REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE

MINISTERE DE L'ENSEIGNEMENT SUPERIEUR ET DE LA RECHERCHE SCIENTIFIQUE

UNIVERSITE DR. YAHIA FARES DE MEDEA

FACULTE DES SCIENCES

Département mathématique et informatique





Mémoire de Fin d'Études

Présenté pour l'obtention du Diplôme
De Master en

Option : Ingénierie des systèmes et technologies du web.

THEME

Utilisation du hachage parfait minimal pour l'amélioration des performances des SGBDs

Proposé et dirigé par :

Dr : BELAZZOUGUI Djamal

Mr : TOULAIT Sofiane

Préparé et présenté par :

Melle : LABIDI Amel

Melle : KHELLADI Maroua

Année Universitaire 2019 - 2020

Dédicace

A Allah, Tout puissant Qui m'a inspiré, Qui m'a guidé dans le bon chemin. Louanges et remerciements pour votre clémence et miséricorde.

À MES CHERS PARENTS

Aucune dédicace ne saurait exprimer mon respect, mon amour éternel et ma considération pour les sacrifices que vous avez consenti pour mon instruction et mon bien être. Je vous remercie pour tout le soutien et l'amour que vous me portez depuis mon enfance et j'espère que votre bénédiction m'accompagne toujours.

Puisse Dieu, vous accorder santé, bonheur et longue vie.

À mes adorable sœurs Hadia et Lydia pour leurs encouragements permanents et leur soutien et ma nièce Iline

A mes chers cousine Feriel, Nihel et Nessrine.

Et toute la famille KHELLADI & BENHADDAD.

A ma binômes Amel les mots ne suffisent guerre pour exprimer l'attachement, l'amour et l'affection que je porte pour vous.

A tous mes amis, vous êtes pour moi des sœurs : Zidouma, Hanane, Salima et Zahia, Khadîdja, Zola.

A tous les enseignants qui m'ont enseigné depuis mon enfance.

A tous ceux que j'aime et qui m'aiment



Maroua

Dédicace

Louange à Dieu, le miséricordieux, sans lui rien de tout cela n'aurait pu être.
Je dédie ce précieux travail aux êtres les plus chers à mon cœur et à qui je dois énormément toute ma vie, mon très cher père pour son soutien, ainsi que ma tendre maman qui n'a jamais cessé de m'encourager dans les moments les plus difficiles et pour ses incomparables sacrifices.
À mes chères sœurs Nassima et Nerimene. Et mes frère Kamel et Youcef.
À mon fiancé Walid et ma deuxième famille.
À ma binômes Maroua les mots ne suffisent guerre pour exprimer l'attachement, l'amour et l'affection que je porte pour vous.
Je tiens à présenter mes remerciements à mes chères amies : Atika ;Bouchra ; Hanane ; Salima ; Zahia.

À tous mes proches.

Amel

Remerciements

Nous remercions Dieu pour nous avoir donné la force, la patience et volonté pour réussir notre mémoire.

Nous remercions nos parents pour leur aide et leurs soutiens en toutes circonstances.

Nous remercions particulièrement nos encadreurs Mr TOULAIT Sofiane et Dr BELAZZOUGUI Djamal pour leurs conseils, leurs aides durant toute la période du travail et la confiance qu'ils nous ont accordée.

Nous remercions le président du jury et ses membres.

Nous remercions tous nos amis et camarades

Nous souhaitons également exprimer notre profonde gratitude à tous ceux qui du près ou de loin ont participé à la réalisation du présent travail.

Résumé :

Les structures de données basées sur le hachage connaissent des améliorations considérables ces dernières années, en effet, leur utilisation est intéressante dans plusieurs domaines, en particulier dans le domaine des bases de données.

Dans de ce mémoire une variante de technique de hachage est utilisée elle s'appelle « le hachage parfait minimal ». Le mot parfait veut dire qu'il n'y 'a pas de collisions dans la table de hachage et le mot minimal veut dire la structure de données occupe un espace minimal égal au nombre de clés (la table de hachage ne contient pas de cases vides).

Le but de travail est de montrer l'efficacité du hachage parfait minimal lorsqu'il est utilisé avec la fragmentation horizontal fait à base d'un attribut donné (diffèrent de la clé primaire) d'une table de la base de données et lorsqu'une requête est faite sur une clé se trouvant dans l'une des partitions.

Le hachage parfait minimal nous évitera alors de parcourir toutes les partitions ce qui constitue un gain de temps. Des tests sont effectués sur une table implémentée sous le SGBD MySQL, ce qu'il montre l'utilité du travail (temps de repense réduit).

Mots Clés : Hachage, index, Mysql, hachage parfait minimal, requêtes, base de données.

Abstract:

The hash-based data structures have undergone considerable improvements in recent years, indeed, their use is interesting in several fields, in particular in the field of databases.

In this memo, a variation of the hashing technique is used, referred to as "minimal perfect hash". The word perfect means that there are no collisions in the hash table, and the word minimal means that the data structure occupies a minimum space equal to the number of keys (the hash table does not contain empty boxes).

The aim of the work is to show the efficiency of the minimal perfect hash when used with horizontal fragmentation made on the basis of a given attribute (different from the primary key) of a database table and when a request is made on a key located in one of the partitions.

The minimal perfect hash will then prevent us from iterating through all partitions, which saves time. The tests are carried out on a table implemented under the MySQL DBMS, which shows the usefulness of the work (reduced reflection time).

Keywords: Hash, index, Mysql, minimal perfect hash, queries, database.

Table des matières

Dédicace....A
Remerciement....B
Résumé....D
Abstract....E
INTRODUCTION GENERALE....1
Chapitre 1 : Généralités sur les bases de données....2
1 Introduction aux bases de données....3
1.2 Qu 'est-ce qu'une base de données....3
1.3 Table (ou Relation) et Attribut....3
1.3.1 Clés primaires....4
1.3.2 Clés étrangères....4
1.4 Utilité d'une base de données ?....5
1.5 La gestion des bases de données....5
1.6 Les principaux SGBD....6
2 Système de gestion de base de données....7
2.1 Dispositif de stockage....7
2.2 Mémoires....7
2.3 Performances des mémoires....9
2.4 Disques....10
2.5 Structures d'index (le hachage)....11
2.6 Recherche dans une table de hachage....11
3 La structure de données....12
4 Requêtes à intervalles (range queries)....12
4.2 SQL BETWEEN....12
4.3 Syntaxe....13
5 Résolues en utilisant les B-trees(arbre-B)....13
5.1 Conception logique de la base....13
5.2 Definitions Les arbres-B....13
5.2.1 Qu 'est ce qu'un index B-tree....13
6 Fragmentation....14
6.1 Pourquoi fragmenter ?....15
6.2 Comment fragmenter ?....15
6.3 Fragmentation horizontale....15
6.4 Techniques de Fragmentation horizontale....16
6.4.1 Fragmentation horizontale primaire....16

6.4.2 Fragmentation horizontale dérivée 17
6.5 Fragmentation verticale 17
6.5 Fragmentation mixte 17

7 MySQL 18
7.1 Caractéristique 18
Conclusion 18
Chapitre 2 : Techniques de hachage 19
1 Introduction 20
2 Le hachage 20
2.1 Définition de hachage 21
2.2 Fonctions de hachage 22
3 Les méthodes de résolution des collisions 27
4 Le Hachage parfait 28
4.1 Hachage parfait non minimale 29
4.2 Hachage parfait minimal 30
4.2.1 Exemple de fonction de hachage parfaite minimale 31
5 Quelle la différence entre le hachage parfait et le hachage traditionnel ? 32
6 La technique de résolution de requêtes à intervalles en temps constant en se basant sur le hachage 33
7 L'index 33
7.1 Définition d'index 33
7.2 Intérêt des index 33
7.3 désavantages 33
7.4 Comparaison entre index arbre-B et Bitmap 34
7.4.1 Index Bitmap 34
7.4.1 Index B-Arbre 34

8 Introduction aux CMPH - C Minimal Perfect Hashing Library 35
8.1 Définition CMPH 36
8.1.1 Caractéristiques de la bibliothèque CMPH 37
8.2 Algorithmes pris en charge 37
Conclusion 39
Chapitre 3 : Protocole d'expérimentation 40
1 Introduction 41

2 Définitions Partitionnement horizontal 41
2.1 Les avantages du partitionnement 41
2.2 Le partitionnement horizontal avec MySQL.....42
2.2.1 Les méthodes de partitionnement de MySQL 42
3 Description de l'approche 43
4 Plan global du travail 44
4.1 Plan détaillé du travail 45
Conclusion 50
Chapitre 4 : Test et évaluation 51
1 Introduction 52
2 Environnement de développement 52
2.1 Oracle VM VirtualBox 52
2.1.1 Fonctionnalités 52
2.2 Ubuntu 53
2.3 GCC.....54
2.4 Make.....55
2.5 Télécharger et installer cmph-2.0.2.....55
2.6 Mysql 56
3 Implémentation 57
4 Explication de travail 63
5 Les résultats.....66
Conclusion 68
Conclusion générale.....70
Bibliographie.....71

LISTE DES FIGURES

Figure 1 : Modélisation d'une base de données.....3
Figure 2 : Schéma d'une relation.....4
Figure 3 : Schéma La gestion des bases de données.....6
Figure 4: Hiérarchie des mémoires….....8
Figure 5 : Le cache et le disque, ressources mémoires allouées au SGBD...…….....10
Figure 6 : la structure du B-tree.....14
Figure 7 : Fragmentation Horizontale.....16
Figure 8 : Structures de données…….....21
Figure 9 : Fonction de hachage cryptographique.....22
Figure 10 : Utilisation de fonction de hachage h pour mapper l'univers u de toutes les clés…...23
Figure 11 : Méthode de hachage……...…….....25
Figure 12 : Non utilisation d'une fonction hachages...…….26
Figure 13 : Utilisation d'une fonction hachages……...…….....27
Figure 14 : Schéma Comment la structure de données S doit-elle être organisée…..…….....28
Figure 15 : Collision…….....28
Figure 16 : Fonction hachage parfaite non minimale……...29
Figure 17 : Fonction hachage parfaite minimale……...…….....30
Figure 18 : Fonction de hachage parfaite minimale (MPHF)……..31
Figure 19 **:** La structure B-Arbre…….....…35
Figure 20 : Bibliothèque CMPH……..…….36
Figure 21 : Représente la partitionnement horizontal…….…..41
Figure 22 : Plan global de l'approche proposée…….…...44
Figure 23 : Exemple de création d'un index de 5 cases……....48
Figure 24 : Exemple de recherche de 5 codes dans les 10 tables……..49
Figure 25 : Exemple de recherche 5 codes dans la table d'origine…….…….49
Figure 26 : Exemple de recherche d'un code_emp en utilisant l'index……...50
Figure 27 : Oracle VM VirtualBox 6.1...……...…….……...53
Figure 28 : Ubuntu 20.04 Focal Fossa……...…….……...54
Figure 29 : Installer gcc……..…….……...…...55
Figure 30 : Installer make……...…….....…….55
Figure 31 : Création des 11 tables ……..…….…..…..57
Figure 32 : Remplissage des 11 tables ……..……..……..…..58
Figure 33 : Compté le nombre d'enregistrements dans chacune des 11 tables……..…….....58
Figure 34 : Compilation et exécution du programme 4……...…….59
Figure 35 : Compilation et exécution du programme 5……...…….60

LISTE DES FIGURES

Figure 36 : Compilation et exécution du programme 6……61

Figure 37 : Compilation et exécution du programme 7……62

Figure 38 : Compilation et exécution du programme 8……63

Figure 39 : Résultat d'exécution programme 6……..64

Figure 40 : Résultat d'exécution programme 7……...……65

Figure 41 : Résultat d'exécution programme 8……..66

Figure 42 : Diagramme du temps d'exécution (sec)……..……..67

LISTE DES TABLEAUX

Tableau 1 : Performance des divers types de mémoire…….9
Tableau 2 : Temps d'exécution de chaque programme……..69

Liste des acronymes

SGBD : système de gestion de bases de données.

MPHF : minimal Perfect hash function

CMPH : C minimal Perfect hash function Library

INTRODUCTION GÉNÉRALE

En programmation informatique, les structures de données permettent de :

- ❖ Stocker les données
- ❖ Faciliter la manipulation des données par les algorithmes

L'indexation est une technique qui permet d'accéder aux données indexées rapidement. Les index ont donc une grande importance dans les bases de données. Ils sont utilisés pour accélérer les requêtes (notamment les requêtes impliquant plusieurs tables, ou les requêtes de recherche), dans les SGBDs.

Les structures de données basées sur le hachage sont confrontées aux problèmes de collisions. Les chercheurs ou donc pensé à proposer des techniques de hachage sans collision, ce qui a donné le hachage parfait. Ainsi un index basé sur le hachage parfait n'a pas de problèmes de collision, ce qui permet de gagner du temps dans les recherches. Un autre problème des indexes est l'espace d'occupé dans la mémoire. Les chercheurs ont donc pensé à éliminer les cases vides non utiles dans les indexes, ce qui a conduit à la proposition du « hachage parfait minimal », qui :

- ❖ Évite le problème des collisions
- ❖ Évite les cases vides à intérieure de l'index

Dans le cadre de ce mémoire nous allons montrer l'utilité du hachage parfait minimal par l'implémentation de trois tests sur 1000 requêtes, et nous allons faire la comparaison des temps d'exécution.

Chapitre 1 : Généralités sur les bases de données

1. Introduction aux bases de données :

Dans ce premier chapitre, nous allons présenter les bases de données en rappelant les principes et les concepts plus les techniques utilisées de base. Par la suite, nous révélerons les problèmes les plus importantes auxquels nous sommes confrontés dans la capacité de stockage et le temps que cela prend et Les méthodes de résolution des collisions.

1.2. Qu'est-ce qu'une base de données :

Une base de données (son abréviation est BD, en anglais data base DB, est une entité dans laquelle il est possible de stocker des données de façon structurée et avec le moins de redondance possible. Ces données doivent pouvoir être utilisées par des programmes, par des utilisateurs différents. Ainsi, la notion de base de données est généralement couplée à celle de réseau, afin de pouvoir mettre en commun ces informations, d'où le nom de base. On parle généralement de système d'information pour désigner toute la structure regroupant les moyens mis en place pour pouvoir partager des données [1].



Figure 1 : Modélisation d'une base de données

1.3. Table (ou Relation) et Attribut :

Une table (ou relation) est un tableau à deux dimensions composées de colonnes appelée attribut et de lignes, l'intersection de ces derniers contenant les données. Une table peut en contenir un nombre quelconque de lignes. On trouve aussi ce qu'on appel enregistrement, n-

uplet ou tuple pour désigner une ligne et un champ pour désigner une colonne ; un enregistrement est donc un ensemble de valeurs, chacun renseignant un champ [2].

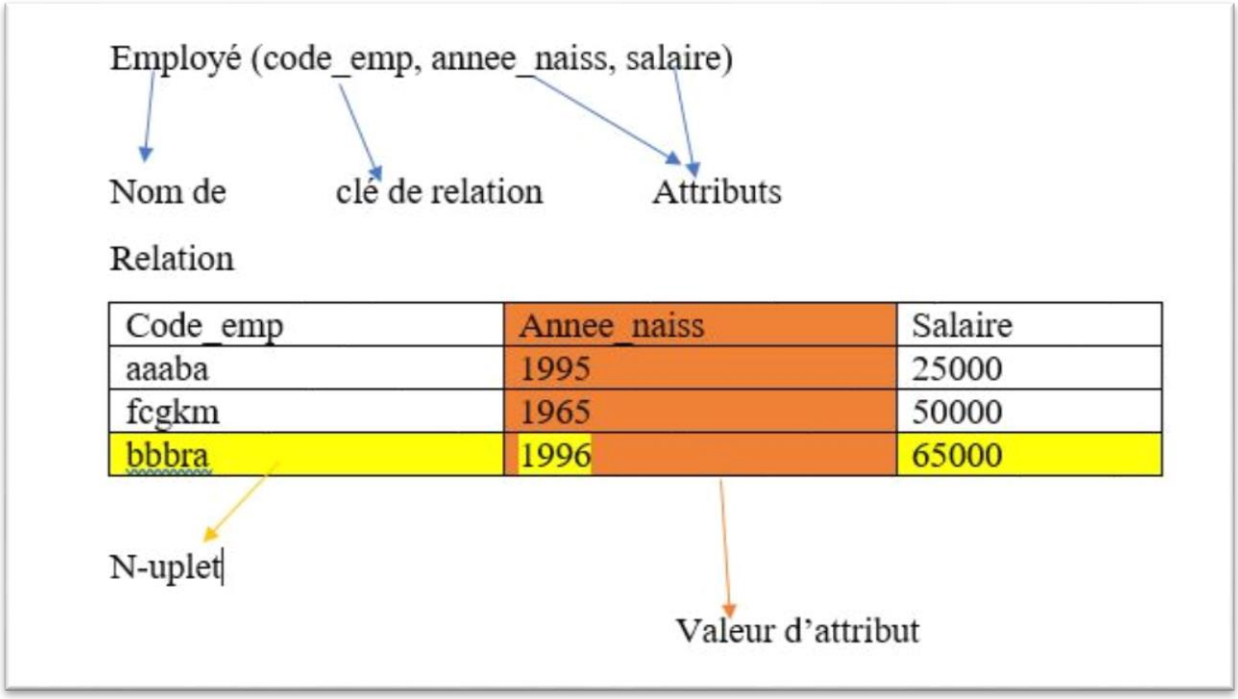


Figure 2 : Schéma d'une relation

1.3.1. Clés primaires :

Dans la table de l'exemple ci-dessus, le code_emp est un attribut constitué d'un code unique. Cet attribut particulier, appelé clé, est indispensable qui permet d'identifier chaque employé d'une manière unique. En effet, l'attribut annee_naiss peut prendre la même valeur plusieurs fois et donc ne peut être la clé de la table, de même si l'on avait disposé du salaire des employé, la combinaison d'attributs <annee_naiss et salaire> n'aurait pas été d'avantage satisfaisante comme clé, car il est possible que deux employés portent le même annee_naiss et le même salaire à la fois.

1.3.2. Clés étrangères :

Une clé étrangère est la duplication de la clé primaire d'une table dans une autre table dans le but de traduire une association conceptuelle entre les deux entités représentées par les deux tables. Ce concept est un des concepts majeurs du modèle relationnel.

1.4. Utilité d'une base de données ?

Une base de données permet de mettre des données à la disposition d'utilisateurs pour une consultation, une saisie ou bien une mise à jour, tout en s'assurant des droits accordés à ces derniers. Elle peut être locale, c'est-à-dire utilisable sur une machine par un utilisateur, ou bien répartie, c'est-à-dire que les informations sont stockées sur des machines distantes et accessibles par réseau.

L'avantage majeur de l'utilisation de bases de données est la possibilité de pouvoir être accédées par plusieurs utilisateurs simultanément [1].

Pour concevoir une base de données efficace et utile, vous devez suivre le bon processus, qui comprend les phases suivantes :

1. Analyse des besoins, c'est-à-dire l'identification de l'objet de votre base de données
2. Organisation des données en tables
3. Spécification des clés primaires et analyse des relations
4. Normalisation des tables [3].

1.5. La gestion des bases de données :

Afin de pouvoir contrôler les données ainsi que les utilisateurs, le besoin d'un système de gestion s'est vite fait ressentir. La gestion de la base de données se fait grâce à un système appelé SGBDs (système de gestion de bases de données) ou en anglais DBMS (Database management system). Le SGBDs est un ensemble de services (applications logicielles) permettant de gérer les bases de données, c'est-à-dire:

- Permettre l'accès aux données de façon simple.
- Autoriser un accès aux informations à de multiples utilisateurs.
- Manipuler les données présentes dans la base de données (insertion, suppression, modification).

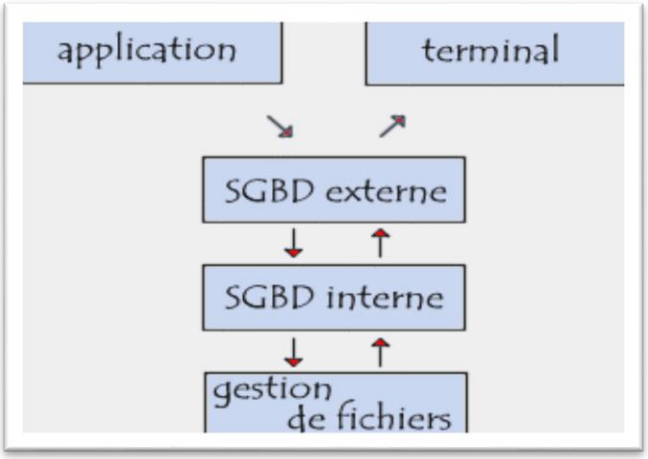


Figure 3 : Schéma La gestion des bases de données

Le SGBDs peut se décomposer en trois sous-systèmes :

- Le système de gestion de fichiers : il permet le stockage des informations sur un support physique
- Le SGBDs interne : il gère l'ordonnancement des informations
- Le SGBDs externe : il représente l'interface avec l'utilisateur [1].

1.6. Les principaux SGBDs:

Les principaux systèmes de gestion de bases de données sont les suivants :

- Borland Paradox
- Filemaker
- IBM DB2
- Ingres
- Interbase
- Microsoft SQL server
- Microsoft Access
- Microsoft FoxPro
- Oracle
- Sybase
- MySQL
- PostgreSQL
- mSQL

- SQL Server 11[1].

2. Système de gestion de base de données :

2.1. Dispositif de stockage:

Une base de données est constituée, matériellement d'un ou plusieurs fichiers stockés sur un support non volatile. Le support le plus couramment employé est le disque magnétique (« disque dur ») qui présente un bon compromis en termes de capacité de stockage, de prix et de performance. Un concurrent sérieux est le Solid State Drive, dont les performances sont nettement supérieures, et le coût en baisse constante, ce qui le rend de plus en plus concurrentiel.

Il y a deux raisons principales à l'utilisation de fichiers. Tout d'abord il est possible d'avoir affaire à des bases de données dont la taille dépasse de loin celle de la mémoire principale. Ensuite – et c'est la justification principale du recours aux fichiers, même pour des bases de petite taille – une base de données doit survivre à l'arrêt de l'ordinateur qui l'héberge, que cet arrêt soit normal ou dû à un incident matériel.

Un système informatique offre plusieurs mécanismes de stockage de l'information, ou mémoires. Ces mémoires se différencient par leur prix, leur rapidité, le mode d'accès aux données (séquentiel ou par adresse) et enfin leur durabilité.

▪ Les mémoires volatiles perdent leur contenu quand le système est interrompu, soit par un arrêt volontaire, soit à cause d'une panne.

▪ Les mémoires persistantes comme les disques magnétiques, les SSD, les CD ou les bandes magnétiques, préservent leur contenu même en l'absence d'alimentation électrique [4].

2.2. Mémoires:

D'une manière générale, plus une mémoire est rapide, plus elle est chère et – conséquence directe – plus sa capacité est réduite. Les différentes mémoires utilisées par un ordinateur constituent donc une hiérarchie (Figure5), allant de la mémoire la plus petite mais la plus efficace à la mémoire la plus volumineuse mais la plus lente.

1. La mémoire cache est une mémoire intermédiaire permettant au processeur d'accéder très rapidement aux données à traiter ;

2. La mémoire vive, ou mémoire principale stocke les données et les processus constituant l'espace de travail de la machine ; toute information (donnée ou programme) doit être en mémoire principale pour pouvoir être traitée par un processeur ;
3. Les disques magnétiques constituent le principal périphérique de mémoire persistante ils offrent une grande capacité de stockage tout en gardant des accès en lecture et en écriture relativement efficaces ;
4. Les Solid State Drive ou SSD sont une alternative récente aux disques magnétiques ; leurs peformances sont supérieures, mais leur coût élevé.
5. Enfin les CD ou les bandes magnétiques sont des supports très économiques mais leur lenteur les destine plutôt aux sauvegardes à long terme.

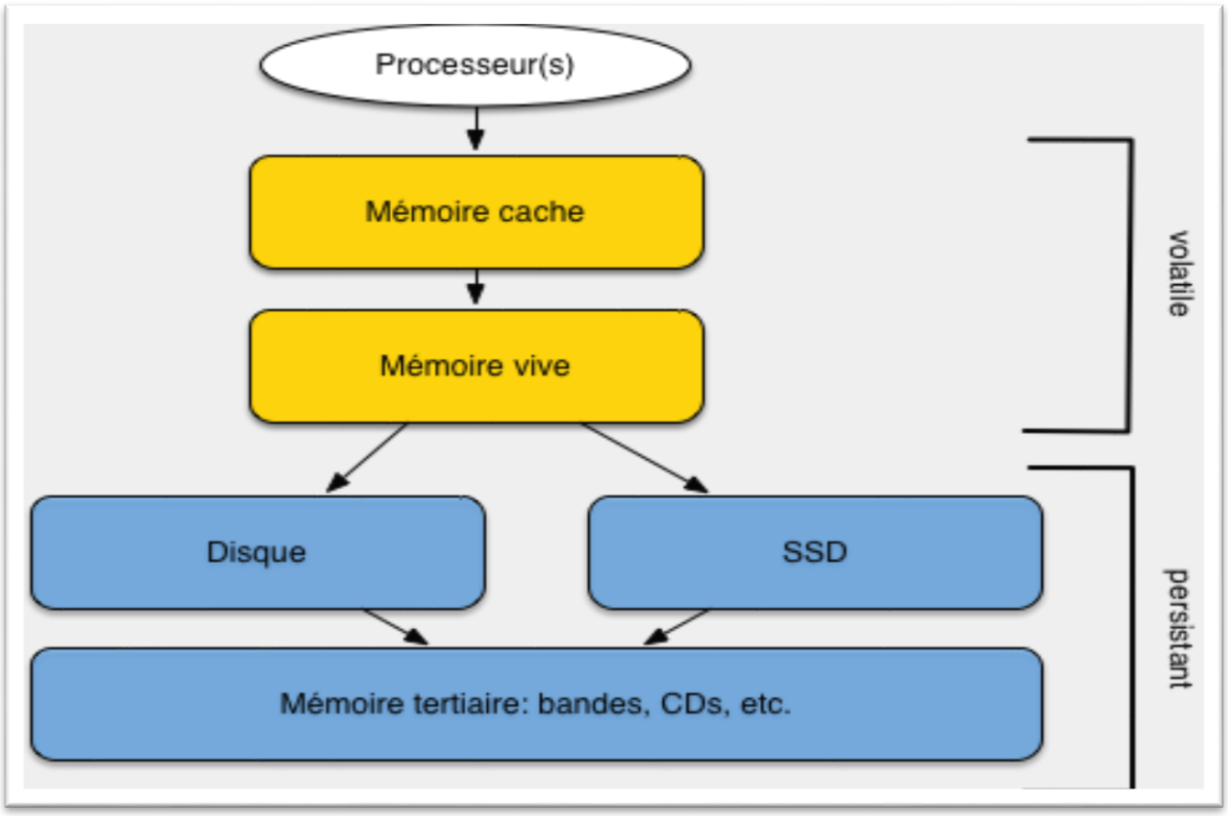


Figure 4: Hiérarchie des mémoires

2.3. Performances des mémoires:

Comment mesurer les performances d'une mémoire ? Nous retiendrons deux critères essentiels :

- **Temps d'accès :** connaissant l'adresse d'un enregistrement, quel est le temps nécessaire pour aller à l'emplacement mémoire indiqué par cette adresse et obtenir l'information ? On parle de lecture par clé ou encore d'accès direct pour cette opération ;
- **Débit :** quel est le volume de données lues par unité de temps dans le meilleur des cas?

Le premier critère est important quand on effectue des accès dits aléatoires. Ce terme indique que deux accès successifs s'effectuent à des adresses indépendantes l'une de l'autre, qui peuvent donc être très éloignées. Le second critère est important pour les accès dits séquentiels dans lesquels on lit une collection d'information, dans un ordre donné. Ces deux notions sont essentielles.

Notion : accès direct/accès séquentiel

Retenez les notions suivantes :

- Accès direct : étant donné une adresse dans la mémoire (et principalement sur un disque), on accède à la donnée stockée à cette adresse.
- Accès séquentiel : on parcours la mémoire (et principalement un disque) dans un certain ordre en lisant les enregistrements au fur et à mesure [4].

Type mémoire	Taille	Temps d'accès aléatoire	Temps d'accès séquentiel
Mémoire *cache* (Static RAM)	Quelques Mo	$\approx 10^{-8}$ (10 nanosec.)	Plusieurs dizaines de Gos par seconde
Mémoire principale (Dynamic RAM)	Quelques Go	$\approx 10^{-8} - 10^{-7}$ (10-100 nanosec.)	Quelques Go par seconde
Disque magnétique	Quelques Tos	$\approx 10^{-2}$ (10 millisec.)	Env. 100 Mo par seconde.
SSD	Quelques Tos	$\approx 10^{-4}$ (0,1 millisec.)	Jusqu'à quelques Gos par seconde.

Table 1: Performance des divers types de mémoire

2.4. Disques:

Un disque est une mémoire à accès dit semi-direct. Contrairement à une bande magnétique par exemple, il est possible d'accéder à une information située n'importe où sur le disque sans avoir à parcourir séquentiellement tout le support. Mais, contrairement à la mémoire principale, avant d'accéder à une adresse, il faut attendre un temps variable lié au mécanisme de rotation du disque.

L'accès est fondé sur une adresse donnée à chaque bloc au moment de l'initialisation du disque par le système d'exploitation. Cette adresse est composée des trois éléments suivants:

1. Le numéro du disque dans la pile ou le numéro de la surface si les disques sont à double-face ;
2. Le numéro de la piste.
3. Le numéro du bloc sur la piste [4].

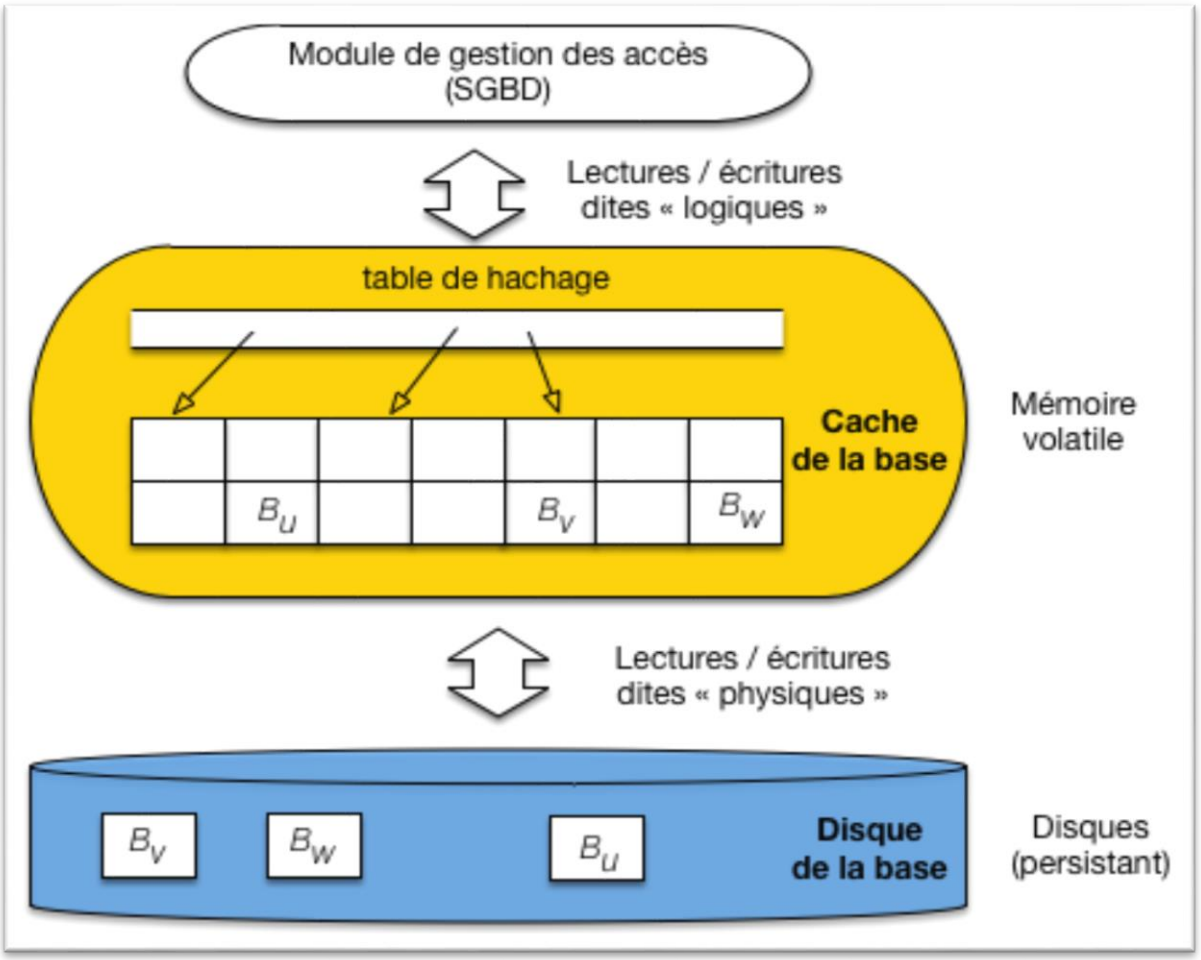


Figure 5 : Le cache et le disque, ressources mémoires allouées au SGBD

2.5. Structures d'index (le hachage):

Les tables de hachage sont des structures très couramment utilisées en mémoire centrale pour organiser des ensembles et fournir un accès performant à ses éléments. Le hachage est également utilisé par les SGBDs pour organiser de grandes collections de données sur mémoire persistante. Une technique intermédiaire, le hachage hybride, consiste à créer une seule structure de hachage dont les données sont partiellement en mémoire RAM et partiellement sur le disque.

nous étudions les structures de hachage utilisées pour indexer de grandes collections de données. La plus simple, le hachage statique ne fonctionne correctement que pour des collections de tailles fixes, ce qui exclut des tables évolutives (le cas le plus courant). Le hachage dynamique, qui s'adapte à la taille de la collection indexée, est présenté en section il repose sur un répertoire (directory) dont la taille peut croître au point de devenir un problème. Enfin la troisième section introduit le hachage linéaire, une structure qui apporte toute l'efficacité du hachage tout en maintenant une taille de répertoire réduite [5].

2.6. Recherche dans une table de hachage:

- La structure n'occupe aucun espace disque additionnel aux données elles-mêmes, contrairement à l'arbre-B.
- Elle permet d'effectuer les recherches par clé par accès direct (calculé) au fragment susceptible de contenir les enregistrements.

la recherche par préfixe n'est plus possible. La hachage ne permet pas non plus d'optimiser les recherches par intervalle, puisque l'organisation des enregistrements ne s'appuie pas sur l'ordre des clés.

La requête suivante par exemple ne peut être résolule que par le parcours de tous les blocs de la structure, même si trois films seulement sont concernés

```
select *
from table_emp
where salaire between 25000 and 30000;
```

Cette incapacité à effectuer efficacement des recherches par intervalle doit mener à préférer l'arbre-B dans tous les cas où ce type de recherche est envisgeable. Si la clé est par exemple une date, il est probable que des recherches seront effectuées sur un intervalle de temps, et l'utilisation du hachage peut s'avérer un mauvais choix. Mais dans le cas, fréquent, où on utilise une clé séquentielle pour identifier les enregistrements pas un numéro indépendant

de leurs attributs, le hachage est tout à fait approprié car une recherche par intervalle ne présente alors pas de sens et tous les accès se feront par la clé.

En résumé, les avantages et inconvénients du hachage statique, comparé à l'arbre-B, sont les suivantes :

- **Avantages** : (1) recherche par accès direct, en temps constant ; (2) n'occupe pas d'espace disque.
- **Inconvénients** : (1) pas de recherche par intervalle ; (2) pas de dynamicité [5].

3. La structure de données:

- L'application la plus intéressante pour les fonctions de hachage parfaites minimales est probablement son utilisation comme structure d'indexation pour les bases de données.
- Alors la structure de données la plus utilisée comme structure d'indexation dans les bases de données est B+tree.
- En fait, B+tree est très utilisée pour les applications dynamiques avec de fréquentes insertions et suppressions_d'enregistrements. Cependant, pour les applications avec des modifications sporadiques et un grand nombre de requêtes, B+tree. n'est pas la meilleure option, car les déploiements pratiques de cette structure sont extrêmement complexes et fonctionnent mal avec de très grands ensembles de clés tels que ceux requis pour les nouvelles applications de base de données de frontières. .

Par exemple, dans le domaine de la recherche d'informations, le travail avec d'énormes collections est une tâche quotidienne. La simple attribution d'identifiants aux pages Web d'une collection peut être une tâche difficile. Alors que les bases de données traditionnelles ne peuvent tout simplement pas gérer plus de trafic une fois que l'ensemble de travail des URL de page Web ne tient plus dans la mémoire principale, les fonctions de hachage parfait minimales peuvent facilement atteindre des centaines de millions d'entrées, en utilisant du matériel de stock [6].

4. Requêtes à intervalles (range queries):

4.1. SQL BETWEEN:

L'opérateur BETWEEN est utilisé dans une requête SQL pour sélectionner un intervalle de données dans une requête utilisant WHERE. L'intervalle peut être constitué de chaînes de

caractères, de nombres ou de dates. L'exemple le plus concrete consiste par exemple à récupérer uniquement les enregistrements entre 2 dates définies [7].

4.2. Syntaxe:

L'utilisation de la commande BETWEEN s'effectue de la manière suivante :
SELECT *FROM table WHERE nom_colonne BETWEEN 'valeur1' AND 'valeur2' [7].

5. Résolues en utilisant les B-tree (arbre-B) :

5.1. Conception logique de la base :

Il existe plusieurs organisations primaires de fichiers qui déterminent la façon dont les enregistrements sont placés physiquement sur le disque, et donc la façon dont il est possible d'accéder à ces enregistrements [8].

- Dans un fichier plat (ou fichier non ordonné)

Les enregistrements sont placés sur le disque sans observer d'ordre particulier et les nouveaux enregistrements sont simplement ajoutés en fin de fichier.

- Dans un fichier trié (ou fichier séquentiel), ils sont ordonnés en fonction de la valeur d'un champ particulier nommé clé de tri.
- Un fichier haché utilise une fonction de hachage appliquée à un champ particulier (La clé de hachage).
- Les arbres-B utilisent des structures arborescentes.
- Les organisations secondaires, ou structures d'accès auxiliaire, permettent d'accéder rapidement aux enregistrements en se basant sur d'autres champs que ceux qui ont servi pour l'organisation primaire. La plupart de ceux-ci sont des index.

5.2. Definitions Les arbres B-:

5.2.1. Qu 'est ce qu'un index B-tree ?

L'index B-tree est une arborescence ordonnée composée de nœuds d'index. Il contient une entrée par enregistrement. Il existe des index créés automatiquement pendant la création d'une contrainte de type **« PRIMARY KEY »** ou **« UNIQUE »**. En effet, ce sont des index uniques, l'unicité étant gérée par Oracle à travers des index.

Il est possible de créer un index de ce type uniquement si les valeurs du ou des champs qui composent l'index sont uniques sur l'ensemble des valeurs de la table [8].

- **Remarque** : - Un B-tree est un arbre équilibré — pas un arbre binaire.
  - Il utilisé pour les requêtes d'égalité : =, <> ;>, <,>= et <=.
  - Les valeurs des clés de l'index sont enregistrés de façon ordonnée dans une structure arborescente (sous forme d'arbre).
- **Important** : L'index B-tree permet de trouver un nœud feuille rapidement.

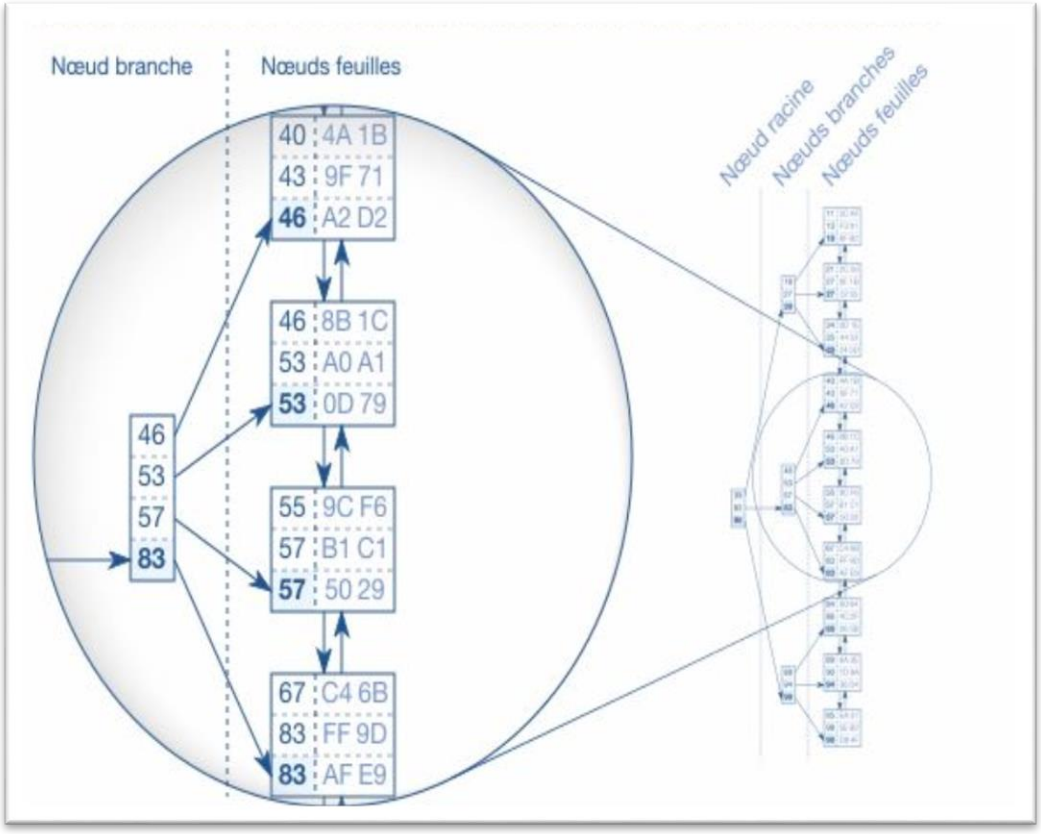


Figure 6 : la structure du B-tree

6. Fragmentation:

La fragmentation est une fonction du serveur de base de données qui vous permet de contrôler l'emplacement de stockage des données au niveau de la table. La fragmentation vous permet de définir des groupes de lignes ou des clés d'index dans une table, conformément à un algorithme ou un *s*chéma. Vous pouvez stocker chaque groupe ou fragment (également appelé

une *partition*) dans un dbspace distinct, associé à un disque physique spécifique.Utilisez des instructions SQL pour créer les fragments et attribuez-les à des dbspaces [9].

6.1. Pourquoi fragmenter?

Envisagez la fragmentation de vos tables si votre but est d'améliorer au moins un des éléments suivants :

- Temps de réponse pour un seul utilisateur
- Accès simultané
- Disponibilité
- Caractéristiques de sauvegarde-restauration
- Chargement des données [9].

6.2 Comment fragmenter?

- On distingue trois possibilités de fragmentation :

1- Fragmentation Horizontale

2- Fragmentation Verticale

3- Fragmentation Hybride

6.3. Fragmentation horizontale:

- ✓ La fragmentation horizontale concerne les données.
- ✓ Chaque fragment représente un ensemble de tuples.
- ✓ Pour fragmenter, on a besoin d'information sur la BD (schéma global…) et les applications (requêtes utilisées…).
- ✓ Les fragments sont définis par des opérations de sélection sur les relations.

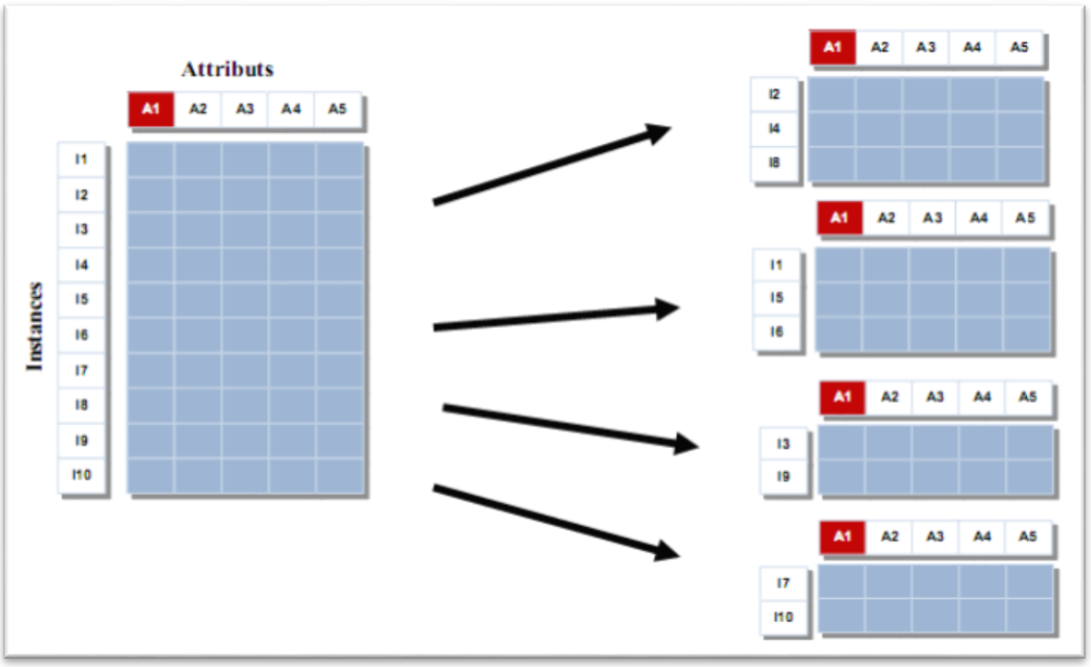


Figure 7 : Fragmentation Horizontale

- La reconstruction de la relation est définie par l'union des fragments.

Les requêtes de l'utilisateur incluent des prédicats plus complexes qui sont des combinaisons des prédicats simples. Puisqu'on peut toujours transformer une expression booléenne en une forme normale conjonctive, on se base sur les prédicats conjonctifs pour fragmenter.

6.4. Techniques de Fragmentation horizontale :

Décomposition de la table en groupes de lignes

Exemple : Table Client (NCL, Nom, Ville)

Il existe deux types de fragmentation horizontale :

- Primaire
- Dérivée [10].

6.4.1. Fragmentation horizontale primaire :

La Fragmentation horizontale est définie par l'opération de sélection

Exemple : Client (NCL, Nom, Ville) peut être fragmenté :

Client1= SELECT * FROM Client WHERE Ville = «Kinshasa»

Client2= SELECT * FROM Client WHERE Ville <> «Kinshasa»

Reconstruction de la relation initiale :

Client = Client1 ? Client2

6.4.2. Fragmentation horizontale dérivée :

La Fragmentation d'une table en fonction des fragments horizontaux d'une autre table. (Cette fragmentation est obtenue dans le cas de lien père-fils)

Exemple : Commande (NCL, N°Produit, Date, Qte, N°Représentant)

Commande 1= SELECT * FROM Commande WHERE NCL IN(SELECT NCL FROM CLIENT1)

Commande 2= SELECT * FROM Commande WHERE NCL IN (SELECT NCL FROM CLIENT2)

- Reconstruction de la relation initiale :

  Commande = Commande1 ? Commande2

6.5. Fragmentation verticale

La fragmentation verticale est obtenue par décomposition de la table en groupes de colonnes. Fragmentation verticale est définie par l'opération de projection.

Exemple : Table Client (N°Client, Nom, Sexe, Ville)

Client (N°Client, Nom, Sexe, Ville) peut être fragmenté :

Client1= SELECT N°Client, Nom FROM Client

Client2= SELECT N°Client, Sexe, Ville FROM Client

- Reconstruction de la relation initiale :

  Client = Client1 join Client2 [10].

6.6. Fragmentation mixte

La Fragmentation mixte résulte de l'application successive d'opérations de fragmentation horizontale et de fragmentation verticale [10].

7. MySQL :

Est un Système de Gestion de Base de Données (SGBD) parmi les plus populaires au monde. Il est distribué sous double licence, un licence publique générale GNU et une propriétaire selon l'utilisation qui en est faites. La première version de MySQL est apparue en 1995 et l'outil est régulièrement entretenu.

Ce système est particulièrement connu des développeurs pour faire partit des célèbres quatuors : WAMP (Windows, Apache, MySQL et PHP), LAMP (Linux) et MAMP (Mac). Ces packages sont si populaires et simples à mettre en œuvre que MySQL est largement connu et exploité comme système de gestion de base de données pour des applications utilisant PHP. C'est d'ailleurs pour cette raison que la plupart des hébergeurs web proposent PHP et MySQL [11].

7.1. Caractéristiques:

MySQL est un serveur de base de données relationnelles SQL qui fonctionne sur de nombreux systèmes d'exploitation (dont Linux, Mac OS X, Windows, Solaris, FreeBSD…) et qui est accessible en écriture par de nombreux langages de programmation, incluant notamment PHP, Java, Ruby, C, C++, .NET, Python …

L'une des spécificités de MySQL c'est qu'il inclus plusieurs moteurs de bases de données et qu'il est par ailleurs possibles au sein d'une même base de définir un moteur différent pour les tables qui composent la base. Cette technique est astucieuse et permet de mieux optimiser les performances d'une application. Les 2 moteurs les plus connus étant MyISAM (moteur par défaut) et InnoDB.

La réplication est possible avec MySQL et permet ainsi de répartir la charge sur plusieurs machines, d'optimiser les performances ou d'effectuer facilement des sauvegardes des données [11].

Conclusion :

Les bases de données sont considérées comme les éléments les plus importants dans le monde Informatique. Mais La plus importantes tâche est la conception physique des bases de données, qui inclut l'organisation des données et l'amélioration de l'accès à ces données. Pour améliorer les temps d'accès, l'administrateur emploie en général des index par le hachage parfait minimal pour rechercher rapidement les informations nécessaires à une requête sans parcourir toutes les données.

Chapitre 2 : Techniques de hachage

1. Introduction :

Dans ce chapitre, nous allons présenter une description générale du hachage et les différents techniques (hachage parfait, hachage parfait minimal) et leur fonction, à la fin nous donnons une introduction aux librairies CMPH.

2. Le hachage :

Une structure de données est un format spécial destiné à organiser, traiter, extraire et stocker des données. 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.

Le type d'une structure de données est déterminé par le type d'opération requis ou par les algorithmes qui seront appliqués. Les types possibles sont les suivants :

- Tableau : Un tableau stocke un ensemble d'éléments dans des emplacements de mémoire contigus. Les éléments de même type sont stockés ensemble afin de faciliter le calcul de leur emplacement ou leur extraction. La longueur d'un tableau peut être fixe ou variable.
- Pile : Une pile stocke un ensemble d'éléments en suivant l'ordre linéaire dans lequel les opérations sont appliquées. Par exemple, dernier entré, premier sorti (LIFO, Last In First Out) ou premier entré, premier sorti (FIFO, First In First Out).
- File : Une file stocke un ensemble d'éléments de façon similaire à une pile, mais l'ordre des opérations ne peut être que de type premier entré, premier sorti.
- Liste chaînée : 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.
- Arbre : Un arbre stocke un ensemble d'éléments sous une forme hiérarchique abstraite. Chaque nœud est relié aux autres et peut contenir plusieurs sous-valeurs appelées enfants.
- Graphe : Un graphe stocke un ensemble d'éléments de façon non linéaire. Il se compose d'un ensemble fini de nœuds, appelés sommets, et de lignes, les arêtes, qui relient les sommets entre eux. Les graphes permettent notamment de représenter des systèmes réels, comme des réseaux informatiques.
- Trie : Une trie est une structure de données qui stocke des chaînes en tant qu'éléments de données pouvant être représentés visuellement sous forme graphique.

• Table de hachage_: Une table de hachage stocke un ensemble d'éléments dans un tableau associatif qui fait correspondre des clés à des valeurs. Cette structure utilise une fonction de hachage pour convertir un indice en tableau dont les alvéoles contiennent l'élément de données souhaité [12].

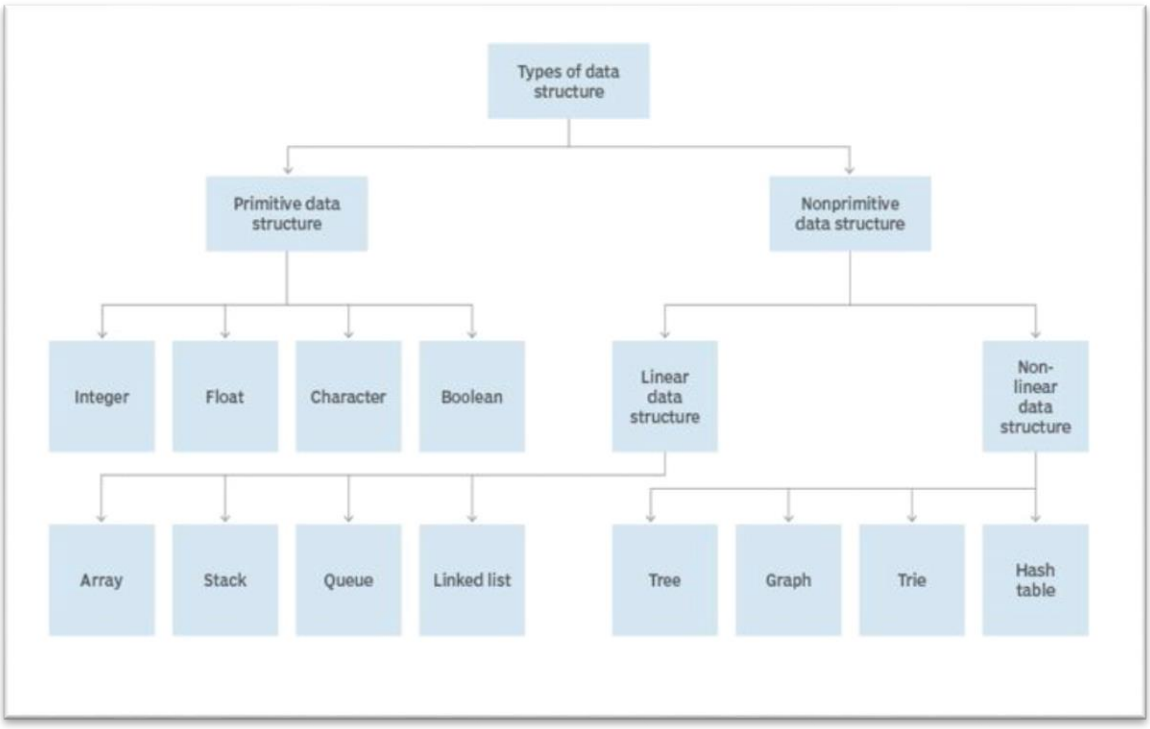


Figure 8 : Structures de données

2.1. Définition de hachage :

Le hachage est une technique bien connu qui permet d'avoir de recherche par clé ou des mise-à-jours(insertion suppression) en un temps O(1) en moyenne. Le hachage est donc avantageux par rapport aux structures d'arbres qui ont des accès en O(log(n)) pour les structures en mémoire centrale et O(log(n)/log(B)) en mémoire externe(B représente la taille du bloc de la mémoire externe). Le hachage repose essentiellement sur l'utilisation d'une fonction de hachage qui associe un nombre unique à chaque [13].

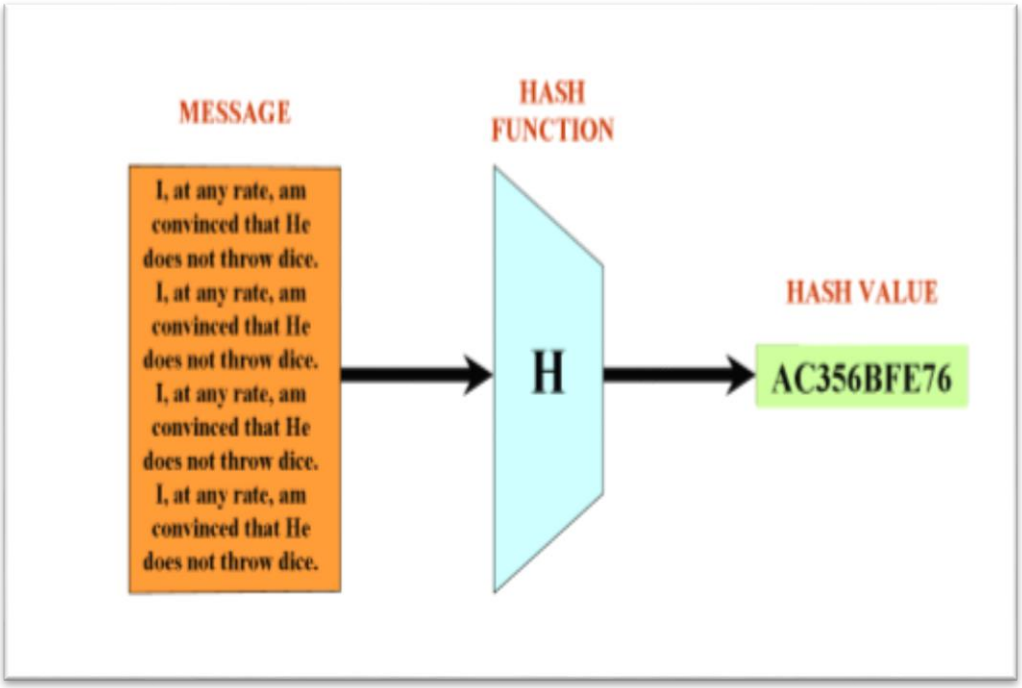


Figure 9 : Fonction de hachage cryptographique

2.2. Fonctions de hachage :

L'algorithme utilisé est appelé *fonction de hachage*, probablement parce que la valeur résultante est en quelque sorte une version « hachée » de la chaîne qu'elle représente. Cette fonction permet d'indexer la valeur d'origine (ou clé) afin de l'utiliser par la suite chaque fois que les données qui y sont associées doivent être extraites. Par conséquent, il s'agit toujours d'une opération à sens unique et il n'est pas nécessaire « d'inverser » la fonction en analysant les valeurs de hachage. Dans l'idéal, il ne devrait d'ailleurs pas être possible de déduire une fonction de hachage au moyen d'une analyse de ce type. En outre, pour être efficace, cette fonction ne doit pas produire une même valeur de hachage pour deux entrées différentes car il y aurait dans ce cas collision.

Il s'agit de trouver une fonction f telle que :

$0 <= f(K) < M$

qui réduit au maximum le nombre de collisions.

Exemple de fonction de hachage :

h(K)= K MOD M

M premier constitue généralement un bon choix.

Il existe plusieurs façons de résoudre les collisions. Les méthodes suivantes sont les plus classiques :

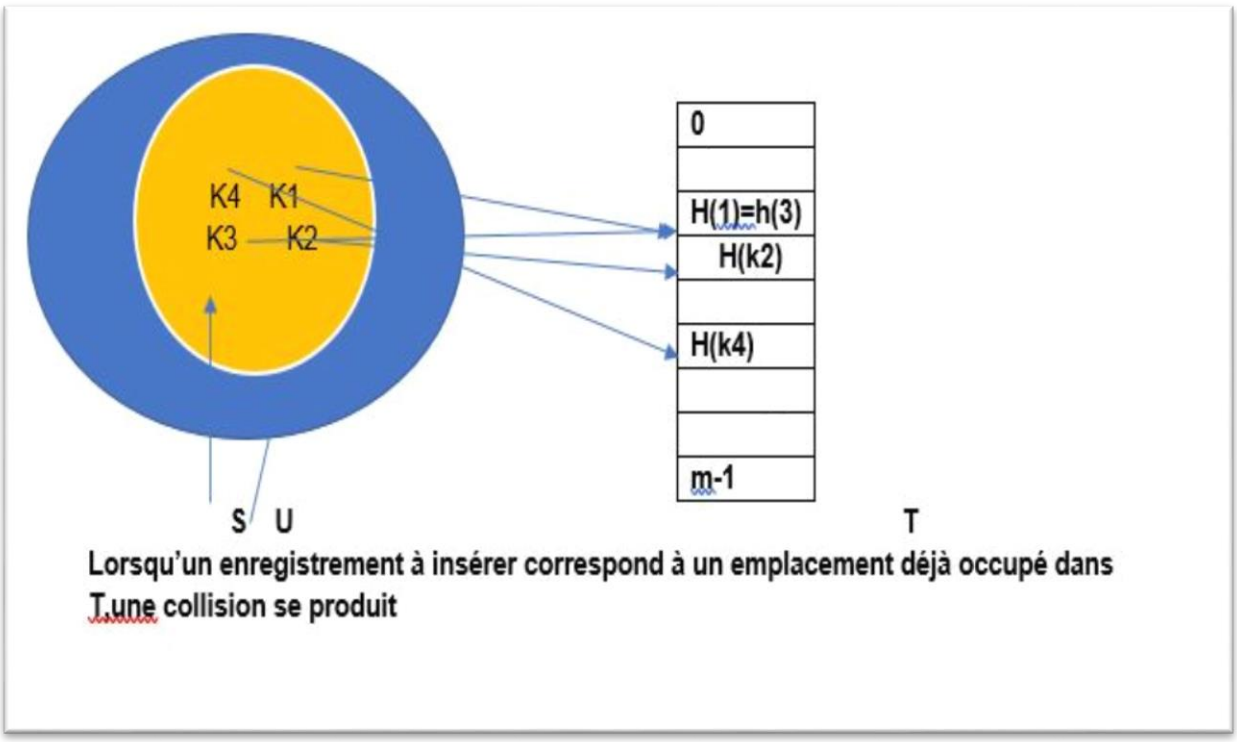


Figure 10 : Utilisation de fonction de hachage h pour mapper l'univers u de toutes les clés

- **Essai linéaire:**

S'il se produit une collision sur la case k du tableau T[0..M-1], on insère la donnée, si elle n'existe pas, dans la première case libre fournie par la séquence cyclique

h(k)-1, ..., 0, M-1, M-2, ...h(k)+1

En d'autres termes, on essaie de chercher linéairement une case libre dans la séquence ci-dessous. D'ou l'appellation de la méthode.

- **Double hachage:**

Cette méthode est presque analogue à la précédente mais au lieu que la séquence soit linéaire, elle est construite par une autre fonction de hachage. D'où l'appellation de la méthode.

Autrement dit, s'il y a collision sur une case k, on calcule un pas p par une autre fonction de hachage et la séquence cyclique à consulter serait

h(k)-p, h(k)-2p, ... Les valeurs sont calculées modulo M, la taille de la table.

Deux fonctions de hachage sont alors utilisées h(K) et h'(K).

Le choix de M est très important. Un mauvais choix peut entraîner une non-couverture de l'ensemble des adresses possibles.

On démontre que lorsque M est un nombre premier et la fonction de hachage est aléatoire, il y a couverture de l'ensemble des adresses.

- **Chaînage interne:**

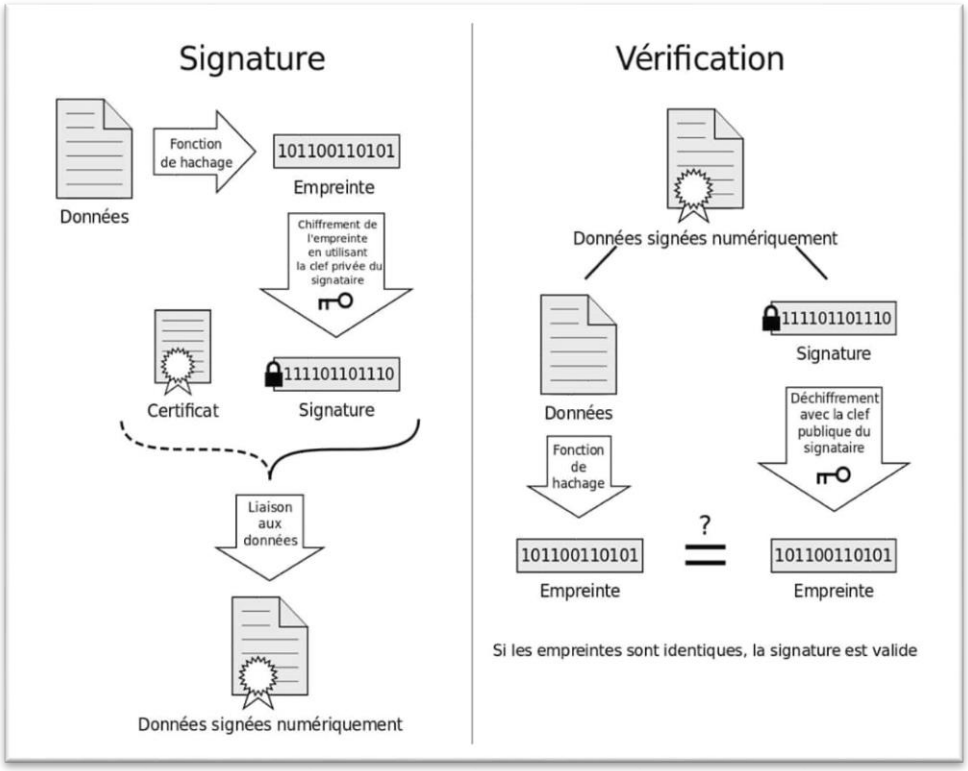
Afin de réduire le temps de recherche on ajoute au niveau de chaque élément du tableau un champ LIEN. Ce champ contient la prochaine position de la table à examiner pour la recherche d'un élément donné.
Les synonymes sont ainsi rangés dans une liste linéaire chaînée représentée dans un tableau. D'où l'appellation de la méthode.
une liste contient des groupes de synonymes.

Quand une collision se produit sur une case k, on parcourt la liste des débordements commençant en k. Si la donnée n'est pas trouvée, on cherche un emplacement vide dans le tableau.

Une bonne façon de rechercher une position vide c'est de parcourir le tableau à partir de la position R (initialisée à N+1). On décrémente R une ou plusieurs fois jusqu'à ce que l'on trouve une position vide. Ainsi, à un moment donné, toutes les positions p telles que p > R sont occupées**.**

- **Chaînage séparé:**

Les synonymes sont rangés dans une liste linéaire chaînée séparée. D'où l'appellation de la méthode.[14]



Figue 11 : Méthode de hachage

Cet algorithme à plusieurs fonctions comme :

- MD4 (Message Digest)
- MD5 (Message Digest) : produit des valeurs de hachage de 128 bits en travaillant les données originales par blocs de 512 bits.
- SHA-1 (Secure Hash Algorithm 1) : produit par contre des condensés de 160 bits en sortie et il fonctionne également à partir de blocs de 512 bits de données
- SHA-2 (Secure Hash Algorithm 2) : produit des hachés de 256,384 ou 512 bits
- RIPEMD-160 (Ripe Message Digest) : est la dernière version de l'algorithme RIPEMD. La version précédente produisait des condensés de 128 bits mais présentait des failles de sécurité importantes [15].

Exemple 1 :

➔ Parfois, Les mots de passe des utilisateurs circulent et sont stockés en clair, et sont facilement lisibles par toute personne qui accède au fichier contenant les mots de passe, c'est très mauvais pour la sécurité, heureusement c'est de moins en moins le cas [16].

Et si on représente ça par un schéma :

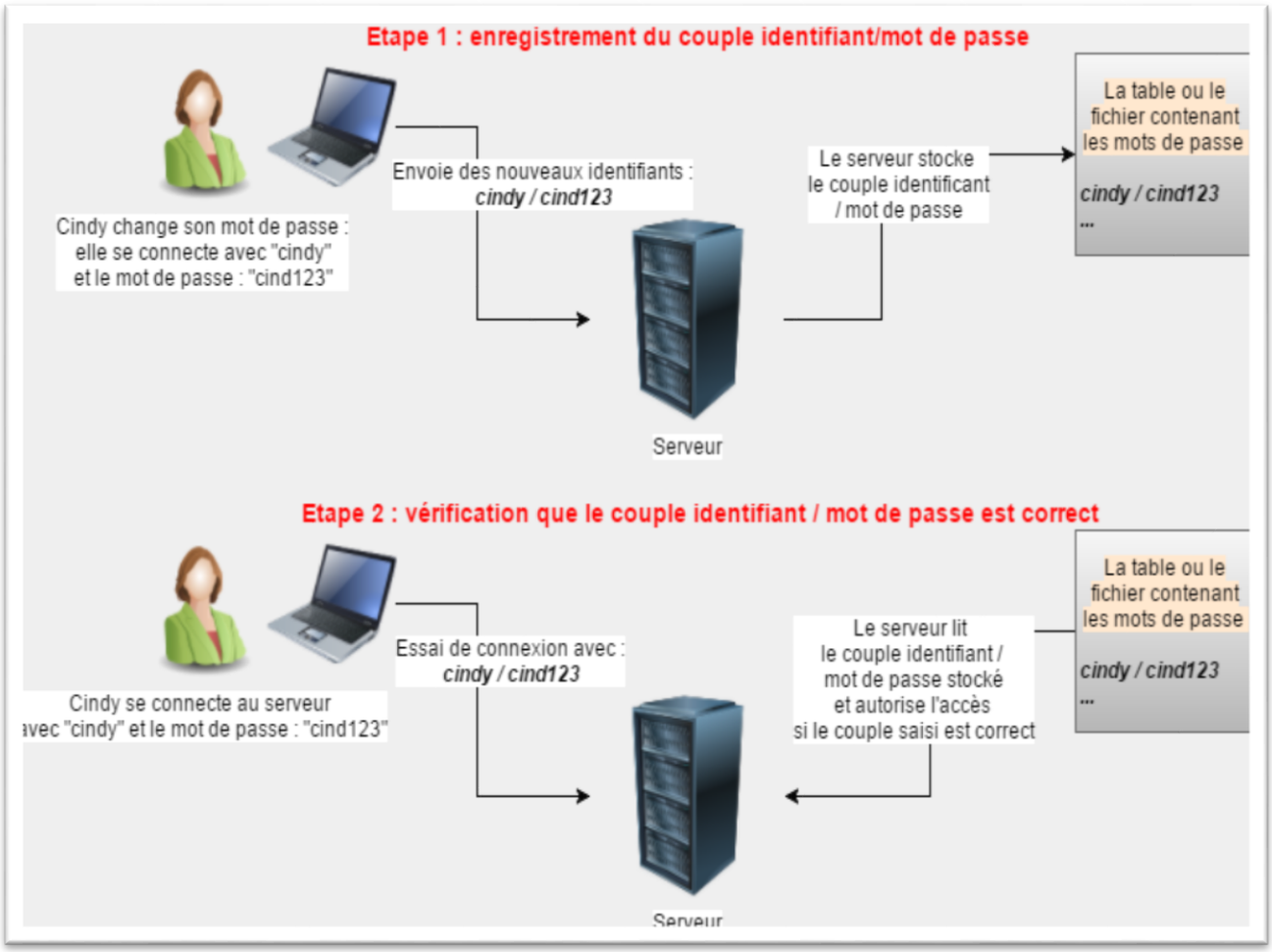


Figure 12 : Non utilisation d'une fonction hachages

Exemple 2 :

➔ L'ordinateur ne va pas envoyer le mot de passe au serveur, mais une signature du mot de passe.

➔ Le serveur ne va enregistrer le mot de passe mais enregistrera cette signature.

➔ Lorsque l'utilisateur se connectera, le serveur va vérifier que la signature du mot de passe saisi est bien la même que la signature du mot de passe enregistré [16].

Voyons ce que cela donne avec le même schéma qu'au-dessus mais avec utilisation d'une fonction de hachage.

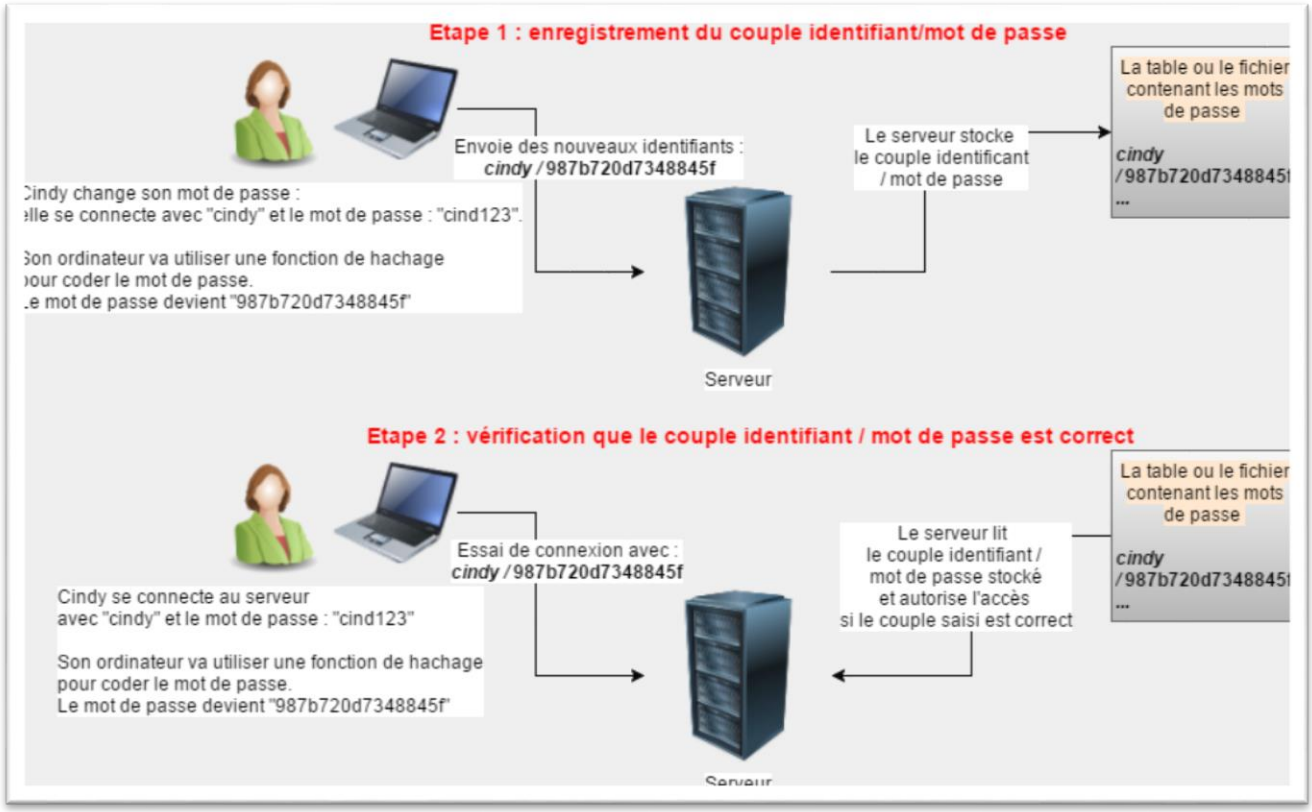


Figure 13 : Utilisation d'une fonction hachages

3. Les méthodes de résolution des collisions :

Une façon de ranger une donnée dans un tableau est de lui affecter un emplacement calculé par une fonction f. f(K) est alors l'emplacement dans le tableau où sera rangée la donnée K. Dans ce type de rangement, que ce soit pour insérer ou rechercher une donnée, on procédera toujours aussi rapidement. Cette classe d'algorithmes est appelée hachage ou technique de rangement dispersé. Le problème avec ce nouveau type de rangement est la détermination d'une fonction bijective, c'est à dire une fonction qui attribue pour chaque donnée à insérer un nouvel emplacement dans le tableau. Le fameux paradoxe de l'anniversaire, cité dans Knuth, nous laisse entendre qu'il est impossible d'essayer de chercher une fonction qui soit bijective. En d'autres termes, il existe toujours des données distinctes K1 et K2 pour lesquelles f(K1) = f(K2). Une telle situation est appelée collision. Pour utiliser une table à rangement dispersé ou, d'une façon générale les méthodes de hachage, l'utilisateur doit définir :

- la fonction de hachage

- une méthode dite méthode de résolution des collisions [14].

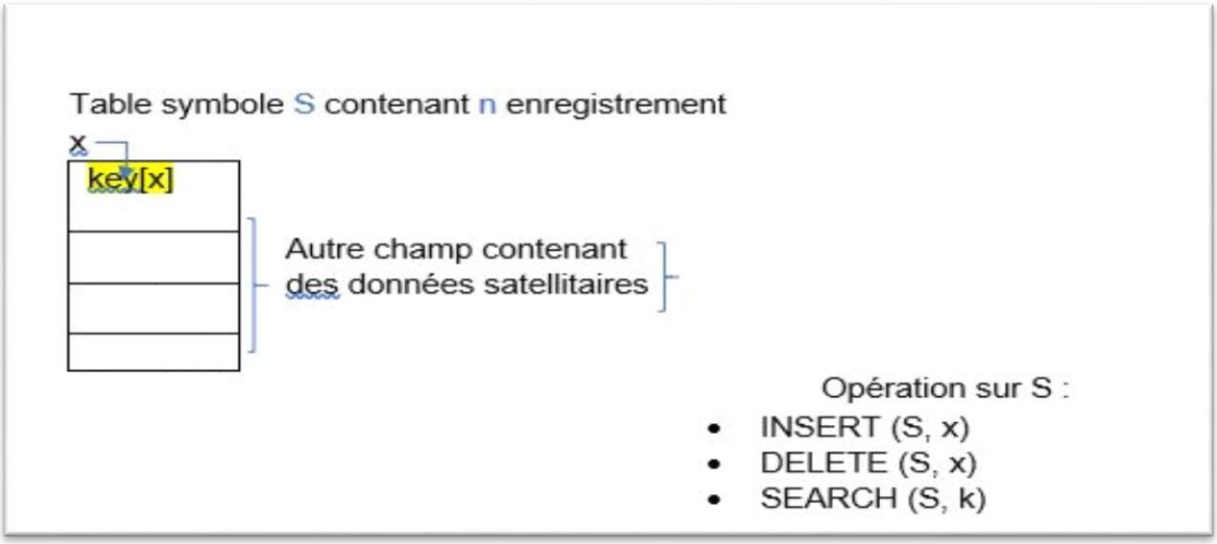


Figure 14 : Schéma Comment la structure de données S doit-elle être organisée

4. Le Hachage parfait :

Une fonction de hachage souffre le plus souvent du risque de collision. Ce dernier désigne une situation dans laquelle deux données ont un résultat identique avec la même fonction de hachage[16].

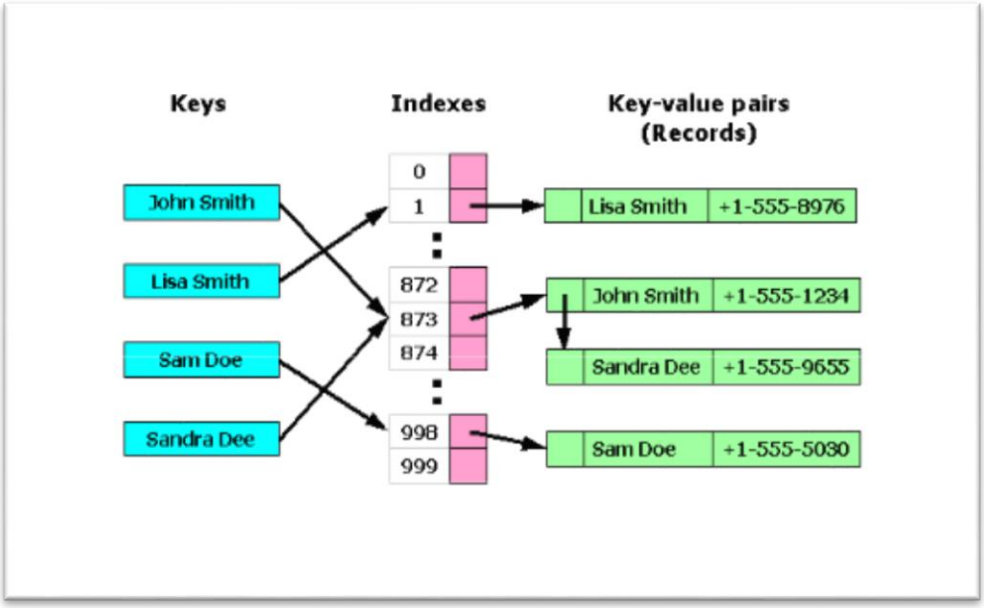


Figure 15 : Collision

➢ Bonne fonction de hachage :

- Engendre le moins de collisions
- Possède une complexité faible

➢ Le calcul du hachage se fait parfois en 2 temps :

- Une fonction de hachage particulière à l'application est utilisée pour produire un nombre entier à partir de la donnée d'origine. pp
- Ce nombre entier est converti en une position possible de la table, en général en calculant le reste modulo la taille de la table

➢ Taille de la table de hachage :

- En théorie : souvent des nombres premiers pour éviter les problèmes de diviseurs communs, qui créeraient un nombre important de collisions.
- En pratique : on utilise une puissance de deux, ce qui permet de réaliser l'opération modulo par de simples décalages, et donc de gagner en rapidité [16].

Pour éviter ce risque nous utilisons la fonction de Hachage parfait. Ce dernier à trois types : hachage parfait, hachage parfait minimal, hachage parfait monotone. Formellement :
Soit un ensemble S de n éléments (chaînes de caractères, nombres etc.…) tels que $S \subset U$.

Une fonction de hachage parfait est une fonction injective depuis l'ensemble U vers l'ensemble [0..m] ou $m \geq n$.
La fonction est dite minimale si on a $m = n$.

4.1. Hachage parfait non minimale:



Figure 16 : Fonction hachage parfaite non minimale

Le hachage parfait est un type spécial de hachage dans lequel le temps de recherche est O(1) dans le pire des cas. Nous nous intéressons uniquement au cas du hachage statique qui

est plus simple à traiter. Soit, un ensemble S de n clé(|S|=n) , on dit qu'une fonction f est une fonction de hachage parfait si cette fonction donne une valeur différente à chacune des n clés. Ces valeurs sont comprises dans un intervalle [0, m-1] ou m>=n. Ainsi donc le hachage parfait diffère du hachage traditionnel par le fait qu'il assure qu'il n'y a pas de collisions.

D'autre façon la fonction de hachage parfaite mappe un ensemble statique de n clés en un ensemble de m nombres entiers sans collisions, où m est supérieur ou égal à n. Si m est égal à n, la fonction est appelée minimale.

4.2. Hachage parfait minimal :

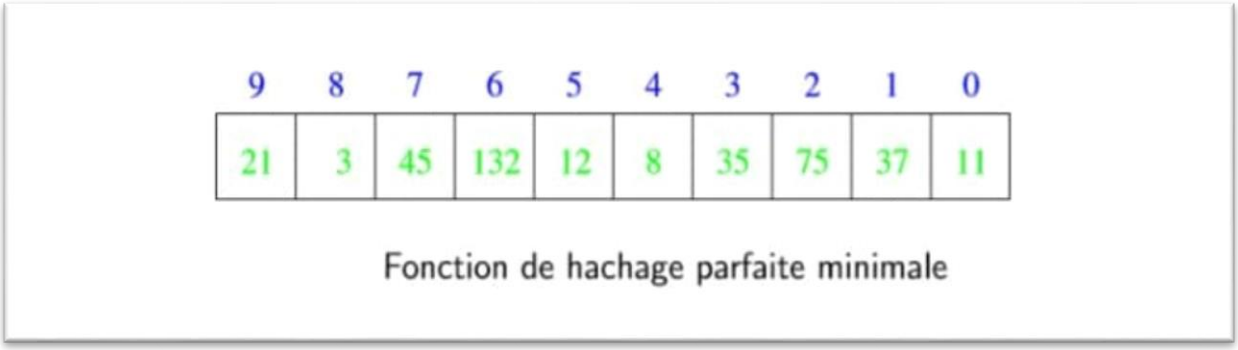


Figure 17 : Fonction hachage parfaite minimale

Une fonction de hachage parfaite minimale est une (structure de données fournissant a) une carte bijective d'un ensemble S de n clés à l'ensemble des n premiers nombres naturels. Dans le cas statique (c'est-à-dire lorsque l'ensemble S est connu à l'avance), il existe un large éventail de solutions disponibles, offrant différents compromis en termes de temps de construction, de temps d'accès et de taille de la structure de données.

Les fonctions de hachage parfaites minimales sont largement utilisées pour le stockage efficace de la mémoire et la récupération rapide d'éléments à partir d'ensembles statiques, tels que des mots en langage naturel, des mots réservés dans des langages de programmation ou des systèmes interactifs, des emplacements de ressources universelles (URL) dans les moteurs de recherche Web ou des ensembles d'éléments dans les techniques d'exploration de données. Par conséquent, il existe des applications pour des fonctions de hachage parfaites minimales dans les systèmes de recherche d'informations, les systèmes de bases de données, les systèmes de traduction de langue, les systèmes de commerce électronique, les compilateurs, les systèmes d'exploitation, entre autres.

L'utilisation de fonctions de hachage parfait minimales est, jusqu'à présent, limitée aux scénarios où l'ensemble de clés hachées est petit, en raison des limitations des algorithmes actuels. Mais dans de nombreux cas, il est crucial de gérer un ensemble énorme de clés. Ainsi, ce projet donne à la communauté du logiciel libre une API qui fonctionnera avec des ensembles de l'ordre du milliard de clés.

L'application la plus intéressante pour les fonctions de hachage parfaites minimales est probablement son utilisation comme structure d'indexation pour les bases de données. La structure de données la plus utilisée comme structure d'indexation dans les bases de données est B+tree. En fait, B+tree est très utilisée pour les applications dynamiques avec de fréquentes insertions et suppressions d'enregistrements.

4.2.1 Exemple de fonction de hachage parfaite minimale :

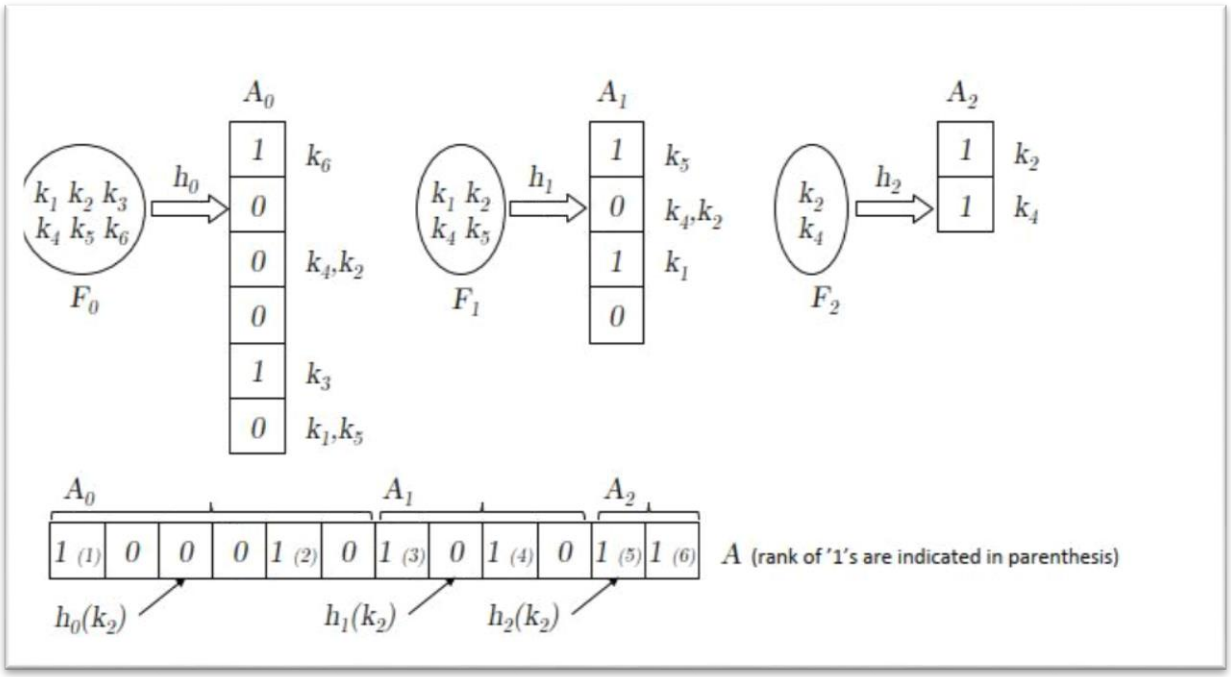


Figure 18 : Fonction de hachage parfaite minimale (MPHF)

L'entrée est un ensemble F0 composé de N = 6 touches (k1 à k6). Toutes les clés sont hachées à l'aide d'une fonction de hachage h0 et sont tentées d'être placées dans un tableau A0 aux positions données par la fonction de hachage. Les clés k3 et k6 n'ont pas de collisions dans le tableau, ainsi les bits correspondants dans A0 sont mis à "1". Les autres clés de F0

impliquées dans les collisions sont placés dans un nouvel ensemble F1. Au deuxième niveau, les clés de F1 sont hachées à l'aide d'une fonction de hachage h1.

Les clés k1 et k5 sont placées de manière unique tandis que k2 et k4 entrent en collision, elles sont donc stockées dans l'ensemble F2. Avec la fonction de hachage h2, les clés de F2 n'ont pas de collision et le processus se termine. L'opération de requête MPHF est très similaire à l'algorithme de construction. Soit A la concaténation de A0, A1, A2 (voir la partie inférieure de la figure). Pour interroger k2, la clé est d'abord hachée avec h0. Le la valeur associée dans A0 est «0», donc k2 est alors haché avec h1. La valeur associée à A1 est à nouveau "0".

Lorsqu'elle est finalement hachée avec h2, la valeur associée dans A2 est «1» et donc la requête s'arrête ici. Le index renvoyé par le MPHF est le rang de ce «1» (ici, 5) dans A. Dans cet exemple, les valeurs MPHF renvoyé lors de l'interrogation de k1, k2, k3, k4, k5 et k6 sont respectivement 4,5,2,6,3 et 1. taille | F0 | est créé de telle sorte qu'il y ait un 1 à la position i si et seulement si exactement un élément de F0 a une valeur de hachage de i. On dit qu'il y a collision chaque fois que deux clés de F0 ont le même valeur de hachage. Les clés de F0 impliquées dans une collision sont insérées dans un nouvel ensemble F1. Le processus se répète avec F1 et une nouvelle fonction de hachage h1. Un nouveau tableau de bits A1 de taille | F1 | est créé en utilisant la même procédure que pour A0 (sauf que F1 est utilisé à la place de F0, et h1 au lieu de h0).

Le processus est répété avec F2, F3,. . . jusqu'à ce que l'un de ces ensembles, Flast + 1, soit vide. On obtient un MPHF en concaténant les tableaux de bits A0, A1,. . . , Alast dans un tableau A. Pour effectuer une requête, une clé est hachée successivement avec les fonctions de hachage h0, h1,. . . aussi longtemps que la valeur en Ai ($i \geq 0$) à la position donnée par la fonction de hachage hi est 0. Finalement, par construction, nous atteignons un 1 à une certaine position de A pour un certain i = d. Nous disons que le niveau de la clé est d. L'indice renvoyé par le MPHF est le rang de celui-ci en A [17].

5. Quelle la différence entre le hachage parfait et le hachage traditionnel ?

La différence fondamentale entre le hachage parfait et le hachage traditionnel est que la fonction elle-même n'est pas fixe. Cette fonction est plutôt construite à partir des n clés. L'espace occupé par la fonction dépend en fait des deux paramètres n et m :

Le nombre de clés et le nombre de cases. Si n=m, on dit que la fonction de hachage est minimale. L'espace nécessaire pour encoder une fonction de hachage parfait minimale est au

minimum de 1.44n bits [rajoutez une citation]. Ainsi le hachage parfait est plus considéré comme une structure de données car il occupe un espace proportionnel à n.

6. La technique de résolution de requêtes à intervalles en temps constant en se basant sur le hachage :

Il y a plusieurs requêtes à intervalles en temps parmi ses requêtes Jointures qui permet de construire une nouvelle relation T à partir de deux relations R et S en joignant deux à les paires de tuples $<t_i,t_j>$ avec t_i appartient à R et t_i appartient à S tels que les deux tuples vérifie une certaine condition sur un attribut donné.

7. L'index :

7.1. Définition d'index :

Un index désigne dans le monde informatique une structure chargée d'ordonner et de trier des données afin de pouvoir les retrouver plus rapidement. La mission d'un index s'apparente ainsi à une opération de tri et de recherche de données, via un système de gestion de base de données (SGBD) [18].

7.2. Intérêt des index :

Vous devriez avoir compris maintenant que tout l'intérêt des index est d'accélérer les requêtes qui utilisent des colonnes indexées comme critères de recherche. Par conséquent, si vous savez que, dans votre application, vous ferez énormément de recherches sur la colonne *X*, ajoutez donc un index sur cette colonne, vous ne vous en porterez que mieux. Les index permettent aussi d'assurer l'intégrité des données de la base. Pour cela, il existe plusieurs types d'index différents, et deux types de "clés". Lorsque je parle de garantir l'intégrité de vos données, cela signifie garantir la qualité de vos données, vous assurer que vos données ont du sens. Grâce aux clés et aux index, vous pouvez par exemple avoir la garantie que tous les clients auxquels vous faites référence dans la table Commande existent bien dans la table Client [19].

7.3. Désavantages :

Si tout ce que fait un index, c'est accélérer les requêtes utilisant les critères de recherche correspondants, autant en mettre partout et en profiter à chaque requête ! Sauf qu'évidemment, ce n'est pas si simple. Les index ont deux inconvénients.

- Ils prennent de la place en mémoire.
- Ils ralentissent les requêtes d'insertion, modification et suppression, puisqu'à chaque fois, il faut remettre l'index à jour en plus de la table.

Par conséquent, n'ajoutez pas d'index lorsque ce n'est pas vraiment utile [19].

7.4.Comparaison entre index arbre-B et Bitmap :

7.4.1 Index Bitmap :

Un index bitmap est une structure de données définie dans un SGBD, utilisée pour optimiser l'accès aux données dans les bases de données. C'est un type d'indexation qui est particulièrement intéressant et performant dans le cadre des requêtes de sélection. L'index bitmap d'un attribut est codé sur des bits, d'où son faible coût en terme d'espace occupé. Toutes les valeurs possibles de l'attribut sont considérées, que la valeur soit présente ou non dans la table. A chacune de ces valeurs correspond un tableau de bits, appelé bitmap, qui contient autant de bits que de n-uplets présents dans la table. Ainsi, ce type d'index est très efficace lorsque les attributs ont un faible nombre de valeurs distinctes. Chaque bit représente donc la valeur d'un attribut, pour un n-uplet donné. Pour chacun des bits, il y a un codage de présence/absence (1/0), ce qui traduit le fait qu'un n-uplet présente ou non la valeur caractérisée par le bitmap.

Les index bitmap possèdent une propriété très intéressante qui consiste à répondre à certains types de requêtes sans retourner aux données elles-mêmes, optimisant ainsi les temps de réponse. Cela est possible grâce aux opérations de comptage (COUNT) et aux opérateurs logiques (AND, OR, etc) qui agissent "bit à bit" sur les bitmaps [20].

7.4.2 Index B-Arbre :

L'index B-arbre stocke les pointeurs d'index et les valeurs à d'autres nœuds d'index en utilisant une structure d'arbre récursive.

Les données sont facilement relevées par les traces des pointeurs. Le plus haut niveau de l'index est appelé racine pendant que le niveau le plus bas on l'appelle nœud de feuille ou «leaf node».Tous les autres niveaux entre eux sont appelés des branches ou nœuds internes. Toutes les racines et les branches contiennent des entrées qui pointent vers le niveau suivant de

l'index Nœuds feuilles se composent de la clé d'index et des pointeurs pointant vers l'emplacement physique des enregistrements. La structure B-Arbre est utilisée par le serveur de base de données pour configurer l'index[20].

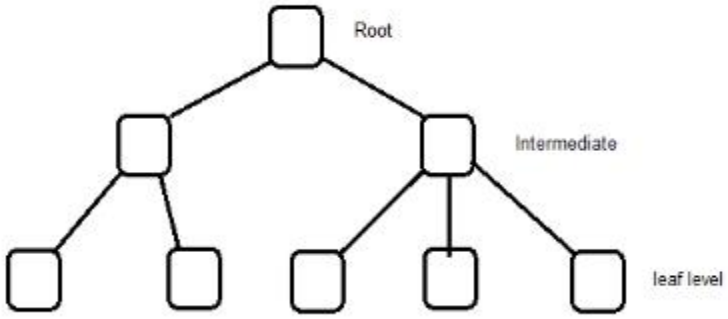


Figure 19 : La structure B-Arbre

Root ou racine : plus haut niveau de l'index pointe vers les niveaux suivants des nodes branches.

Nœuds intermédiaires ou branches : contiennent des pointeurs vers le niveau suivant des branches ou vers les nœuds de feuilles.

Nœud de feuilles ou leaf nodes : le plus bas niveau de l'index pointe vers d'autres nœuds de feuilles

8. Introduction aux CMPH - C Minimal Perfect Hashing Library :

Comme il existe de nombreuses applications pour des fonctions de hachage parfaites minimales, il est important d'implémenter des algorithmes efficaces en termes de mémoire et de temps pour construire de telles fonctions. Le manque de bibliothèques similaires dans le monde du logiciel libre a été la principale motivation pour créer la bibliothèque de hachage C Minimal Perfect[6].

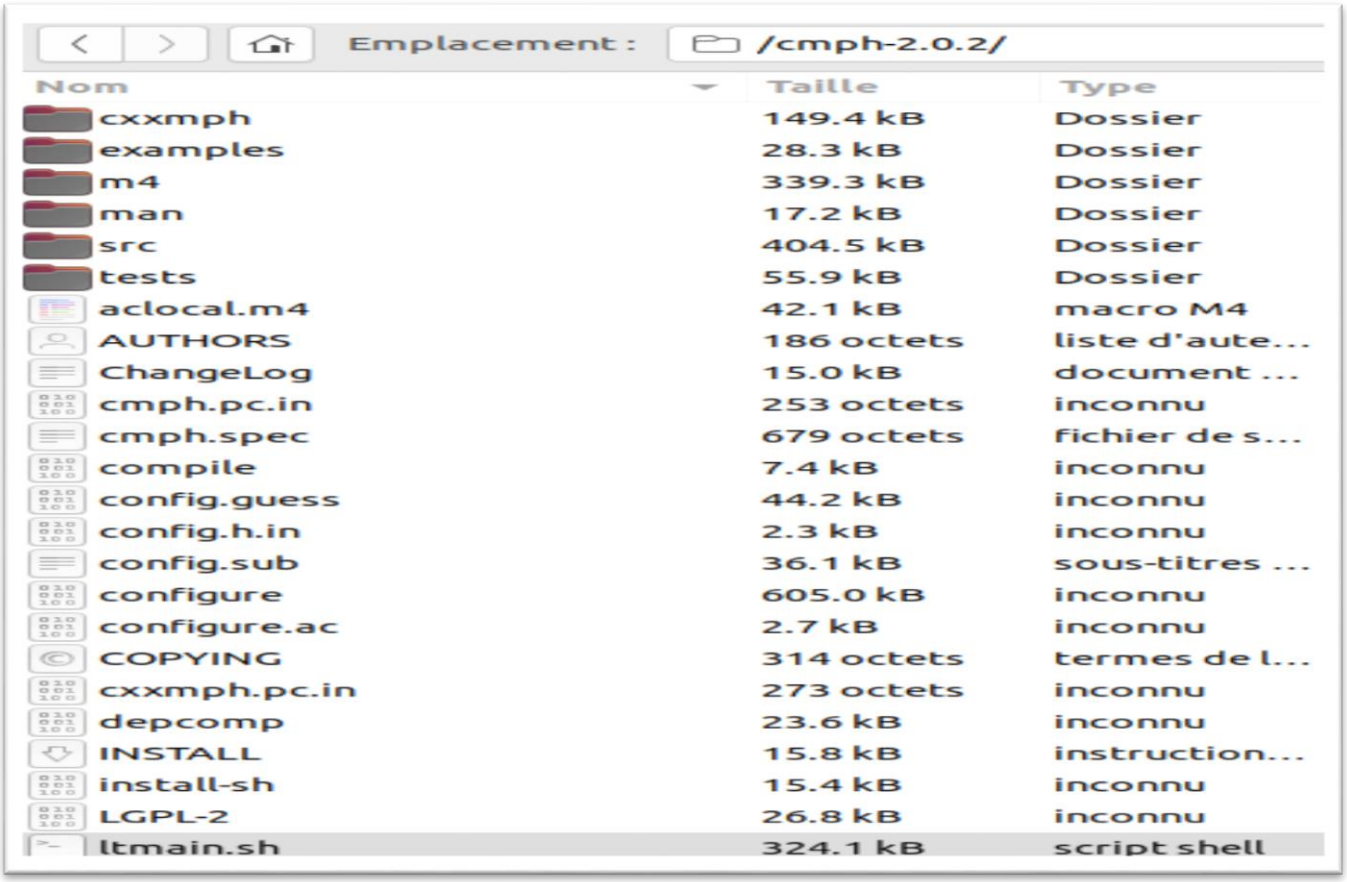


Figure 20 : Bibliothèque CMPH

8.1. Définition CMPH :

CMPH (C Minimal Perfect Hashing Library) : est une bibliothèque portable LGPL pour générer et travailler avec des fonctions de hachage parfait minimales très efficaces [6].

La bibliothèque CMPH encapsule les algorithmes les plus récents et les plus efficaces dans une API rapide, facile à utiliser et de qualité production. La bibliothèque a été conçue pour fonctionner avec de grandes entrées qui ne peuvent pas tenir dans la mémoire principale. Il a été utilisé avec succès pour construire des fonctions de hachage parfaites minimales pour des ensembles de plus de 100 millions de clés, et nous avons l'intention d'étendre ce nombre à l'ordre du milliard de clés.

8.1.1. Caractéristiques de la bibliothèque CMPH :

- Peu encombrant avec une utilisation de la mémoire principale soigneusement documentée.
- Les meilleurs algorithmes modernes sont disponibles
- Fonctionne avec les jeux de clés sur disque grâce à l'utilisation du modèle d'adaptateur.
- Sérialisation des fonctions de hachage.
- Code C portable (fonctionne actuellement sur GNU / Linux et WIN32 et est signalé comme fonctionnant sous OpenBSD et Solaris).
- Implémentation orientée objet.
- Facilement extensible.
- API bien encapsulée visant la compatibilité binaire via les versions.
- Vite[6].

8.2. Algorithmes pris en charge :

✓ Algorithme CHD :

- C'est l'algorithme le plus rapide pour construire des PHF et des MPHF en temps linéaire.
- Il peut générer des PHF avec un facteur de charge jusqu'à 99%
- C'est un schéma à deux niveaux. Il utilise une fonction de hachage de premier niveau pour diviser le jeu de clés en seaux de taille moyenne déterminée. Au

Deuxième niveau, il utilise des valeurs de déplacement pour résoudre les collisions qui ont donné naissance aux seaux.

- C'est un schéma à deux niveaux. Il utilise une fonction de hachage de premier niveau pour diviser le jeu de clés en seaux de taille moyenne déterminée par un paramètre b dans la plage [1,32]. Au deuxième niveau, il utilise des valeurs de déplacement pour résoudre les collisions qui ont donné naissance aux seaux.
- Pour un facteur de charge égal au maximum atteint par l'algorithme BDZ (81%), les PHF résultants sont stockés dans environ 1,40 bits par clé[6].

✓ Algorithme BDZ :

- C'est très simple et efficace. Il surpasse tous ceux ci-dessous. Il construit à la fois des PHF et des MPHF en temps linéaire.

Le facteur de charge maximal que l'on peut atteindre pour un PHF est de 1 / 1,23.

- Il est basé sur des 3 graphiques aléatoires acycliques. Un 3-graphe est une généralisation d'un graphe où chaque arête relie 3 sommets au lieu de seulement 2.
- Les MPHF qui en résultent ne sont pas des conservateurs d'ordre.
- Les MPHF résultants ne peuvent être stockés que dans (2 + x) bits cn, où c doit être supérieur ou égal à 1,23 et x est une constante supérieure à 0 (en fait, x = 1 / b et b est un paramètre qui doit être plus grand que 2). Pour c = 1,23 et b = 8, les fonctions résultantes sont stockées dans environ 2,6 bits par clé.
- Pour son facteur de charge maximal (81%), les PHF résultants sont stockés dans environ 1,95 bits par clé[6].

✓ Algorithme BMZ :

- Construisez des MPHF en temps linéaire.
- Il est basé sur des graphes aléatoires cycliques. Cela le rend plus rapide que l'algorithme CHM.
- Les MPHF qui en résultent ne sont pas des conservateurs d'ordre.
- Les MPHF résultants sont plus compacts que ceux générés par l'algorithme CHM et peuvent être stockés dans 4cn octets, où c'est dans la plage [0,93,1,15] [6].

✓ Algorithme BRZ :

- Un algorithme basé sur une mémoire externe très rapide pour construire des fonctions de hachage parfaites minimales pour des ensembles de l'ordre de milliards de clés.
- Cela fonctionne en temps linéaire.
- Les MPHF qui en résultent ne sont pas des conservateurs d'ordre.
- Les MPHF résultants peuvent être stockés en utilisant moins de 8,0 bits par clé[6].

✓ Algorithme CHM :

- Construisez des MPHF minimales en temps linéaire.
- Il est basé sur des graphes aléatoires acycliques,
- Les MPHF qui en résultent préservent l'ordre.
- Les MPHF résultants sont stockés dans 4cn octets, où c'est supérieur à 2.
- Les MPHF qui en résultent préservent l'ordre.
- Les MPHF résultants sont stockés dans 4cn octets, où c'est supérieur à 2[6].

✓ Algorithme FCH :

- Construisez des fonctions de hachage parfait minimales qui nécessitent moins de 4 bits par clé pour être stockées.
- Les MPHF résultants sont très compacts et très efficaces au moment de l'évaluation,
- L'algorithme n'est efficace que pour les petits ensembles.
- Il est utilisé comme algorithme interne dans l'algorithme BRZ pour résoudre efficacement des problèmes plus importants et même ainsi pour générer des MPHF qui nécessitent environ 4,1 bits par clé pour être stockés. Pour cela, il vous suffit de définir les paramètres -a sur brz et -c sur une valeur supérieure ou égale à 2.6 [6].

Conclusion :

Grâce à la conception du système que nous avons fait, nous concluons que hachage parfait minimal nous pouvons travailler plus facilement pour obtenir des résultats précis pour nous permettre de connaître les plus importants positifs de hachage parfait minimal et nous l'avons étudié à travers plusieurs programmes pour voir ses résultats que nous verrons dans le chapitre suivant.

Chapitre 3 :
Protocole d'expérimentation

1. Introduction

Ce chapitre montre comment améliorer les performances de requêtes grâce à une utilisation judicieuse du partitionnement de tables. Tous les exemples se basent sur MySQL.

Nous allons nous intéresser au partitionnement dans le domaine des bases de données, et plus exactement au partitionnement de tables. Enfin nous présenter notre approche proposée une et expliquez chaque étape en détail.

2. Définitions 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 [21].

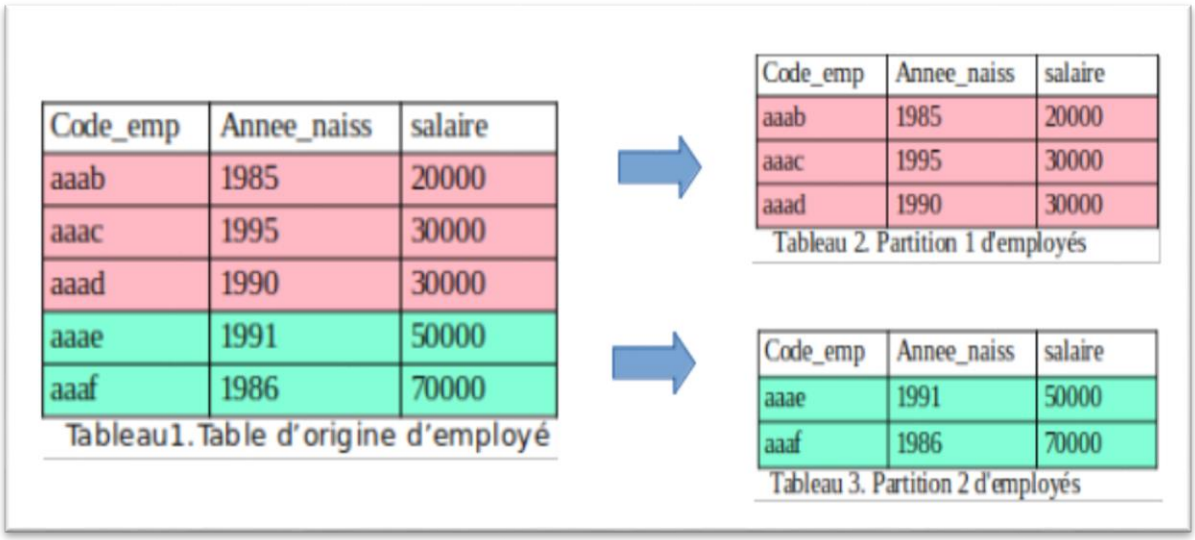


Figure 21 : Représente le partitionnement horizontal

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.

2.1. Les avantages du partitionnement :

Le partitionnement apporte plusieurs avantages à un administrateur de base de données. Voici les principaux intérêts du partitionnement :

- 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 (partitions, disques, serveurs…) différents ;
- 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. Enfin, notons que le partitionnement de tables est généralement utilisé avec des bases de données réparties où les données sont placées sur des sites géographiques (éloignés ou proches) différents. Ce chapitre se limite au partitionnement sur site local, mais les principes évoqués sont valides pour le partitionnement distant [22].

2.2. Le partitionnement horizontal avec MySQL

Commençons par voir le partitionnement horizontal, qui pour rappel, consiste à répartir les enregistrements d'une table dans différentes partitions. Pour déterminer à quelle partition appartient chaque enregistrement, il faut définir un critère de partitionnement.

2.2.1. Les méthodes de partitionnement de MySQL

Actuellement, MySQL propose quatre moyens d'exprimer ce critère et donc quatre types de partitionnement horizontal [22].

- **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.

Exemple : soit la table employé (code_emp, annee_naiss, salaire) et un partitionnement en fonction de la valeur de la colonne salaire. Nous décidons de définir trois partitions :

- Partition 1 : salaire < 25000 ;
- Partition 2 : 25000 <= salaire < 30000 ;
- Partition 3 : salaire >= 30000.

Les enregistrements suivants seront répartis ainsi :

(aaaab,1965,20000) ---→partition0.

(aabbs,1990,30000) ---→partition1.

- **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.

Exemple : soit la table employé (id, annee_naiss, service, salaire) et un partitionnement en fonction de la valeur de la colonne service. Nous décidons de définir deux partitions :

Partition 1 : service = {'Administratif', 'RHC'}

Partition 2 : service = {'Technique', 'Informatique'}

Les enregistrements suivants seront répartis ainsi :

(aaaab,1965, Administratif,20000) →partition0.

(aaaab,1965, Technique,20000) →partition3.

- **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.
- **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 [22].

3. Description de l'approche :

Notre travail est devisé en trois étapes principales :

I. Creation des 10 tables et table origine.

II. Remplissage des 10 tables et table origine.

III. Créer l'index en utilisant la librairie cmph, et cet index est un tableau de dimension 100000.

IV. Créer 1000 code_emp aléatoires. Il commence par générer 1000 nombres aléatoires entre 0 et 99999.

V. Former 1000 requêtes à partir des 1000 code_emp aléatoires et pour chaque code aléatoire il construit une requête pour rechercher ce code dans les 10 tables.

VI. Former 1000 requêtes à partir des 1000 code_emp aléatoires et pour chaque code aléatoire il construit une requête pour rechercher ce code dans la table origine.

VII. Charger l'index qui était sur le disque et de charger les 1000 code_emp stockés dans le fichier « mes_codes_aleatoires.txt ».Et ensuite il permet de former 1000 requêtes à partir des 1000 code_emp aléatoires et pour chaque code aléatoire code_emp il applique la fonction de hachage parfait minimal, et il obtient une valeur de hachage qui représente le numéro de case dans l'index(qui est un tableau) .et dans cette case de l'index il trouve le numéro de partition dans laquelle il va chercher ce code_emp, et donc il construit une requête pour rechercher ce code_emp dans cette partition , et il fait le même travail pour tous les code_emp aléatoires.

4. Plan global du travail :

Dans la figure suivante, nous expliquons en général les étapes principales de l'approche proposée pour notre projet :

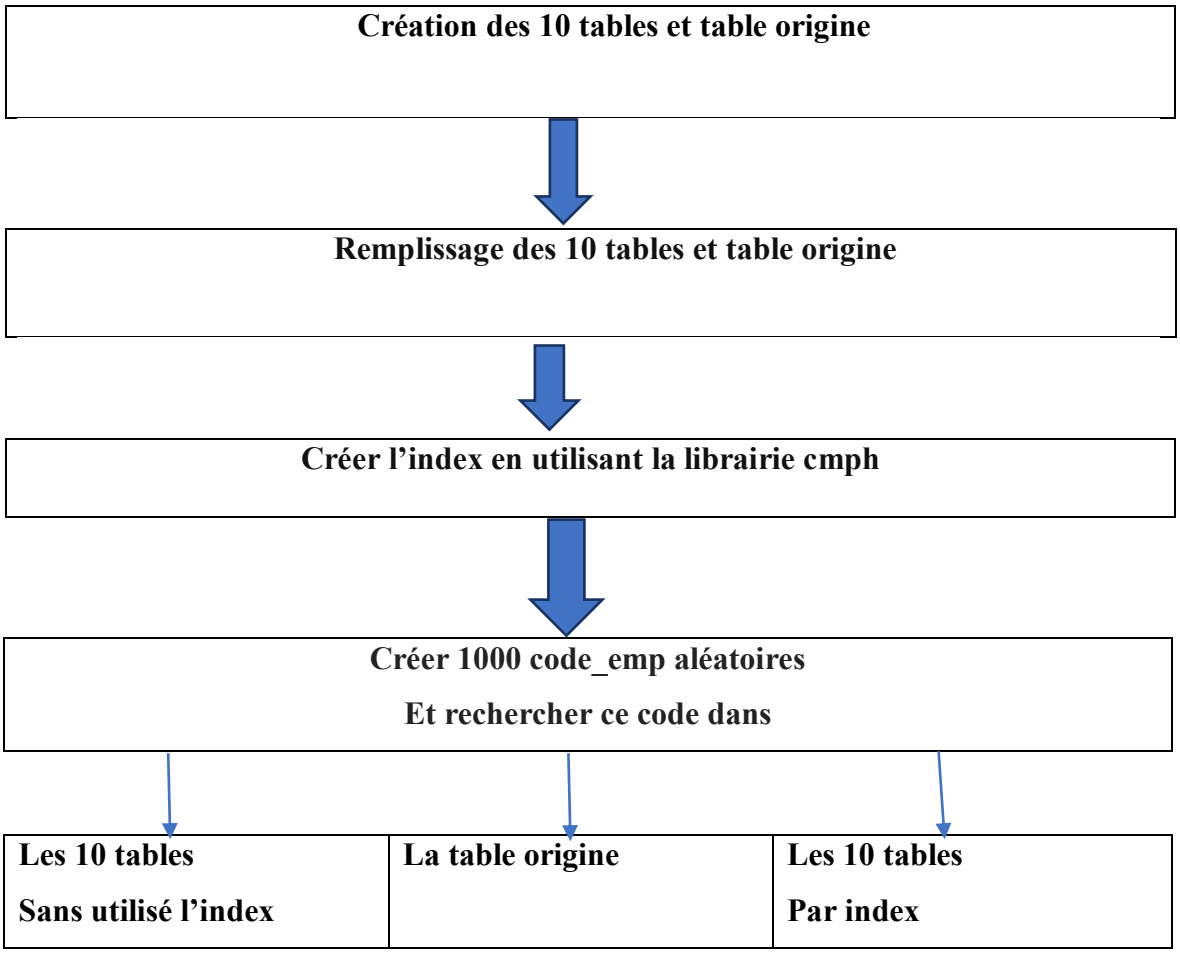


Figure 22 : Plan global de l'approche proposée

4.1. Plan détaillé du travail :

Nous avons mentionné précédemment que notre projet contient trois étapes principales. Afin de nous bien expliquer le travail à faire et l'utilité du hachage parfait minimal, et de comprendre ce que vous allez commencer à implémenter pour la première partie, Maintenant nous allons les détaillées par la suite :

La base de données : son nom est : « base_de_donnee_emp », elle contient 11 tables.

1. Creation La table origine :

Soit une table « **table_origine** » qui contient 100000 enregistrements d'employés, chaque enregistrement contient 3 champs :

code_emp : qui est le code de l'employé et qui repésente la clé primaire.

- Type : VARCHAR
- Taille : 5

annee_naiss: qui représente l'âge de l'employé, cet âge varie entre 1965 et 1995, et c'est un nombre aléatoire qui varie entre 1965 et 1995.

- Type : INT

salaire : qui représente le salaire de l'employé dans l'intervalle 20000 DA et 70000 DA. Ces partitions sont obtenues en appliquant le partitionnement horizontal sur la table d'origine.

- Type : INT

2. Création des 10 partitions(les 10 tables) :

La **table origine** contient 100000 enregistrements dans ces cas à chaque fois qu'on a une requête sur le champ « **salaire** » on doit parcourir la table « **table_origine** » et on s'arrête lorsqu'on a trouvé la réponse, à la requête. Par exemple :

SELECT **code_emp** FROM **table_origine** WHERE salaire =55000.

Donc logiquement afin de réduire le temps des opérations sur la table, on va appliquer le partitionnement horizontal sur la table « **table_origine** » afin de créer 10 partitions :

- La partition 0 : contient les enregistrements des employés dont le salaire appartient à L'intervalle de : 20000 DA à 25000 DA.
- La partition 1 : contient les enregistrements des employés dont le salaire appartient à L'intervalle de : 25000 DA à 30000 DA.
- La partition 2 : contient les enregistrements des employés dont le salaire appartient à L'intervalle de : 30000 DA à 35000 DA.

- La partition 3 : contient les enregistrements des employés dont le salaire appartient à l'intervalle de : 35000 DA à 40000 DA.
- La partition 4 : contient les enregistrements des employés dont le salaire appartient à l'intervalle de : 40000 DA à 45000 DA.
- La partition 5 : contient les enregistrements des employés dont le salaire appartient à l'intervalle de : 45000 DA à 50000 DA.
- La partition 6 : contient les enregistrements des employés dont le salaire appartient à l'intervalle de : 50000 DA à 55000 DA.
- La partition 7 : contient les enregistrements des employés dont le salaire appartient à l'intervalle de : 55000 DA à 60000 DA.
- La partition 8 : contient les enregistrements des employés dont le salaire appartient à l'intervalle de : 60000 DA à 65000 DA.
- La partition 9 : contient les enregistrements des employés dont le salaire appartient à l'intervalle de : 65000 DA à 70000 DA.

Chaque partition constituée une table dans la base de données et Chaque enregistrement contient 3 champs : code_emp, annee_naiss, salaire

1- **code_emp** : qui est le code de l'employé et qui représente la clé primaire.

2-annee_naiss : qui représente l'âge de l'employé, cet âge varie entre 1965 et 1995, et c'est un nombre aléatoire qui varie entre 1965 et 1995.

3- **salaire** : qui représente le salaire de l'employé dans l'intervalle 20000 DA et 70000 DA. Ces partitions sont obtenues en appliquant le partitionnement horizontal sur la table d'origine.

Et donc on va avoir 10 tables au lieu d'avoir une seule table « **table_origine** », c'est-à-dire

- La table « table_emp_0 » : contient les enregistrements des employés dont le salaire appartient à l'intervalle de : 20000 DA à 25000 DA.
- La table « table_emp_1 » : contient les enregistrements des employés dont le salaire appartient à l'intervalle de : 25000 DA à 30000 DA.
- La table « table_emp_2 » : contient les enregistrements des employés dont le salaire appartient à l'intervalle de : 30000 DA à 35000 DA.
- La table « table_emp_3 » : contient les enregistrements des employés dont le salaire appartient à l'intervalle de : 35000 DA à 40000 DA.

- La table « table_emp_4 » : contient les enregistrements des employés dont le salaire appartient à l'intervalle de : 40000 DA à 45000 DA.
- La table « table_emp_5 » : contient les enregistrements des employés dont le salaire appartient à l'intervalle de : 45000 DA à 50000 DA.
- La table « table_emp_6 » : contient les enregistrements des employés dont le salaire appartient à l'intervalle de : 50000 DA à 55000 DA.
- La table « table_emp_7 » : contient les enregistrements des employés dont le salaire appartient à l'intervalle de : 55000 DA à 60000 DA.
- La table « table_emp_8 » : contient les enregistrements des employés dont le salaire appartient à l'intervalle de : 60000 DA à 65000 DA.
- La table « table_emp_9 » : contient les enregistrements des employés dont le salaire appartient à l'intervalle de : 65000 DA à 70000 DA.

3. Remplissage des 10 tables et table origine :

Il suffit de générer aléatoirement les valeurs des champs par programme :

Code_emp: est la clé primaire de la table et elle est générée par la concaténation des lettres de l'alphabet du français et s'arrêter à 100000 d'enregistrements

Annee_naiss : générer un nombre aléatoire entre 1965 et 1995 et s'arrêter à 100000 d'enregistrements

Salaire : générer un nombre aléatoire entre 20000 et 70000

- Afin de faire le partitionnement horizontal de la table « table_origine » en créant 10 tables comme indiqué précédemment. Avec ces 10 tables si on a une requête sur le champ salaire, alors on interroge uniquement une seule table parmi les 10 tables, et donc on a un gain de temps, ce qui démontre l'utilité du partitionnement horizontal.

Le problème si on a une requête sur le champ de la clé primaire « **code_emp** », alors on doit exécuter la même requête sur les 10 tables, donc on va exécuter la même requête 10 fois, ce qui est très couteux.

4. Créer l'index en utilisant la librairie cmph

La solution est de créer un index en utilisant le **minimal perfect hashing.** Et pour faire ce travail nous allez utiliser une fonction de hachage implémentée dans la librairie **« cmph-2.0.2.tar ».** Cette fonction de hachage va permettre de générer pour chaque clé primaire « **code_emp** » un entier compris entre 0 et 99999, c'est-à-dire 100000 entiers,

et donc l'index sera un tableau de dimension 100000 et chaque case du tableau va contenir un entier compris entre 0 et 9 qui indique le numéro de la table parmi les 10 tables. Ce qui veut dire que pour exécuter une requête sur « **code_emp** », on va juste interroger une seule table et donc on va gagner en temps d'exécution pour ce type de requêtes.

Exemple :

Soit la clé « aaaaaab »

Vous appliquez la fonction de hachage minimal

Le résultat est : id=105

Vous allez dans la case 105 du tableau et vous allez mettre le numéro de la table dans laquelle se trouve la clé « aaaaaab » par exemple si la clé se trouve dans la table « table_emp_4 », donc vous allez mettre la valeur 4 dans la case 105.

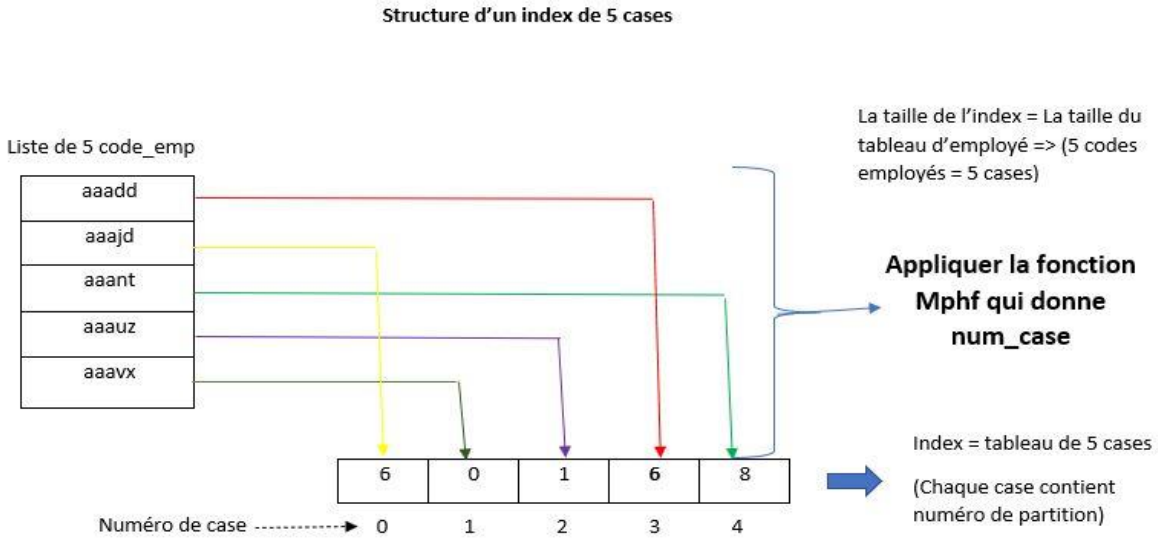


Figure 23 : Exemple de création d'un index de 5 cases

5. Dans les figures suivantes, nous expliquons en détails les étapes principales de l'approche proposée pour notre projet :

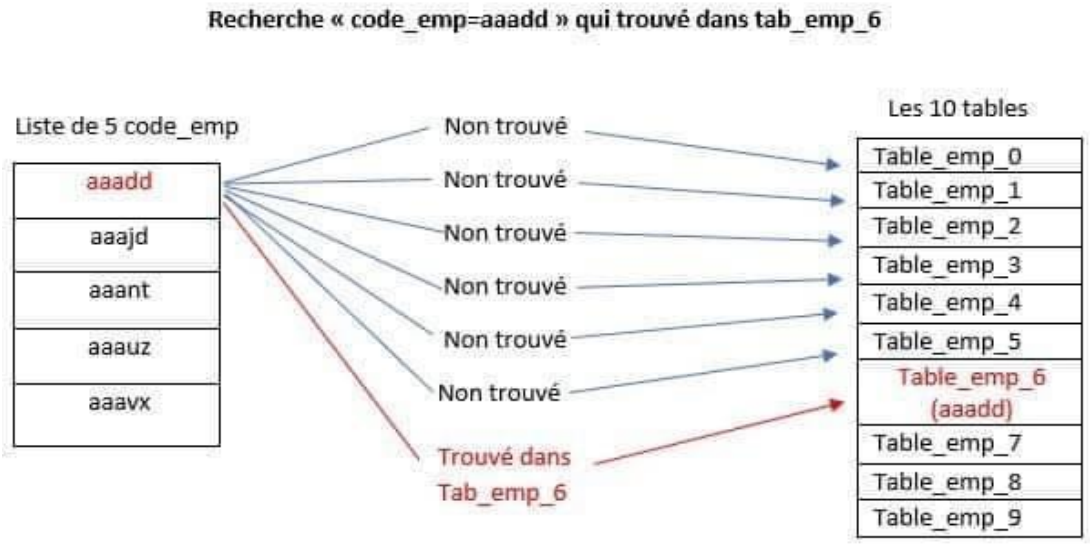


Figure 24 : Exemple de recherche de 5 codes dans les 10 tables

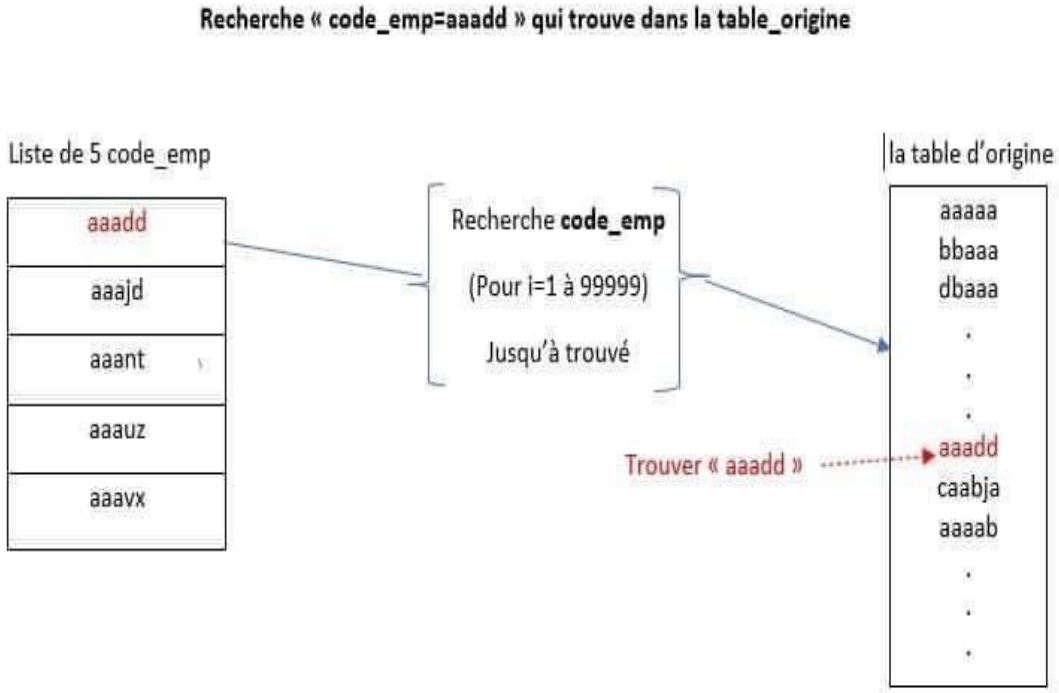


Figure 25 : Exemple de recherche 5 codes dans la table d'origine

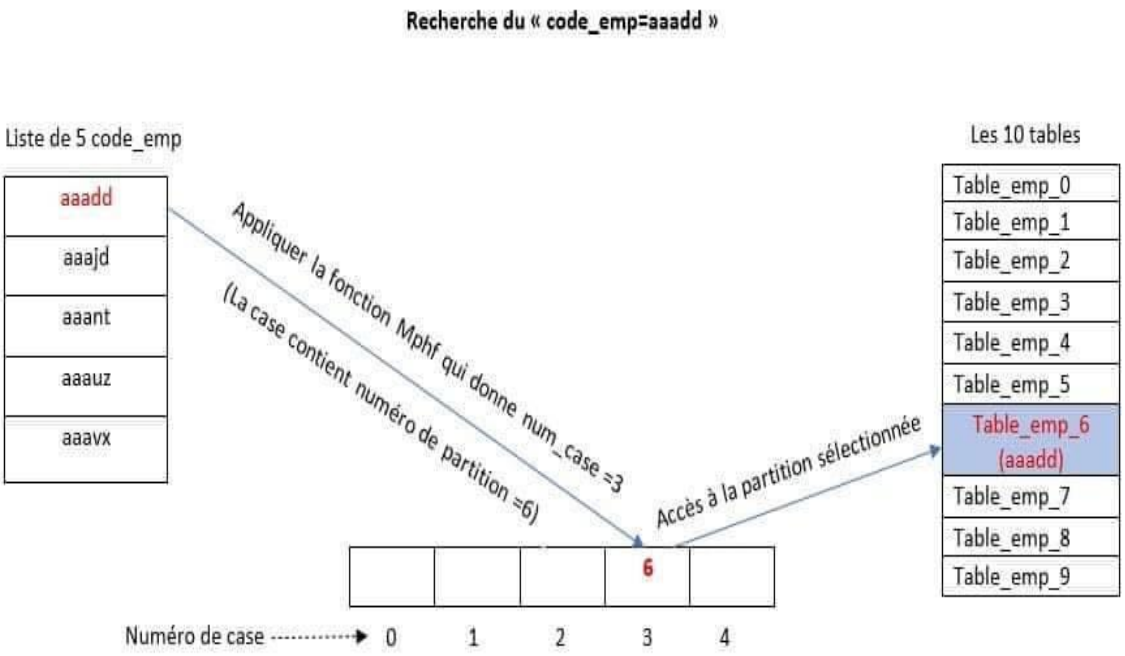


Figure 26 : Exemple de recherche d'un code_emp en utilisant l'index

Conclusion :

Dans ce chapitre nous avons expliqué en détail les déférentes étapes de notre projet, Nous avons expliqué l'utilité du partitionnement horizontal sur notre base de données, Aussi nous avons proposé une solution si va exécuter la même requête 10 fois, ce qui est très couteux. Ensuit nous avons expliqué comment nous avons étudié l'utilisation des indexes et la fonction de hachage parfait minimal. Finalement, notre approche consiste à faire des tests pour faire une comparaison entre les trois méthodes (recherche dans la table origine et dans les 10 tables par index et sans index).

Chapitre 4 : Test et évaluation

1. Introduction :

Dans ce chapitre nous présentons l'implémentation de notre système proposé, nous commençons tout d'abord par la présentation de l'environnement de développement, en détaillant les différents outils utilisés, et les langages de programmation. Puis nous expliquons notre travail ainsi que le temps dans le traitement de certaines méthode dans les bases de données Ensuite nous présentons les résultats obtenus dans les différentes étapes de notre projet par des tableaux et des graphes pour faciliter la compréhension des résultats obtenue, et la comparaison entre les résultats obtenue par les trois méthodes (recherche dans la table origine et dans les 10 tables par index et sans index). Enfin nous interprétons et nous commentons les résultats obtenus pour analyser notre travail et voir si nous avons atteint nos objectifs.

2. Environnement de développement :

Nous avons développé notre projet dans linux système distribution Ubuntu 20.04 LTS.

2.1. Oracle VM VirtualBox :

Oracle VM VirtualBox (anciennement VirtualBox) est un logiciel libre de virtualisation publié par Oracle

VirtualBox permet la création d'un ou de plusieurs ordinateurs virtuels dans lesquels s'installent d'autres systèmes d'exploitation. Il fonctionne sous Windows mais aussi sur Mac, Linux[23].

2.1.1. Fonctionnalités :

La virtualisation permet, sur un même PC, de faire tourner plusieurs systèmes d'exploitation en concurrence. Le système d'exploitation « principal » fait tourner par le biais d'une application servant au partage des ressources et à l'émulation d'un PC un autre système d'exploitation [23].



Figure 27 : Oracle VM VirtualBox 6.1

2.2. Ubuntu :

Ubuntu est un système d'exploitation GNU/Linux basé sur la distribution Linux Debian. Il est développé, commercialisé et maintenu pour les ordinateurs individuels (desktop), les serveurs (Server) et les objets connectés (Core) par la société Canonical.

Ubuntu se définit comme « un système d'exploitation utilisé par des millions de PC à travers le monde »et avec une interface « simple, intuitive, et sécurisée ». Elle est la distribution la plus consultée sur Internet d'après le site Alexa, et le système d'exploitation le plus utilisé sur les systèmes Cloud ainsi que sur les serveurs informatiques

Ubuntu se divise en deux versions :

- La version principale stable dite LTS. Une nouvelle version sort tous les deux ans, le support de sécurité et des mises à jour sont assurés pendant cinq ans. La dernière
- version 20.04, nom de code « Focal Fossa », est sortie le 25 avril 2020. C'est la version recommandée sur le site de Canonical pour le grand public.
- La version secondaire. Une nouvelle mise à jour sort tous les six mois et le support de sécurité et de mises à jour sont assurés pendant 9 mois. L'avant dernière version en date est la 19.04, nom de code « Disco Dingo » qui est sortie le 16 avril 2019 et la dernière version est la 19.10, nom de code « Eoan Ermine » qui est sortie le 17 octobre 2019. C'est la version

recommandée sur le site de Canonical pour ceux qui veulent bénéficier des dernières nouveautés [24].



Figure 28 : Ubuntu 20.04 *Focal Fossa*.

Voici dans ce qui suit les étapes d'installation et configuration du gcc et make

2.3. GCC :

La collection de compilateurs GNU (GCC) est une collection de compilateurs et de bibliothèques pour les langages de programmation C, C ++, Objective-C, Fortran, Ada, Go et D. Un grand nombre de projets open source, y compris le noyau Linux et les outils GNU, sont compilés à l'aide de GCC.

Les référentiels Ubuntu par défaut contiennent un méta-paquet nommé «build-essential» qui inclut la collection de compilateurs GNU, le débogueur GNU et d'autres bibliothèques de développement et outils nécessaires à la compilation de logiciels [25].

```
mimi@mimi-Lenovo-G50-70:~$ sudo apt-get install gcc
[sudo] Mot de passe de mimi :
Lecture des listes de paquets... Fait
Construction de l'arbre des dépendances
Lecture des informations d'état... Fait
gcc est déjà la version la plus récente (4:9.3.0-1ubuntu2).
gcc passé en « installé manuellement ».
```

Figure 29 : Installer gcc

2.4. Make :

Make Le but de l'utilitaire make est de déterminer automatiquement quelles parties d'un grand programme doivent être recompilées et de lancer les commandes pour les recompiler.vous pouvez utiliser make avec n'importe quel langage de programmation dont le compilateur peut être exécuté avec une commande shell. En fait, make ne se limite pas aux programmes. Vous pouvez l'utiliser pour décrire toute tâche où certains fichiers doivent être mis à jour automatiquement à partir d'autres chaque fois que les autres changent.

Pour vous préparer à utiliser make, vous devez écrire un fichier appelé makefile qui décrit les relations entre les fichiers de votre programme et les commandes de mise à jour de chaque fichier. Dans un programme, le fichier exécutable est généralement mis à jour à partir de fichiers objets, qui sont à leur tour créés en compilant des fichiers source [26]

```
mimi@mimi-Lenovo-G50-70:~$ sudo apt-get install make
Lecture des listes de paquets... Fait
Construction de l'arbre des dépendances
Lecture des informations d'état... Fait
make est déjà la version la plus récente (4.2.1-1.2).
make passé en « installé manuellement ».
```

Figure 30 : Installer make

2.5. Télécharger et installer cmph-2.0.2 :

La bibliothèque CMPH encapsule les algorithmes les plus récents et les plus efficaces dans une API rapide, facile à utiliser et de qualité production. La bibliothèque a été conçue pour

fonctionner avec de grandes entrées qui ne peuvent pas tenir dans la mémoire principale. Il a été utilisé avec succès pour construire des fonctions de hachage parfaites minimales pour des ensembles de plus de 100 millions de clés, et nous avons l'intention d'étendre ce nombre à l'ordre du milliard de clés. Bien qu'il y ait un manque de bibliothèques similaires [6].

En exécutant la commande suivante :

1. Installer libcmph-dev_2.0-2_amd64.deb :

   Update the package index: sudo apt-get update

   Install libcmph-dev deb package: sudo apt-get install libcmph-dev

2. Pour exécuter le cmph tapez les commandes suivantes :

   - Etape de configuration, vous tapez ". /configure"

   - Etape de vérification, vous tapez "make"

   - Etape de vérification, vous tapez "make check"

   - Etape d'installation en tant que root (besoin du mot de passe),

Vous tapez "sudo make install"

C'est bon, la librairie est maintenant installée.

- Optionnellement, vous pouvez nettoyer les fichiers temporaires de compilation : "make clean".

2.6. Mysql :

MySQL est un système de gestion de base de données open-source, couramment installé dans le cadre de la populaire pile LAMP (Linux, Apache, MySQL, PHP / Python / Perl). Il implémente le modèle relationnel et utilise le langage de requête structuré (mieux connu sous le nom de SQL) pour gérer ses données [27].

3. Implémentation :

Pour s'assurer que cette hypothèse est correcte, nous avons créé 8 programmes (où appliquons-nous la fonction de hachage parfait minimal, nous avons utilisée l'algorithme de BDZ) :

Premièrement, on a créé des 10 tables qui correspondent aux 10 partitions et une table d'origine qui contient tous les enregistrements (100000 d'enregistrements).



Figure 31 : Création des 11 tables

Ensuite, on a le remplissage des 10 tables et la table d'origine tel que la table d'origine contient 100000 d'enregistrements et 10 tables représentent les 10 partitions, pour que la structure des tables est :

La table d'origine contient :

- Clé primaire (appelé code_emp, de type VARCHAR et taille 5)
- L'âge de l'employé (appelé annee_naiss, de, de type INT et cet âge varie entre 1965 et 1995, et c'est un nombre aléatoire qui varie entre 1965 et 1995)
- Le salaire de l'employé (appelé salaire, de type INT et ce salaire varie entre 20000 DA et 70000 DA, et c'est un nombre aléatoire qui varie entre 20000 et 70000)

Les 10 tables :

- Clé primaire (appelé code_emp, de type VARCHAR et taille 5)
- L'âge de l'employé (appelé annee_naiss, de type INT et cet âge varie entre 1965 et 1995, et c'est un nombre aléatoire qui varie entre 1965 et 1995)
- Le salaire de l'employé (appelé salaire, de type INT et ce salaire varie entre 20000 DA et 70000 DA, Ces partitions sont obtenues en appliquant le partitionnement horizontal sur la table d'origine).

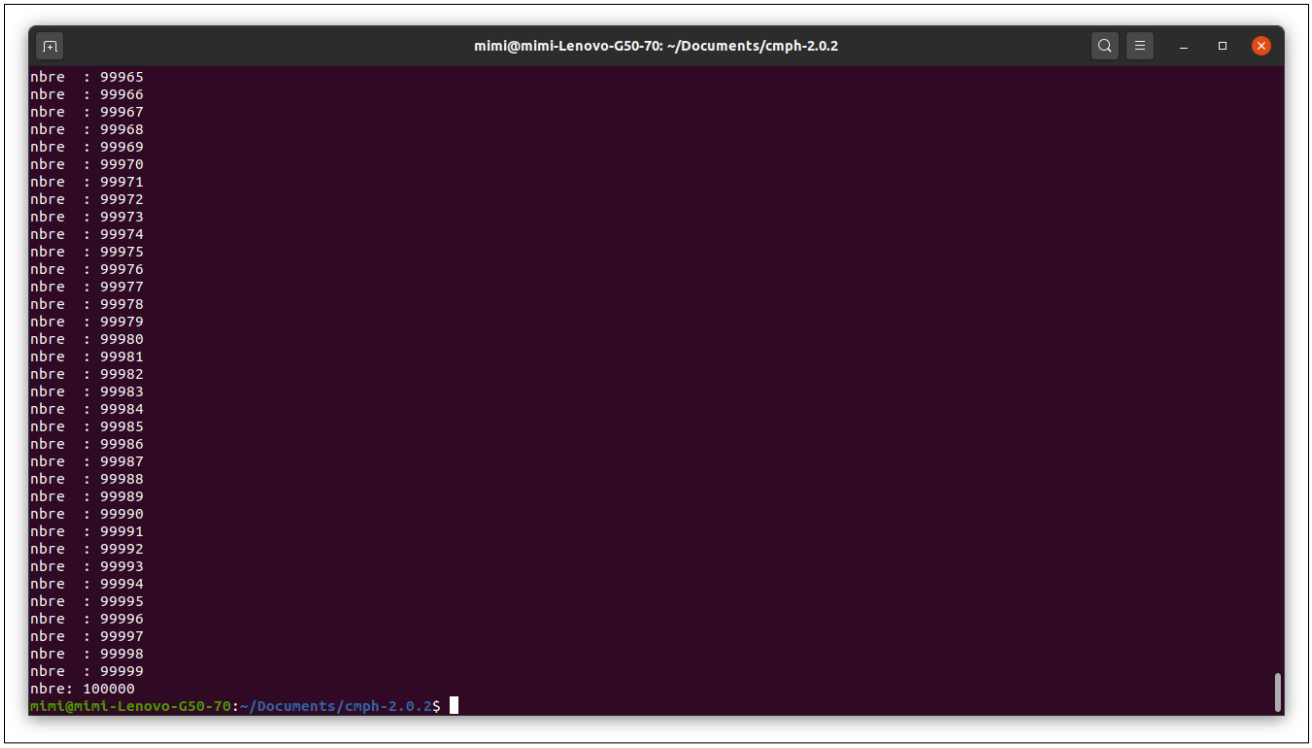


Figure 32 : Remplissage des11 tables

Aussi on a compté le nombre d'enregistrements dans chacune des 10 tables et il permet aussi de compter le nombre d'enregistrements dans la table d'origine.

```
mimi@mimi-Lenovo-G50-70:~/Documents/cmph-2.0.2$ gcc programme_3_compter_nombre_enregistrements_des_10_tables_et_table_origine.c -O3 -lcmph -o
programme_3_compter_nombre_enregistrements_des_10_tables_et_table_origine -lmysqlclient
mimi@mimi-Lenovo-G50-70:~/Documents/cmph-2.0.2$ ./programme_3_compter_nombre_enregistrements_des_10_tables_et_table_origine
 je suis connecté
nombre d'enregistrements dans --- table_origine  :  100000
nombre d'enregistrements dans --- table_emp_0  :  9991
nombre d'enregistrements dans --- table_emp_1  :  9986
nombre d'enregistrements dans --- table_emp_2  :  10082
nombre d'enregistrements dans --- table_emp_3  :  10041
nombre d'enregistrements dans --- table_emp_4  :  9968
nombre d'enregistrements dans --- table_emp_5  :  9982
nombre d'enregistrements dans --- table_emp_6  :  9892
nombre d'enregistrements dans --- table_emp_7  :  10046
nombre d'enregistrements dans --- table_emp_8  :  10111
nombre d'enregistrements dans --- table_emp_9  :  9901
mimi@mimi-Lenovo-G50-70:~/Documents/cmph-2.0.2$
```

Figure 33 : Compté le nombre d'enregistrements dans chacune des 11 tables

Programme 4 : Ce programme nous l'avons appelé « programme_4_creation_index.c », il permet de créer l'index en utilisant la librairie cmph, et cet index est un tableau de dimension 100000, et ensuite il enregistre cet index sur le disque. L'espace occupé par l'index sur le disque est optimal, et est organisé dans une structure particulière afin de gagner en espace de stockage.

Pour compiler ce programme taper_: gcc bits_table.c programme_4_creation_index.c -Wall -O3 -lcmph -o programme_4_creation_index -lmysqlclient

Ensuite, l'exécuter avec_: ./programme_4_creation_index

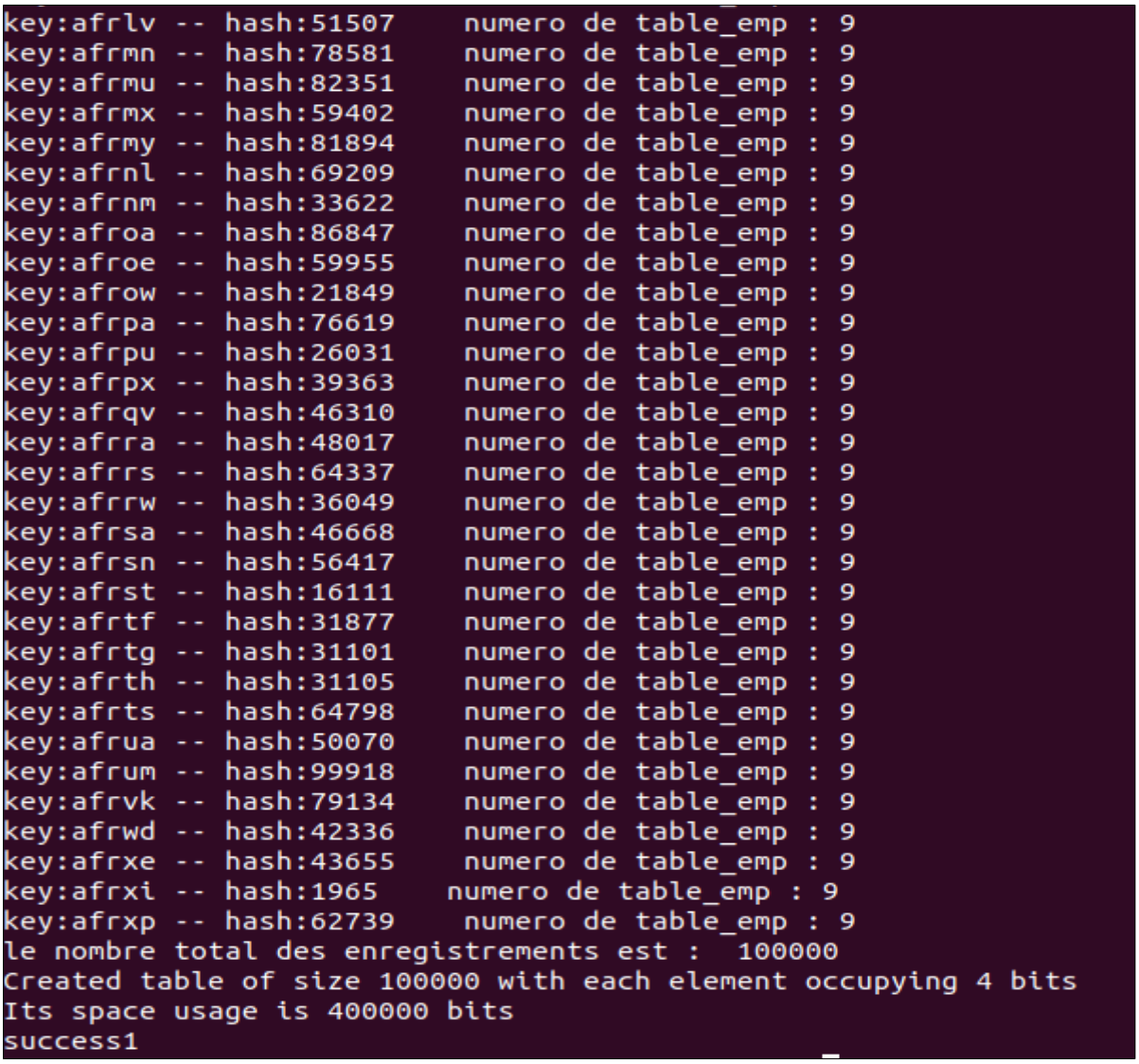


Figure 34 : Compilation et exécution du programme 4

Programme 5 : Ce programme nous l'avons appelé « programme_5_ creation _liste _ codes _ aleatoires.c », il permet de créer 1000 code_emp aléatoires. Il commence par générer 1000 nombres aléatoires entre 0 et 99999 et ensuite construit tous les codes_emp de la table d'origine et à chaque fois qu'il atteint un code_emp qui a un numéro d'ordre égal à un certain nombre aléatoire, alors ce code_emp est conservé. On continue ainsi jusqu'à trouver tous les codes_emp aléatoires. Ensuite on conserve ces codes_emp dans un fichier texte qui s'appelle : « mes_codes_aleatoires.txt ».

Pour compiler ce programme taper : gcc programme_5_creation_liste_codes_aleatoires.c -O3 -lcmph -o programme_5_creation_liste_codes_aleatoires -lmysqlclient

Ensuite, l'exécuter avec : ./programme_5_creation_liste_codes_aleatoires

```
mimi@mimi-Lenovo-G50-70:~/Documents/cmph-2.0.2$ gcc programme_5_creation_liste_codes_aleatoires.c -O3 -lcmph -o programme_5_creation_liste_cod
es_aleatoires -lmysqlclient
mimi@mimi-Lenovo-G50-70:~/Documents/cmph-2.0.2$ ./^C
mimi@mimi-Lenovo-G50-70:~/Documents/cmph-2.0.2$ ./programme_5_creation_liste_codes_aleatoires
 je suis connecté
nombre aléatoire: 81  code_emp :  aaadd
nombre aléatoire: 237  code_emp :  aaajd
nombre aléatoire: 346  code_emp :  aaani
nombre aléatoire: 545  code_emp :  aaauz
nombre aléatoire: 569  code_emp :  aaavx
nombre aléatoire: 606  code_emp :  aaaxi
nombre aléatoire: 657  code_emp :  aaazh
nombre aléatoire: 681  code_emp :  aabaf
nombre aléatoire: 723  code_emp :  aabbv
nombre aléatoire: 925  code_emp :  aabjp
```

Figure 35 : Compilation et exécution du programme 5

Programme 6 : Ce programme nous l'avons appelé « programme_6_ recherche_de_mille _ requetes_dans_les_10_tables.c»,il permet de former 1000 requêtes à partir des 1000 « code_emp aléatoires » et pour chaque code aléatoire il construit une requête pour rechercher ce code dans la (table_emp_0) et si ce code ne se trouve pas dans (table_emp_0), alors il va le chercher dans (table_emp_1), et il continue ainsi jusqu'à trouver le code_emp et il s'arrête et il passe au prochain code_emp et il fait le même travail pour tous les code_emp aléatoires.

Pour compiler ce programme taper_: gcc programme_6_ recherche_ de_ mille _ requetes _ dans_les_10_tables.c -O3 -lcmph -o programme_6_ recherche_de _mille_requetes _dans_les_ 10_tables -lmysqlclient

Ensuite , l'exécuter avec_:

./programme_6_recherche_de_mille_requetes_dans_les_10_tables

```
mimi@mimi-Lenovo-G50-70:~/Documents/cmph-2.0.2$ gcc programme_6_recherche_de_mille_requetes_dans_les_10_tables.c -O3 -lcmph -o programme_6_rec
herche_de_mille_requetes_dans_les_10_tables -lmysqlclient
mimi@mimi-Lenovo-G50-70:~/Documents/cmph-2.0.2$ ./programme_6_recherche_de_mille_requetes_dans_les_10_tables
 je suis connecté

 requete : SELECT * FROM table_emp_7 WHERE code_emp='aaadd';    trouvé
 requete : SELECT * FROM table_emp_5 WHERE code_emp='aaajd';    trouvé
 requete : SELECT * FROM table_emp_2 WHERE code_emp='aaani';    trouvé
 requete : SELECT * FROM table_emp_8 WHERE code_emp='aaauz';    trouvé
 requete : SELECT * FROM table_emp_9 WHERE code_emp='aaavx';    trouvé
 requete : SELECT * FROM table_emp_1 WHERE code_emp='aaaxi';    trouvé
 requete : SELECT * FROM table_emp_6 WHERE code_emp='aaazh';    trouvé
 requete : SELECT * FROM table_emp_8 WHERE code_emp='aabaf';    trouvé
 requete : SELECT * FROM table_emp_1 WHERE code_emp='aabbv';    trouvé
 requete : SELECT * FROM table_emp_4 WHERE code_emp='aabjp';    trouvé
 requete : SELECT * FROM table_emp_7 WHERE code_emp='aabll';    trouvé
 requete : SELECT * FROM table_emp_4 WHERE code_emp='aabnz';    trouvé
 requete : SELECT * FROM table_emp_7 WHERE code_emp='aabom';    trouvé
 requete : SELECT * FROM table_emp_0 WHERE code_emp='aabox';    trouvé
 requete : SELECT * FROM table_emp_1 WHERE code_emp='aabwh';    trouvé
 requete : SELECT * FROM table_emp_0 WHERE code_emp='aacai';    trouvé
 requete : SELECT * FROM table_emp_7 WHERE code_emp='aaces';    trouvé
 requete : SELECT * FROM table_emp_2 WHERE code_emp='aacst';    trouvé
 requete : SELECT * FROM table_emp_7 WHERE code_emp='aacwx';    trouvé
 requete : SELECT * FROM table_emp_9 WHERE code_emp='aacxl';    trouvé
 requete : SELECT * FROM table_emp_6 WHERE code_emp='aacxm';    trouvé
 requete : SELECT * FROM table_emp_6 WHERE code_emp='aaczt';    trouvé
 requete : SELECT * FROM table_emp_2 WHERE code_emp='aadhf';    trouvé
 requete : SELECT * FROM table_emp_5 WHERE code_emp='aadij';    trouvé
 requete : SELECT * FROM table_emp_1 WHERE code_emp='aadis';    trouvé
```

Figure 36 : Compilation et exécution du programme 6

Programme 7 : Ce programme nous l'avons appelé « programme_7_ recherche_de _ mille _ requetes_dans_table_origine.c », il permet de former 1000 requêtes à partir des 1000 « code_emp aléatoires » et pour chaque code aléatoire il construit une requête pour rechercher ce code dans la table_origine, et il fait le même travail pour tous les code_emp aléatoires.

Pour compiler ce programme taper : gcc programme_7_ recherche_de _mille _ requetes _dans_table_origine.c -O3 -lcmph -o programme_7_ recherche_de_mille _requetes_ dans _table_origine -lmysqlclient

Ensuite, l'exécuter avec :

./programme_7_recherche_de_mille_requetes_dans_table_origine



Figure 37 : Compilation et exécution du programme 7

Programme 8 : Ce programme nous l'avons appelé « programme_8 _recherche _ de _ mille _ requetes_dans_les_10_tables_par_index.c » ,ce programme permet de charger l'index qui était sur le disque et de charger les 1000 code_emp stockés dans le fichier « mes_codes_aleatoires.txt » ,et ensuite il permet de former 1000 requêtes à partir des 1000 code_emp aléatoires et pour chaque code aléatoire code_emp il applique la fonction de hachage parfait minimal, et il obtient une valeur de hachage qui représente le numéro de case dans l'index(qui est un tableau) et dans cette case de l'index il trouve le numéro de partition dans laquelle il va chercher ce code_emp, et donc il construit une requête pour rechercher ce code_emp dans cette partition , et il fait le même travail pour tous les code_emp aléatoires.

Pour compiler ce programme taper : gcc bits_table.c programme_8_ recherche _de_mille_requetes_dans_les_10_tables_par_index.c -Wall -O3 -lcmph -o programme _8 _ recherche_de_mille_requetes_dans_les_10_tables_par_index -lmysqlclient

Ensuite, l'exécuter avec :

./programme_8_recherche_de_mille_requetes_dans_les_10_tables_par_index

```
mimi@mimi-Lenovo-G50-70:~/Documents/cmph-2.0.2$ gcc bits_table.c programme_8_recherche_de_mille_requetes_dans_les_10_tables_par_index.c -Wall
-O3 -lcmph -o programme_8_recherche_de_mille_requetes_dans_les_10_tables_par_index -lmysqlclient
mimi@mimi-Lenovo-G50-70:~/Documents/cmph-2.0.2$ ./programme_8_recherche_de_mille_requetes_dans_les_10_tables_par_index
 je suis connecté
Loaded table of size 100000 with each element occupying 4 bits
Its space usage is 400000 bits

 requete : SELECT * FROM table_emp_7 WHERE code_emp='aaadd';    trouvé
 requete : SELECT * FROM table_emp_5 WHERE code_emp='aaajd';    trouvé
 requete : SELECT * FROM table_emp_2 WHERE code_emp='aaani';    trouvé
 requete : SELECT * FROM table_emp_8 WHERE code_emp='aaauz';    trouvé
 requete : SELECT * FROM table_emp_9 WHERE code_emp='aaavx';    trouvé
 requete : SELECT * FROM table_emp_1 WHERE code_emp='aaaxi';    trouvé
 requete : SELECT * FROM table_emp_6 WHERE code_emp='aaazh';    trouvé
 requete : SELECT * FROM table_emp_8 WHERE code_emp='aabaf';    trouvé
 requete : SELECT * FROM table_emp_1 WHERE code_emp='aabbv';    trouvé
 requete : SELECT * FROM table_emp_4 WHERE code_emp='aabjp';    trouvé
```

Figure 38 : Compilation et exécution du programme 8

4. Explication de travail :

A rappeler que le travail de ce mémoire vise à utiliser le hachage parfait minimal afin de gagner du temps dans le traitement de certaines applications dans les bases de données relationnelles. En effet, la construction de 10 tables à partir de la base de données d'origine est le résultat du partitionnement de la base de données d'origine selon les salaires des employés. Et l'objectif de l'utilisation de l'index est lié aux requêtes faites sur le champ « code_emp ».

Afin de montrer l'utilité de la conception présentée dans le chapitre 3 (conception du système), nous avons sélectionné aléatoirement 1000 « code_emp » parmi tous les « code_emp » de la base de données d'origine (table_origine), ensuite nous avons adopté le plan d'expérimentation suivant :

1- Lancer 1000 requêtes sur la clé primaire « code_emp » en utilisant les « code_emp » choisis précédemment. Ces 1000 requêtes aléatoires sont exécutées sur les 10 partitions (10 tables) mais sans utiliser notre index. Dans ce cas la recherche se fera sur le champ « code_emp », et on doit chercher dans la table « table_emp_0 » et si on ne trouve pas l'enregistrement on continue à chercher dans « table_emp_1 », si on ne trouve pas l'enregistrement on continue à chercher dans « table_emp_1 » ...etc. et on continue jusqu'à trouver l'enregistrement définis par la clé primaire « code_emp ». Nous avons également calculé le temps total pris par la machine afin d'exécuter ces mille requêtes.

```
requete : SELECT * FROM table_emp_1 WHERE code_emp='afpny';    trouvé
requete : SELECT * FROM table_emp_5 WHERE code_emp='afpsc';    trouvé
requete : SELECT * FROM table_emp_5 WHERE code_emp='afpss';    trouvé
requete : SELECT * FROM table_emp_2 WHERE code_emp='afptx';    trouvé
requete : SELECT * FROM table_emp_7 WHERE code_emp='afpvf';    trouvé
requete : SELECT * FROM table_emp_8 WHERE code_emp='afpvu';    trouvé
requete : SELECT * FROM table_emp_4 WHERE code_emp='afqeo';    trouvé
requete : SELECT * FROM table_emp_1 WHERE code_emp='afqhu';    trouvé
requete : SELECT * FROM table_emp_1 WHERE code_emp='afqix';    trouvé
requete : SELECT * FROM table_emp_3 WHERE code_emp='afqku';    trouvé
requete : SELECT * FROM table_emp_7 WHERE code_emp='afqlf';    trouvé
requete : SELECT * FROM table_emp_2 WHERE code_emp='afqnr';    trouvé
requete : SELECT * FROM table_emp_4 WHERE code_emp='afqqw';    trouvé
requete : SELECT * FROM table_emp_8 WHERE code_emp='afqsg';    trouvé
requete : SELECT * FROM table_emp_3 WHERE code_emp='afqsr';    trouvé
requete : SELECT * FROM table_emp_8 WHERE code_emp='afqum';    trouvé
requete : SELECT * FROM table_emp_4 WHERE code_emp='afqyq';    trouvé
requete : SELECT * FROM table_emp_7 WHERE code_emp='afrjz';    trouvé
requete : SELECT * FROM table_emp_9 WHERE code_emp='afrmy';    trouvé
requete : SELECT * FROM table_emp_7 WHERE code_emp='afros';    trouvé
requete : SELECT * FROM table_emp_6 WHERE code_emp='afrox';    trouvé
requete : SELECT * FROM table_emp_9 WHERE code_emp='afrum';    trouvé
requete : SELECT * FROM table_emp_5 WHERE code_emp='afrvo';    trouvé
le temps d'exécution des 1000 requetes en secondes  est :  0.145355
imi@mimi-Lenovo-G50-70:~/Documents/cmph-2.0.2$
```

Figure 39 : Résultat d'exécution programme 6

2- Lancer 1000 requêtes sur la clé primaire « code_emp » en utilisant les « code_emp » choisis précédemment. Ces 1000 requêtes aléatoires sont exécutées sur la base de données d'origine (table_origine). Nous avons également calculé le temps total pris par la machine afin d'exécuter ces mille requêtes.

```
requete : SELECT * FROM table_origine WHERE code_emp='afpss';    trouvé
requete : SELECT * FROM table_origine WHERE code_emp='afptx';    trouvé
requete : SELECT * FROM table_origine WHERE code_emp='afpvf';    trouvé
requete : SELECT * FROM table_origine WHERE code_emp='afpvu';    trouvé
requete : SELECT * FROM table_origine WHERE code_emp='afqeo';    trouvé
requete : SELECT * FROM table_origine WHERE code_emp='afqhu';    trouvé
requete : SELECT * FROM table_origine WHERE code_emp='afqix';    trouvé
requete : SELECT * FROM table_origine WHERE code_emp='afqku';    trouvé
requete : SELECT * FROM table_origine WHERE code_emp='afqlf';    trouvé
requete : SELECT * FROM table_origine WHERE code_emp='afqnr';    trouvé
requete : SELECT * FROM table_origine WHERE code_emp='afqqw';    trouvé
requete : SELECT * FROM table_origine WHERE code_emp='afqsg';    trouvé
requete : SELECT * FROM table_origine WHERE code_emp='afqsr';    trouvé
requete : SELECT * FROM table_origine WHERE code_emp='afqum';    trouvé
requete : SELECT * FROM table_origine WHERE code_emp='afqyq';    trouvé
requete : SELECT * FROM table_origine WHERE code_emp='afrjz';    trouvé
requete : SELECT * FROM table_origine WHERE code_emp='afrmy';    trouvé
requete : SELECT * FROM table_origine WHERE code_emp='afros';    trouvé
requete : SELECT * FROM table_origine WHERE code_emp='afrox';    trouvé
requete : SELECT * FROM table_origine WHERE code_emp='afrum';    trouvé
requete : SELECT * FROM table_origine WHERE code_emp='afrvo';    trouvé
le temps d'exécution des 1000 requetes en secondes  est :  0.039826

mimi@mimi-Lenovo-G50-70:~/Documents/cmph-2.0.2$
```

Figure 40 : Résultat d'exécution programme 7

3- Les mêmes mille requêtes sur les partitions : table_emp_0, table_emp_1..., table_emp_9, mais en utilisant notre index. Du coup chaque requête se fera sur une seule des dix tables. Dans ce cas on va suivre les étapes suivantes :

- Appliquer la fonction du hachage parfait minimal afin de trouver le numéro de case de l'index.
- On va voir le contenu de la case correspondante dans l'index et le contenu de cette case est le numéro de la partition ou on va chercher l'enregistrement. Dans le contenu des cases de l'index sont des valeurs allant de 0, 1, … ,9.
- Lancer la recherche dans la partition sélectionnée afin de trouver l'enregistrement.

• Nous avons également calculé le temps total pris par la machine afin d'exécuter ces mille requêtes.



Figure 41 : Résultat d'exécution programme 8

5. Les résultats :

D'après les résultats d'exécution du 3 dernier programmes (programme 6, programme 7, programme 8), nous avons donné le tableau suivant :

Nom de programme	temps d'exécution
Programme_6_recherche_de_mille_requetes_dans_les_10_tables.c	0,145355
Programme_7_recherche_de_mille_requetes_dans_table_origine.c	0,0398226
Programme_8_recherche_de_mille_requetes_dans_les_10_tables_par_index.c	0,031494

Table 2 : Temps d'exécution de chaque programme

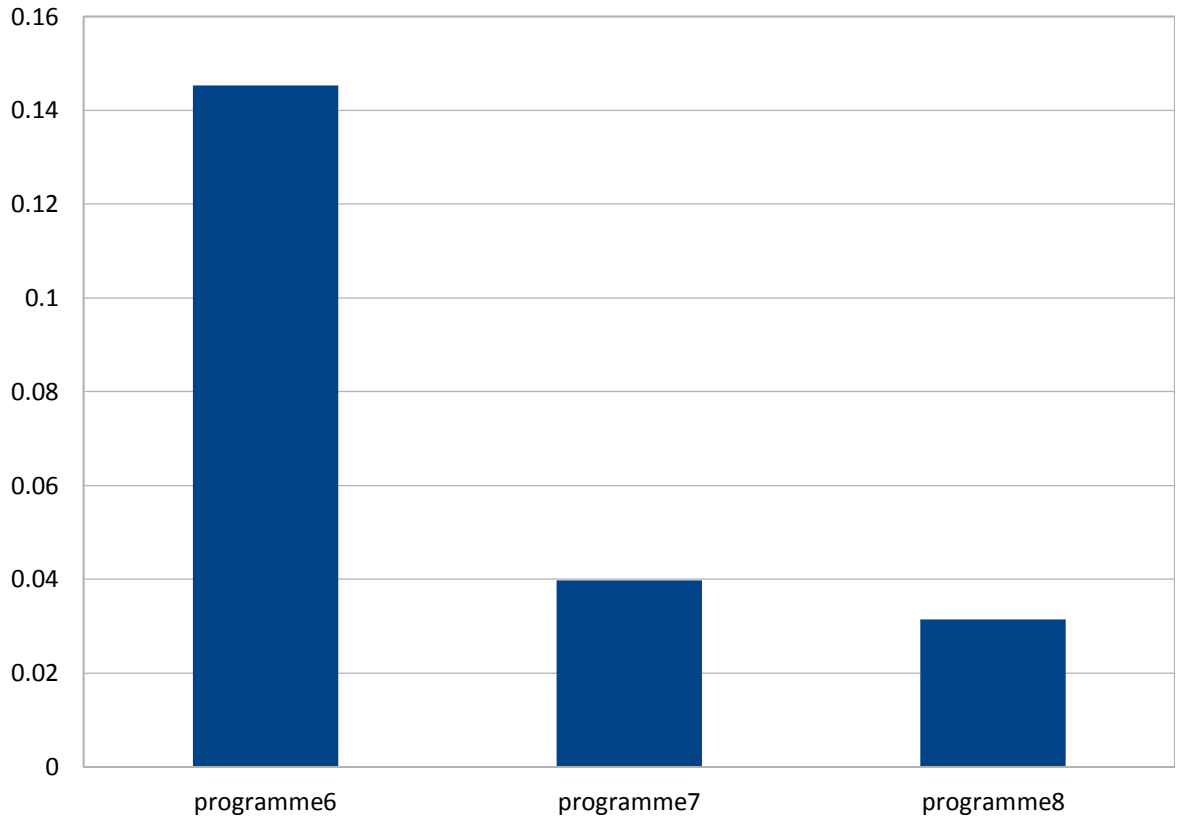


Figure 42 : Diagramme du temps d'exécution (sec)**.**

À partir de ces résultats, nous remarquons que le temps d'exécution des 1000 requête dans la table d'origine sur 7^{em} programme est moins que le temps d'exécution dans les 10 tables sur 6^{em} programme. Comme nous avons remarqué la recherche

Nous remarquons que le temps d'exécution des 1000 requête dans 10 tables sur le 8^{em} programme est moins que le temps d'exécution des autres deux programmes.

D'autre façon, la recherche où nous avons appliqué la fonction de hachage parfait minimal (en utilisant notre index) est mieux que les autres méthodes de recherche

Conclusion :

Dans ce chapitre nous avons parlé sur test et l'évaluation de notre projet, nous avons conclu que le hachage parfait minimal par l'index a donné des résultats acceptables et satisfait. Nous avons présenté dans ce chapitre les différentes ressources utilisées, comme ubuntu et ces différentes bibliothèques comme cmph, Aussi l'environnement de développement qui aide à manipuler les codes en C, ensuite nous avons expliqué le fonctionnement des trois méthodes utilisé : recherche dans une seule table, et recherche dans une 10 table par index avec l'application de la fonction de hachage parfait minimal et sans utiliser l'index, et finalement nous avons présenté le temps d'exécution par les trois méthodes de recherche.

Après les ressources, finalement nous avons discuté sur les résultats obtenus dans ce travail, par la comparaison entre les étapes du projet ou par des tableaux et des courbes qui donnent une vision totale sur le projet.

Conclusion générale

Conclusion générale :

L'explosion de la volumétrie des données, qui reflète le changement d'échelle des volumes, nombres et types, a imposé aux différents chercheurs depuis quelques années de nouveaux défis, et il est question de chercher à concevoir de nouvelles technologies et chercher les meilleures solutions pour contenir et traiter ces volumes énormes de données. Cette nouvelle informatique, avec ses nouvelles exigences, a conduit aux nouveaux concepts qui ont concouru à l'émergence de nouvelles techniques d'indexation, basé en particulier sur le hachage.

Pour justifier et motiver l'utilisation du hachage parfait minimal, nous avons procédé dans le cadre de notre PFE à une étude comparative des performances entre les trois façons de recherche de 1000 requêtes aléatoires :

- Dans la table d'origine qui contient tous les enregistrements
- Dans les 10 tables sans utiliser l'index (10 partitions).
- Dans les 10 tables par index en utilisant la fonction de hachage parfait minimal (librairie cmph).

Afin de comparer les performances des trois tests nous avons rempli la table d'origine avec 100000 enregistrements et les 10 tables sont obtenues en effectuent un partitionnement horizontal de la table d'origine selon le champ « salaire ». Nous avons ensuite calculé le temps d'exécution total des 1000 requêtes dans les trois cas. Rappelons que ces 1000 requêtes sont effectuées sur 1000 code_emp générés aléatoirement.

Les résultats des trois tests (temps d 'exécution) obtenus dans notre étude, sont en faveur de la solution d'utilisation d'un index par rapport aux deux autres méthodes. L'utilisation de la fonction de hachage parfait minimal permet de remplir chaque case de l'index (qui est un tableau) par un numéro de partition et le fait d'utiliser ces numéros de partition permet d'accéder directement vers la partition qui contient le code_emp concerné.

Suite aux résultats des tests effectués, nous pouvons conclure que le hachage parfait minimal permet d'améliorer les performances de recherche des enregistrements dans les tables implémentées dans SGBD MySQL.

Bibliographie

[1] Jean-François Pillou, https://www.commentcamarche.net/contents/104-bases-de-donnees-introduction, Dernière modification le jeudi 12 avril 2018

[2] BEKADDOUR Abderazak , BECHLAGHEM Seyf-Allah, Etude comparative des performances des bases de données SQL et NoSQL MySQL vs MongoDB, Mémoire master informatique, Université Abou Bakr Belkaid– Tlemcen,2017.

[3]https://www.lucidchart.com/pages/fr/comment-concevoir-une-base-de-donnees#:~:text=Pour%20concevoir%20une%20base%20de,primaires%20et%20analyse%20des%20relations

[4] Philippe Rigaux, http://sys.bdpedia.fr/stock.html?fbclid=IwAR3qCPtepXetjN-qoXqO4yibGxN5BbTZnk27fzolgo-K6sOi_4SUNq0SVHU

[5] Philippe Rigaux, http://sys.bdpedia.fr/hachage.html

[6] Djamel Belazzougui, Davi de Castro Reis, Fabiano Cupertino Botelho,Nivio Ziviani, http://cmph.sourceforge.net/, Last Updated: Sat Jun 9 2012

[7] https://sql.sh/cours/where/between

[8] Cours Bases de donnéesStockage des données, indexation optimisation des requêtes 3° année (MISI)

[9]https://www.ibm.com/support/knowledgecenter/fr/SSGU8G_12.1.0/com.ibm.ddi.doc/ids_ddi_084.htm

[10] https://www.memoireonline.com/01/20/11419/m_Mise-en-place-dune-base-de-donnees-repartie-sous-Oracle-Cas-de-la-gestion-du-dossier-judiciair10.html

[11] https://sql.sh/sgbd/mysql

[12] https://www.lemagit.fr/definition/Structure-de-donnees

[13] Djamal Belazzougui, Hachage parfait,minimal et monotone,LIAFA, Univ. Paris Diderot - Paris 7

[14] http://zegour.esi.dz/Publication/Livre2/Partie-sdd/Part1 hachage.htm

[15] https://people.montefiore.uliege.be/herbiet/crypto/10Algorithmes%20de%20hachage.pdf

[16] http://deptinfo.unice.fr/~regin/cours/cours/ASD/C5_AlgoSdd_Hash.pdf

[17] https://hal.inria.fr/hal-01566246/document

[18] https://www.linternaute.fr/dictionnaire/fr/definition/index/

[19] https://openclassrooms.com/fr/courses/1959476-administrez-vos-bases-de-donnees-avec-mysql/1962880-index

[20] El Amin Aoulad Abdelouarit - Faculté des sciences de Tétouan, Mohamed Merouani – Faculté poly disciplinaire de Tétouan, Les index pour les entrepôts de données : comparaison entre index arbre-B et Bitmap

[21] https://openclassrooms.com/fr/courses/1959476-administrez-vos-bases-de-donnees-avec-mysql/1962880-index

[22] Jonathan Krier, https://krierjon.developpez.com/mysql/partitionnement/, Publié le 1er août 2006 - Mis à jour le 24 mars 2016

[23] https://www.pack-logiciels-libres.fr/spip.php?logiciel35

[24] Didier Roche, https://fr.wikipedia.org/wiki/Ubuntu_(syst%C3%A8me_d%27exploitation)

[25] https://linuxize.com/post/how-to-install-gcc-on-ubuntu-20-04/

[26] https ://www. tutorialspoint.com/unix_commands/ make.htm

[27] https://www.atechtown.com/install-and-configure-mysql-on-ubuntu/