République Algérienne Démocratique et Populaire

Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
Université Mouloud MAMMERI, Tizi-Ouzou

Faculté de Génie Electrique et d'Informatique
Département d'Informatique

MEMOIRE DE FIN D'ETUDES
De MASTER ACADEMIQUE

Domaine : **Mathématique et Informatique**
Filière : **Informatique**
Spécialité : **Conduite de Projets Informatiques**

Thème

Simulation de la méthode Range Partitioning RP*c

Proposé et encadré par :

Mr AIT EL HADJ

Présenté par :

Mme LABADI Naima

Mr TIGUERCHA Aghiles

Année universitaire 2019/2020

RESUME

L'informatique d'aujourd'hui est centrée sur le réseau et non plus sur l'ordinateur isolé. C'est là une évolution majeure. La généralisation des réseaux et l'irruption d'Internet en sont des preuves éclatantes. Des avancées majeures ont été accomplies dans les performances des réseaux (standardisation de TCP/IP, Fast Ethernet, ATM, commutateurs larges bandes etc...). Un nouveau concept est apparu: celui de multi ordinateur, offrant des capacités quasi illimitées de calcul, de mémoire vive et de stockage.

Les structures de données traditionnelles n'arrivent pas à tirer profit des capacités supplémentaires en mémoire vive. Une nouvelle famille de structures de données appelée SDDS a vu le jour pour pallier cette insuffisance. Il s'agit de fichiers résidants en mémoire vive distribuée permettant un accès beaucoup plus rapide que les structures de données classiques sur des disques. Ainsi elle permet d'assurer la disponibilité des données, leur distribution et leur scalabilité (maintien des performances en cas d'accroissement du volume de données stockées).

Plusieurs de ces SDDS ont été proposées par Litwin, Neimat et Schneider. Les plus connues sont classifiées en deux grandes familles: celles basées sur le hachage linéaire, Linear Haching en anglais (LH*) et celles utilisant le partitionnement par intervalle dite Range Partitioning (RP*).

Le but de notre mémoire est de démontrer les performances et les capacités des SDDS et pour cela nous avons étudié de plus près la méthode RP*c. Notre travail consiste à simuler le comportement d'un fichier RP*c distribué sur un certain nombre de serveurs. Et nous avons pu atteindre notre objectif en utilisant trois clients RP*c qui vont solliciter des serveurs RP*c pour insérer des clés dans un fichier RP*c.

Mot clé : Internet, TCP/IP, Fast Ethernet, ATM, commutateurs larges bandes, multi-ordinateur, structures de données, SDDS, hachage linéaire Linear Haching (LH*), partitionnement par intervalle, Range Partitioning (RP*), RP*c distribué

Remerciements

Nos remerciements sincères vont :

A notre promoteur ***Mr A. AIT EL HADJ*** *pour son aide précieuse et pour le temps qu'il nous a accordé pour nous accompagner tout au long de notre travail.*

Aux membres de jury qui sont les honorables examinateurs de ce mémoire, pour leur lecture attentive, ainsi que pour leurs critiques et suggestions constructives.

A toutes les personnes qui nous ont aidées, encouragé et soutenu afin que ce travail puisse être accompli, nous leur exprimons notre gratitude.

Dédicaces

Je dédie ce modeste travail accompagné d'un profond amour :

A la mémoire de :

***Ma très chère et regrettée mère**, qui nous a quittés trop tôt. « Ton départ n'effacera jamais le souvenir des jours heureux pleins d'affections et d'amour. Je ne saurais exprimer mon grand chagrin en ton absence »*

***Mon cher frère Mourad et ma chère nièce Dyhia** qui sont partis très jeunes, laissant derrière eux un vide immense plein de chagrin.*

***Mon cher beau-père** de qui j'ai gardé l'image d'un homme brave et gentil.*

« Que dieu vous accueille tous dans son vaste paradis »

***A mon très cher père**, qui est la source de ma volonté et de mes forces, car il m'a toujours encouragé et poussé vers l'avant. Si je suis arrivée à ce niveau d'étude c'est grâce à toi Papa, un grand merci*

***A mon très cher mari**, qui m'a soutenu et encouragé pour arriver au bout de mon objectif. Merci pour ton soutien, ta patience et ta complicité.*

***A mes trois filles adorées : Dassine, Alycia, Eléna**. Vous êtes mes princesses, ma raison de vivre et ma source de joie. Je souhaite voir le jour de vos réussites.*

***A mes chères sœurs,** qui m'ont soutenu et encouragé, et leurs petites familles*

***A mes chers frères Nacère et Youcef,** qui ont toujours souhaité ma réussite, et leurs petites familles*

***A mon cher frère kamal**. Que dieu te protège et t'ouvre les portes de la réussite*

***A ma chère belle-mère, mes belles-sœurs** et leurs petites familles, **mon beau-frère** et sa petite famille.*

***A mon cher camarade Mr TIGUERCHA Aghiles**, je te souhaite une succession de réussite dans ta vie, que cette dernière soit pleine de bonheur et de prospérité*

Naima

Je dédie ce modeste travail :

𝒜 ma très chère mère :

Autant de phrases aussi expressives soient-elles ne sauraient montrer le degré d'amour que j'éprouve pour toi. Tu m'as comblé avec ta tendresse tout au long de mon parcours. Tu n'as cessé de me soutenir et de m'encourager durant toutes les années de mes études ; Que Dieu te procure bonne santé et longue vie.

𝒜 mon très cher père :

Autant de phrases et d'expressions aussi éloquentes soient-elles ne sauraient exprimer ma gratitude. Tu as su m'inculquer le sens de la responsabilité, de l'optimisme et de la confiance en soi face aux difficultés de la vie. Tes conseils ont toujours guidé mes pas vers la réussite ; Que Dieu le tout puissant te préserve, t'accorde santé, bonheur, quiétude d'esprit et te protège de tout mal.

𝒜 mes chères sœurs :

Que j'aime énormément et qui m'ont beaucoup encouragé et soutenu au moment que j'allais baisser les bras ; Que Dieu vous protège.

𝒜 mes chers frères :

En souvenir d'une enfance dont je suis le plus petit d'entre vous, nous avons partagé les meilleurs et les plus agréables moments.

Pour toute la complicité et l'entente qui nous unissent.

𝒜 ma très chère fiancée Souhila :

Ton encouragement et ton soutien étaient la bouffé d'oxygène qui me ressourçait dans les moments pénibles, de solitude et de souffrance. Merci d'être toujours à mes côtés. Je prie Dieu le tout puissant pour qu'il te donne bonheur et prospérité.

𝒜 ma chère camarade Mme LABADI Naima :

Puisse Dieu, le tout puissant vous préserve du mal, vous comble de santé, de bonheur et vous procure une longue vie.

Aghiles

Table des matières

Liste des abréviations

RAM Mémoire vive (Random Access Memory)

SGBD Système de Gestion de Bases de Données

SDDS Structure de Données Distribuées et Scalables

LH* Linear Haching

RP* Range Partitioning

RP*n Range Partitioning de type n

RP*c Range Partitioning de type c

RP*s Range Partitioning de type s

RP*HA Range Partitioning de type HA

TDA Type de Données Abstrait

Sdd Structure de données

LIFO Last In, First Out

FIFO First In, First Out

IAM Image Adjustement Message

Arbre 1-d Arbre 1 dimensionnel

Arbre k-d Arbre k dimensionnel

Liste des figures

Introduction Générale

En programmation informatique, une structure de données peut être sélectionnée ou conçue pour stocker des données de manière à pouvoir manipuler ces dernières à l'aide de plusieurs algorithmes. Chaque structure de données contient des informations sur la valeur des données, les relations entre elles et les fonctions applicables. Elle est l'implémentation explicite d'un ensemble organisé d'objets, avec la réalisation des opérations d'accès, de construction et de modification afférentes. S'il existe plusieurs types de structures plus ou moins complexes, tous visent à organiser les données pour répondre à un besoin précis, afin de pouvoir y accéder et les traiter de façon appropriée.

Dans les langages de programmation, les structures de données permettent d'organiser le code et les informations dans l'espace numérique. Par exemple, les listes et les dictionnaires Python, ou les tableaux et objets JavaScript sont des structures de codage couramment utilisées pour stocker et récupérer des informations. Les structures de données constituent également un élément essentiel dans la conception de logiciels efficaces.

Avec ces définitions, une structure de données est la réalisation, l'implémentation explicite d'un type de données. Décrire un type de données, le spécifier comme on dit, c'est décrire les opérations possibles et licites, et leur effet. Décrire une structure de données, c'est expliciter comment les objets sont représentés et comment les opérations sont implémentées.

Les structures de données sont souvent classées d'après leurs caractéristiques. Elles peuvent être linéaires ou non linéaires, homogènes ou non homogènes, comme elles peuvent être statiques ou dynamiques.

Les structures de données sont indispensables pour gérer efficacement de grandes quantités de données, comme les informations stockées dans une base de données ou des services d'indexation. La bonne gestion d'un système de données exige la capacité d'identifier la mémoire allouée, les relations entre les données et les processus de données. Or, les structures de données facilitent ces opérations.

Par ailleurs, non seulement il est important d'utiliser des structures de données, mais il est également indispensable de choisir la structure adaptée à chaque tâche. Choisir une structure de données universelle pourrait entraîner un ralentissement des temps de traitement ou une absence de réponse du code. Plusieurs facteurs sont à prendre en compte lors du choix d'une structure de données : le type d'information qui sera stocké, l'emplacement de stockage des données existantes, le mode de stockage des données, et la quantité de mémoire à réserver

aux données. Le type d'une structure de données est déterminé par le type d'opération requis ou par les algorithmes qui seront appliqués et parmi ces types on distingue : les tableaux, les piles, les files, les listes chainées, les arbres, les graphes, les tables de hachage …

Mais nous assistons, du jour en jour, à une évolution poussée de l'informatique dans le domaine de la technologie notamment celle des réseaux informatiques. Ces derniers offrent des débits de plus en plus élevés. Plusieurs ordinateurs peuvent être interconnectés, permettant ainsi de cumuler leurs ressources en mémoire et disque pour le stockage de données, et d'augmenter leur puissance de calculs. De nouveaux concepts sont proposés pour désigner ces systèmes, tels que : multi-ordinateur, ou Réseau de Stations de Travail « Network of Workstation » ou, plus récemment, « Peer-to-Peer Computing » ou « Grid Computing ».

Bien que n'étant pas parfaitement identiques, ces différents concepts visent tous la création d'une plate-forme capable d'exploiter les ressources cumulées de stockage et de traitement. Cela nécessite tout particulièrement de nouvelles méthodes de gestion des données, efficaces pour de grands volumes de donnés réparties. Il s'agit notamment des fichiers résidant en mémoire centrale (RAM) distribuée, pour assurer des temps d'accès beaucoup plus rapides comparés aux structures de données classiques sur des disques. On exige aussi de plus en plus aux fichiers d'un multi-ordinateur la propriété de scalabilité qui signifie la capacité de maintenir les performances d'accès constantes malgré l'accroissement du volume de données stockées. La scalabilité est devenue une exigence capitale pour des applications modernes à usage intensive de données, telles qu'un SGBD, un serveur WEB, un serveur multimédia, un système de calcul scientifique à hautes performances, etc.

Cependant, le problème qui se pose est que les structures de données traditionnelles ne tirent pas profit maximum de ces avancées à cause des nouvelles exigences, à savoir : traitement distribué et parallèle, RAM distribuées, fichiers disque distribués et scalabilité. II est donc nécessaire de trouver une structure de données adéquate à ce nouveau type de super ordinateur.

L'objectif de notre travail est de démontrer l'existence d'une autre structure de données qui est apparue pour pallier à ce manque, et afin d'ouvrir de nouvelles perspectives pour la gestion des données sur les multi ordinateurs. Il s'agit des Structures de Données Distribuées et Scalable, cette classe a été baptisée SDDS : Scalable Distributed Data Structures. Elle est

caractérisée par l'absence d'un site maître, l'absence de dialogue entre les clients et la scalabilté. Cette dernière caractéristique stipule que le fichier peut grandir indéfiniment sans dégradation des performances des opérations sur le fichier (les performances d'accès sont indépendantes de la taille du fichier SDDS).

Les Structures de Données Distribuées et Scalables (SDDS) sont une classe de structures introduites vers les années 1993 par Dr. Witold Litwin et autres, spécifiquement pour la gestion de fichiers sur un multi-ordinateur. Les SDDS utilisent des machines à partage de rien (Nothing sharing), et c'est ces machines qui constituent le multi ordinateur (réseau d'ordinateurs). Un fichier SDDS peut s'étendre dynamiquement, au fur et à mesure des insertions, d'un seul site de stockage à n'importe quel nombre de sites. Cette nouvelle structure supporte le traitement parallèle et assure un temps d'accès aux données beaucoup plus rapide que celui des fichiers classiques stockés sur disque, en outre elle assure une capacité de stockage potentiellement illimitée.

Plusieurs SDDS ont été proposées. Les plus connues sont classifiées en deux grandes familles : celles basées sur le hachage linéaire, Linear Haching en anglais (LH*) et celles utilisant le partitionnement par intervalle dite Range Partitioning (RP*). Nous nous limitons dans notre projet à la présentation de la deuxième famille (RP*) qui constitue le cadre de nos travaux.

Comme toute SDDS, les SDDS RP* sont logées sur des serveurs qui peuvent être accédées par des sites clients en utilisant des requêtes de recherche d'une clé ou des requêtes par intervalles, selon la technique d'adressage utilisée au niveau du client. On trouve trois variétés des SDDS RP* qui sont : les RP*n, les RP*c et les RP*s. on trouve aussi une SDDS à haute disponibilité dénommée RP*HA.

Pour atteindre notre objectif nous avons choisi la méthode RP*c comme principe de nos travaux. Ainsi l'organisation de notre mémoire sera comme suit :

- **Un premier chapitre : Partitionnement des données**, qui explique les notions de base, les avantages et les différents types du partitionnement.
- **Un second chapitre : Structure de données**, dans cette partie nous citons les caractéristiques des structures de données, les différents types existants (linéaire et non linéaire) et les avantages et inconvénients de chacun d'entre eux.

- **Un troisième chapitre : SDDS,** c'est dans ce chapitre que nous avons introduit la nouvelle structure de données appelée (SDDS), et où est expliqué le principe des deux catégories distinguées des SDDS : les SDDS LH* et les SDDS RP*.
- **Un quatrième chapitre : RP* et RP*c**, où nous rentrons dans le détail des différentes variantes de la méthode RP* et par la suite, nous avons étudié plus profondément la variante RP*c.
- **Un cinquième chapitre : Réalisation**, nous avons expliqué les principes de notre programme réalisé et les résultats obtenus après l'exécution de celui-ci.
- Et enfin, nous terminons avec une **Conclusion** pour achever notre travail et entrevoir les perspectives futures.

Chapitre 1

Partitionnement des données

I. Introduction

Les entrepôts de données contiennent souvent de grandes tables, et nécessitent des techniques à la fois pour gérer ces grandes tables et pour fournir de bonnes performances de requête sur ces grandes tables. Pour répondre à ces besoins, une méthodologie clé est nécessaire qui est le partitionnement.

Les fonctionnalités de partitionnement améliorent considérablement l'accès aux données et améliorent aussi les performances globales des applications. Cela est particulièrement vrai pour les applications qui accèdent aux tables et aux index contenant des millions de lignes et de nombreux giga-octets de données.

Le partitionnement consiste à découper une même table ou index, sur des critères logiques, en plusieurs sous-tables (partitions) manipulables en tant que tables à part entières, afin de pouvoir définir des critères de stockage différents pour chacune des partitions et permet un accès plus rapide aux informations (moins de lectures disques à effectuer en particulier). Les tables et index partitionnés facilitent les opérations administratives en permettant à ces opérations de fonctionner sur des sous-ensembles de données.

L'utilisation des méthodes de partitionnement peut nous aider à régler les instructions pour éviter les analyses inutiles des index et des tables (à l'aide de l'élagage de partition). Nous pouvons également améliorer les performances des opérations de jointure massives lorsque de grandes quantités de données (par exemple, plusieurs millions de lignes) sont jointes à l'aide de jointures par partition. Enfin, le partitionnement des données améliore considérablement la gérabilité des très grandes bases de données et réduit considérablement le temps requis pour les tâches administratives telles que la sauvegarde et la restauration.

La granularité d'un schéma de partitionnement peut être facilement modifiée en fractionnant ou en fusionnant des partitions. Ainsi, si les données d'une table sont biaisées pour remplir certaines partitions plus que d'autres, celles qui contiennent le plus de données peuvent être divisées pour obtenir une distribution plus uniforme. Le partitionnement permet également d'échanger des partitions avec une table. En pouvant facilement ajouter, supprimer ou échanger rapidement une grande quantité de données, l'échange peut être utilisé pour garder une grande quantité de données en cours de chargement inaccessible jusqu'à ce que le chargement soit terminé, ou peut être utilisé pour organiser des données entre différentes phases d'utilisation.

II. Définition

Le partitionnement de données (ou data clustering en anglais) est une méthode en analyse des données. Elle vise à diviser un ensemble de données en différents « paquets » homogènes, en ce sens que les données de chaque sous-ensemble partagent des caractéristiques communes, qui correspondent le plus souvent à des critères de proximité (similarité informatique) que l'on définit en introduisant des mesures et classes de distance entre objets.

III. Pourquoi partitionner les données ?

• Améliorer l'évolutivité : Quand on procède à la montée en puissance d'un système de base de données unique, celui-ci finit par atteindre une limite liée au matériel physique. Si on divise les données en plusieurs partitions, chacune hébergée sur un serveur distinct, on peut faire évoluer le système presque indéfiniment.

• Améliorer les performances : Les opérations d'accès aux données présentes au sein de chaque partition interviennent sur un plus petit volume de données. Une fois correctement terminé, le partitionnement peut rendre notre système plus efficace. Les opérations qui affectent plusieurs partitions peuvent s'exécuter en parallèle.

• Améliorer la sécurité : Dans certains cas, on peut séparer les données sensibles et non sensibles en différentes partitions et appliquer différents contrôles de sécurité aux données sensibles.

• Procurer une flexibilité opérationnelle : Le partage offre de nombreuses possibilités d'affiner les opérations, de maximiser l'efficacité administrative et de minimiser les coûts. Par exemple, on peut définir différentes stratégies de gestion, surveillance, sauvegarde et restauration et autres tâches d'administration en fonction de l'importance des données présentes dans chaque partition.

• Faire correspondre le magasin de données au modèle d'utilisation : Le partitionnement permet le déploiement de chaque partition sur un type de magasin de données différent, en fonction du coût et des fonctionnalités intégrées proposées par le magasin de données. Par exemple, il est possible de stocker les données binaires volumineuses dans un stockage d'objet blob et de stocker les données plus structurées dans une base de données de documents.

• Améliorer la disponibilité : Diviser les données au sein de plusieurs serveurs permet d'éviter un point de défaillance unique. Lorsqu'une instance échoue, seules les données de cette partition sont indisponibles. Les opérations intervenant sur les autres partitions peuvent se poursuivre.

IV. Avantages du partitionnement

Le partitionnement apporte plusieurs avantages ses principaux intérêts :

• pouvoir créer des tables plus grandes que la taille permise par un disque dur ou par une partition du système de fichiers : il est tout à fait possible de stocker des partitions à des endroits différents (disques, serveurs…)

• pouvoir supprimer très rapidement des données qui ne sont plus utiles et utilisées : si ces données sont placées sur une partition séparée, il suffit de détruire la partition pour supprimer toutes les données ;

• optimiser grandement certaines requêtes : les données étant organisées dans différentes partitions, le SGBD n'accède qu'aux données nécessaires lors des requêtes. Sans partitionnement, tous les enregistrements sont pris en compte.

V. Le choix du type de partitionnement

Le choix de la méthode de partitionnement d'une table doit être fait avec beaucoup d'attention car les performances de l'optimisation des requêtes et leur exécution peuvent être fortement affectées négativement par un mauvais design.

Souvent, le meilleur choix revient à partitionner par la (ou les) colonne(s) qui apparaissent le plus fréquemment dans les clauses WHERE des requêtes en cours d'exécution sur la table partitionnée. Les éléments de la clause WHERE qui correspondent ou sont compatibles avec la clé de partitionnement peuvent être utilisés pour ignorer les partitions inutiles. La suppression des données inutiles est aussi un facteur à considérer lors de la conception de la stratégie de partitionnement. Une partition entière peut être détachée rapidement, donc il peut être bénéfique de concevoir la stratégie de partitionnement d'une telle façon que toutes les données à supprimer d'un coup soient concentrées sur une seule partition.

Il est aussi important de considérer la surcharge du partitionnement lors de l'optimisation et de l'exécution. Les durées d'optimisation deviennent plus longues et la

consommation de mémoire devient plus importante au fur et à mesure de l'ajout de partitions. Ceci est tout particulièrement vrai pour les commandes UPDATE et DELETE. Une autre raison de se soucier d'un grand nombre de partitions est que la consommation mémoire du serveur pourrait grossir de façon significative sur une période de temps, et tout spécialement si beaucoup de sessions touchent un grand nombre de partitions. Ceci est dû au chargement des métadonnées nécessaires pour chaque partition en mémoire locale.

Avec une charge de type entrepôt de données, il peut être sensé d'utiliser un grand nombre de partitions. En général, dans les entrepôts de données, le temps d'optimisation d'une requête est peu important parce que la majorité du temps de traitement est passée sur l'exécution de la requête. Il est important de prendre les bonnes décisions dès le début, car le repartitionnement de grosses quantités de données peut être très lent. Les simulations de la charge attendue sont souvent bénéfiques pour optimiser la stratégie de partitionnement. Ne jamais supposer qu'un plus grand nombre de partitions est toujours mieux qu'un petit nombre de partitions, et vice-versa.

VI. Les types de partitionnement

Le partitionnement d'une table consiste à découper cette table afin qu'elle soit moins volumineuse, permettant ainsi d'optimiser certains traitements sur cette table.

On distingue :

1. Partitionnement vertical

Le partitionnement vertical consiste à segmenter une table en plusieurs tables contenant chacune un sous-ensemble de colonnes et le même nombre de lignes que la table partitionnée. Les tables de partition ont la même clé primaire.

Les colonnes d'une table sont réparties dans plusieurs partitions. Cela peut être pratique pour écarter des données fréquemment utilisées d'autres auxquelles l'accès est plus rare.

Exemple : nous disposons d'une table Amis contenant les prénoms et photo de chacun de nos amis. Les photos prenant de la place et étant rarement accédées, nous décidons de les écarter des autres données.

Table amis

Id	Nom	Photo
1	Alycia	[BLOB]
2	Lucy	[BLOB]
3	Xavier	[BLOB]
4	Luc	[BLOB]
5	Sophie	[BLOB]

Partition 1

Id	Nom
1	Alycia
2	Lucy
3	Xavier
4	Luc
5	Sophie

Partition 2

Id	Photo
1	[BLOB]
2	[BLOB]
3	[BLOB]
4	[BLOB]
5	[BLOB]

Nous remarquons que, les deux partitions contiennent l'identifiant des amis. Cela est nécessaire afin de garder le lien entre les données de chaque enregistrement. Ainsi, pour récupérer toutes les informations des amis, il suffit de faire une jointure entre les deux partitions.

A- Avantages et inconvénients

Le partitionnement vertical permet aux requêtes de parcourir une quantité moins importante de données, et améliore donc leur temps de réponse. Il permet également de scinder les données qui requièrent différents niveaux de protection, et vous pouvez alors conserver des informations confidentielles dans un partition spéciale.

Cependant, le partitionnement vertical implique également de :

- Créer des jointures et unions pour extraire des données réparties sur plusieurs tables
- Créer des requêtes plus sophistiquées pour déterminer la table contenant les données recherchées
- Ajouter des métadonnées supplémentaires pour décrire la table partitionnée

B- Cas d'utilisation

Un tel découpage permet d'isoler des attributs peut utiliser, d'autres très utilisés, et ainsi améliore les performances lorsque l'on travaille avec les attributs très utilisés (la table étant plus petite).

Cette technique diminue les performances des opérations portant sur des attributs ayant été répartis sur des tables différentes (une opération de jointure étant à présent requise).

Le partitionnement vertical est bien entendu sans intérêt sur les tables comportant peu d'attributs.

2. Partitionnement horizontal

Les enregistrements (= lignes) d'une table sont répartis dans plusieurs partitions. Il est nécessaire de définir une condition de partitionnement, qui servira de règle pour déterminer dans quelle partition ira chaque enregistrement.

Prenons comme exemple une table Amis et nous choisissons de la partitionner en deux partitions :

1. Les amis dont la première lettre du prénom est comprise entre A et M,
2. Les amis dont la première lettre du prénom est comprise entre N et Z.

Table Amis

Id	Nom	Photo
1	Alycia	[BLOB]
2	Lucy	[BLOB]
3	Xavier	[BLOB]
4	Luc	[BLOB]
5	Sophie	[BLOB]

Partition 1

Id	Nom
1	Alycia
2	Lucy
4	Luc

Partition 2

Id	Nom
3	Xavier
5	Sophie

Pour récupérer la liste complète de nos amis, il sera nécessaire de regrouper le contenu de nos deux partitions. Pour ce faire, une simple opération d'union d'ensembles suffit ;

A- Types de partitionnement horizontal

1. **Partitionnement par intervalles (Range partitioning) :** les enregistrements sont répartis en fonction de la valeur d'une colonne, par rapport à un ensemble d'intervalles définissant le domaine d'appartenance de chaque partition.

   Le partitionnement par plage est généralement utilisé pour organiser les données par intervalles de temps sur une colonne de type DATE.

2. **Partitionnement par listes (List partitioning) :** même principe que le partitionnement par intervalles, mais la partition d'un enregistrement est déterminée à partir de listes de valeurs.

3. **Partitionnement par hachage (Hash partitioning) :** la partition à laquelle appartient un enregistrement est déterminée à partir de la valeur de retour d'une fonction définie par l'utilisateur. Cette fonction de hachage doit donc opérer un traitement sur la valeur d'une ou plusieurs colonnes des enregistrements.

4. **Partitionnement par clé (Key partitioning) :** cette méthode de partitionnement est similaire au partitionnement par hachage, à l'exception que la fonction de hachage est fournie par le serveur MySQL.

B- Avantages et inconvénients

Le partitionnement horizontal permet de diviser une grande table en plusieurs autres tables plus petites dans les buts suivants :

- Améliorer le temps de traitement des requêtes
- Accélérer la sauvegarde et la reprise incrémentale
- Réduire le temps de chargement des tables indexées

Toutefois, le partitionnement horizontal nous contraint à :

- Créer des jointures et unions pour extraire des données réparties sur plusieurs tables
- Créer des requêtes plus sophistiquées pour déterminer la table contenant les données recherchées

- Utiliser des métadonnées supplémentaires pour décrire la table partitionnée

C- Cas s'utilisation

Un tel découpage permet d'isoler des enregistrements peu utilisés d'autres très utilisés, et ainsi améliore les performances lorsque l'on travaille avec les enregistrements très utilisés (la table étant plus petite). C'est le cas typique de l'archivage. Un autre critère d'usage est le fait que les enregistrements soient toujours utilisés selon un partitionnement donné (par exemple le mois de l'année). Cette technique diminue les performances des opérations portant sur des enregistrements ayant été répartis sur des tables différentes (une opération d'union étant à présent requise). Le partitionnement horizontal est bien entendu sans intérêt sur les tables comportant peu d'enregistrements.

VII. Conclusion

Dans ce chapitre, nous avons vu que le partitionnement peut améliorer l'évolutivité, réduire les conflits et optimiser les performances. Il peut également fournir un mécanisme pour diviser les données par modèle d'utilisation.

Dans de nombreuses solutions à grande échelle, les données sont divisées en partitions qui peuvent être gérées et accessibles séparément.

Et après avoir cité les différents types de partitionnements, nous insistons sur le fait que la stratégie de partitionnement doit être choisie avec soin pour maximiser les avantages tout en minimisant les effets indésirables.

Nous connaissons maintenant les deux types de partitionnement. Il est bien sûr possible d'utiliser un partitionnement vertical et un partitionnement horizontal sur une même table, ainsi que de partitionner sur plusieurs niveaux, c'est-à-dire définir des partitions de partitions.

Chapitre 2

Structure de données

I- Introduction

En informatique, une donnée est la représentation d'une information sous une forme conventionnelle adaptée à son exploitation. Une variable est alors un objet informatique qui associe un nom à cette représentation et qui, selon les langages, est implicitement ou explicitement typée.

Une structure de données permet de gérer un ensemble de données à partir d'un jeu réduit de méthodes qui sont les seuls moyens d'accéder à tel ou tel élément, de modifier l'ensemble des données, d'en créer un nouveau, etc.

Le principe de base d'une structure de données, c'est de stocker des éléments auxquels le programmeur veut pouvoir accéder plus tard. Une structure de données est la réalisation, l'implémentation explicite d'un type de données. Décrire un type de données, le spécifier comme on dit, c'est décrire les opérations possibles et licites, et leur effet. Décrire une structure de données, c'est expliciter comment les objets sont représentés et comment les opérations sont implémentées. Pour l'évaluation d'algorithmes, il importe de plus de connaître leur coût en temps et en place.

II- Définition

Une structure de données est une méthode de stockage et d'organisation des données pour en faciliter l'accès et la modification. Elle regroupe des données à gérer et un ensemble d'opérations qu'on peut leur appliquer. En général, il existe plusieurs manières de représenter ces données et plusieurs implémentations de leur manipulation.

L'interface sera définie dans un type de données abstrait (TDA). Il spécifie précisément la nature et les propriétés des données à stocker et les modalités des opérations.

III- Importance des structures de données

Les structures de données sont indispensables pour gérer efficacement de grandes quantités de données, comme les informations stockées dans une base de données ou des services d'indexation. La bonne gestion d'un système de données exige la capacité d'identifier la mémoire allouée, les relations entre les données et les processus de données. Or, les structures de données facilitent ces opérations.

Par ailleurs, non seulement il est important d'utiliser des structures de données, mais il est également indispensable de choisir la structure adaptée à chaque tâche. Choisir une structure de données universelle pourrait entraîner un ralentissement des temps de traitement ou une absence de réponse du code. Plusieurs facteurs sont à prendre en compte lors du choix d'une structure de données : le type d'information qui sera stocké, l'emplacement de stockage des données existantes, le mode de stockage des données, et la quantité de mémoire à réserver aux données.

IV- Caractéristiques des structures de données

Les structures de données sont souvent classées d'après leurs caractéristiques :

- Linéaires ou non linéaires : cette caractéristique indique si les éléments de données sont organisés chronologiquement, comme dans un tableau, ou de façon non ordonnée, comme dans un graphe.

- Homogènes ou non homogènes : cette caractéristique indique si tous les éléments de données d'un référentiel spécifique sont du même type ou de types différents.

- Statiques ou dynamiques : cette caractéristique décrit la façon dont les structures de données sont compilées. Les structures statiques présentent des tailles, des structures et des emplacements de mémoire fixes au moment de la compilation. Dans une structure de données dynamique, en revanche, la taille, les structures et les emplacements de mémoire peuvent diminuer ou s'agrandir, selon l'utilisation qui en est faite.

V- Structures de données linéaires

1. Définition

Les structures de données sont considérées comme linéaire si les éléments construisent une séquence d'une liste linéaire. Les éléments sont adjacents les uns aux autres et dans un ordre spécifié. On consomme un espace mémoire linéaire, autrement dit, les éléments doivent être stockés de manière séquentielle en mémoire. Lors de l'implémentation d'une structure de données linéaire, la quantité de mémoire nécessaire est déclarée au préalable. Il n'y a pas une bonne utilisation de la mémoire ce qui entraîne un gaspillage de mémoire. Les éléments sont parcourus de manière séquentielle un par un.

Les structures linéaires tirent leur nom du fait que les données y sont organisées sous forme d'une liste dans laquelle elles sont mises les unes derrière les autres. On peut représenter une telle liste à l'aide d'un tableau unidimensionnel ou sous la forme d'une liste chaînée. Selon la nature des opérations autorisées, on obtient différents types de listes, en particulier les piles et les files.

2. Types de structures de données linéaires

a. Tableau

Un tableau (array en anglais) est une Sdd de base qui est un ensemble d'éléments, auquel on accède à travers un numéro d'indice.

Les tableaux forment une suite de variables de même type associées à des emplacements consécutifs de la mémoire.

Puisque tous les emplacements sont de même type, ils occupent tous le même nombre d de cases mémoire ; connaissant l'adresse a de la première case du tableau, on accède en coût constant à l'adresse de la case d'indice k en calculant a + kd. En revanche, ce type de structure est statique : une fois un tableau créé, la taille de ce dernier ne peut plus être modifiée faute de pouvoir garantir qu'il y a encore un espace mémoire disponible au-delà de la dernière case.

En résumé :

– un tableau est une structure de données statique ;

– les éléments du tableau sont accessibles en lecture et en écriture en temps constant O (1).

Un tableau peut avoir une dimension, on parle alors de vecteur. Comme il peut avoir plusieurs dimensions, on dit qu'il est multidimensionnel. On note T[i][k] quand on a 2 dimensions.

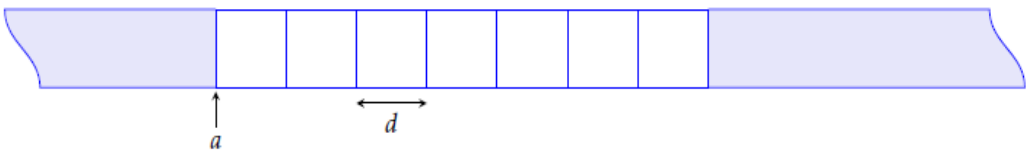


FIGURE 1 Représentation d'un tableau en mémoire

a.1 Avantages

- On peut maintenir une grande quantité de données aux mêmes endroits.
- Chacun des éléments peut être accédé en temps constant O (1) via son index.
- Comme un tableau est fixe au niveau de sa taille, il limite fortement tous les problèmes de mémoire type Overflow

a.2 Inconvénients

- Comme un tableau est fixe au niveau de sa taille, il est impossible de le faire grossir sans créer un nouveau tableau.
- Un tableau ne peut contenir qu'un seul type de donnée à la fois.

b. Liste

Une liste chaînée stocke un ensemble d'éléments de façon linéaire. Chaque élément ou nœud d'une liste chaînée contient un élément de données ainsi qu'une référence, ou lien, vers l'élément suivant de la liste.

Les listes associent à chaque donnée (de même type) un pointeur indiquant la localisation dans la mémoire de la donnée suivante (à l'exception de la dernière, qui pointe vers une valeur particulière indiquant la fin de la liste).

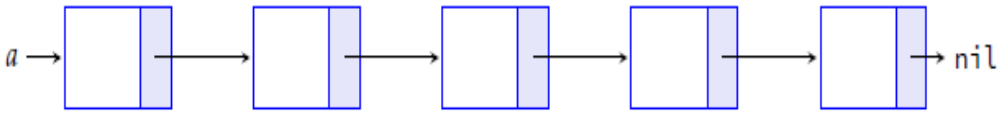


FIGURE 2 Représentation d'une liste linéaire en mémoire

Dans une liste, il est impossible de connaître à l'avance l'adresse d'une case en particulier, à l'exception de la première. Pour accéder à la n^e case il faut donc parcourir les n - 1 précédentes : le coût de l'accès à une case est linéaire. En contrepartie, ce type de structure est dynamique : une fois la liste crée, il est toujours possible de modifier un pointeur pour insérer une case supplémentaire.

En résumé :

– une liste est une structure de donnée dynamique ;

– le $n^{ième}$ élément d'une liste est accessible en temps O(n).

On notera que le type de liste que l'on vient de présenter est le plus courant (il s'agit de listes *chaînées*) mais il en existe d'autres : listes *doublement chaînées* permettant l'accès non seulement à la donnée suivante mais aussi à la donnée précédente, *listes circulaires* dans lesquelles la dernière case pointe vers la première, etc.

Une liste chainée L est entièrement définie par son maillon de tête *L.tete*, c'est à dire l'adresse de son premier maillon.

Il n'existe pas de normalisation pour les primitives de manipulation de liste ; les plus fréquentes sont :

- *est_vide (L)* : renvoie vrai si la liste est vide
- *taille (L)* : renvoie le nombre d'éléments de la liste
- *get_dernier_maillon (L)* : renvoie le dernier élément de la liste
- *get_maillon_indice (L, i)* : renvoie le maillon d'indice i
- *ajouter_debut (L, d)* : ajoute un élément au début de la liste
- *ajouter_fin (L, d)* : ajout un maillon à la fin de la liste
- *inserer_apres (L, i, M)* : insert un maillon à l'indice i
- *supprimer_apres (L, M)* : supprime le maillon suivant le maillon M

Contrairement à ce que pourrait laisser croire son nom, la classe *List* en python n'est pas une liste au sens qu'on vient de lui donner, mais une structure de donnée plus complexe qui cherche à concilier les avantages des tableaux et des listes, à savoir être une structure de donnée dynamique dans laquelle les éléments sont accessibles à coût constant.

b.1 Implémentation

Plusieurs implémentations des listes sont possibles ; le choix de l'implémentation dépend des opérations effectivement utilisées. Si seules les opérations de base sont utilisées, on peut employer une implémentation simple ; pour plus de souplesse, on utilisera une implémentation par liste doublement chaînée. C'est elle qui permet notamment de concaténer deux listes, ou de scinder une liste en temps constant.

b.2 Avantages

- La taille d'une liste chaînée n'est pas fixée dans la mémoire, c'est donc très flexible à l'utilisation
- Certaines opérations d'insertion et de suppression peuvent être faites en temps constant O (1).

b.3 Inconvénients

De l'espace mémoire, en plus de la valeur, est utilisé pour chaque nœud pour les pointeurs.

c. Pile

Il s'agit de structures linéaires dynamiques qui se distinguent par les conditions d'ajout et d'accès aux éléments.

Les piles sont fondées sur le principe du « dernier arrivé, premier sorti » ; on les dit de type LIFO (Last In, First Out). C'est le principe même de la pile d'assiette : c'est la dernière assiette posée sur la pile d'assiettes sales qui sera la première lavée.

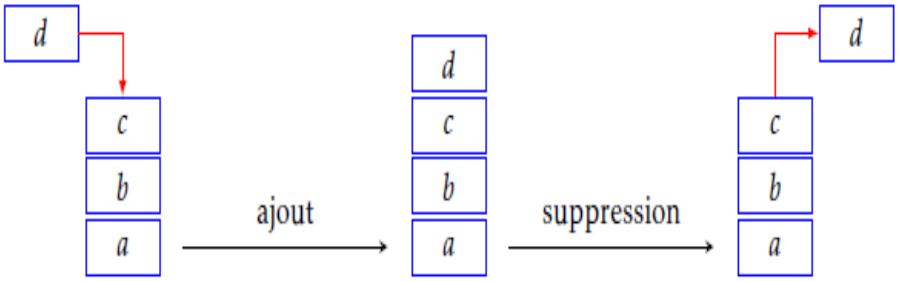


FIGURE 3 Mécanisme d'ajout et de suppression dans une pile

Ce mode de fonctionnement dit Last In First Out (LIFO) est géré via une interface avec quatre opérations principales :

- Push : ajouter un nouvel élément au sommet de la pile
- Pop : enlever l'élément du sommet de la pile
- Top : retourner l'élément du sommet de la pile
- IsEmpty : retourner true si la pile est vide

Ainsi que toutes ces opérations sont exécutées en temps constant O (1), ce qui rend les données traitements extrêmement rapides. Et c'est ça l'avantage d'utiliser des structures de données.

c.1 Avantages

- Permets de gérer des données en mode LIFO
- Certaines opérations seront faites en temps constant O (1).

c.2 Inconvénients

- De par la nature LIFO de la pile, on ne peut pas accéder directement à un élément au milieu de la pile en temps constant O (1) comme avec un tableau
- Créer trop d'éléments dans une pile peut amener à un débordement. La fameuse pile Overflow.

d. File

Comme les piles, les files sont des structures linéaires dynamiques qui se distinguent par les conditions d'ajout et d'accès aux éléments.

Les files sont fondées sur le principe du « premier arrivé, premier sorti » ; on les dit de type FIFO (*First In, First Out*). C'est le principe de la file d'attente devant un guichet.

Comme dans les piles, ce mode de fonctionnement dit First In First Out (FIFO) est géré via une interface avec quatre opérations principales :

- Push (enqueue) : ajouté un nouvel élément à l'arrière de la file
- Pop (dequeue) : enlever un élément à l'avant de la file
- Front (peek) : retourner l'élément à l'avant de la file
- IsEmpty : retourner true si la file est vide

Evidemment, tout ça en temps constant O (1) comme pour la pile, les files sont également présentes partout en informatique.

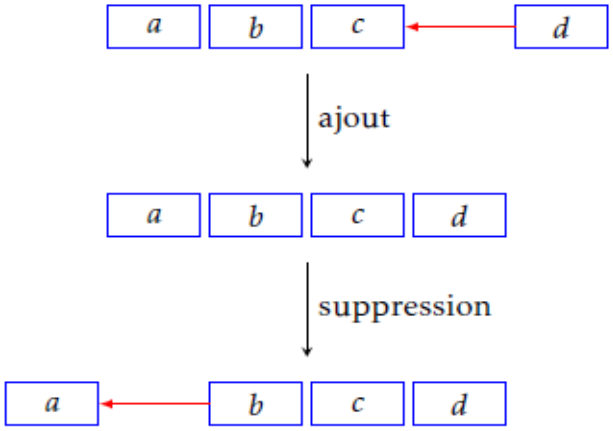


FIGURE 4 Mécanisme d'ajout et de suppression dans une file.

d.1 Avantages

- Permet de gérer de la data en mode FIFO.
- Certaines opérations seront faites en temps constant O (1).

d.2 Inconvénients

- De par la nature FIFO de la file, on ne peut pas accéder directement à un élément au milieu de la file en temps constant O (1) comme avec un tableau.

3. Conclusion

Comme la pile ne permet l'accès qu'à un seul de ses éléments, son usage est limité. Cependant, elle peut être très utile pour supprimer la récursivité d'une fonction. La différence entre la modélisation par liste chaînée et la modélisation par tableau est très faible.

L'inconvénient du tableau est que sa taille est fixée à l'avance, contrairement à la liste chaînée qui n'est limitée que par la taille de la mémoire centrale de l'ordinateur. En contrepartie, la liste chaînée effectue une allocation dynamique de mémoire à chaque ajout d'élément et une libération de mémoire à chaque retrait du sommet de la pile.

Par contre la file est une structure de données très utilisée par les mécanismes d'attente. C'est le cas notamment d'une imprimante en réseau, où les tâches d'impressions arrivent aléatoirement de n'importe quel ordinateur connecté. Les tâches sont placées dans une file d'attente, ce qui permet de les traiter selon leur ordre d'arrivée. Les remarques concernant les

différences la modélisation par liste chaînée et la modélisation par tableau circulaire sont les mêmes que pour la structure de file. Très peu de différences sont constatées.

En résumé, la modélisation par liste chaînée sera un peu plus lente, mais plus souple quant à sa taille.

VI- Structures des données non linéaires

Les structures de données non linéaires n'organisent pas les données de façon consécutive, mais plutôt dans un ordre trié. Pour cela, les éléments peuvent être attachés à plus d'un élément présentant une relation hiérarchique qui implique la relation entre enfant, parent et grand-parent. Dans une structure de données non linéaire, le parcoure de données et l'insertion ou la suppression ne sont pas effectuées de manière séquentielle.

1. Types de structures des données non linéaires

a. Graphe

Un graphe est une collection de nœuds (ou sommets, ou vertices en anglais) dont certains sont reliés entre eux par des branches (ou liens, ou arêtes, ou arcs, ou edges en anglais). Les liens entre les nœuds peuvent être orientés ou non orientés. Dans ce dernier cas les liens sont symétriques.

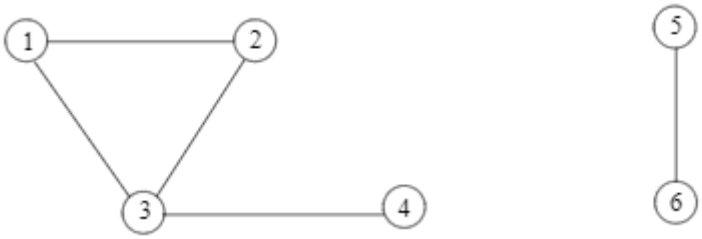
a.1 Graphe non orienté

Un graphe (simple) non orienté G est un couple (S, A), où :

S est un ensemble dont les éléments sont appelés les sommets.

A est un ensemble de paires (non ordonnées) d'éléments de S, ces paires sont appelées arêtes.

- Par exemple :



En termes d'applications, les graphes non orientés peuvent servir à modéliser, entre autres :

- Un réseau ferré ou de métro.
- Un réseau routier à grande échelle : chaque ville est un sommet, chaque route entre deux villes est un arc (en général, elle n'est pas en sens unique).
- Un réseau informatique.

Soit $G = (S, A)$ un graphe non orienté.

- Si $X = \{a, b\} \in A$, on dit que a et b sont voisins.
- On appelle chaîne de G une suite (finie) d'arêtes de G telle que 2 arêtes consécutives dans la suite ont un sommet commun.
- Un circuit est une chaîne dont l'origine et l'extrémité sont confondues.
- On dit que G est connexe si et seulement si, pour toute paire de sommets $\{x, y\}$ de S, il existe une chaîne entre les sommets x et y.
- Quand on parle de connexité pour un graphe orienté, on considère non pas ce graphe mais le graphe non-orienté correspondant.

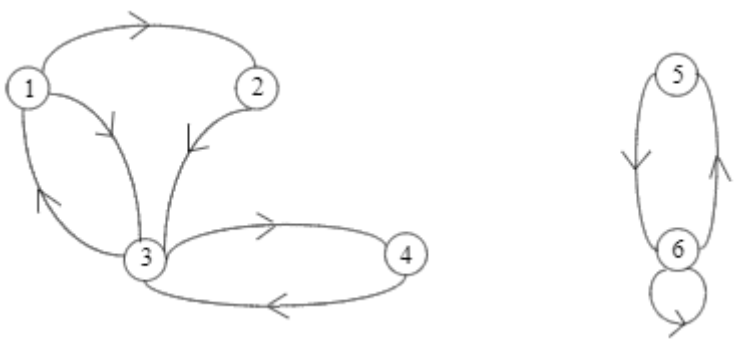
a.2 Graphe orienté

Un graphe (simple) orienté G est un couple (S, A), où :

S est un ensemble dont les éléments sont appelés les sommets.

A est un ensemble de couples (ordonnés) d'éléments de S, ces couples sont appelés arcs.

Par exemple :
Soient S= {1,2,3,4,5,6} et A= {(1;2), (1;3), (2;3), (3;1), (3;4), (4;3), (5;6), (6;5), (6;6)}.
(S, A) est un graphe orienté qui peut être représenté par :



En termes d'applications, les graphes orientés peuvent servir à modéliser, entre autres :

- Un réseau routier à petite échelle : chaque intersection est un sommet, chaque tronçon de rue entre deux intersections est un arc. L'orientation peut alors permettre de gérer les sens interdits.
- Dans le même ordre d'idée : un réseau de bus.
- Un site web : chaque page est un sommet, chaque lien hypertexte est un arc (de la page qui le contient vers la page pointée).

Soit $G = (S, A)$ un graphe orienté. Si $X = (a, b) \in A$, on dit que :

- a est adjacent à b
- a est un prédécesseur de b.
- b est un successeur de a.
- a est l'origine de l'arc X.
- b est l'extrémité de l'arc X.

De plus, si $a = b$, on dit que X est une boucle.

a.3 Parcours de graphes

Pour parcourir tous les sommets d'un graphe qui sont accessibles à partir d'un sommet v donné, il faut que les conditions suivantes doivent être remplies :

Un sommet v′ est accessible à partir de v si :

- soit v′ = v,
- soit v′ est adjacent à v,
- soit v′ est adjacent à un sommet v″ qui est accessible à partir de v

a.4 Différents types de parcours

- En profondeur d'abord (*depth-first*)

- En largeur d'abord (*breadth-first*)

Les algorithmes de parcours de graphe les plus étudiés sont le parcours en profondeur et le parcours en largeur qui ont certains cas d'usage différents.

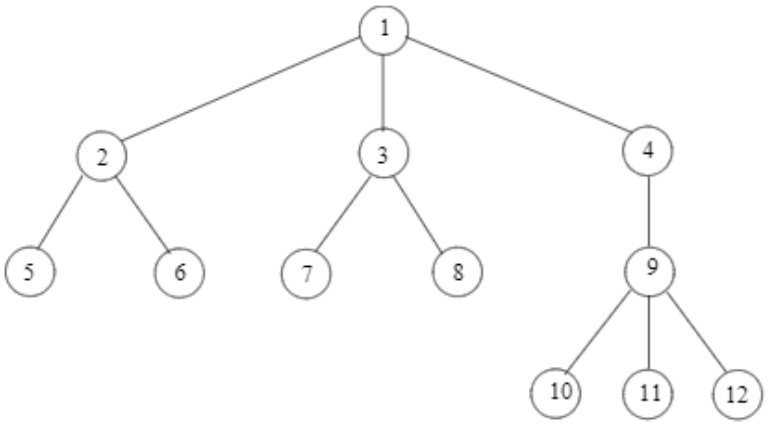
b. Arbre

Un arbre est un ensemble de nœuds (appelés aussi parfois sommets) reliés par des arcs tel que chaque nœud possède exactement un arc pointant vers lui hormis un nœud spécial appelé racine qui n'en possède aucun. La racine est donc un nœud particulier puisqu'il n'a pas de prédécesseur. Les feuilles sont les nœuds sans successeur.

Ainsi pour avoir une meilleure vision de l'arbre et, on le représente avec la racine en haut et, pour chaque nœud, tous ses successeurs au même niveau.

De plus, l'orientation des arcs étant définie, nous les représenterons de la même façon que les arêtes.

- Par exemple :



- Remarque :

- Chaque nœud possède une étiquette, qui est en quelque sorte la « valeur » du nœud. L'étiquette peut être très simple : un nombre entier, un nom, etc. Mais aussi plus complexe : un objet, un ou plusieurs pointeurs, etc.
- Un arbre peut être aussi défini pas récurrence. On peut dire qu'un arbre est une structure formée :
  - d'un élément appelé racine
  - et d'un ensemble d'arbres de même type, aussi appelés sous-arbres.

Les arbres sont rarement utilisés en tant que tels. De nombreux types d'arbres avec une structure plus restrictive existent. Ils sont couramment utilisés en algorithmique car ils permettent des recherches plus efficaces. Les principaux exemples sont des arbres binaires.

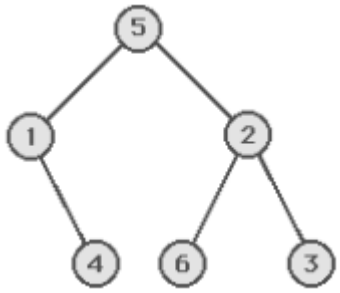
c. Arbre binaire

Un arbre binaire est un arbre dans lequel chaque nœud possède aux plus deux fils.

- Remarque :

• Dans un arbre binaire, le fils unique d'un nœud ne sera jamais placé à l'aplomb de son père, mais toujours dirigé soit vers la gauche soit vers la droite.

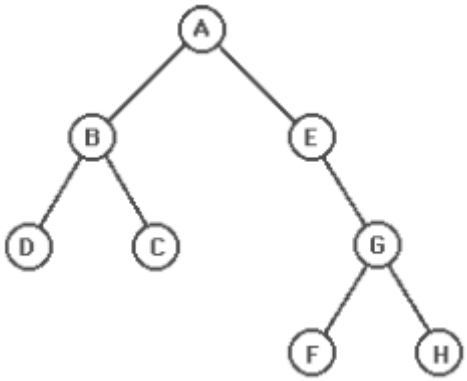
- Par exemple :



On parlera alors de fils droit et de fils gauche ou encore de sous arbre droit et de sous arbre gauche.

c.1 Implantation en mémoire

On considère l'arbre binaire suivant :



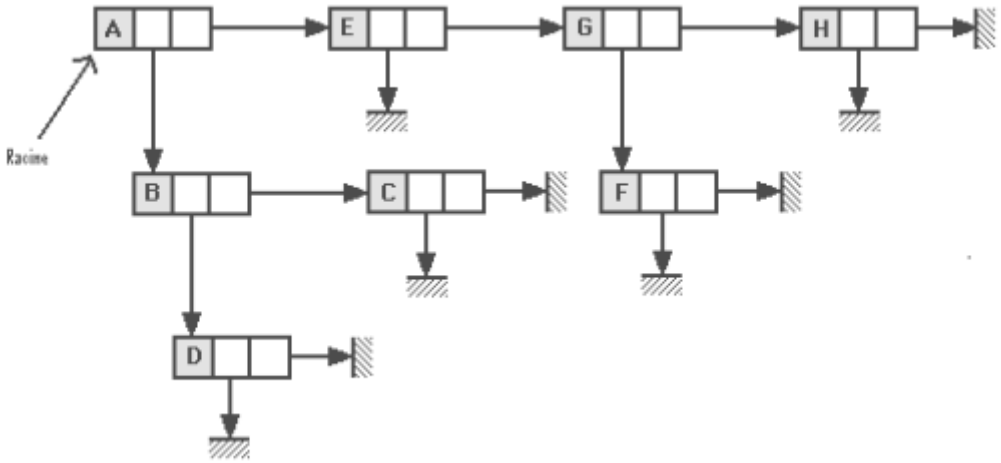
Pour un nœud quelconque de l'arbre binaire, les seules informations nécessaires sont :

- l'information sur son nom.
- les informations concernant ses fils.

On peut alors décider que, pour chaque nœud, on construit un article du type :

Nom du nœud	Adresse du fils gauche	Adresse du fils droit

On obtient un chaînage. Ce qui donne pour notre arbre :



Il pourra également être implanté avec des pointeurs (comme les listes doublement chaînées) mais aussi sous forme de tableau à 2 dimensions :

AdrRac

	1	2	3	4	5	6	7	8
NŒUD	B	A	C	D	E	F	G	H
ADRFILSG	4	1	0	0	0	0	6	0
ADRFILSD	3	5	0	0	7	0	8	0

c.2 Parcours d'un arbre binaire

Lorsque l'on désire connaître les informations contenues dans un arbre (valeurs ou contenus de ses nœuds), cela nécessite de parcourir (ou visiter) celui-ci. Pour cela, il existe différentes méthodes pour effectuer la visite d'un arbre binaire.

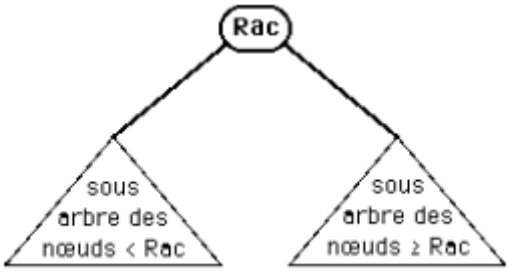
1. Parcours en profondeur
    a. Infixe (en ordre) : chaque nœud est visité après son fils gauche et avant son fils droit.
    b. Préfixe (en préordre) : chaque nœud est visité avant ses fils.
    c. Suffixe (en postordre) : chaque nœud est visité après ses fils.
2. Parcours en largeur : on visite le nœud le plus proche de la racine qui n'a pas déjà été visité. Correspond à une visite de nœud de profondeurs 1, puis 2, …

d. Arbre binaire de recherche (ABR)

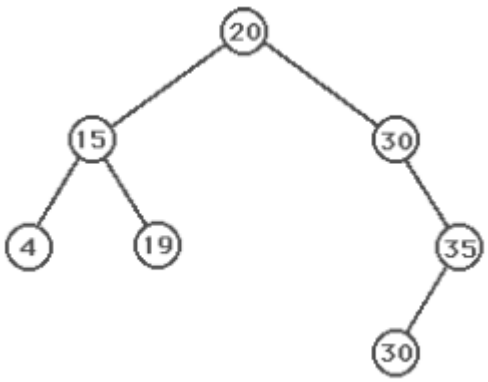
Un arbre binaire de recherche est un arbre binaire possédant récursivement la propriété suivante :

Les nœuds du sous arbre gauche sont tous inférieurs à la racine, elle-même inférieure aux nœuds du sous arbre droit (les éléments égaux pouvant être arbitrairement placés à gauche ou à droite (choix à faire pour un algorithme d'insertion).

Ce qui peut se visualiser de la façon suivante :



- Par exemple :



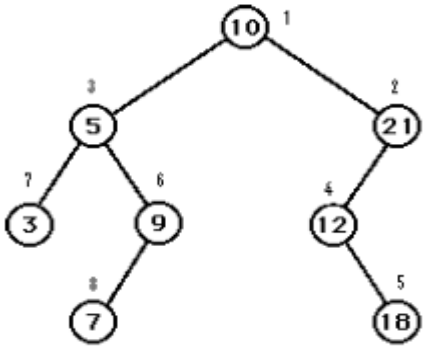
- Remarque :

Ainsi que son nom l'indique, cet arbre est particulièrement utile pour résoudre les problèmes de recherche d'un nœud dans un arbre.

En effet, si *X* est l'élément cherché, il suffit de comparer celui-ci avec la racine pour éliminer tous les candidats se trouvant dans le sous arbre gauche ou dans le sous arbre droit (en fonction du résultat du test avec la racine). C'est évidemment plus intéressant si l'arbre est équilibré.

- Par exemple :

On considère la suite d'éléments suivantes : 10, 21, 5,12, 18, 9, 3, 7. On crée, par l'insertion successive des éléments, l'arbre binaire de recherche suivant :



d.1 Recherche

La recherche dans un arbre binaire d'un nœud ayant une étiquette particulière est un procédé récursif. On commence par examiner la racine. Si l'étiquette de la racine est l'étiquette recherchée, l'algorithme se termine et renvoie la racine. Si l'étiquette cherchée est inférieure, alors elle est dans le sous-arbre gauche, sur lequel on effectue alors récursivement la recherche. De même, si l'étiquette recherchée est strictement supérieure à l'étiquette de la racine, la recherche continue sur le sous-arbre droit. Si on atteint une feuille dont l'étiquette n'est pas celle recherchée, on sait alors que cette étiquette n'est pas dans l'arbre.

d.2 Insertion

L'insertion d'un nœud commence par une recherche : on cherche l'étiquette du nœud à insérer. Si on la trouve, on ne fait rien. Sinon, lorsqu'on ne peut plus descendre dans l'arbre, cela signifie qu'on a trouvé le père. On insère le nœud en comparant son étiquette à celle de

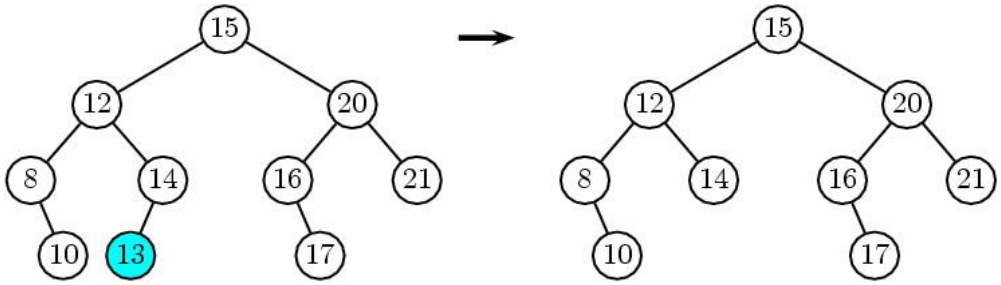
son père : si elle est inférieure, le nouveau nœud sera son fils gauche ; sinon il sera son fils droit. Ainsi, chaque nœud ajouté sera une feuille.

d.3 Suppression

Plusieurs cas sont à considérer, une fois que le nœud à supprimer a été trouvé :

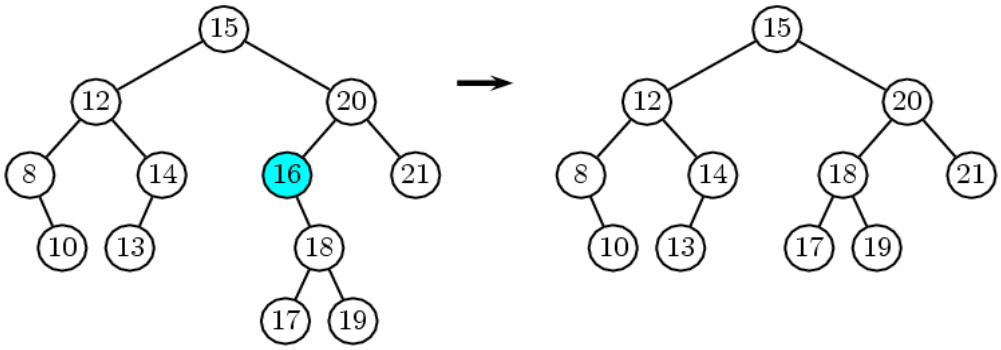
1. Suppression d'une feuille

Il suffit de l'enlever de l'arbre étant donné qu'elle n'a pas de fils.



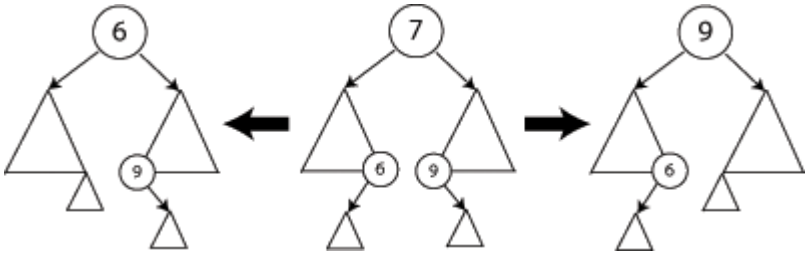
2. Suppression d'un nœud avec un seul fils

On l'enlève de l'arbre et on le remplace par son fils.

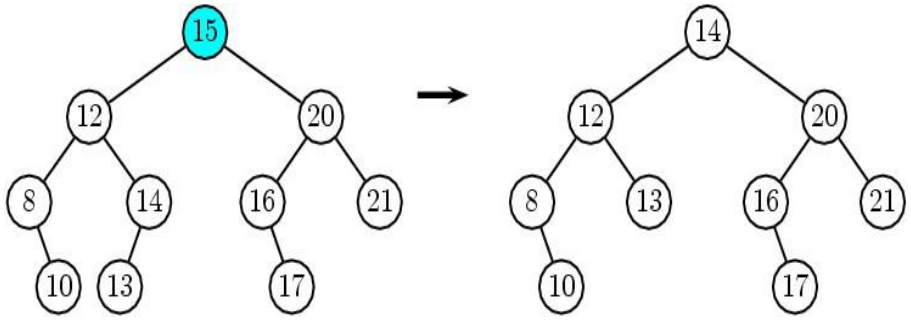


3. Suppression d'un nœud avec deux fils

Supposons que le nœud à supprimer soit appelé *N* (le nœud de valeur 7 dans le schéma ci-dessous). On le remplace alors par son successeur le plus proche, donc le nœud le plus à gauche du sous-arbre droit (ci-après, le nœud de valeur 9) ou son plus proche prédécesseur, donc le nœud le plus à droite du sous-arbre gauche (ci-dessous, le nœud de valeur 6).



Cela permet de garder une structure d'arbre binaire de recherche. Puis on applique à nouveau la procédure de suppression à *N*, qui est maintenant une feuille ou un nœud avec un seul fils.



Pour une implémentation efficace, il est déconseillé d'utiliser uniquement le successeur ou le prédécesseur, car cela contribue à déséquilibrer l'arbre.

VII- Conclusion

Une structure de données est une organisation logique des données permettant de simplifier ou d'accélérer leur traitement.

Dans ce chapitre, nous avons mis l'accent sur l'essentiel des structures de données parmi les plus largement utilisées dans plusieurs domaines et, particulièrement, dans la majorité des systèmes informatiques. Par la suite nous avons détaillé les différents types de structure de données linéaire et non linéaire tout en évoquant leurs avantages et inconvénients.

Chapitre 3
Structure de données distribuées et scalables (SDDS)

I- Introduction

Aujourd'hui avec les progrès technologiques, plusieurs concepts ont évolués et d'autres nouveaux sont apparus, notamment les architectures constituées d'ordinateurs géographiquement distribués. Ces nouveaux systèmes sont désignés par une variété d'appellations comme multi ordinateurs, réseau de stations de travail, grille de calcul, etc.

L'objectif principal pour lequel ces systèmes ont vu le jour est l'exploitation maximale des ressources cumulées de stockage et de traitement ; ce que les structures de données classiques n'ont pu réaliser. Pour cela une nouvelle famille de structure de données a été introduite, dénommée SDDS (structure de données distribuées et scalables), il s'agit de fichiers résidents en mémoire centrale distribuée (DRAM).

Cette nouvelle structure supporte le traitement parallèle et assure un temps d'accès aux données beaucoup plus rapide que celui des fichiers classiques stockés sur disque, en outre elle assure une capacité de stockage potentiellement illimitée.

Un fichier SDDS débute sur un site et peut s'étendre dynamiquement sur plusieurs sites suite à des opérations d'insertion ; pour cela les algorithmes d'adressage d'une SDDS ont été conçus spécifiquement pour être scalables (la scalabilité ici consiste à maintenir les performances d'un système quand le volume de données stockées augmente).

Depuis l'apparition de cette nouvelle structure, plusieurs types de SDDS ont été proposés par Litwin, Neimat et Schneider, les plus connus sont celles basées sur le hachage linéaire (LH*) et celles utilisant le partitionnement par intervalle (RP*).

Dans ce qui suit, nous expliquons ces nouvelles structures de données tout en invoquant leurs deux grandes classes (les LH* & les RP*).

II- Les multi ordinateurs

Des recherches avancées sont menées pour mieux exploiter la puissance de calcul d'un ensemble d'ordinateurs interconnectés à travers des réseaux à haut débit (>10MBits).

De telles configurations existent déjà dans plusieurs organisations. Des termes sont apparus pour désigner les machines organisées de la sorte : multi-ordinateurs, Réseau de Stations de Travail (« *Network of Workstations* ») ou, plus récemment, de « *Peer-to-Peer Computing* » ou de « *Grid Computing* ». Les capacités cumulées de traitement parallèle et de stockage d'un multi-ordinateur sont impressionnantes et même supérieures aux performances

des gros systèmes. De telles configurations sont évolutives et exploitent au mieux les progrès constants au niveau du matériel.

Les multi-ordinateurs se caractérisent par la manière dont leurs composants de base (la mémoire principale, le processeur (CPU) et les mémoires secondaires) sont interconnectés.

Ainsi les trois architectures, décrites par la suite, ont été proposées et implantées ces dernières années.

- **Architecture à mémoire partagée**

Dans un multi-ordinateur à mémoire partagée ou en grappe (Shared-memory), tous les processeurs accèdent à une grande mémoire commune. L'accès aux disques est également partagé. Ainsi, chaque processeur a un accès direct à toutes les portions de la mémoire ou des disques. Tout le système est en général regroupé dans une même machine. L'équilibrage de charge ou la synchronisation des processeurs se réalise facilement à travers la mémoire partagée.

L'accès concurrent à la mémoire peut devenir un goulot d'étranglement quand le nombre de processeurs devient important. Egalement, les accès fréquents aux disques entraînent un flux de données trop important sur le réseau. Cette surcharge du réseau ralentit tout le système.

La Figure 5 schématise un multi-ordinateur à mémoire partagée avec les lettres *P* qui symbolisent les processeurs, *M* la mémoire centrale et *D* les disques.

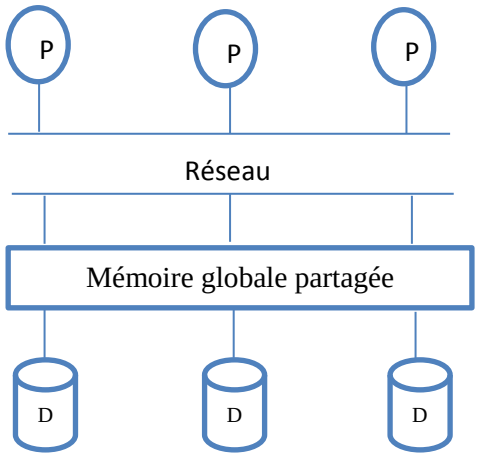


FIGURE 5 Multi-ordinateur avec mémoire partagée

- **Architecture à disques partagés**

Dans un multi-ordinateur à disques partagés (Shared-disk), chaque processeur dispose d'un accès direct à une mémoire privée. Seul l'accès aux disques est partagé. Cette architecture élimine les interférences des processeurs sur la mémoire principale et réduit la congestion du réseau. La Figure 6 représente un multi-ordinateur à disques partagés.

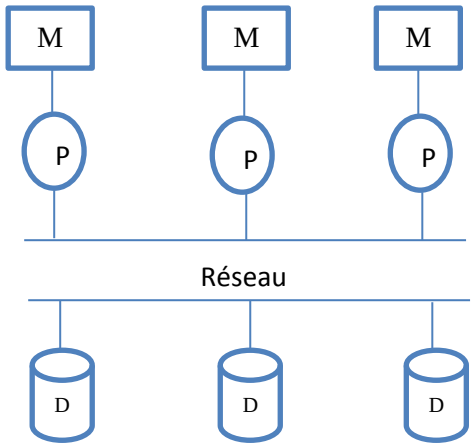


FIGURE 6 Multi-ordinateur à disques partagés

- **Architecture sans mémoire partagée**

Dans un multi-ordinateur sans mémoire partagée (Shared-nothing), chaque processeur dispose d'un accès exclusif à la mémoire et aux disques qui lui sont reliés. L'ensemble (processeur, mémoire et disque) est appelé nœud. Aucun processeur ne peut accéder directement à une mémoire ou un disque sur un poste distant. L'échange d'informations entre deux nœuds se fait à travers une connexion réseau. La Figure 7 présente un multi-ordinateur sans mémoire partagée.

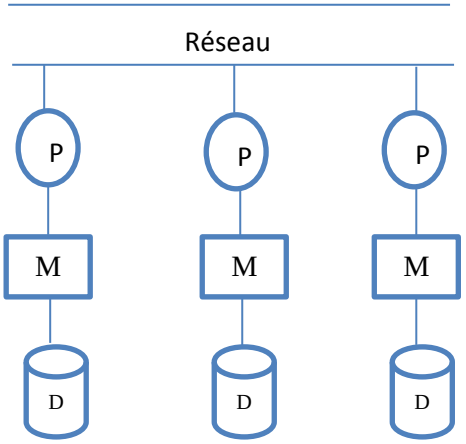


FIGURE 7 Multi-ordinateur sans mémoire partagée

En réduisant les ressources partagées, cette architecture élimine les interférences entre processeurs. Un multi-ordinateur sans mémoire partagée supporte la monté en charge mieux que les deux autres architectures. L'équilibrage de charge entre les différents nœuds est plus difficile à mettre en œuvre dans les multi-ordinateurs sans mémoire partagée.

III- Les structures de données distribuées et scalables (SDDS)

Les SDDS sont une nouvelle classe de structure de données conçues spécialement pour les multi ordinateurs. Elles stockent les données sur des sites appelés serveurs qui sont accédés par d'autres sites appelés clients.

Un fichier SDDS est manipulé par des sites clients, chaque client possède son propre schéma d'adressage appelé "image" qui lui permet d'accéder au bon serveur.

Le fichier SDDS débute sur un seul site serveur et peut être étendu par insertion à un nombre quelconque de sites, ceci rend sa capacité de stockage potentiellement illimitée.

1. Règles de base des SDDS

Un fichier SDDS est stocké sur des sites désignés *serveurs*. Sur chaque *serveur*, les enregistrements sont stockés sur un espace mémoire appelé *case*. Un paramètre important d'une case est sa capacité qui détermine le nombre maximum d'enregistrements qu'elle peut contenir.

Un enregistrement comporte un champ-clé et des champs non-clés. Le champ clé identifie l'enregistrement de manière unique sur l'ensemble des serveurs. Les requêtes sont formulées à partir de sites autonomes désignés *clients*. Il n'y a pas de répertoire central d'accès. Chaque client dispose de sa propre image de la structure du fichier. Dès qu'un serveur atteint sa capacité maximale, il transfère la moitié de ses enregistrements vers un nouveau serveur. Les mises à jour de la structure d'une SDDS ne sont pas envoyées aux clients d'une manière synchrone. Un client peut faire des erreurs d'adressage.

Chaque serveur vérifie l'adresse de la requête et l'achemine vers un autre serveur si une erreur est détectée. Le serveur adéquat envoie alors un message correctif
(*IAM* - Image Adjustment Message) au client ayant commis l'erreur d'adressage. Ce dernier ajuste son image pour ne plus faire la même erreur. Les *IAMs* font converger l'image d'un client vers celle réelle.

La fragmentation des données sur les sites de stockage se fait par hachage ou par intervalle. RP* réalise la fragmentation dynamique par intervalle (Range Partitioning) et

supporte, comme les arbres-B des requêtes à intervalle ou un parcours ordonné du fichier par rapport à la clé.

D'où, les SDDS sont régies par un ensemble de règles qui sont :

- L'absence de répertoire central d'accès, ce qui favorise l'élimination des goulots d'étranglement, et augmente les performances d'accès du système.
- Les fichiers SDDS s'étendent sur plusieurs serveurs par des opérations d'éclatement et se rétrécissent par des fusions de cases (une case est une mémoire d'un serveur stockant les données de la SDDS) et ceci est fait d'une façon dynamique et transparente pour l'application.
- Chaque client gère sa propre image du fichier, car les mises à jour de la structure d'une SDDS ne sont pas envoyées aux clients d'une façon synchrone. Un client peut avoir une image incorrecte, ce qui provoquerait une erreur d'adressage de sa part.
- Un serveur est capable de détecter les erreurs d'adressage produites, et d'acheminer les requêtes - provenant du client - vers le serveur adéquat, ainsi ce dernier envoie un message d'ajustement d'image (IAM) au client qui ajuste son image pour éviter de refaire la même erreur une autre fois.

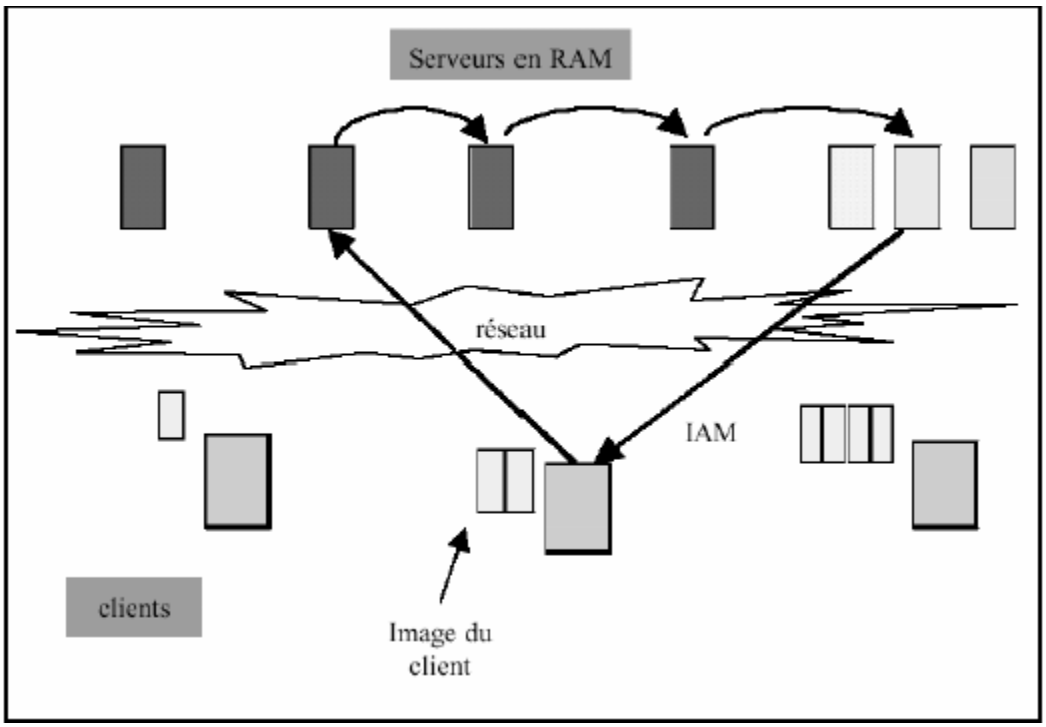


FIGURE 8 Schéma d'une structure de données distribuées

2. Les contraintes des SDDS

Dr. Litwin a indiqué deux contraintes :

- ✓ Si une SDDS n'évolue plus, alors les messages d'ajustement d'image (IAM- Image Adjustment Message) font converger toute image d'un client vers celle actuelle.
- ✓ L'ensemble des renvois à la suite d'une erreur d'adressage ne se fait qu'en quelques messages.

3. Les caractéristiques des SDDS

Les SDDS disposent d'un ensemble de caractéristiques parmi lesquelles :

A- La scalabilité

Elle consiste à prendre en charge n'importe quelle quantité de données et à maintenir les performances quand le volume de données stockées varie, c'est-à-dire il n'y a pas de limites théoriques de capacité taille, pas de réorganisation totale de la structure et en pratique le temps d'accès aux données est plus ou moins constant.

B- La distribution

Souvent lorsque la quantité des données augmente, un seul site devient incapable de la retenir et de la manipuler quelles que soient ses capacités de stockage et de calcul, alors, il devient nécessaire dans ce cas de distribuer ces données sur d'autres sites. Généralement, deux types de distributions peuvent être utilisés, le premier consiste en la distribution par hachage et le deuxième consiste en la distribution par intervalle. (Ils seront expliqués ci-après).

C- La disponibilité

Consiste à assurer l'existence des données quelles que soient les conditions (même en cas de panne d'un ou de plusieurs sites serveurs).

4. Passage des structures de données classiques aux SDDS

Divers projets ont été effectués dans le domaine des structures de données distribuées, ils consistent tous à chercher une meilleure solution pour assurer la disponibilité et la scalabilité des données, pour cela, les chercheurs se sont basés sur les structures de données classiques et ont essayé de les adapter aux environnements distribués. C'est le cas de toutes les SDDS proposées jusqu'à présent.

Dans ce qui suit, nous essayons d'indiquer l'ensemble des structures classiques et les SDDS qui en découlent.

a. Structures basées sur le hachage

Plusieurs chercheurs considèrent que les SDDS constituent une extension des schémas de hachage classiques sur les multi ordinateurs comme par exemple les SDDS : LH*, DDH*, EH*, IH* et leurs dérivés qui utilisent dans leur principes les structures de données classiques basées sur le hachage linéaire, dynamique, extensible ou autres.

Exemple d'une structure classique basée-hachage et de la SDDS correspondante

DLH est une distribution de LH sur une machine parallèle à mémoire partagée, elle consiste à distribuer les paramètres i et n de fichier LH sur tous les processeurs de la machine, ce qui entraîne des erreurs d'adressage. Son inconvénient principal est que le nombre de processeurs sur une machine parallèle est limité (32 processeurs au maximum), donc DLH n'est pas une SDDS car elle ne satisfait pas la propriété de scalabilité.

Pour remédier à ce problème, une SDDS été proposée, la LH*, qui ne diffère pas beaucoup dans ses principes généraux de ceux de DLH.

Donc, un fichier LH* est distribué sur plusieurs serveurs LH*, il est accédé par des applications fonctionnant sur des machines autonomes dites clients LH*. Le fichier est constitué d'enregistrements regroupés en M cases de taille b résidantes en mémoires principales, à raison d'une case par serveur.

L'adresse physique en réseau de chaque case est sauvegardée dans des tables d'allocation physiques disponibles chez les clients et chez les serveurs. Une trace de niveau j (indice de la fonction de hachage) est sauvegardée dans chaque serveur et un serveur particulier appelé coordinateur d'éclatement garde une trace des valeurs réelles de i et n de fichier.

b. Structures basées sur les arbres (structures ordonnées)

A leur tour, les SDDS basées sur les arbres étendent les structures ordonnées traditionnelles aux environnements distribués comme par exemple les SDDS : RP*, DRT, DRT*n, arbres B+ distribué, K-RP*, K-DRT et leurs dérivés qui exploitent les principes des B-arbres, arbres B+, K-d-arbres ainsi que d'autres organisations classiques.

Exemple d'une structure classique basée-arbre et de la SDDS correspondante

Dans le domaine de stockage des données, un arbre est une structure de données hiérarchique et généralement dynamique, représentée par un ensemble non vide de nœuds (sommets) et des arrêtes assujettis à certaines conditions.

Les arbres ordonnés sont des arbres dont les données associées aux feuilles (nœud externe : qui n'a pas de successeur) sont triées en parcourant toutes les feuilles de gauche à droite. Les

principaux avantages des arbres ordonnés se résument en des recherches rapides non séquentielles et des insertions ou suppressions ne provoquant pas des décalages.

Le principal inconvénient des arbres ordonnés est qu'ils deviennent déséquilibrés après plusieurs insertions ou suppressions des feuilles d'une manière non uniforme, les accès deviennent ainsi presque séquentiels.

Pour remédier à ce problème, les arbres B (B-arbres) été proposés, ils permettent d'assurer l'équilibre de l'arbre au détriment de la perte d'une partie de l'espace de stockage.

Plusieurs schémas de structures de fichiers distribuées basés sur les arbres ont été proposés, range-partitioning (RP) a été celui qui a connu le plus de succès, mais il reste limité par sa non scalabilité. Pour pallier à ce problème, une SDDS appelée RP* a été proposée en généralisant le concept des arbres B aux environnements distribués.

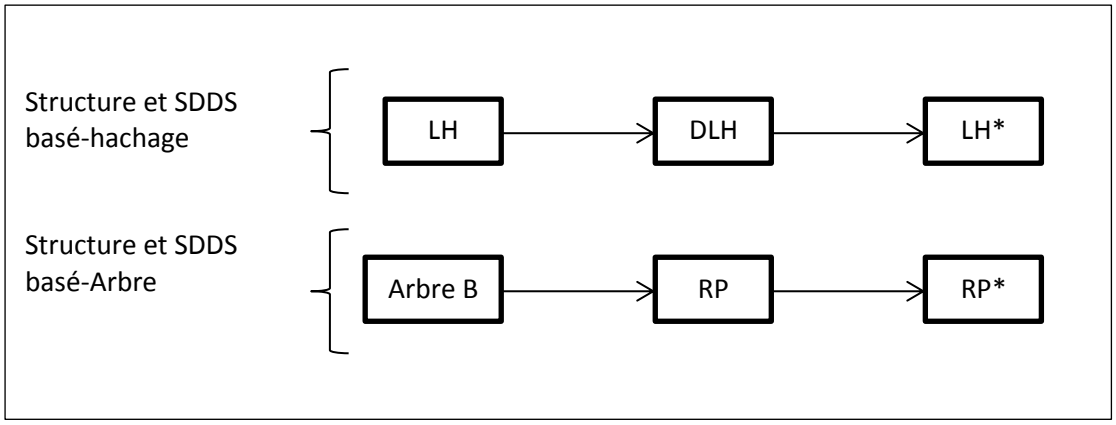


FIGURE 9 Illustration des organisations classiques et des SDDS qui en découlent

5. Classification des SDDS

Les SDDS sont regroupées en deux grandes catégories :

- SDDS basées sur les algorithmes de hachage comme les LH*
- SDDS pour les fichiers ordonnés (par rapport à la clé primaire des enregistrements) comme les RP*.

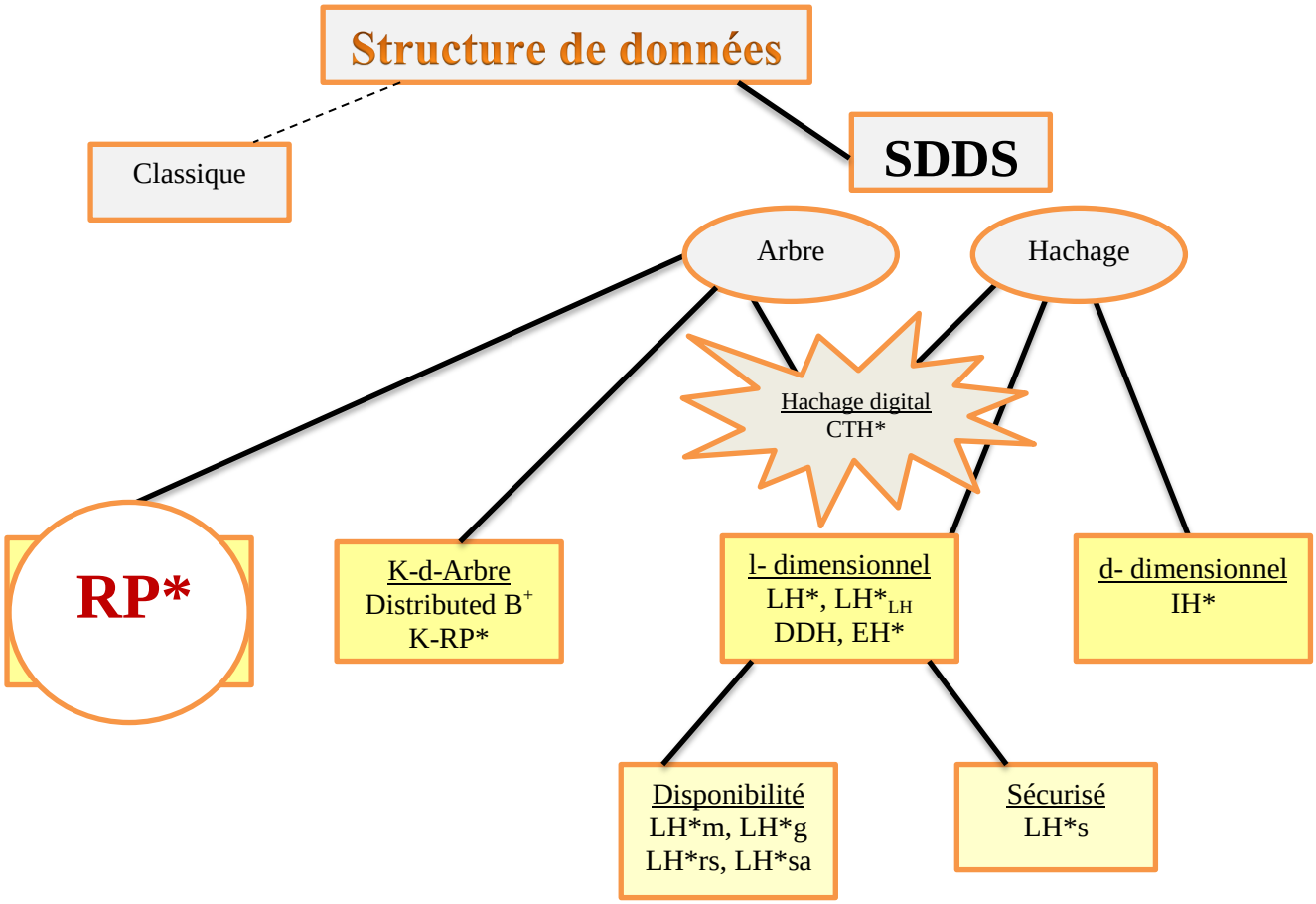


FIGURE 10 Classification des SDDS

A. Les SDDS basées sur le hachage

Les SDDS basées sur le hachage constituent une extension des schémas de hachage classique sur les multi ordinateurs, on trouve parmi les SDDS de cette catégorie les SDDS suivantes :

- LH* (Distributed Linear Hashing) et ses dérivées.
- DDH* (Distributed Dynamique Hashing).
- EH* (Distributed Extensible Hashing).
- IH* (Distributed Interpolation-based Enashing), … et bien d'autres.

a. Présentation des SDDS LH*

La SDDS LH* est la première SDDS qui a été proposée par Litwin, Neimat et Schneider, c'est la plus connue des SDDS existantes, elle est basée sur le hachage linéaire. Plusieurs variantes ont été proposées dans ce type de SDDS (exemple : LH*LH, LH*m, LH*g, LH*s, LH*sa, LH*rs), elles visent surtout à garantir la haute disponibilité des données et elles font appel à des principes de récupération de données, tel que : le miroitage (miroring), la fragmentation (striping), le groupement d'enregistrements (grouping) et le Reed-Solomon Code.

b. Les principes des SDDS LH*

Un fichier LH* est un ensemble d'enregistrements identifié par une clé primaire, ces enregistrements sont stockés dans des paquets (ou cases) numérotés qui ont une capacité **b** (b>>1), le fichier commence avec le paquet 0, il peut s'étendre avec des insertions ou se rétréci avec des fusions de cases.

Les éclatements sont déclenchés par des débordements de paquets, chaque éclatement déplace presque la moitié des enregistrements vers un autre serveur.

Dans LH*, le paquet débordé rapporte le débordement à un site dédié appelé coordinateur qui initie la procédure d'éclatement de la case n. pour effectuer l'adressage d'un enregistrement, une fonction de hachage **hi (C) = C mod N * 2** $^{\mathbf{i}}$, i=0,1,… est appliquée sur la clé de cet enregistrement (comme le cas du hachage linéaire classique), avec :
hi : la fonction de hachage.
i = 0,1,2,3,……
n : le nombre initial des paquets.

Algorithme d'adressage d'un enregistrement de clé C

```
a : =hi(C);
si a<n alors a : = hi+1(C);
```

L'éclatement résulte du remplacement de la fonction (hi) actuellement utilisée pour le paquet n par la fonction (hi+1), la valeur i est le niveau du paquet.

A n'importe quel moment, un fichier LH* peut avoir des paquets avec seulement un niveau i ou un niveau i+1.

Algorithme de mise à jour du couple (i, n) après l'éclatement d'une case

```
n :=n+1
Si n>=2^i alors n :=0; i :=i+1;
Fin si
```

Inversement à l'éclatement dû au débordement du fichier, il existe la fusion qui résulte de la diminution du taux de remplissage du fichier (après des suppressions par exemple) et qui provoque la mise à jour du couple (i, n) selon l'algorithme suivant :

Algorithme de mise à jour du couple (i, n) après la fusion de deux cases

n :=n-1;
Si *n<0* alors *i :=i-1; n :=2* ^{i} *-1;*
Fin si

Comme toute SDDS, un client LH* possède sa propre image de fichier, il possède deux valeurs i' et n' qui sont utilisées pour le calcul de l'adresse a' de l'enregistrement identifié par la clé C en utilisant son propre algorithme d'adressage :

Algorithme d'adressage exécuté par le client

a' := hi'(C);
si *(a'<n')* alors
a' := hi'+1(C) ;

Le site coordinateur maintient les valeurs de i et n ; le client possède une copie de ces valeurs qui seront changées dès qu'un éclatement aura lieu.

Lorsque le client fait une erreur d'adressage, il reçoit un message d'ajustement d'image de la part du serveur qui reçoit ses requêtes.

Algorithme d'ajustement de l'image du client

Si *j > i'* alors *i'* : = *j* – 1;
n' : = *a* +1 ;
Si *n'* ≥2^*i'* alors
n' = 0;
i' : = *i'* +1 ;

La requête envoyée par le client est vérifiée par la case si elle lui est destinée ou non, sinon elle la traite et l'envoi vers une autre case LH*, le schéma LH* garantit qu'une requête atteint le bon serveur au maximum au bout de deux renvois, et voici l'algorithme de test et de renvoi utilisé dans cette opération :

Algorithme de test et de renvoi exécuté par la case LH*

a' := hj (c) ;
si *a' = a alors* accepter *c ;*
sinon *a'' := hj* - 1 *(c) ;*
si *a'' > a* et *a'' < a'* alors *a' := a'' ;*
envoyer *c* au paquet *a' ;*

B. Les SDDS basées sur la distribution par intervalle

Une SDDS basée sur la distribution par intervalle répartit l'espace des clés par intervalles de cases. Dans cette organisation, un intervalle unique de l'espace des clés est associé à chaque case du fichier SDDS.

Parmi les SDDS connues de cette distribution, on trouve celles qui se basent sur les arbres 1-d comme :

- RP* (Distributed Rang Partitionning) et ses variétés.
- DRT (Distributed Relaxed Trees).
- DRT* (Distributed Random Tree).…

Et celles qui se basent sur les arbres k-d comme :

- K-RP* et ses variétés.
- K-DRT.
- Arbre B+ distribué (Distributed B+ tree). …
- Et bien d'autres.

Nous nous limiterons à l'étude des RP* et ses variantes, que nous détaillerons dans le prochain chapitre IV.

IV- Conclusion

Dans ce chapitre on a expliqué les SDDS d'une manière simple, on a aussi pris soins de discerner les SDDS basées-hachage et les SDDS basées-arbre. Pour en savoir, les types ou les variétés des SDDS sont en progression continue, et ceci afin de faciliter la manipulation des données et de tirer profit des ressources des multi ordinateurs en termes de stockage, de sécurité et de vitesse de traitement.

Chapitre 4

Partitionnement par intervalles ou range partitioning RP* et RP*c

I- Introduction

Dans un premier temps, des SDDS basées sur le hachage ont été proposées tels que LH* et ses variantes, DDH et EH*, celles-ci constituent une généralisation des algorithmes de hachage classiques. Cependant, dans le cas où le fichier doit supporter des requêtes par intervalles ou un parcours transversal de tous les enregistrements, les structures de données ordonnées telles que les arbres B offrent de bonnes performances par rapport aux techniques de hachage qui ne préservent pas l'ordre. Sur cette base, une autre famille de SDDS a été proposée pour construire des fichiers distribués qui préservent l'ordre des enregistrements. Citons à cet égard : RP* que nous allons détaillée dans ce qui suit.

II- RP*

1. Les SDDS basées sur la distribution par intervalles

Une SDDS basée sur la distribution par intervalle répartit l'espace des clés par intervalles de cases. Dans cette organisation, un intervalle unique de l'espace des clés est associé à chaque case du fichier SDDS.

Parmi les SDDS connues de cette distribution, on trouve celles qui se basent sur les arbres 1-d comme: - RP* (Distributed Range Partitioning) et ses variétés.

- DRT (Distributed Relaxed Trees).
- DRT* (Distributed Random Tree) …

Et celles qui se basent sur les arbres k-d comme:

- K-RP* et ses variétés.
- K-DRT.
- Arbre B+ distribué (Distributed B+ tree), et bien d'autres.

2. Présentation des SDDS RP*

Comme toute SDDS, les SDDS RP* sont logées sur des serveurs qui peuvent être accédés par des sites clients en utilisant des requêtes de recherche d'une clé ou des requêtes par intervalles, selon la technique d'adressage utilisée au niveau du client, on trouve trois variétés des SDDS RP* qui sont : les RP*n, les RP*c et les RP*s. on trouve aussi une SDDS à haute disponibilité dénommée RP*HA.

3. Principes des SDDS RP*

Les SDDS RP* constituent une généralisation des algorithmes de hachage. Leur but est d'offrir un fichier distribué qui préserve l'ordre des enregistrements. Un fichier peut débuter sur une machine et s'étendre par des insertions sur un nombre quelconque de machines. Le fichier peut être manipulé par plusieurs clients répartis sur le multi ordinateur et supporte les requêtes parallèles.

4. Structure et évolution du fichier RP*

Un fichier RP* est constitué d'enregistrements identifiés chacun par une clé primaire. Un enregistrement comporte un champ-clé et des champs non-clé. Le champ clé identifie l'enregistrement. L'espace des champs-clé est totalement ordonné. Les enregistrements sont groupés en paquets appelés buckets et ils sont stockés sur les serveurs. Un bucket ou case SDDS a une capacité de b enregistrements. Les enregistrements sont répartis dans les différentes cases suivant la valeur de leur champ-clé. Logiquement, les enregistrements sont rangés suivant l'ordre ascendant des clés dans une case.

Chaque case est munie d'un en-tête contenant deux valeurs : une clé minimale notée λ et une clé maximale notéeΛ, l'intervalle $[\lambda,\Lambda]$ est appelé portée ou rang de la case. La structure d'une case SDDS RP* est illustrée par la figure 11 ci-après. Un enregistrement de clé C appartient à une case si $\lambda < C \leq \Lambda$ c'est-à-dire $C \in]\lambda,\Lambda]$.

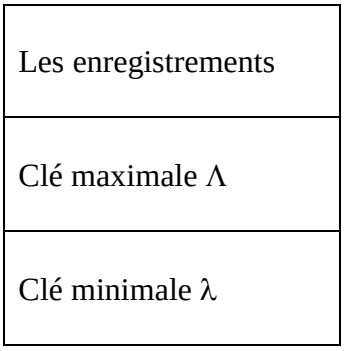


FIGURE 11 Structure d'une case SDDS RP*

Un fichier RP* comporte une case unique notée case 0 avec $\lambda = -\infty$ et $\Lambda = +\infty$.
Initialement, toutes les insertions d'enregistrements vont à la case 0. Cette case éclatera en deux dès que son contenu aura atteint la capacité permise pour une case. Cet éclatement se fait après l'identification de la clé du milieu, il en résulte la création d'une nouvelle case

numérotée case 1 qui recevra la moitié des enregistrements de la case 0. La case 1 est d'une portée de [C, Λ] et la portée de la case 0 deviendra [λ, C]; avec C la clé de l'enregistrement en milieu.

La figure 12 suivante illustre l'évolution d'un fichier SDDS RP* avec une capacité des cases fixée à 4. Les enregistrements sont symbolisés par leur champ clé.

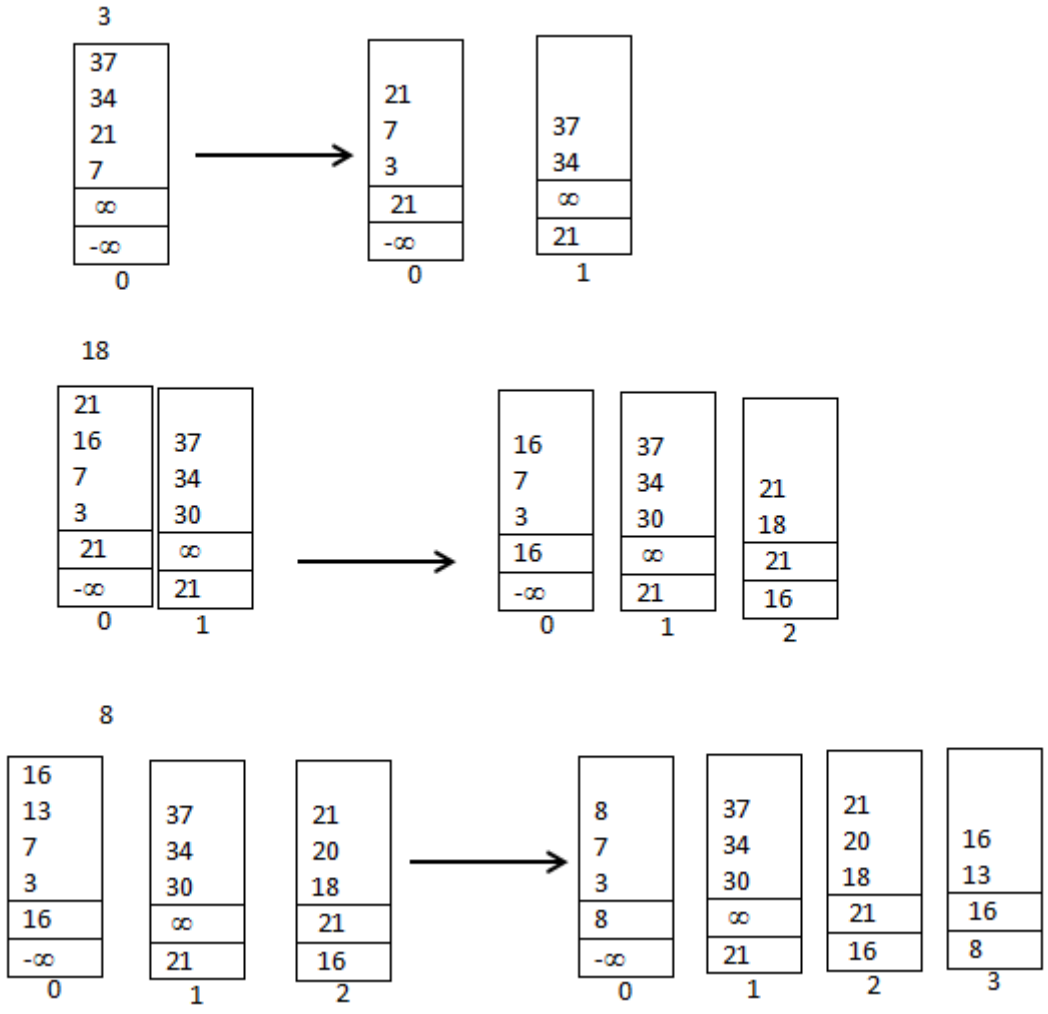


FIGURE 12 Evolution du fichier RP* à la suite d'insertions d'enregistrements

- **Algorithme d'éclatement d'une case**

Si le nombre d'enregistrements d'une case i atteint la capacité b, alors toute tentative d'insertion dans cette case entraîne son éclatement. L'éclatement consiste en trois opérations : création d'une nouvelle case j, migration de la moitié des enregistrements de la case i vers la case j, modification de la portée de la case i et détermination de celle de la case j. Si C_m est la clé de l'enregistrement du milieu de la case en débordement, alors on obtient :

Case i[λ,Λ] => Case i[λ, C_m] et Case j[C_m, Λ].

Ainsi, chaque éclatement d'une case affine le partitionnement de l'espace des clés.

a. Les étapes de l'algorithme

1) Déterminer Cm la clé de l'enregistrement du milieu de la case en débordement.

2) Créer une nouvelle case j.

3) Déterminer l'en-tête de la case J

$\lambda j := Cm$;

$\Lambda j := \Lambda$;

Copier dans la case j les enregistrements de la case i dont la clé C > Cm.

4) Modifier l'entête de la case i

$\lambda i := \lambda$

$\Lambda i := Cm$

Effacer les enregistrements de la case i dont la clé C > Cm.

b. Opérations sur le fichier

Le fichier est manipulé par des requêtes émises à partir des sites clients. Une requête peut porter sur un enregistrement, un intervalle d'enregistrements ou un ensemble d'enregistrements qui vérifient une condition sur un champ non-clé. Une requête du type recherche, modification ou suppression d'un enregistrement est dite simple. Une requête par intervalle correspond à la recherche des enregistrements dont les clés sont comprises dans l'intervalle de recherche. Une recherche est dite générale quand la condition porte sur un champ non-clé. Un client peut aussi effectuer un parcours transversal du fichier qui consiste à examiner séquentiellement tous les enregistrements suivant l'ordre croissant ou décroissant des clés.

5. Variantes de RP*

Les SDDS de type RP* se distinguent selon la présence ou l'absence d'index et la manière de communication entre les clients et les serveurs. Généralement, on distingue deux types de messages de communication, le multicast qui est destiné à un ensemble de machines appartenant à un même groupe sur un réseau donné, et l'unicast qui est destiné à une seule machine du réseau.

a. RP*n

Décrit un partitionnement des enregistrements par intervalle définit sur l'espace des clés, dans leurs communications, les clients émis des messages multicasts aux serveurs qui répondent avec des messages unicasts.

b. RP*c

Ajoute à RP* une image au niveau de chaque client, ainsi le mode de communication entre les clients et les serveurs sera basé sur les messages unicasts.
Les messages multicasts ne seront utilisés que pour les redirections (cas d'erreur).

c. RP*s

Ajoute un index réparti au niveau des serveurs, ce qui permet l'utilisation des messages unicasts lors des redirections.

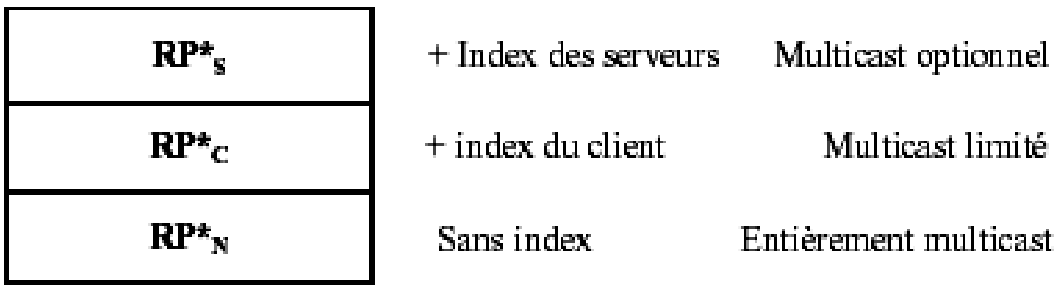


FIGURE 13 Structure de la famille des SDDS RP*

d. RP*ha

Combine entre la récupération efficace des enregistrements et le parcours ordonné avec la possibilité de tolérance aux pannes, ceci est assuré par la réplication de chaque intervalle de l'espace des clés sur deux serveurs. Il y'a aussi un hôte dédié qui se comporte comme un

coordinateur d'éclatement et de recouvrement, il connaît toutes les organisations courantes du fichier.

L'éclatement d'un paquet coûte cinq messages, seulement un parmi eux transporte un grand volume de données, tandis que le recouvrement d'un paquet inclut un message multicast et cinq messages unicast où deux parmi eux transportent les données.

III- RP*C

1. Les Algorithmes RP*C

L'algorithme RP*C est une reprise de RP*N auquel on ajoute une image du fichier au niveau des sites clients pour réduire l'usage du multicast. L'image du fichier est une collection d'intervalles et d'adresses de sites Serveurs qui traduit la répartition des enregistrements sur les cases et les serveurs qui les hébergent. Un client peut faire une erreur d'adressage par suite d'une image inadéquate de la structure du fichier SDDS. Dans ce cas, le serveur qui a reçu la requête l'achemine par multicast vers les autres cases. Le serveur adéquat envoie alors un message correctif (IAM) au client. Le client ajuste son image pour ne plus faire la même erreur. Les IAM qui sont encapsulés dans les messages de réponse font converger toute image d'un client vers celle actuelle.

2. Structure de l'image du client

RP*c crée une image du fichier sur chaque client pour réduire l'usage de message Multicast. Cette image n'est qu'une collection des intervalles et des adresses des différentes cases déjà accédées.

Le rôle de l'image est de permettre à chaque client d'utiliser un message unicast lorsqu'il envoie une requête simple à une case déjà référencée. Le multicast est utilisé dans le cas contraire.

L'image du client est une table dynamique *T [0, 1, ...]*. Chaque élément *T[i]* de cette table définit l'adresse d'une case et son intervalle. Logiquement, la table T est une liste ordonnée de couples *T[i] = (A, C)* avec :

- *A* : Adresse d'une case du fichier SDDS

On pose *A* = * (une adresse multicast) si elle correspond à une adresse inconnue.

- *C :* Clé maximale Λ(A) que peut contenir la case A.

Initialement *T = [(0,∞)]*, elle évolue en fonction des messages correctifs (IAM) reçus qui entraînent l'insertion ou la mise à jour de couples.

3. Envoi d'une requête RP*C par un Client

Une requête sur un enregistrement de clé *c* est exécutée de la manière suivante :

Le client parcourt les couples $t = (A, C)$ de son image et recherche le premier dont la clé est immédiatement supérieure à la clé c.

- Si $A(t) \neq *$ alors envoyer une requête unicast au *bucket* d'adresse *A*.
- Sinon, envoyer une requête en multicast.

4. Traitement d'une requête RP*C par un serveur

Chaque serveur qui reçoit une requête vérifie si la clé qui y est contenue appartient à sa portée

1) Si la clé n'appartient pas à sa portée :

Si le message est multicast, alors sans suite ;

Si le message est unicast, alors renvoyer la requête plus l'adresse et la portée de la case en multicast aux autres serveurs : il s'agit de la redirection d'une requête.

2) Si la clé appartient à sa portée :

Si c'est un message redirigé, alors envoyer la réponse, l'adresse et la portée de la case intermédiaire, l'adresse et la portée de la case au client.

Sinon envoyer la réponse plus l'adresse et la portée de la case au client.

5. Ajustement de l'image du client

La réponse à une requête contient un champ IAM qui permet au client de corriger son image du fichier (Figure 14 ci-dessous). L'IAM se présente sous forme d'un ou de deux triplets (λ, a, Λ) où $[\lambda, \Lambda]$ est la portée de la case serveur qui a traité la requête et « a » son adresse.

L'ajustement de l'image du client se fait donc de manière asynchrone suivant l'algorithme ci-dessous : *(a)* S'il n'existe pas un élément *t* appartenant à *T* avec $C(t) = \lambda$ et $\lambda \neq -\infty$

alors insérer $(*, \lambda)$ dans *T*.

(b) S'il existe un élément *t* appartenant à *T* avec $C(t) > \Lambda$

alors : si $C(t) = +\infty$ alors $t = (a, \Lambda)$ et ajouter $(*, +\infty)$ dans *T*.

si $C(t) < +\infty$ alors $t = (a, \Lambda)$.

(c) S'il existe un élément *t* appartenant à *T* avec $t = (*, \Lambda)$

alors $t = (a, \Lambda)$.

(d) S'il n'existe pas d'élément $t = (a, \Lambda)$ appartenant à *T*

alors insérer (a, Λ) dans *T*.

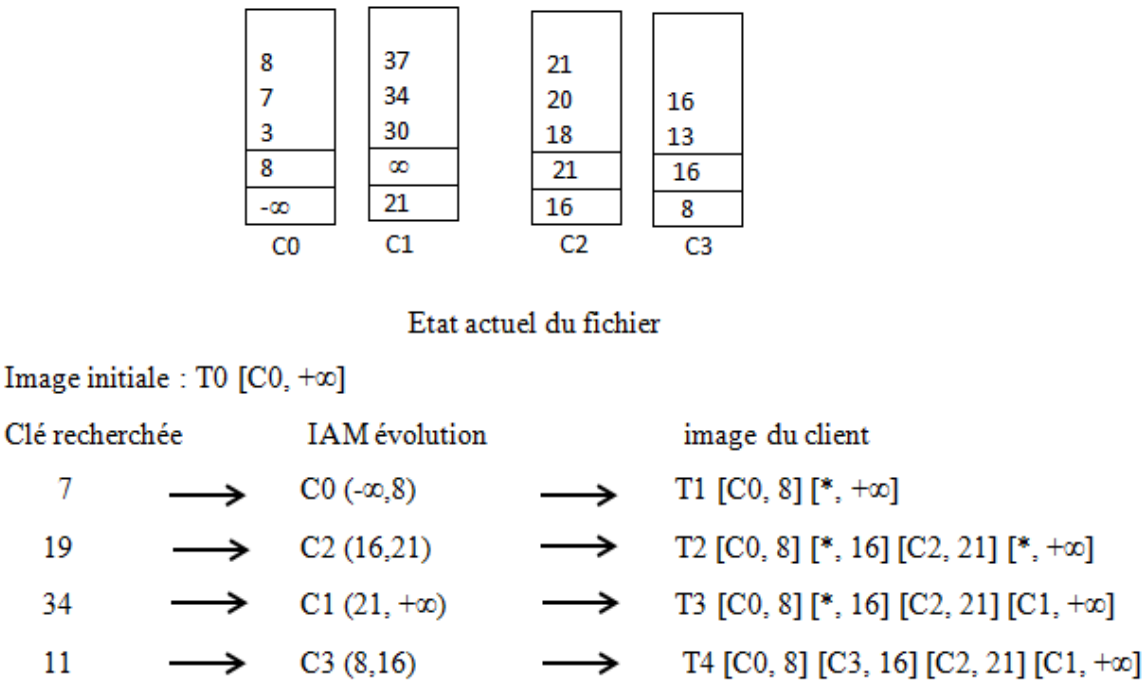


FIGURE 14 Evolution du fichier à la suite de la recherche des enregistrements (7, 19, 34,11)

6. Traitement des messages

a. Message de données

Pour pouvoir transférer des données de la manière la plus efficace, il semble important de pouvoir choisir le protocole de transport de données le plus adapté. Une négociation préalable de ce protocole est faite, avant toute transaction, de la manière suivante : le demandeur envoie la liste des protocoles dont il dispose et l'émetteur choisit le protocole correspondant le mieux au type de données à transférer en fonction de ceux dont il dispose.

Le transfert de données se fait dans le sens client vers serveur pour les opérations d'insertion et de modification d'enregistrement. Il se fait dans le sens serveur vers client en réponse à une requête de recherche. Les serveurs s'échangent des données lors de l'éclatement d'une case.

b. Messages de service

Les messages de service utilisent le protocole UDP. Ces messages contiennent les différentes commandes fonctionnelles (les informations sur les opérations à exécuter). Ces messages peuvent être unicast ou multicast en fonction du type d'opération souhaitée. Un message unicast est envoyé à un destinataire unique et le multicast est destiné à un groupe particulier de machines.

c. Messages de redirection (les forwards) RP*C

Grâce à l'utilisation de messages courts contenant des informations sur l'opération et l'émetteur, en préavis des différentes requêtes, les redirections de serveur en serveur n'ont pas à véhiculer les données (pour une requête en insertion, par exemple). C'est le serveur pouvant accepter la requête qui envoie au demandeur un préavis de transaction. La redirection arrive si un serveur reçoit un message unicast avec une clé n'appartenant pas à sa portée. Il insère alors sa portée et l'adresse de l'expéditeur dans le message reçu pour former le forward à envoyer aux autres serveurs en multicast. Le champ client est ajouté pour permettre au dernier serveur d'identifier le destinataire de la réponse. La portée de la case qui a reçu en premier le message permettra au client de corriger son image du fichier. Lorsque le nouveau serveur recevra le message, il saura, par le nom de la fonction, qu'il s'agit d'une redirection. Dans RP*C, il faut au maximum une seule redirection à un message pour arriver sur le bon serveur.
Cela est dû au fait que les messages de redirection sont reçus par tous les serveurs.

d. Messages d'accusé de réception RP*C

L'accusé de réception est automatique pour tous les messages. Il contient une valeur booléenne pour préciser s'il s'agit d'un acquittement ou pas et l'IAM qui contient les portées des cases par lesquelles est passée la requête. En cas de redirection, c'est le dernier serveur qui prend en charge l'envoi du message d'accusé de réception.

7. Protocole de communication RP*C

Chaque client a une image du fichier. Avant d'effectuer une opération, le client calcule l'adresse du serveur concerné. S'il en trouve une, il envoie un message unicast ; à défaut, il envoie un message multicast aux serveurs. L'adresse trouvée peut être erronée si le serveur concerné a subi un éclatement qui n'est pas encore pris en compte dans l'image du client.
A chaque fois qu'un serveur reçoit un message, il vérifie si la clé contenue dans le message appartient à sa portée. Si c'est le cas, il traite le message et envoie un accusé de réception au client. Si la clé n'appartient pas à sa portée et si le message était unicast, il le redirige en multicast aux autres serveurs. Pour toute requête d'un client, on se retrouve dans l'une de ces trois situations:

- Le client trouve à partir de son image une adresse correcte du serveur ;
- Le client trouve à partir de son image une adresse incorrecte du serveur ;
- Le client ne trouve pas à partir de son image une adresse du serveur.

a. Protocole d'insertion

A- Insertion avec image correcte

Le client envoie un message de demande d'insertion (*insert_request*) au serveur dont il a trouvé l'adresse. L'enregistrement est encapsulé dans la demande d'insertion si sa taille est inférieure à 64 Ko et le protocole utilisé est UDP. Apres l'insertion, le serveur envoie un acquittement avec sa portée au client (Figure 15). Si la taille de l'enregistrement est supérieure à 64 Ko, la demande d'insertion contient alors uniquement la clé. Le serveur répond au client par un message d'invitation à envoyer les données (*insert_reply*). Le transfert de l'enregistrement du client au serveur se fait sur un canal de données sous TCP par des messages *send_data* (Figure 16).

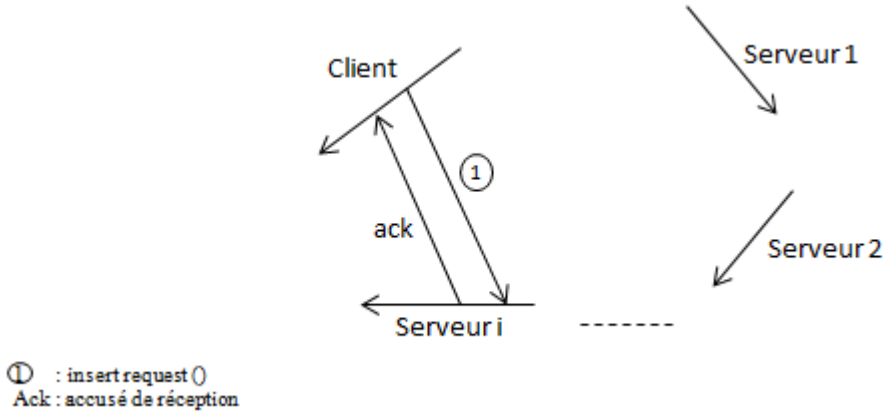


FIGURE 15 Demande d'insertion (UDP avec image correcte du fichier)

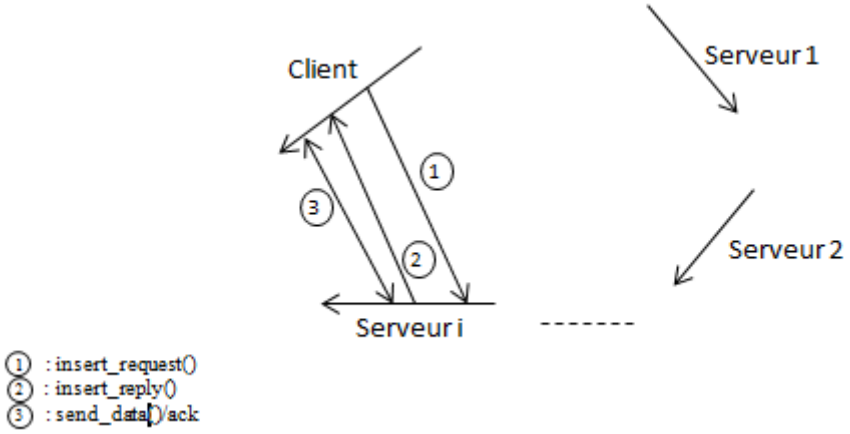


FIGURE 16 Demande d'insertion (TCP avec image correcte du fichier)

B- Insertion sans adresse serveur

Le client envoie la demande d'insertion (*insert_request*) en multicast aux serveurs. Seul le serveur concerné exécute l'opération et envoie au client un acquittement avec sa portée. Le client choisit le protocole en fonction de la taille des données : UDP (Figure 17) ou TCP (Figure 18).

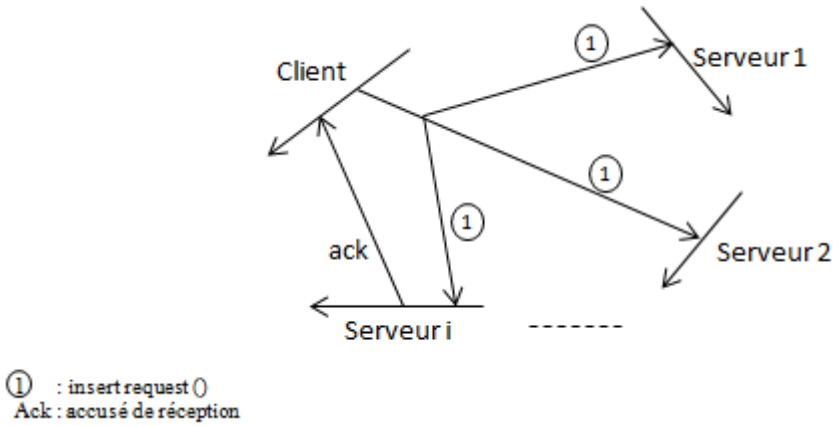


FIGURE 17 Demande d'insertion (UDP sans adresse du serveur)

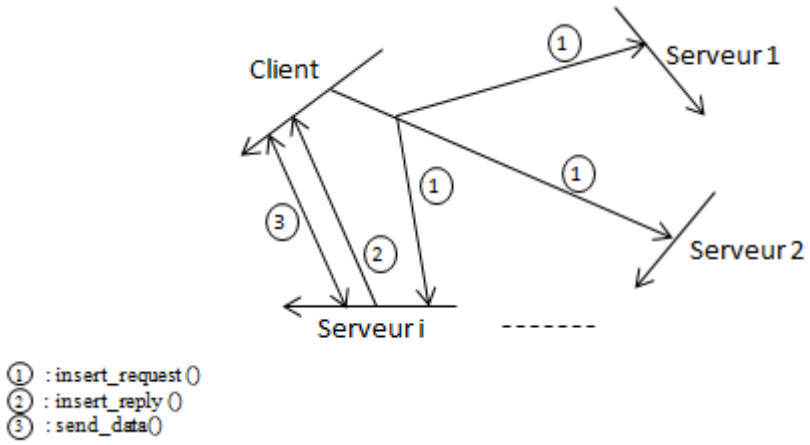


FIGURE 18 Demande d'insertion (TCP sans adresse du serveur)

C- Insertion avec image fausse

Le client envoie au serveur dont il a trouvé l'adresse une demande d'insertion. Le serveur compare la clé contenue dans le message avec sa portée et constate l'erreur d'adressage. Il redirige alors le message vers les autres serveurs. Le serveur adéquat exécute la requête et envoie un acquittement avec sa portée au client. Le client se servira de cet acquittement pour corriger son image du fichier (Figure 19, Figure 20).

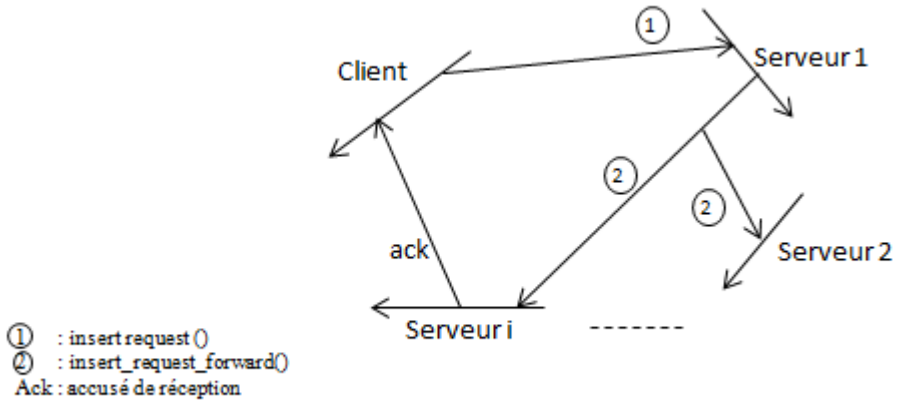


FIGURE 19 Demande d'insertion (UDP avec image fausse du fichier)

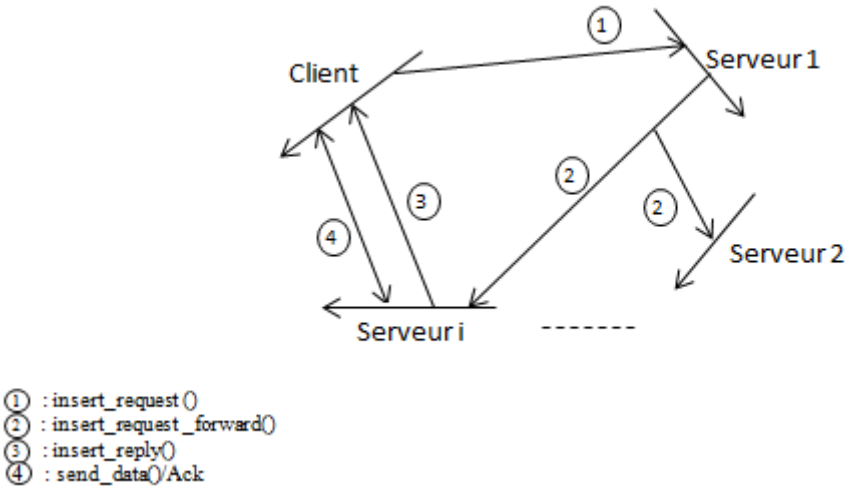


FIGURE 20 Demande d'insertion (TCP avec image fausse du fichier)

b. Protocole de Suppression

A- Suppression avec image correcte

Le client envoie au serveur une demande de suppression d'un enregistrement (delete_request). Le serveur vérifie si l'enregistrement dont la clé se trouve dans le message delete_request est bien dans sa case. Il supprime ensuite l'enregistrement et retourne un message d'acquittement avec sa portée au client. Si une erreur quelconque se produit, il envoie une notification au client (Figure 21).

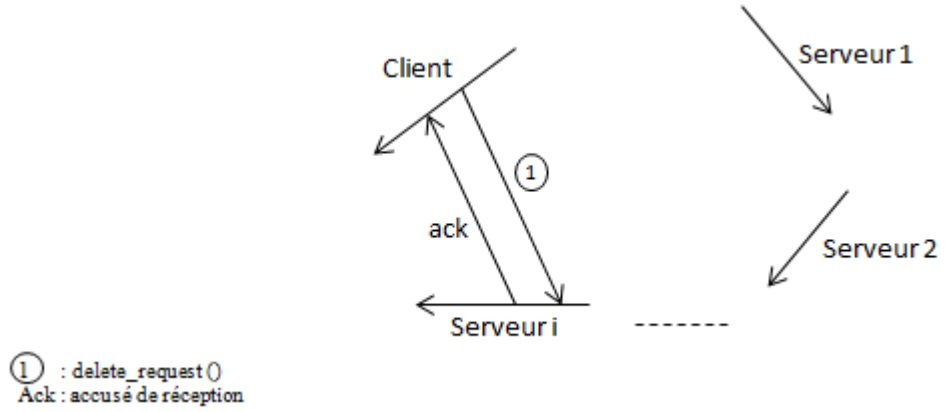


FIGURE 21 Demande de suppression (avec image correcte du fichier)

B- Suppression sans adresse du serveur

Le client envoie la demande de suppression en multicast aux serveurs. Le serveur concerné exécute la requête et répond au client par un message d'acquittement ou un message d'erreur (Figure 22).

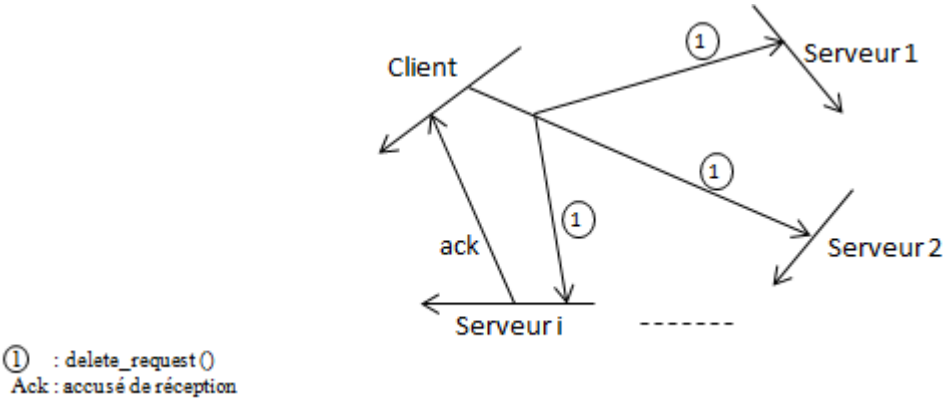


FIGURE 22 Demande de suppression (sans adresse du serveur)

C- Suppression avec image fausse

Le client envoie une demande de suppression (delete_request) à un serveur avec la clé d'un enregistrement n'appartenant pas à sa portée. Le serveur encapsule sa portée et l'adresse du client dans le message delete_request pour former le message de redirection (delete_request_forward). Ce message est envoyé vers les autres serveurs. Le serveur adéquat reçoit ce message, traite la requête et envoie une réponse au client (Figure 23).

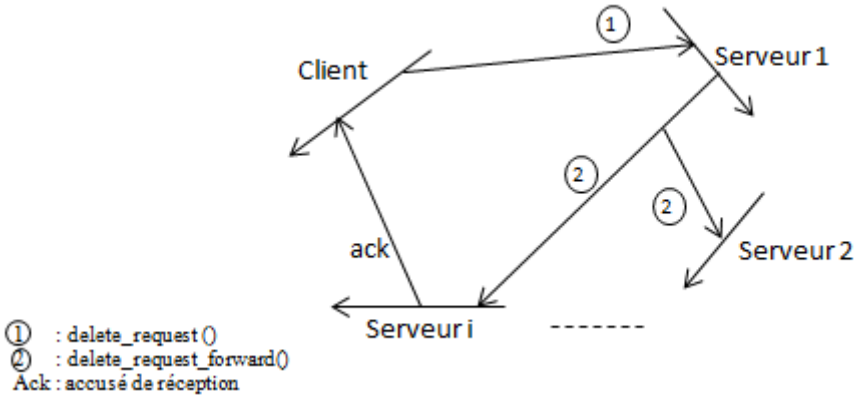


FIGURE 23 Demande de suppression (avec image fausse du fichier)

c. Recherche d'un enregistrement

A- Recherche avec image correcte

Le client trouve l'adresse correcte du serveur où se trouve l'enregistrement cherché. Il envoie une requête de recherche (seek_request) avec la clé de l'enregistrement. Le serveur envoie les données directement dans le message de réponse (seek_reply) par UDP si la taille de l'enregistrement est inférieure à 64 Ko (Figure 24). Si la taille de l'enregistrement est supérieure à 64 Ko, alors le message seek_reply contient uniquement l'adresse du serveur, l'envoi des données se faisant par un canal de données sous TCP (Figure 25).

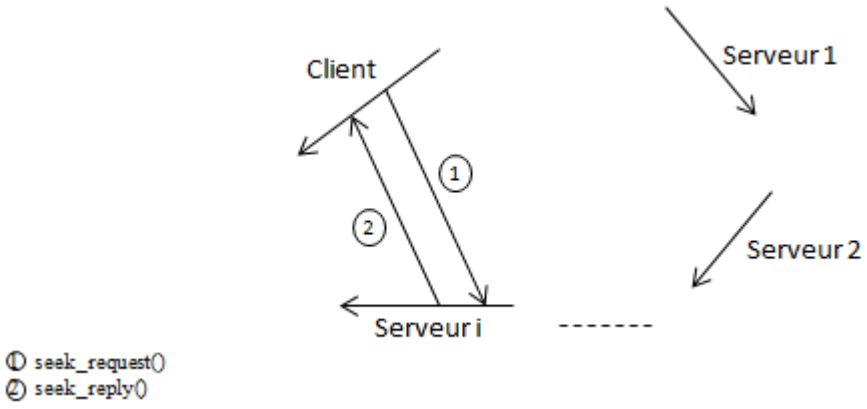


FIGURE 24 Recherche d'un enregistrement (UDP avec image correcte du fichier).

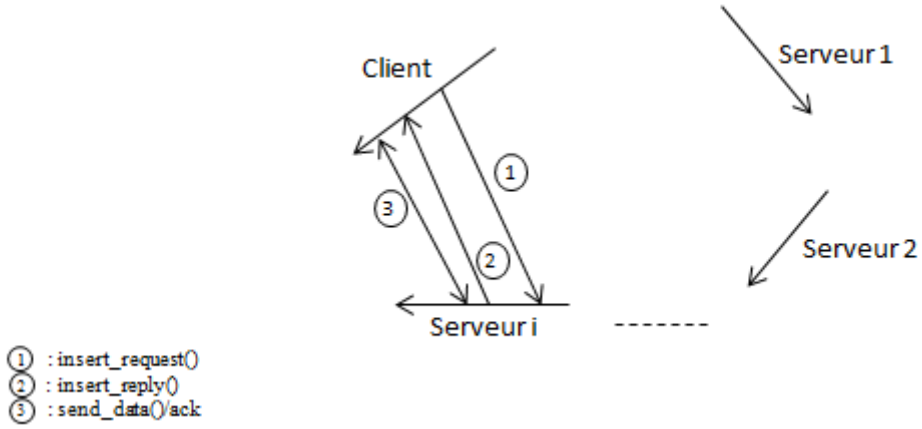


FIGURE 25 Recherche d'un enregistrement (TCP avec image correcte du fichier)

B- Recherche sans adresse du serveur

Le message seek_request est envoyé en multicast aux serveurs. Le serveur concerné répond directement par le message seek_reply avec les données sous UDP (Figure 26) ou bien par un canal de données vers le client sous TCP (Figure 27).

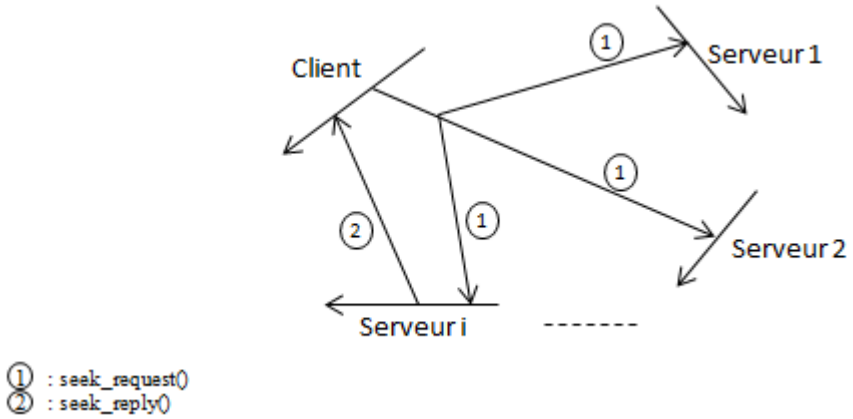


FIGURE 26 Recherche d'un enregistrement (UDP sans adresse du serveur)

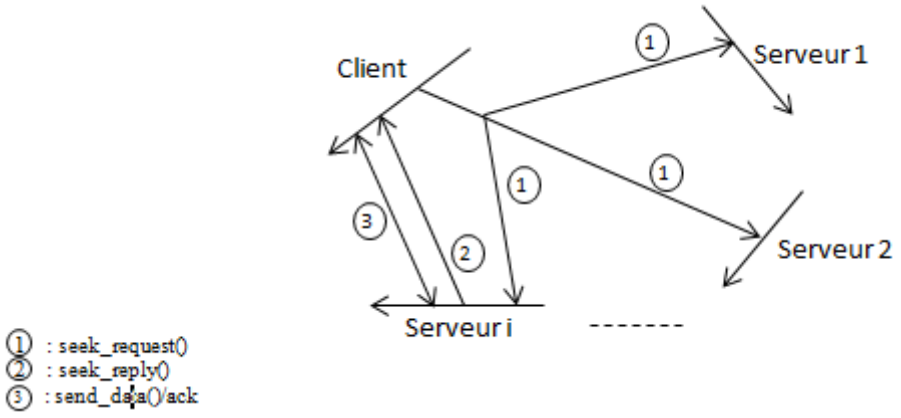


FIGURE 27 Recherche d'un enregistrement (TCP sans adresse du serveur)

C- Recherche avec image fausse

Le client envoie à un serveur le message seek_request. Le serveur constate l'erreur d'adressage et envoie un message de redirection (seek_request_forward) vers les autres serveurs. Le serveur correct répond par un message seek_reply avec les données sous UDP (Figure 28) ou par un canal de données sous TCP (Figure 29)

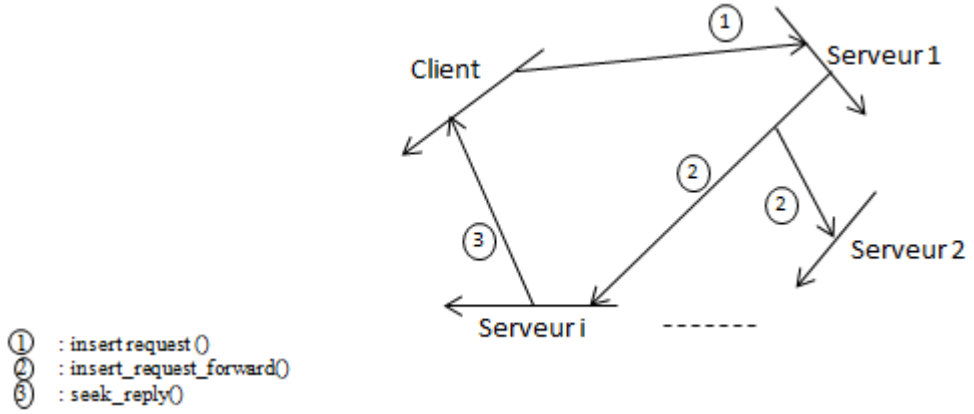


FIGURE 28 Recherche d'un enregistrement (UDP avec image fausse du fichier)

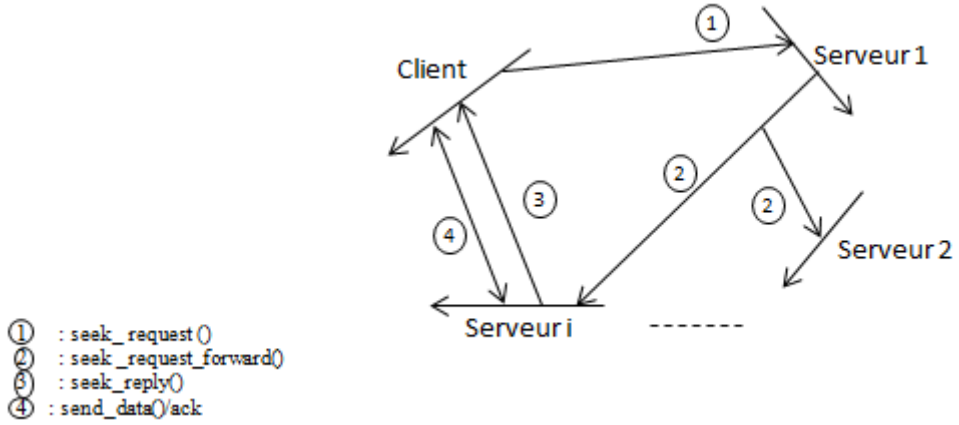


FIGURE 29 Recherche d'un enregistrement (TCP avec image fausse du fichier)

8. Description des messages

a. Insertion

A- Demande d'insertion par le client

insert_request (Id_Cde, algo, file, CB, ack, protocol_list, key, data);

Id_Cde : identifiant de la commande passée par le message : insertion, recherche, suppression, éclatement, etc.

algo : identifiant de l'algorithme : LH*, RP*N, RP*C, RP*S…

file : identifiant du fichier où les données seront insérées.

key : clé de l'article à insérer.

CB : Control Block. Contrôle d'accès, droits d'accès...

ack : pour la version 0, il est toujours à zéro ; c'est-à-dire pas d'acquittement.

protocol_list : liste des protocoles reconnus et acceptés par le client (TCP, UDP).

Data : données à insérer.

B- Redirection du message

insert_request_forward (Id_Cde, algo, file, client, CB, IAM, ack, protocol_list, key, data);

La plupart des paramètres correspondent à ceux du message insert_request. Seuls les champs Client et IAM ont été ajoutés.

Client : Adresse du client qui a initié la requête

L'IAM (Image Adjustment Message) : portée de la case SDDS du serveur qui envoie le message de redirection (ce champ n'est pas utilisé par l'algorithme RP*N)

C- Réponse du serveur où l'article doit être inséré

insert_reply (Id_Cde, algo, file, CB, forward, IAM, ack, data_port, data_protocol, key);

forward : une valeur TRUE indiquera au client que son message insert_request a été redirigé et qu'il doit ainsi prendre en compte la valeur du message IAM.

IAM : Image Adjustement Message. Portée de la case SDDS du serveur où le client a envoyé la clé.

ack : Zéro s'il n'y a pas d'acquittement, un s'il y'a acquittement.

data_port : port par lequel devront arriver les données du client.

data_protocol : protocole choisi par le serveur dans la liste *protocol_list* envoyée par le client. C'est ce protocole que le client devra utiliser pour envoyer les données.

D- Envoi des données par le client

send_data (Id_Cde, algo, file, ack, key, data) ;

Ce message utilise le protocole spécifié dans la réponse (*insert_reply*) du serveur qui va recevoir les données.

b. Suppression

A- Demande de suppression d'un article

delete_request (Id_Cde, algo, file, key, time-out);

B- Redirection de la demande de suppression d'un article

delete_request_forward (Id_Cde, algo, file, key, Client, IAM, time-out);

C- Réponse du serveur

Delete_ack (Id_Cde, algo, file, key, CB) ;

c. Recherche d'un enregistrement

A- Demande de recherche d'une clé

seek_request (Id_Cde, algo, file, key, CB, data_port, protocol_list)

B- Redirection d'une demande de recherche d'une clé

seek_request_forward (Id_Cde, algo, file, key, client, CB, data_port, protocol_list, IAM)

C- **Réponse du serveur - Message de données**.

seek_reply (Id_Cde, algo, file, key, CB, data).

d. Recherche par intervalle

A- Requête multicast

seek_range_request (Id_Cde, algo, file, servers_multicast, key_min, key_max, data_port, data_protocol, time-out)

key_min : borne inférieure de l'intervalle de recherche

key_max : borne supérieure

data_port : port par lequel devront arriver les résultats.

servers_multicast : adresse du groupe de serveurs qui recevront le message.

B- Réponse d'un serveur

seek_range_reply (Id_Cde, algo, file, key_min, key_max, CB, data).

e. Eclatement

A- Demande d'éclatement de serveur i à serveur j

eclat_request_server (Id_Cde, algo, file, level, size_bucket, CB, protocol_list, time-out);

level : portée de la case en débordement

size_bucket : taille de la case à éclater dans le serveur i.

protocol_list : liste des protocoles reconnus et acceptés par le serveur i.

time-out : le serveur i attend la réponse du serveur j jusqu'à l'expiration de ce time-out.

B- Demande d'éclatement de serveur i au coordinateur

eclat_request_coord (Id_Cde, algo, file, level, size_bucket, CB, protocol_list, time-out).

C- Réponse du serveur récepteur à une demande d'éclatement

eclat_reply (Id_Cde, algo, file, CB, data_port, data_protocol, time-out).

D- Envoi des données par le serveur en débordement

eclat_send_data (Id_Cde, algo, file, data).

E- Compte-rendu du serveur récepteur au serveur émetteur

eclat_ack_data (Id_Cde, algo, file, CB).

IV- Conclusion

Dans ce chapitre nous avons pu constater que les performances d'accès d'une SDDS RP*C sont liées à l'état de l'image du client sur le fichier. Un nouveau client produira des erreurs d'adressage jusqu'à la convergence de son image.

Les opérations de recherche, d'insertion, de modification et de suppression nécessitent alors trois messages (requête, redirection multicast, réponse). La rapidité de convergence dépend du nombre de cases qui constituent le fichier.

Pour un client avec une image correcte, les opérations de recherche, d'insertion, de modification et de suppression nécessitent deux messages point-à-point (requête, réponse).

Chapitre 5

Réalisation

I- Description du projet :

Dans le contexte de la méthode RP*C ce programme a pour finalité de simuler le comportement d'un fichier RP*C distribué sur un certain nombre de serveurs.

Pour être utile, la configuration de ce fichier est définie comme suit:

1/ Trois(3) clients sollicitant des serveurs qui sont supposés héberger le fichier RP*C. Pour les besoins de cette simulation le nombre maximum de serveurs est fixé à 30. La capacité d'un serveur est fixée à 50 articles que l'on représente par des clés alphabétiques.

Ses serveurs sont désignés par la variable "server_rpc" de type tableau de packed array dans le programme.

2/ A chaque serveur qui est en principe représenté par une case est associé un intervalle où figurent respectivement sa clé minimale et sa clé maximale. L'intervalle en question est représenté par la variable nommée "interval" dans le programme. On associe aussi à chaque serveur un élément pointeur nommé indice_remp_ser : qui indique si le serveur adressé est saturé. Cet élément constitue en même temps l'index de la dernière clé déjà insérée dans la case qui représente le serveur.

3/ En ce qui concerne les clients, Ils sont représentés chacun par sa table, que l'on désigne respectivement par la variable image_client.

Ces dernières sont de type tableau de packed array. Leur taille a été fixée à 30 dans le programme pour le faire correspondre au nombre de serveurs.

Dans le cadre de cette réalisation on se limite à trois(3) clients.

4/ IAM est représenté par le triplet (landa,passer,omega) permettant de mettre à jour l'image d'un client. En appliquant l'algorithme, ce triplet permet de modifier la table d'un client afin de rapprocher son image de la réalité détenue par les serveurs sollicités.

5/ Remarques:

a- Pour rappel, ce programme est chargé d'insérer dans les serveurs les clés qui sont envoyées par les trois(3) clients envisagés.

b- Quand une clé existe(ou a été déjà insérée dans un serveur) si un client envoie une nouvelle clé pour insertion elle ne sera pas dupliquée dans le fichier serveur, mais un IAM sera envoyé vers le client qui a formulé la requête d'insertion.

II- L'application SDDS-RP*C / Client

Le rôle de l'application client est de permettre à des applications de formuler des requêtes sur le fichier RP*C comme si celui-ci se trouvait sur disque, rendant ainsi transparente l'émission des requêtes vers les sites serveurs et la réception des réponses. Les applications destinées à manipuler le fichier RP*C vont donc faire appel à des routines implémentées au niveau des applications client RP*C.

L'application Client est composée de deux modules de traitement qui tournent en parallèle : le premier écoute les messages en provenance des applications et le second, les messages en provenance des sites serveurs.

Le module d'écoute des messages émis par les applications adopte le fonctionnement suivant :

E1 - Attendre l'arrivée d'une requête émise par une application ;

E2 - Chercher l'adresse du serveur adéquat ;

E3 - Envoyer la requête vers ce serveur ;

E4 – Retourner à l'étape E1.

Le module d'écoute des réponses émises par les applications serveurs SDDS-RP*C adopte le fonctionnement suivant :

A1 - Attendre l'arrivée d'une réponse à une requête ;

A2 – Mettre à jour l'image du client sur le fichier ;

A3 – Vérifier à quelle application appartient la réponse reçue ;

A4 – Retourner la réponse à l'application trouvée ;

A5 – Retourner à l'étape A1.

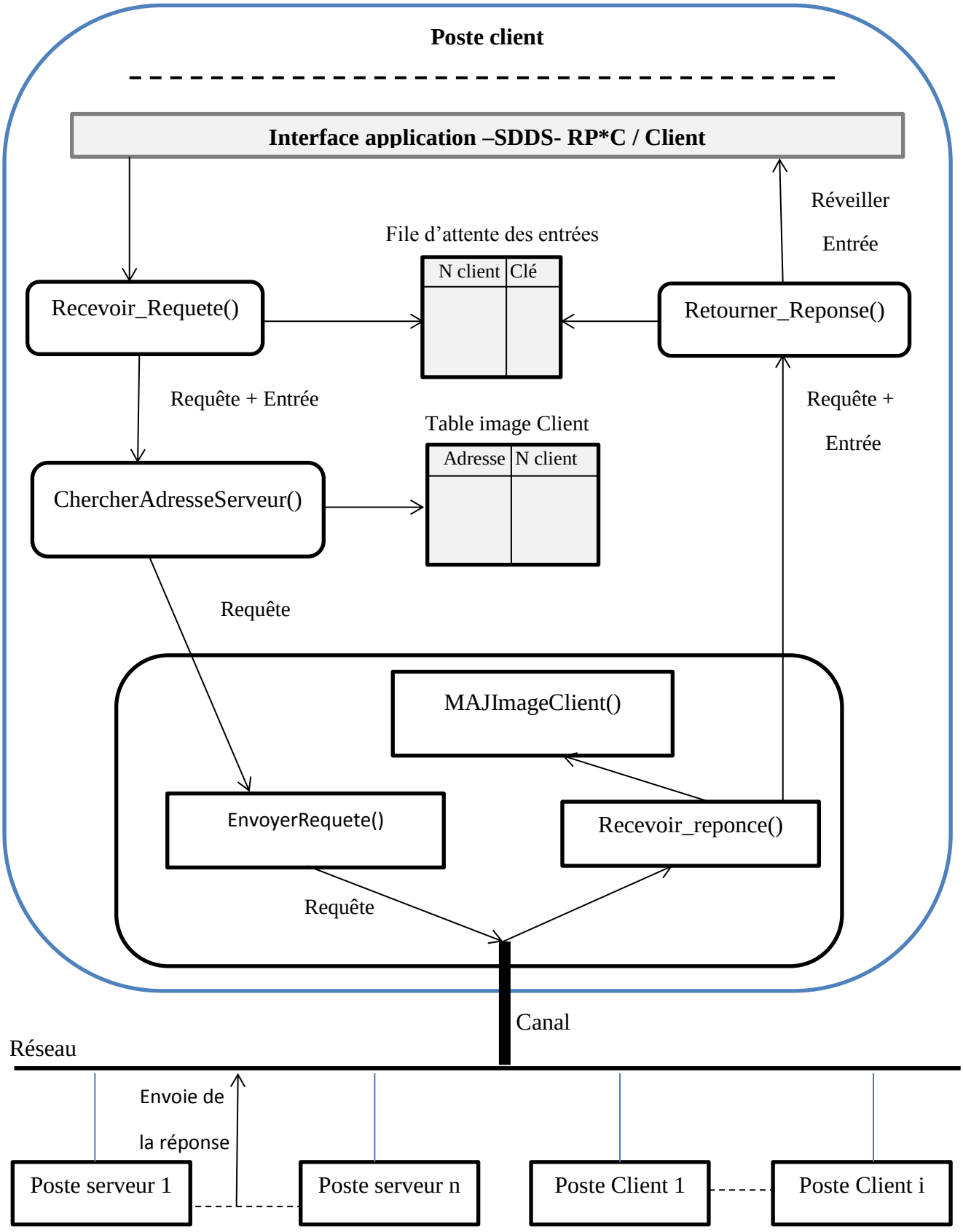


Figure 30 : SDDS-RP*C / Client

III- L'application SDDS-RP* / Serveur

Le rôle des applications serveurs est de recevoir les requêtes émises par les applications Client RP*C, d'effectuer le traitement demandé sur le fichier RP*C et de retourner la réponse. C'est l'application serveur qui accède au fichier RP*C stocké en mémoire vive.
Le serveur SDDS-RP*C est composé d'une partie Communication qui se charge de recevoir les requêtes émises par les sites client et de retourner les réponses une fois le traitement effectué. Une autre partie dite Traitement se chargera d'analyser et d'effectuer les traitements demandés dans la requête.
Conçu pour servir un grand nombre de clients, le programme serveur utilise un thread qui se charge de mettre les requêtes clients reçues dans une file d'attente. Le serveur maintient également une file d'attente de thread-de-travail dont le nombre est fixé en fonction de la charge au niveau du serveur.

La partie Communication adopte le fonctionnement suivant :
E1 - Attendre l'arrivée d'une requête émise par un client RP*C ;
E2 - Mettre la requête dans la file d'attente et signaler l'événement ;
E3 - Retourner à l'étape E1.

La partie Traitement adopte le fonctionnement suivant :
T1 - Attendre que l'événement arrivée requête soit signalé ;
T2 - Prendre une requête de la file d'attente ;
T3 - Analyser la requête pour voir le traitement demandé ;
T4 - Selon le cas, lancer le traitement local, la redirection de la requête ou l'éclatement de la case ;
T5 - Retourner à l'étape T1.

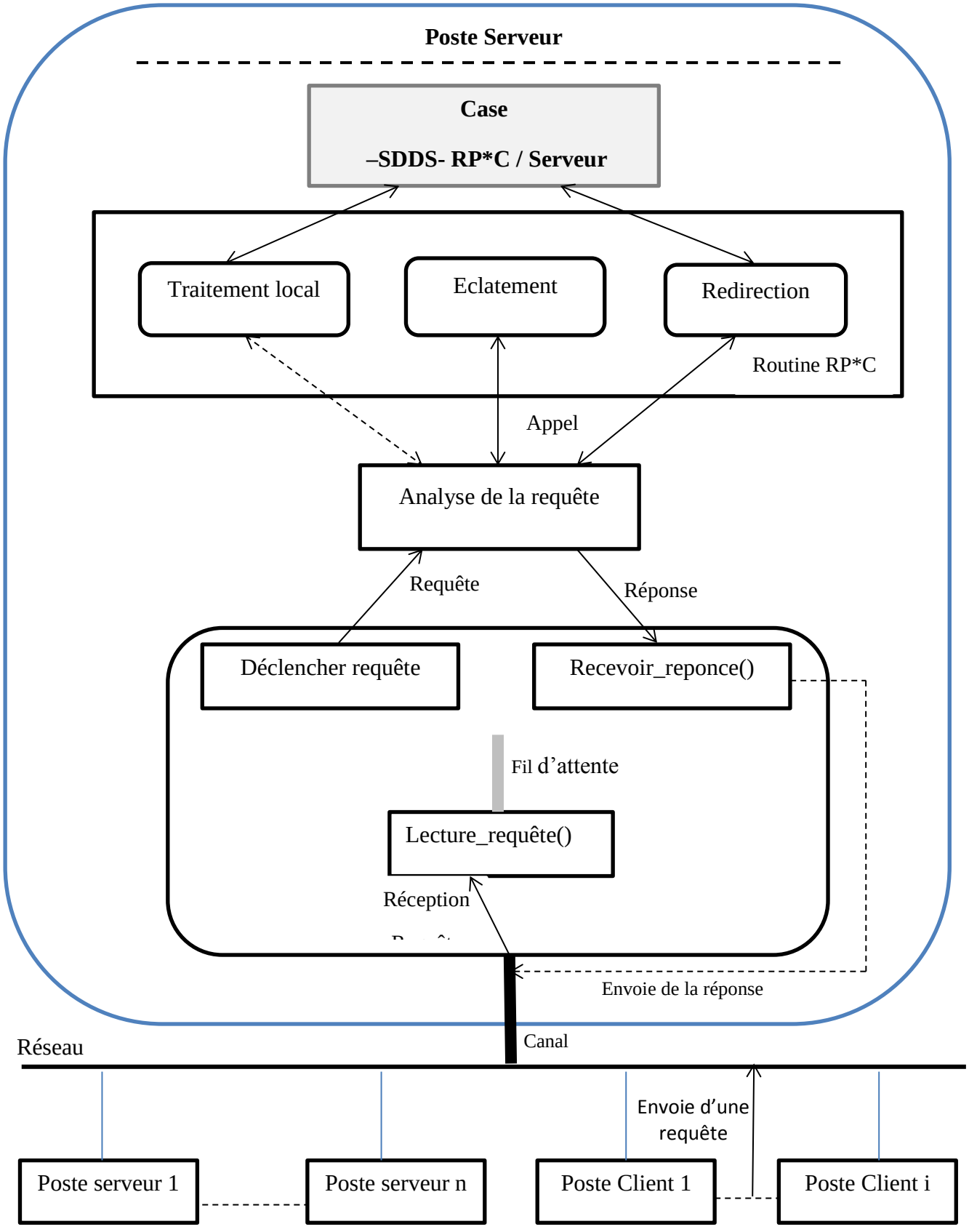


Figure 31 SDDS-RP*C / Serveur

Un client RP*C dispose d'une image. Celle-ci lui permet d'utiliser des messages unicast à chaque fois qu'il s'adresse à un serveur déjà référencé dans l'image. Dans le cas contraire, il utilise un message multicast adressé à un serveur déjà connu, le plus proche dans l'ordre des intervalles de celui ou ceux à trouver.

Le serveur recevant une requête à un fichier RP*C vérifie d'abord si la clé appartient à l'intervalle de sa case. Si la requête est un multicast et la clé est hors l'intervalle, alors elle est ignorée. Si c'est une requête unicast et la clé est hors intervalle, alors le serveur du fichier RP*C envoie la requête en multicast à l'ensemble des serveurs en ajoutant son adresse et son intervalle de clés.

IV- Les principales méthodes utilisées dans le programme principal :

1- **Multicast :** cette méthode permet de calculer l'adresse du serveur auquel doit être envoyé la requête passée par le client. Ce calcul consiste à parcourir l'image du client sue le fichier et à rechercher le serveur dont la portée de la case recouvre la clé contenue dans la requête.

2- **Insert :** méthode d'insertion d'une clé dans un serveur si celle-ci n'existe pas dans le serveur car elle ne sera pas dupliquée dans le fichier serveur.

3- **Eclatement :** Ce processus est déclenché quand la case a atteint le nombre maximum d'enregistrements qu'elle peut contenir. Il s'agit de :

- Envoyer un message de recherche de serveur disponible ;
- Dès réception d'une réponse, se connecter au serveur disponible ;
- Déplacer la moitié des enregistrements de la case en débordement vers la nouvelle case ;
- Mettre à jour la portée de la case initiale.

4- **IAM :** méthode de mise à jour des images clients, cette fonction se chargera d'extraire le champ IAM de la réponse du serveur et mettra à jour l'image du client sur le fichier RP*C.

5- **Server :** méthode de gestion des serveurs qui est chargé du contrôle du vecteur indice du remplissage des serveurs.

V- Conception

Pour la conception de notre programme nous avons opté pour le langage de programmation Python, qui est un langage de programmation interprété, multi-paradigme et multiplateformes.

Il favorise la programmation impérative structurée, fonctionnelle et orientée objet. Il est doté d'un typage dynamique fort, d'une gestion automatique de la mémoire par ramasse-miettes et d'un système de gestion d'exceptions.

Le langage Python fonctionne sur la plupart des plates-formes informatiques. Il est conçu pour optimiser la productivité des programmeurs en offrant des outils de haut niveau et une syntaxe simple à utiliser.

Python est un langage de programmation qui peut s'utiliser dans de nombreux contextes et s'adapter à tout type d'utilisation grâce à des bibliothèques spécialisées

Conclusion

Ainsi se termine notre mémoire alors que la recherche dans le domaine des SDDS ne finira pas de se développer. Car l'apparition de ces nouvelles structures de données a ouvert un vaste champ d'exploration et de recherches dans la technologie informatique aux horizons sans limites.

Avec l'évolution des réseaux informatiques, un nouveau concept a été introduit : celui des multiordinateurs, et afin de profiter au maximum des ressources de stockage et de traitement de ces réseaux d'ordinateurs, la structure de données distribuées et scalables (SDDS) a été inventée. Depuis, plusieurs SDDS ont été proposées, à savoir les SDDS basées sur le hachage constituant une généralisation des méthodes d'accès classiques sur les multiordinateurs et les SDDS RP* qui représentent l'élargissement des structures classiques d'accès ordonné sur un environnement distribué.

Notre objectif étant de concevoir un prototype qui montre le comportement d'un fichier RP*c distribué sur un certain nombre de serveurs, et à travers les résultats obtenus, nous avons pu démontrer les performance de la méthode RP*c, d'augmenter la capacité des serveurs en procédant à l'éclatement des cases tout en préservant l'ordre des enregistrements, et l'extension d'un fichier sur un nombre quelconque de machines suite à des insertions.

Notre recherche et la documentation sur les structures de donnée notamment sur les SDDS nous a permis d'approfondir nos connaissances dans ce domaine.

Enfin, notre travail ouvre l'horizon à plusieurs directions de recherche. De nombreux problèmes méritent une étude spécifique tels que la consistance des données, la sécurité d'accès aux fichiers SDDS, les opérations complexes sur des fichiers SDDS (jointure, fonctions agrégat...).

Bibliographie :

Structures de données et algorithmes, Notes de cours de Bruno, Malenfant, Université du Québec à Montréal 2017

Structures de données, eduscol.education.fr/ Ministère de l'Éducation nationale et de la jeunesse - juillet 2020

Signatures Algébriques dans la Gestion de Structures de Données Distribuées et Scalables, thèse de doctorat, Riad MOKADEM, Juin 2006

Un Framework pour le traitement parallèle d'une très grande quantité de données, Mohamed Aridj, 2015

Nouvelle implémentation des SDDS dans un environnement Java/Linux, Samba NDIAYE, Balla Wade DIACK, Université Cheikh Anta DIOP de Dakar, 2002

Structure d'une case SDDS RP*, Mohamed Aridj, 2015

Proposition d'une solution de journalisation pour les structures de données distribuées et scalables(SDDS) de type hachage linéaire distribué (LH*), mémoire de magister, BOUCETTE Mohammed, 2011

SDDS LH*$_{TT}$: Une solution pour la scalabilité d'une relation temporelle de transaction standard, Mohamed Najeh ISSAOUI et Rafik BOUAZIZ, SETIT 2009