

Master d'Informatique : M1 parcours DAC

SAM 4IN803

Stockage et Accès aux Mégadonnées

Hubert Naacke

Support de cours

Contenu partiel à compléter en séance

- Volontairement incomplet
- Assiduité fortement recommandée
 - Présence au cours nécessaire
 - Interaction pendant les TD/TME

Objectifs

Présenter les architectures des systèmes de gestion de bases de données (réparties) et les techniques permettant de les implémenter.

Techniques d'implémentation des SGBD relationnels.

Architecture des bases de données réparties et du Web.

Conception, interrogation et manipulation de données réparties.

Mise en pratique : chaque séance comporte 2h de TME

Plan

- Méthodes d'accès et indexation
- Structure d'index (hachage, arbre B+)
- Optimisation de requêtes
- Bases de données réparties: fragmentation
- Interrogation de bases de données réparties
- SGBD parallèles
- Transactions réparties
- Reprise sur pannes
- Gestion de données hétérogènes et réparties
- TME : Oracle, JDBC, SimJava...

Bibliographie

- H. Garcia-Molina, J.D.Ullman, J. Widom : *Database System Implementation*, Prentice Hall, 2000.
- M.T.Özsu, P. Valduriez : *Principles of Distributed Database Systems*, 3rd edition, Prentice Hall, 2011
- R. Ramakrishnan – J. Gehrke : *Database Management Systems*, Mc-Graw Hill
- S. Abiteboul, P. Buneman, D. Suciu : *Data on the Web : from relations to semistructured data and XML*, Morgan Kaufmann, 1999.
- [Articles récents cités sur la page de l'UE](#)

Cours 1

Méthodes d'accès

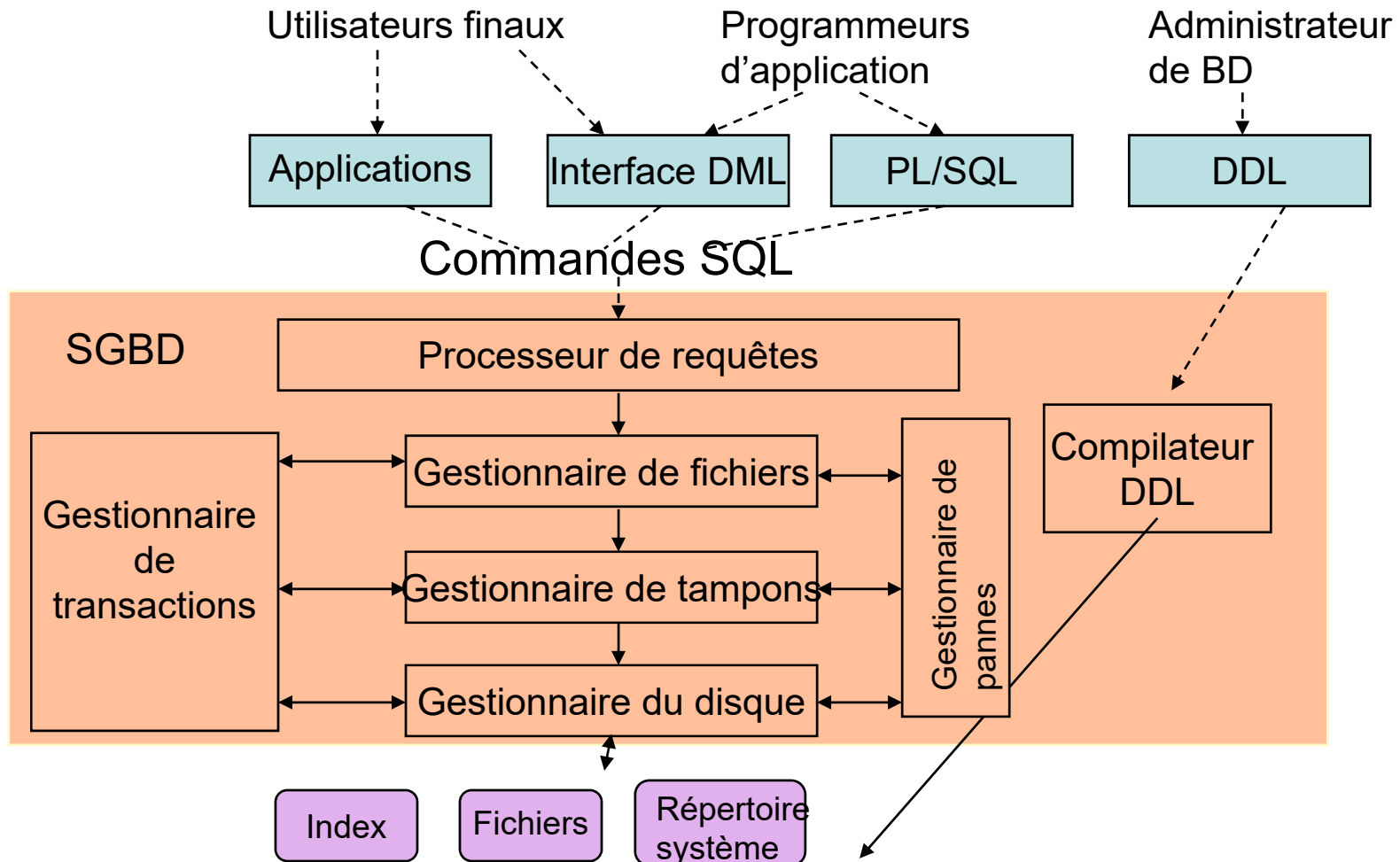
Plan

- Fonctions et structure des SGBD
- Structures physiques
 - Stockage des données
 - Organisation de fichiers et indexation
 - index
 - arbres B+
 - hachage

Objectifs des SGBD (rappel)

- Contrôle intégré des données
 - Cohérence (transaction) et intégrité (CI)
 - partage
 - performances d'accès
 - sécurité
- Indépendance des données
 - logique : cache les détails de l'organisation conceptuelle des données (définir des vues)
 - physique : cache les détails du stockage physique des données (accès relationnel vs chemins d'accès physiques)

Architecture d'un SGBD



Le SGBD gère son espace mémoire et disque :
Gestion dédiée plus efficace qu'un OS généraliste.



Stockage en pages

- Les données sont stockées sur un support persistant
 - Disque magnétique, flash (carte SD, disque SSD), bande magnétique,
- Gestion de l'espace disque
 - L'unité de stockage est : **la page**
 - La taille d'1 page est fixe pour un SGBD (souvent 8Ko, parfois plus)
- Opérations élémentaires pour accéder aux données stockées :
 - lire une page, écrire une page



Enregistrement et **ROWID**

- Un **enregistrement** = donnée stockée
= une ligne d'une table
- Stocké dans les **pages** d'un fichier
- Un enregistrement a un identificateur unique :
 - **ROWID = adresse localisant un enregistrement**
 - **ROWID = (nomFichier, n° page, position)**

Avantage du ROWID : accès direct à la page contenant un enregistrement.

Méthode d'accès

- Façon d'organiser les enregistrements dans les pages d'un fichier
- Impact important sur les performances.
 - Elle dépend du type de requêtes.
 - Ex. OLTP (ligne) vs. OLAP (colonne).
 - Elle dépend aussi du type de mémoire.
 - Ex. Flash très lent écriture.
- Un SGBD offre plusieurs **méthodes d'accès**.
 - Quelle méthode d'accès est la plus rapide ?

Coût d'une opération

- Le coût d'une opération SQL = durée
- Durée = temps de lecture et écriture des pages
+ temps de calcul relativement négligeable
- Unité de mesure du coût proportionnelle à la durée
 - Exple nombre de pages lues et écrites
- Le coût dépend de la méthode d'accès
 - Chaque méthode d'accès a un nombre de pages à lire différent
- **Utilité du coût**
 - **prévoir** la durée d'une opération SQL
 - Choisir une méthode d'accès rapide



Organisation séquentielle

- Non trié :
 - Très facile à maintenir en mise à jour
 - Parcourir toutes les pages quelque soit la requête
- Trié :
 - Un peu plus difficile à maintenir
 - Parcours raccourci car on peut s'arrêter dès qu'on a les données cherchées

presque toujours un compromis à faire entre lecture et écriture

Organisation indexée (1/4)

Clé

- Clé de l'index = un ou plusieurs attributs
- Stockage en **regroupant ou triant** les données selon la clé de l'index
- Les enregistrements ayant la même valeur de clé:
 - sont contigus dans une page
 - et dans les pages contigües suivantes si nécessaire
- Retrouver des enregistrements à partir d'une **clé de recherche**
- Exemple d'une bibliothèque
 - Les ouvrages sont rangés par thème
 - On peut retrouver les ouvrages d'un thème en allant vers l'étagère de ce thème

Organisation indexée (2/4)

Index plaçant

- Définir l'organisation des données lors de la création de la table.
- **Tri**
 - create table...organization index
- **Regroupement**
 - create cluster...
- Index pour atteindre rapidement la page correspondant à la valeur de la clé
 - Exple bibliothèque : accès plus rapide si on connaît l'association thème → étagère
- Evidemment, pas plus d'un index plaçant par table
 - Appelé index principal

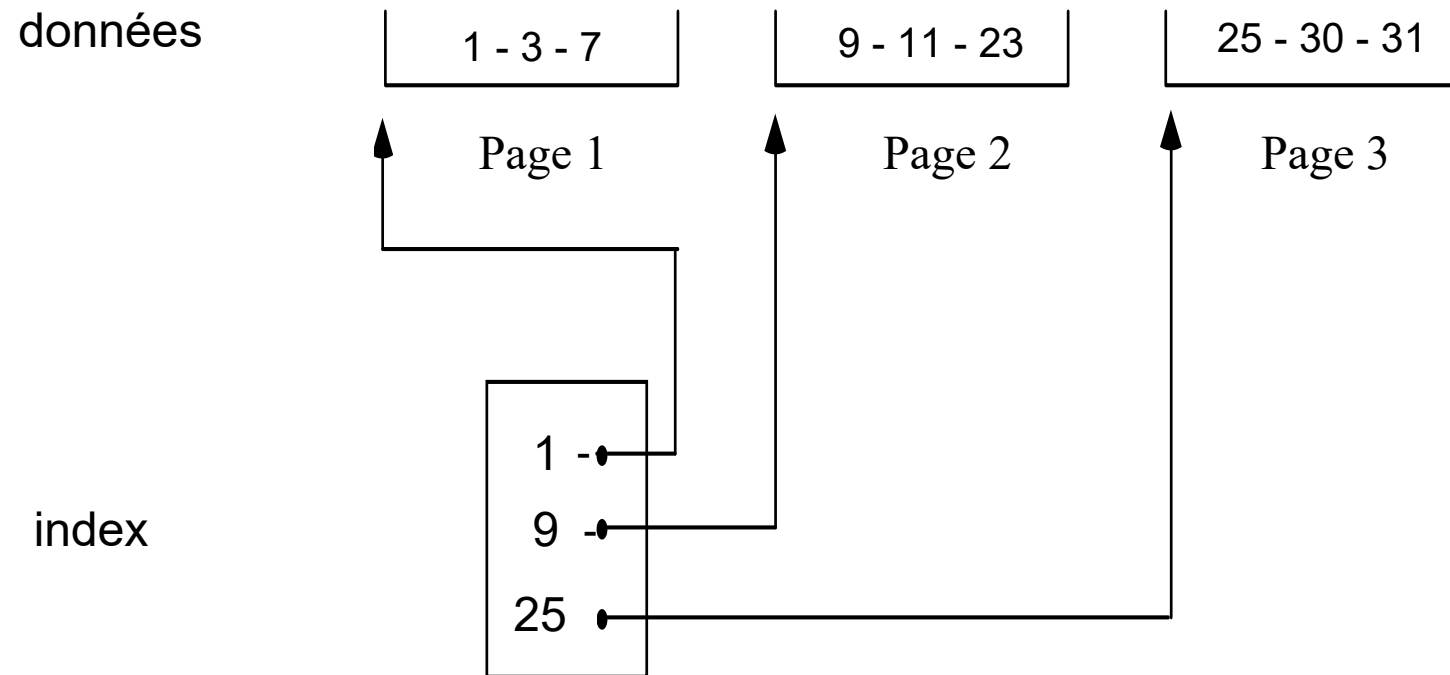
Organisation indexée (3/4)

Index plaçant **non dense**

- Objectif: obtenir un index occupant moins de place lorsque les données sont triées
- Méthode: enlever des entrées. Ne garder que les entrées nécessaires pour atteindre la page de données contenant les enregistrements recherchés
 - Garder l'entrée ayant la plus petite (ou la plus grande) clé de chaque page.
 - Ne pas indexer 2 fois la même clé dans 2 pages consécutives
- Inconvénient: toutes les valeurs de l'attribut indexé ne sont pas dans l'index. Cf diapo (index couvrant une requête)
- Rmq: Un index contenant **toutes** les valeurs de la clé est dit **dense**

Organisation indexée (4/4)

Exemple d'index plaçant **non dense**





Index non plaçant (1/4)

Définition

- Un index **non plaçant** est dit secondaire
 - Structure auxiliaire "à côté" d'une table
- Permet d'indexer des données quelle que soit l'organisation existante du stockage
 - Données stockées sans être triées
 - Données triées selon un attribut **autre** que celui indexé
- Définir un index non plaçant en SQL
 - **create index** *Nom* **on** *NomTable* (*Attributs*);
 - **create index** IndexAge **on** Personne(âge);



Index non plaçant (2/4)

Entrée d'un index

- Une **entrée** = structure associative
clé → ROWID des enregistrements

Le ROWID permet de lire la page
contenant l'enregistrement



Index non plaçant (3/4)

Entrée unique

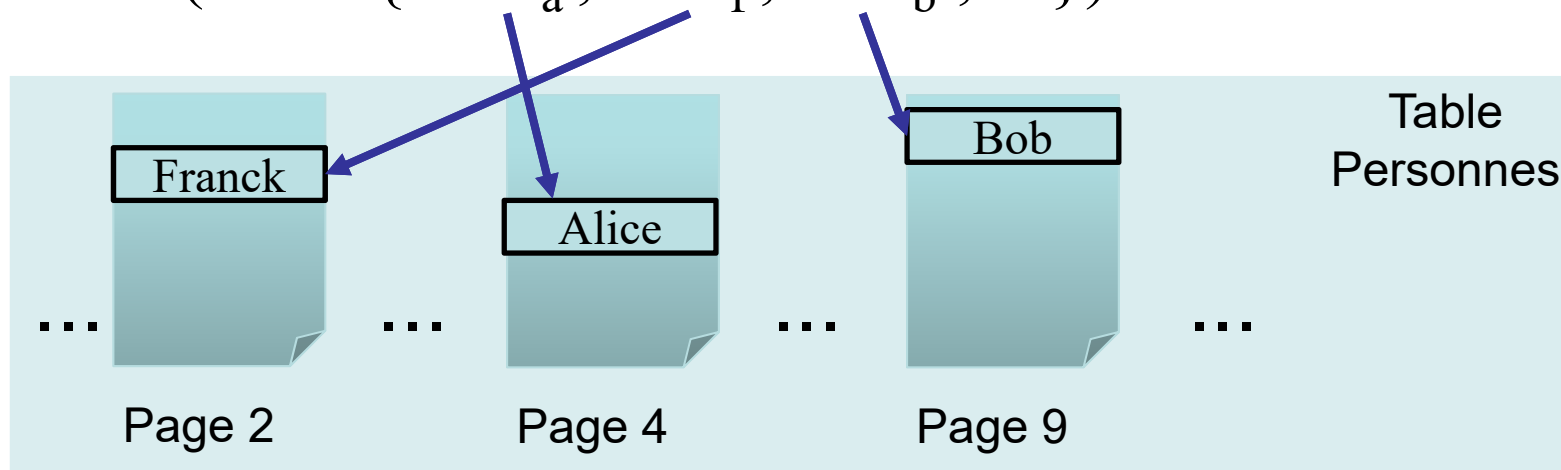
- Un index est dit **unique** si l'attribut indexé satisfait une contrainte d'unicité
 - Contrainte d'intégrité dans un *create table* ...
 - *Primary key (attribut1, ...)*
 - *Unique (attribut1, ..)*
- Entrée d'index = (clé, **ROWID**)
 - **un seul** enregistrement par valeur



Index non plaçant (4/4)

Entrée multiple

- Lorsque l'attribut indexé n'est **pas** unique
- Entrée d'index = (clé, **liste** de ROWID)
 - **plusieurs** enregistrements par valeur
- Exple de l'entrée 18 pour indexAge
 - $(18 \rightarrow \{ row_a, row_f, row_b, ... \})$





Accès par index

- Sert pour évaluer une sélection
 - une **égalité**: prénom = 'Alice'
 - l'accès est dit 'ciblé' si l'attribut est unique
 - un **intervalle** : age between 7 and 77
 - une **inégalité** : age > 18 <, >, ≤, ≥
 - une comparaison de **préfixe** : prénom like 'Ch%'
 - Rmq : un index ne permet **pas** d'évaluer une comparaison de suffixe. Exple prénom like '%ne'
 - Rmq: si les entrées de l'index ne sont pas triées (cas d'une table de hachage), seule l'égalité est possible



Index couvrant une requête

- Un index **couvre une requête** s'il est possible d'évaluer la requête **sans** lire les données
- Tous les attributs mentionnés dans la requête doivent être indexés
- Index couvrant une sélection
 - Pour chaque prédicat p de la clause *where*, il faut un index capable d'évaluer p .
- Index couvrant une projection
 - Pour chaque attribut de la clause *select*, il faut un index
- Avantage
 - Evite de lire les données, évaluation plus rapide d'une requête
- Rmq : Un index plaçant non dense n'est jamais couvrant car il ne contient pas toutes les valeurs de l'attribut indexé



Index composé

- Clé composée considérée comme une clé simple formée de la concaténation des attributs
 - create index on NomTable(a1, a2, a3, ..., an)
- Sélection par **préfixe** de la clé composée
 - Il existe n préfixes : (a1), (a1,a2) ,, (a1,a2, ...,an)
 - Rmq: (a2,a3) n'est pas un préfixe
- Accès par index composé pour une requête
 - **Règle d'utilisation:** **une seule traversée** de l'index depuis la racine vers une feuille, suivie éventuellement d'un parcours latéral des feuilles
 - On appelle (p1, p2, ...p_m) les attributs mentionnés dans le prédicat de sélection
 - On détermine le plus grand préfixe de la clé composée : (p1, p2, ...p_k) tq:
 - Prédicat d'**égalité** pour tous les attributs p1 à p_{k-1}
 - Égalité, inégalité ou comparaison de préfixe pour le dernier attribut p_k
 - Les prédicats p_{k+1} à p_m sont évalués par un *filtre* après l'accès à l'index

Index composé : Exemples

- Création d'un index : create index I1 on Personne(âge, ville)
- Utilisable pour les requêtes : Select * from Personne ...
 - Where âge = 18 and ville = 'Paris'
 - Where âge = 18 and ville like 'M%'
 - **racine** → 18 → première ville commençant par M
 - puis avec **parcours latéral** des feuilles situées à droite
 - Where âge ≥ 18
 - **racine** → 18 → **ville1** puis **parcours latéral** des feuilles situées à droite.
 - Where âge ≥ 18 and ville = 'Paris':
 - index seulement pour âge ≥ 18 : **racine** → 18 → **Paris**
 - puis **parcours latéral** des feuilles situées à droite et **filtre** (ville='Paris')
 - Ne **pas** traverser les branches 19 → Paris, 20 → Paris ... 99 → Paris: trop de traversées
- **Parcours latéral** de toutes les feuilles de l'index pour :
 - Where ville = 'Paris'
 - **Exception** : accès *Index Skip Scan* si le domaine du premier attribut est petit



Choix entre un accès séquentiel ou un accès par index

- Définir un ou plusieurs index
- Poser des requêtes. Le SGBD utilise automatiquement les index existants
 - s'il estime que c'est plus rapide que le parcours séquentiel des données.
 - Décision basée sur des règles heuristiques ou sur une estimation de la durée de la requête (voir TME)
- L'utilisateur peut **forcer/interdire** le choix d'un index

Select *

From Personne Where age < 18

- Devient

Select /*+ **index(Personne IndexAge)** */ *

From Personne Where age < 18

- Syntaxe d'une directive :

- **index**(*NomTable NomIndex*)
- **no_index**(*NomTable NomIndex*)
- **index_combine**(*NomTable NomIndex1 NomIndex2*)

Index hiérarchisé

- Lorsque le nombre d'entrées de l'index est très grand
- L'ensemble des entrées d'un index peuvent, à leur tour, être indexées. Cela forme un index hiérarchisé en plusieurs niveaux
 - Le niveau le plus bas est l'index des données
 - Le niveau n est l'index du niveau $n+1$
 - Intéressant pour gérer efficacement de gros fichiers

Arbre B+

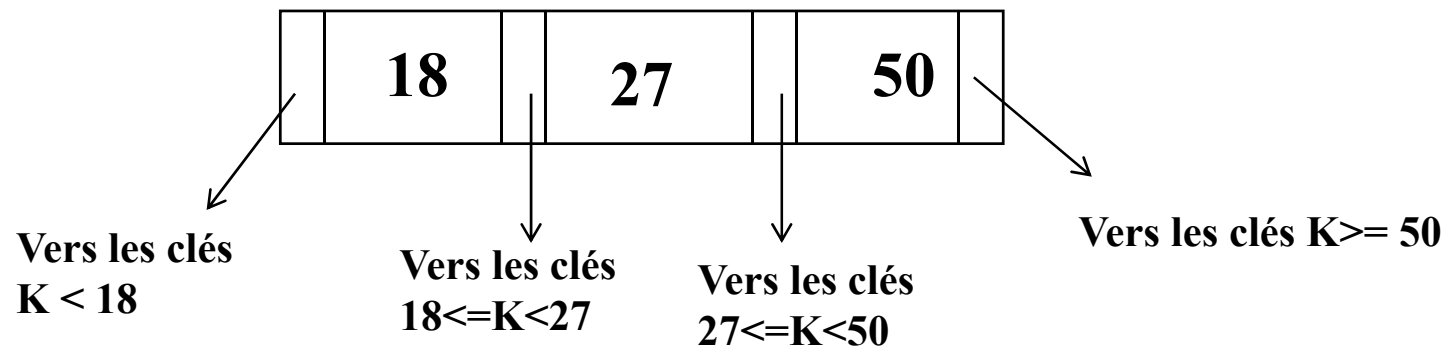
- Les arbres B+ sont des index hiérarchiques
- Ils améliorent l'efficacité des recherches
 - L'arbre est peu profond.
 - Accès rapide à un enregistrement : chemin court de la racine vers une feuille
 - Rmq: l'arbre peut être très large, sans inconvénient
 - L'arbre est toujours équilibré
 - *Balanced tree* en anglais
 - Tous les chemins de la racine aux feuilles ont la même longueur
 - L'arbre est suffisamment compact
 - Peut souvent tenir en mémoire
 - Un noeud est au moins à moitié rempli

Arbre B+ : coût d'accès

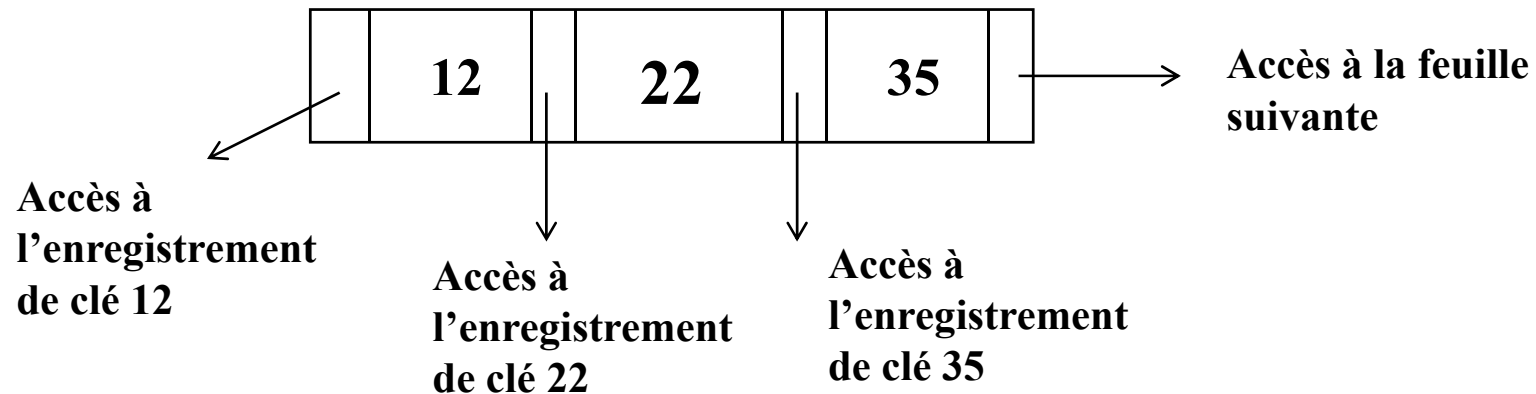
- Le **coût d'accès** est
 - proportionnel à la longueur d'un chemin
 - Identique quelle que soit la feuille atteinte
 - **coût d'accès prévisible**
- **Avantage :**
 - permet d'estimer le coût d'accès, a priori, pour décider d'utiliser ou non un index
- **Mesure du coût:**
 - Nombre de nœud lus / écrits
 - = Nombre de pages de données lues / écrites

Arbre B+

- Les **nœuds internes** servent à atteindre une feuille



- Les **feuilles** donnent accès aux enregistrements



Arbre B+ : 3 types de nœuds

- Racine
 - point de **départ** d'une recherche
- Nœud intermédiaire
 - Peut contenir une valeur pour laquelle il n'existe aucun enregistrement
- Feuille
 - Les feuilles contiennent **toutes les clés** pour lesquelles il existe un enregistrement
 - Les feuilles contiennent **seulement les clés**
(et aucune autre valeur de clé)



Ordre d'un arbre, degré d'un noeud

- La capacité d'un nœud de l'arbre s'appelle l'**ordre**
- Un arbre-B+ est d'**ordre d** ssi
 - Pour un nœud intermédiaire et une feuille : $d \leq n \leq 2d$
 - Pour la racine: $1 \leq n \leq 2d$
- Degré sortant d'un nœud
 - Un nœud intermédiaire (et la racine) ayant **n** valeurs de clés a **n+1** pointeurs vers ses fils
 - Une feuille n'a pas de fils

Nombre de clés dans les feuilles

- Dépend de l'ordre d et du nombre de niveaux p
- Nombre maxi de clés dans l'arbre
- Arbre à 1 niveau (arbre réduit à sa seule racine): $2d$ clés maxi
- Arbre à 2 niveaux :
 - racine: $2d$ clés maxi
 - $2d+1$ feuilles, soit $2d \times (2d+1)$ clés maxi dans les feuilles
- Arbre à p niveaux :
 - Nbre maxi de clés dans les feuilles: $2d(2d+1)^{(p-1)}$
- En pratique, un arbre B+ a rarement plus de 4 niveaux car d est grand (de l'ordre de la centaine)
- Nombre mini de clés dans les feuilles : ...

Arbre-B⁺ : chainage des feuilles

- But: supporter les requêtes d'intervalle
 - Exple de requête: ... where age between 18 and 25
 - Traverser l'index pour atteindre une borne de l'intervalle, puis parcours séquentiel des feuilles
- Chainage double pour supporter les requêtes avec une inégalité
 - Ex: ... where age < 6

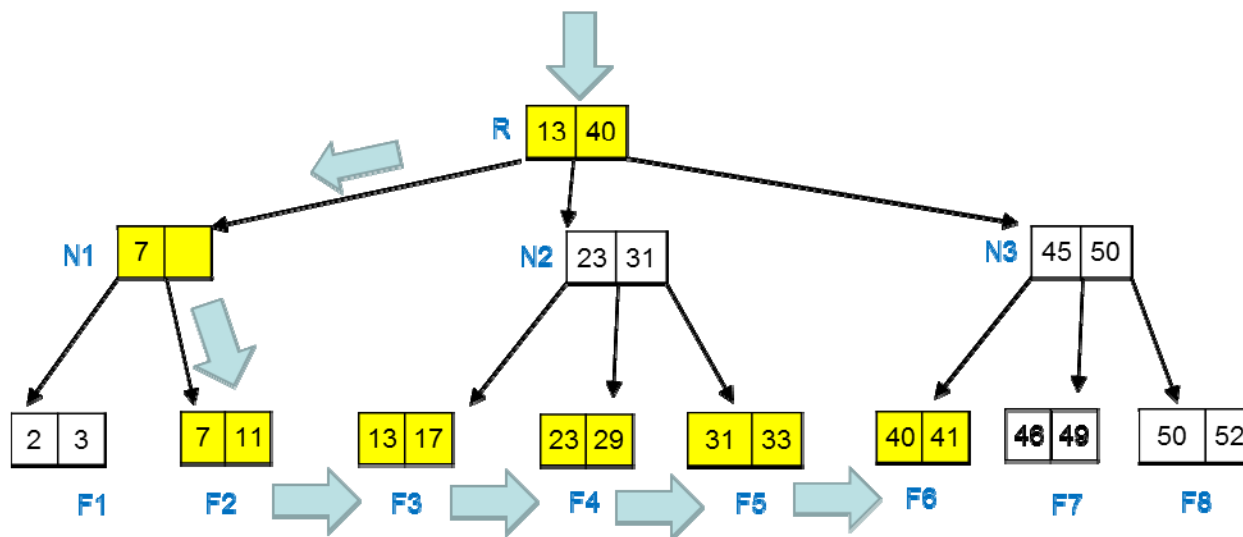
Parcours du chainage

- **Avantage :**
 - lire un seul chemin (moins de lectures)

Chainage des feuilles

Select * from Personne
Where age between 10 and 40

Légende : Noeud lu





Insertion

éclatement d'une feuille

- Rechercher la feuille où insérer la nouvelle valeur.
- Insérer la valeur dans la feuille s'il y a de la place.
 - Maintenir les valeurs triées dans la feuille
- Si la feuille est pleine (2d valeurs), il y a éclatement.
Il faut créer un nouveau nœud :
 - Insérer les $d+1$ premières valeurs dans le nœud original, et les d autres dans le nouveau nœud (à droite du premier).
 - La plus petite valeur du nouveau nœud est insérée dans le nœud parent, ainsi qu'un pointeur vers ce nouveau nœud.
 - Résultat correct : les deux feuilles sont au moins à moitié pleines



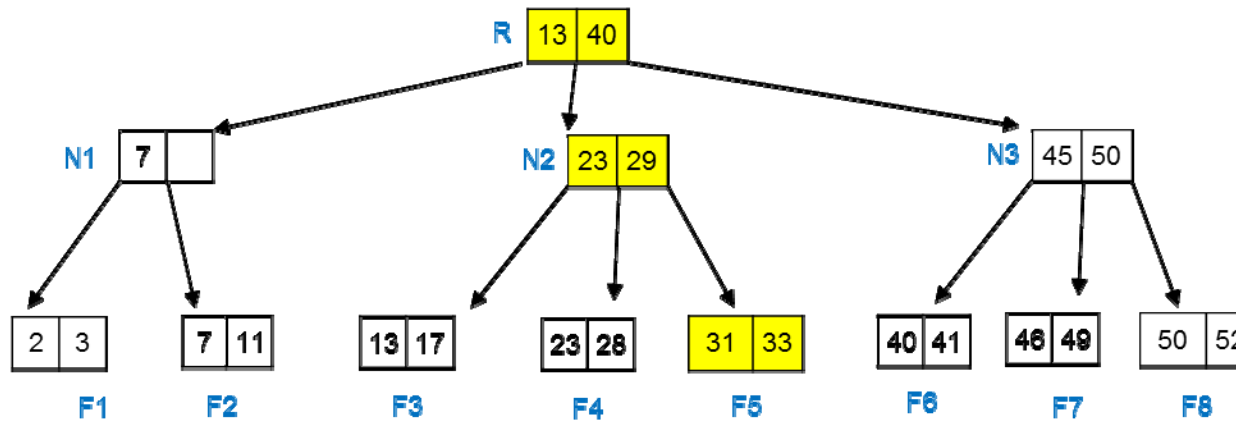
Insertion

éclatement d'un nœud intermédiaire

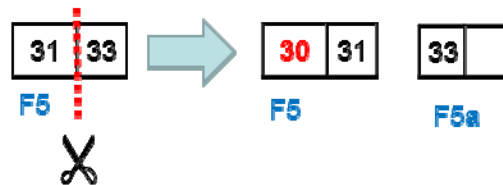
- S'il y a éclatement dans le parent, il faut créer un nouveau nœud frère M, à droite du premier
 - Les **d premières** valeurs restent dans le nœud N
 - Les **d dernières** vont dans le nouveau nœud M
 - La **valeur restante est insérée dans le parent** de N et M pour atteindre M
 - Résultat correct: M et N ont bien chacun $d+1$ fils
- Les éclatements peuvent se propager jusqu'à la racine et créer un nouveau niveau pour l'arbre.

Insertion

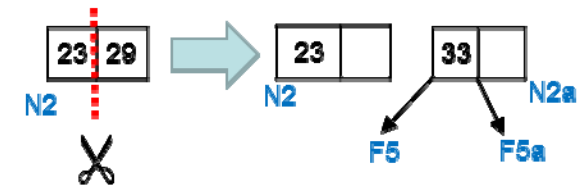
Insérer 30 ?



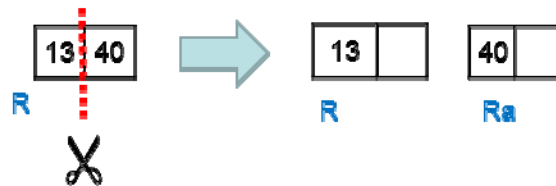
insérer 30 dans F5



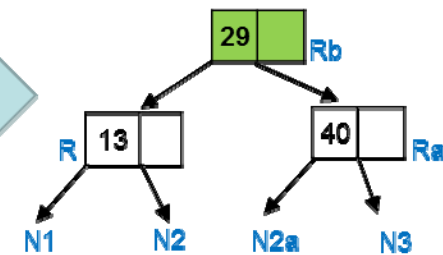
Puis insérer 33 dans N2



Puis insérer 29 dans R



Puis Nouvelle racine Rb





Suppression

- Supprimer la valeur de la feuille (et le pointeur vers l'enregistrement)
- Si la feuille est encore suffisamment pleine, il n'y a rien d'autre à faire.
- Sinon, redistribuer les valeurs avec une feuille **ayant le même parent**, afin que toutes les feuilles aient le nombre minimum de valeurs requis.
 - Ajuster, dans le nœud père, la valeur délimitant les 2 nœuds impliqués dans la redistribution
- Si la redistribution est **impossible**, alors fusionner 2 feuilles
 - Supprimer une feuille et la 'décrocher' en supprimant une valeur dans son père.
- Si le parent n'est pas suffisamment plein, appliquer récursivement l'algorithme de suppression
- Remarque1 : la propagation récursive peut entraîner la perte d'un niveau.

Redistribution entre 2 nœuds intermédiaires

- Redistribution entre deux nœuds intermédiaires ayant le même parent
- La valeur contenue dans le parent participe à la redistribution
 - la valeur du parent "descend" dans le nœud à remplir,
 - la valeur à redistribuer "monte" dans le parent

Résumé des opérations

- Insertion
 - simple
 - éclatement
 - d'une feuille
 - d'une feuille puis éclatement d'ancêtres
- Suppression
 - simple
 - redistribution
 - entre 2 feuilles
 - entre 2 feuilles puis redistribution ou fusion d'ancêtres
 - fusion
 - entre 2 feuilles
 - entre 2 feuilles puis redistribution ou fusion d'ancêtres
- Rmq
 - Toujours insérer/supprimer une clé au niveau des **feuilles**
 - Jamais de redistribution lors d'une insertion. L'éclatement est préférable pour faciliter les prochaines insertions.

Avantages et Inconvénients

- Avantages des organisations indexées par arbre b (b+) :
 - Régularité = pas de réorganisation du fichier nécessaires après de multiples mises à jour.
 - Lecture séquentielle rapide: possibilité de séquentiel physique et logique (trié)
 - Accès rapide en 3 E/S pour des fichiers de 1 M d'articles
- Inconvénients :
 - Les suppressions génèrent des trous difficiles à récupérer
 - Avec un index non plaçant, l'accès à plusieurs enregistrements (intervalle ou valeur non unique) aboutit à lire plusieurs enregistrements non contigus. Lire de nombreuses pages non contiguës dure longtemps
 - Taille de l'index pouvant être importante.

Exercice Arbre B+

- Un arbre B+ a 3 niveaux. La racine et les nœuds intermédiaires ont 1 ou 2 clés, les feuilles 2 ou 3 clés.
- Les feuilles ont les clés 1, 4, 9, 16, 25, 36, 49, 54, 61, 70, 81, 84, 87, 88, 95, 99
- Les nœuds intermédiaires ont les clés 9, 54, 70, 88
- La racine contient 2 clés. Choisir les valeurs les plus petites possibles parmi celles des feuilles
- Représenter l'arbre, puis insérer la clé 32

Exercice (suite)

- Etant donné les valeurs intermédiaires, on en déduit :
 - $v < 9$: (1, 4) peut tenir dans une seule feuille
 - $9 \leq v < 54$: 9, 16, 25, 36, 49
 - 5 valeurs nécessitent deux feuilles: (9, 16) et (25,36,49)
 - $54 \leq v < 70$: (54, 61) peut tenir dans une seule feuille
 - $70 \leq v < 88$: 70, 81, 84, 87
 - 4 valeurs nécessitent deux feuilles (70, 81) et (84,87)
 - $v \geq 88$: (88, 95, 99) peut tenir dans une seule feuille
- Les valeurs 9 et 54 ne peuvent pas être dans le même nœud intermédiaire car il y a 2 feuilles dans l'intervalle $[9, 54[$
 - Idem pour $[70,88[$
- Donc les trois nœuds intermédiaires sont (9) (54,70) (88)
- Racine : (25, 84)

Complément : Arbre B+ distribué

Quels sont les travaux de recherche sur les arbres B+ distribués ?

Efficient B-tree Based Indexing for Cloud Data Processing

Sai Wu #¹, Dawei Jiang #², Beng Chin Ooi #³, * Kun-Lung Wu §⁴

#*School of Computing, National University of Singapore, Singapore*

^{1,2,3}{wusai, jiangdw, ooibc}@comp.nus.edu.sg

§*IBM T. J. Watson Research Center*

⁴klwu@us.ibm.com

Conference : Very Large DataBases

2010

<http://www.vldbarc.org/pvldb/vldb2010/papers/R107.pdf>

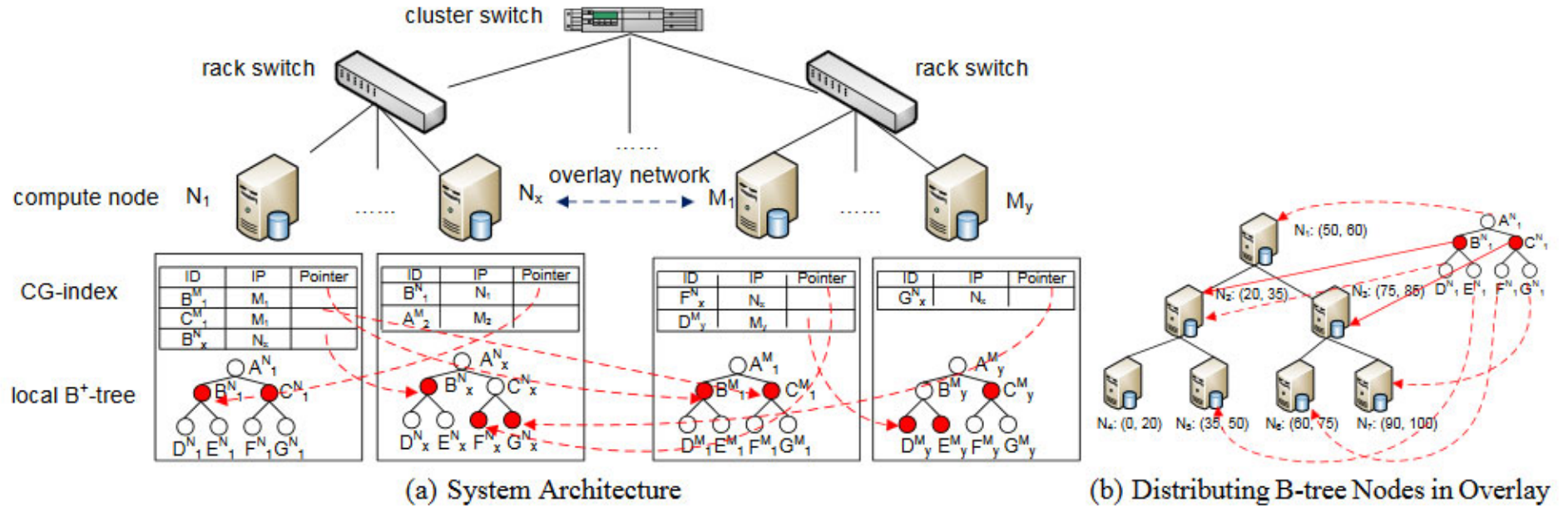


Figure 1: System Overview

Our approach can be summarized as follows. First, we build a **local B⁺-tree** index for each compute node which only indexes data residing on the node. Second, we **organize** the compute nodes as a structured overlay and **publish** a portion of the **local B⁺-tree nodes** to the overlay for efficient query processing. Finally, we propose an adaptive algorithm to select the published B⁺-tree nodes according to query patterns. We conduct extensive **experiments on Amazon's EC2**, and the results demonstrate that our indexing scheme is **dynamic, efficient and scalable**.

Ce travail a-t-il été réutilisé?
Pérennité dans le temps ?

Cité par d'autres travaux d'autres auteurs ?

Efficient B-tree based indexing for cloud data processing

[S Wu](#), [D Jiang](#), [BC Ooi](#), [KL Wu](#) - [Proceedings of the VLDB Endowment](#), 2010 - [dl.acm.org](#)

A Cloud may be seen as a type of flexible computing infrastructure consisting of many compute nodes, where resizable computing capacities can be provided to different customers. To fully harness the power of the Cloud, efficient data management is needed to handle huge volumes of data and support a large number of concurrent end users. To achieve that, a scalable and high-throughput indexing scheme is generally required. Such an indexing scheme must not only incur a low maintenance cost but also support parallel ...



☆ 57 Cited by 164 Related articles All 18 versions 🔗

Showing the best result for this search. [See all results](#)

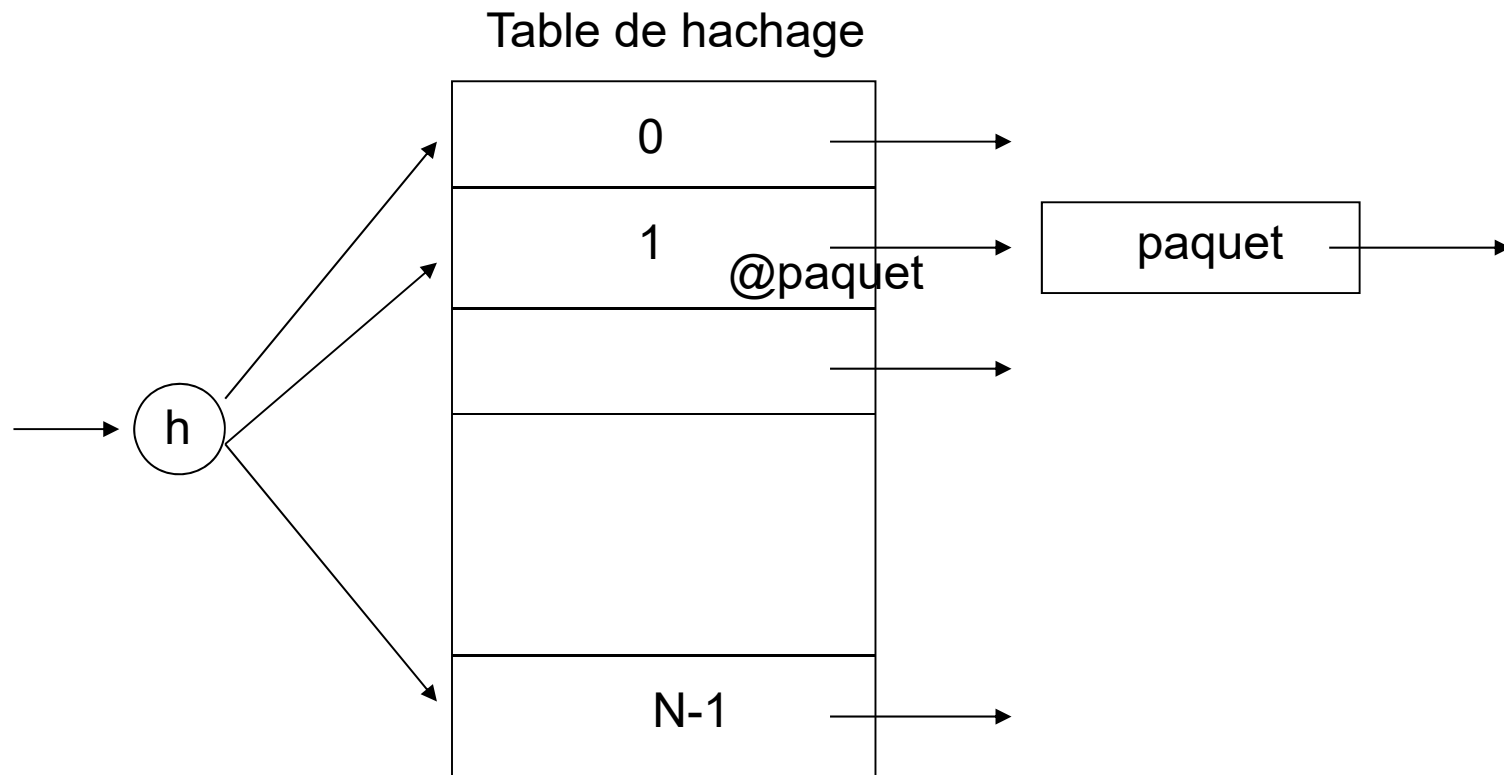
Index par hachage

Organisations par Hachage

- Fichier haché statique (Static hashed file)
 - Fichier de taille fixe dans lequel les articles sont placés dans des paquets dont l'adresse est calculée à l'aide d'une fonction de hachage fixe appliquée à la clé.
 - On peut rajouter une indirection : table de hachage.
 - $H(k)$ donne la position d'une cellule dans la table.
 - Cellule contient adresse paquet
 - Souplesse (ex. suppression d'un paquet)
- Différents types de fonctions :
 - Conversion en nb entier
 - Modulo P
 - Pliage de la clé (combinaison de bits de la clé)
 - Peuvent être composées

Défi : Obtenir une distribution uniforme pour éviter les collisions (saturation)

Hachage statique



Hachage statique

- Très efficace pour la recherche (condition d'égalité) : on retrouve le bon paquet en une lecture de bloc.
- Bonne méthode quand il y a peu d'évolution
- Choix de la fonction de hachage :
 - Mauvaise fonction de hachage ==> Saturation locale et perte de place
 - Solution : autoriser les débordements

Techniques de débordement

- l'adressage ouvert
 - place l'article qui devrait aller dans un paquet plein dans le premier paquet suivant ayant de la place libre; il faut alors mémoriser tous les paquets dans lequel un paquet plein a débordé.
- le chaînage
 - constitue un paquet logique par chaînage d'un paquet de débordement à un paquet plein.
- le rehachage
 - applique une deuxième fonction de hachage lorsqu'un paquet est plein, puis une troisième, etc..., toujours dans le même ordre.

Le chaînage est la solution la plus souvent utilisée. Mais si trop de débordement, on perd tout l'intérêt du hachage (séquentiel)

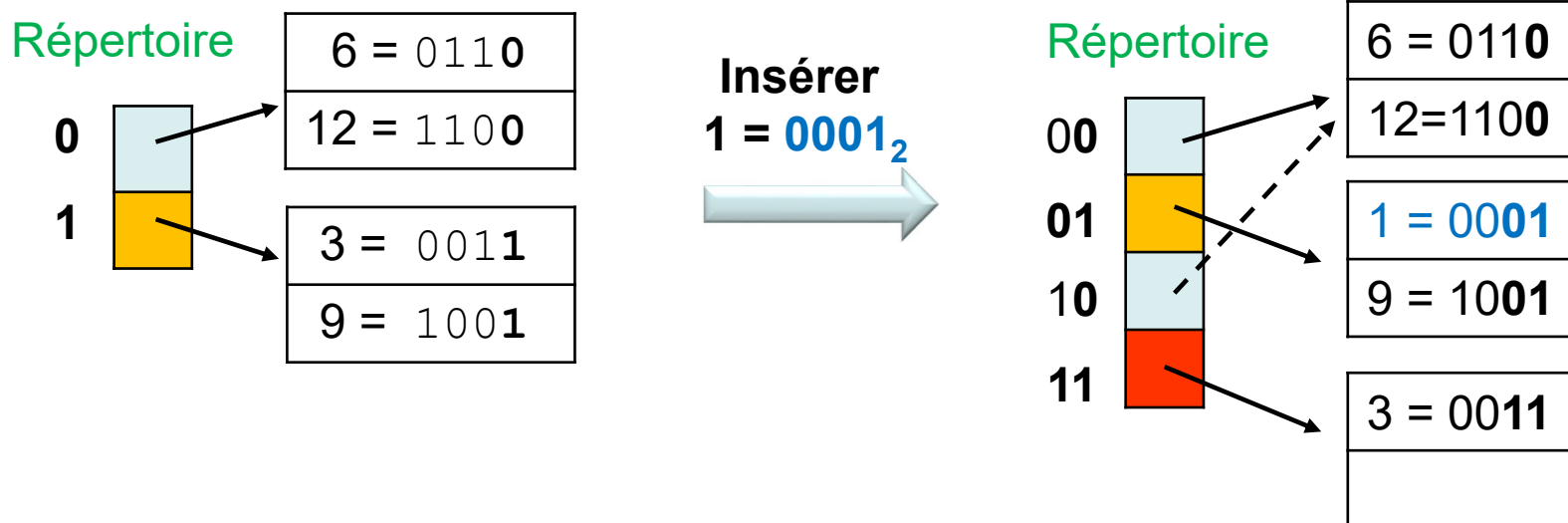
Hachage dynamique

- Hachage dynamique :
 - techniques permettant de faire grandir progressivement un fichier haché saturé en distribuant les enregistrements dans de nouvelles régions allouées au fichier.
- Deux techniques principales
 - Hachage extensible
 - Hachage linéaire

Hachage extensible

Hachage extensible

- Ajout d'un niveau d'indirection vers les paquets
 - répertoire = liste de pointeurs vers des paquets
- Jamais de débordement
 - Accès direct à tout paquet via le répertoire (i.e, une seule indirection)
 - La **taille** du répertoire peut varier



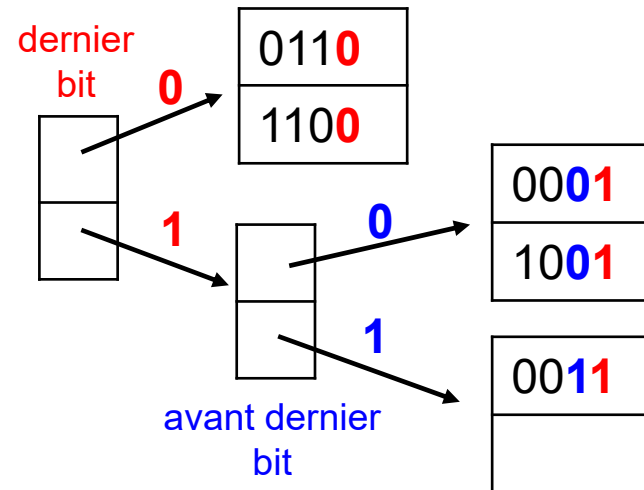
Hachage extensible

- Répertoire organisé (presque) comme un arbre à préfixe (*trie*)
 - on considérant la représentation binaire des valeurs à indexer
 - Hiérarchie basée sur le suffixe (au lieu du préfixe)
 - Le suffixe est « lu » droite à gauche, en commençant par le bit de poids le plus faible
 - longueur du suffixe utilisé pour l'indexation = longueur de la branche = **PROFONDEUR**

Exemple

3 = 0011
6 = 0110
9 = 1001
12 = 1100

1 = 0001





Hachage extensible

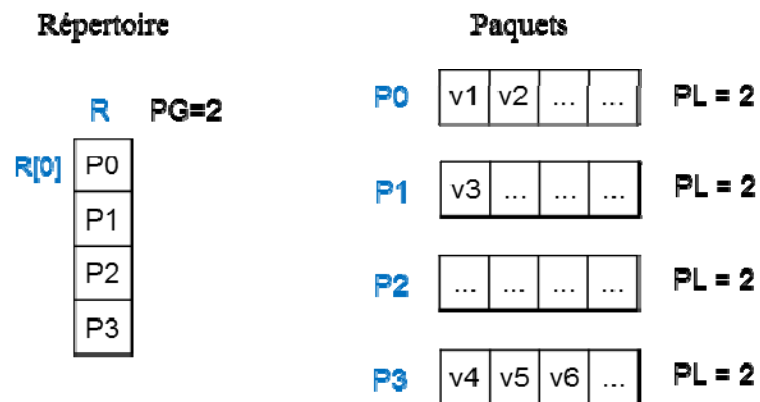
Profondeur Globale et Locale

- Profondeur globale (PG)
 - Sert à **atteindre** (retrouver) une entrée
 - En invoquant la fonction $h_{PG}(v) = v \bmod 2^{PG}$ pour lire la case $h_{PG}(v)$
 - Avantage d'utiliser la fonction modulo 2^{PG}
 - évite de re-répartir toutes les valeurs de la table quand on l'agrandit.
- Profondeur locale (PL)
 - Sert à **ne pas éclater tous les paquets** en même temps
 - Avantage: répartir parmi plusieurs insertions, le surcoût d'agrandissement de la table de hachage
 - Indique pour chaque paquet, s'il peut éclater "rapidement" sans agrandir le répertoire



Hachage extensible : notations

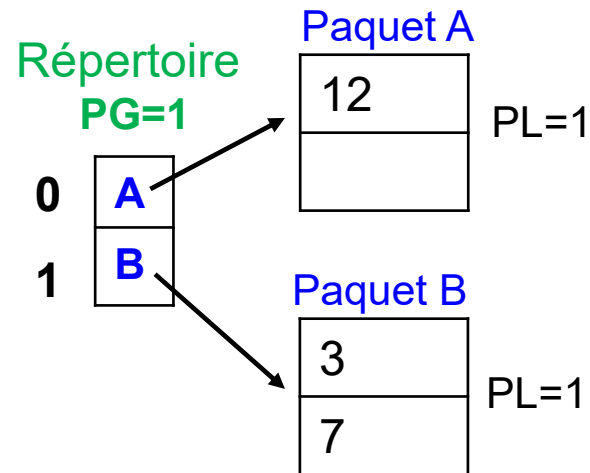
- Le répertoire est noté $\mathbf{R}[P_0, P_1, P_2, \dots, P_k] \mathbf{PG=pg}$ avec
 - P_i ... les noms d'un paquet,
 - pg la profondeur globale.
- Rmq : le répertoire contient k cases avec $k = 2^{pg}$
- Un paquet est noté $P_i(v_j, \dots, \dots) \mathbf{PL=pl}$ avec
 - P_i le nom du paquet, par exemple A,B, ... ,
 - V_j . les valeurs que contient le paquet,
 - pl la profondeur locale.
- On peut aussi préciser le contenu d'une case particulière du répertoire avec
 - $R[i]=L$ (avec $R[0]$ étant la 1^{ère} case)
- La valeur v se trouve le paquet référencé dans la case $R[v \text{ modulo } 2^{pg}]$



Hachage extensible

Création du répertoire

- Etat initial : N valeurs à indexer dans des paquets pouvant contenir p valeurs.
 - Il faut au moins N/p paquets
 - La taille initiale du répertoire est $k=2^{PG}$ tq $2^{PG} \geq N/p$
 - On a k paquets donc $PL = PG$ pour tous les paquets initiaux

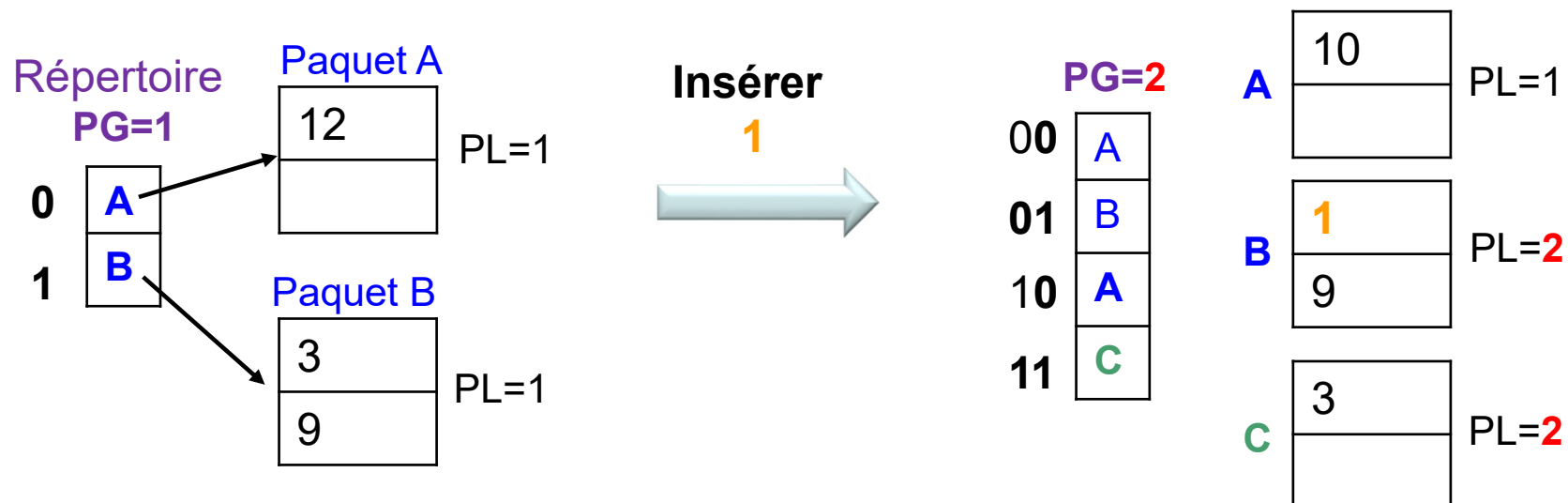
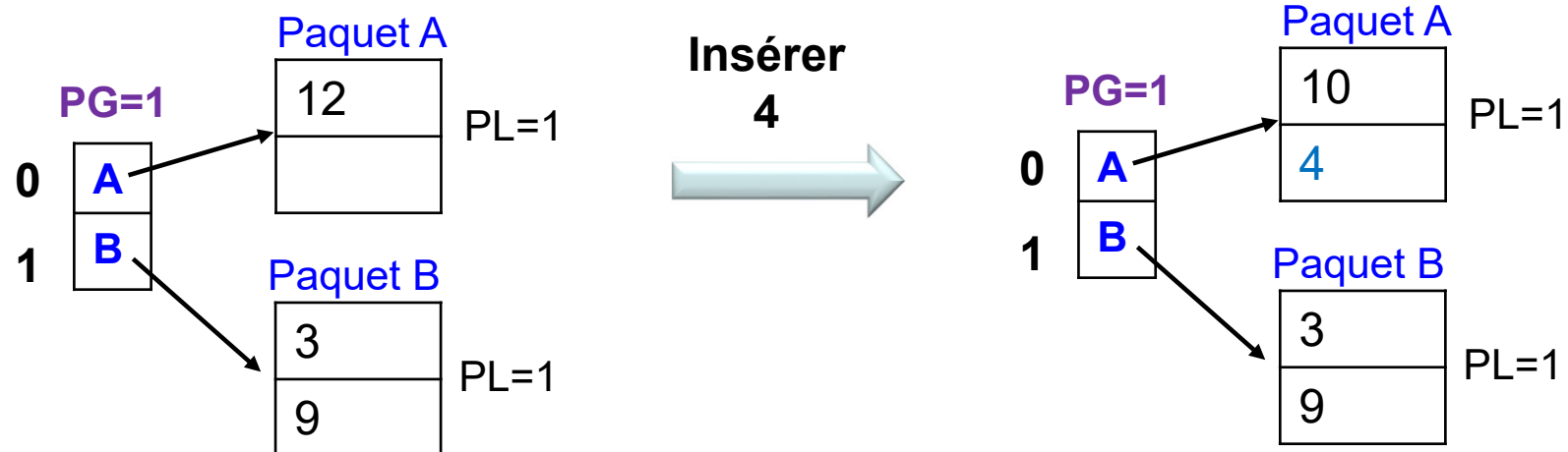




Hachage extensible : Insertion

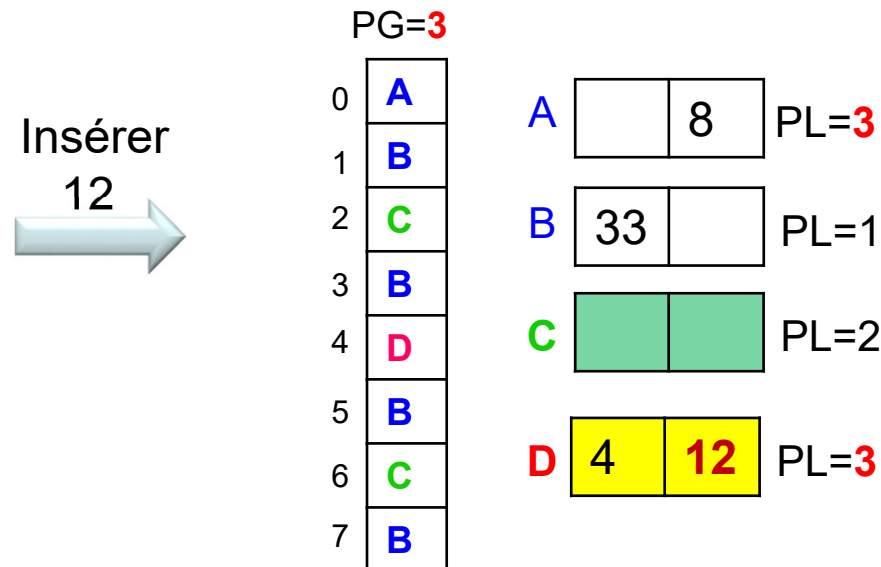
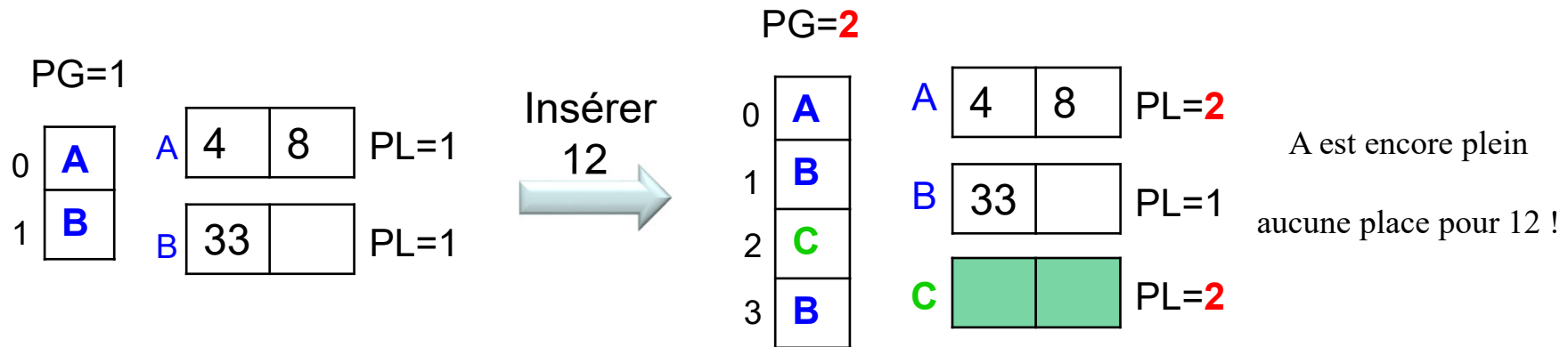
- Insertion de v dans le paquet P_i
- **Cas 1)** P_i n'est **pas** plein, insertion immédiate dans P_i
- **Cas 2)** P_i est **plein** et $PL_i < PG$ alors éclater P_i
 - Créer un nouveau paquet P_j et l'accrocher dans le répertoire
 - Remplacer l'adresse de P_i par celle de P_j dans la 2^{ème} case contenant P_i
 - Si 4 cases contiennent P_i , "recopier" le remplacement dans le reste du répertoire
 - Incrémenter les profondeurs locales de P_i et P_j ($PL = PL+1$)
 - Répartir les valeurs de P_i et v entre P_i et P_j
 - Si P_i est encore plein, réappliquer l'algo d'insertion : cas 2) ou 3)
- **Cas 3)** P_i est **plein** et $PL_i = PG$ alors doubler le répertoire
 - **Recopier** le contenu des k premières cases dans les k nouvelles cases suivantes
 - $PG = PG+1$ puis appliquer le **Cas 2)**

Hachage extensible : Insertions



Hachage extensible :

Insertion avec plusieurs éclatements





Hachage extensible

Suppression et fusion

- Lors d'une suppression, si un paquet P_i devient **vide** :
- Si $PL_i = PG$ alors
 - Fusionner P_i avec le paquet P_j référencé dans la case ayant le même suffixe que celle qui référence P_i
 - Suffixe commun (en base 2) de longueur $PG - 1$
 - Exple si $PL_i = PG = 3$, les cases ayant le même suffixe (de longueur 2) sont :
 - $R[0]$ et $R[4]$
 - $R[1]$ et $R[5]$
 - ...
 - $R[3]$ et $R[7]$
 - Supprimer P_i et le "décrocher" du répertoire : dans la case contenant P_i , remplacer P_i par P_j
 - Décrémenter la profondeur locale de P_j
- Sinon (on a $PL_i < PG$) : pas de fusion, P_i **reste vide**.
- Si pour tous les paquets restants on a $PL < PG$ alors ne garder que la première moitié du répertoire (division par 2) et décrémenter PG .

Hachage extensible

Exemples de suppression

- Suppression dans un paquet ayant $PL < PG$

Supprimer

12 = 1100_2

Répertoire

PG=2

00	A
01	B
10	A
11	C

A	1100	PL=1
B	0001	PL=2
	1001	
C	0011	PL=2

Hachage extensible

Exemple de suppression

- Suppression dans un paquet ayant **PL=PG**

Supprimer
12 = **1100**₂



Répertoire
PG=2

00	A
01	B
10	C
11	D

A	1100	PL=2
B	0001 1001	PL=2
C	0110	PL=2
D	0011	PL=2

Répertoire
PG=2

00	C
01	B
10	C
11	D

A	1100	PL=2
B	0001 1001	PL=2
C	0110	PL=1
D	0011	PL=2

Hachage extensible (suite)

- Avantage : accès à un seul bloc (si le répertoire tient en mémoire)
- Profondeur locale/globale :
 - modification **progressive** de la table de hachage
- Inconvénient :
 - interruption de service lors du doublement du répertoire.
 - Peut ne plus tenir en mémoire.
 - Si peu d'enregistrement par page, le répertoire peut être inutilement gros

Hachage linéaire

Hachage linéaire

- Principes
- Structure
- Insertion
- Exercice

Hachage linéaire

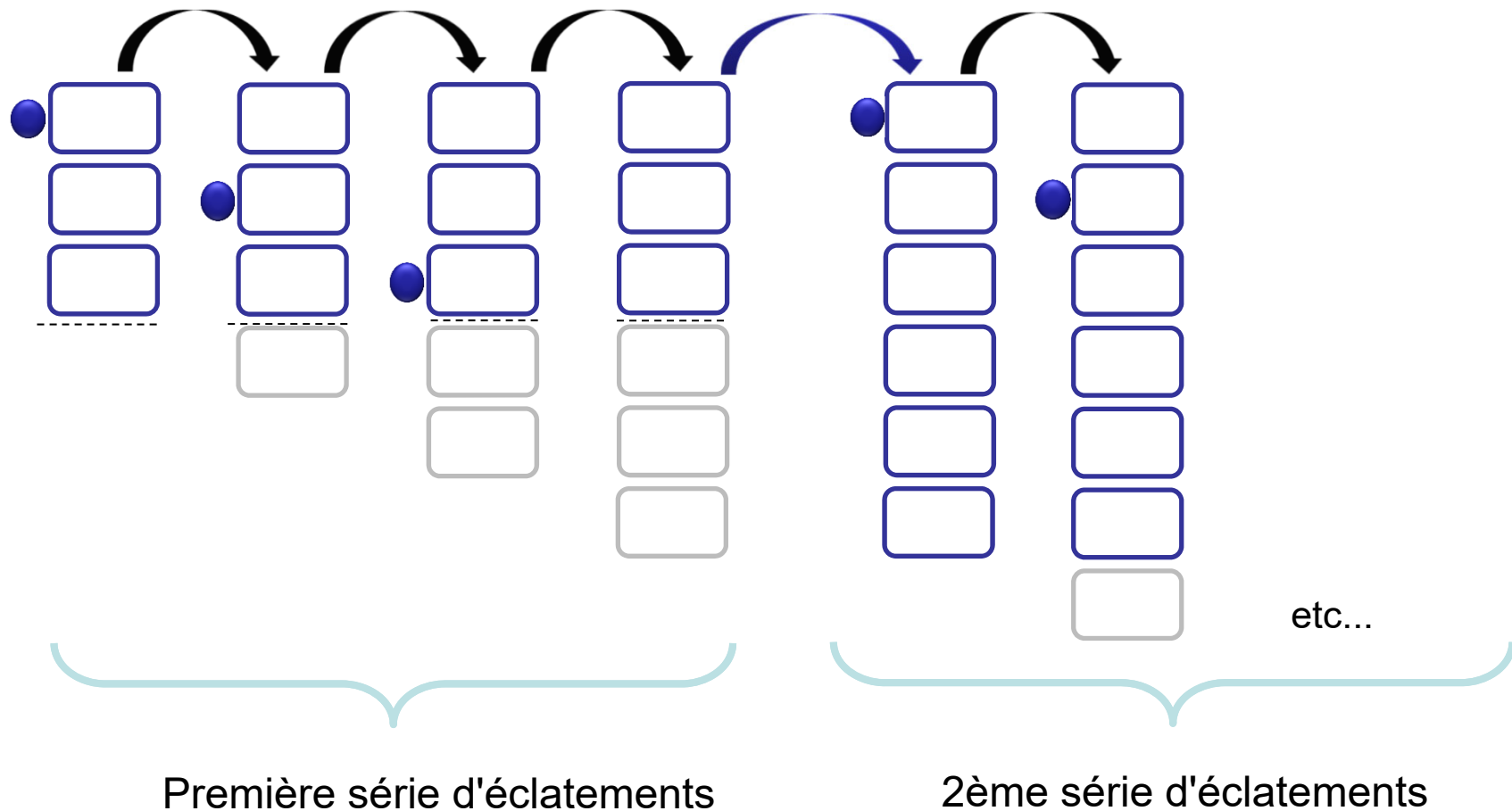
- Garantit que le nombre moyen d'enregistrements par paquet ne dépasse pas un certain seuil
 - taux d'occupation moyen d'un paquet $< 80\%$
- Ajouter les nouveaux paquets au fur et à mesure, en éclatant chaque paquet dans l'ordre, un par un du premier au $N^{\text{ième}}$ paquet.
- Avantage par rapport au hachage extensible
 - pas besoin de répertoire
 - plus rapide
- Inconvénient
 - débordement "temporaire"
 - Résolu lorsque le paquet qui déborde fini par éclater
 - débordement "permanent" possible de certains paquets
 - si les données ne sont pas uniformément réparties dans les paquets
 - un paquet peut être plein bien que le taux d'occupation moyen reste inférieur au seuil




Hachage linéaire

- Etat initial : N paquets
 - numérotés de 0 à $N-1$
- Avoir une famille de fonctions de hachage dont les domaines doublent : $N, 2N, 4N, \dots$:
 - $h_0(x) = x \bmod N$: fonction de hachage initiale
 - $h_i(x) = h(x) \bmod (2^i \cdot N)$: $i^{\text{ème}}$ fonction de hachage
- Marquer quel est le prochain paquet à éclater
 - noté p , $p = 0$
- Eclater les paquets par série de N éclatements
 - Quand les N paquets ont éclaté, on obtient $2N$ paquets
 - On démarre la série suivante pour éclater $2N$ paquets

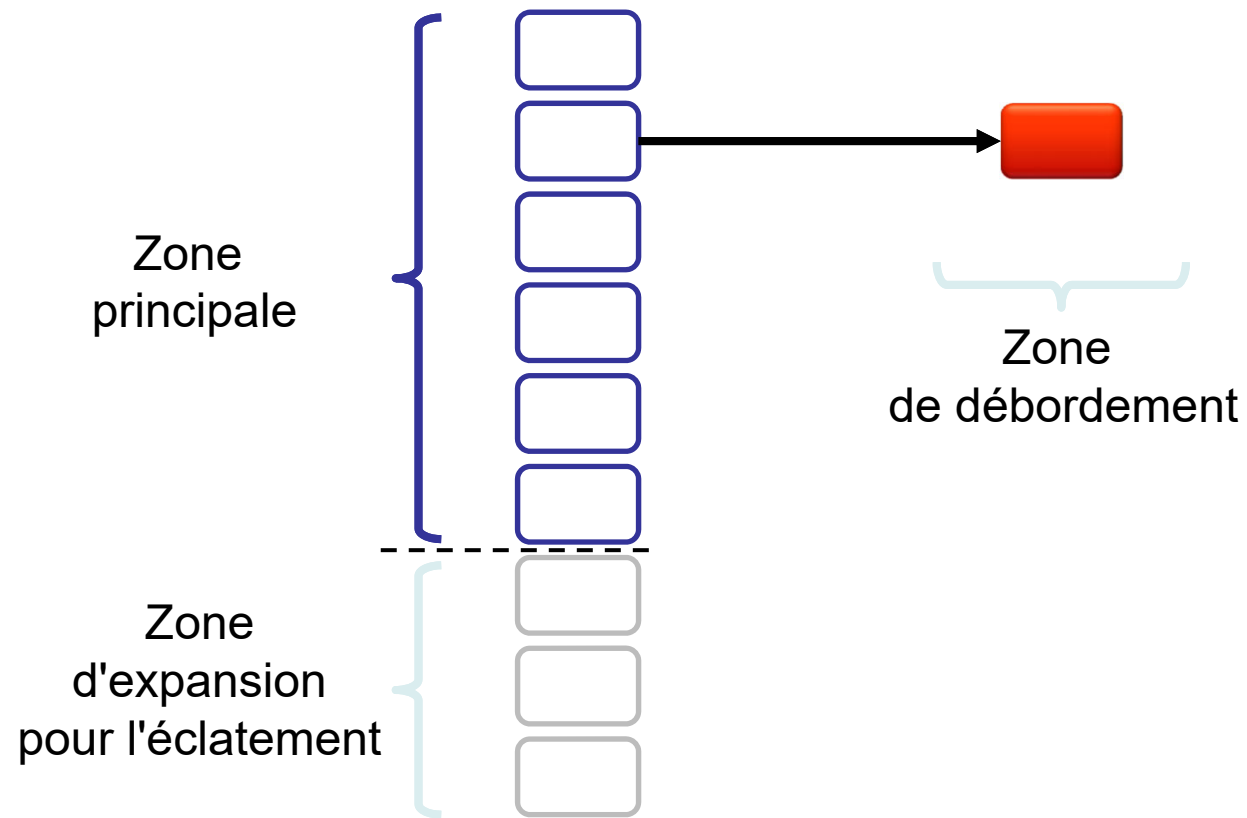
Illustration avec $N=3$



 : prochain éclatement



Zones





Hachage linéaire : taux d'occupation

- Taux d'occupation = V/C

V : Nombre total de **valeurs** dans **tous** les paquets

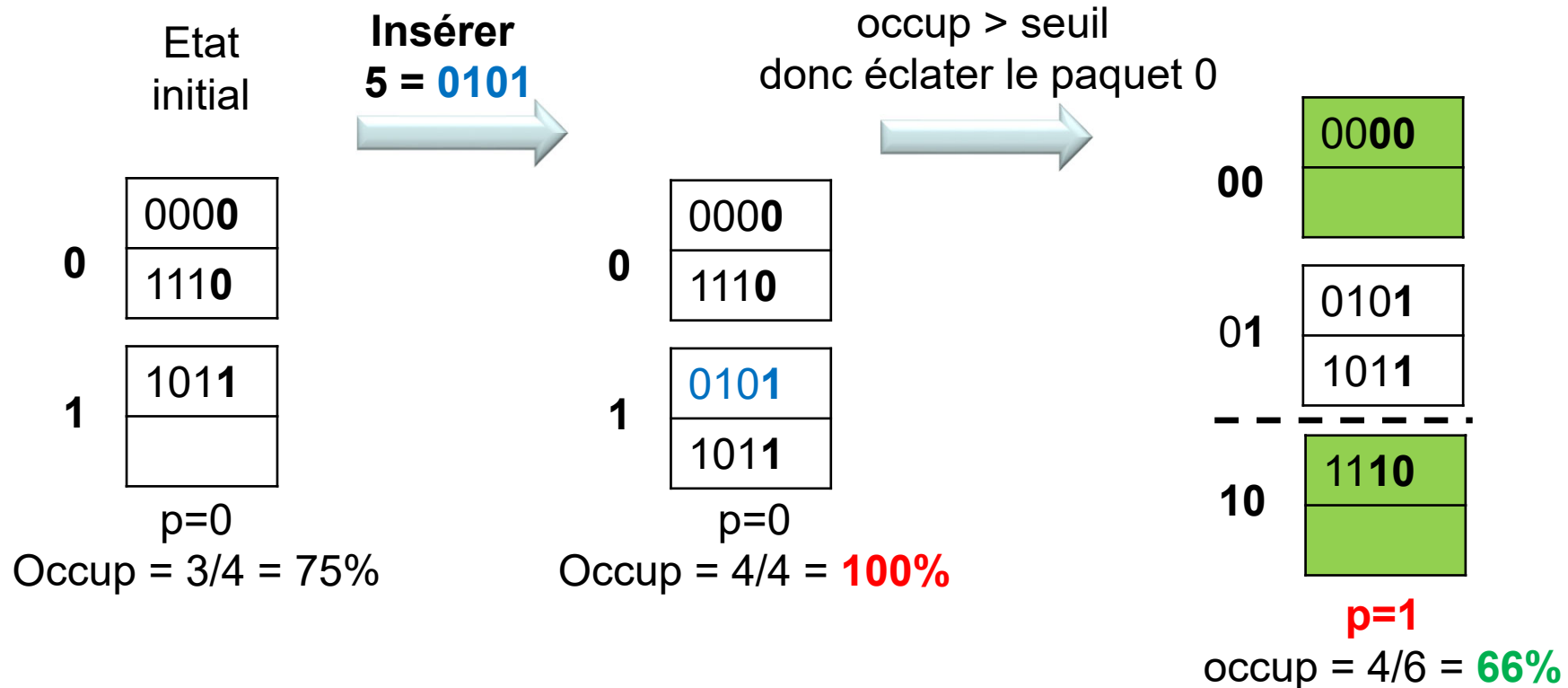
- Dans les 3 zones : principale + expansion + débordement

C : Nombre de **cases** seulement dans les paquets à accès direct

- Dans les 2 zones : principale + expansion
- ne **pas** compter les paquets dans la zone de débordement

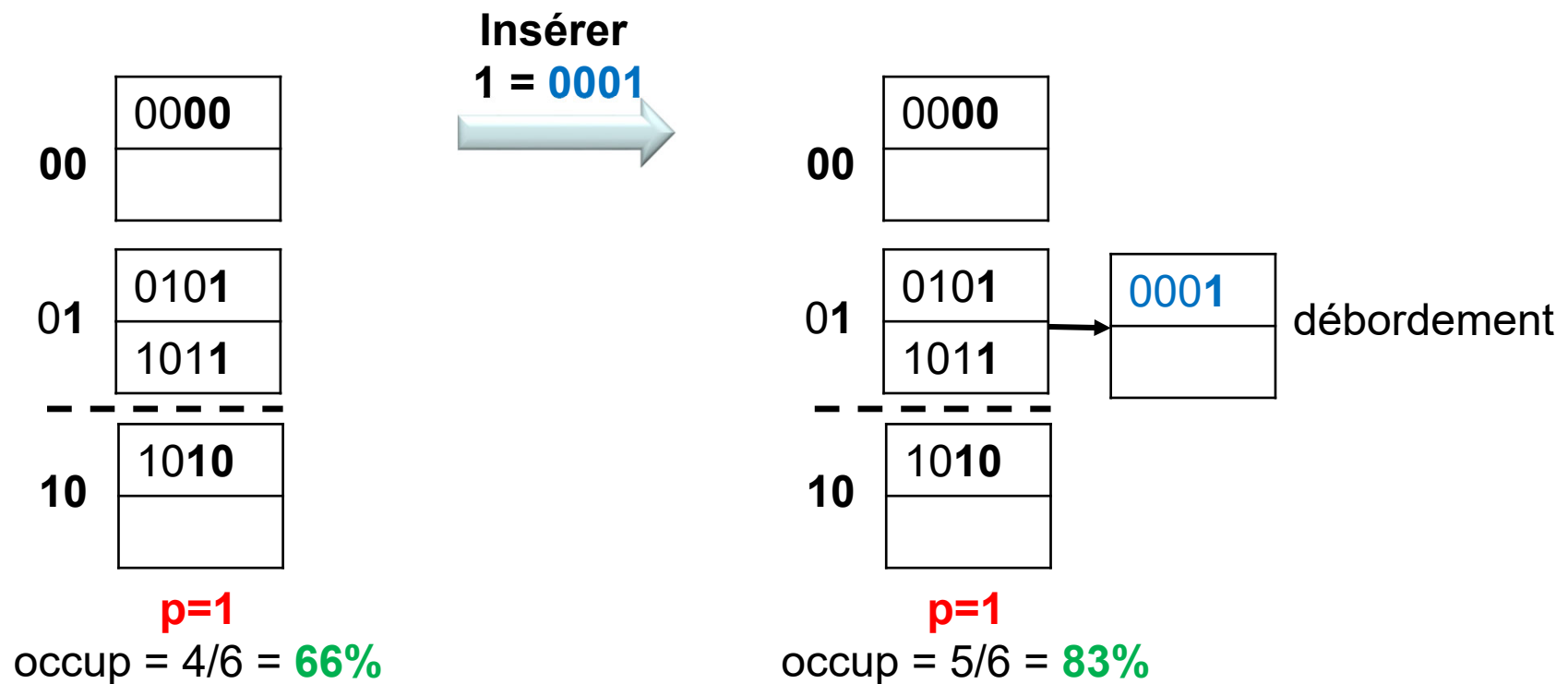
Hachage linéaire : insertion

- Exemple avec 2 paquets initiaux. $N=2$, seuil = 90%
- $h_0(x) = x \bmod 2$, $h_1(x) = x \bmod 4$
- $p=0$: le prochain paquet à éclater est le paquet 0



Insertion avec débordement

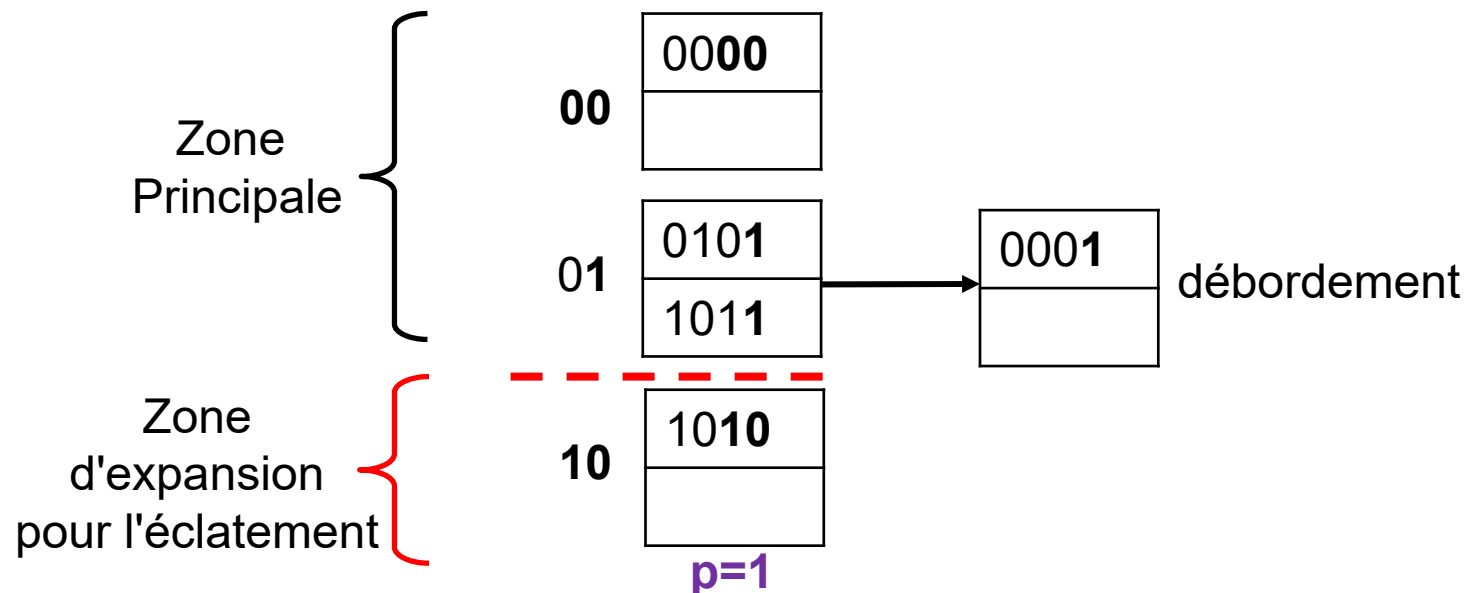
- Insérer 1 dans le paquet 1
- Débordement du paquet (pas d'éclatement car le taux d'occupation reste inférieur au seuil)



Hachage linéaire : accès

Méthode pour accéder à la valeur v :

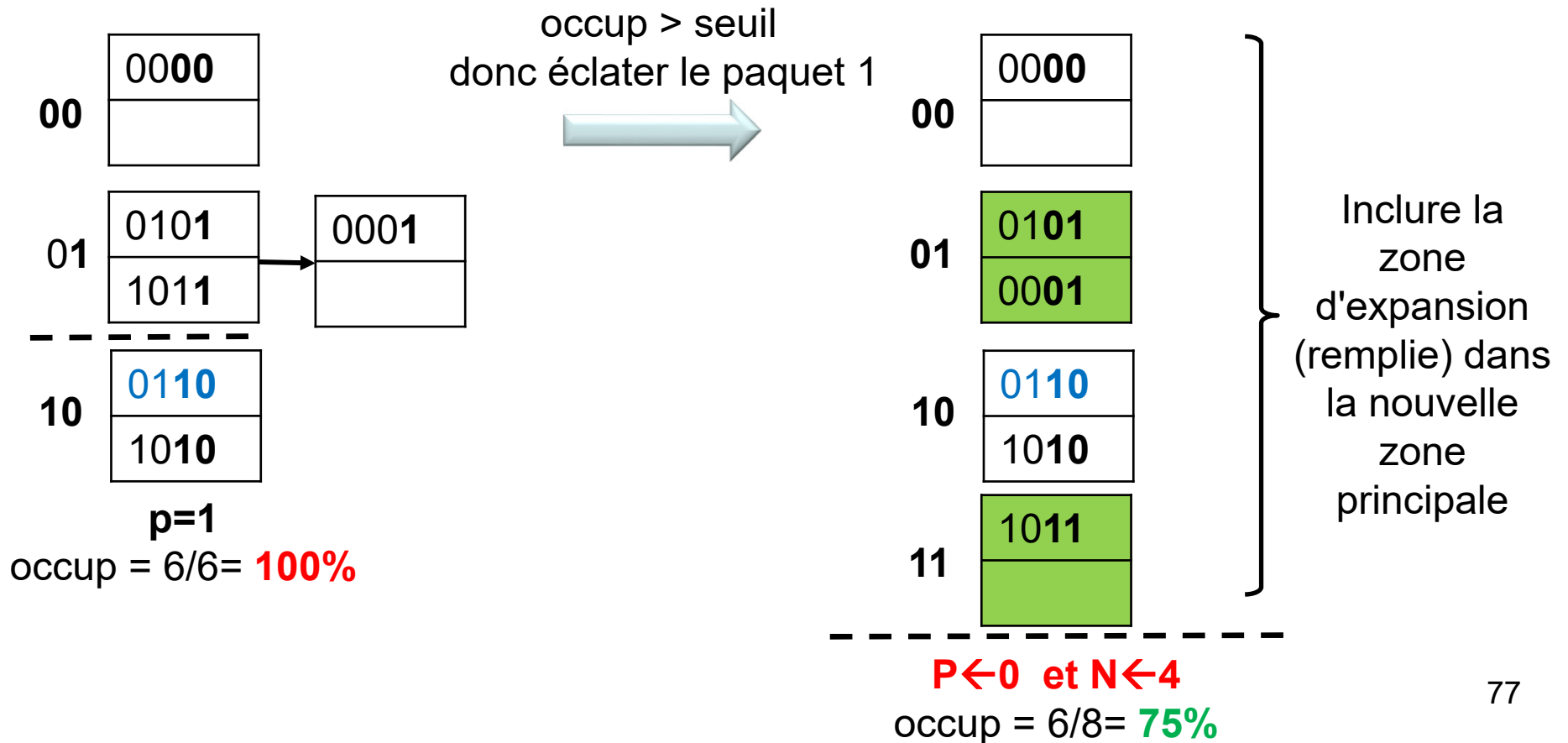
- Rappel: la zone principale a une taille de $N \cdot 2^i$
- Calculer $k = h_i(v)$ et $k' = h_{i+1}(v)$
- Si $k \geq p$ alors lire le paquet k sinon lire le paquet k'
- Exemple pour $N=2$, $i=0$, $p=1$:
 - Lire la valeur 13 (1011) dans le paquet $k = 13 \bmod 2 = 1$
 - Lire la valeur 10 (1010) dans le paquet $k' = 10 \bmod 4 = 2$ (car $k=0$ et $k < p$)



Insertion

et fin d'une série d'éclatements

- Insérer $6 = 0110$ dans le paquet $k' = 10 \bmod 4 = 2$ ($= 10_2$)
- Le dernier paquet de la zone principale éclate
- On redémarre à $p=0$ après avoir agrandi la zone principale



Exercice hachage linéaire

- Soit la table Département(nom, num)
- Construire un fichier dont le contenu est organisé par hachage linéaire. La capacité d'une page : 5 enregistrements.
- Les enregistrements sont:
 - Allier 1001 Indre 1000 Cher 1010 Paris 0101
 - Jura 0101 Ariège 1011 Tarn 0100 Aude 1101
 - Aveyron 1011 Doubs 0110 Savoie 1101 Meuse 1111
 - Cantal 1100 Marne 1100 Loire 0110 Landes 0100
 - Calvados 1100 Gard 1100 Vaucluse 0111 Ardeche 1001

Exercice hachage extensible

- Chaque paquet **contient au plus 2 valeurs**.
- **Question 1.** On considère un répertoire R de profondeur globale $PG=1$. Avec 2 paquets $P0$ et $P1$ $R=\{P0, P1\}$. Initialement les deux paquets contiennent:
 - $P0(4,8)$ $P1(1,3)$
 - Insérer la valeur 12.
 - Quelle est la profondeur globale après insertion ?
 - Détailler le contenu du répertoire et des paquets modifiés ou créés, et leur profondeur locale (PL).

Conclusion

Deux catégories principales de structures d'index

- Hachage
 - Utilisé pour les index plaçants : données stockées par paquet.
 - Très rapides pour une sélection par égalité
 - Ne supporte pas les sélections par intervalle
- Index B+
 - Utilisé pour les index plaçants : données stockées de manière triée
 - Utilisé pour les index non plaçants
 - Rapide pour une sélection par égalité ou intervalle

Perspectives

- Autres index arborescents :
 - quadtree (quadrants), k-d-tree (d dimensions), R-tree (region), cache conscious tree
- Autres structures : bitmap index
- Index pour les SGBD en mémoire
 - “ Generalized prefix tree source ”
 - <http://wwwdb.inf.tu-dresden.de/research-projects/projects/dexter/core-indexing-structure-and-techniques>
 - M. Boehm, Vainqueur Sigmod contest 2009
 - The Adaptive Radix Tree: ARTful Indexing for Main-Memory Databases
 - Viktor Leis, Alfons Kemper, Thomas Neumann
 - ICDE 2013
- Index réparti, décentralisé