérifie la cohérence de la base de données. Si la cohérence de la base est respectée alors la transaction est validée, sinon la transaction est annulée **[BER09][KOR89]**. Cette méthode se déroule en trois phases :

1- L'exécution de la transaction: changement des données au niveau de la transaction (locale).
2- Le test de cohérence de l'exécution.
3- S'il n'y a pas d'incohérence alors rapporter les modifications dans la base, sinon la transaction est annulé.

Cette méthode donne de bons résultats dans le cas où la majorité des transactions ne font que des lectures (pas de conflit) **[AMR04]**.

I.5.3.Contrôle de concurrence par estampillage

Afin d'assurer la sérialisabilité, on utilise l'ordre d'entrée des transactions dans le système pour réordonner les opérations au niveau du contrôleur de concurrence **[HID07].** Pour ce faire, on associe à chaque transaction Ti un numéro unique TS(Ti), appelé estampille, qui déterminera par la suite l'ordre d'exécution des opérations conflictuelles dans un contexte d'exécution concurrente.

L'estampille vérifie la propriété fondamentale suivante : si une transaction Tj arrive dans le système après une transaction Ti alors TS(Ti) < TS(Tj) et inversement. Cette relation d'ordre nous permettra par la suite d'ordonner les opérations conflictuelles, et d'assurer l'exécution des opérations conflictuelles de Ti avant les opérations conflictuelles de Tj, si son entrée dans le système précède celle de Tj.

De plus, deux autres estampilles sont associées à chaque élément de données x, comme suit :

- RE (x): valeur de l'estampille de la plus « jeune » transaction (le MAX de TS(Ti)) ayant consulté x.

- WE(x): valeur de l'estampille de la plus jeune transaction (le MAX de TS(Ti)) ayant modifié x.

Ces deux informations nous permettent d'effectuer un ordonnancement chronologique de l'exécution des transactions par l'algorithme suivant **:**

- Quand une opération read(x) portant l'estampille i arrive au niveau du contrôleur de concurrence, l'opération sera rejetée (et la transaction annulée) si i est inférieur à WE(x),

sinon elle sera exécutée et RE(x) mise à jour à la valeur maximale entre i et RE(x) (valeur actuelle).

• Quand une opération write(x) portant l'estampille i arrive au niveau du contrôleur de concurrence, l'opération sera rejetée (et la transaction annulée) si i est inférieur à RE(x), sinon l'opération sera ignorée si i< WE(x) sinon elle sera exécutée et WE(x) mise à jour à la valeur maximale entre i et WE(x) (valeur actuelle).

• Dans les deux cas, une transaction annulée est relancée avec une nouvelle estampille (plus grande) pour éviter de choisir la même transaction victime indéfiniment.

Cette technique est simple et évite le problème d'inter blocage caractéristique des méthodes de verrouillage. Les transactions n'attendent pas, soit elles exécutent leurs opérations, soit elles sont annulées et relancées. Le taux relativement élevé des annulations constitue son principal défaut **[HID07].**

I.5.4.Contrôle de concurrence par verrouillage

L'idée est d'associer à chaque donnée manipulée un verrou permettant à une transaction d'en réserver l'accès. Cette méthode tente de conserver la propriété d'isolation dans un contexte concurrent **[HID07]**.

On distingue généralement deux types de verrous. Le premier, partagé, en lecture, permet à une transaction de lire une donnée mais pas de la modifier. Le second, exclusif, en écriture, donne le droit de lecture et d'écriture sur la donnée auquel il est appliqué **[AMR04].**

Ainsi, une transaction voulant lire une donnée, est obligée d'obtenir, au préalable, un verrou en lecture. Si cette donnée est verrouillée de manière exclusive (écriture) par une autre transaction, elle doit attendre sa libération. Plusieurs transactions en concurrence peuvent obtenir un verrou partagé en lecture sur une même donnée.

Si une transaction souhaite modifier une donnée, elle doit d'abord obtenir un verrou exclusif en écriture. Cette donnée ne doit être verrouillée par aucune transaction, si c'est le cas, la transaction est bloquée en attente de libération. Ainsi, à tout moment, il ne peut y avoir, qu'au plus, une transaction avec un verrou d'écriture sur une même donnée **[HID07]**.

On dit que l'opération d'écriture sur un élément de données, est en conflit avec les opérations de lectures et d'écritures des autres transactions, sur ce même élément.

Le verrouillage des données ne suffit pas pour produire une exécution sérialisable correcte **[AMR04]**. Le protocole de verrouillage à deux phases (2PL) a pour but d'assurer cette sérialisabilité en utilisant les verrouillages pour régler les problèmes liés à la concurrence.

Le verrouillage à deux phases [AMR09]

Le 2PL est le protocole le plus répandu pour assurer des exécutions concurrentes correctes.

Principe

En plus des opérations de lectures et d'écritures, une transaction doit acquérir des verrous et en libérer.

Règles d'utilisation

- Une transaction ne peut lire ou écrire un élément de données sans l'avoir préalablement verrouillé.

- Si une transaction verrouille un élément de données, elle doit le libérer plus tard. Le protocole 2PL utilise des verrous en lecture notés RL (readlock) et des verrous en écriture WL (writelock).

- **RLi(x) :** la transaction Ti a posé un verrou en lecture sur x.

- **WLi(x)** : Ti a posé un verrou en écriture sur x.

- **RUi(x) :** Ti a libéré (relâché) le verrou en lecture posé sur x.

- **WUi(x)** : Ti a libéré le verrou en écriture sur x.

Définition : Deux verrous PLi(x) et QLj(y) sont en conflit si et seulement si : x et y sont un seul et même élément de données, Ti et Tj sont différentes, et au moins un des verrous est en écriture.

Algorithme 2PL

- Le **CC** reçoit une opération Pi(x) (où P=R ou W) et consulte les verrous QLj(x) déjà posés sur x s'ils existent.

- Si PLi(x) est en conflit avec QLj(x), Pi(x) est retardée et la transaction Ti mise en attente.
- Sinon, l'opération Pi(x) est exécutée et le verrou PLi(x) posé sur x.
- Dès que la transaction Ti relâche le verrou, elle ne peut plus en obtenir d'autres.

Ainsi, une transaction comporte deux phases : une phase pour demander les verrous et une phase pour les libérer.

Remarque : Toute exécution obtenue par verrouillage à deux phases est sérialisable.

L'interblocage

C'est l'inconvénient majeur de ce protocole. Le problème arrive lorsque des transactions concurrentes se bloquent mutuellement dans l'attente du déverrouillage de données. Prenant un exemple simple :

Soit T1 et T2 deux transactions : T1= R1(x), W1(y), C1. T2= W2(y), W2(x), C2.

Soit l'histoire H = R1(x), W2(y) … ***interblocage***.

Quand la transaction T1 exécute R1(x) elle obtient un verrou partagé sur x, ensuite la transaction T2 exécute W2 (y) et obtient un verrou exclusif sur y. Quand T1 essaye d'exécuterW1(y) elle est mise en attente, car elle ne peut pas obtenir un verrou exclusif sur y (T1 attendT2) et quand T2 essaye d'exécuter W2 (x) elle est aussi mise en attente car elle ne peut pas obtenir un verrou exclusif sur x (T2 attend T1). A ce niveau, il y a inter blocage et aucune des deux transactions ne peut continuer son exécution **[HID07]**.

Le contrôleur de concurrence, pour traiter ce problème, peut guérir, i.e. utiliser des méthodes de résolution une fois l'inter blocage arrivé, ou bien prévenir, i.e. prendre des précautions à l'avance.

Dans la première solution, un graphe d'attente est maintenu à jour et analysé périodiquement afin de détecter des circuits, cas auquel on remédie par annulation d'une transaction dite victime. Dans ce graphe les sommets représentent les transactions en attente et Ti-> Tj veut dire que Ti attend un verrou détenu par Tj.

Dans la deuxième méthode, un protocole d'estampillage de transactions afin de déterminer un ordre chronologique est utilisé, et ceci selon les deux stratégies Wait-Die et Wound-Die**[TAN94]**.

D'un point de vue performance, les deux méthodes sont similaires. Dans un système réparti, les méthodes Wound-Wait et Wait-Die, s'avèrent moins complexes, et par conséquent plus répandues.

Dans Wait-Die si une opération Pi(x) (P= R ou W) arrive et que x est déjà verrouillée en mode conflictuel par Tj :

- si TS(Ti) < TS(Tj) alors Ti attend la libération du verrou Sinon annuler Ti.

Dans Wound-Wait si une opération Pi(x) (P= R ou W) arrive et que x est déjà verrouillée en mode conflictuel par Tj :

- si TS(Ti) < TS(Tj) alors annuler Tj Sinon Ti attend la libération du verrou.

Dans ces deux méthodes quand une transaction est annulée, elle sera relancée avec la même estampille

Variantes du 2PL

a. Version 2PL conservatrice : chaque transaction est obligée de déclarer ses verrous avant de démarrer l'exécution. Elle évite l'inter blocage, car on ne laisse une transaction s'exécuter qu'après avoir obtenu tous ses verrous **[BER87].**

b. Version 2PL stricte : permet à une transaction de garder ses verrous durant toute son exécution. La libération se fait qu'après validation de cette transaction. Ce protocole permet une exécution stricte : sérialisable et recouvrable **[BER87]**.

I.5.5.Contrôle de concurrence dans un environnement distribué

Une transaction distribuée T a un site origine (site coordinateur), où son exécution a démarré. T envoie ses opérations au gestionnaire de transactions local (voir Figure 0-3 : Système transactionnel réparti), qui se charge de les adresser aux sites appropriés (site participant). La requête, envoyée à un site distant, est prise en charge comme une requête locale à ce dernier, et les résultats sont renvoyés au site d'origine.

La différence entre un système transactionnel centralisé et un autre distribué est flagrante à l'étape de validation de la transaction. La question posée à cette étape est : quel site doit valider cette transaction ?

La validation n'est pas le seul problème posé par la distribution des opérations. En effet, une des caractéristiques des systèmes distribués, qui, en général, est synonyme de fiabilité, est le fait que la plupart des pannes sont partielles. Cette propriété complique l'élaboration d'une solution en ajoutant des exigences de reprise après panne ; pannes qui sont multiples et touchent plusieurs aspects du système réparti, comme décrit par la suite. Les protocoles de validation atomique apportent une solution aux problèmes posés, en tenant compte, selon les versions, de plusieurs configurations possibles du système réparti **[BER87]**.

I.6.Protocole de validation à deux phases

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 à deux phases est généralement utilisé. Ce protocole a été proposé dans un contexte de système réparti mais est aussi utile dans un contexte centralisé multi-processus. La validation à deux phases 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 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 l'annulation atomique, selon que la première étape s'est bien ou mal passée.

I.6.1.Le principe de fonctionnement

Le protocole de validation à 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 base de données 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 base de données distribuée.

Le contrôle du système est centralisé sous la direction d'un processus appelé coordinateur (gestionnaire des transactions du site origine de la transaction), les autres étant des participants (sites où la transaction s'est exécutée).

Lors de la 1ere étape, le coordinateur demande aux autres sites s'ils sont prêts à valider leurs mises à jour par l'intermédiaire du message **PREPARER (PREPAREZ-VOUS)**. Si

tous les participants répondent positivement (prêt), le message **VALIDER** est diffusé : tous les participants effectuent leur validation sur ordre du coordinateur. Si un participant n'est pas prêt et répond négativement ou ne répond pas (timeout), le coordinateur demande à tous les participants de défaire la transaction (**ANNULER**).

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 de continuer la validation éventuelle. Après l'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.

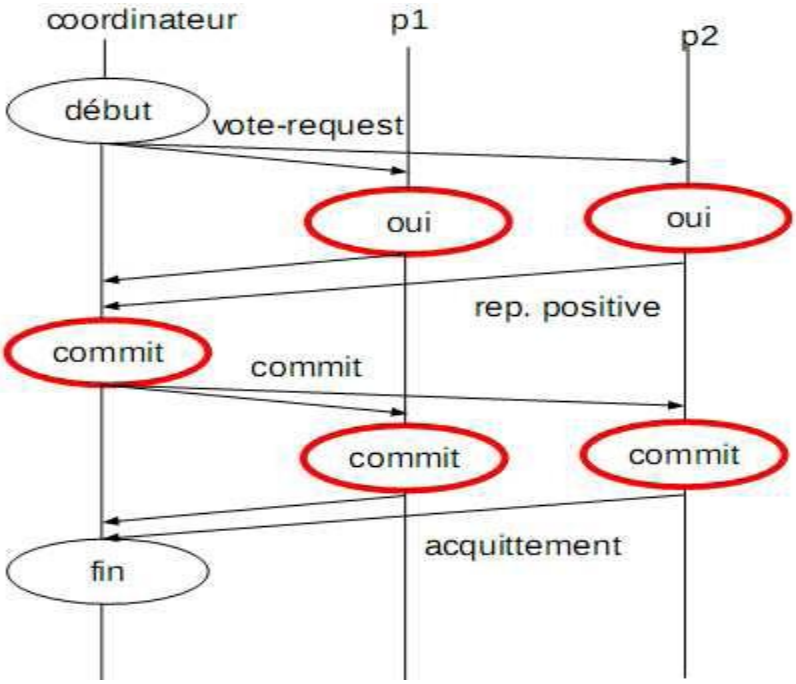


Figure 0-6 : validation à deux phases[LOU05].

I.6.2.Journalisation des transactions distribuées

Durant l'application du protocole de validation à deux phases, les divers états atteints par le coordinateur et les participants sont journalisés de sorte que, en cas de panne, la procédure de reprise puisse reprendre les phases nécessaires du protocole.

Pour cela, chaque site maintient un journal DTLog (Distributed Transaction Log) dans lequel les coordinateurs et les participants dans ce site peuvent stocker les informations nécessaires sur les transactions distribuées.

Le DTLog qui doit être sauvegardé en mémoire secondaire est géré de la manière suivante **[BER87]** :

- Quand le coordinateur envoie le message ″ préparez-vous ″ aux participants, il insert l'enregistrement ″ début 2PC ″ dans son DTLog avant ou après l'envoie du message ″ préparez-vous ″. Cet enregistrement contient les identifiants des participants.
- Si un participant se déclare ″ prêt à la validation ″, il insert l'enregistrement ″ prêt ″ dans son DTLog avant d'envoyer le message ″ prêt ″ au coordinateur.

Par contre, si le participant n'est pas prêt pour la validation, il insert l'enregistrement ″ non prêt ″ dans son DTLog avant ou après l'envoie du message ″ non prêt.″

- Avant que le coordinateur n'envoie le message ″ valider ″ aux participants, il insert l'enregistrement ″ valider ″ dans son DTLog.
- Quand le coordinateur déclare d'annuler la transaction, il insert l'enregistrement ″ annuler ″ dans son DTLog avant ou après l'envoie du message ″ annuler ″ aux participants.

Après que le participant reçoit le message ″ annuler ″ ou ″ valider ″ du coordinateur, il insert dans son DTLog.

Notons que c'est l 'insertion de l'enregistrement ″ valider ″ ou ″ annuler ″ dans le DTLog qui fait que la transaction est validée ou annulée.

I.7.Conclusion

Ce chapitre a permis d'aborder les notions générales entourant le concept de transactions, et citer les propriétés qu'elles doivent respecter afin d'assurer une cohérence de la base, et ceci à tout moment.

Plus en détails on a présenté le cadre formel (la théorie de la sérialisabilité) qui permet d'étudier de manière rigoureuse les propriétés des exécutions de transactions (Histoires). Quelques protocoles de contrôle de concurrence parmi les plus utilisées et de validation atomique ont aussi été décrits.

Chapitre II

Les Structures de Données Distribuées et Scalables (SDDS)

II.1.Introduction

Avec le développement des réseaux et l'apparition de nouvelle architecture telle que multi ordinateur, cluster ou de réseau de station de travail, de nouvelles techniques de stockage et d'accès aux données plus adaptées à ce type d'environnement sont apparues.

Les SDDS (Scalable distributed data structures) sont un exemple de telles techniques.

Ce chapitre est consacré à la présentation générale de cette nouvelle classe de structures de données répartie et détaille un peu plus l'une de ces méthodes, LH*(Distributed Lenear Hashing).

II.2.Définition

Les SDDS constituent une nouvelle classe d'organisation de données; elles sont définies spécifiquement pour l'architecture multi-ordinateur. Elles sont conçues pour surmonter les insuffisances des structures de données distribuées classiques, notamment en ce qui concerne leur approche de point d'accès unique ou de schéma de fragmentation statique. Les SDDS utilisent plutôt une approche de fragmentation dynamique. Le premier schéma appelé LH* (Scalable Distributed LH), proposé par [LIT93], était une adaptation du hachage linéaire aux multi-ordinateurs [LIT96a]. Parce que le temps d'accès à la mémoire vive d'un site distant est plus court que celui au disque du même site, les données d'un fichier SDDS résident en la première (mémoire vive distribuée).

Une caractéristique des SDDSs par rapport aux traditionnelles méthodes de fragmentation est que le nombre de serveurs est dynamique. C'est à dire, qu'au départ, il n'y a qu'un seul serveur SDDS associé au fichier. Quand celui-ci grandit (à cause des insertions envoyées par des clients SDDS), le nombre de serveurs augmente afin de maintenir de bonnes performances d'accès quelle que soit la taille du fichier. Ce changement du nombre de serveurs implique que le schéma de fragmentation change aussi mais la réorganisation n'est que partielle et à faible cout car elle ne concerne qu'une seule case du fichier **[HID07].**

Quand le serveur initial déborde, suite à des insertions, une partie des enregistrements (en général la moitié) est transférée vers un nouveau serveur alloué au fichier. Ce processus s'appelle éclatement (split). Il se répète autant de fois que nécessaire pour les différents serveurs du fichier **[LIT93][LIT94]**. Un fichier peut ainsi s'étendre sur un nombre quelconque de serveurs. Ceci rend sa capacité de stockage potentiellement illimitée et satisfait le principe de la scalabilité. Ce dernier est défini comme la capacité d'une application à maintenir le même niveau de performance face à tout accroissement de charge. La technique utilisée consiste à ajouter des serveurs afin de s'adapter à l'évolution des besoins voir (Figure II-1 : Architecture d'un fichier SDDS.).

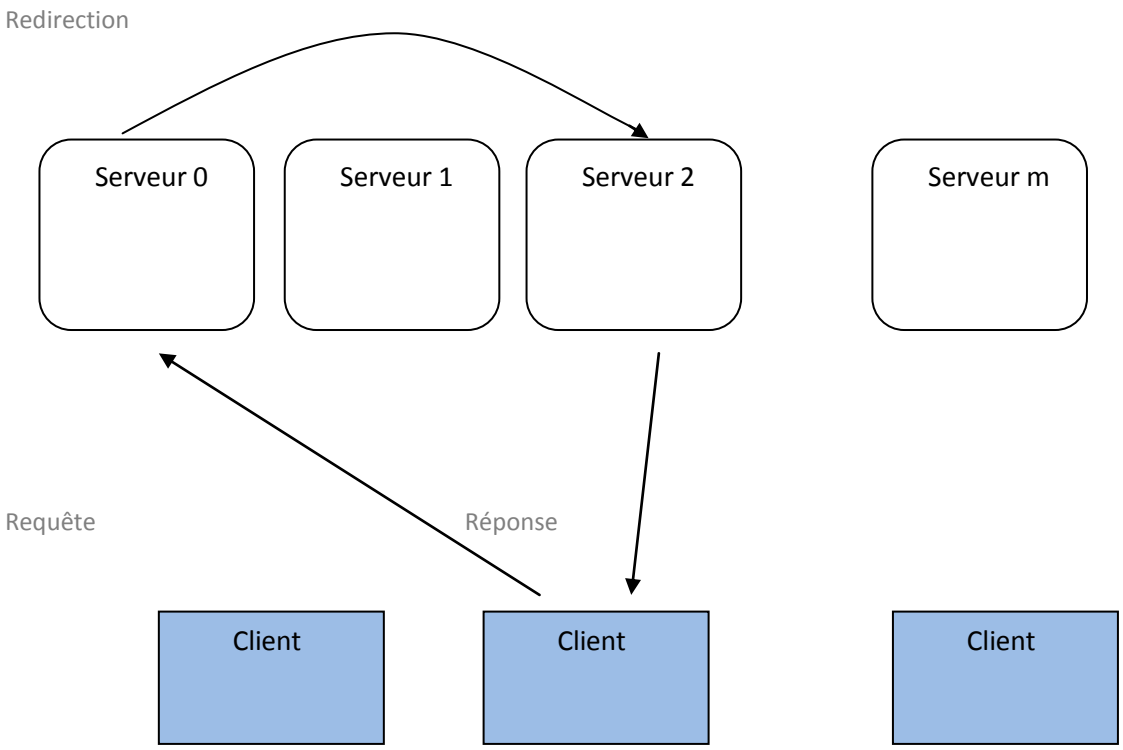


Figure 0-1 : *Architecture d'un fichier SDDS[HID07].*

II.3.Principe de fonctionnement des SDDSs

Un fichier SDDS réside initialement sur un seul site de stockage appelé serveur; il est ensuite étendu dynamiquement à n'importe quel nombre de sites du multi-ordinateur.

Un serveur contient une case pour le stockage des enregistrements du fichier. Quand le nombre d'enregistrements dépasse la capacité de la case, on dit qu'elle a débordée. Ces enregistrements sont répartis sur les serveurs, suivant la valeur de leurs clés.

Les clients utilisent un algorithme (fonction d'accès) pour retrouver l'adresse du serveur contenant l'enregistrement de clé C. Cette fonction est appelée Image du client.

Pour réaliser un degré élevé de scalabilité, une SDDS n'utilise pas un répertoire central d'accès puisque son existence peut créer un goulot d'étranglement et dégrader les performances d'accès du système. Par conséquent, chaque client a sa propre image de la structure du fichier. Elle lui permet de calculer l'adresse du serveur où est supposé se trouver un enregistrement de la clé C (Figure II-2 : Accès à un fichier SDDS-(état A)). Le chargement du nombre de serveurs d'un fichier (à cause des éclatements et fusions) n'est pas modifié aux clients de manière synchrone, de ce fait l'image de client. Cette image s'améliore à chaque fois qu'un client fait une opération (Figure II-2-(état B)).

Par exemple, si un client envoie une requête à un serveur qui ne contient pas les données demandées, le serveur lui renvoie alors un message correctif appelé message d'ajustement de l'image du client (IAM : Image Ajustement Message). Ce message IAM permet au client d'ajuster son image pour ne plus refaire la même erreur. Parallèlement à la mise a jour de l'image du client, le serveur redirige la requête vers un autre serveur. Ce dernier l'exécute et s'il n'y a pas d'erreur d'adressage il renvoie la réponse au client.

En résumé, les principales exigences d'un fichier SDDS sont les suivantes :

- Un fichier SDDS n'a pas de répertoire central d'accès contenant l'image correcte du fichier. Cela permet d'éviter les goulots d'étranglement d'accès.
- Chaque client accède au fichier à travers sa propre image approximative de la répartition des données. Cette image est mise à jour de manière asynchrone par les serveurs. Une mise à jour intervient uniquement lorsque le client commet une erreur d'adressage.

- Les serveurs sont responsables du traitement des requêtes et de leur redirection *(forward)* en cas d'erreur d'adressage. Ils sont aussi responsables de la mise à jour des images des clients.

Finalement, Une SDDS bien conçue doit produire peu d'erreurs d'adressage, peu de redirections et doit permettre en général, de bonnes performance d'accès.

La capacité des SDDSs à assurer une gestion scalable des données distribuées, à travers une réorganisation automatique du fichier par l'acquisition de nouveaux nœuds, ouvre de nouvelles perspectives au stockage et à la gestion de très grands volumes de données (bases de données relationnelles-objets, serveurs WEB, systèmes temps réel, serveurs vidéo, calcul scientifique, ...).

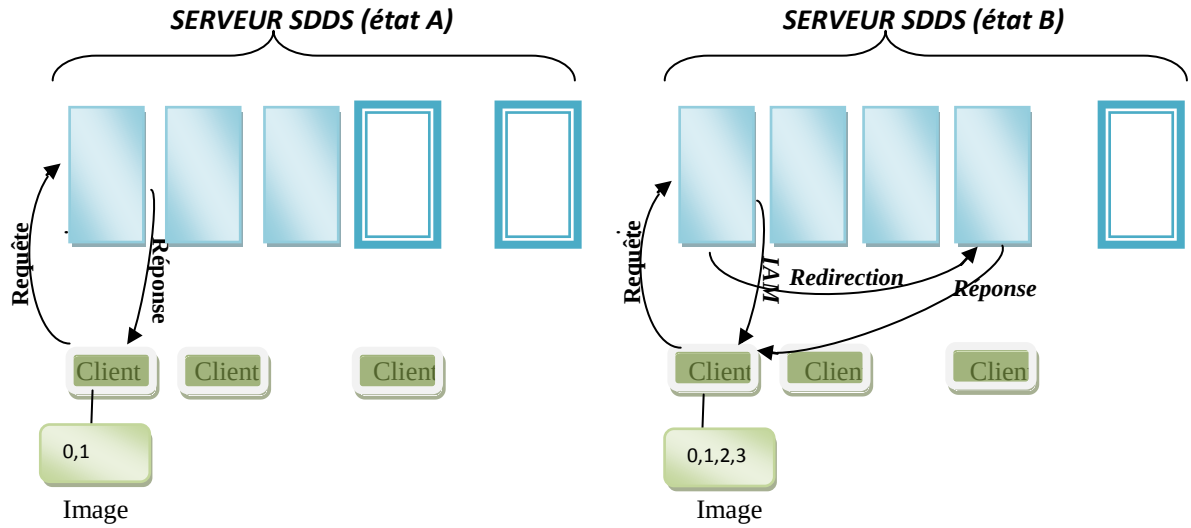


Figure 0-1 : Accès à un fichier SDDS

II.4.Caractéristiques des SDDSs

Les SDDSs possèdent plusieurs caractéristiques qui découlent essentiellement des systèmes distribués. Ce sont essentiellement : La scalabilité, la distribution et la disponibilité **[KAR97].**

II.4.1.La scalabilité

La scalabilité (*scalability* en anglais, peut se traduire par extensibilité d'un système) est le fait de maintenir les performances d'un système quand le volume de données stockées augmente. Une structure de données scalable est caractérisée par le faite que :

- Le temps d'accès est indépendant du nombre d'éléments stockés, il est plus ou moins constant;
- Elle peut prendre en charge n'importe quelle quantité de données, il n'y a aucune limite théorique à partir de laquelle les performances se dégradent;
- En outre, la structure doit pouvoir augmenter et diminuer de taille progressivement et de manière élégante et flexible sans avoir recours à une réorganisation totale.

II.4.2.La distribution

Il arrive dans certaines situations que la quantité de données manipulées dépasse de loin la capacité de stockage et de traitement d'une seule station de travail : même si la capacité de stockage d'un ordinateur est largement suffisante, sa capacité de calcul est généralement insuffisante pour manipuler une telle quantité de données. Les SDDSs permettent la distribution des données sur plusieurs stations de travail (serveurs) autorisant ainsi la gestion de fichier de très grande taille. C'est la technique la plus adaptée pour bénéficier de la puissance de stockage cumulée du multi-ordinateur. Cette stratégie permet aussi une parallélisation accrue et un équilibrage de charge plus facile à maintenir.

La distribution s'impose donc afin de pouvoir exploiter un plus grand nombre de ressources disponibles de façon optimale.

II.4.3.La disponibilité

Comme n'importe quel système, les machines sont vulnérables aux pannes, et les données risquent d'être perdues. La structure de données doit être capable de reproduire les données perdues; soit en dupliquant les données soit en rajoutant des informations qui permettent de reconstruire les données perdues, tels que les blocs de parité. La disponibilité est une caractéristique si importante qu'on accepte de perdre l'espace disque ou de consommer un temps CPU supplémentaire pour l'assurer. Un cas pratique des systèmes qui exigent cette propriété est celui des systèmes bancaires.

LH est une méthode de hachage pour les disques extensibles ou les fichiers mémoires. Un fichier LH est une collection de cases (pages) dans le disque ou dans la RAM adressables via un catalogue de pair de fonction de hachage $h_i$et h_{i+1}; *i*=0, 1, 2,…... Cette fonction (h_i) hache les clés (primaires) en N*****2^iadresses, tel que N≥ 1 est le nombre de cases initiales.

Le fichier s'étend par l'effet des insertions qui provoquent les éclatements des cases débordées. Une valeur spéciale **n** appelée ***pointeur*** s'accroît avec l'extension du fichier (plus précisément il passe de 0 à N, puis de 0 à 2*N**,**…etc.), il indique la case suivante à éclater ainsi la fonction $h_i$ou h_{i+1}que doit être appliquée à la clé courante.

Le résultat du hachage est l'adresse de la case où doit être insérée la clé donnée, s'il n'y a pas de débordement la clé est insérée à cette case, si non un éclatement s'effectue à la case pointée par **n** (qui sera incrémenté) et la moitié de ses clés seront transférées vers une nouvelle case d'adresse (N*n) +2^i.

L'éclatement engendré par les collisions est appelé éclatement ***non contrôlé***, un autre type d'éclatement existe, qui est l'éclatement ***contrôlé*** qui résulte du contrôle du facteur de chargement par un certain seuil. Le processus d'éclatement peut continuer indéfiniment et chaque éclatement donne une case à la fois.

II.5.Classification des SDDSs

Plusieurs recherches sont effectuées dans le domaine des SDDSs. Ces recherches consistent souvent à prendre la version centralisée d'une structure de données et d'essayer de la distribuer. Selon la stratégie de distribution de données adoptée, les SDDSs peuvent être classées en deux grandes familles à savoir : Les SDDSs basées sur la distribution par hachage et celles basées sur les arbres.

II.5.1.SDDSs basées sue le hachage

Le hachage dynamique est un concept élaboré par **Litwin** et **Larson**en 1978. Il utilise des fonctions dynamiques, ce qui permet d'adapter la fonction d'accès selon les données existantes sur le fichier et la rendre ainsi la plus uniforme possible pour minimiser le nombre de collisions.

Plusieurs structures de données distribuées utilisent le hachage comme méthode d'accès à des données distribuées, notamment LH* **[LIT93]**, EH* **[HIL97]** ou DDH **[DEV93]**.

II.5.1.1.**Hachage linéaire distribué (LH*)**

La famille des SDDSs LH* utilise une version distribuée du hachage linéaire (LH : Linear haching) **[LIT80]**.

Description de LH

LH est un algorithme de hachage extensible par lequel on étend l'espace d'adressage primaire d'un fichier disque pour éviter l'accumulation de débordements et la détérioration des performances d'accès. Un fichier LH est divisé en une ou plusieurs cases, chaque case peut contenir b données au maximum, les données de débordements sont gérées indépendamment dans des structures (généralement des listes linéaires chaînées) associées chacune à une case.
Le nombre de cases n'est pas fixe, il augmente avec les insertions et diminue avec les suppressions.

Le fichier contient au départ N cases, l'accès se fait en appliquant aux clés une fonction$h_i$définie comme suit : h_i(c)= c mod (N*2^i), avec *i* représente le niveau de fichier, il est initialement nul et c est une clé primaire.

Lorsqu'une case déborde on dit qu'il y'a une collision sur cette case, la clé en débordement est insérée dans une liste associée.

L'amélioration apportée par LH est qu'à chaque collision il y'a un éclatement d'une case indiquée par la variable spéciale n appelé ***pointeur*** (n n'indique pas forcément la case qui déborde). Une partie des clés de la case débordée sera transférée vers une nouvelle case *S* crée en fin de fichier (*S*=2^i.N+n). Le contenu de *S* est accédé par la fonction h_{i+1}.

Pour bien comprendre le mécanisme d'éclatement dans LH, examinant l'exemple suivant :

La figure suivante montre l'état initial d'un fichier LH.

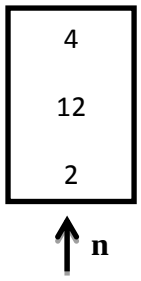


Figure 0-2 : Etat initial d'un fichier SDDS de type LH

Selon la fonction d'accès H_0, l'insertion de la nouvelle clé 11 engendre un débordement dans la case 0 (i.e. H_0 (11)=11mod2 =1). D'après le principe de LH, cette clé est insérée en débordement de la case 0 et la case pointée par n=0 sera éclatée en deux cases selon la fonction $H_1 = c \bmod N * 2^1$ comme montre la figure suivante :

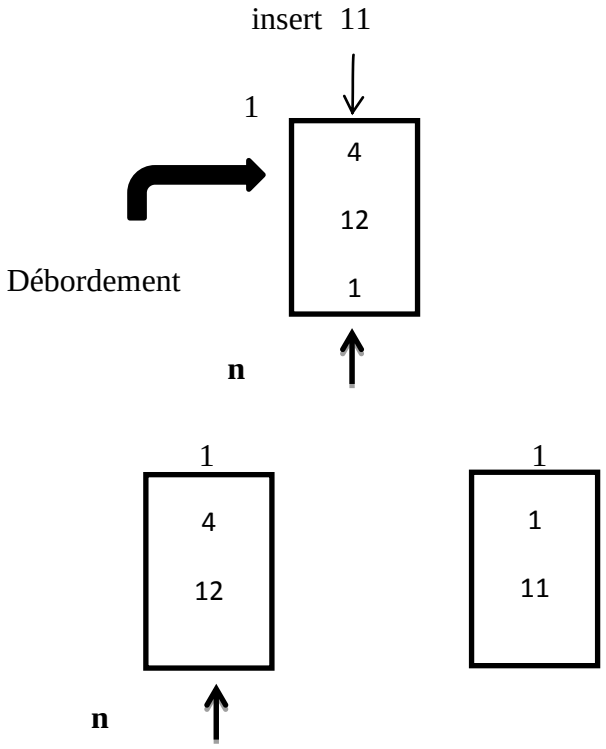
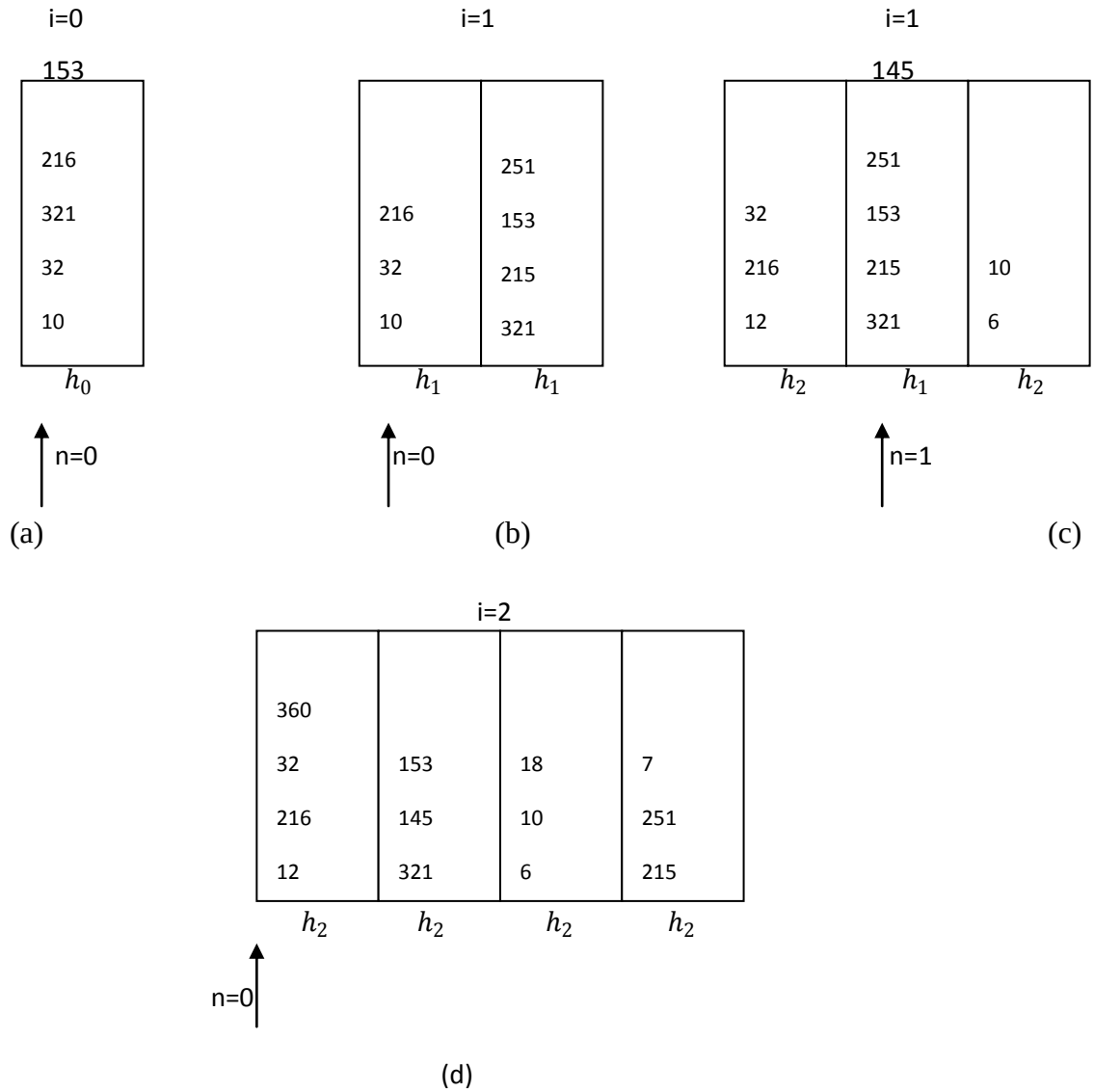


Figure 0-3 : *Éclatement d'une case LH*

Si n=N*2^i alors n a dépassé le nombre total de cases (les cases sont numérotées à partir du 0) alors on remet n à 0 (i.e. les éclatements des cases dans LH se font d'une manière cyclique) et le niveau de fichier est incrémenté (i.e. A chaque cycle d'éclatement, le nombre de cases est double), la nouvelle fonction d'adressage sera h_{i+1} au lieu de h_i.

Donc on peut présenter des déférents états d'un fichier LH dans la figure suivante : (Figure II-5 : Hachage Linéaire.) :



(a) fichier initial W/153 causer une collision à la case 0.

(b)après qu'éclatement de la case 0 et des insertions de 251 et de 215.

(c) insertion de collision de 145 causer d'un éclatement de la case 0 ; insérer 6 et 12.

(d) insertion de 7 faire d'un éclatement à la case 1 ; insérer 360 et 18.

Figure 0-4 : Hachage Linéaire.

Adaptation de LH dans les environnements distribués

L'adaptation de LH aux environnements distribues consiste à affecter à chaque **serveur LH*** une adresse logique, une case de données et un niveau et il est accédé par des applications tournantes sur des machines autonomes dites **clients LH***. Chaque client LH* possède une image sur le système qu'est constituée d'un niveau de la fonction d'accès ainsi le pointeur de la prochaine case à éclater.

LH se base sur le principe qu'il existe un seul compteur de niveau i et un seul pointeur vers la prochaine case à éclater n. Or les contraintes des environnements distribués font que ce principe ne peut pas être respecte. D'où la nécessité d'utiliser un mécanisme qui permet de maintenir la cohérence de ces paramètres. La solution la plus répandue consiste à utiliser un site central appelé coordinateur d'éclatement.

Description de la méthode LH*:

La SDDS LH* : parmi les SDDSs on a la SDDS LH* (Hachage Linéaire)

Dans LH* l'accès aux données utilise une fonction de hachage (pour adresser les données), cette fonction est approximative, chaque client a sa propre fonction, qui avait la forme h_i(cle)= cle mod 2^i N. Tel que i signifie de combien de fois le nombre d'éclatements soient doublées (de manière exponentiel), N nombre initial de cases du fichier (on prend souvent N=1) alors cette fonction fonctionne d'une manière dynamique.

LH* est constituée de trois composants qui sont :

- **Client LH*** : Contient les paramètres (n : le pointeur, i : le niveau) qui constituent son image. Suivant les règles des SDDSs, cette image du client n'est pas nécessairement correcte. Alors, lorsque le client fait une erreur d'adressage, le serveur LH* renverrait au client un message d'ajustement d'image. Le message permet au client d'actualiser son image, il comporte j (le niveau de la case correcte) et a (l'adresse logique de la case correcte). La différence entre les clients et la fonction de hachage est que chaque client a ses propres valeurs (i et n), tel que n est une valeur qui indique la case qui va éclater, après l'éclatement n s'incrémente (à tout moment , le pointeur n indique donc la prochaine case à éclater) et vis vers ça jusqu'à ce que n soit égal à 2^i, à ce moment-là i va incrémenter et n va remiser à zéro (n modulo 2^i

ou n modulo la taille du fichier), alors cette opération sera répétée de manière cyclique (et l'inverse après une suppression). La (Figure 0-5 : Principe de LH*. propose un exemple que chaque client a sa connaissance sur la fonction de hachage, et les valeurs exactes de n et i sont ambigües aux clients, alors chaque client veut envoyer une requête, il peut faire ça, il applique ses valeurs propres n et i et calcule h (clé) comme l'algorithme A1 **[HID07]**.

- **Serveur LH* :** Reçois le message (requête) d'un client, qui n'est pas forcément égale à j (le niveau de la case correcte), qui est l'image représentée par cette valeur locale, un serveur LH* commence par vérifier si la requête lui est destinée ou non, et donc traite la requête ou la renvoie à un autre serveur.
- **Coordinateur d'éclatement** : Il coordonne les éclatements des cases LH*. Il est le seul à avoir une image correcte du fichier. Il est implanté au-dessus de la première case LH*. Quand une case LH* déborde, elle l'avertie au coordinateur.

L'opération d'éclatement ne concerne pas forcément la case qui a débordé. Elle concerne plutôt la case dont le numéro correspond à un pointeur d'éclatement, noté n, dans l'objectif d'instaurer une meilleure distribution des actions d'éclatement des cases (Tab 2.1); n, initialisé à 0, est incrémenté de 1, suite à chaque éclatement. Il ne redevient nul que s'il dépasse la valeur $2^i - 1$; son évolution est conforme à l'évolution de i:

i	0	1	2	3	4	5	…	i
n	0	0,1	0, 1, 2,3	0,1,..,7	0,1,..,15	0,1,..,31	…	$0,1,..,2^i-1$

Tableau 0-1: Valeurs de n en fonction de i

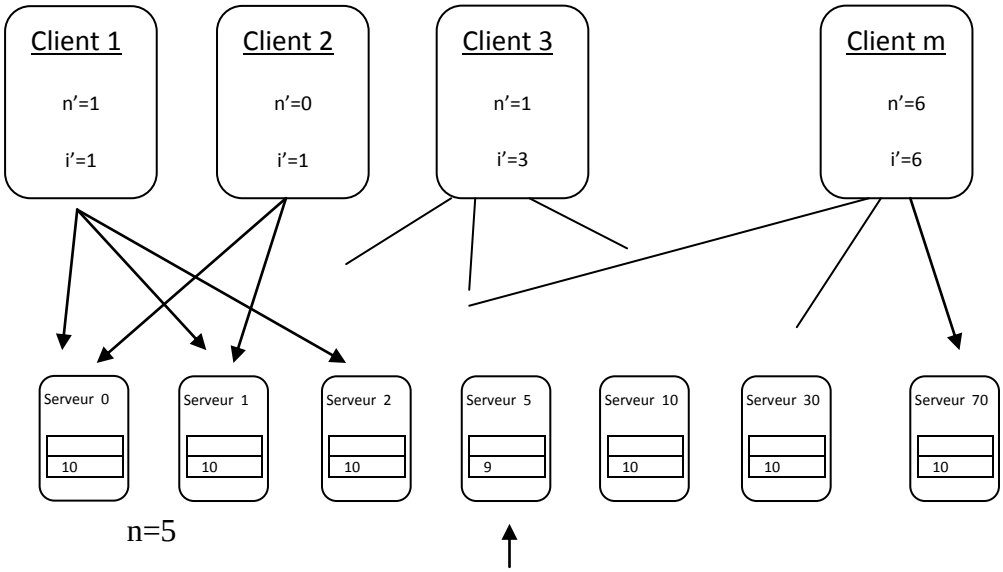


Figure 0-5 : Principe de LH*.

Nous les présentons conformément à l'ordre des étapes correspondant à une requête déclenchée par un client:

(1) Le client envoie sa requête (insertion, modification, suppression ou recherche), relative à la clé c, au serveur numéro a, en appliquant l'algorithme suivant :

1. a= $h_{i'}$ (clé) ;
2. si a<n′ alors a= $h_{i'+1}$(clé) ; fsi
3. envoi de clé à la case a ;

Algorithme 0-1 : hachage de client

En cas d'erreur d'adressage, la requête relative à la clé c n'arrive donc au serveur concerné (correct) qu'après quelques éventuelles redirections.

(2) Le serveur n° a, qui reçoit la clé c, exécute l'algorithme suivant:

1. Réception de clé par la case a
2. $j_{a'} \leftarrow j_a$; /* j est la valeur présumée du niveau de hachage dans le serveur a */
3. a'$\leftarrow h_j$ (clé) ;
4. Si (a' = a) Alors
5. /* Le client a calculé correctement l'adresse d'envoi */
6. Accepter clé ;
7. Sinon/*Cas d'erreur d'adressage */
8. a" $\leftarrow h_{j-1}$(clé) ;

9. Si (a< a" < a') alors /* Cas où la redirection peut concerner un serveur plus proche */
10. a' ← a" ;
11. fsi
12. Envoi de c à la case a′ ; fsi

Algorithme 0-2 : Hachage de serveur a

Il est à noter que cet algorithme garantit que l'image de tout client est contenue dans l'extension actuelle du fichier et qu'un client ne répète pas une même erreur **[LIT93]**.

(3) En cas de redirections, le serveur correct envoie au client ayant fait l'erreur d'adressage un message correctif " IAM (Image Adjustment Message)", lui permettant d'ajuster son image. Ce message comporte la valeur du niveau de hachage j de la case numéro a, à laquelle le client avait envoyé c. Cet ajustement permet de corriger l'erreur d'adressage afin de se rapprocher le plus possible du serveur correct et de réduire ainsi le nombre de redirections. Notons qu'il a été prouvé que les redirections dans une SDDS LH* sont très rares: toute requête soumise trouve le serveur correct au maximum en deux redirections **[LIT93][LIT94]**.

(4) Le client recevant l'IAM, met à jour son image (i', n') en appliquant l'algorithme suivant:

1. Si (j> i') Alors
2. i' ←j – 1 ;
3. /* En effet, les 2 fonctions de hachage à utiliser sont dorénavant $h_{i'}$ et $h_{i'+1}$ */
4. n' ← a + 1;
5. Si (n' ≥ $2^{i'}$) Alors
6. n' ← 0 ; i' ← i' + 1;
7. /* Conformément à l'évolution du pointeur d'éclatement */
8. fsi
9. fsi

Algorithme 0-3 : Message d'adjustement IAM

Pour une SDDS LH*, le niveau de hachage global i et le pointeur d'éclatement n sont gérés par un site spécial appelé site coordinateur (SC) qui est généralement le serveur n°0 **[LIT96a].** Lorsque le SC reçoit un message de débordement (collision message), de la part d'un serveur sur lequel est constaté un débordement, il envoie un message d'éclatement

"Eclatez-vous " vers le serveur pointé par n. Par ailleurs, le SC calcule les nouvelles valeurs de n et de i, en appliquant l'algorithme suivant: (Algorithme 0-4 : calcule de n et i après un éclatement)

1. n←n+ 1 ; /* Incrémentation de la valeur de n; la valeur de i reste a priori inchangée */
2. Si ($n \geq 2^i$) Alors/* Fin d'un cycle d'éclatement constatée : n redevient nul et i s'incrémente */
3. n←0; i←i+1 ;
4. fsi

Algorithme 0-4 : calcule de n et i après un éclatement

Le serveur qui reçoit l'ordre d'éclatement procède comme suit:

(a) Il ordonne la création de la case numéro n + 2j dans un nouveau serveur avec un niveau de hachage local égal à j + 1 ; j étant le niveau de hachage local de la case à éclater ; j a la même valeur que i avant son éventuelle modification.

(b) Il éclate sa case, en envoyant une partie de ses enregistrements à la nouvelle case. Cette partie est déterminée par la fonction de hachageh_{j+1}.

(c) Il met à jour j : j ←j +1.

(d) Il envoie un message de fin d'éclatement au SC.

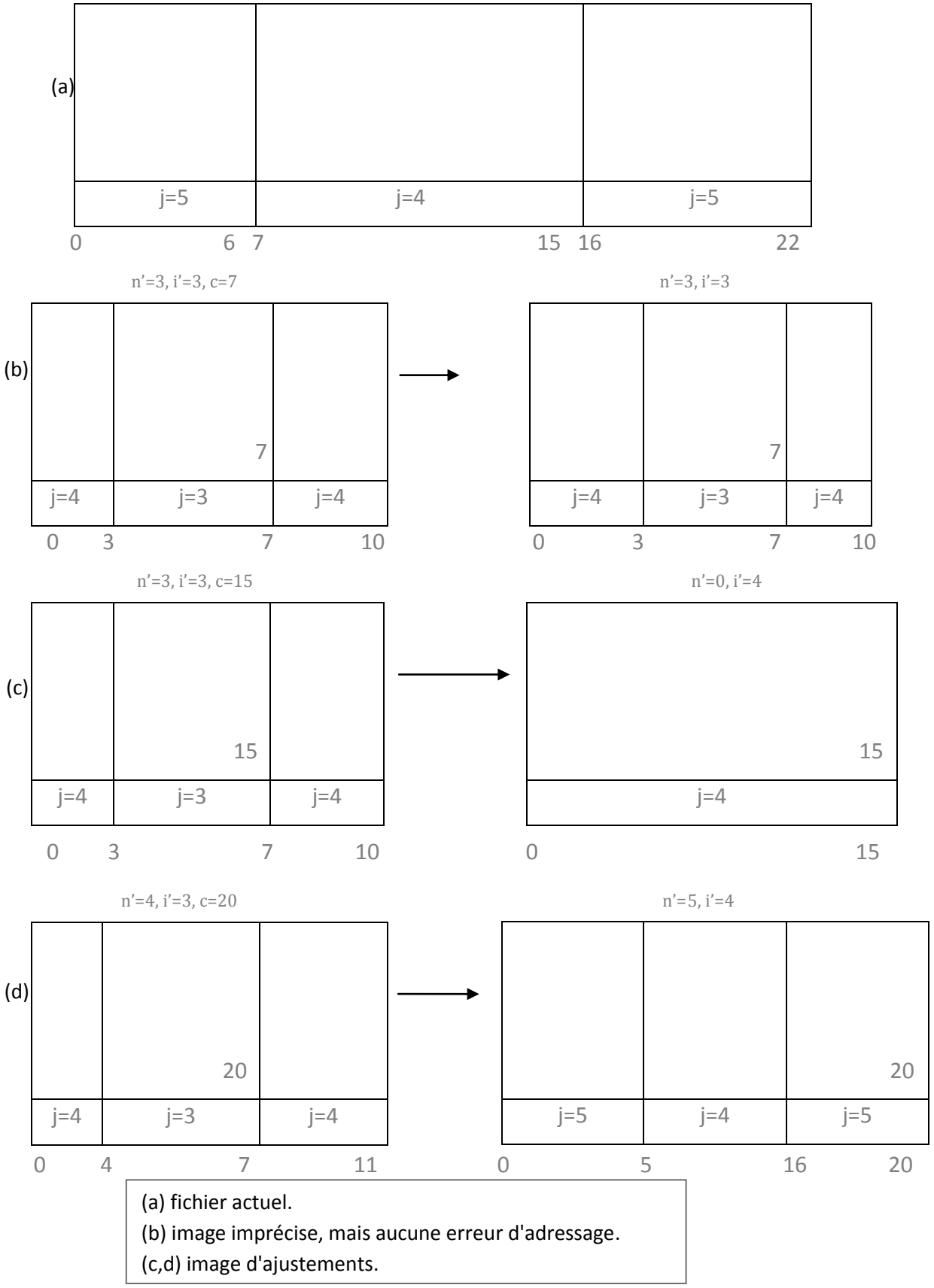
L'exemple suivant explique comment est figuré un fichier LH*

Par exemple dans la figure ci-dessous, on a les valeurs réels de n et i qui sont 7 et 4 successivement ; (partie (a)), et dans la partie (b), on a un client qu'il a les valeurs présumées n'=3, et i'= 3, avec une clef c=7, alors le client va calculer **a**=$h_{i'}(c)=h_{i'}(7)=7 \bmod 2^3=7$ (Algorithme 0-1 : hachage de client). Alors le client va envoyer la clef au serveur 7, le serveur concerné va recalculer $a'= h_j=h_4=7$, alors le serveur 7 va accepter la clef 7 parce que $a = a'$ (Algorithme 0-2 : Hachage de serveur a). D'autre côté dans la partie (c), le client a les valeurs présumées n'=3, et i'= 3, avec une clef c=15, alors le client va calculer **a** $=h_{i'}(c)=h_3(15)=15 \bmod 2^3=7$(Algorithme 0-1 : hachage de client).alors le client va envoyer la clef au serveur 7, le serveur concerné va recalculer $a'=h_j=h_4=15$ (Algorithme 0-2 : Hachage de serveur a), alors le serveur va rediriger la requête vers le serveur 15, et envoyer une image d'ajustement (IAM) vers le client, le client va ajuster leur i' et n' selon (Algorithme 0-3 : Message d'adjustement IAM), et après la réception de requête de clé 15 par le serveur 15, ce dernier aura calculé $a'' = h_{j-1}(15)$; (=7), selon l'algorithme (A2), la vérification de

(a< a′′< a′) n'est pas vérifiée, alors la clé 15 sera acceptée depuis le serveur 15. Et enfin dans la partie (d) a le même scénario de partie (c).

Maintenant supposons qu'un client a les valeurs présumées sont (n'=4 et i'=3) et une clé (c=20), alors au début, le client va calculer $a=h_{i'}$(c)=h_3(20)=20 mod 2^3=4(Algorithme 0-1 : hachage de client).alors le client va envoyer la clef au serveur 4, le serveur concerné va recalculer $a'=h_j=h_5$=20 (Algorithme 0-2 : Hachage de serveur a), alors il y a un erreur d'adressage, il faut réutiliser la fonction de hachage avec (j-1) comme suite : $a'' = h_{j-1}$(20); (=4) le serveur va rediriger la donnée vers le serveur 20 parce que la vérification de (a< a''< a') est acceptée, donc une image d'ajustement (IAM) aura envoyé vers le client, le client va ajuster leur i' et n' selon (Algorithme 0-3 : Message d'adjustement IAM), et enfin, la réception de la donnée de clé 20 par est acceptée le serveur 20, voir la partie (d).

n=7, i=4



(a) fichier actuel.
(b) image imprécise, mais aucune erreur d'adressage.
(c,d) image d'ajustements.

***Figure 0-6** : image d'un fichier LH**

Analyse de performances de LH* [LIT93]

Dans ce qui suit, nous donnons quelques performances de la méthode LH* :

- Le facteur de chargement de fichier LH* est le même que celui de LH, i.e. α≈65-70%.
- Une insertion nécessite généralement un message, trois au pire des cas.
- Une recherche utilise habituellement deux messages, quatre au plus.
- Une opération parallèle sur un fichier de M cases utilise au plus (2M+1) messages

Variantes de LH*

Plusieurs variantes du schéma LH* ont été proposées, les principales variantes sont brièvement décrites dans ce qui suit :

LH*$_{LH}$ [KAR96] est une variante de LH* visant les architectures multiprocesseurs à partage de rien. Ce schéma utilise un premier niveau d'index correspondant à LH*, permettant aux clients d'accéder aux serveurs. Le deuxième niveau d'index correspond à l'organisation interne des cases de données suivant l'algorithme LH.

Les variantes à haute disponibilité et sécurité

Ces variantes sont conçues pour assurer aux systèmes SDDSs la continuité de fonctionner de manière transparente, quand un ou plusieurs de ses sites serveurs tombent en panne. Ces variantes font généralement appel à certains principes de redondance et de récupération de données.

- ***LH*$_M$ (with Mirroring)*** **[LIT96b]**:C'est une SDDS a tolérance de fautes (fault tolérant) qui utilise avec le schéma de base LH* un autre fichier miroir organisé en LH* lui aussi. Des algorithmes sont donnés pour la cohérence du miroir ainsi que pour la récupération d'informations et le remplacement de serveurs. Le principal inconvénient de cette SDDS est le coût de stockage qui est facteur du nombre de répliques.
- **LH*s (*with Stripping*) [LIT97a]** : Cette SDDS introduit le partitionnement, i.e. chaque enregistrement est fragmenté en *k* segments. Chaque segment est inséré dans un fichier segment S_i contenant tous les segments s_i de tous les enregistrements. A chaque enregistrement est ajouté un segment s_{k+1} appelé segment de parité, calculé à partir des segments $s_1, s_2, \ldots, s_k$ et est inséré dans le fichier segment S_{k+1}.

  Un fichier LH*s est un ensemble de *k*+1 fichiers LH* indépendants appelés fichiers segment. La gestion de l'ensemble des fichiers segment ainsi que la récupération est effectuée par un seul coordinateur SC. Notons que cette SDDS LH*s nécessite moins

d'espace de stockage que la SDDS LH*M mais les opérations de mises à jour demandent plus de temps et utilisent plus de messages.

- **LH*G (*with Record Grouping*) [LIT97b]** : Cette SDDS ajoute un mécanisme de haute disponibilité au schéma LH*. Des pages de parités sont ajoutées afin d'augmenter la disponibilité du fichier. Des algorithmes spécifiques à la méthode décrivent la création des pages de parité ainsi que la récupération des pages manquantes.
- **LH*SA (*Scalable Availability*) [LIT98]** : Cette SDDS étend le schéma LH*G pour pouvoir récupérer plusieurs erreurs en même temps. Ceci est rendu possible par l'utilisation de plusieurs fichiers LH* de parité (k-disponibilité).
- **LH*RS (*Scalable Availabilityusing Reed Solomn Codes*) [LIT99]**: Cette méthode améliore le schéma LH*SA par l'introduction d'une gestion dynamique de la k-disponibilité.

En effet, Dans LH*RS, chaque enregistrement appartient à un seul groupe. Le calcul de parité utilise les codes **Reed Solomon** ou **Codes RS [MOU04]**, c'est un outil mathématique très performant, de plus il a prouvé son efficacité dans la pratique.

Une variante multidimensionnelle : IH*

IH* **[BOU02]** est la première structure de données distribuées et scalable basée le hachage linéaire multidimensionnel. IH* constitue, à la fois, une extension du hachage par interpolation de Burkhard **[BUR83]** aux environnements distribués et une introduction de la notion d'ordre et de l'aspect multidimensionnel a la SDDS LH* de Litwin **[LIT93]**. Son but est d'offrir un fichier multi-attributs distribue et scalable avec préservation de l'ordre des enregistrements.

Une variante pour les environnements P2P : LH*P2P

LH*P2P[YAK07] est une proposition qui consiste à adapter la SDDS LH* aux environnements pair-à-pair. Le gain majeur de cette proposition est la réduction du nombre maximal de renvois des requêtes SDDS. En effet, si le fichier LH*P2P, ne comporte que des nœuds pairs ou serveurs, toute requête de recherche d'un enregistrement a besoin au maximum d'un renvoi.

II.5.1.2.**Hachage extensible distribué**

Hilford, Bastani et Cukic ont proposé une SDDS: EH* **[HIL97]**. EH* distribue les articles en utilisant l'algorithme de hachage extensible (EH : Extensible Hashing) proposé par Fagin**[FAG79]**.

II.5.1.3.**DDH (Distributed Dynamic Hashing)**

Le schéma DDH **[DEV93]** est basé sur le hachage dynamique **[LAR78]**, il généralise le principe de hachage dynamique aux environnements distribués.

La SDDS DDH a été proposé pour remédier aux problèmes liés au coordinateur d'éclatement dans LH* et assurer une autonomie locale des serveurs. Chaque serveur décide lui-même d'éclater ou de fusionner ses cases. Une politique constante est commune à tous les serveurs (pas de serveurs particulier), i.e. le même algorithme tourne sur tous les serveurs.

La performance de DDH dans le cas de recherches et d'insertions aléatoires est proche de celle de LH*, si l'on assume que les clients inactifs sont rares mais pour les redirections (forwards), elles peuvent nécessiter plus de messages par rapport à LH*.

II.5.1.4.**Autres extensions :**

D'autres recherches ont été faite également par exemple dans **[LUK08]** sur une autre variante de SDDS basée sur le hachage linéaire (LH*) dénommé par DSDDS (Decentralized SDDS).

II.5.2.SDDSs basées sur les arbres

RP* est une famille de SDDS établissant un ordre par clé au niveau des enregistrements de la base, elle a été proposée en 1994 **[LIT94]**. Un enregistrement est constitué d'un champ clé et des champs non-clé. La clé permet d'identifier l'enregistrement et prend sa valeur dans un espace de clés totalement ordonné. Les enregistrements se trouvent dans des cases appelées cases (buckets) ayant une capacité de b enregistrements avec b >>1. Chaque case représente un serveur RP. Les enregistrements sont disposés logiquement dans l'ordre ascendant des clés.

Sous famille de RP*

La SDDS RP* se subdivise en trois sous famille à savoir : RP*N, RP*C et RP*S. La figure suivante présente les relations qui existent entre ces différents schémas de base.

- **RP*N(No Index)** : définit un partitionnement des enregistrements par intervalles définis sur l'espace des clés. Le protocole de communication s'appuie sur des messages multicast émis par les clients et des messages unicast pour la réponse des serveurs.
- **RP*C(Client Index)** : ajoute à RP*n une image au niveau de chaque client. Une telle organisation favorise l'utilisation de messages unicast aussi bien lors des requêtes clients que des réponses serveurs; le multicast n'est employé que pour les redirections.
- **RP*S (Server Index)** : ajoute a RP*c un index reparti sur des serveurs de nœuds, ce qui permet l'utilisation de messages unicast lors des redirections.

II.5.2.1.**Variantes RP* de hautes disponibilités**

RP*Rs est une SDDS proposée en 2001 **[DIE01]** pour pallier le problème de disponibilité des données dans le schéma de base RP*. RP*Rs inspire son mécanisme de fonctionnement de celui de SDDS LH*Rs.

II.5.2.2.**Une variante multi-attributs**

k-RP* est une SDDS offre la possibilité de traiter des requêtes multicritères (i.e. sur plusieurs champs). C'est une adaptation des B-arbres multidimensionnels aux environnements distribues. Elle est similaire a RP* sauf qu'elle permet de traiter des requêtes contenant plusieurs clés.

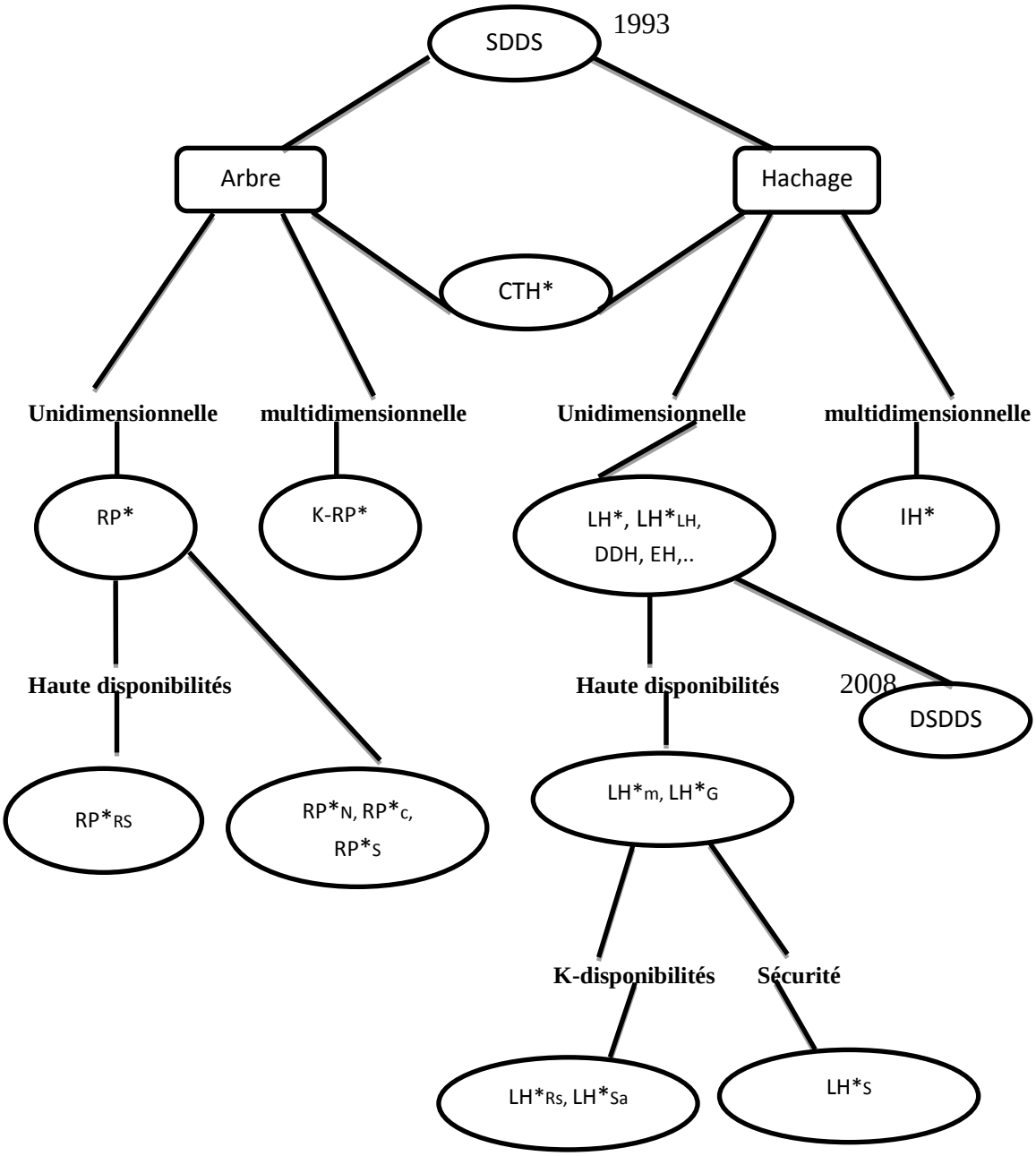


Figure 0-7 : Classification des SDDSs[LOU05].

II.6.Conclusion

Nous avons présenté dans ce chapitre les SDDS en générale, et nous avons mis l'accent sur les SDDS de type LH*. Une des caractéristiques principales de ces méthodes est l'utilisation de la mémoire centrale comme support de stockage permanent. En cas de panne d'un ou plusieurs serveurs, la cohérence des données n'est donc plus assurée. La journalisation sur disque est une des solutions que l'on peut envisager. Cet aspect sera discuté dans le prochain chapitre.

Chapitre III

journalisation des opérations de mises à jour dans LH*

III.1.Introduction

On a vu dans le chapitre précédent les SDDS de type LH*, et on a vu aussi comment ils fonctionnent. Ainsi, lors des insertions provoquant des éclatements ou des suppressions provoquant des fusions de cases, il y a des transferts de données entre les serveurs. Ces déplacements de données doivent être journalisés de manière appropriée afin d'éviter toute perte de données en cas de panne.

Dans ce chapitre, nous proposerons une solution qui pourra assurer le recouvrement des données après une panne, cette solution utilisera un journal principal Redo (au niveau du coordinateur) et permet reprendre tous les serveurs. L'idée est de remplir toutes les cases par d'anciennes données (au moment du CheckPoint), et à partir de ce dernier CheckPoint on va exécuter le journal Redo depuis la mémoire stable. Nous aurons ainsi le dernier état cohérent avant la panne. Une deuxième solution est aussi proposée pour éviter la gestion centralisée du journal Redo principal.

III.2.Architecture du système proposé

Dans notre proposition, chaque serveur LH* contiendra en plus des modules de gestions de transactions traditionnelles (gestionnaire de transactions, de concurrences et de données), les structures de données suivantes :

- une case renfermant les données
- une table de transactions actives. Ces deux structures sont maintenues en mémoire centrale.

Chaque entrée dans la table de transactions contient la liste des instructions qu'une transaction a exécutée dans ce serveur. C'est son journal Undo/Redo. Voir (Figure 0-1 : Composants d'un serveur SDDS (a)).

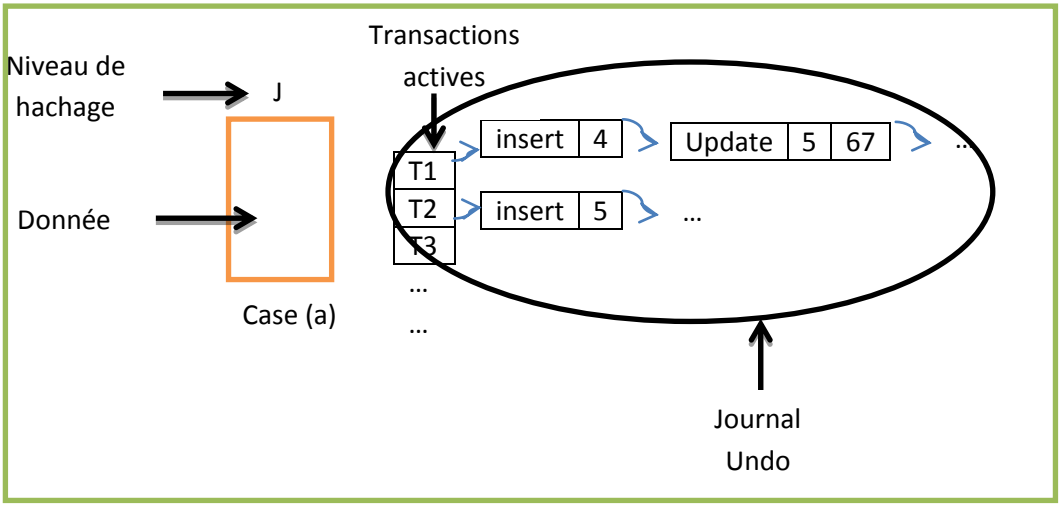


Figure 0-1 : Composants d'un serveur SDDS (a)

Les clients SDDS sont responsables de la gestion des transactions. C'est-à-dire pour chaque opération de la transaction, le client calcule le numéro du serveur concerné à l'aide de la fonction de hachage LH* et envoie l'opération au serveur SDDS. Quand un client désir valider ou annuler sa transaction il envoie l'opération de terminaison (validation ou annulation) à tous les serveurs SDDS qui ont participé aux opérations de mises à jour de la transaction.

Chaque fois qu'une transaction modifie la case du serveur (ajout, suppression ou modification de données), cette opération sera immédiatement enregistré dans le journal Undo (Tableau 0-1 : représentation d'une structure d'un journal Undo) de la transaction au niveau de ce serveur.

Lors de la validation d'une transaction, le gestionnaire de données envoie son journal Redo au coordinateur qui se chargera de sauvegarder son contenu dans le journal Redo global sur disque. Ensuite l'entrée de la transaction dans la table des transactions actives est supprimée y compris son journal Undo/Redo.

En cas d'annulation d'une transaction, le gestionnaire de données va parcourir le journal Undo de la fin vers le début pour exécuter ses opérations et retourner l'état de la case à son état initial avant le lancement de cette transaction. L'entrée de la transaction (y compris son

journal Undo/Redo) dans la table des transactions actives sera aussi supprimée à la fin de l'annulation.

Le coordinateur stock dans son propre disque le journal Redo global contenant les images après de toutes les transactions validées du système. Il renferme aussi les dernières copies validées des cases de serveurs (CheckPoint) voir (Figure 0-4 : Les données dans le disque du serveur SDDS global).

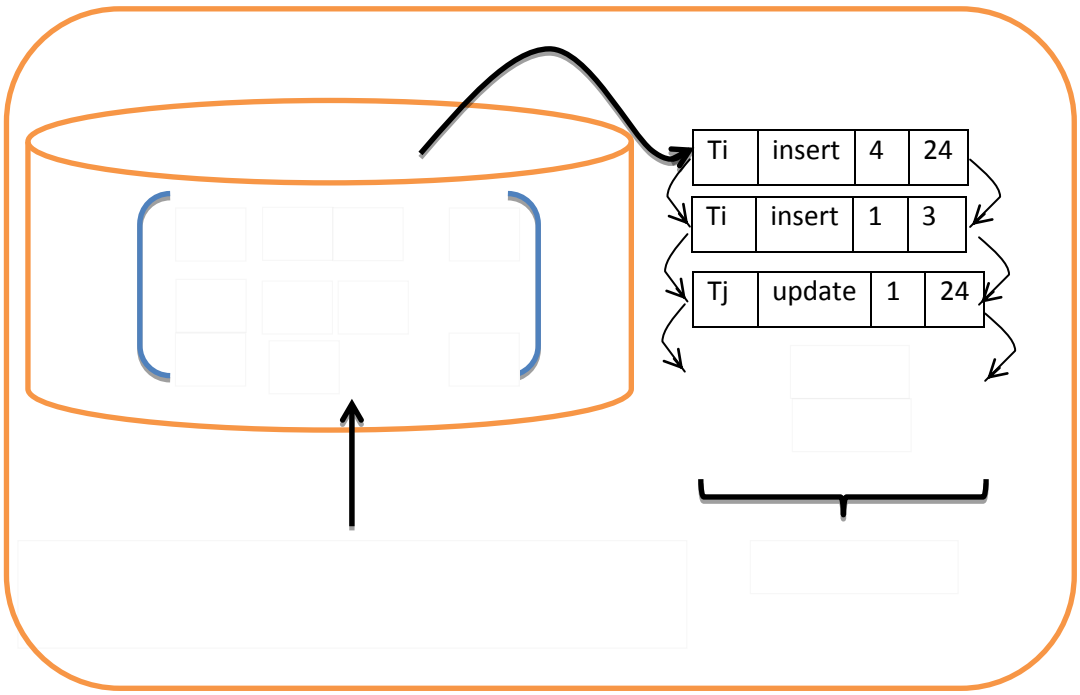


Figure 0-2 : Les données dans le disque du serveur SDDS global

III.3.Transfert de données entre cases

Comme il a été dit dans le chapitre II, certaines insertions vont provoquer des éclatements de cases et certaines suppressions vont provoquer des fusions de cases. Ce type d'opérations génère donc des transferts des données entre serveurs afin de garder un bon équilibrage dans le stockage de données. Donc certaines parties des journaux doivent aussi être déplacées pour suivre pour pouvoir correctement valider et annuler les transactions ayant manipulé ces données transférées.

III.3.1.Eclatements

L'opération d'éclatement (Figure 0-12 : journal Undo pendant une opération d'éclatement.) doit faire un parcours de toutes les données qui résident dans la case *n* pour

recalculer leurs nouvelles adresse aves la fonction $h_{j+1}(c)$; tel que c, la clé de la donnée. si le résultat est différent de *n* alors transférer cette donnée vers la nouvelle case d'adresse (n $+2^j$), sinon laissé cette donnée dans sa case originale. Après cette opération, le niveau de hachage j de la case *n* sera incrémenté, et le niveau de hachage de la nouvelle case prend la nouvelle valeur j de la case mère.

Le compteur n au niveau du coordinateur sera aussi incrémenté, puis comparer avec 2^i s'ils sont égaux, le compteur n revient à 0 et le i est incrémenté.

Durant l'opération d'éclatement la case du serveur où a lieu l'insertion ainsi que celle du nouveau serveur reste verrouillées.

Exemple :

Supposons qu'il existe d'un fichier formé par deux cases voir (Figure 0-3 : Opération d'éclatement. partie A), et le chargement de ce fichier est actuellement de 60%. Si on ajoute une donnée (par exemple de clé 9), le chargement du fichier va augmenter à 80% alors un éclatement va toucher la case *n* (dans cet exemple i=1 et n=0), et le résultat est montré dans la partie B de la figure.

Nous avons su que l'éclatement oblige l'utilisation de la fonction de hachage, pour cela les données ne se divisent pas forcément en deux parties égales (50% à chaque case), et l'exemple ci-dessous prouve ce point.

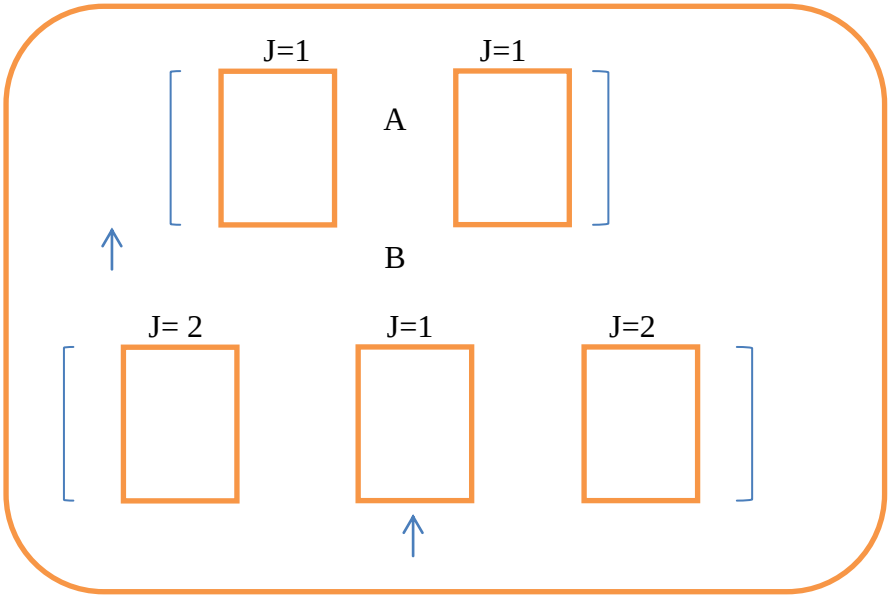


Figure 0-3 : Opération d'éclatement.

III.3.2..Fusions

L'opération de la fusion (Figure 0-12 : Reprise après panne et fusionnement de deux cases.) oblige le **GT** de verrouiller les deux cases concernées : la dernière case du fichier (2^i +n-1) et la dernière case à avoir éclatée (la case n-1 si n>0 ou alors la case 2^i-1 si n=0). En plus les données de la dernière case sont déplacées vers la dernière case à avoir éclatée. Le serveur associé à la dernière case sera libéré et le compteur n décrémenté. Le niveau de hachage j de la case n, sera aussi décrémenté.

Exemple :

A l'inverse de l'exemple précédent (Figure 0-3 : Opération d'éclatement.) si on a premièrement la partie (B), et nous allons supprimer la clé 9 le seuil de fichier va diminuer vers30%, alors après la suppression de cette clé le gestionnaire va fusionner les deux cases et le résultat est montré dans la partie (A) de la même figure.

III.3.3..Pannes et reprise après panne

Parmi les évènements qui peuvent advenir sur une case, la panne et la reprise après panne. Car si un serveur tombe en panne cela implique la perte de toutes ses données, ainsi que le journal Undo (*Figure 0-11 :* panne d'un serveur.). Après une panne il faut appliquer une méthode pour récupérer les données qui sont perdues (Figure 0-12 : Reprise après panne et fusionnement de deux cases.), plus de détails seront présentés plus loin dans ce chapitre (III.6).

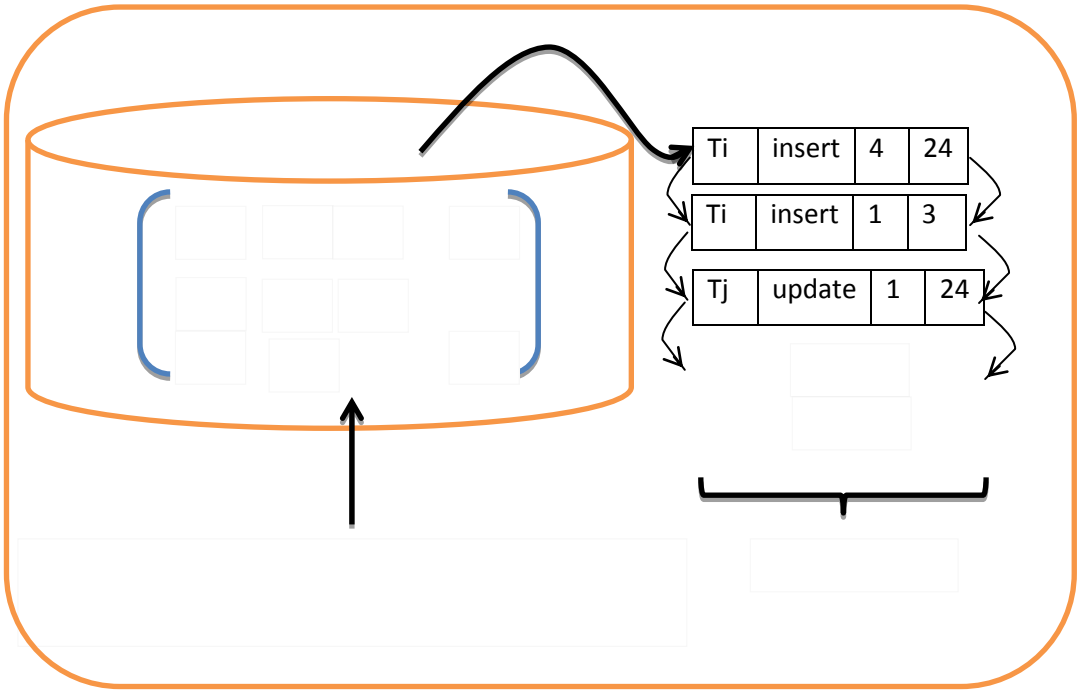


Figure 0-4 : Les données dans le disque du serveur SDDS global

III.4.Verrouillage des données

Les clients lancent des transactions qui s'exécutent d'une manière concurrente, (manuellement on peut lancer une transaction en même temps que la précédente est en cours d'exécution), ces exécutions peuvent ajouter, supprimer, chercher ou bien modifier des données, ces données ont des clés ; chacune a sa propre clé, qui permet de trouver la case qui la contient ; bien sûr après l'application de la fonction de hachage comme nous avons vu dans le chapitre précédent. Ces opérations doivent-être sécurisées par des verrous selon le type de l'opération, on divise le verrouillage en trois types comme suit.

III.4.1.Pendant la modification

Pendant une opération de modification, le verrouillage sur la clé concernée est suffisant afin de garantir l'homogénéité du fichier quand il n'existe pas des transactions qui ajoutent ou suppriment des données à cette case. Car ces dernières peuvent provoquer des éclatements ou des fusions de cases, impliquant donc le déplacement des données entre cases. Alors plusieurs transactions peuvent accéder à la même case pour modifier des données, le verrouillage sur la

clé est suffisant, mais il faut interdire l'éclatement de cette case comme nous avons dit dans la partie (III.3)

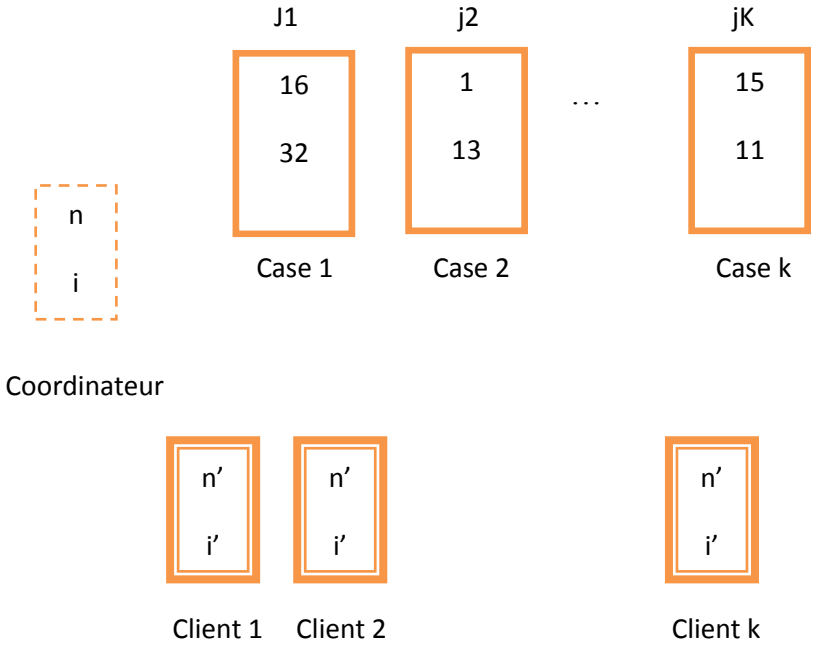


***Figure 0-5:** Représentation des données*

III.4.2..Pendant l'ajout et la suppression

Pendant l'ajout et la suppression des données, le verrouillage seulement sur la clé n'est pas suffisant pour garantir l'homogénéité du fichier (des cases), pour cela il faudra verrouiller toute la case qui concerne l'opération d'ajout (Figure III-6 : Suppression d'une donnée.) ou de suppression (Figure III-7 : Insertion d'une donnée.), car opérations peuvent nécessiter un éclatement (Figure III-3 : Opération d'éclatement.), ou bien une fusion de cases (Figure III-18 : Reprise après panne et fusionnement de deux cases.). Alors pour éviter les problèmes d'incohérence, nous bloquons les autres transactions voulant accéder à cette case jusqu'à ce que la première termine son exécution par une validation (Figure III-4 : Scénario de validation d'une transaction.), ou bien par une annulation (Figure III-13 : Annulation d'une transaction.).

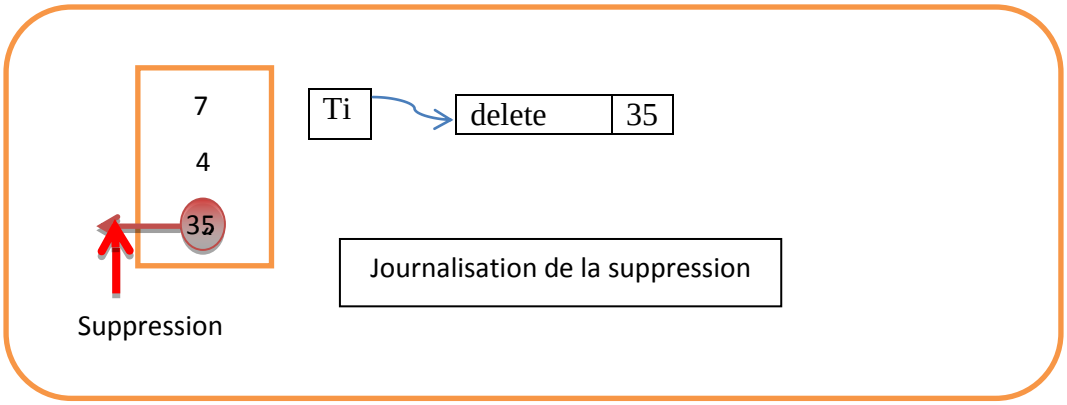


***Figure 0-6 :** Suppression d'une donnée.*

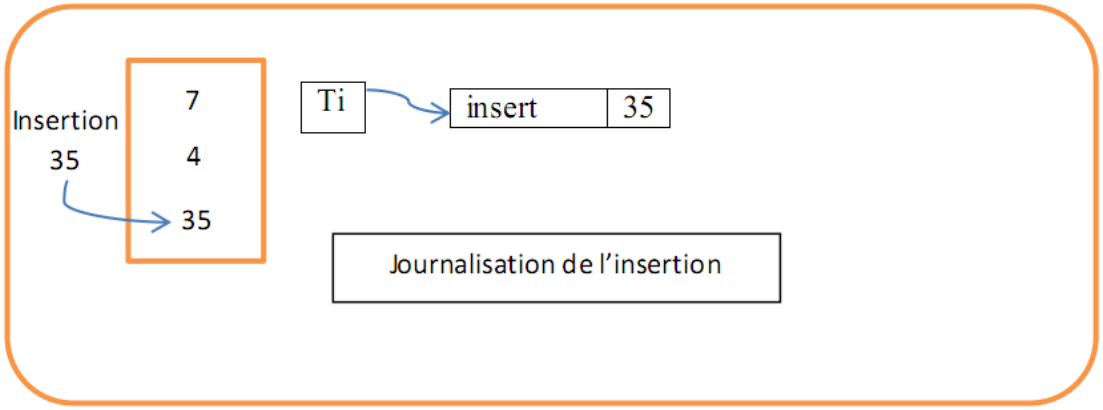


Figure III-7 : Insertion d'une donnée.

III.4.3.Pendant l'annulation

Une annulation d'une transaction (Figure III-13 : Annulation d'une transaction.) signifie qu'il faut défaire tous les changements qu'elle a apporté aux cases du fichier. Cela peut inclure un ensemble d'ajouts, de suppressions et des modifications, qui ont pu générer des éclatements et/ou des fusions. L'annulation utilise pour cela le journal Undo/Redo avec un parcours inverse (de la fin vers le début). De plus il peut exister des données journalisées dans une case et après un éclatement ou une fusion elles vont aller vers une autre case.

Pour cela nous proposons une solution au problème du journal Undo/Redo, on va la voir dans la partie de journalisation ci-dessous, et pour le verrouillage il faut verrouiller toutes les cases qui ont été manipulées par la transaction qui a été annulée.

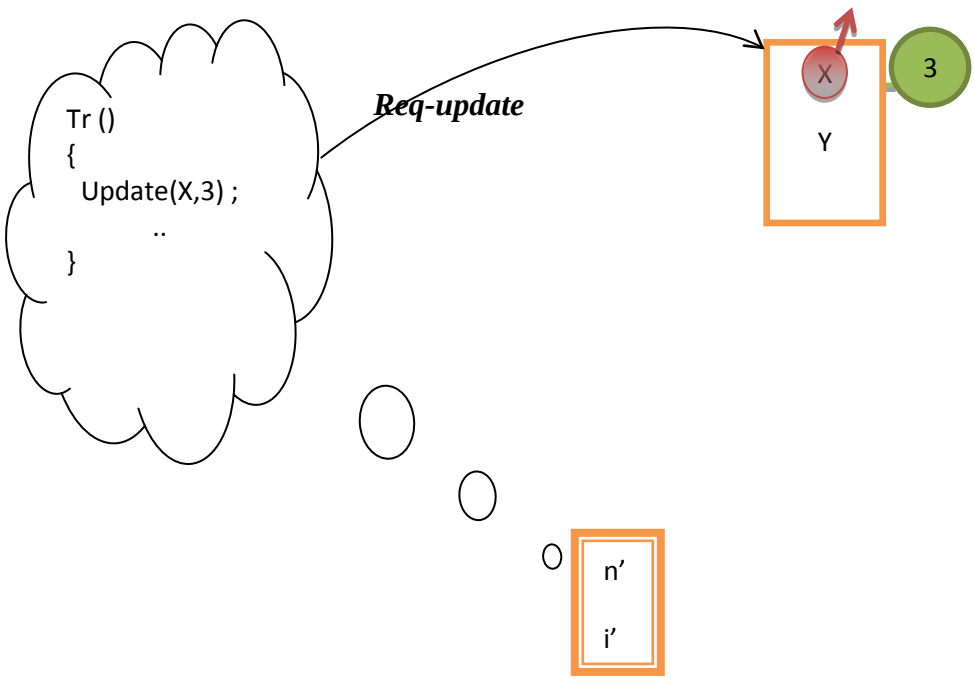


Figure 0-5 : *Opération de modification*

III.5.Journalisation

III.5.1.Journal Undo/Redo en mémoire centrale

Le journal Undo/Redo est un journal qui contient des traces des transactions actives ; données et opérations (Ajout, Suppression, mise-à-jour), chaque case contient sa partie du journal, comme illustré dans la figure (Figure III-11 : exemple de fusionnement d'un journal Undo), chacune des cases contient une table des transactions actives où chaque entrée renferme le journal Undo/Redo d'une transaction. Ce journal est utilisé pour défaire les effets d'une transaction qui va s'annuler, pour retrouver l'ancien état du fichier avant le lancement de cette transaction, ou bien on l'utilise pour recopier ces effets dans le journal global Redo sur disque si celle-là est validée. Et dans les deux cas, l'entrée de cette transaction dans la table des transactions actives, doit être supprimée de la case.

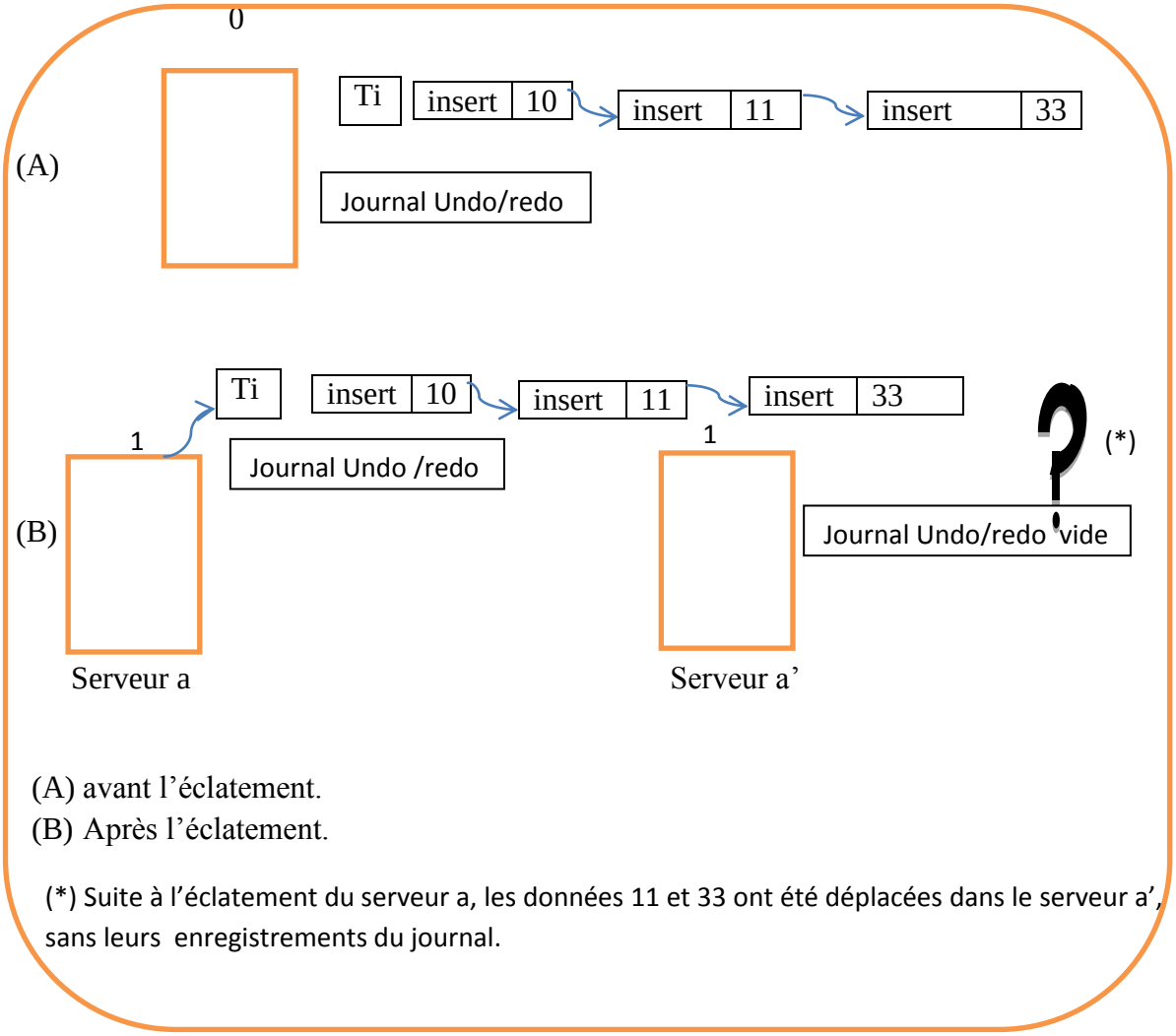


Figure 0-9 : représentation de journal Undo

Ce journal se compose des champs comme suit :

Cod_Op	Anc_Don	Nouv_Don	Clé

Tableau 0-1 : représentation d'une structure d'un journal Undo

Tel que :

Cod_Op : le code d'opération qui va exécuter, par exemple 1 pour une opération d'insertion, 2 pour une opération de suppression et 3 pour une modification.

Anc_Don : pour sauvegarder une donnée qui a existé avant sa suppression, ou modification, et rester vide si une opération d'insertion, c'est-à-dire n'a pas d'ancienne

donnée. On va utiliser ce champ pour récupérer l'ancienne donnée si la transaction va être annulée (défaire).

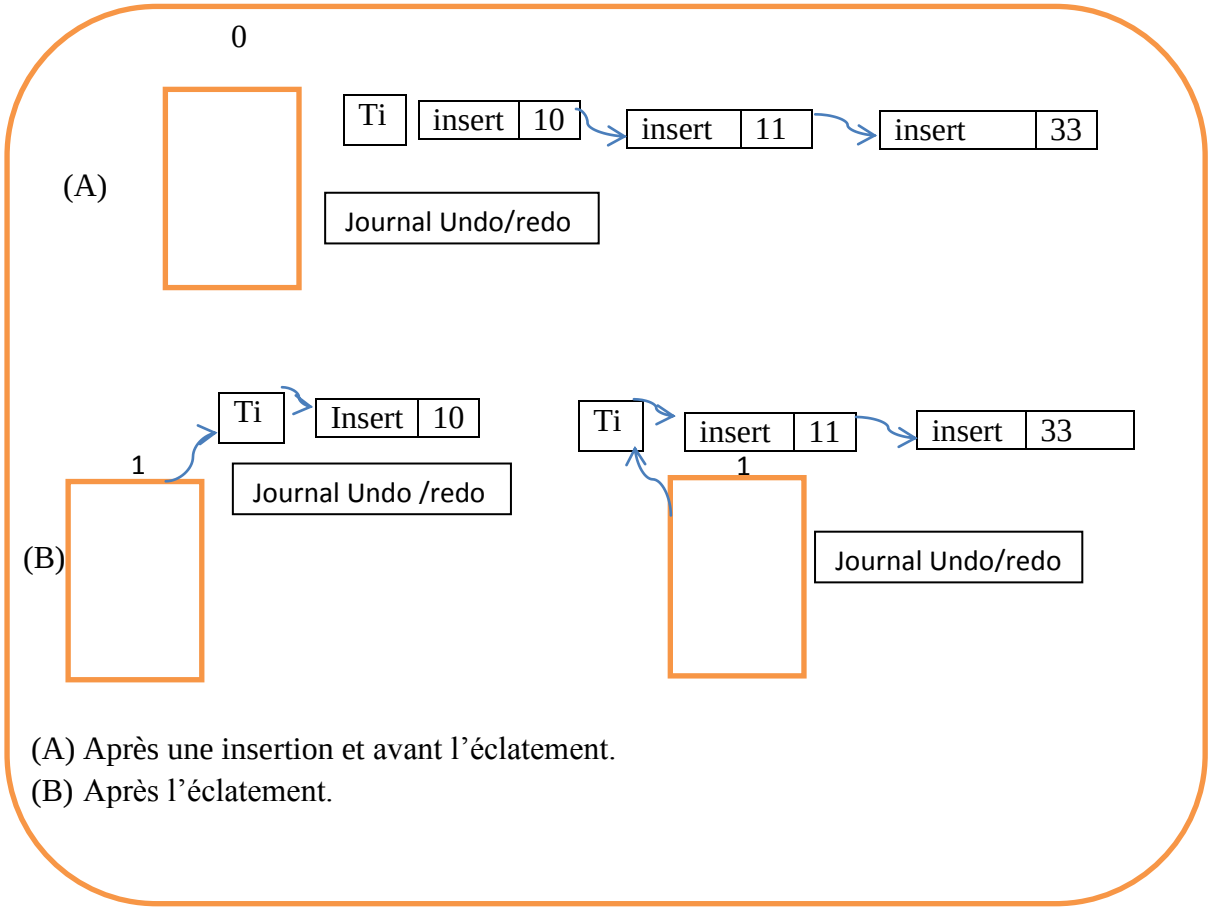


Figure 0-10 : représentation de la méthode de transfert du journal

Nouv_Don : pour sauvegarder la nouvelle donnée qui va exister après un ajout, ou modification, et rester vide si une opération de suppression, c'est-à-dire n'a pas de nouvelle donnée. On va utiliser ce champ pour recopier cette donnée dans le journal Redo sur disque si la transaction est validée, et après le copiage sous disque, on l'utilise en cas de restauration après une panne prévisible.

Clé : c'est la clé de la donnée qui va être modifiée, ajoutée ou bien supprimée.

L'utilisation du journal Undo, pour annuler une transaction comme nous avons dit, cette annulation va défaire les effets de cette transaction de la fin vers le début, mais en inversant les opérations, (ajout → suppression, suppression →ajout et modification d'une ancienne donnée par une nouvelle → modification d'une nouvelle donnée par une ancienne), et ainsi de

suite jusqu'à la fin du journal de cette transaction. Ce travail sera refait sur chaque serveur (case) où cette transaction existe.

Nous avons vu dans la section III.3 comment faire l'éclatement et la fusion, comme nous avons vu la gestion de la panne (reprise après panne). Le problème qui peut se poser est que les données modifiées par une transaction T dans une case, peuvent avoir été déplacées à cause des opérations d'éclatements ou de fusions, après leurs modifications. De ce fait, les enregistrements du journal concernant ces données ne sont plus forcément stockés dans les mêmes serveurs que leurs données correspondantes. Lors de l'exécution des opérations de validations ou d'annulations, un serveur a besoin d'accéder aux enregistrements du journal de la transaction concernée, mais risque de ne pas les trouver en conséquence. Voir la figure (Figure 0-9 : représentation de journal Undo).

- PROPOSITION D'UNE SOLUTION :

Pour produire une solution pour régler la problématique qu'on vient de poser, on va discuter la solution avec tous les cas possibles ; selon les différentes opérations sur une case quelconque, et au début, parlons sur le cas de l'éclatement, puis de la fusion, et enfin le cas de l'annulation d'une transaction, parce qu'elle peut faire des éclatements ou des fusions.

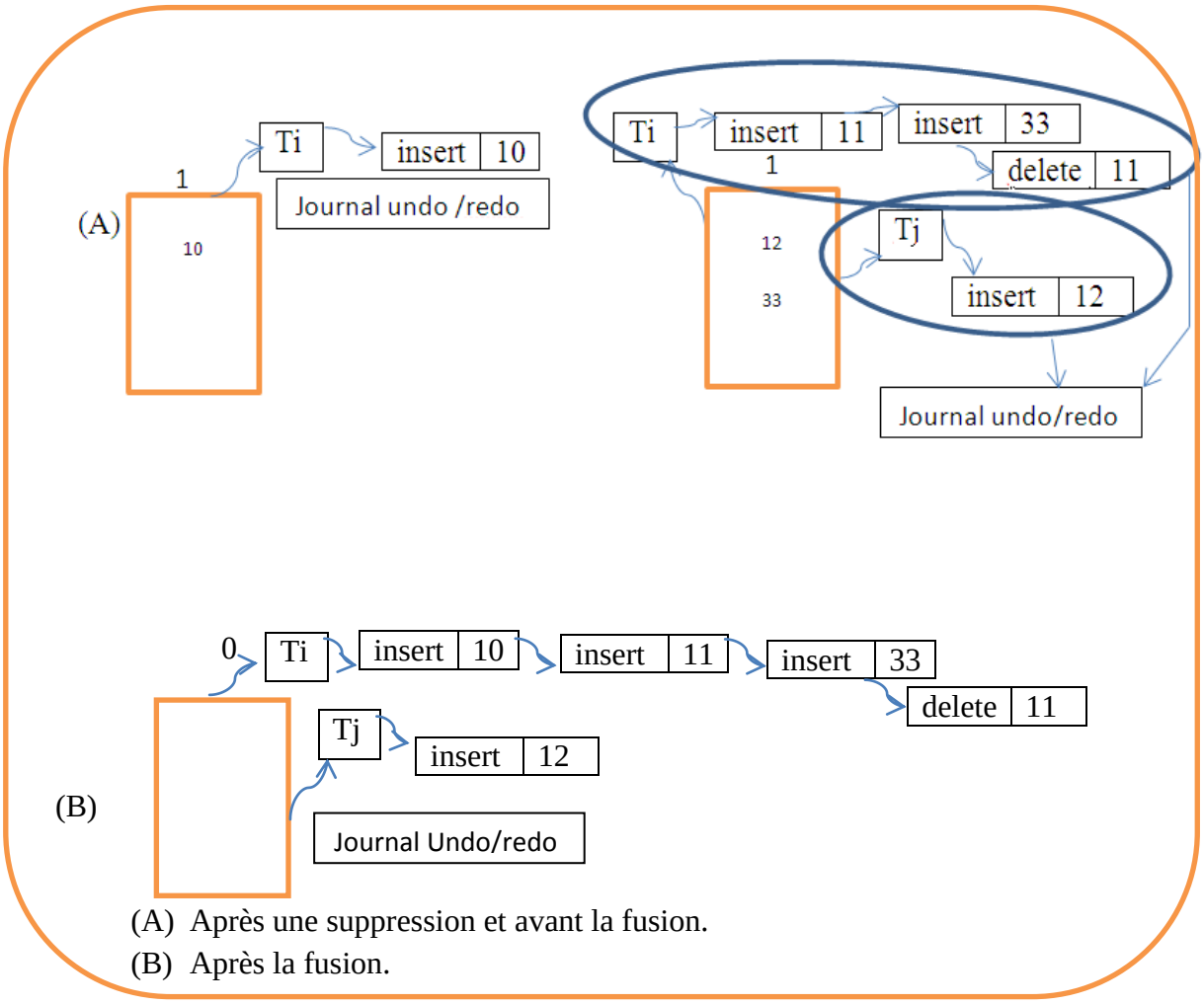


Figure 0-6 : exemple de fusionnement d'un journal Undo

III.5.1.1.**Eclatements**

Notre proposition dans ce cas est de transférer les parties du journal Undo/Redo, qui concernent les données déplacées vers la nouvelle case voir (Figure 0-10 : représentation de la méthode de transfert du journal). On rappelle que lors d'un éclatement de la case n, toutes les opérations liées à cette case restent bloquées temporairement jusqu'à la fin de l'éclatement, et que la case n ne participe pas à la validation d'une transaction durant cet éclatement. Donc il peut exister plusieurs transactions actives dans ce serveur qui sont à l'état temporairement bloqué. Pour transférer les enregistrements Undo/Redo de ces transactions vers la nouvelle case, trois cas possibles peuvent avoir lieu pour chaque transaction active T :

- Une partie des données manipulées par T a été déplacée par l'éclatement. Dans ce cas une partie du journal Undo/Redo de T (celle concernant les données

déplacées) sera envoyée vers le nouveau serveur. T devient donc active dans les deux serveurs (cases) en même temps.

- ❖ Aucune donnée manipulée par T n'a été déplacée par l'éclatement. Dans ce cas, la table des transactions actives du nouveau serveur, ne contiendra pas le journal de la transaction T.
- ❖ Toutes les données manipulées par T ont été déplacées par l'éclatement. Dans ce cas c'est tout le journal Undo/Redo de T qui sera déplacé vers le nouveau serveur. L'entrée de T dans La table des transactions actives de l'ancien serveur sera supprimée.

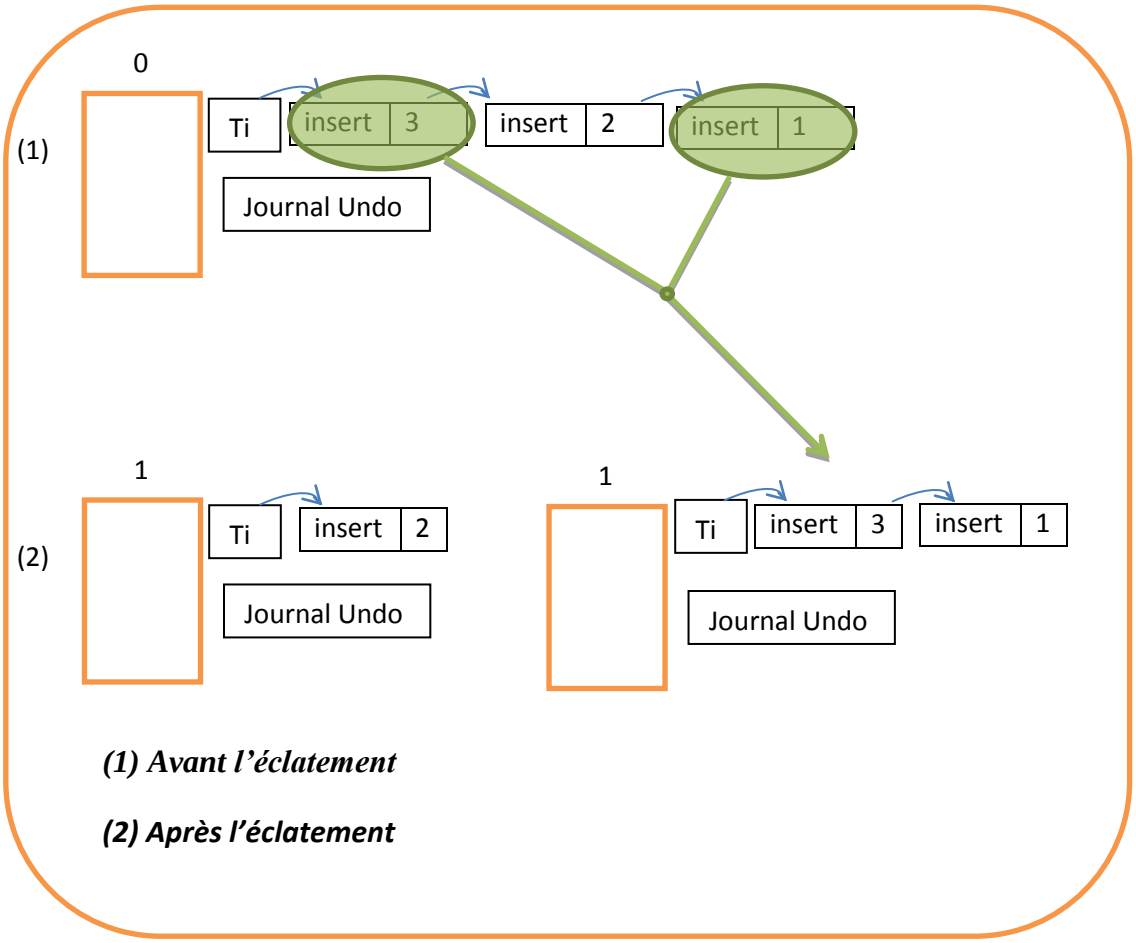


Figure 0-12 : journal Undo pendant une opération d'éclatement.

III.5.1.2.**Fusions**

Symétriquement au cas de l'éclatement, on traitera la fusion en transférant tous les journaux Undo/Redo depuis la dernière case du fichier (serveur a') vers la dernière case ayant

subi un éclatement (serveur a, avec a=n-1 ou bien $a=2^i$ -1si n=0). Lors de ce transfert les opérations reçu par ces deux serveurs restent bloquées jusqu'à la fin de la fusion. Toutes les entrées de la tables des transactions actives du dernier serveur seront déplacées vers le serveur a.

Si une transaction T était active dans a et a' en même temps, son entrée dans la table des transactions actives de a' sera concaténée avec l'entrée de T dans la table des transactions actives de a.

A la fin de l'opération de transfert, les opérations qui étaient bloquées dans a' seront redirigées vers a. le serveur a' restera actif pendant un certain temps pour rediriger les clients qui tentent d'accéder à ses données vers le serveur a. Ceci permettra aux clients de corriger leurs images. Après un certain délai le serveur a' sera complètement désactivé, et s'il existe encore un client voulant accéder au serveur a' sa requête restera sans réponse, et après un délai il retransmet sa requête au serveur 0 qui finira par la rediriger vers le bon serveur a.

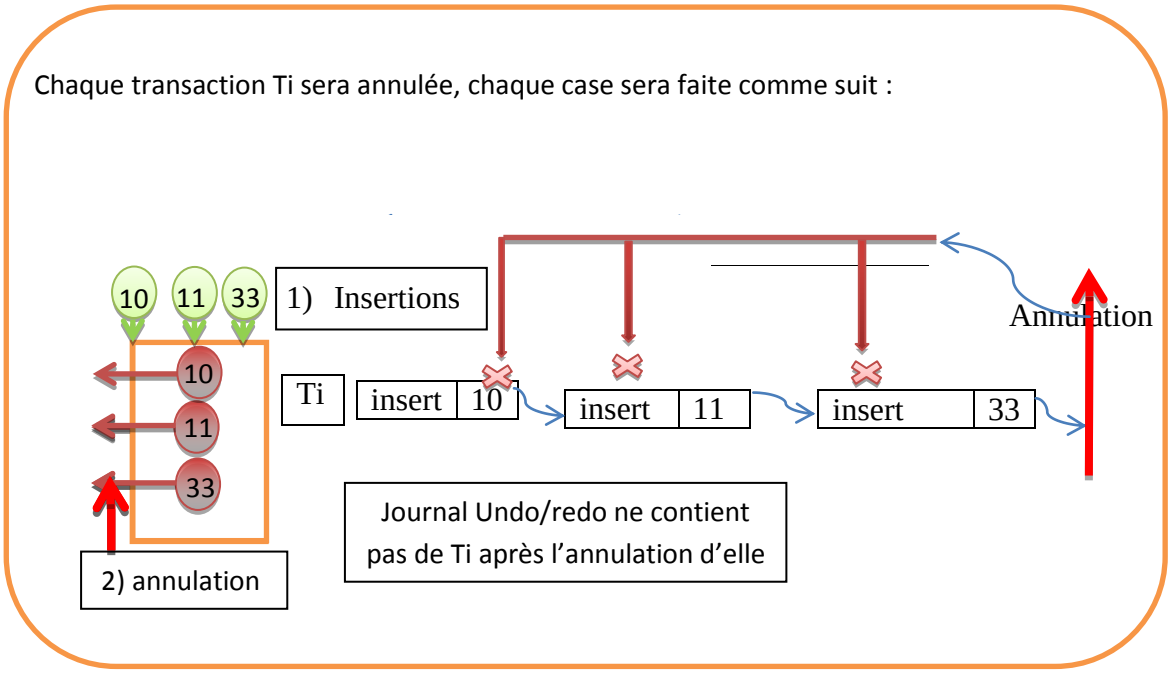


Figure 0-7 : *Annulation d'une transaction.*

Pour plus de clarté voir (Figure 0-6 : exemple de fusionnement d'un journal Undo), où nous allons voir comment se fait la fusion du journal Undo/Redo (l'état (A) vers l'état (B)).

III.5.1.3.**Annulations**

L'annulation d'une transaction concerne seulement celles qui sont actives. En cas d'annulation d'une transaction, on va la défaire, c'est-à-dire on exécute le journal Undo/Redo (la partie qui est construite en effet de la transaction concernée) de la fin vers le début, exactement comme une transaction qui s'exécute normalement. Pour défaire une mise à jour on affecte l'ancienne valeur (sauvegardée dans le journal Undo). Pour défaire une insertion on réalise une suppression, et pour défaire une suppression on réalise une insertion. Ces deux opérations peuvent provoquer des fusions ou des éclatements.

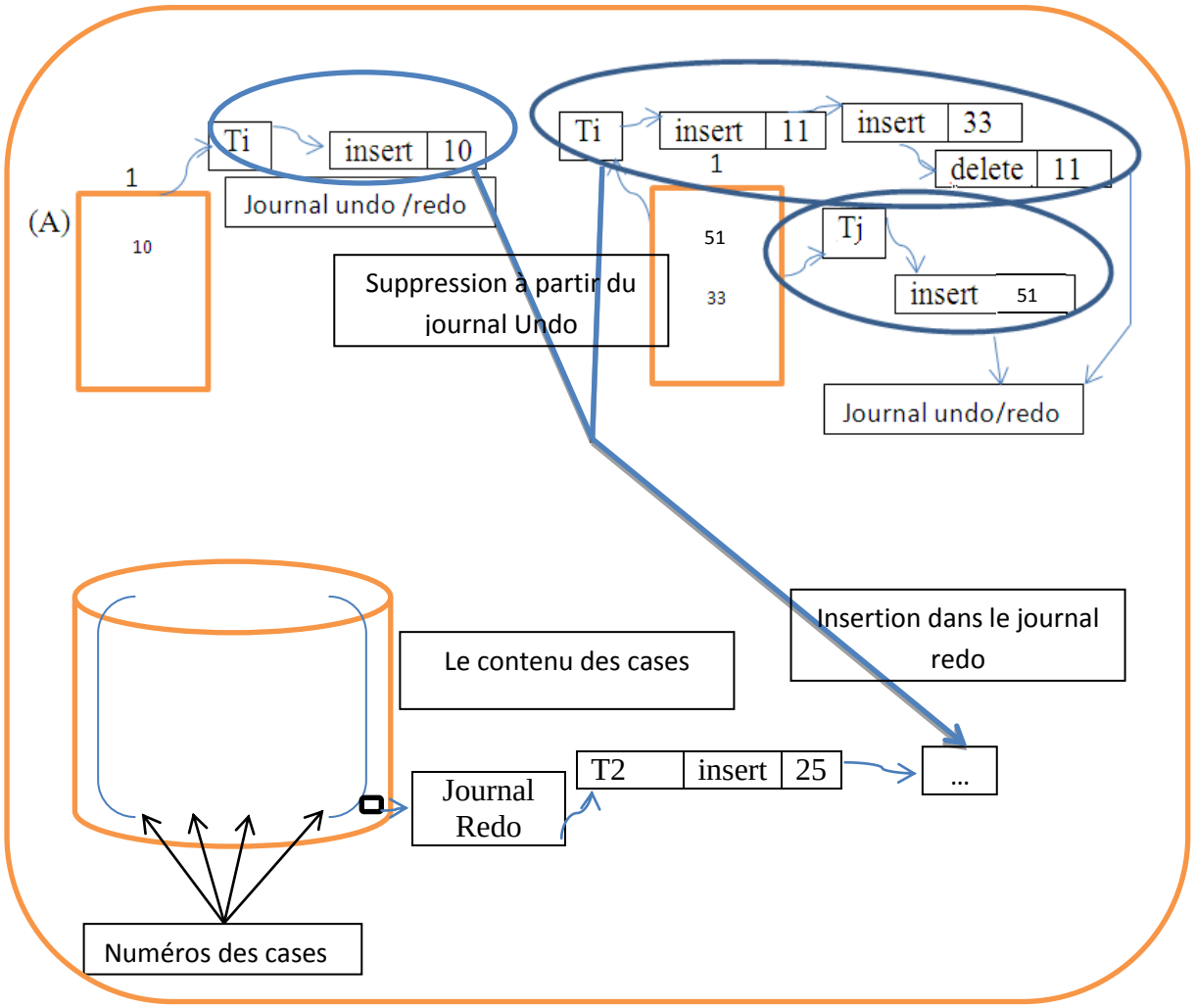


Figure 0-8 : journalisation de validation d'une transaction

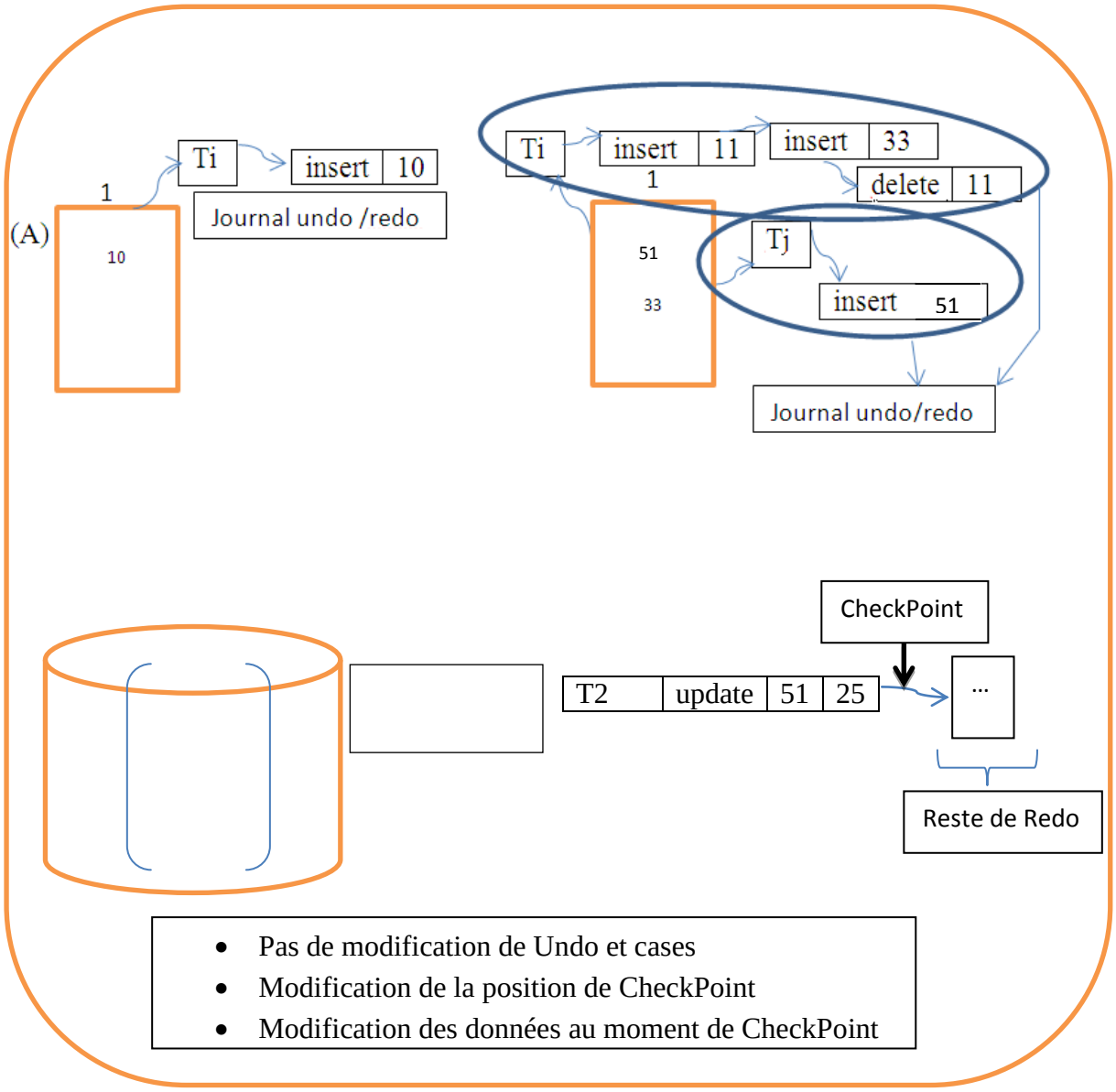


Figure 0-9 : *Point de contrôle (CheckPoint).*

III.5.2.Journal Redo

Le journal Redo est un journal qui contient les traces des transactions qui sont validées, on utilise ce journal pour récupérer des données d'un serveur (d'une case) qui vient de tomber en panne, ce journal se compose des champs suivants.

Cod_Tran	Cod_Op	Clé	Nouv_Don

Tableau 0-2 : Représentation d'une structure d'un journal Redo.

Tel que :

Cod_Tran : le code de la transaction, qui fait l'opération.

Cod_Op : Code d'Opération, par exemple 1 pour une opération d'insertion, 2 pour une opération de suppression et 3 pour une opération de modification.

Clé : c'est la clé de la donnée qui sera modifiée, ajoutée ou bien supprimée.

Nouv_Don : pour sauvegarder la nouvelle donnée qui va exister après un jout, ou une modification. Dans le cas d'une opération de suppression, ce champ restera vide.

Figure 0-10 : Journal Undo

N.B : l'opération de la recherche n'est pas nécessitée à journalisée parce qu'elle ne fait pas des modifications sur le fichier.

On plus de ces enregistrements, le journal Redo peut aussi contenir des enregistrements spéciaux, tels que : marque de CheckPoint, valeur du niveau de hachage j, Début éclatement, Début de fusion et Fin éclatement.

A chaque changement du niveau j d'une case (éclatement ou fusion), la nouvelle valeur est sauvegardée dans le journal Redo. Ainsi lors de la reprise après une panne, un serveur peut retrouver son dernier niveau.

Le journal Redo sera utile dans Deux situations : la validation d'une transaction (Figure III-4 : Scénario de validation d'une transaction.), et la gestion des points de contrôles (CheckPoints) (Figure III-15 : Point de contrôle (CheckPoint).). Nous allons parler de ces deux points.

III.5.2.1.**Validation d'une transaction**

Une transaction peut s'activer dans plusieurs cases comme représenté dans la figure (Figure III-9 : représentation de journal Undo). Pour valider une transaction T, on a adapté le protocole de validation à deux phases comme suit :

- Durant la phase de vote, les participants envoient les journaux Redo de la transaction T vers le coordinateur et se mettent en attente de la décision finale.
- Si le coordinateur reçoit les journaux Redo de tous les participants de T, il les sauvegarde dans son disque avant d'informer les participants de la décision de validation. S'il ne reçoit pas le Redo d'au moins un participant, il informe les autres de la décision d'annulation.
- Durant la deuxième phase, les participants appliquent la décision reçue du coordinateur. S'il s'agit d'une validation, ils effacent l'entrée de la transaction T dans la table des transactions actives et libèrent les verrous. S'il s'agit d'une annulation, ils utilisent leurs journaux Undo pour défaire les effets de la transaction T, et ils effacent son entrée dans la table des transactions actives et libèrent les verrous. Voir (Figure III-4 : Scénario de validation d'une transaction.).

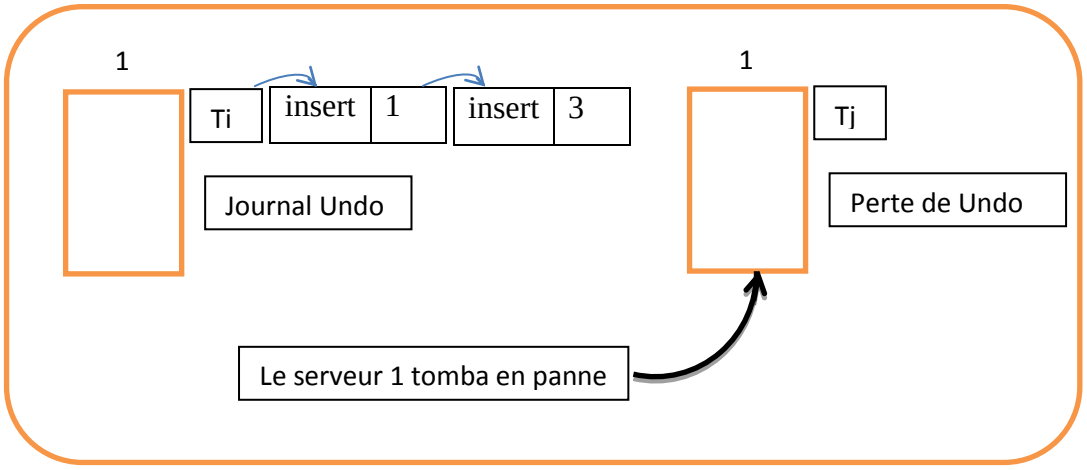


Figure 0-11 : *panne d'un serveur.*

N.B : Si une transaction a été validée on ne peut pas la réutiliser.

III.5.2.2.**Point de contrôle (CheckPoint)**

Un point de contrôle consiste à écrire sur disque certaines informations pour éviter le parcours complet du journal lors de la reprise. Il permet aussi de purger le journal, en éliminant les anciens enregistrements. Dans notre proposition le point de contrôle doit-être fait sur toutes les cases (sauvegarde de toutes les données), lorsqu'il n'y a pas de transaction active. Voir (Figure III-15 : Point de contrôle (CheckPoint).).

Périodiquement, lorsque certaines conditions sont vérifiées, le coordinateur sauvegarde le contenu des cases des serveurs dans son propre disque, puis réinitialise le journal Redo à vide. Pour obtenir un CheckPoint consistant, le coordinateur doit attendre qu'il n'existe aucune transaction active, avant de commencer l'opération de sauvegarde. L'administrateur peut aussi provoquer un CheckPoint manuellement.

III.6.Reprise après panne et recouvrement

Les systèmes des bases de données comme d'autres systèmes, peuvent tomber en panne (Figure III-17 : panne d'un serveur.), nous nous intéressons aux pannes systèmes provoquant la perte de la mémoire centrale ainsi que les pannes de transactions. La reprise est une procédure que le système doit exécuter après toute occurrence de panne système pour retrouver le dernier état cohérent de la base de données avant la panne.

Toutes les modifications faîtes par les transactions actives durant la panne, ne doivent pas figurées dans l'état de la base après la reprise. Ceci est assuré par le faîte que les données sont maintenues en mémoire centrale, donc à chaque panne les dernières modifications sont toujours perdues.

Toutes les modifications faîtes par les transactions validées avant la panne, doivent persister. Pour cela la procédure de la reprise consiste à recharger en mémoire centrale le contenu de chaque serveur qui était en panne à partir des copies du dernier CheckPoint sur le disque du coordinateur. Ensuite, elle refait l'exécution des opérations du journal Redo global, afin d'atteindre le dernier état validé.

Il est à remarquer que la procédure de la reprise est idempotente, c'est-à-dire qu'en cas de panne durant le déroulement de la procédure de reprise, le contenu du journal et des copies de case des CheckPoints reste toujours utilisable pour refaire d'autres reprises consécutives.

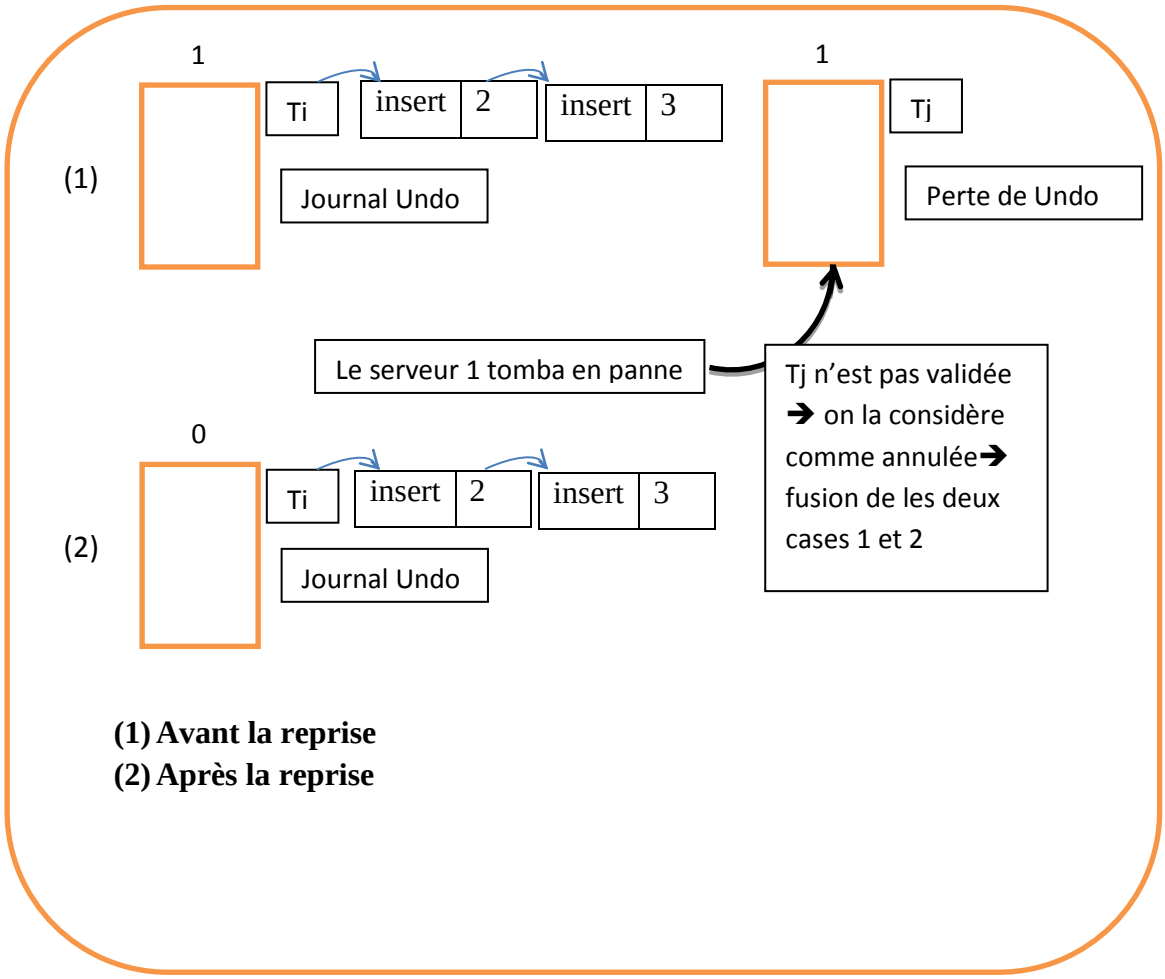


Figure 0-12 : *Reprise après panne et fusionnement de deux cases.*

Plus en détail, lors de la reprise d'un serveur les actions suivantes entreprises :

- Relancé le serveur.
- Chargé sa case à partir de la copie du CheckPoint (si elle existe).
- Parcourir le journal Redo pour extraire et ré-exécuter toutes les opérations destinées à ce serveur (en recalculant l'adresse de chaque clé par la bonne fonction de hachage).

Remarque : le parcours d'une partie du journal Redo coûte cher, car tous les enregistrements concernant tous les serveurs seront examinés. On propose alors une autre méthode de recouvrement qui nous permet de recouvrer les données à moindre coût.

III.7.Deuxième solution pour l'éclatement et la fusion

La solution proposée consiste à sauvegarder le journal Redo dans chaque serveur, chacun stockant uniquement les enregistrements qui le concernent.

Lors d'un éclatement de la case n, celle-ci doit-être bloquée durant le déroulement de cette opération, ses données seront re-hachées pour savoir si elles sont concernées par l'éclatement. Une fois les données concernées seront transférées vers la nouvelle case, on réalise un CheckPoint pour cette dernière. Le dernier CheckPoint et les enregistrements Redo de la case n restent inchangés voir (Figure III-19 : Redo après un éclatement).

Si le serveur ayant subi l'éclatement tombe en panne, on pourra le reconstruire en chargeant le dernier CheckPoint, puis en parcourant son journal REDO à partir de ce CheckPoint. Lors de ces deux opérations, les données manipulés doivent être re-hachée pour vérifier qu'elles concernent bien le serveur en cours de reprise. En effet il peut encore exister dans le CheckPoint et dans le journal REDO, des données qui ont été transférées vers un nouveau serveur lors d'un éclatement et donc ne seront pas concernées par l'opération de reprise de ce serveur.

Si le nouveau serveur issu de l'éclatement tombe en panne, on trouvera dans son CheckPoint et son journal REDO toutes les données validées avant la panne.

Avant que le serveur n commence à faire l'éclatement, il inscrit dans son journal Redo un enregistrement spécial 'Début éclatement'. Quand l'opération se termine il inscrit dans le journal l'enregistrement spécial 'fin éclatement'. Le nouveau serveur issu de l'éclatement, inscrit dans son journal Redo l'enregistrement spécial 'Début éclatement' . La

présence d'un CheckPoint après l'enregistrement 'Début éclatement' indique alors que l'opération d'éclatement a été complètement effectuée au niveau de nouveau serveur.

Si une panne se produit durant l'opération d'éclatement (au niveau du serveur n et/ou du nouveau serveur) les enregistrements 'Début éclatement', 'fin éclatement', CheckPoint, permettent de réaliser une reprise correcte :

- Si la panne concerne le serveur n, lors de sa reprise, il trouve l'enregistrement 'Début éclatement', mais il ne trouve pas l'enregistrement 'Fin éclatement', il effectue alors une reprise à partir de dernier CheckPoint, et du journal Redo, et refait ensuite l'opération d'éclatement.

- Si la panne concerne le nouveau serveur, lors de sa reprise, il trouve l'enregistrement 'Début éclatement', mais il ne trouve pas le CheckPoint indiquant la fin de l'éclatement, donc il informe le serveur n pour que ce dernier refasse l'opération.

Lors d'une opération de fusion de cases (la dernière case avec la case n-1), toutes les données de la dernière case sont transférées vers la case n-1, puis un CheckPoint est effectué sur cette dernière. Ainsi en cas de panne de ce serveur, on est sûr de retrouver toute l'information nécessaire en chargeant le dernier CheckPoint et en parcourant le journal REDO associé voir (Figure III-20 : Redo après une fusion).

Avant que le serveur n-1 commence à faire la fusion, il inscrit dans son journal Redo un enregistrement spécial 'Début fusion'. Quand l'opération se termine il fait un CheckPoint. Le dernier serveur (celui concerné par la fusion), inscrit dans son journal Redo l'enregistrement spécial 'Début fusion'.

Si une panne se produit durant l'opération de fusion (au niveau du serveur n-1 et/ou du dernier serveur) les enregistrements 'Début fusion et CheckPoint, permettent de réaliser une reprise correcte :

- Si la panne concerne le serveur n-1, lors de sa reprise, il trouve l'enregistrement 'Début fusion, mais il ne trouve pas l'enregistrement 'CheckPoint', il effectue alors une reprise à partir de dernier CheckPoint, et du journal Redo, et refait ensuite l'opération de fusion.

➢ Si la panne concerne le dernier serveur, lors de sa reprise (après avoir récupéré le dernier CheckPoint et refait le journal Redo), il trouve l'enregistrement 'Début fusion', donc il informe le serveur n-1 et refait l'opération de fusion.

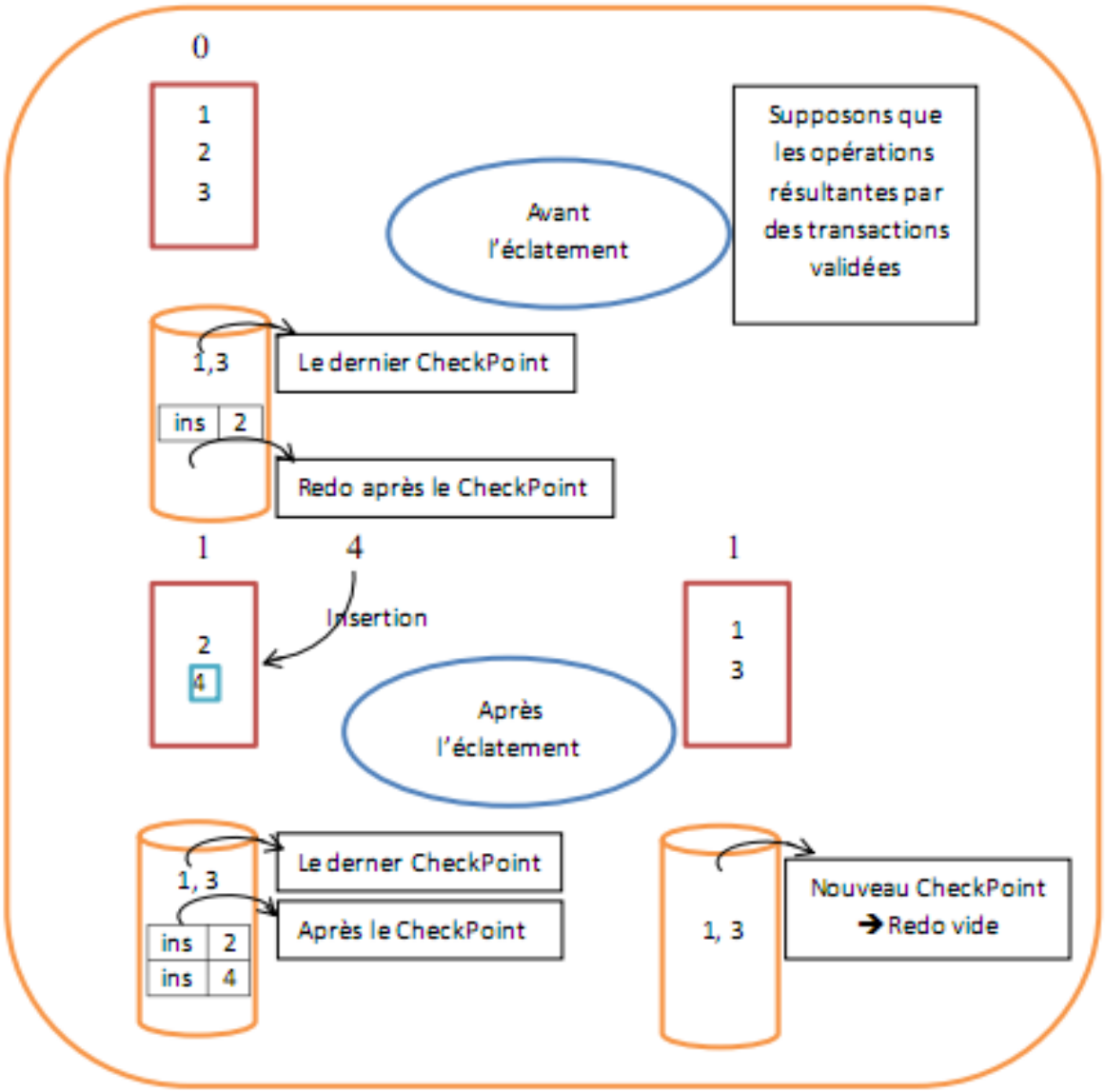


Figure III-19 : *Redo après un éclatement*

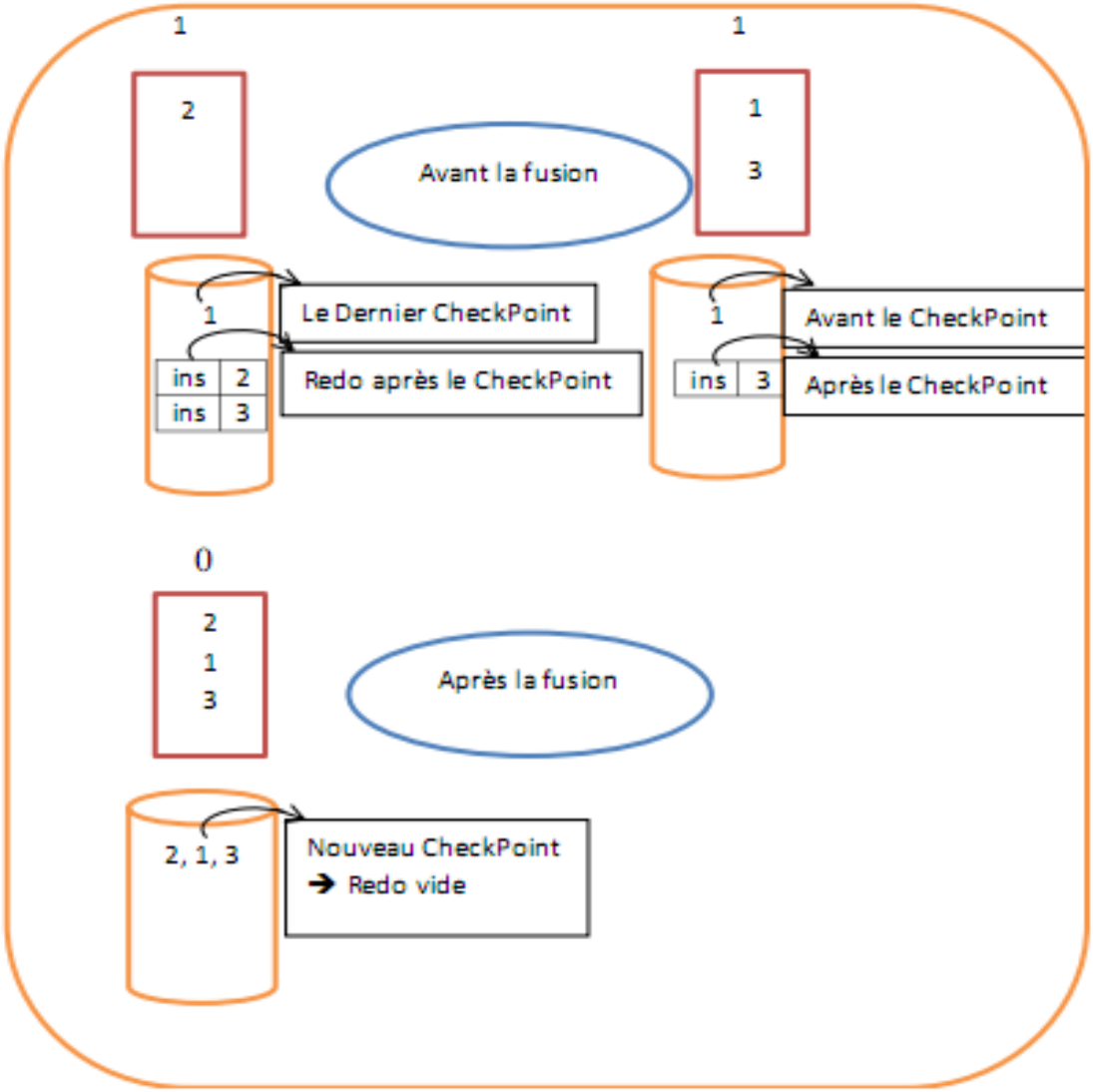


Figure III-20 : *Redo après une fusion*

III.8.Conclusion

Nous avons présenté dans ce chapitre notre méthode de journalisation adaptée à la SDDS LH*, y compris la prise en compte des opérations complexes telles que l'éclatement et la fusion.

Nous avons présenté l'architecture du système de journalisation. Puis nous avons exposé la gestion des journaux Undo/Redo pour rendre possible l'annulation de n'importe quelle transaction active ou préparer sa validation définitive.

Deux solutions assurant la durabilité des transactions validées, ont été détaillées permettant la validation atomique et la reprise après panne.

Chapitre IV

Mise en œuvre

IV.1.Introduction

Nous allons présenter les détails de notre prototype qui nous a permis de mettre en pratique les différentes fonctionnalités proposées dans notre solution du chapitre précédent. A travers des déroulements et des captures d'écrans, nous allons présenter la manière d'utilisation de notre application, et comment se comportent les différentes structures de données au niveau des serveurs face aux opérations d'accès issues des clients.

IV.2.Explication de la manière d'utilisation de l'application

Au début la première image de notre application (Figure IV-1 : choix de type d'exécution), on peut choisir le mode d'exécution ; manuel ou bien automatique. Nous allons utiliser le mode manuel parce qu'il nous donne les mêmes résultats avec le deuxième mode, et en plus, nous pouvons contrôler les différentes taches (ajouts, suppressions, éclatements, verrouillage.. etc.).

Après le choix de la façon manuelle nous obtiendrons la figure (Figure IV-2 : une seule case vide), dans cette figure il y a une seule case (serveur) N=1, n=0, i=0, n`=0 et i`=0. Et des boutons pour : l'ajout, la suppression, la modification et la recherche. Il y a aussi des zones de saisi pour les clés, les données et les nouvelles données en cas de modification, et pour saisir d'autres informations (n° de client, n° de transaction, n° de serveur qui simule une panne). On a aussi d'autres boutons pour contrôler différentes exécutions. Dans la même fenêtre, on a des informations sur le coordinateur qui contient des valeurs réelles de niveau d'éclatement (i) et le pointeur de la case à éclater (n), et aussi sur des clients qui contiennent des valeurs présumées (n` et i` ; chacun d'eux a ses valeurs propres), et enfin les niveaux de hachage (j) pour les différentes cases ou serveurs formant le fichier (chacune d'elles a sa valeur propre).

La représentation des données dans une case est sous forme d'une liste de lignes, chacune d'elles contient un couple de données (x y), le x représente la clé et le y représente le(s) champ(s) non clé voir (Figure IV-3 : ajout des données).

Et après avoir fait des opérations (ajouts, .. etc.), de nouveaux évènements peuvent apparaître, comme les éclatements et fusions (Figure IV-4 : éclatement d'un serveur). Par

exemple, on peut voir un éclatement dans la figure (Figure IV-6 : journal Undo), après l'ajout de la donnée 44 de la clé 4, et ces opérations sont issues de la transaction n°1.

Reprenons le déroulement d'une série d'opérations et voyons comment évoluent les structures de données de la méthode. Cette application dispose de plusieurs fonctions comme nous l'avons dit précédemment, dans ce qui suit, on va voir comment les utilisées. Premièrement l'écran d'accueil permet de choisir la façon de lancer l'exécution ; mode automatique ou bien manuel ; comme on voit dans la figure (Figure IV-1 : choix de type d'exécution). On va choisir le mode manuel.

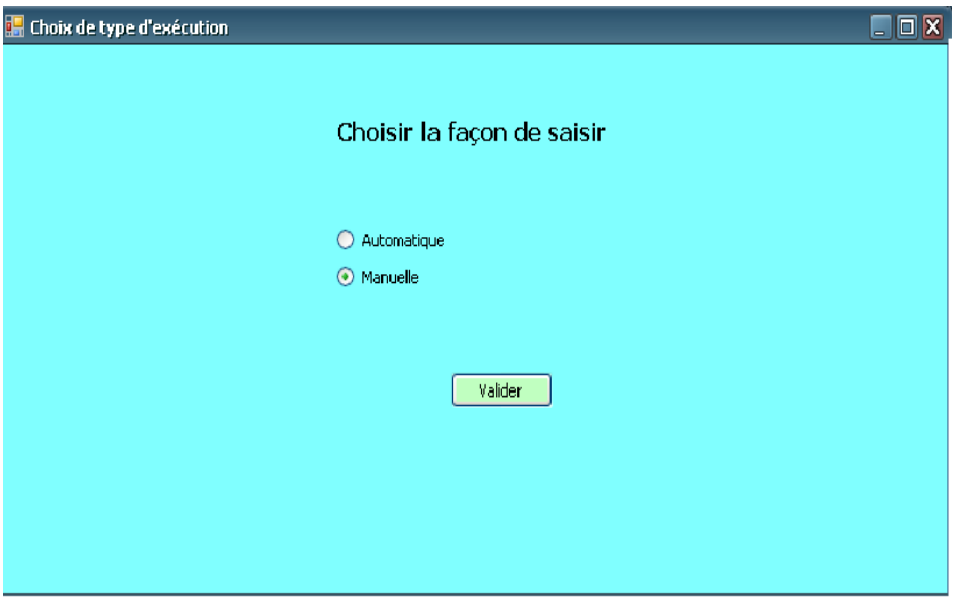


Figure 0-1 : choix de type d'exécution

Après on retrouve une interface pour suivre les différentes valeurs des structures importantes (Figure IV-2 : une seule case vide). Dans cette figure, on voit l'état du système formé par une seule case (un seul serveur), cette case peut contenir de cinq éléments au maximum afin de ne pas encombrer l'affichage.

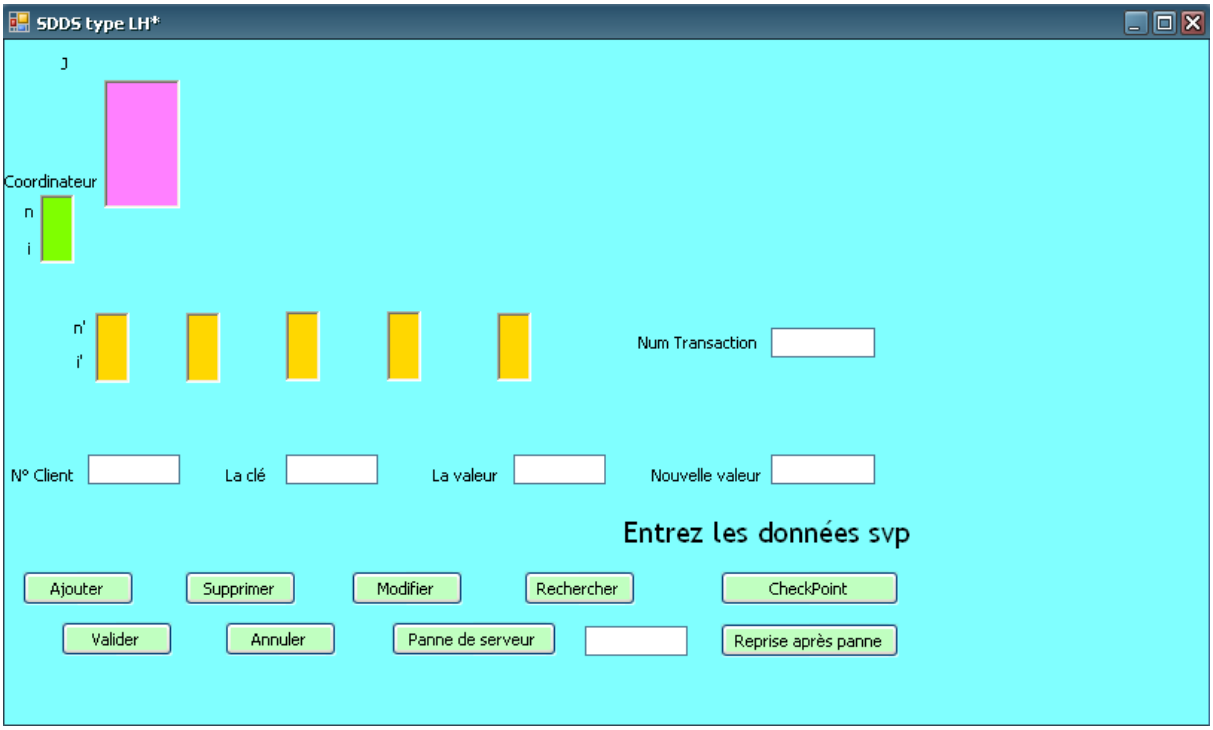


Figure 0-2 : une seule case vide

A l'étape initiale, le pointeur d'éclatement est positionné sur la première case (N=1). On fait une série d'opérations d'ajouts, chacune portant le numéro de la transaction d'appartenance, le numéro du client qui lance cette transaction, la clé de la donnée à ajouter et la donnée elle-même.

Après l'exécution de ces trois opérations d'ajout (clé, donnée) : insert (1,12), insert (2,31), insert (5,51), on va trouver l'état de la figure (Figure IV-3 : ajout des données).

Remarque :

- avant le lancement des transactions, les journaux UNDO et REDO avaient été vidés.
- le journal REDO débute par un CheckPoint ; Maintenant après l'état précédent si on ajoute le couple (4, 44), il y aura un éclatement comme on illustré dans (Figure IV-4 : éclatement d'un serveur).

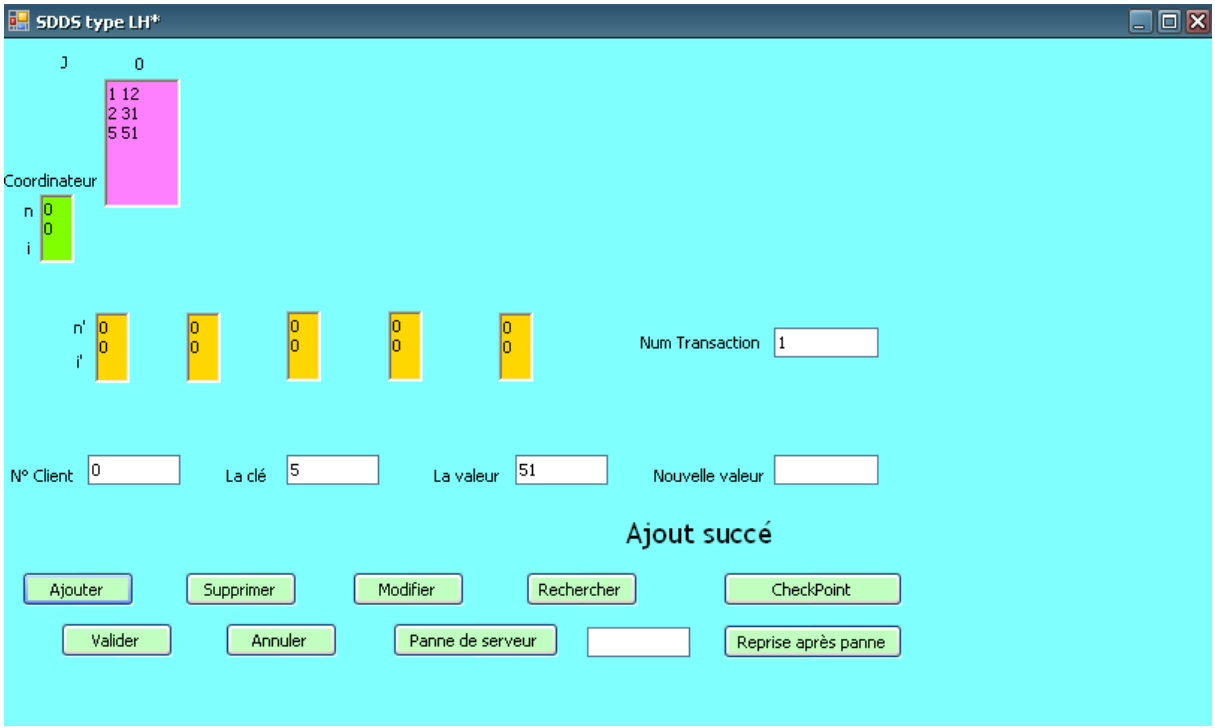


Figure 0-3 : ajout des données

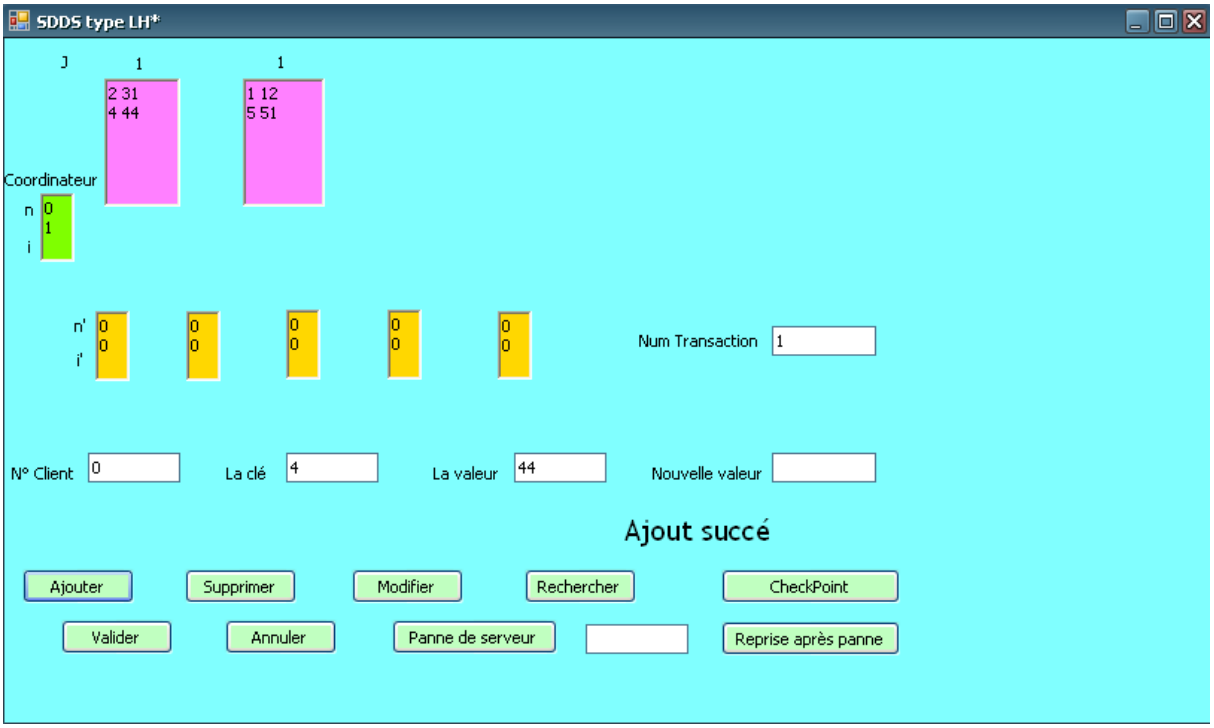


Figure 0-4 : éclatement d'un serveur

Lorsqu'une transaction veut utiliser une donnée modifiée par une autre transaction avant sa validation, le gestionnaire des transactions va interdire cette opération, car la donnée a

déjà été verrouillée par la première voir l'image (Figure IV-5 : donnée verrouillée), tel que la transaction 2 ne peut pas supprimer la clé 4.

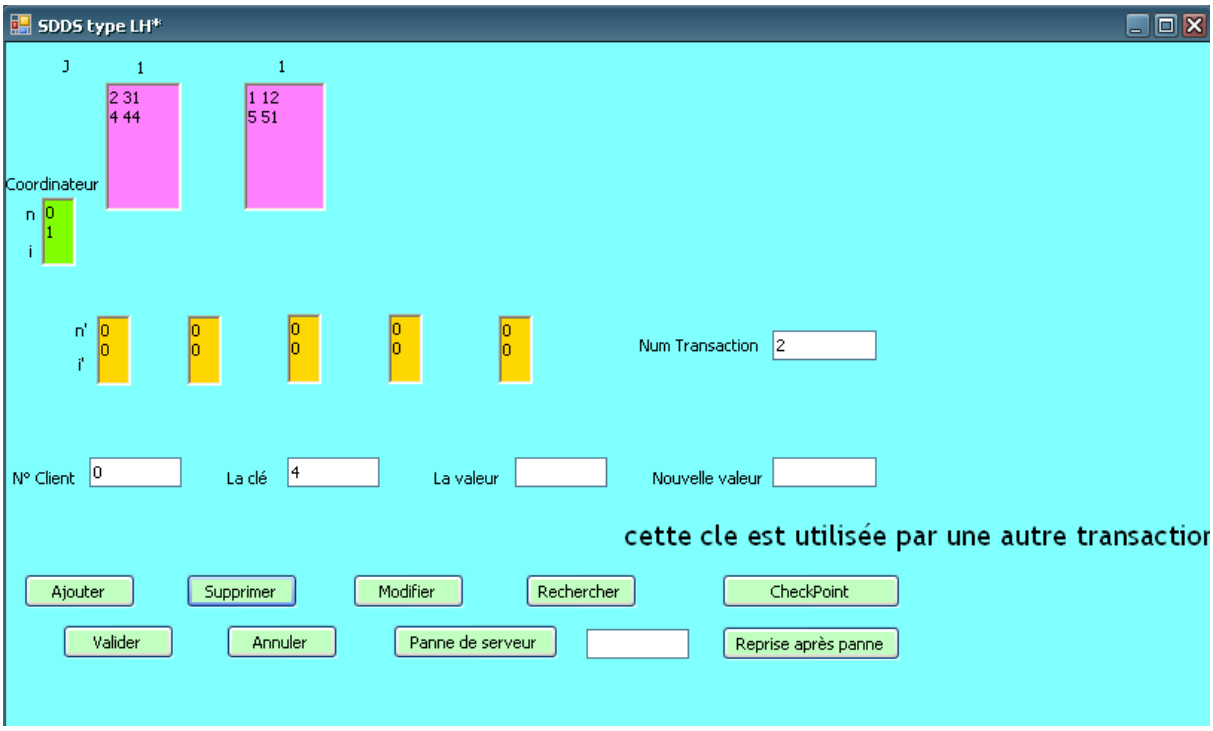


Figure 0-5 : donnée verrouillée

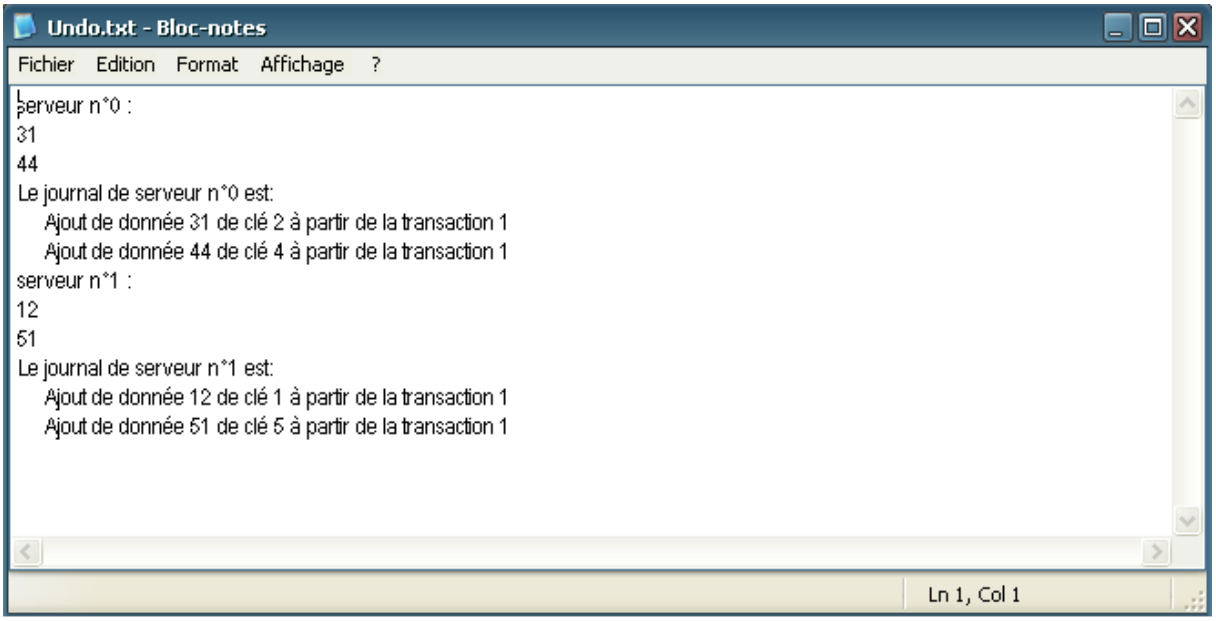


Figure 0-6 : journal UNDO

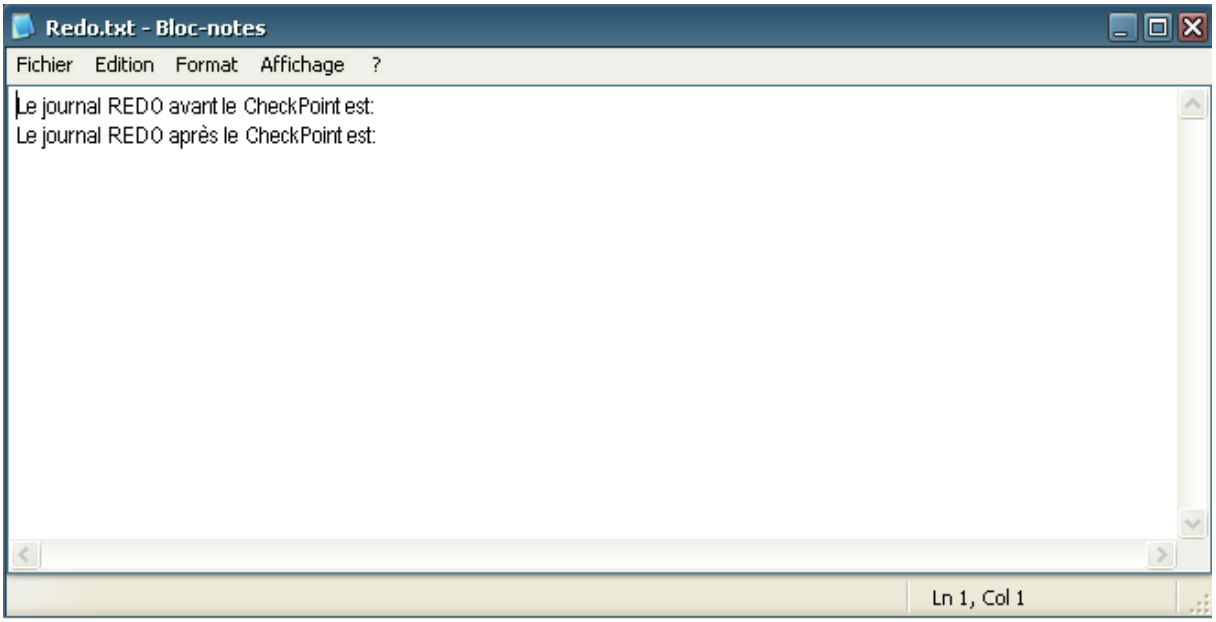


Figure 0-7 : journal REDO

Lorsqu'une transaction veut utiliser une donnée modifiée par une autre transaction avant sa validation, le gestionnaire des transactions va interdire cette opération, car la donnée a déjà été verrouillée par la première voir l'image (Figure IV-5 : donnée verrouillée), tel que la transaction 2 ne peut pas supprimer la clé 4.

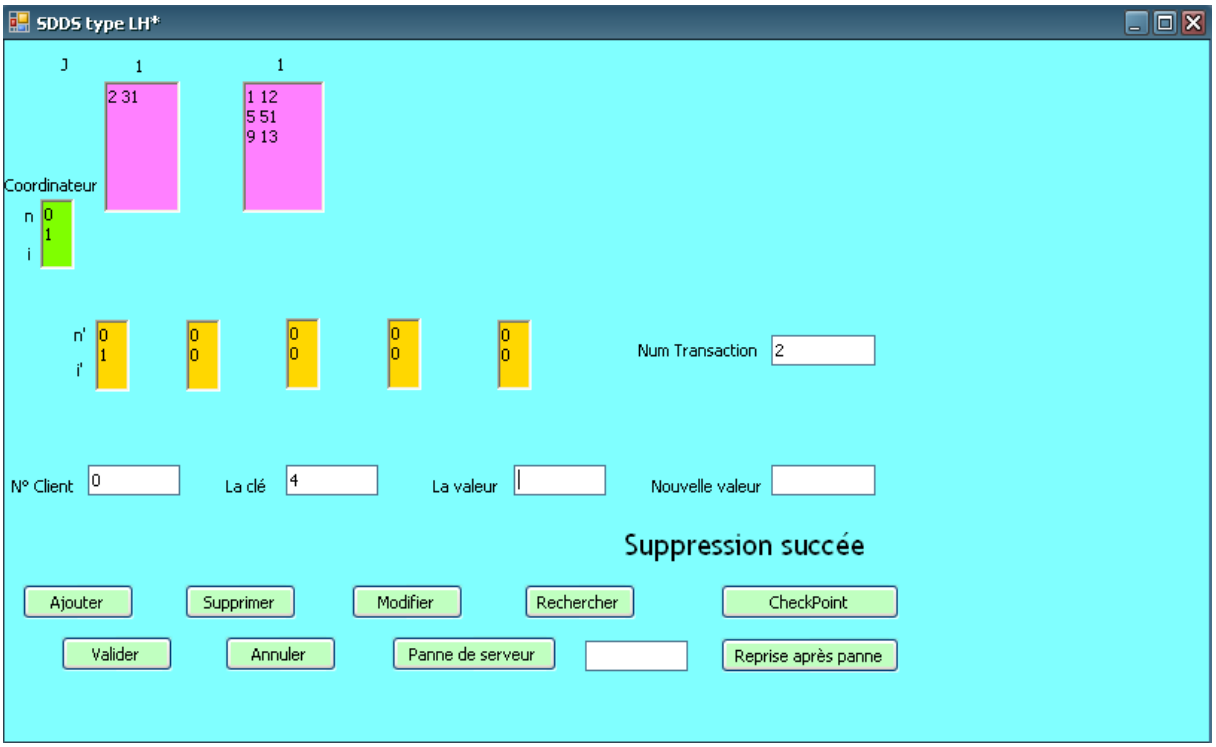


Figure 0-8 : suppression des données

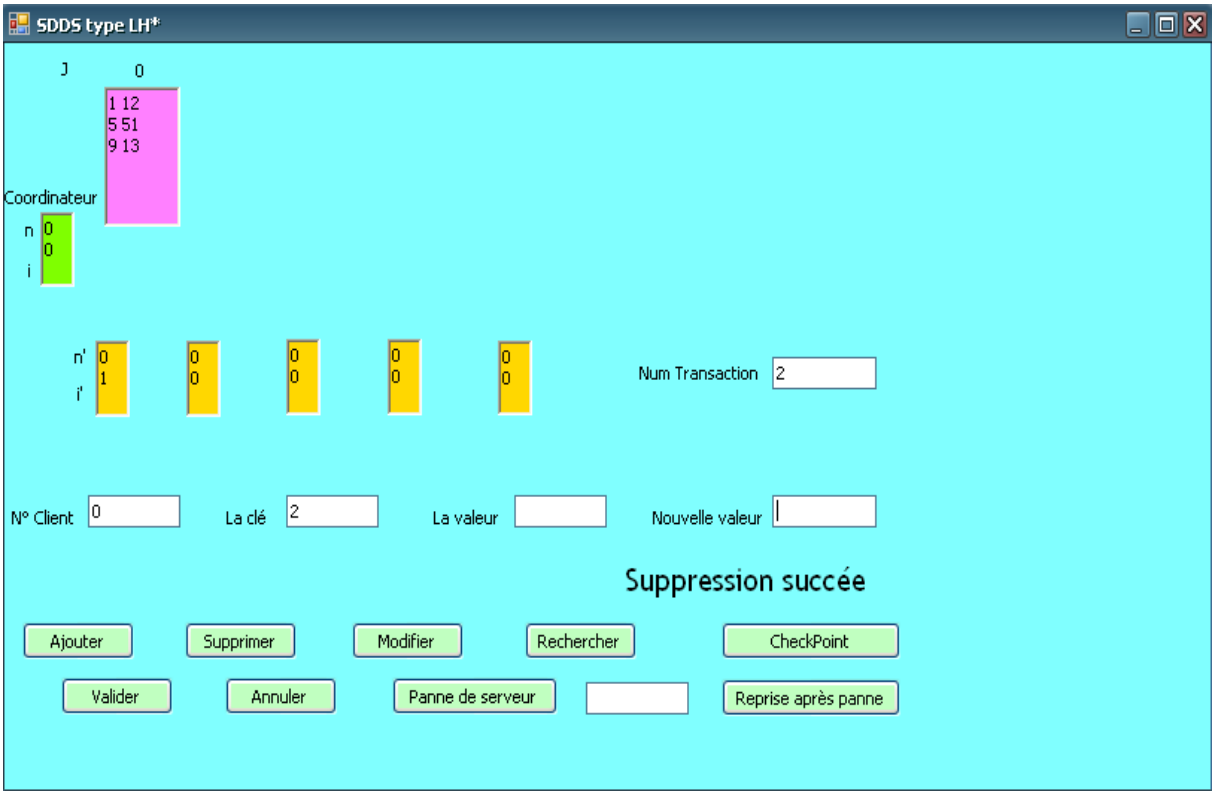


Figure 0-9 : fusion de deux cases

- (Figure IV-4 : éclatement d'un serveur) (Figure IV-9 : fusion de deux cases)

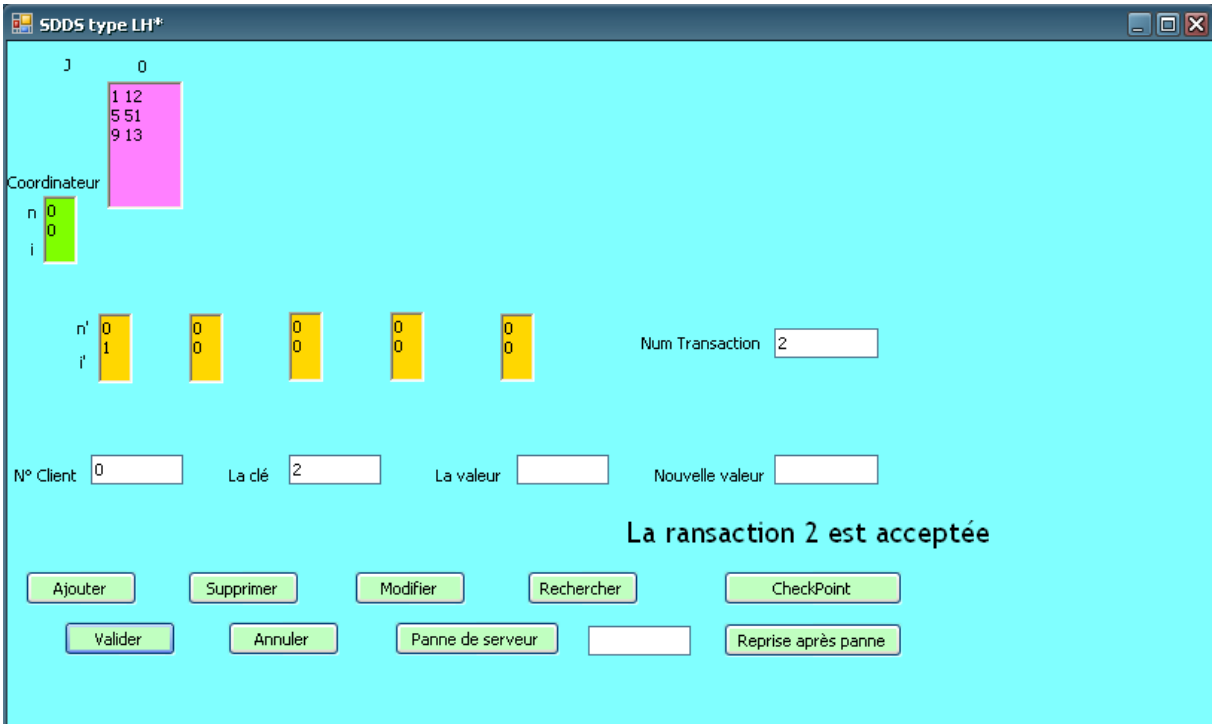


Figure 0-10 : validation d'une transaction

Après la validation de la transaction 2 ; voir (Figure IV-10 : validation d'une transaction), on n'a plus de transaction active, pour cela son journal UNDO doit être vide, voir (Figure IV-11 : journal Undo), et le journal REDO global doit maintenant contenir toutes les modifications des transactions qui sont validées, voir (Figure IV-12 : journal Redo).

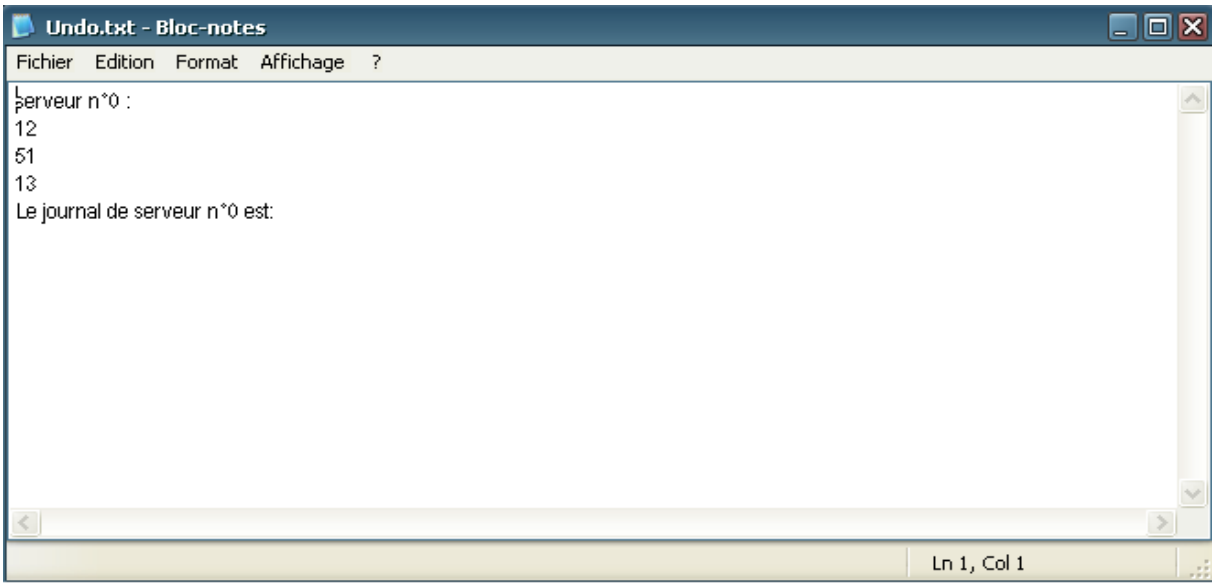


Figure 0-11 : journal UNDO

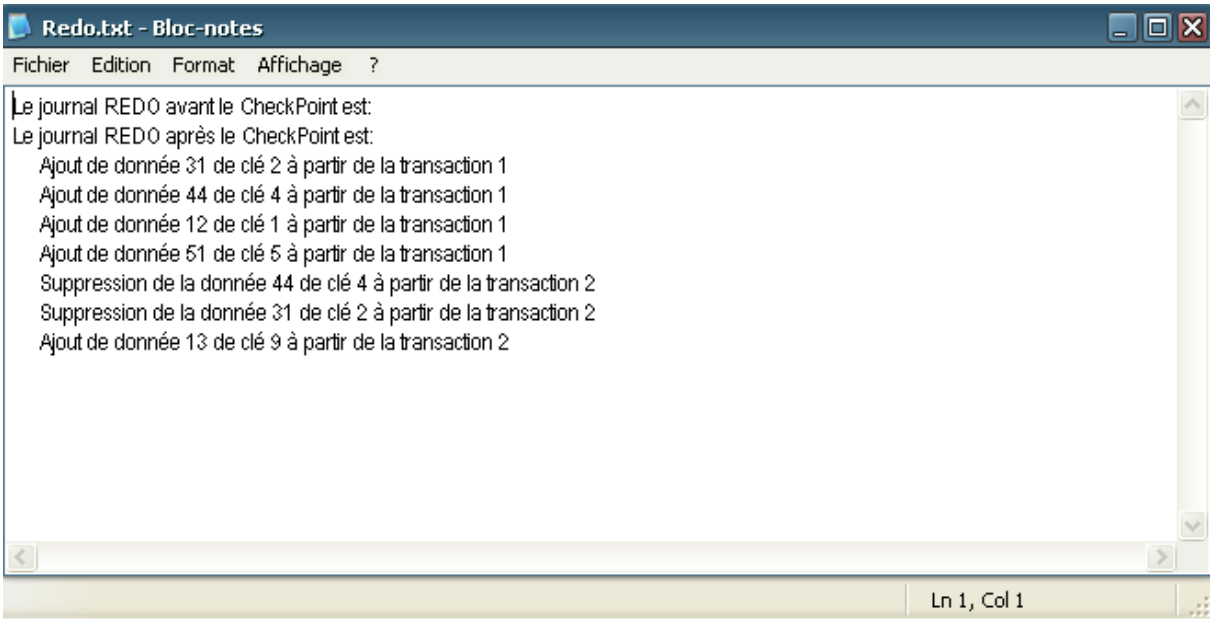


Figure 0-12 : journal REDO

Remarque :

- comme le journal REDO commence toujours par un CheckPoint initial, il ne peut pas y avoir des traces de transaction validées avant la marque du CheckPoint. C'est ce qui est montré dans (Figure IV-12 : journal Redo).

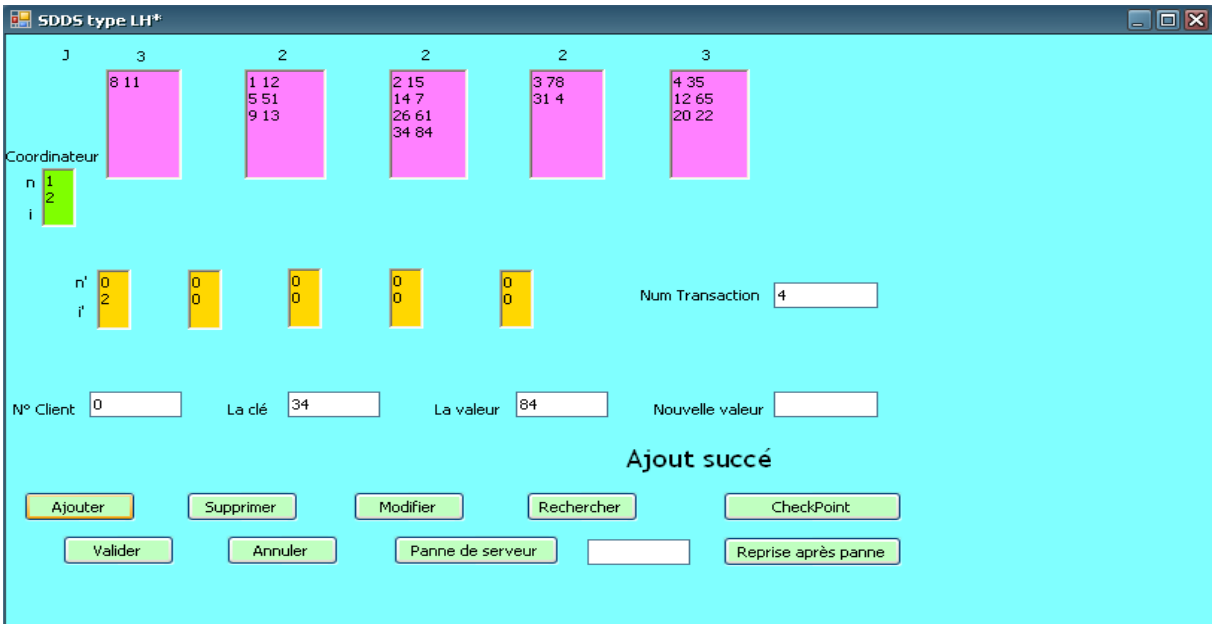


Figure 0-13 : état des cases avant une annulation

Maintenant le client n° 0 lance la transaction 3 qui insère les couples (clé, donnée) dans cet ordre : insert (8, 11), insert (2, 15), insert (3, 78), puis valide son exécution.

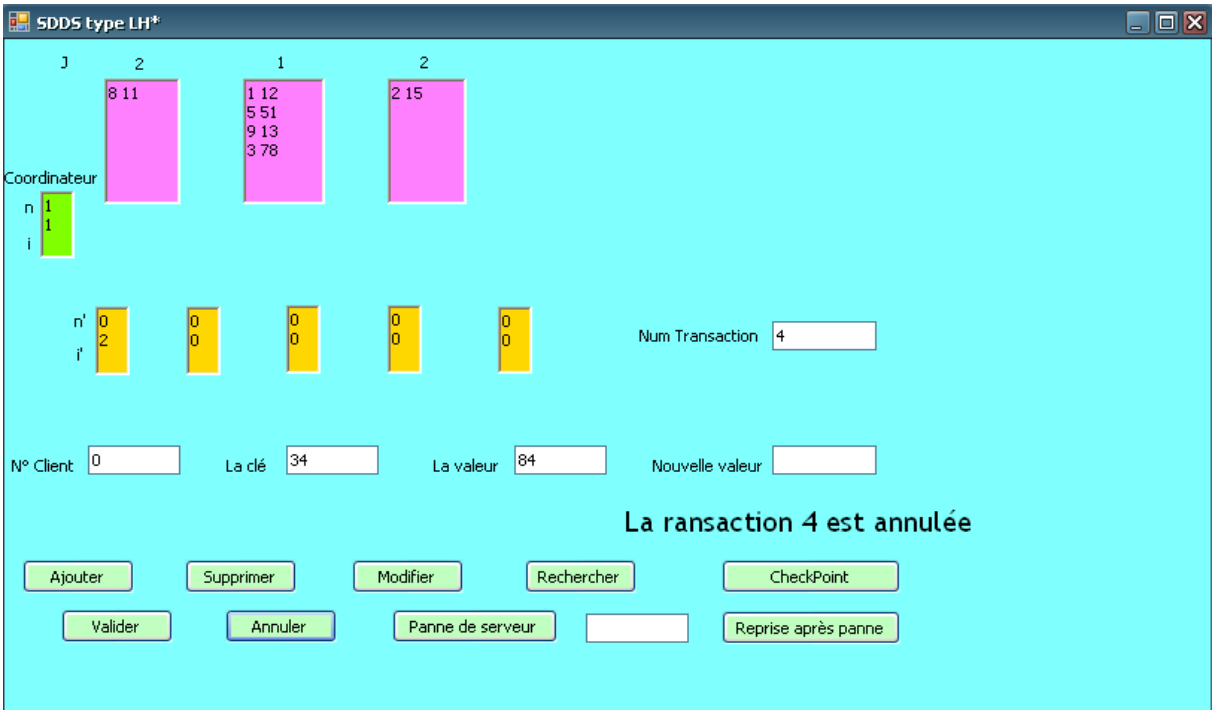


Figure 0-14 : annulation d'une transaction

Le même client (numéro 0) lance alors la transaction n° 4 qui contient les opérations : insert (4, 35), insert (12, 65), insert (14, 7), insert (20, 22), insert (26, 61), insert (31, 4), insert (34, 84). Après ces opérations d'insertions on retrouve l'état illustré dans (Figure IV-13 : état des cases avant une annulation), maintenant supposons que cette transaction (n° 4) va annuler ses effets, donc on retrouvera l'état montré dans (Figure IV-14 : annulation d'une transaction).

Maintenant si on fait un CheckPoint, on aura alors l'état mentionné dans (Figure IV-15 : point de contrôle (CheckPoint)).

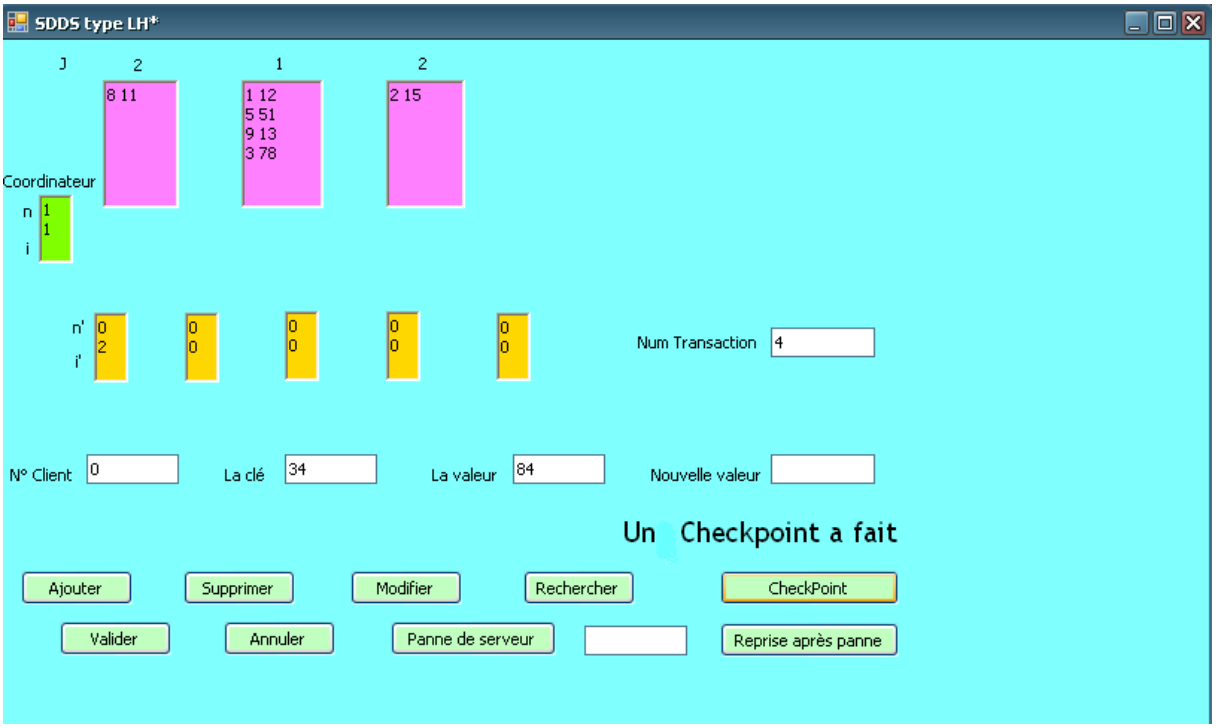


Figure 0-15 : point de contrôle (CheckPoint)

La figure (Figure IV-16 : validation d'une transaction pour voir le Redo) montre l'état du système après que le client n° 0 relance et valide la transaction 4 qui contient une série d'insertion comme suit : insert (6, 33), insert (10, 99), insert (20, 73), insert (46, 122), insert (23, 53), insert (76, 37), insert (81, 19). Le journal REDO va contenir les traces des transactions validées avant et après un point de contrôle (CheckPoint), comme illustré dans (Figure IV-17 : Redo avant et après un CheckPoint).

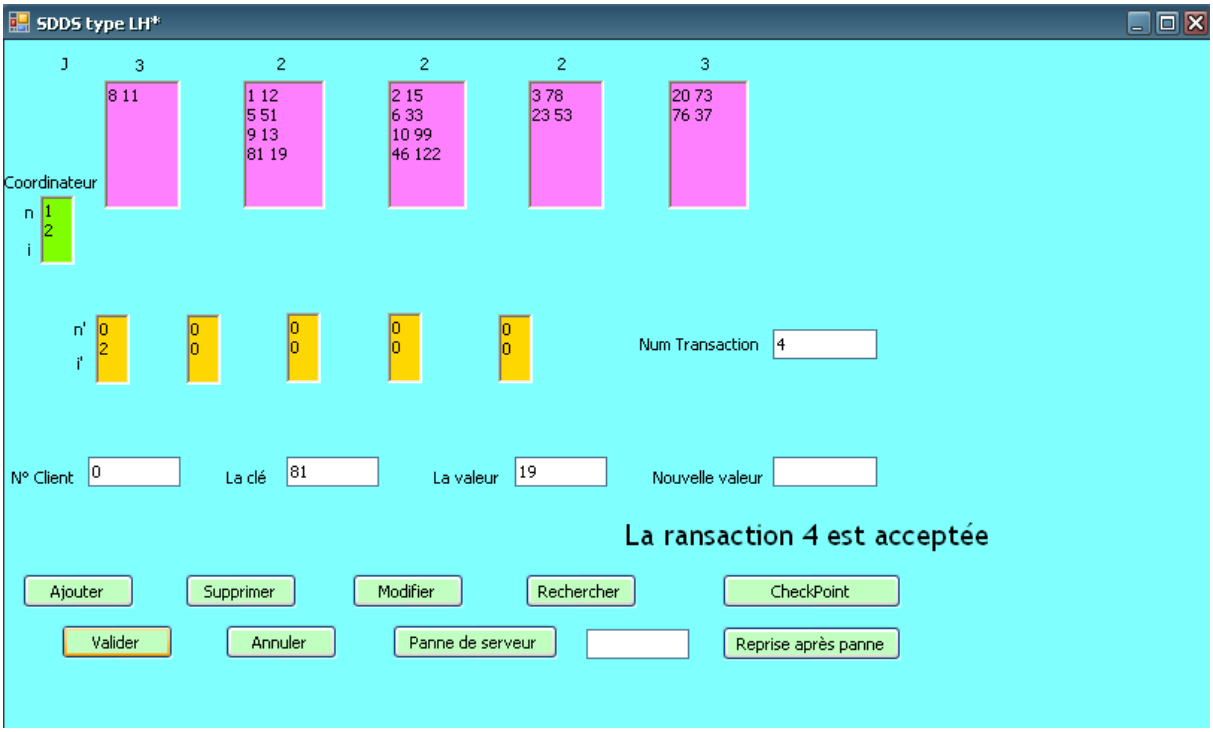


Figure 0-16 : validation d'une transaction pour voir le Redo

Donc la figure ci-dessous montre les données de chaque serveur au moment de faire le CheckPoint, ainsi que le journal REDO complet ; avant et après le CheckPoint.

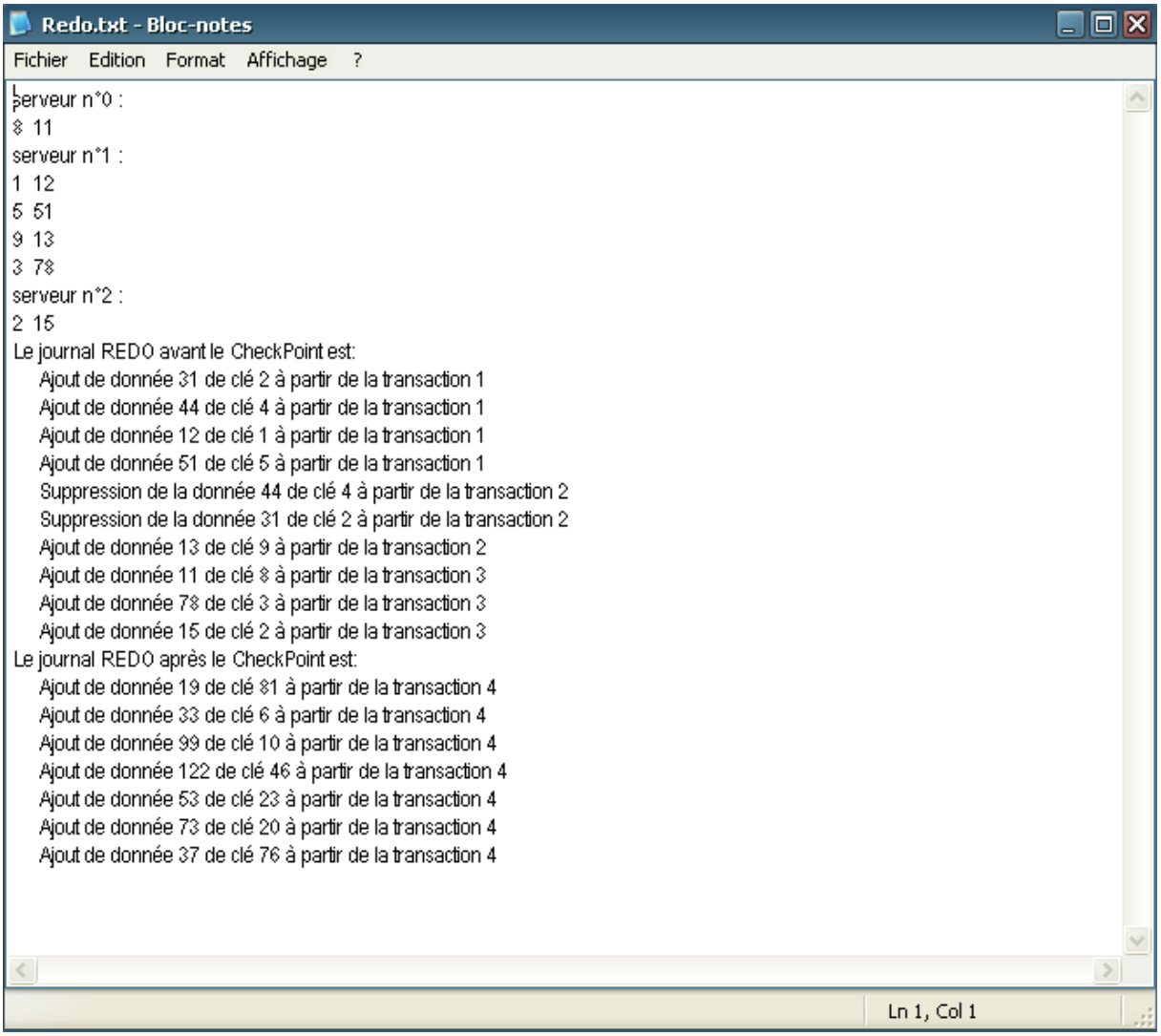


Figure 0-17 : Redo avant et après un CheckPoint

La figure (Figure IV-18 : recherche et valeurs présumées exactes au client n°2) montre une recherche sur la donnée de clé 20, effectuée par la transaction 5 du client n° 2.

Rappelons-nous qu'une opération de recherche exige de remplir les champs de : n° de lient, n° de transaction et la clé cherchée.

Remarque :

Un client peut prendre des valeurs i` et n` correctement à partir de la première image d'ajustement, comme c'est le cas avec le client n° 2 dans la figure (Figure IV-18 : recherche et valeurs présumées exactes au client n°2).

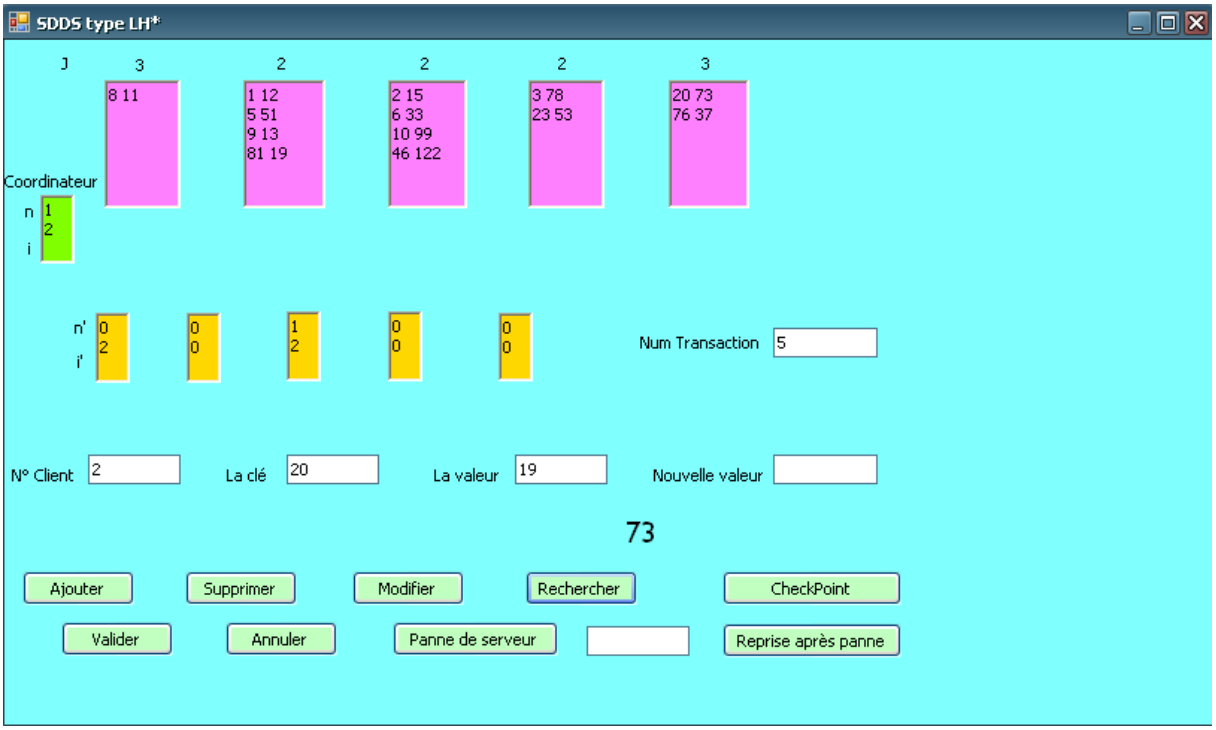


Figure 0-18 : recherche et valeurs présumées exactes au client n°2

Maintenant Dans le cas d'une opération d'une modification d'une donnée, l'enregistrement du journal UNDO/REDO contiendra deux valeurs, l'ancienne et la nouvelle, pour pouvoir faire l'annulation d'une transaction le cas échéant ou alors la valider en stockant ses effets sur la base.

Remarques :

- Le journal REDO global ne va contenir qu'une seule donnée (la nouvelle) de chaque enregistrement UNDO/REDO pour une opération de modification.
- Une opération de modification comporte les champs : n° client, n° transaction, la clé de la donnée qui on va modifier, et enfin la nouvelle donnée, mais l'ancienne donnée qui sera journalisée au niveau du journal UNDO, elle sera récupérée depuis les données dans la case.

La figure (Figure IV-19 : modification d'une donnée) montre la modification d'une donnée de la clé 20 par la donnée 100. Cette opération est issue de la transaction 5, lancée par le client n° 2.

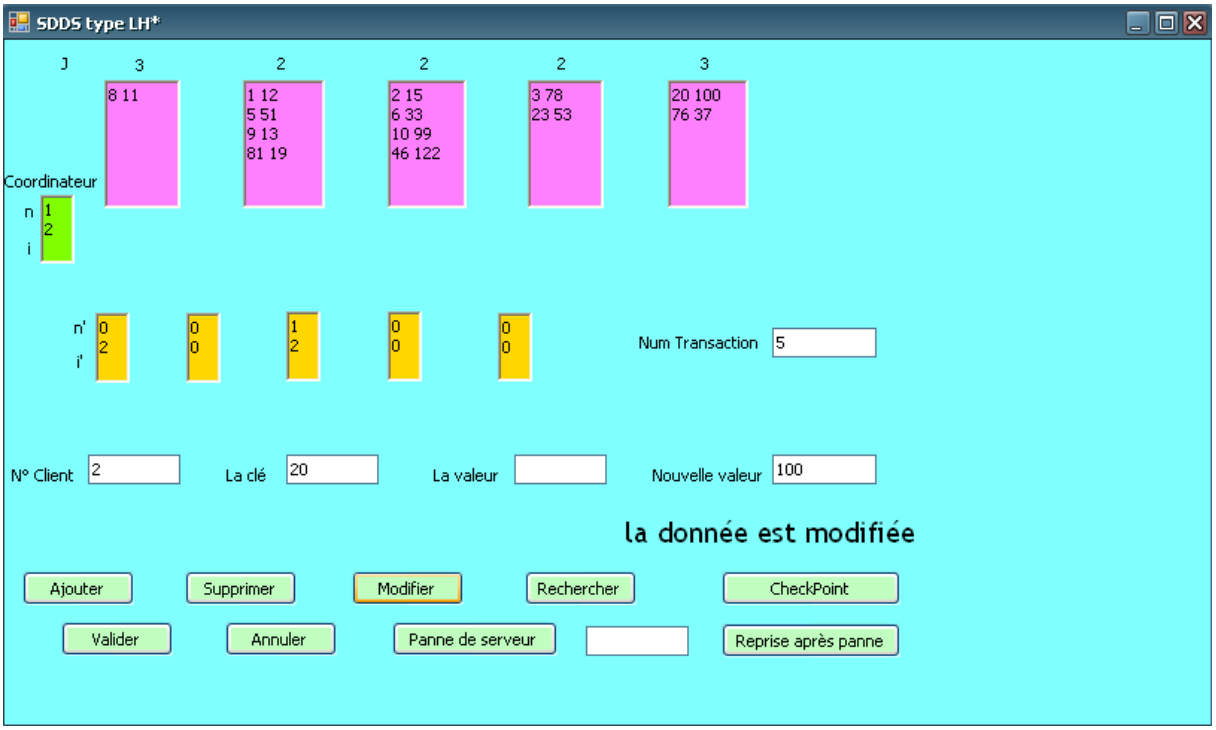


Figure 0-19 : modification d'une donnée

Maintenant et après la validation de la transaction 5, supposons que le serveur n° 1 tombe en panne, comme illustré dans (Figure IV-20 : serveur n°1 tombe en panne).

Remarque :

- On ne peut pas utiliser un serveur durant une panne. Deux situations sont alors possibles : La première correspond à un accès pour utiliser une donnée existante (modification, recherche, suppression) voir (Figure IV-21 : les données du serveur en panne seront temporairement indisponibles) ; la réponse doit être que cette clé n'est pas existé pas. La deuxième façon correspond à une tentative d'ajout d'une nouvelle donnée dans ce serveur, la réponse doit être un refus de l'opération car ce serveur est tombé en panne, voir (Figure IV-22 : pas de possibilité d'insertion dans un serveur tomba en panne).

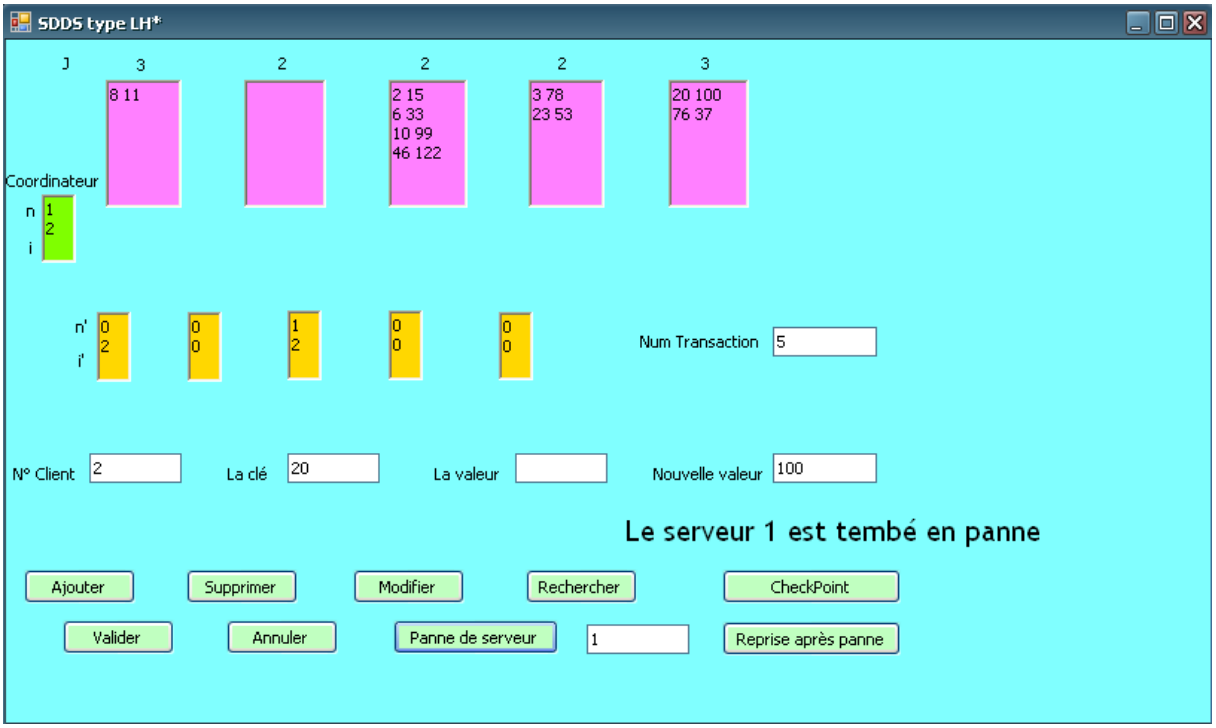


Figure 0-20 : serveur n°1 tombe en panne

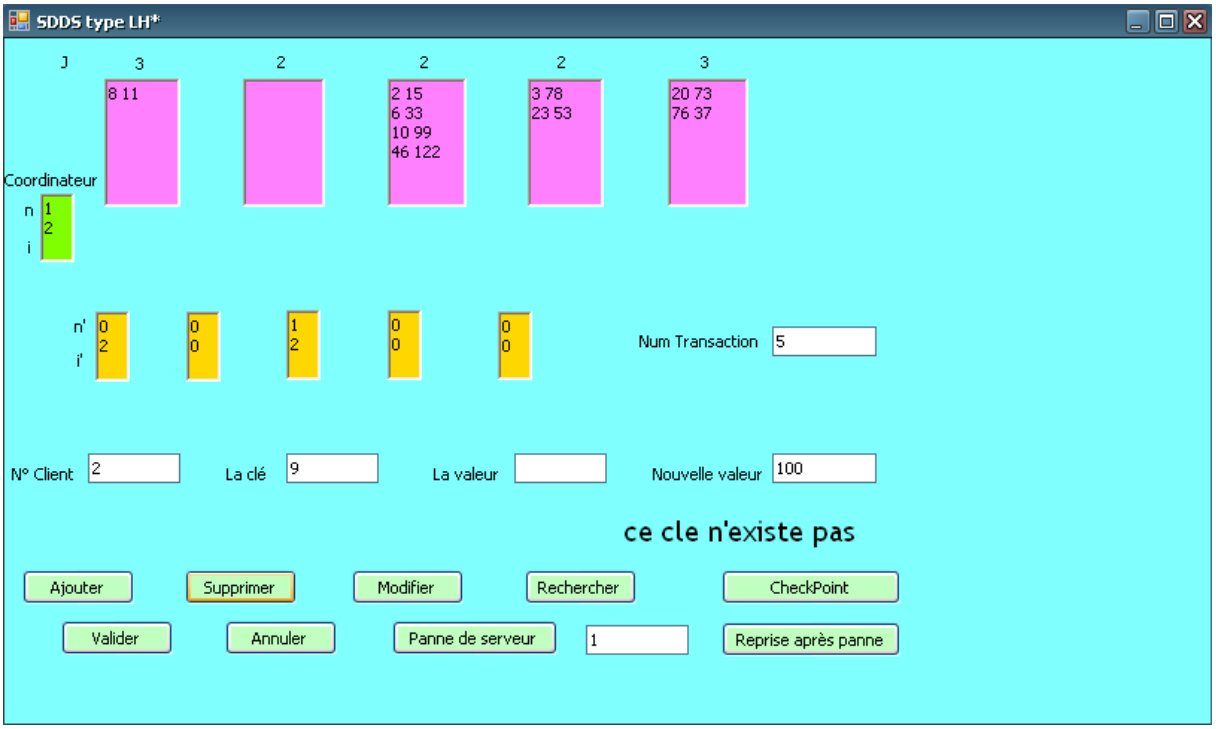


Figure 0-21 : utilisation des données du serveur tomba en panne comme ce sont n'existe pas

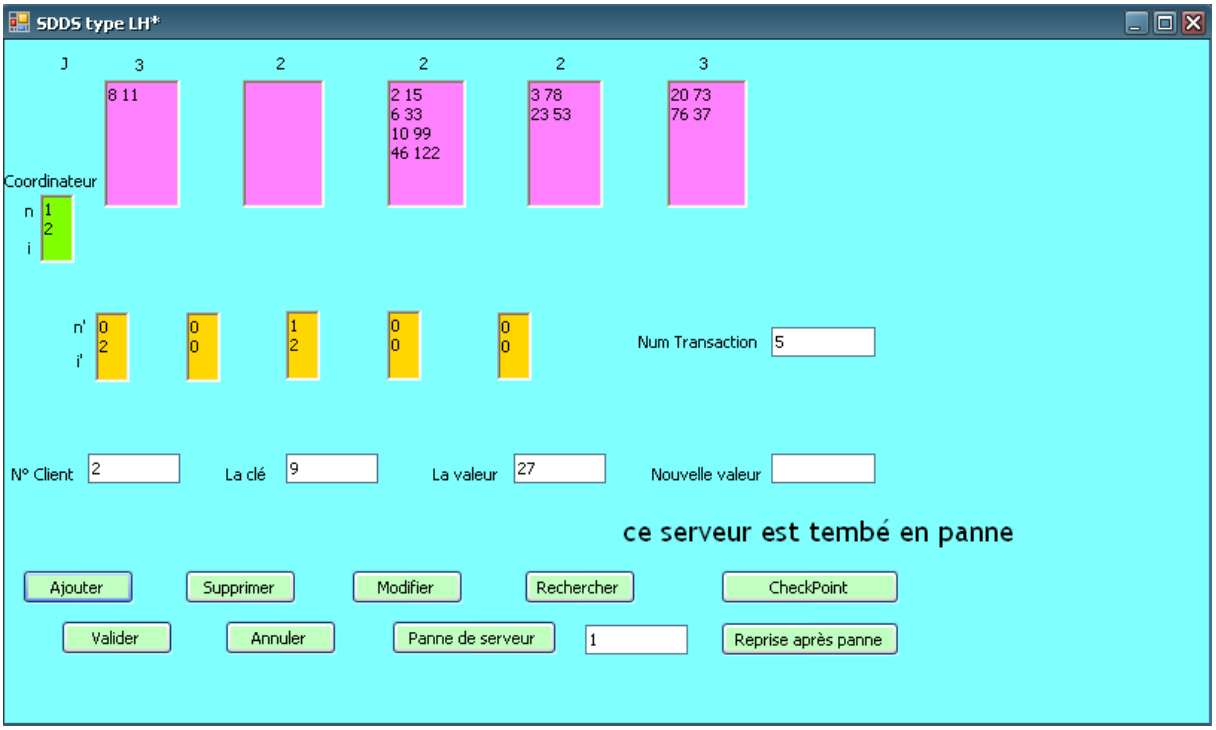


Figure 0-22 : pas de possibilité d'insertion dans un serveur tomba en panne

Donc si on simule la panne d'un serveur (avec le bouton Panne de serveur), le résultat sera de la perte de toutes les données de la case 1, ainsi que son journal UNDO/REDO.

Le bouton Reprise après panne, permet d'enclencher la procédure reprise (Figure IV-23 : reprise du serveur).

Remarque :

On remarque dans (Figure IV-23 : reprise du serveur) ci-dessous que les cases ne contiennent pas les même données qu'elles avaient avant la panne !

La réponse est qu'on a une annulation de la transaction 5 qui n'était pas validée au moment de la panne, donc ses modifications ne seront pas reprises dans le dernier état cohérent de la base.

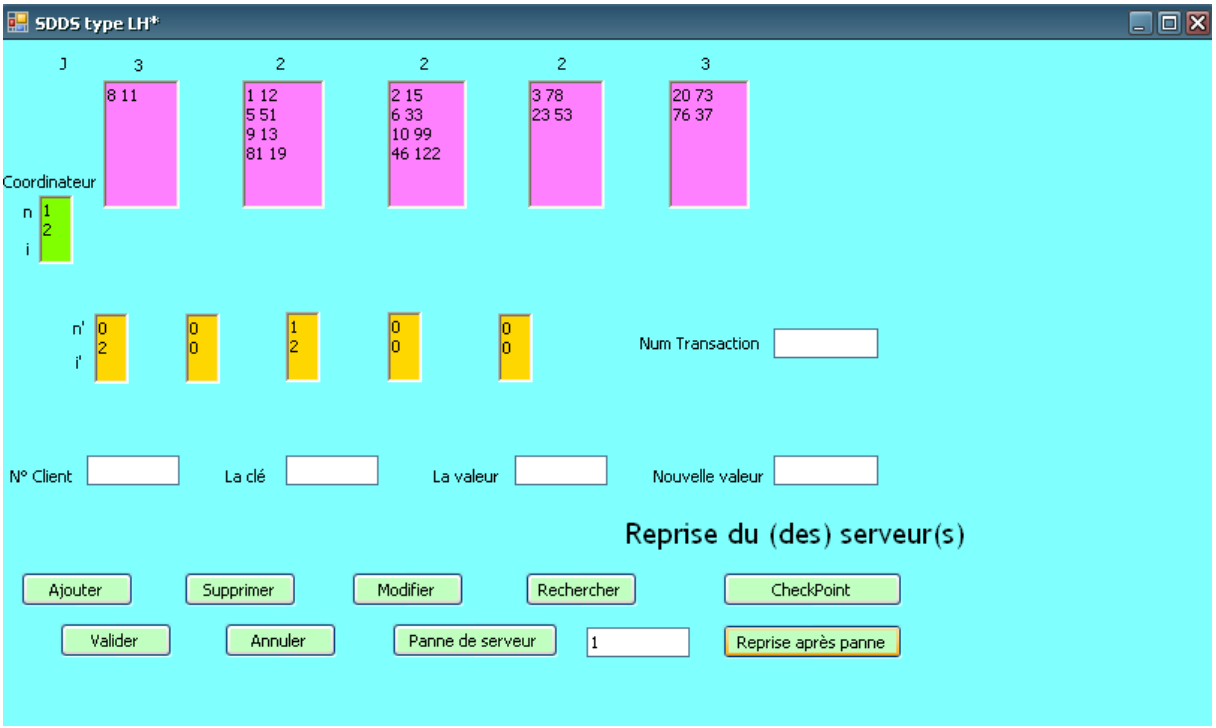


Figure 0-23 : reprise du serveur

Rappel : nous avons parlé précédemment de ce type de la reprise et du recouvrement, et nous avons dit qu'il est très coûteux, alors on va proposer d'une solution à ce problème.

IV.3.Conclusion

Nous avons présenté dans ce chapitre une série de déroulements de notre prototype implémentant les méthodes de journalisation proposées dans le chapitre précédent. Ces déroulements ont pour objectif de mieux mettre en évidence les problèmes liés aux pannes de serveurs LH* provoquant la perte de la mémoire centrale. A travers ces exemples nous avons montré comment on peut retrouver le dernier état cohérent de la base en sauvegardant sur mémoire stable (le disque) les enregistrements de type REDO des transactions validées uniquement. Les opérations d'éclatement et de fusion de cases sont aussi journalisées de manière satisfaisantes. N'importe quelle transaction ayant provoqué de telles opérations peut être annulées (avant sa validation) sans causer aucun problème à la structure du fichier réparti contenant la base. De même les effets de toutes les transactions validées ayant provoquées des éclatements et/ou des fusions de cases, peuvent être rétablit lors d'une reprise après panne, car le journal REDO global et le dernier CheckPoint effectué, contiennent assez d'informations pour reconstruire correctement les serveurs qui étaient en panne.

Conclusion générale

Nous avons présenté dans ce mémoire, des solutions permettant de doter la SDDS LH* de techniques de recouvrement lui permettant de retrouver le dernier état cohérent des données après l'occurrence d'une panne provoquant la perte de la mémoire centrale. L'état cohérent, inclut les modifications de toutes les transactions validées avant la panne.

Notre solution tient compte des opérations d'insertion provoquant des éclatements de cases et des opérations de suppression provoquant des fusions de cases. Elle permet de défaire ou alors refaire toute transaction comportant de telles opérations. Ce qui permet de traiter correctement les échecs de transactions (défaire les transactions annulées) et les reprises après pannes (refaire les transactions validées) et ce même s'il y a eu des modifications dans la structure du fichier de données (éclatement/fusion de cases).

Pour purger la taille des journaux et pour accélérer la procédure de reprise après pannes, notre solution réalise des points de contrôle (checkpoints) dans le journal Redo. Cela permet d'éviter de refaire toutes les transactions validées depuis l'initialisation du système, lors de la reprise d'une panne.

Comme perspectives à ce travail, nous proposons à court termes de réaliser un contrôleur de concurrence basé sur le verrouillage multi-niveau à deux phases strict (2PL) au niveau de chaque serveur LH*, ainsi que la gestion de l'interblocage. Une deuxième proposition à court terme concerne l'implémentation du protocole de validation atomique des transactions (2PC) pour la deuxième solution proposée (avec journaux Redo locaux).

A plus long termes, on propose l'adaptation des checkpoints inconsistant dans notre approche. Cela permettra de réaliser le checkpoint sans arrêter l'exécution des transactions durant l'opération. Nous pouvons aussi proposer comme perspective l'utilisation d'une solution similaire à la nôtre pour d'autre SDDS.

REFERENCE BIBLIOGRAPIQUE

[AMR04] Amrouche Hakim, mémoire pour l'obtention du diplôme de magister en informatique, thème : *TRANS_ACT :* ***Un gestionnaire de transactions pour le SGBD parallèle à base d'acteurs*** *ACT21*. 2004-2005.

[AMR09] Amrouche K., Support de cours de bases de données, 4SIQ, ESI, 2009.

[BER82] Philip A.Bernstein, Nathan Goodman, ***A Sophisticate's Introduction to Distributed Concurrency Control***, International Conference on Very Large Databases, September 8-10, 1982, Mexico.

[BER87] Philip A.Bernstein, Vassos Hadzilacos, and Nathan Goodman, ***Concurrency Control and Recovery In Database Systems***,Addison-Wesly Longman,1987.

[BOU02] D.Boukhelef & D.E Zegour**:** *IH* :* ***Hachage Linéaire Multidimensionnel Distribuée et Scalable***. Conférence Africaine de Recherche en Informatique,CARI 2002. Yaoundé (Cameroun) 14-17 octobre 2002.

[BUR83] W. A. Burkhard, ***Interpolation-based Index Maintenance***, In Proceedings ofthe 2nd Symposium on Principles of databases Systems.PODS, 1983.

[CHE06] S.Chellouche, Z.Gharbi,***Gestion de transactions et reprise après pannes pour la SDDS Compact TreeHashing (CTH*)***, Mémoire d'ingénieur. INI 2006

[DEV93] R. Devine, ***Design and Implementation of DDH: A Distributed Dynamic Hashing Algorithm***, 4th International Foundation of Data Organization and Algorithms–FODO, 1993.

[DIE01] A.W. Diene, ***Contribution à la Gestion de Structures de Données Distribuées et Scalables***, thèse de doctorat, Université Paris Dauphine, novembre 2001.

[DON01] D.Didier, ***Rappel sur les transactions***.Université Joseph Fourier (Grenoble 1).

[DON02] D.Didier, ***Le Contrôle de la Concurrence dans les SGBDs,*** Université Joseph Fourier (Grenoble 1).

[FAG79] R. Fagin, J. Nieverjelt, N. Pippenger, ***Extendible Hashing – A Fast Access Method for Dynamic Files***, ACM Transactions on Database Sytems 1979.

[HID07] W.K. Hidouci, ***Vers une approche des transactions avancées dans les SGBDs Parallèles intégrant les modèles «acteur» et « SDDS » (Scalable Distributed Data Structures),*** Thèse de doctorat, ESI**,** juillet 2007.

[HIL97] V. Hilford, F. B. Bastani, B. Cukic, *EH*: **Extendible Hashing in a Distributed Environment***, COMPSAC '97 - 21st International ComputerSoftware and Applications Conference, 1997.

[KAR96] J. Karlson, W. Litwin, T. Risch, *LH*LH: **A Scalable high performance data structure for switched multicomputers***, EDBT 96, Springer Verlag.

[KAR97] J. Karlson: ***A Scalable Data Structure for A Parallel Data Server***. Thèse de doctorat, 1997.

[LAR78] P. Larson, ***DynamicHashing***, BIT Vol. 18-(2), 1978.

[LIT80] W. Litwin, W. *LINEAR HASHING: a **new tool for file and tables addressing***. Reprint from vldb-80.

[LIT93] W. Litwin. MA. Neimat, D. Schneider, *LH*: **Linear Hashing for Distributed Files***, ACM-SIGMOD International Conference on Management of Data,1993.

[LIT94] W. Litwin, M.A. Neimat& D. Schneider, *RP*: **A Family of Order-preserving Scalable Distributed Data Structures***, In Proceedings of the 20th VLDBConference, Santiago, Chilli, 1994.

[LIT96a] W. Litwin , Marie-Anna Neimat , Donovan A. Schneider, ***LH* a scalable, distributed data structure,*** ACM Transactions on DatabaseSystems (TODS), v.21 n.4, p.480-525, Dec. 1996.

[LIT96b] W. Litwin, M.A. Neimat, ***AHigh-Availability LH* Schemes with Mirroring***,Intl. Conf. on Coope, Inf. Syst. COOPIS-96, Bruxelles, 1996.

[LIT97a] W. Litwin, M.A. Neimat, G. Levy, S. Ndiaye, *LH*S **:A High-availability and High-security Scalable Distributed Data Structure***, IEEE Workshop on Res.Issues in Data Engineering, 1997.

[LIT97b] W. Litwin, T. Risch, *LH*g: **A High-Availability Scalable data Structure by Record Grouping***, Res. Rep. U. Paris9 & U. Linkoping, 1997.

[LIT98] W. Litwin, J. Menon, T. Risch, ***LH* with Scalable Availability***, IBMAlmaden research report RJ 10121 (91937), Mai 1998.

[LIT99] W.Litwin, T.Schwartz, S.J: ***LH*RS, A High Availability Scalable Distributed Data Structure Using Reed Solomon Codes***. Rapport techniqueU Paris IX Dauphine 1999.

[LIU09]L.Liu, T.Özsu : Encyclopedia of database systems, 2009.

[LOU05] N.Lounes, ***RAP_ACT : reprise après pannes dans le SGBD parallèle à base d'acteurs ACT21,*** thèse de Magister, Ecole Supérieure d'informatique(EX : INI),2005.

[MOU04] R. Moussa, ***Contribution à la Conception et l'Implantation de la Structure de Données Distribuée & Scalable à Haute Disponibilité LH*RS,*** thèse dedoctorat, Université Paris Dauphine, 4 octobre 2004.

[TAN94] Andrew. S. Tanenbaum, ***Concepts des systèmes d'exploitation, systèmes centralisés, systèmes distribués,*** Interdisions, 1994.

[YAK07] H. Yakouben, W. Litwin& T. Schwarz. LH*RSP2P: ***A Scalable Distributed Data Structure for the P2P Environment***. 8th annual international conferenceon New Technologies of Distributed Systems. Lyon June 2007.

[ZEG04] D. E. Zegour: ***Scalable distributed compact trie hashing (CTH*)***. Informationand Software Technology 46 (2004) 923-935.