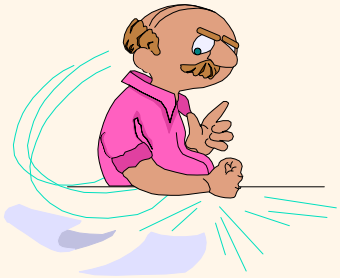
Bases de Données Avancées

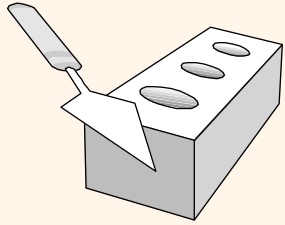
Ioana Ileana

Université Paris Descartes

Cours 6: Index basés sur du hachage (hash indexes)

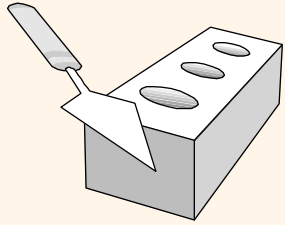


- Diapos traduites et adaptées du matériel fourni en complément du livre Database Management Systems 3ed, par Ramakrishnan et Gehrke ; un grand merci aux auteurs pour la réalisation et la disponibilité de ce matériel !
- Les diapos originales (en anglais) sont disponibles ici :
  http://pages.cs.wisc.edu/~dbbook/openAccess/thirdEdition/slides/slides3ed.html
- Plus particulièrement, ce cours touche aux éléments dans les Chapitre 11 du livre ci-dessus; lecture conseillée! ;)



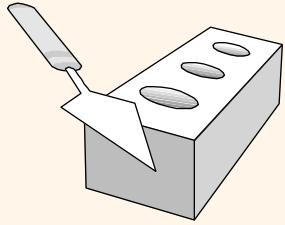
Les hash indexes

- Les meilleurs index pour les recherches / sélections par *egalité*
  - Ex moyenne=12
- **Ne peuvent pas être utilisés** pour les recherches / sélections par **intervalle** !
- Les grandes lignes :
  - Regroupes les pages contenant les entrées de données dans des alvéoles (buckets)
  - Utiliser une fonction de hachage pour convertir les valeurs de la clé de recherche en *indices d'alvéoles (bucket numbers)*
  - On trouve ainsi de manière efficace la bucket où se trouve une entrée



Les hash indexes

- *Comme pour tout index, 3 alternatives pour une entrée de données* **k*** (le choix entre les trois orthogonal à la technique d'indexation)
  - Record avec la clé = **k**
  - <**k**, rid du record avec clé = **k**>
  - <**k**, liste des rids des records avec clé = **k**>
- De nombreuses techniques d'indexation hash ; dans ce cours:
  - Hash index statique
  - Hash index extensible

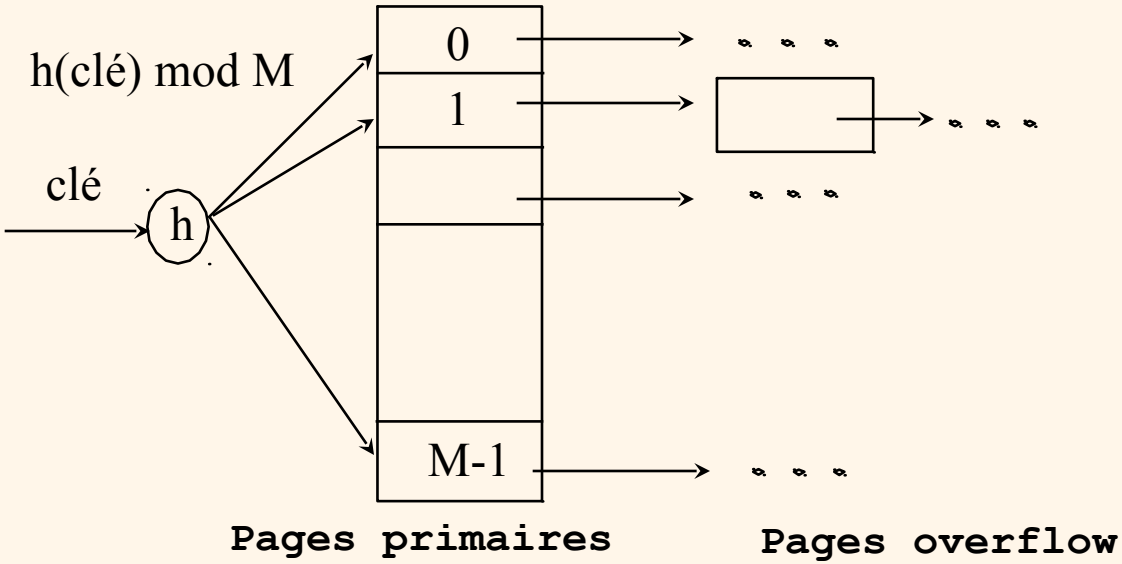


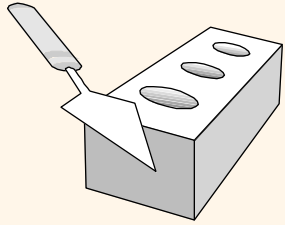
Hash index statique

- M alvéoles (buckets)
- A chaque bucket on associe une page primaire et potentiellement des pages overflow
- Les pages contiennent des entrées de données
- Fonction de hachage: **h**(*k*) mod M = donne l'indice de la bucket à laquelle appartient l'entrée de données qui a la clé *k*.

Hash index statique

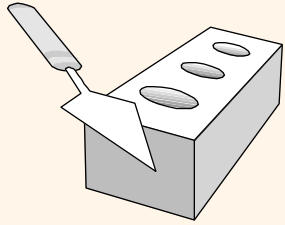
h(*k*) mod M = donne la bucket à laquelle appartient l'entrée de données qui a la clé *k*





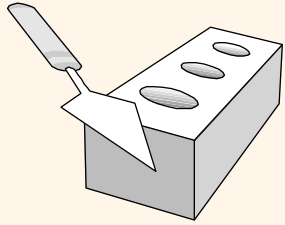
Hash index statique

- Insertion / recherche / suppression : chercher la bucket où se trouve (ou devrait se trouver) l'entrée, en utilisant la fonction de hachage
- Comme dans ISAM, les pages primaires sont « immuables » : en nombre fixe, allouées de manière séquentielle à la construction, ne sont jamais effacées
- De nouvelles pages overflow sont créées quand « il n'y a plus de place » sur les pages d'une bucket
- Les pages overflow sont effacées si elles deviennent vides !
- Combien d'accès disque pour une recherche si pas de pages overflow ?



Hash index statique

- La fonction de hachage doit permettre de “bien distribuer les valeurs de clés” dans les buckets
  - **h**(*clé*) = (a * *clé* + b) marche souvent très bien (avec tout un tas d’études sur le choix des constantes a et b!)
- Malgré un choix attentif de la fonction, les hash index statiques souffrent justement (comme ISAM) de leur caractère statique :
  - Problème posé par un nombre important d’insertions / suppressions après la construction
  - De longues “chaînes de pages overflow” peuvent apparaître et tout comme pour ISAM pénaliser fortement les performances!
- *Les hash index extensibles*: technique *dynamique* pour corriger ce problème!

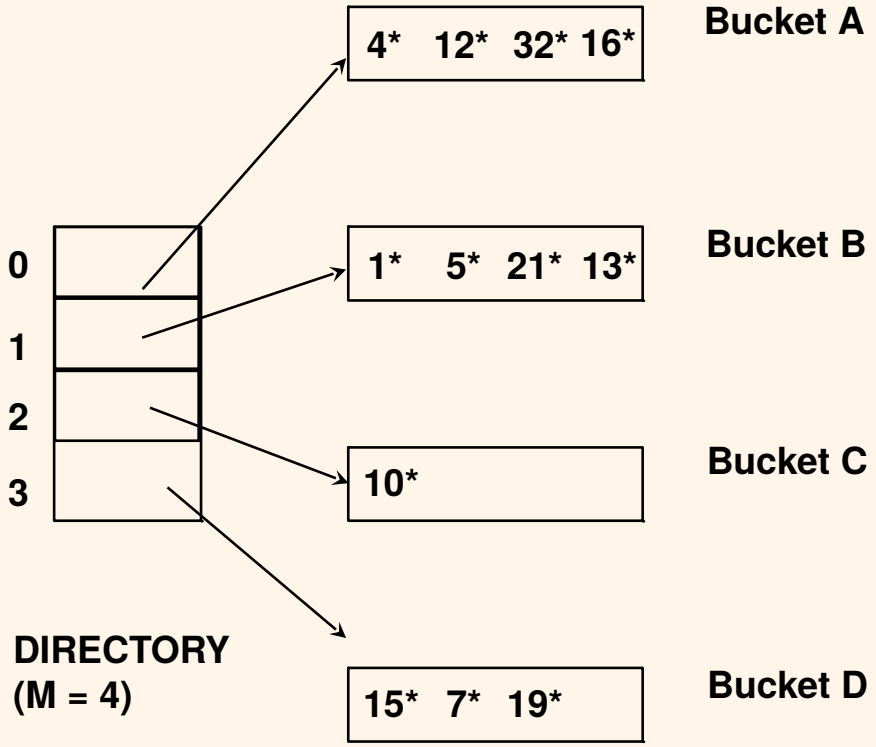


Hash index extensible: idée

- Situation: La page primaire d'une bucket est pleine et on doit faire une insertion ; on voudrait éviter les pages overflow !
- Idée: diviser (split) la bucket
- Problèmes : Comment éviter, suite à l'ajustement du nombre de buckets, de devoir déplacer les entrées partout (le modulo change!), ainsi que potentiellement les pages ? Comment redistribuer les entrées de manière cohérente lors d'un split ?

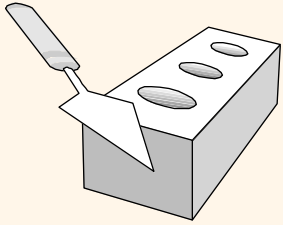
Hash index extensible: pointer directory

- Problème : comment éviter de déplacer les pages ?
- Solution : accès à toutes les pages via des pointeurs qui sont stockés dans un « pointer directory »
  - Similaire au page directory qu'on avait vu pour les heap files !
  - Le directory est lui-même sur une page et nous permet d'accéder aux buckets via les pointeurs !



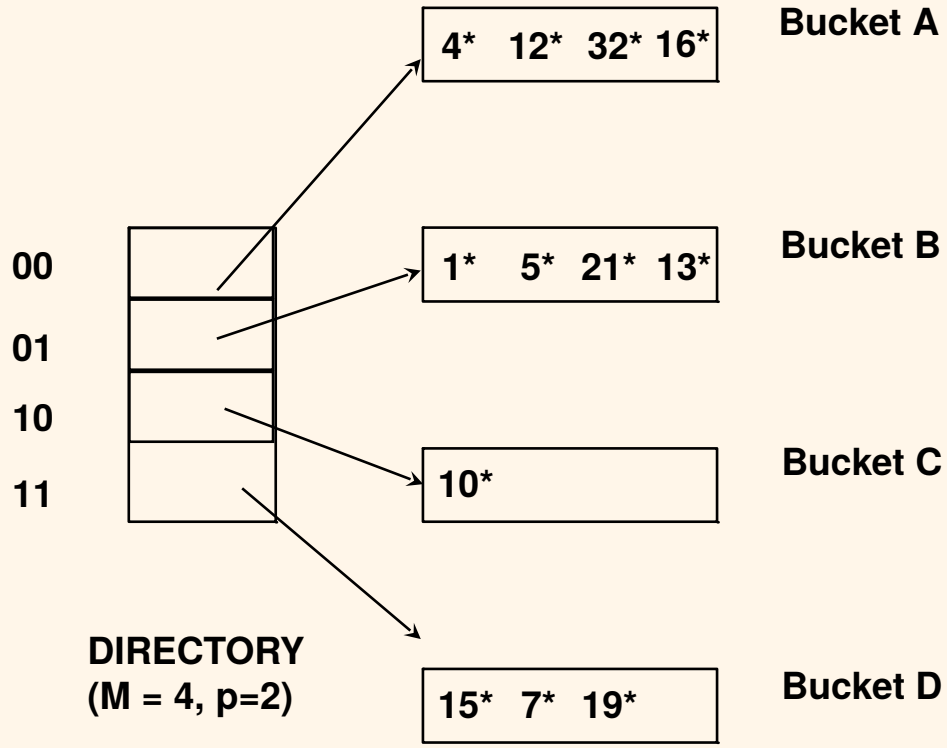
•Dans cette figure et les suivantes nous « simplifions » en mettant, dans les buckets, la valeurs h(k) à la place de la valeur k

Hash index extensible: puissances de 2 et prise en compte des derniers bits



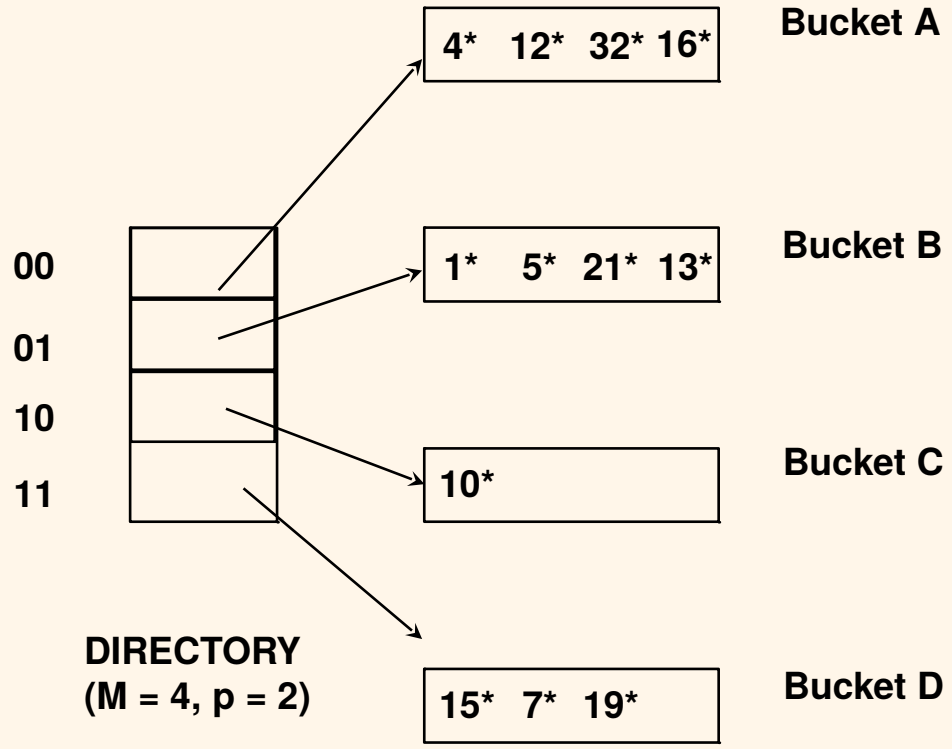
- Problème : comment faire un split sans affecter toutes les entrées ?
- Idée : utiliser 1 bit « discriminant » et faire en sorte que la valeur M qu'on utilise pour calculer le modulo (qui nous donne la bucket) **soit une puissance de deux** !
- **Autrement dit, notre choix de bucket sera le suivant :**
  - **Pour trouver la bucket de k*, on prend les derniers p bits de h(k)**
  - **Cela équivaut à dire qu'on prend h(k) mon M, pour M = 2^p !!!**
- Quand on fait un split, on va simplement incrémenter p (et ainsi on considère un bit de plus pour discriminer les deux buckets qui résultent du split) !

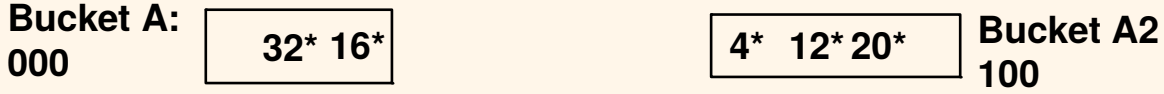
Hash index extensible: derniers bits et modulo



Hash index extensible: insertion

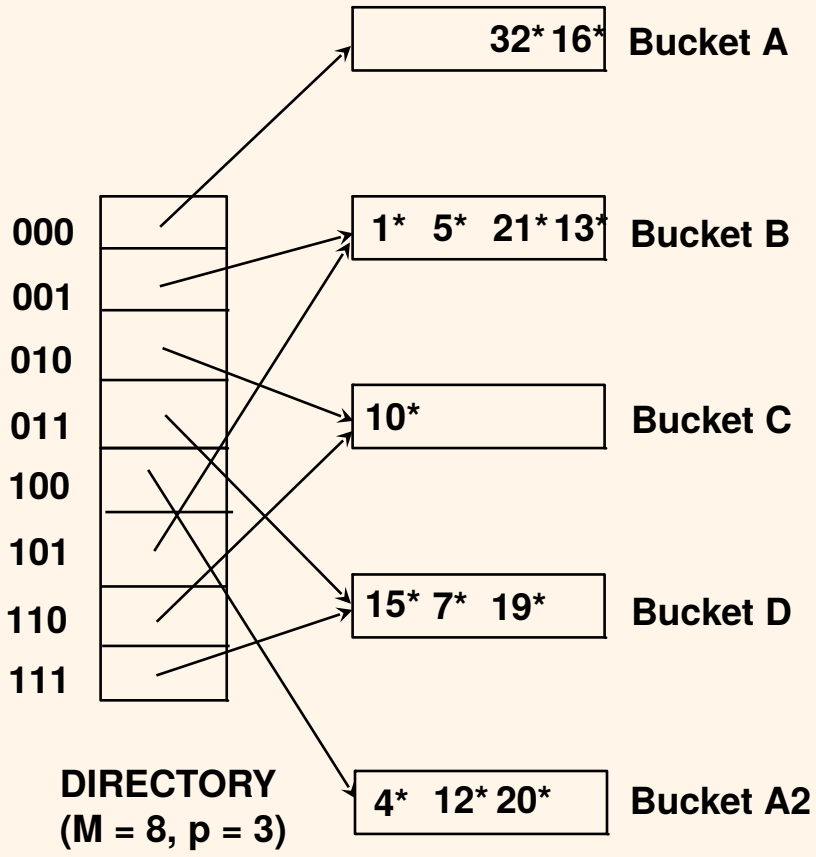
- Supposons qu'on doit insérer une entrée avec h(k) = 20
- Elle devrait aller dans la bucket A, qui est pleine !
- Nous devons splitter la bucket A : Nous prenons le 3ème bit depuis la fin comme bit discriminant pour le split :





Hash index extensible: besoin de doubler le directory

- Problème : le split a incrementé p (nb de bits utilisés) ; donc M = 8 et les entrées du directory *devraient comprendre trois bits*
- Pour les buckets non splittées, nous aurons donc deux entrées qui pointent vers la même bucket !!



Bucket A:
000

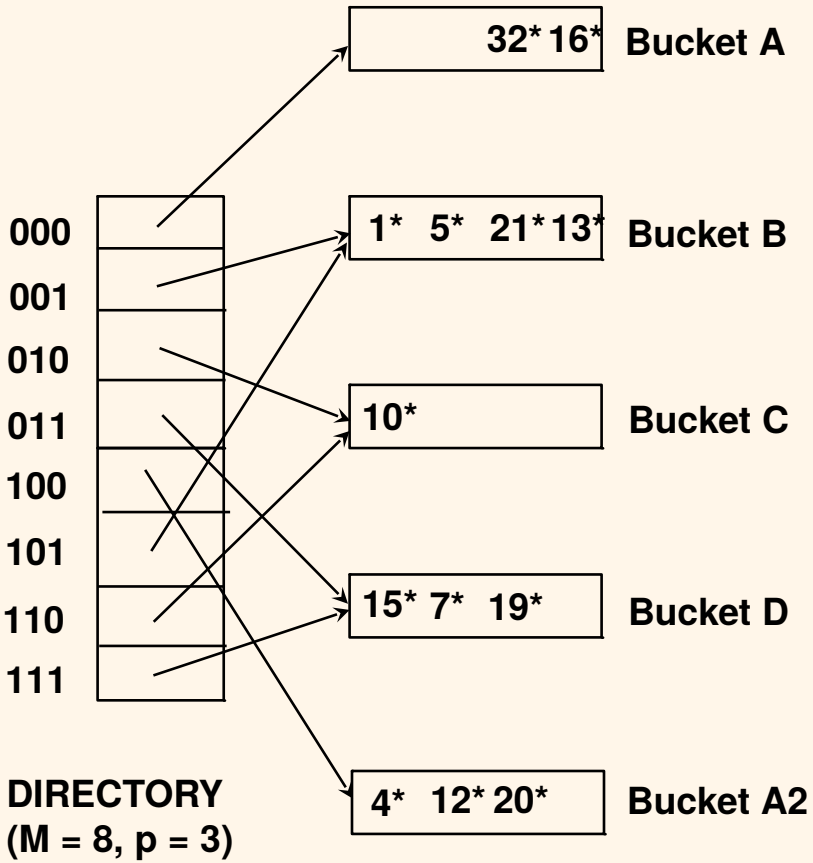
32* 16*

Bucket A2
100:

4* 12* 20*

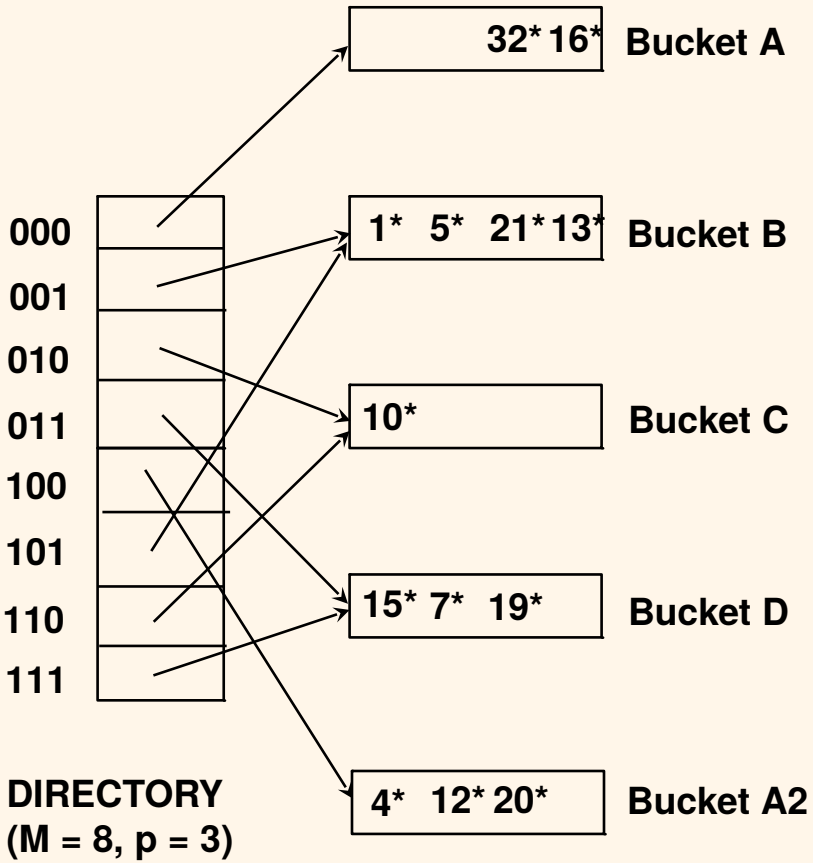
Hash index extensible: split sans doubler le directory

- Supposons maintenant qu'on veut insérer une entrée avec h(k) = 9
- Cette entrée irait normalement dans la bucket B. Comme la bucket B est pleine, on devrait faire un split
- A-t-on vraiment besoin dans ce cas de doubler le directory comme on a fait pour la bucket A ?



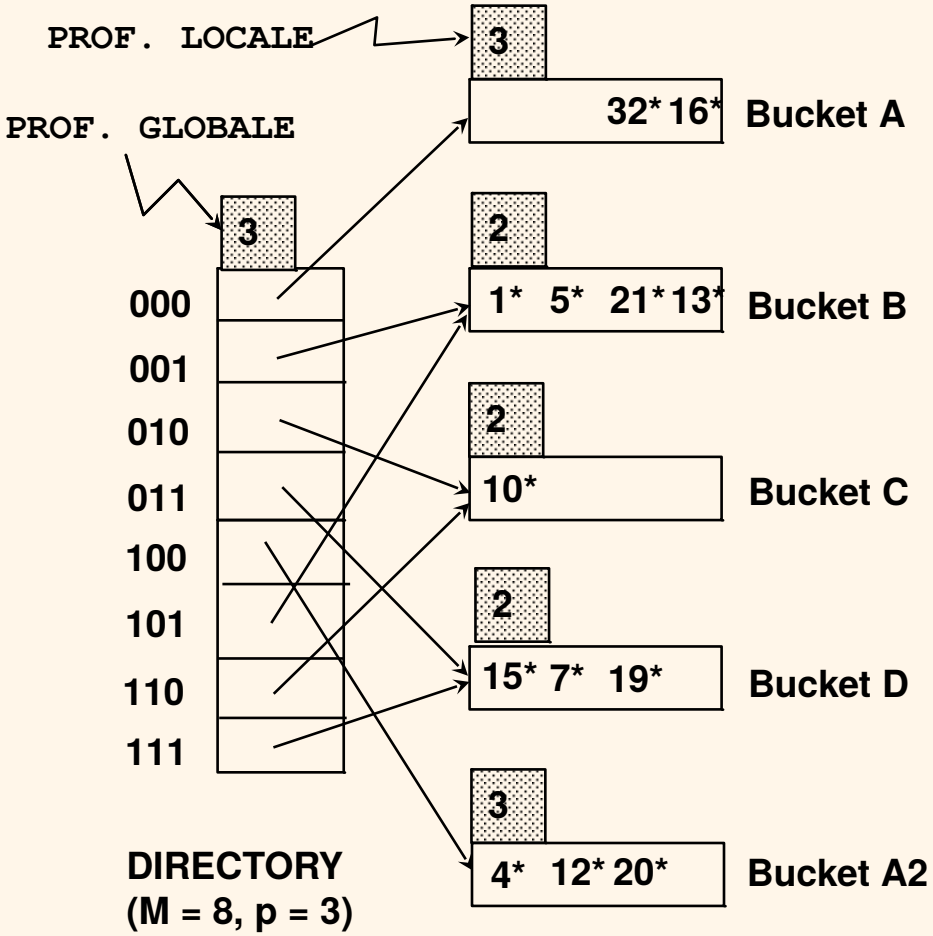
Hash index extensible: split sans doubler le directory

- Supposons qu'on veut insérer une entrée avec h(k) = 9
- Contrairement au cas précédent, nous n'avons pas besoin de doubler, car la bucket B « dispose déjà de deux entrées » ; il suffit de réutiliser une de ces entrées pour la nouvelle bucket résultant du split !
- Pour gérer ce type de cas : la notion de profondeur locale vs profondeur globale !

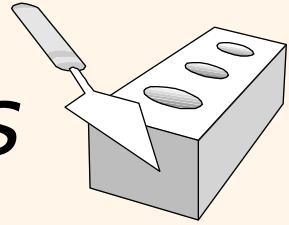


Hash index extensible: profondeur locale et globale

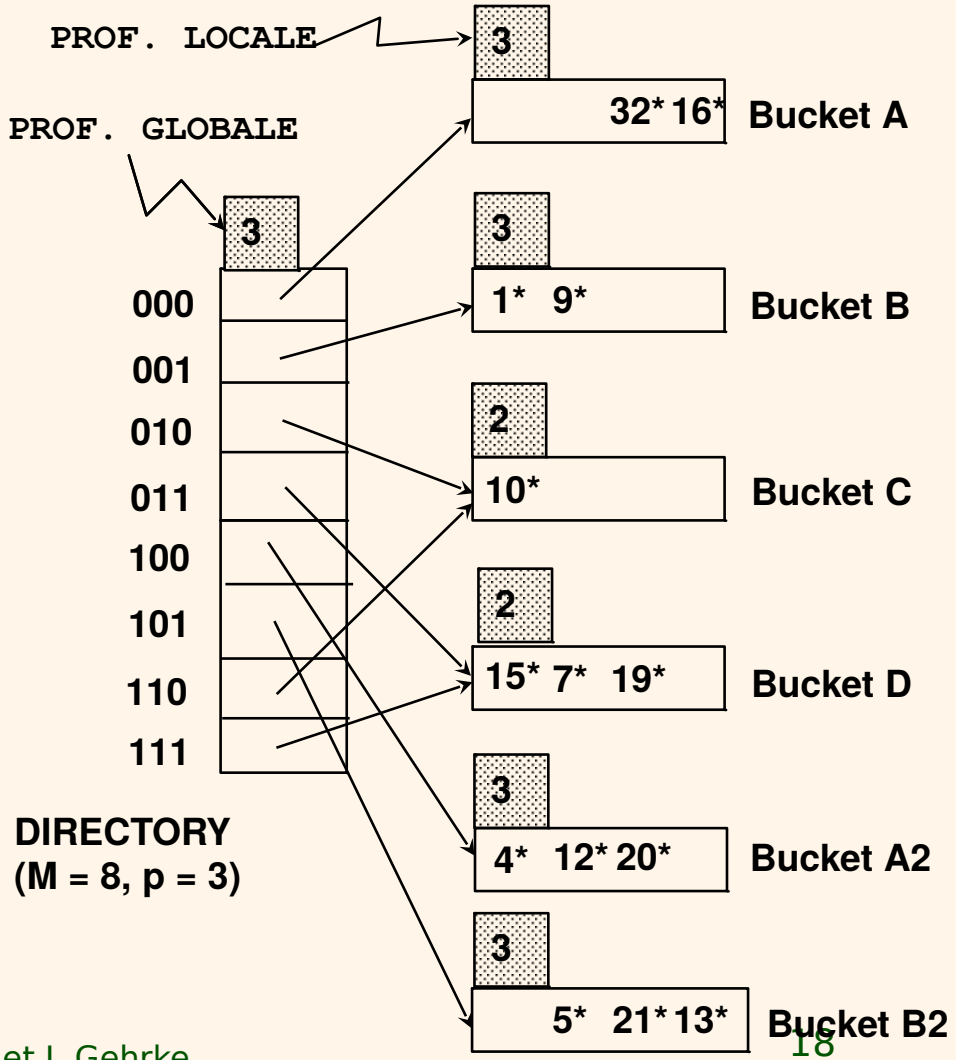
- Profondeur du directory (globale) = p
- Profondeur de chaque bucket (locale) : nombre de bits qui sont réellement analysés pour savoir qu'une entrée s'y trouve !
  - Inférieure ou égale à la profondeur globale
- Si la profondeur locale est inférieure à la profondeur locale, pas besoin de doubler le directory lors d'un split!!!



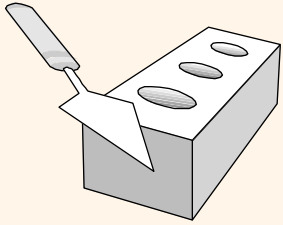
Hash index extensible: split sans doubler et mises à jour



- Insertion de h(k) = 9 → split de la bucket B en B et B2
- Nous ne doublons pas le directory ; nous « réutilisons les deux entrées de la bucket B »
- Et nous mettons à jours (incrémentons) la profondeur locale !

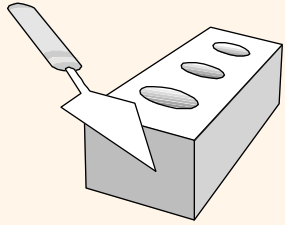


Hash index extensible: suppression



- Suppression:
  - Si une bucket devient vide, elle sera “unifiée” (fusionnée, merged) avec sa “soeur de split” (ex. 101 et 001) et la bucket résultante pointée par les deux entrées
  - La profondeur locale de la bucket résultante est décrémentée
  - Si à un moment donné toutes les buckets de notre index sont pointées par deux entrées ou plus, le directory peut être “réduit à moitié” (inverse du doublement)
    - On « enlève le bit le plus à gauche » ; on décrémente p

Hash index extensible: commentaires



- Un hash index extensible commence en général avec une seule bucket (M=1, p = 0) et se développe avec les splits et les doublements!
- Si le directory peut être gardé en mémoire, toute recherche coûte un seul accès disque
  - Pourquoi ?
- Le directory augmente par « à-coups » et si la distribution des valeurs de la fonction de hash n'est pas uniforme, le directory peut devenir de taille importante;
- Une agglomération d'entrées avec la même valeur de hash est problématique, car peut rendre les pages overflow inévitables !!
  - Pourquoi ? Que faire ?
- Malgré leur efficacité pour les recherches par égalité, les hash index sont peut utilisés dans les DBMS commerciaux