Master d'informatique- M1

UE 4IN803

SAM : Stockage et Accès aux Mégadonnées

Année 2020

site Web :

http://www-bd.lip6.fr/wiki/site/enseignement/master/sam/start

Hubert Naacke et Stéphane Gançarski

Contact : Hubert.Naacke@lip6.fr

Master d'Informatique : M1 parcours DAC

SAM 4IN803
Stockage et **A**ccès aux **M**égadonnées

Hubert Naacke et Stéphane Gançarski

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

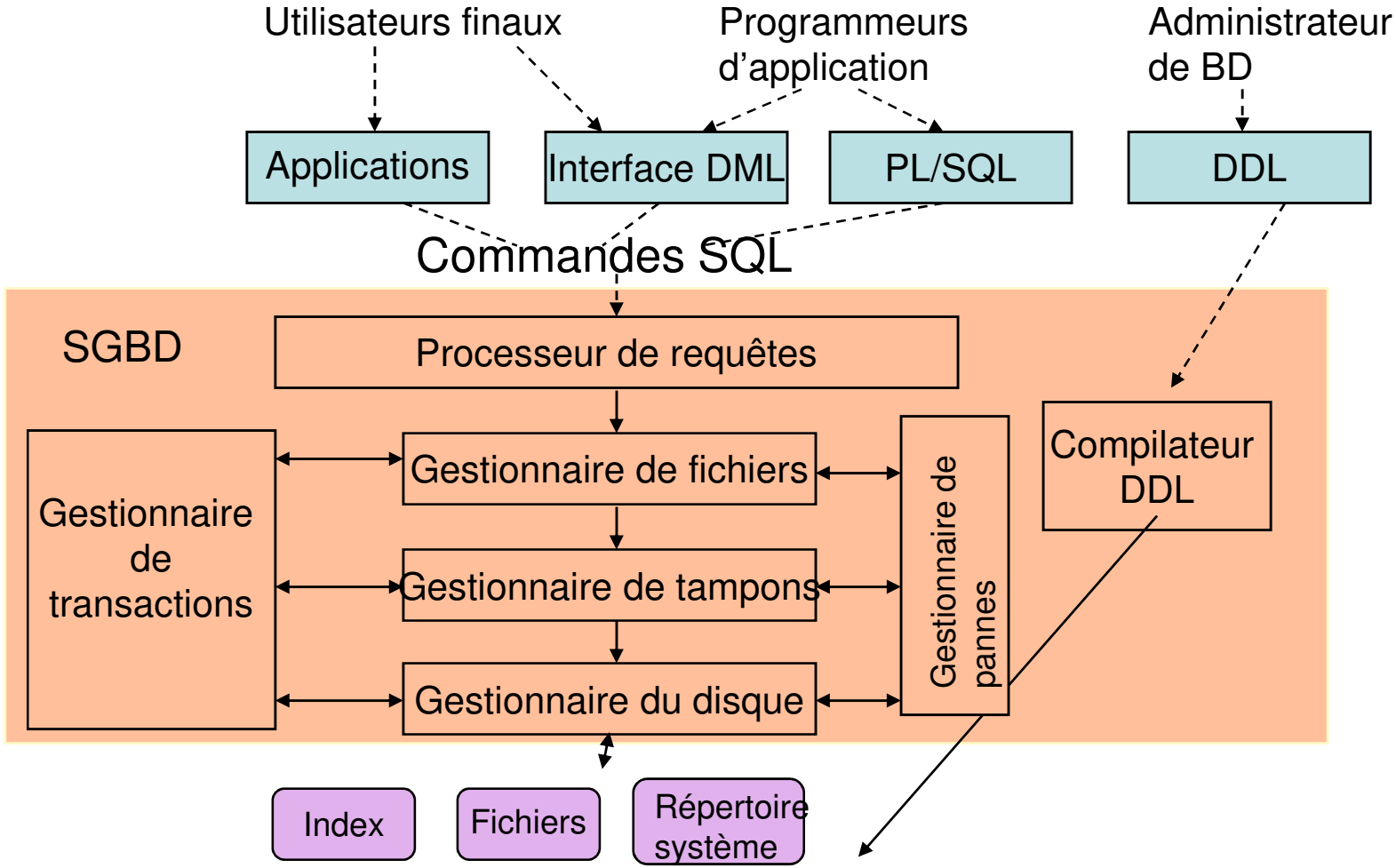
Plan

- 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 au 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 connait l'association thème→ étagère

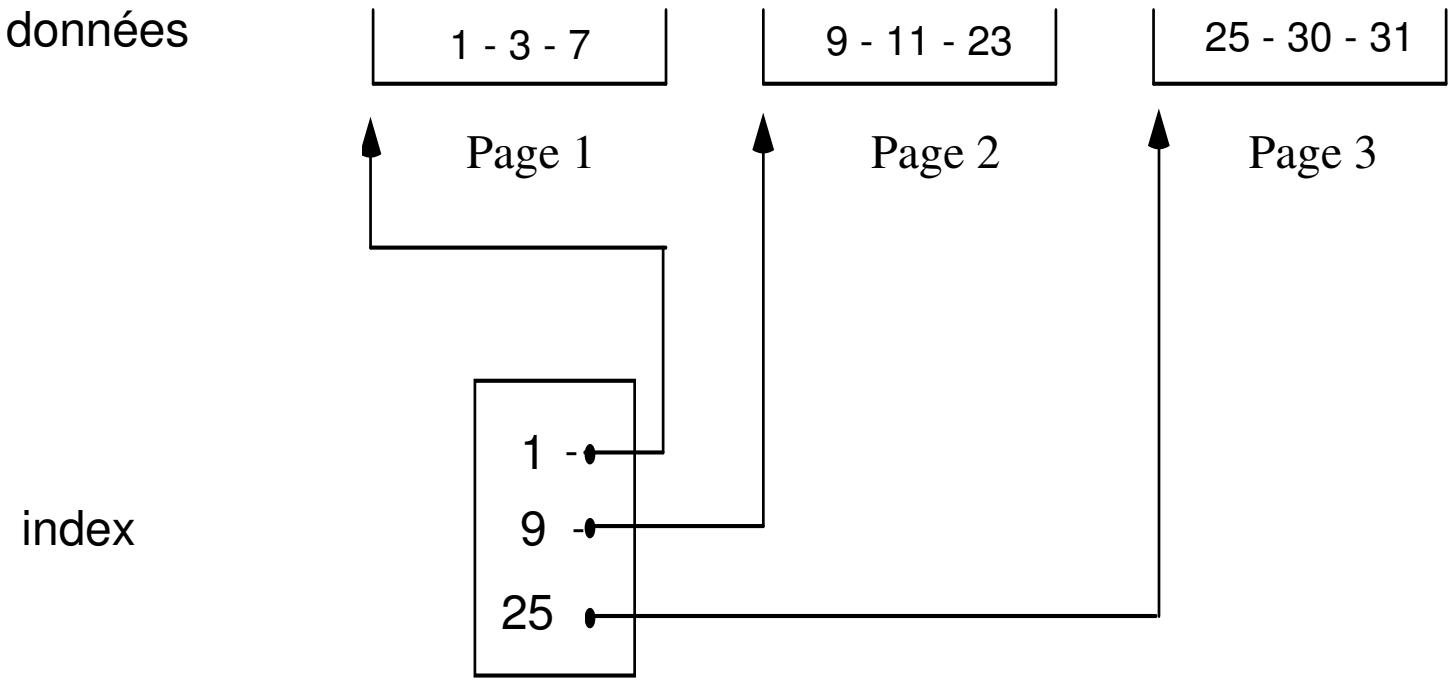
- Evidemment, pas plus d'un index plaçant par table
  - Appelé index principal

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** **Nom*Table*** **(*Attributs*);**
  - create index IndexAge on Personne(âge);

Index non plaçant (2/4)
Entrée d'un index

- Une entrée = structure associative
  clé → ROWID des enregistrements

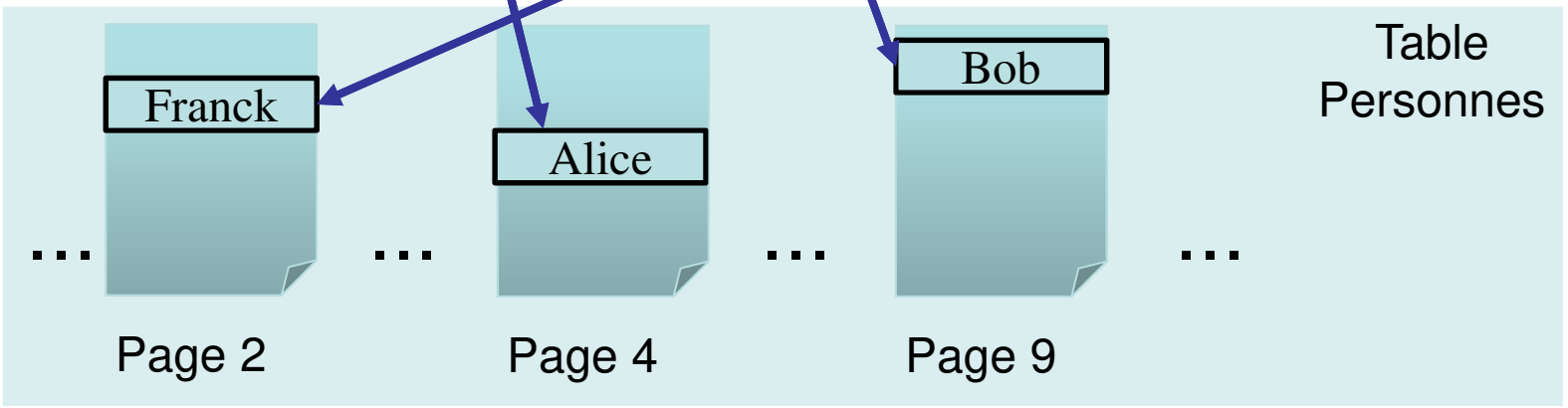
  Le ROWID permet de lire la page contenant l'enregistrement

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→ { 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}
  - Egalité, 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

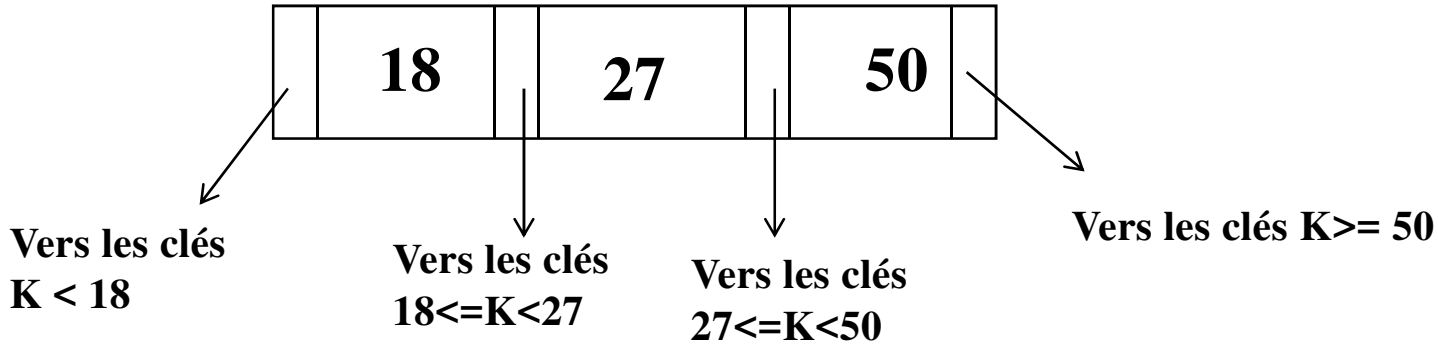
  → coût d'accès prévisible
- Avantage :
  - permet d'estimer le coût d'accès, a priori, pour décider d'utiliser ou non un index
- Mesure du coût:

  Nombre de nœud lus / écrits

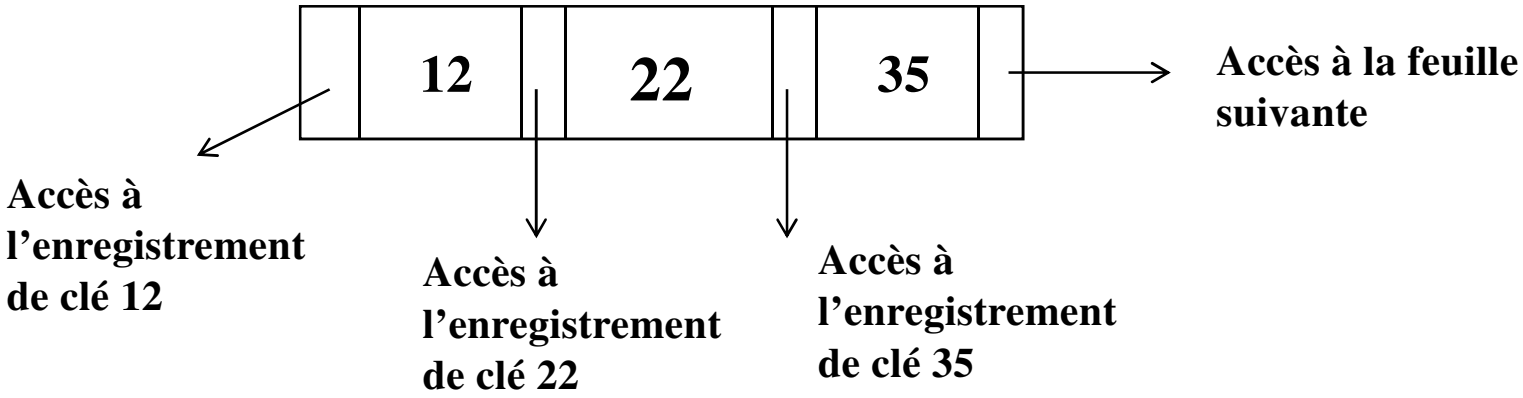
  = Nombre de pages de données lues / écrites

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: $\mathbf{1} \leq n \leq 2d$

- 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

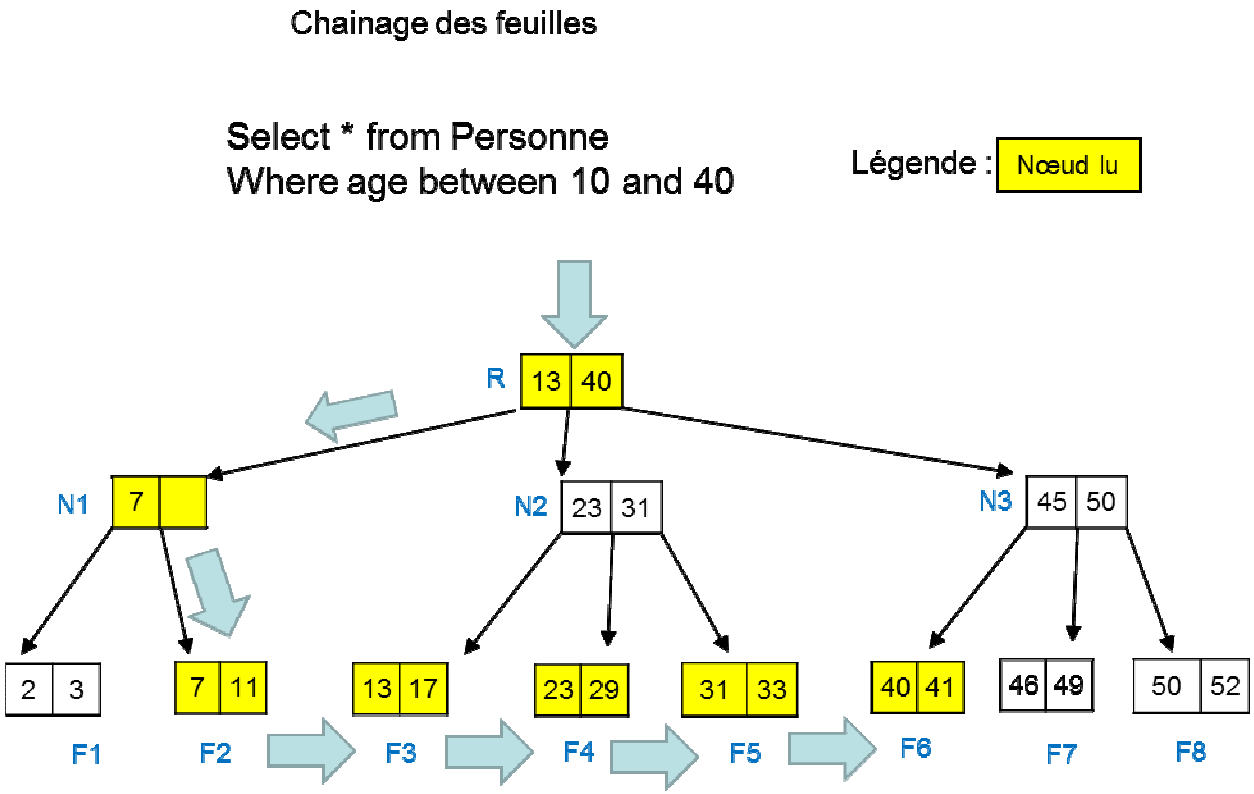
- Dépend de l'ordre d et du nombre de niveaux p
- Nombre maxi de clés dans l'arbre
- Arbre à 1 niveau (arbre réduit à sa seule racine): 2d clés maxi
- Arbre à 2 niveaux :
  - racine: 2d clés maxi
  - 2d+1 feuilles, soit 2d×(2d+1) clés maxi dans les feuilles
- Arbre à p niveaux :
  - Nbre maxi de clés 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)



Insertion
éclatement d'une feuille

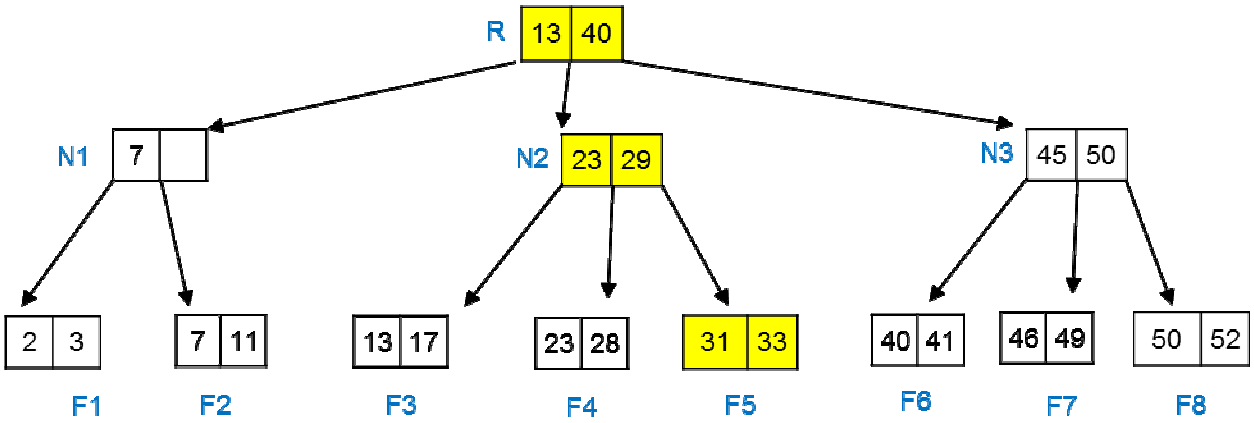
- Rechercher la feuille où insérer la nouvelle valeur.
- Insérer la valeur dans la feuille s'il y a de la place.
  - Maintenir les valeurs triées dans la feuille
- Si la feuille est pleine (2d valeurs), il y a éclatement. Il faut créer un nouveau nœud :
  - Insérer les d+1 premières valeurs dans le nœud 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 feuille sont au moins à moitié pleines

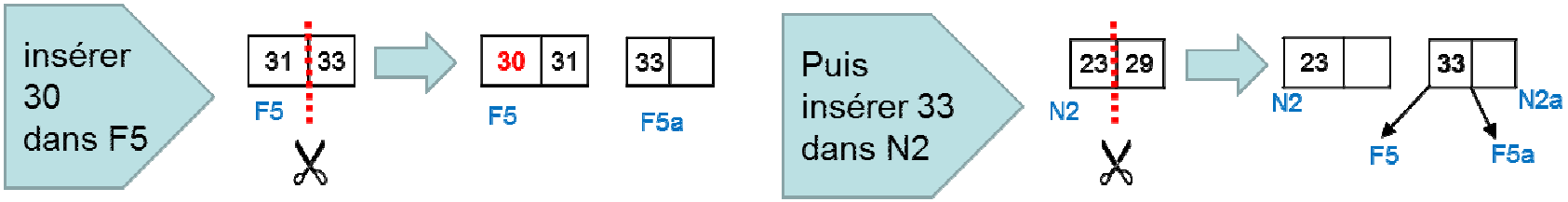
Insertion
éclatement d'un nœud intermédiaire

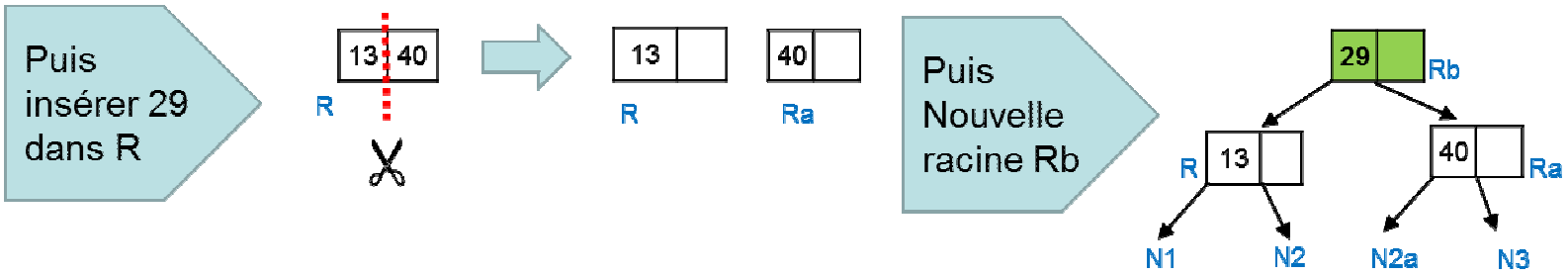
- S'il y a éclatement dans le parent, il faut créer un nouveau nœud frère M, à droite du premier
  - Les **d premières** valeurs restent dans le nœud N
  - Les **d dernières** vont dans le nouveau nœud M
  - La valeur restante est insérée dans le parent de N et M pour atteindre M
  - Résultat correct: 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 ?







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 ≤ v < 54 : 9, 16, 25, 36, 49
    - 5 valeurs nécessitent deux feuilles: (9, 16) et (25,36,49)
  - 54 ≤ v < 70 : (54, 61) peut tenir dans une seule feuille
  - 70 ≤ v < 88 : 70, 81, 84, 87
    - 4 valeurs nécessitent deux feuilles (70, 81) et (84,87)
  - v ≥ 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

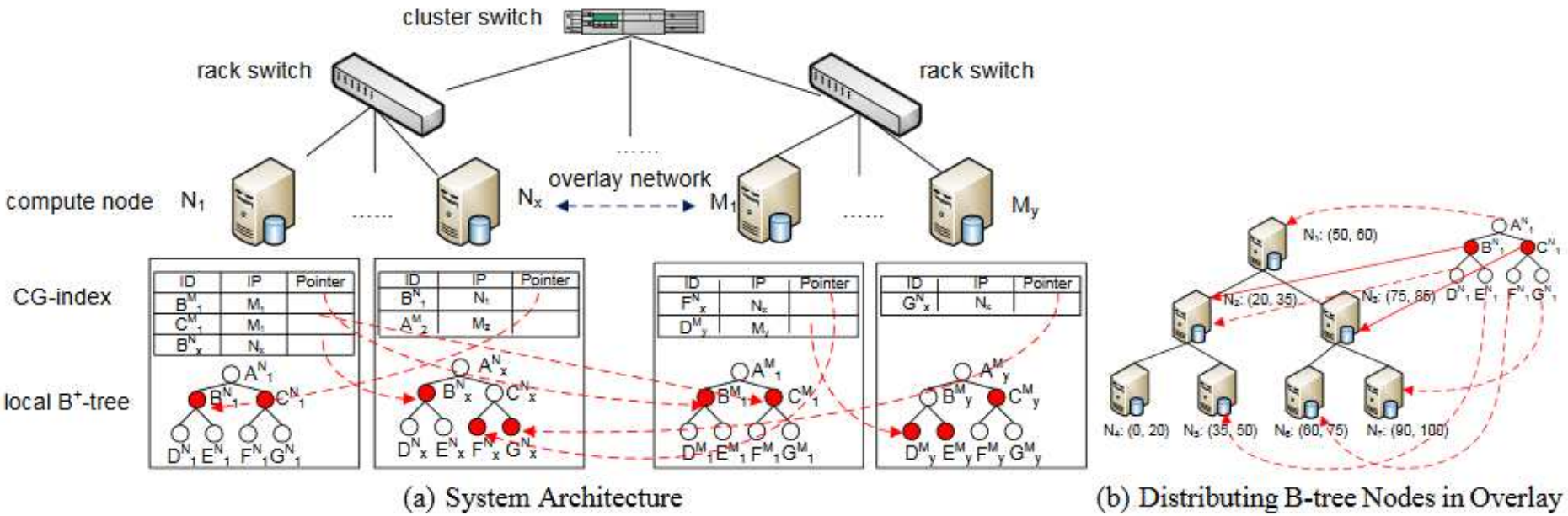
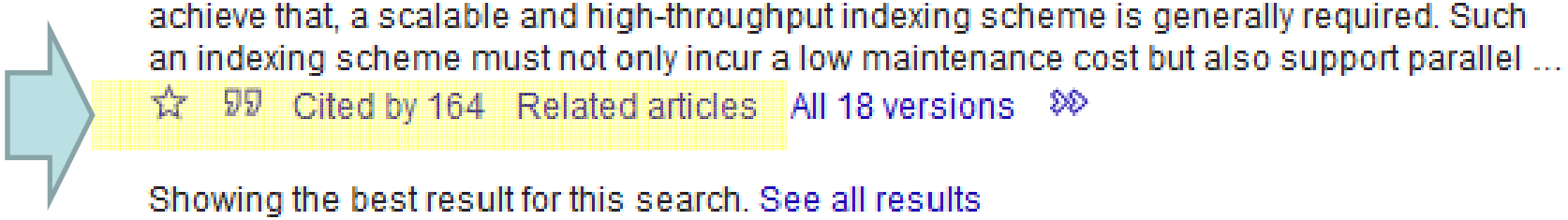


Figure 1: System Overview

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 ?



Index par

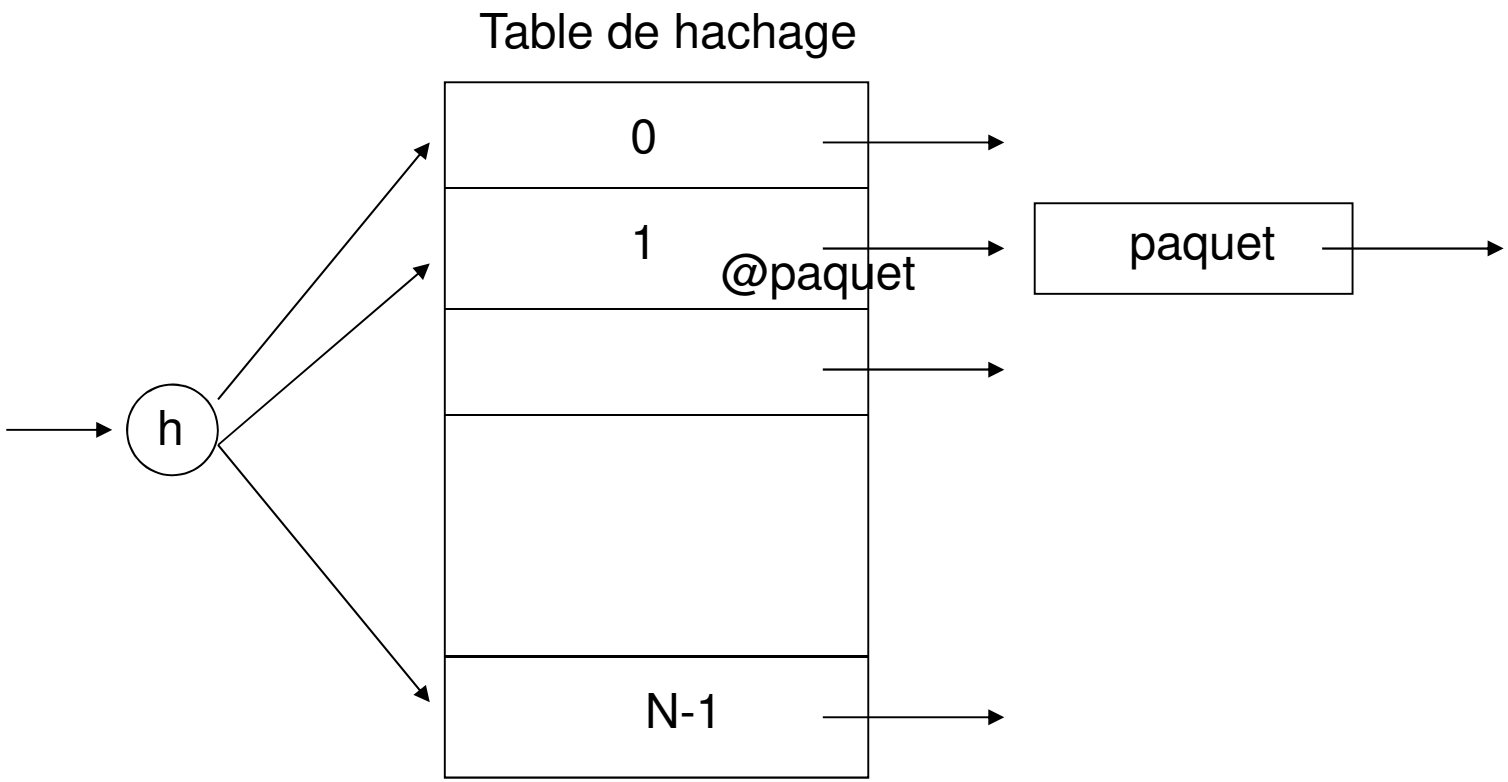
hachage

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

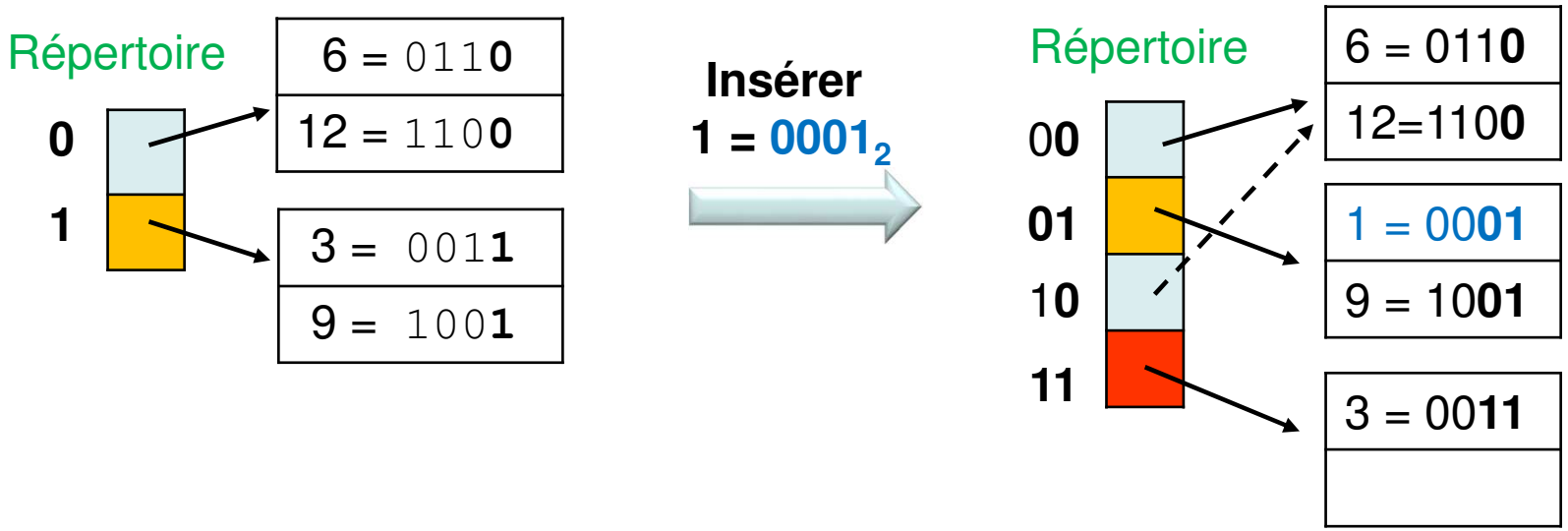
- Hachage dynamique :
  - techniques permettant de faire grandir progressivement un fichier haché saturé en distribuant les enregistrements dans de nouvelles régions allouées au fichier.

- Deux techniques principales
  - Hachage extensible
  - Hachage linéaire

Hachage extensible

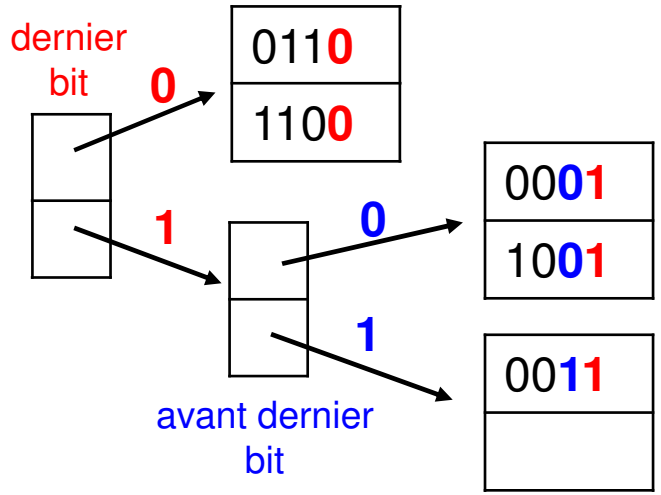
Hachage extensible

- Ajout d'un niveau d'indirection vers les paquets (tableau de pointeurs), qui peut grandir (considérer + de bits) : répertoire
- Jamais de débordement
  - Accès direct à tout paquet via le répertoire (i.e, une seule indirection)



Hachage extensible

- Répertoire similaire à un arbre à préfixe (*trie*)
  - Si on considère les bits en commençant par le dernier (i.e, celui de poids faible)



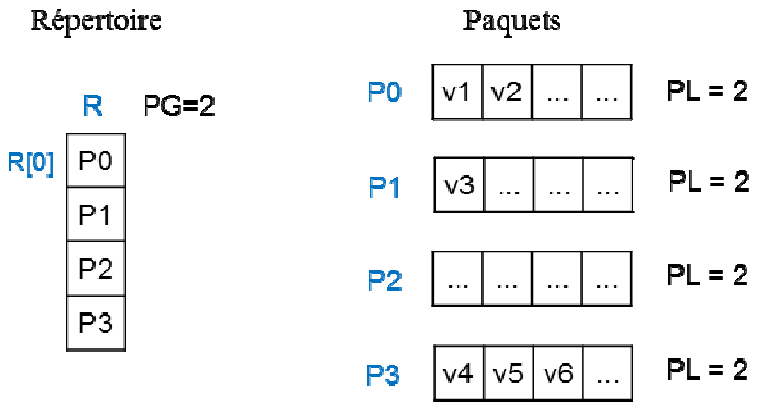
➲ Suffixe utilisé pour l'indexation = profondeur

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

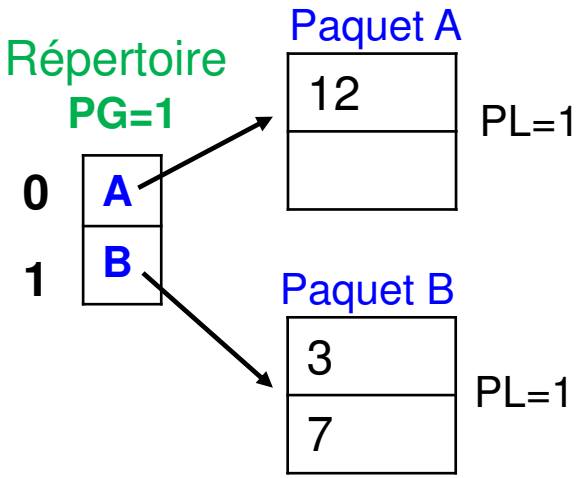
Hachage extensible : notations

- Le répertoire est noté **R[**P_0, P_1, P_2,..., P_k **] 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 $P_i(v_j, ..., ...)$ **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 1ère case)
- La valeur v se trouve le paquet référencé dans la case R[v 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

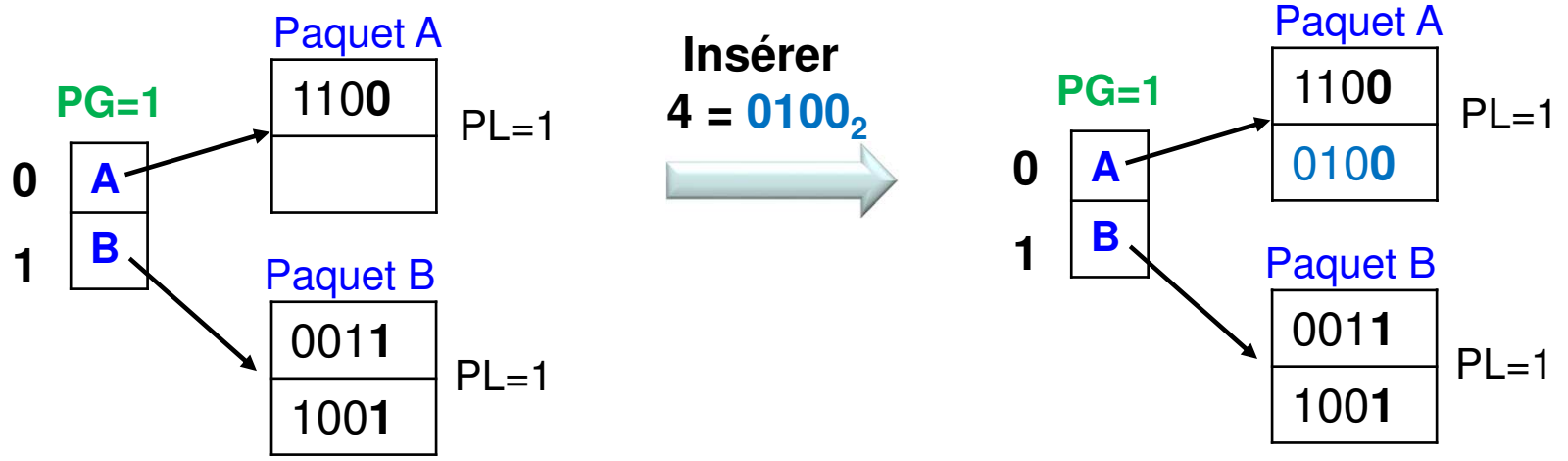


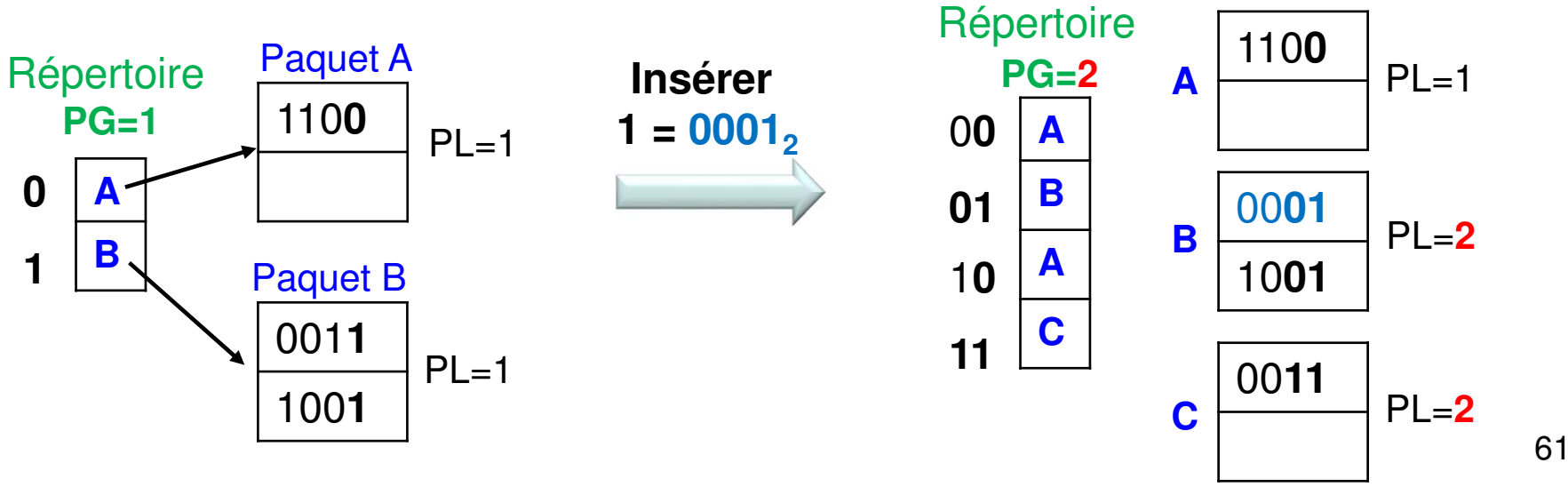
Hachage extensible : Insertion

- Insertion de v dans le paquet Pi

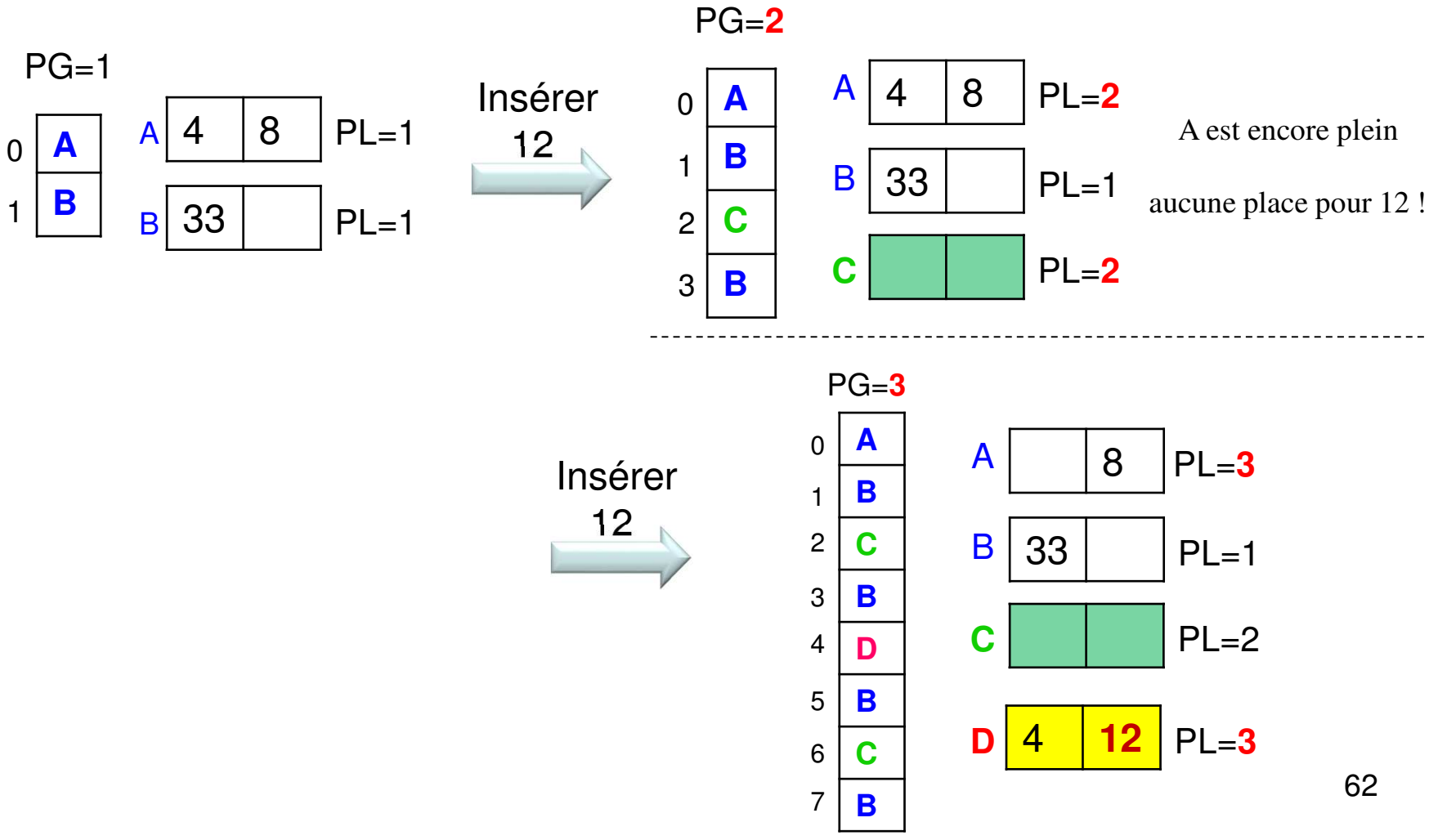
- **Cas 1)** P_i n'est **pas** plein, insertion immédiate dans P_i
- **Cas 2)** P_i est **plein** et **$PL_i < PG$** alors éclater P_i
  - Créer un nouveau paquet P_j et l'accrocher dans le répertoire
    - Remplacer l'adresse de Pi par celle de Pj dans la $2^{ème}$ case contenant Pi
    - Si 4 cases contiennent Pi, "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 Pi et v entre Pi et Pj
    - Si Pi est encore plein, réappliquer l'algo d'insertion : cas 2) ou 3)
- **Cas 3)** Pi est **plein** et **PLi = 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



62

Hachage extensible
Suppression et fusion

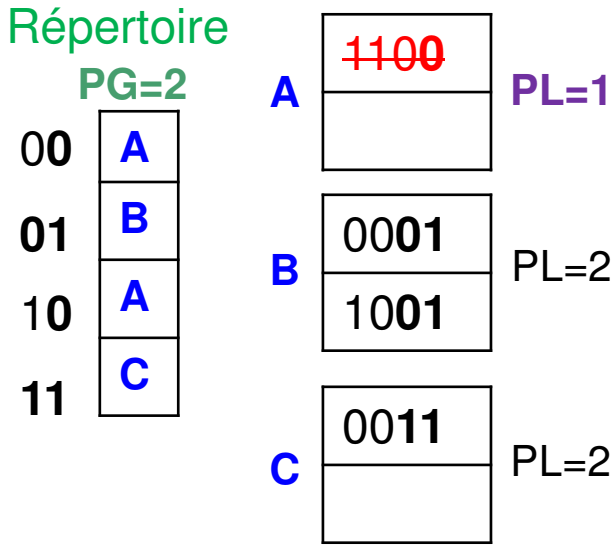
- Lors d'une suppression, si un paquet Pi devient vide :
- Si PL_i=PG alors
  - Fusionner Pi avec le paquet Pj référencé dans la case ayant le même suffixe que celle qui référence Pi
  - Suffixe commun (en base 2) de longueur PG -1
  - Exple si PLi=PG=3, les cases ayant le même suffixe (de longueur 2) sont :
    - R[0] et R[4]
    - R[1] et R[5]
    - ...
    - R[3] et R[7]
  - Supprimer Pi et le "décrocher" du répertoire : dans la case contenant Pi, remplacer Pi par Pj
  - Décrémenter la profondeur locale de Pj
- Sinon (on a PLi < PG) : pas de fusion, Pi 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

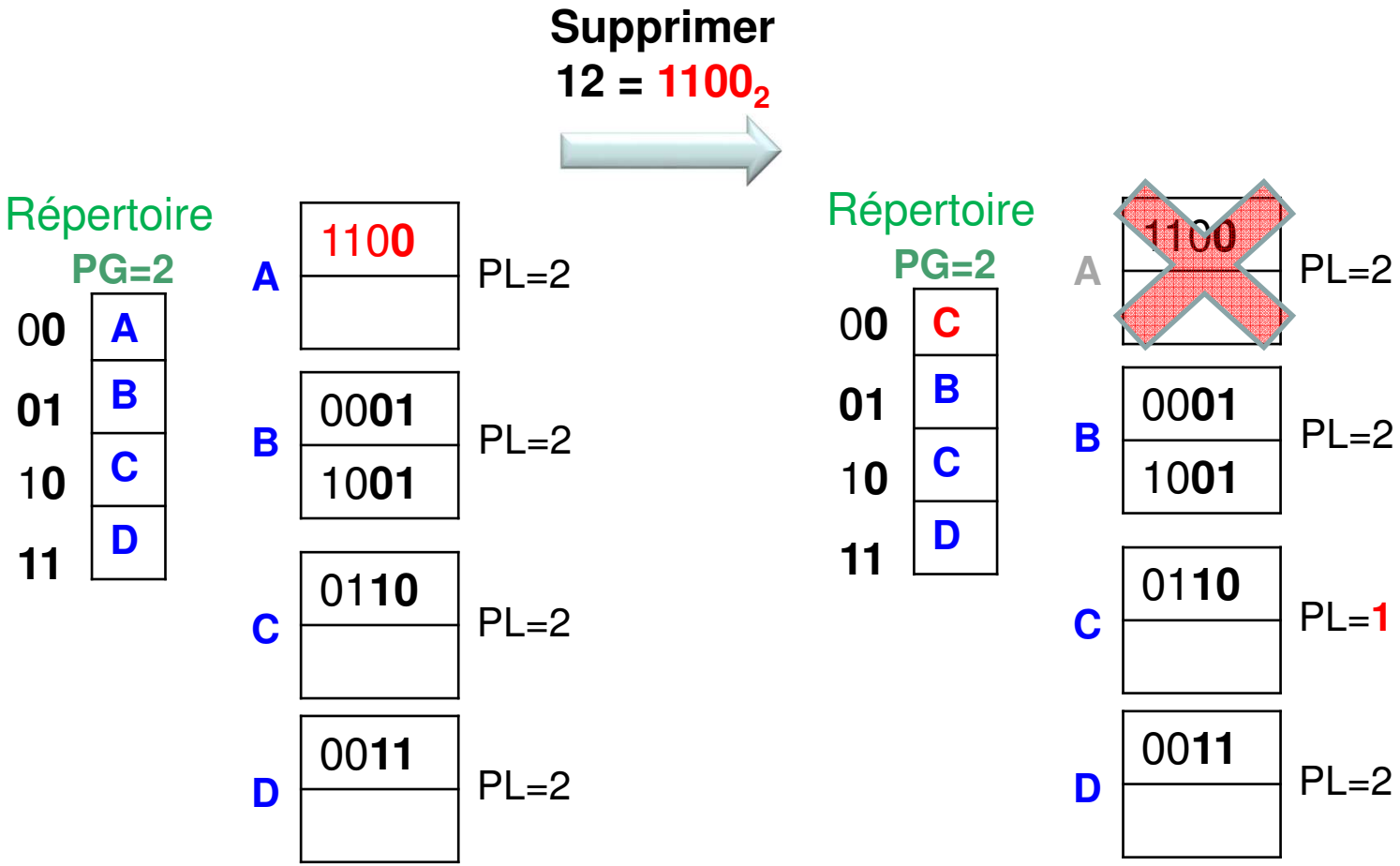
- Suppression dans un paquet ayant PL<PG

Supprimer
12 = 1100_2



Hachage extensible
Exemple de suppression

- Suppression dans un paquet ayant PL=PG



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

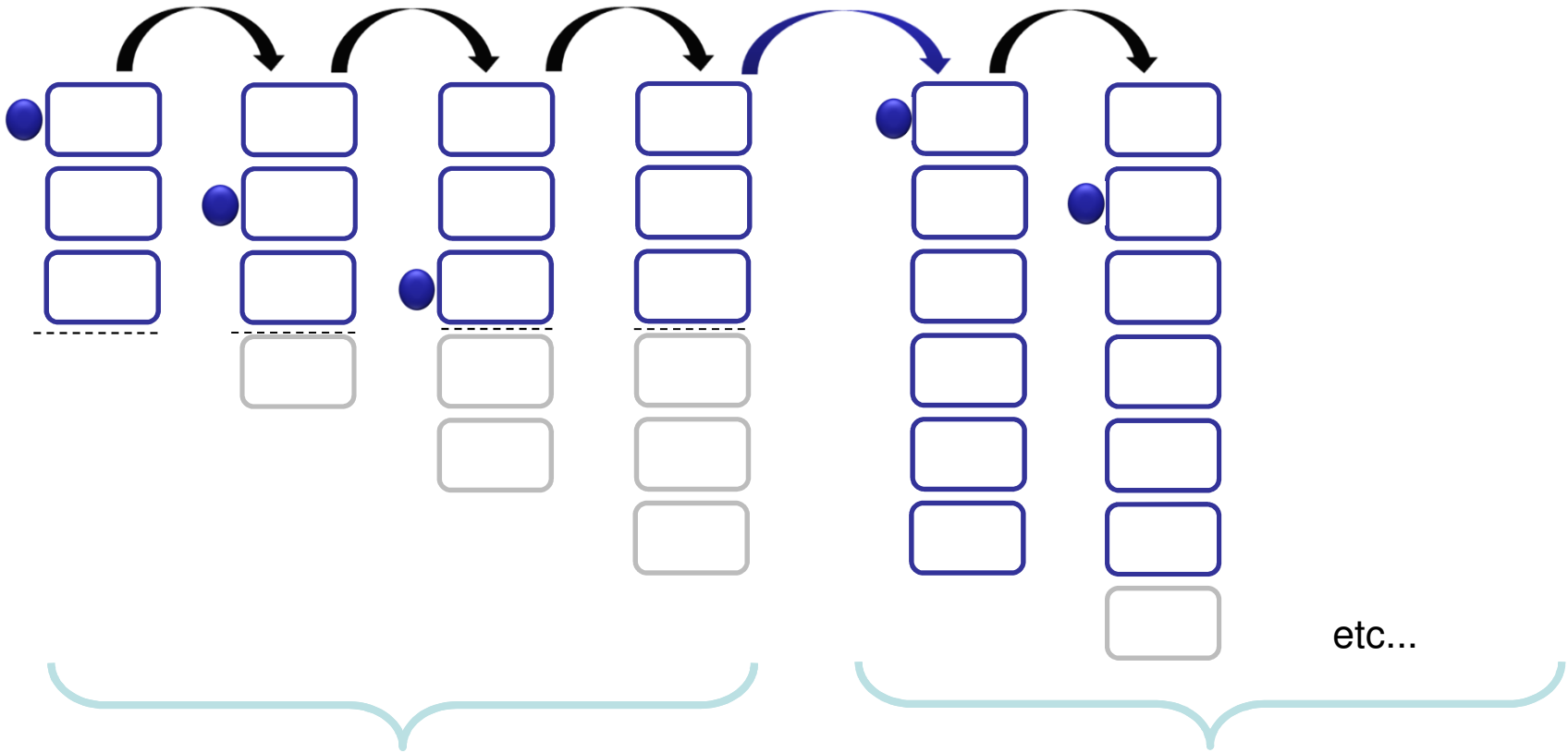
Hachage linéaire

- 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 $N^{ième}$ 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, ... :
  - $h_0(x) = x \bmod N$: fonction de hachage initiale
  - $h_i(x) = h(x) \bmod (2^i . N)$: $i^{ème}$ 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

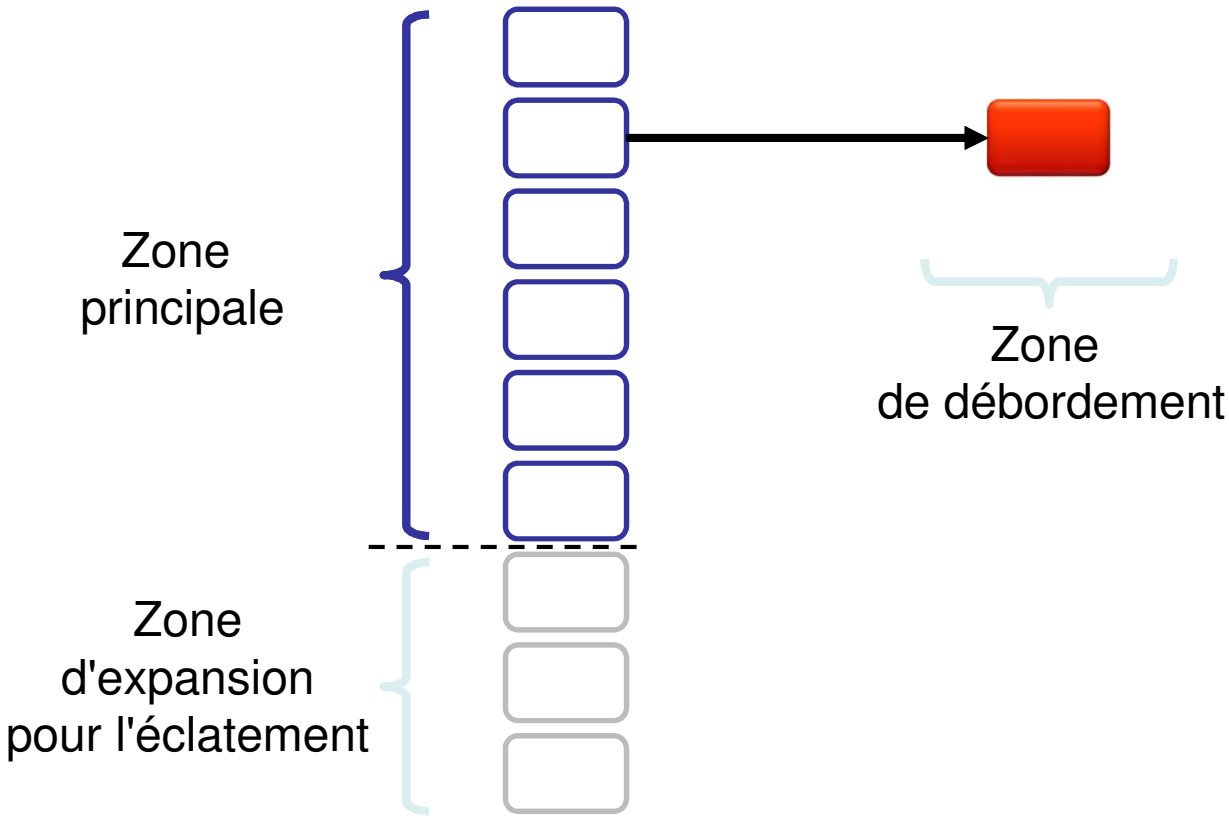


Première série d'éclatements

2ème série d'éclatements

p : prochain éclatement

Zones


Zone
principale
Zone
d'expansion
pour l'éclatement
Zone
de débordement

Hachage linéaire : taux d'occupation

- Taux d'occupation = V/C

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

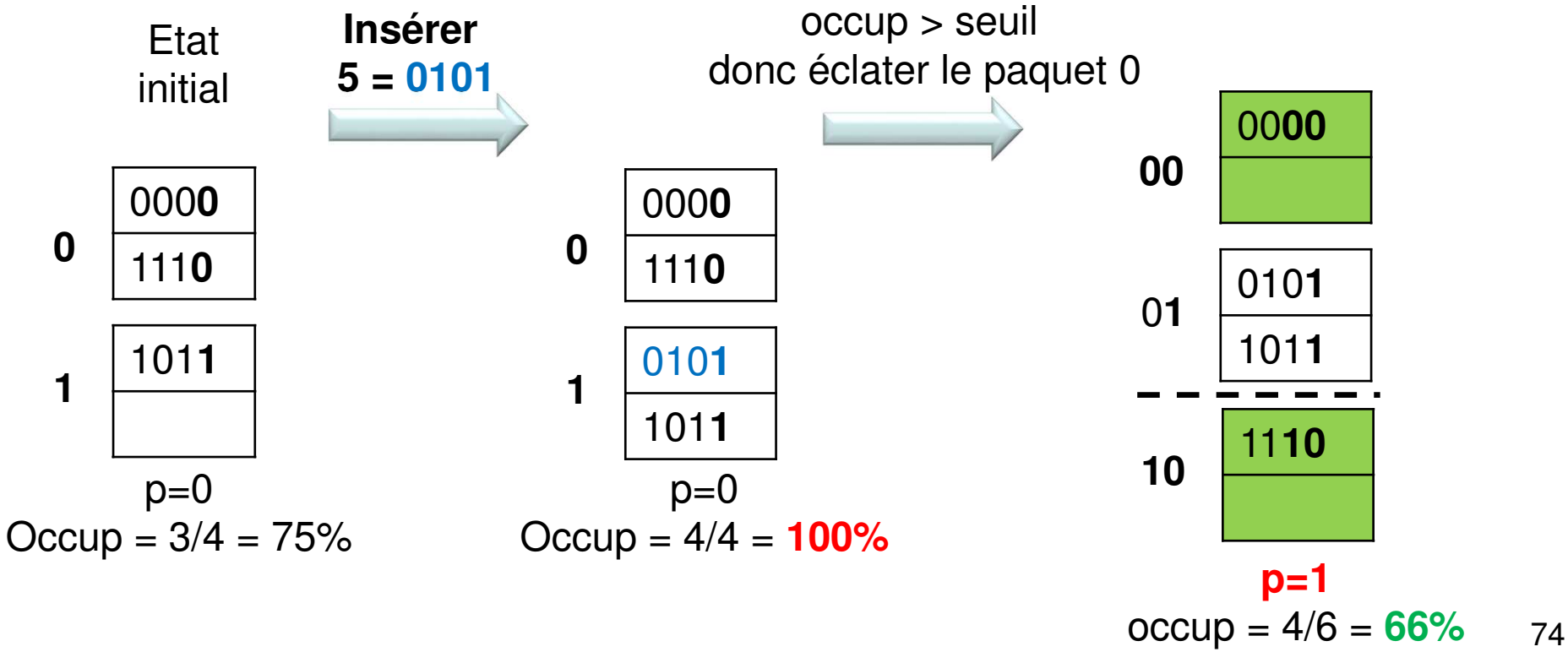
- Dans les 3 zones : principale + expansion + débordement

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

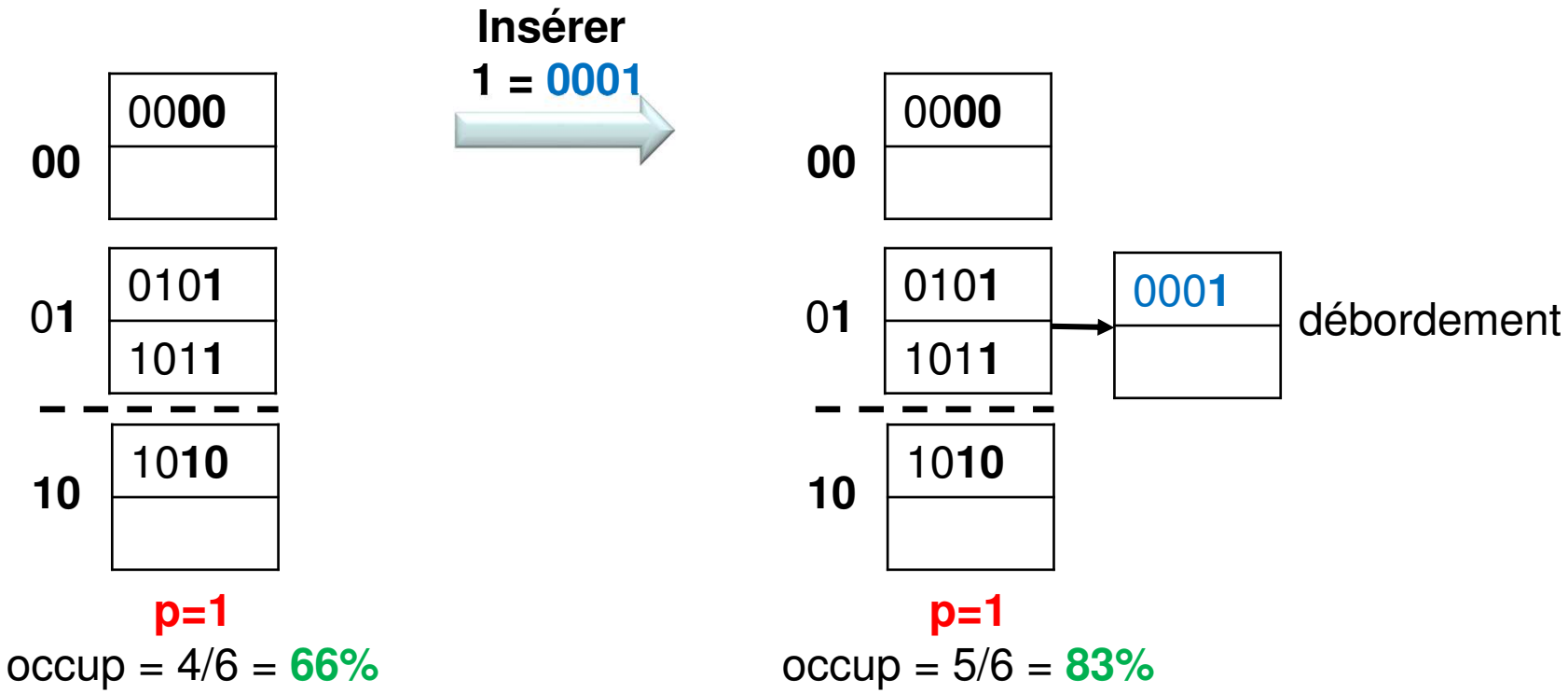
Hachage linéaire : insertion

- 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)

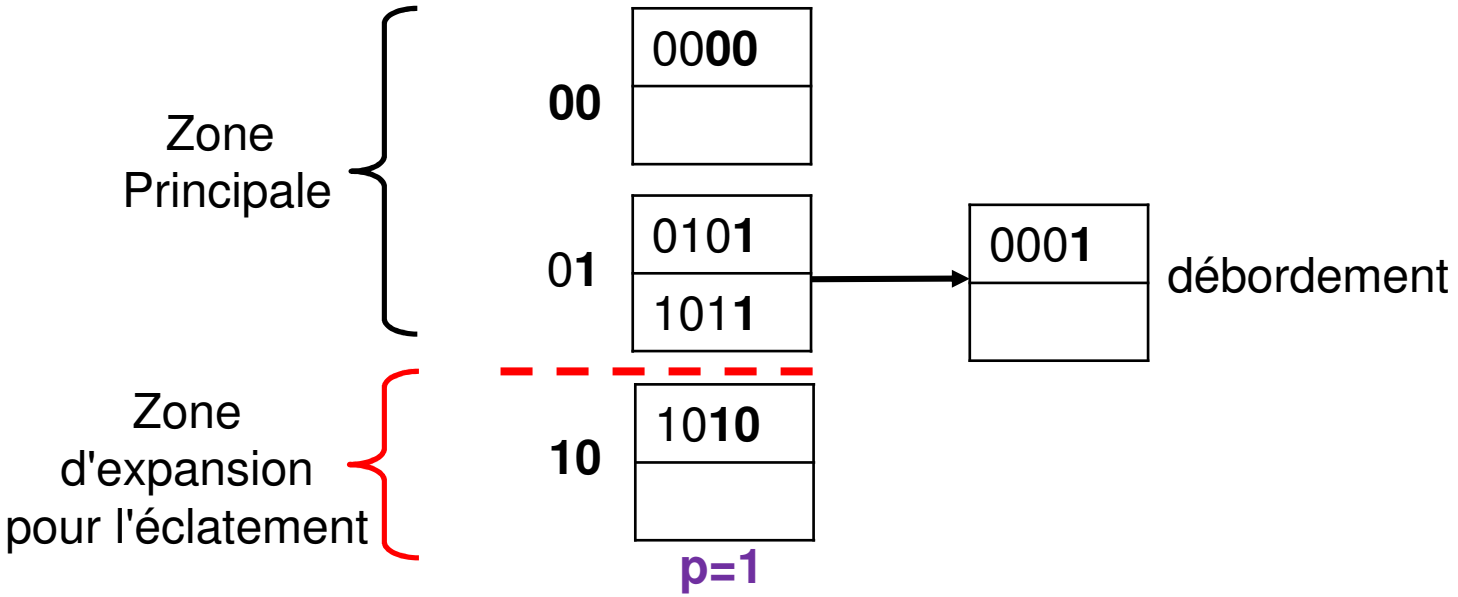


Hachage linéaire : accès

Méthode pour accéder à la valeur v :

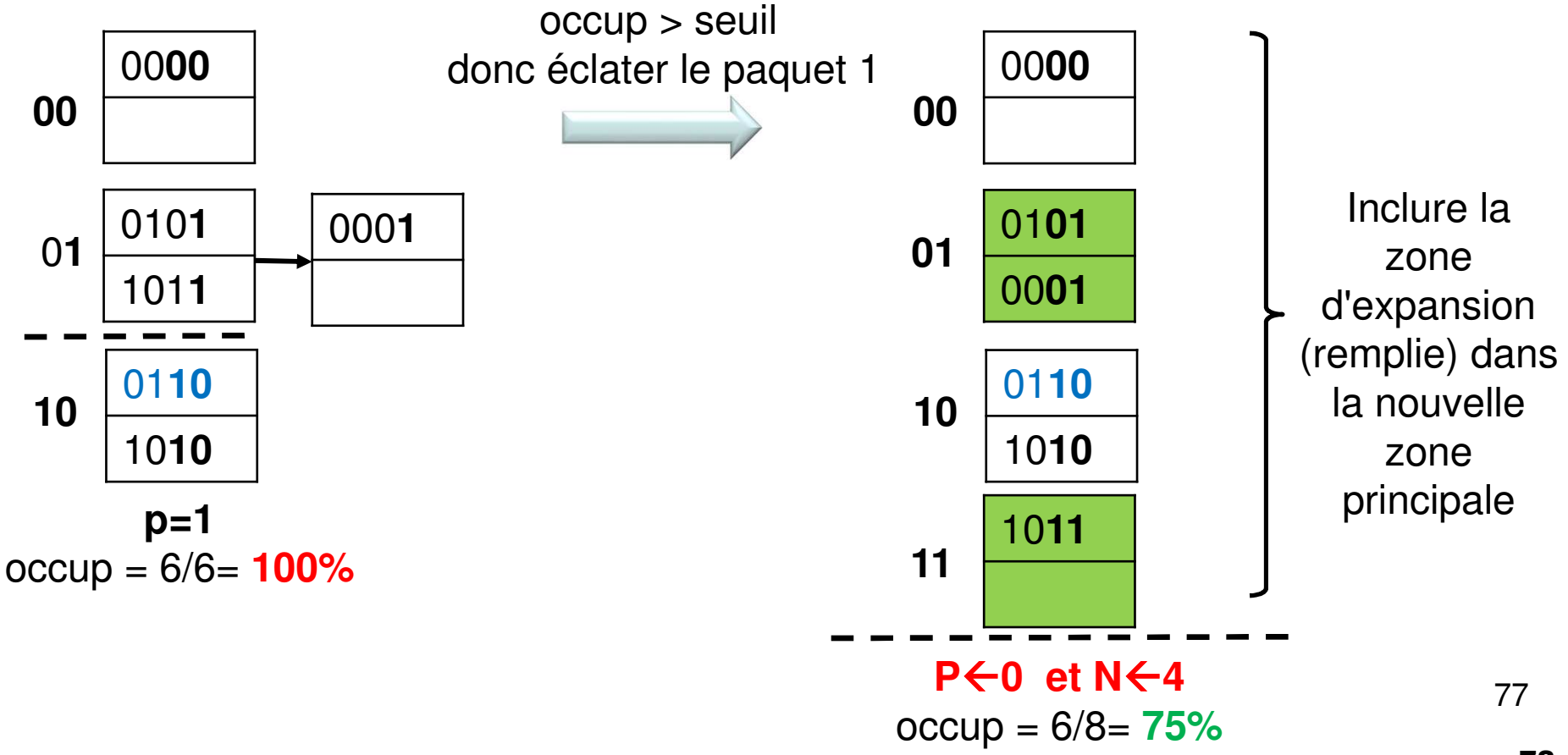
- Rappel: la zone principale a une taille de $N*2^i$
- Calculer $k = h_i(v)$ et $k' = h_{i+1}(v)$
- Si k >= p alors lire le paquet k sinon lire le paquet k'

- Exemple pour N=2, i=0, p=1 :
  - Lire la valeur 13 (1011) dans le paquet k=*13 mod 2* = 1
  - Lire la valeur 10 (1010) dans le paquet k'=*10 mod 4* = 2 (car k=0 et k<p)



Insertion
et fin d'une série d'éclatements

- Insérer 6 = 0110 dans le paquet k'=*10 mod 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 Ariege 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={PO, 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

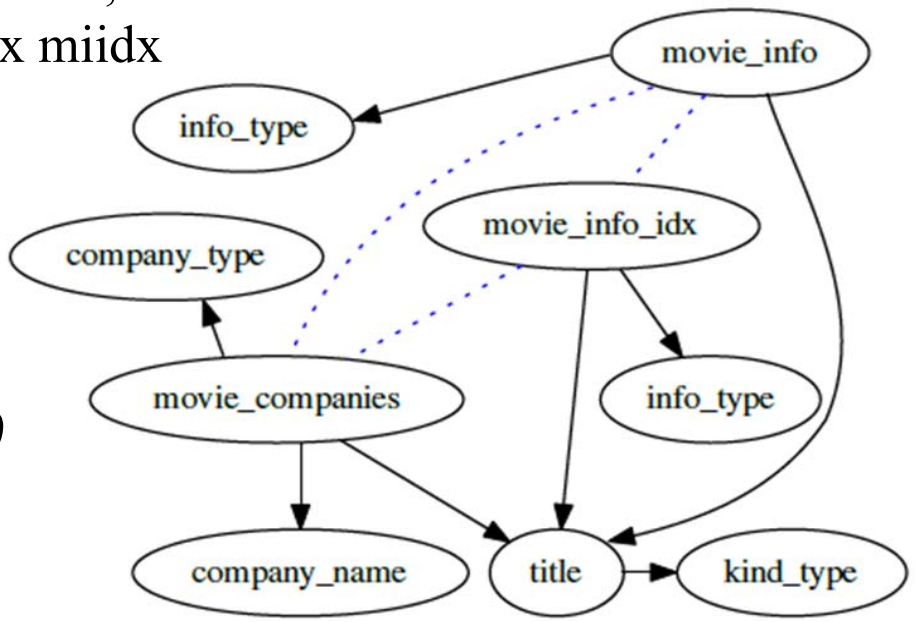
SAM 4I803
Cours 3 : Traitement et optimisation de requêtes

Intro

- La base de films IMDB
  - https://www.imdb.com/interfaces/
- Benchmark réaliste
  - Ref: How Good Are Query Optimizers, Really?
    - VLDB 2015 http://www.vldb.org/pvldb/vol9/p204-leis.pdf
  - Cast_info: 36M tuples
  - Movie_info: 15M tuples
  - Données du benchmark
    - ftp://ftp.fu-berlin.de/pub/misc/movies/database/

Requête réelle

SELECT cn.name, mi.info, miidx.info
FROM company_name cn, company_type ct,
info_type it, info_type it2, title t,
kind_type kt, movie_companies mc,
movie_info mi, movie_info_idx miidx
WHERE cn.country_code =’[us]’
AND ct.kind = ’production companies’
AND it.info = ’rating’
AND it2.info = ’release dates’
AND kt.kind = ’movie’
AND *prédicats de jointure (11 égalités)*



Toutes les requêtes: http://db.in.tum.de/people/sites/leis/qo/job.tgz

Questions

- Cardinalité
  - Estimations précises → requête rapide ?
  - Estimations très approximatives → requêtes lentes ?
- Ordre des jointures important ?
- Important d'explorer beaucoup de plans ?

Problème

EMP(ENO, ENAME, TITLE)
PROJECT(PNO, PNAME, BUDGET)
WORKS(ENO, PNO, RESP, DUR)

Soit la requête

pour chaque projet de budget > 250 qui emploie plus de 2 employés, donner le nom et le titre des employés

Comment l'exprimer en SQL ?

5

Un plan d'exécution possible

```
SELECT  DISTINCT Ename, Title
        FROM    Emp, Project, Works
        WHERE   Budget > 250
        AND     Emp.Eno=Works.Eno
        AND     Project.Pno=Works.Pno
        AND     Project.Pno IN
                (SELECT Pno
                 FROM   Works
                 GROUP BY Pno
                 HAVING COUNT(*) > 2)
```

$T_1 \leftarrow$ *Lire* la table `Project` et *sélectionner* les tuples de `Budget` > 250

$T_2 \leftarrow$ *Joindre* T_1 avec la relation `Works`

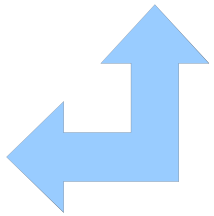
$T_3 \leftarrow$ *Joindre* T_2 avec la relation `Emp`

$T_4 \leftarrow$ *Grouper* les tuples de `Works` sur `Pno` et pour les groupes qui ont plus de 2 tuples, *projeter* sur `Pno`

$T_5 \leftarrow$ *Joindre* T_3 avec T_4

$T_6 \leftarrow$ *Projeter* $T5$ sur `Ename`, `Title`

Résultat $\leftarrow$ *Éliminer doublons* dans T_6



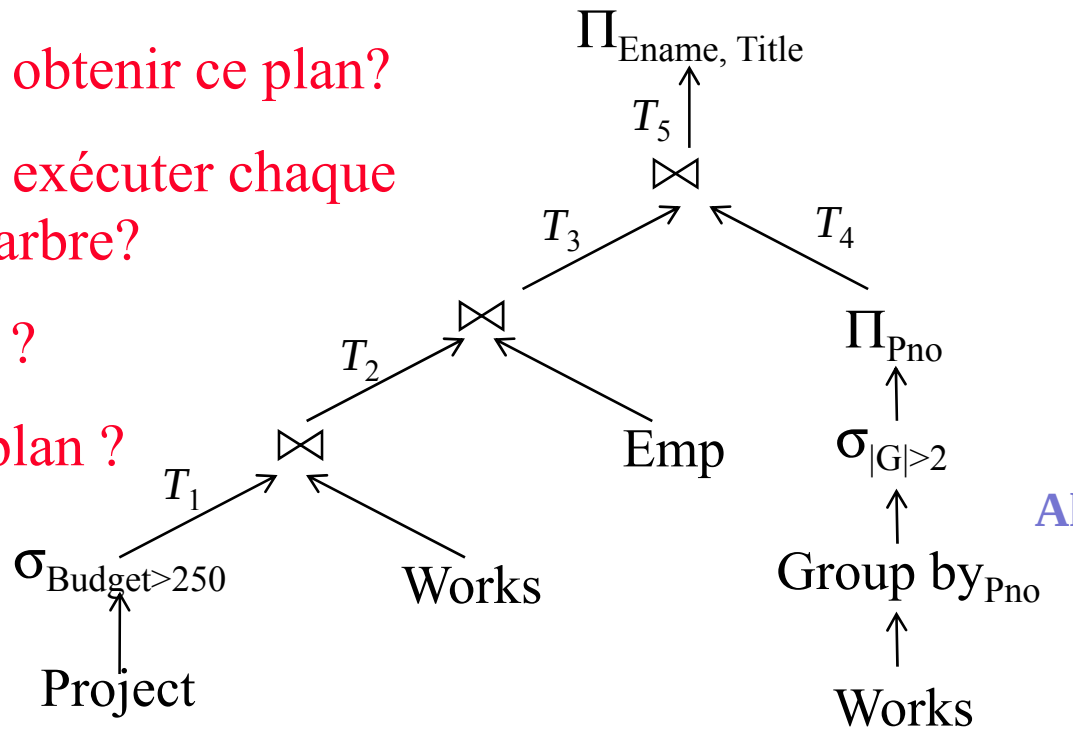
Représentation algébrique

1.Comment obtenir ce plan?

2.Comment exécuter chaque nœud/sous-arbre?

3.Quel coût ?

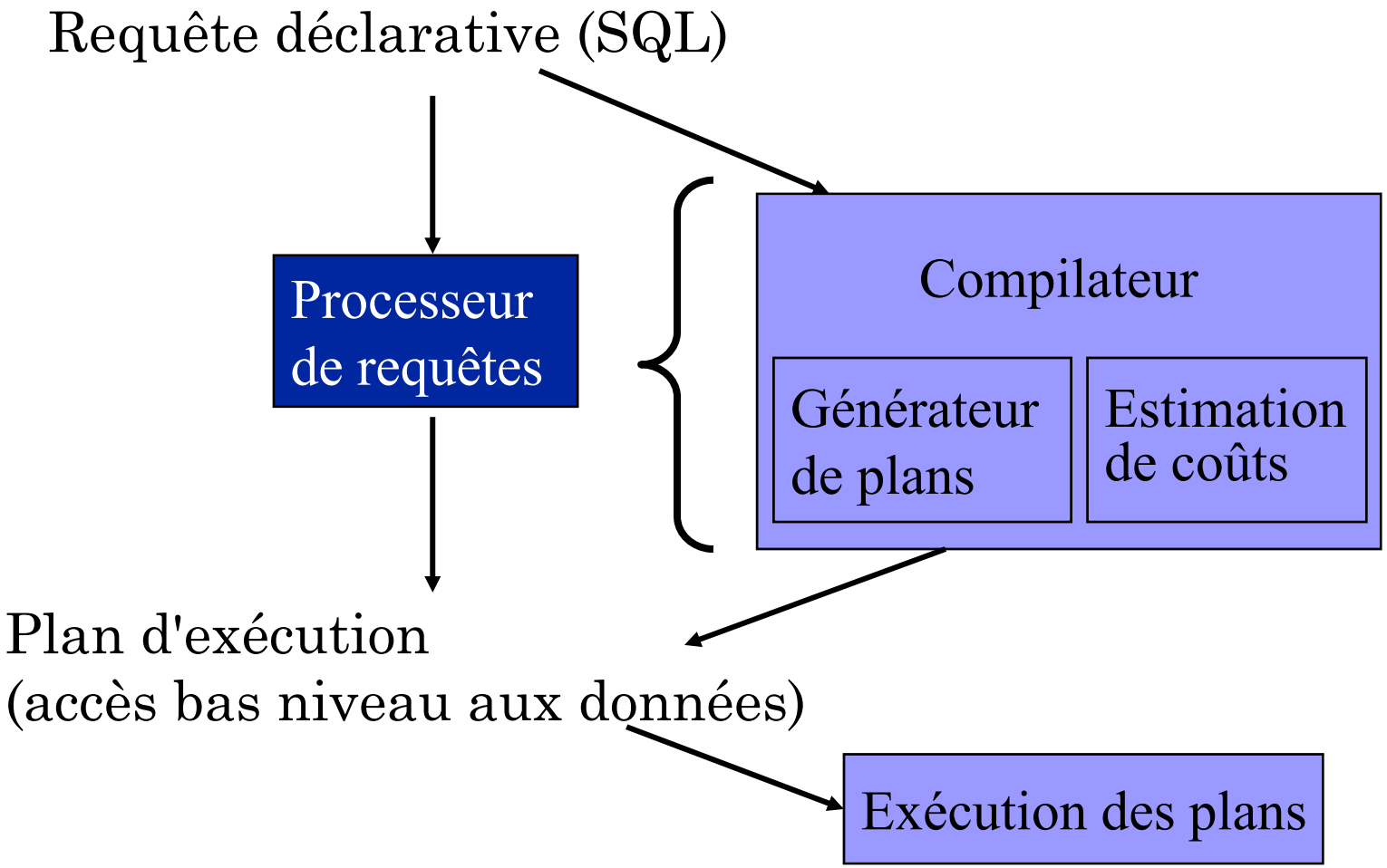
4.Meilleur plan ?



Algèbre étendue

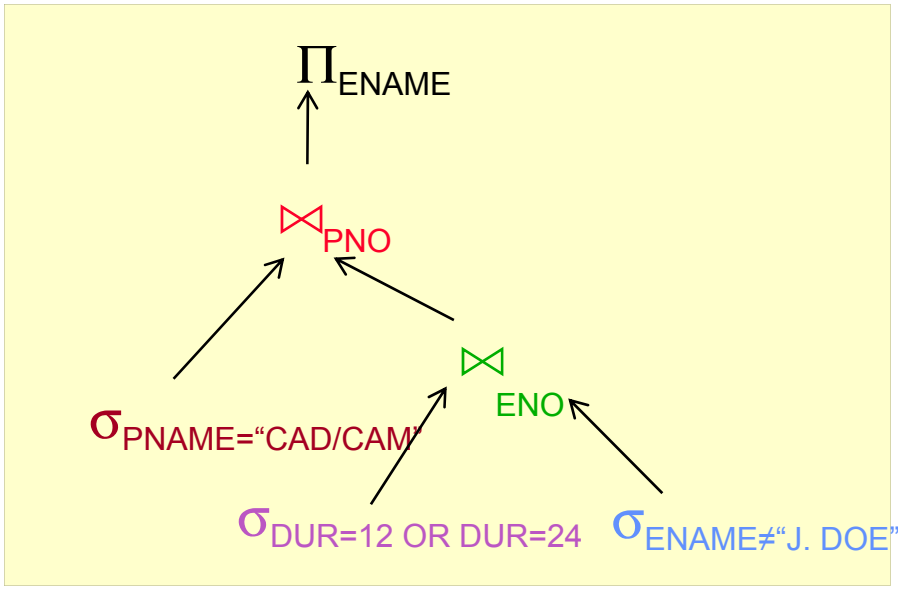
$\Pi_{Ename, Title}(\Pi_{Pno}(\sigma_{|G|>2}\ Group_{Pno}\ (Works)) \bowtie (Emp \bowtie ((\sigma_{Budget>250000}\ Project) \bowtie Works)))$

•Traitement des requêtes



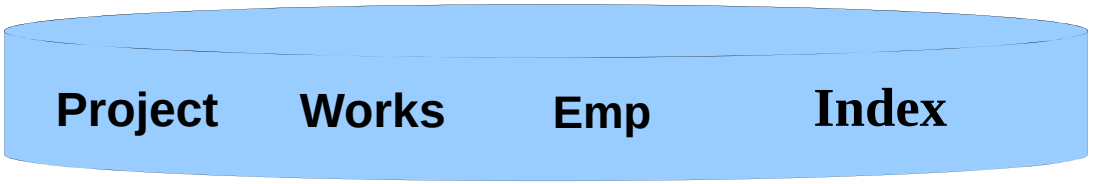
Exécution d'un plan

MÉMOIRE CENTRALE

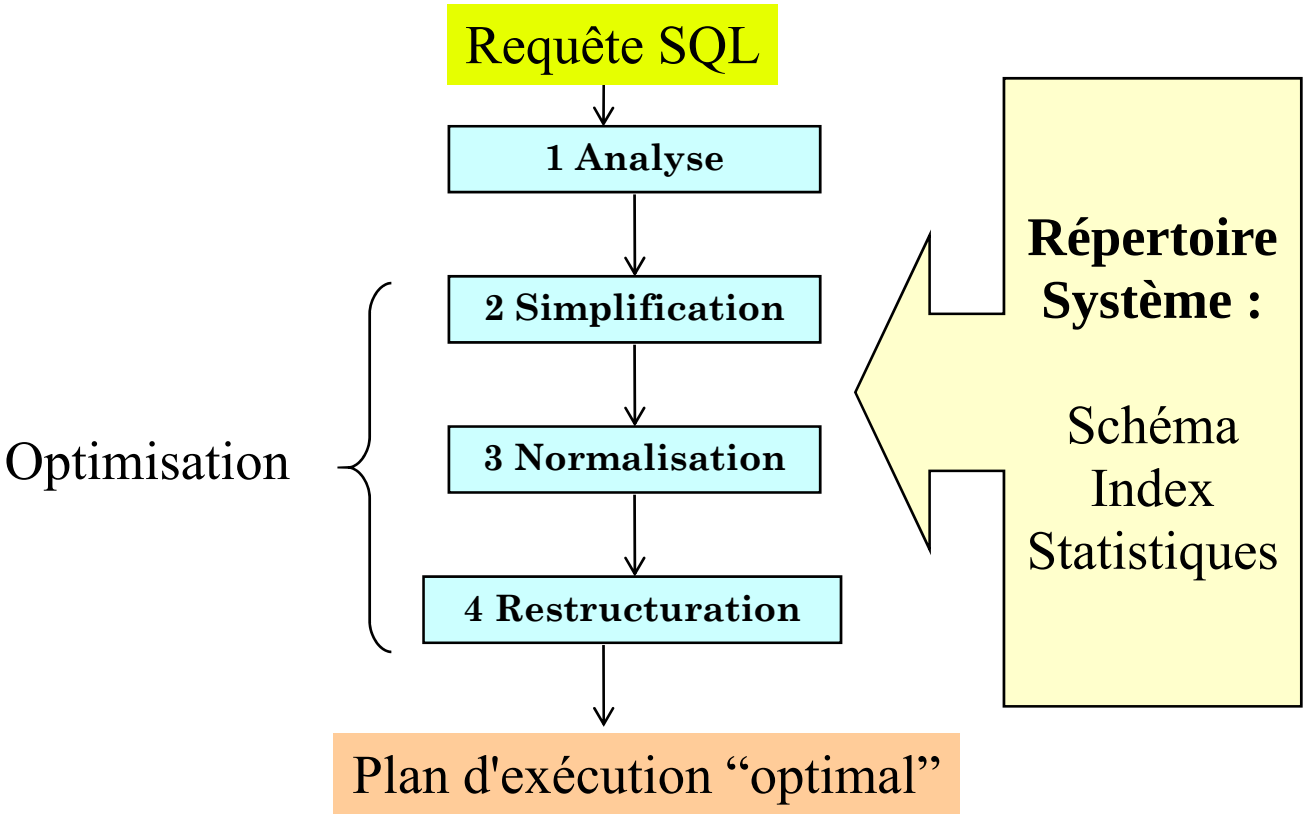


Écriture données temp. ***pages*** Lecture données / index

DISQUE



Étapes du traitement d'une requête



Normalisation de requête

- Analyse lexicale et syntaxique
  - vérification de la validité de la requête
  - vérification des attributs et relations
  - vérification du typage de la qualification
- Mise de la requête en forme normale
  - forme normale conjonctive

    $(p_{11} \vee p_{12} \vee \ldots \vee p_{1n}) \wedge \ldots \wedge (p_{m1} \vee p_{m2} \vee \ldots \vee p_{mn})$
  - forme normale disjonctive

    $(p_{11} \wedge p_{12} \wedge \ldots \wedge p_{1n}) \vee \ldots \vee (p_{m1} \wedge p_{m2} \wedge \ldots \wedge p_{mn})$
  - OR devient union
  - AND devient jointure ou sélection

Simplification

- Pourquoi simplifier?
  - plus une requête est simple, plus son exécution peut être efficace
- Comment? en appliquant des transformations
  - élimination de la redondance
    - règles d'idempotence

      $p_1 \wedge \neg(p_1) \equiv$ faux

      $p_1 \wedge (p_1 \vee p_2) \equiv p_1$

      $p_1 \vee$ faux $\equiv p_1$

      …
  - application de la transitivité (att1=att2 ,att2=att3)
- Éliminer des opérations redondantes : **élagage**
  - ex. : pas besoin de distinct après une projection sur une clé
- utilisation des règles d'intégrité
  - CI : att1 <100 Q: … where att1 > 1000…

Exemple de simplification

```
SELECT      Title
FROM        Emp
WHERE       Ename = 'J. Doe'                    P1
OR             (NOT(Title = 'Programmer')       ¬P2
AND         (Title = 'Programmer'               P2
OR             Title = 'Elect. Eng.')           P3
AND         NOT(Title = 'Elect. Eng.'))         ¬P3
```

$P1 \vee (\neg P2 \wedge (P2 \vee P3) \wedge \neg P3)$

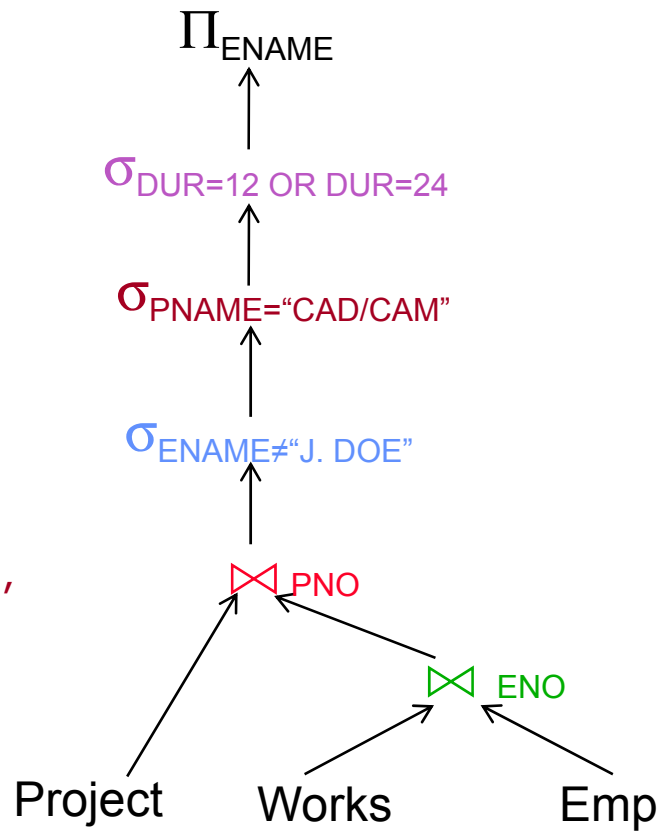
```
SELECT      Title
FROM        Emp
WHERE       Ename = 'J. Doe'
```

Traduction en algèbre

Conversion en arbre algébrique

Exemple :

```
SELECT   Ename
FROM     Emp, Works, Project
WHERE    Emp.Eno = Works.Eno
AND      Works.Pno = Project.Pno
AND      Emp.Ename <>'J.Doe'
AND      Project.name = 'CAD/CAM'
AND      (Works.Dur=12 OR
           Works.Dur=24)
```



Heuristiques

Observation : les opérations manipulant moins de données sont plus rapides (sélectivité corrélée à la performance)

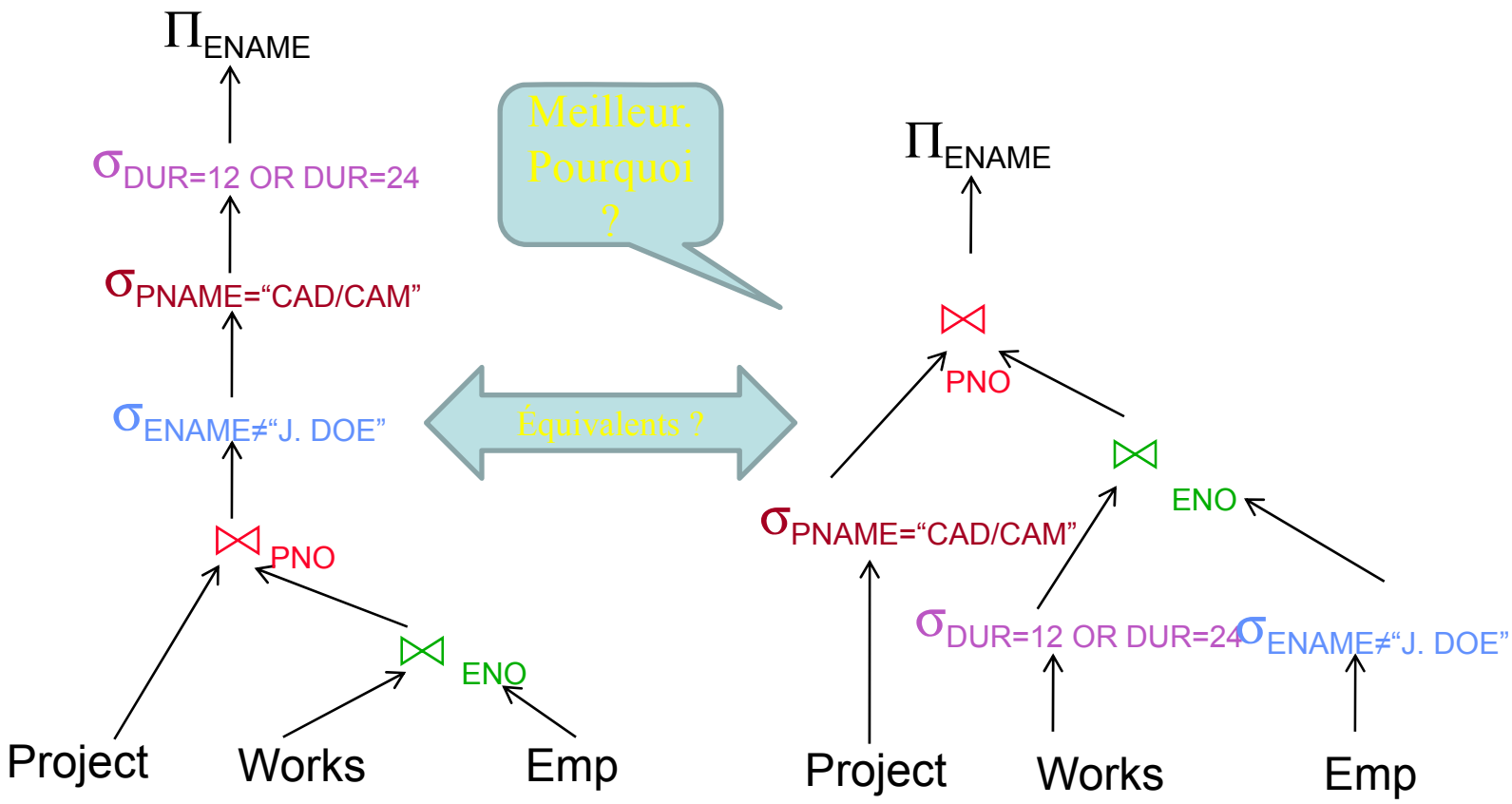
Objectif : déterminer un ordre pour les opérations, censé être efficace.

Méthode:

Commencer par traiter les opérations les **plus sélectives** (projection, sélection) et de manière à réduire la taille des données d'entrée pour les opérateurs suivants (jointures).

La place en mémoire est un facteur primordial pour l'efficacité d'une jointure (cf. algo de jointure, cours suivant)

Exemple


Π ENAME
σ DUR=12 OR DUR=24
σ PNAME="CAD/CAM"
σ ENAME≠"J. DOE"
⋈ PNO
⋈ ENO
Project
Works
Emp
Meilleur. Pourquoi ?
Équivalents ?
Π ENAME
⋈ PNO
⋈ ENO
σ PNAME="CAD/CAM"
σ DUR=12 OR DUR=24
σ ENAME≠"J. DOE"
Project
Works
Emp

Optimisation basée sur le coût

- Elaborer des plans
  - arbre algébrique, restructuration, ordre d'évaluation
- Estimer leurs coûts
  - fonctions de coût
    - en terme de temps d'exécution
    - coût I/O + coût CPU
    - poids très différents
      - par ex. coût I/O = 1000 * coût CPU
- Choisir le meilleur plan
  - Espace de recherche : ensemble des expressions algébriques équivalentes pour une même requête
  - algorithmes de recherche:
    - parcourir l'espace de recherche
    - algorithmes d'optimisation combinatoire

Restructuration

- Objectif : choisir l'ordre d'exécution des opérations algébriques (élaboration du plan logique).

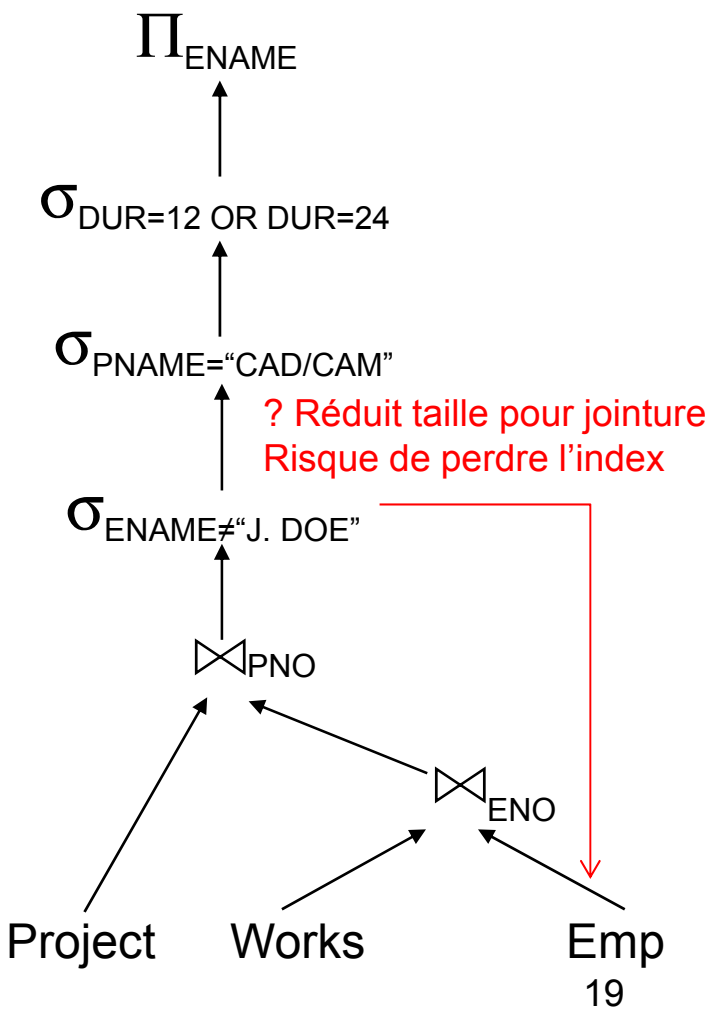
- Conversion en arbre algébrique

- Transformation de l'arbre (optimisation)
  - règles de transformation (équivalence algébriques),
  - estimation du coût des opérations en fonction de la taille
  - Estimation du résultat intermédiaire (taille et ordre?)
  - En déduire l'ordre des jointures

Restructuration

- Conversion en arbre algébrique
- Exemple

```
SELECT  Ename
FROM    Emp, Works, Project
WHERE   Emp.Eno=Works.Eno
AND     Works.Pno=Project.Pno
AND     Ename NOT='J.Doe'
AND     Pname = 'CAD/CAM'
AND     (Dur=12 OR Dur=24)
```



Coût d'un plan

- Fonction de coût = estimation du temps écoulé pour
  - Accéder aux données :
    - temps I/O exprimé en nombre de pages lues ou écrites
  - Calculer le résultat d'une opération à partir des données lues
    - Temps CPU dépend du nombre d'instructions
- Estimation du coût d'exécution de chaque noeud de l'arbre algébrique
  - utilisation de pipelines ou de relations temporaires importante
    - Pipeline : les tuples sont passés directement à l'opérateur suivant.
      - Pas de relations intermédiaires (petites mémoires, ex. carte à puce).
      - Permet de paralléliser (BD réparties, parallèle)
      - Intéressant même pour cas simples : $\sigma_{F \wedge F'}(R)$, index sur F' $\rightarrow \sigma_F(\sigma_{F'}(R))$
    - Relation temporaire : permet de trier mais coût de l'écriture

Estimer la cardinalité d'une opération

- Estimation du nombre de nuplets résultant de chaque nœud par rapport à ses entrées

  – Permet d'estimer le coût de l'opération suivante
  – sélectivité des opérations – "facteur de réduction"
  – propagation d'erreur possible
  – basé sur les statistiques maintenues par le SGBD

Statistiques

- Relation
  - cardinalité : card(R)
  - taille d'un tuple : largeur(R)
  - fraction de tuples participant une jointure / attribut
  - …
- Attribut
  - cardinalité du domaine
  - nombre de valeurs distinctes **D**(R,A) = $\Pi_A(R)$
  - Valeur max, valeur min
- Hypothèses
  - Indépendance entre différentes valeurs d'attributs
  - Distribution uniforme des valeurs d'attribut dans leur domaine
  - Sinon, il faut maintenir des histogrammes (voir diapo suivante)
- Stockage :
  - Les statistiques sont des métadonnées, stockées sous forme relationnelle (cf. TME)
  - Rafraîchies périodiquement, pas à chaque fois.

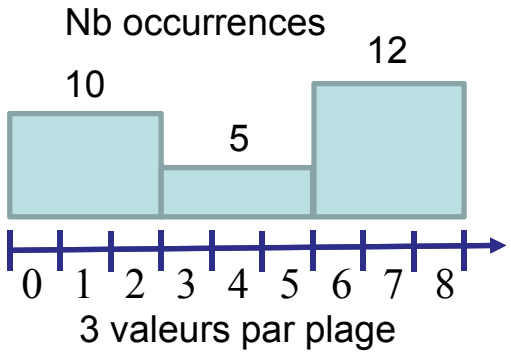
Le fameux compromis L/E

Histogrammes

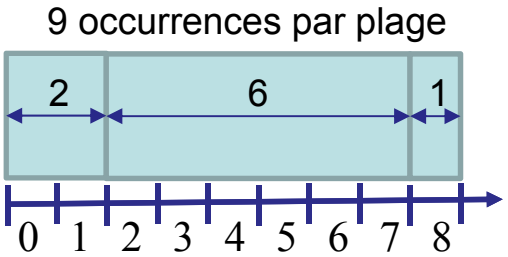
Sert à estimer la cardinalité de la requête :
Select * from R where A = 8 ?

A	card
0	7
1	2
2	1
3	5
4	0
5	0
6	2
7	1
8	9

Histogramme Equilarge
Plages de valeurs de même taille.
Hypothèse d'uniformité
dans un intervalle



Histogramme Equiprofond
Plages de valeurs contenant le même nombre d'occurrences.
Plus précis pour les valeurs fréquentes.



Card **estimée** pour A=8 ?

card = 12 / 3 = **4**

Card = **9**

Cardinalité réelle = **9**

Cardinalité d'une sélection
Facteur de sélectivité

- Soit la **sélection** notée : $\sigma_{pred}(R)$
  - *pred* est le prédicat de sélection
  - En SQL : select * from R where pred

- **Cardinalité** d'une sélection

$$card(\sigma_{pred}(R)) = SF(pred) * card(R)$$

SF est une *estimation* de la **sélectivité du prédicat** *pred*, dont la forme générale est "nombre d'éléments sélectionnés / nombre d'éléments possibles"

Facteur de sélectivité : Sélection sur un attribut

- Egalité : sélection de valeurs
  - Nbval: nombre de valeurs sélectionnées
  - **SF = Nbval / D(R,A)**

pred	Nbval
A = v	1
A in (v1, v2, ..., vk)	k
A= v1 OR A= v2 OR ... OR A= vk	k

- Inégalité : sélection d'un intervalle
  - Intervalle = segment d'un domaine (continu)
  - Bornes du domaine : A ∈[min(A), max(A)]
  - Longueur du domaine : L(A) = max(A) – min(A)
  - **SF = longueur du segment sélectionné / L(A)**

pred	Longueur
A < v	v – min(A)
A > v	max(A) – v
A ≥ v1 and A ≤ v2 s'écrit aussi: A between v1 and v2	v2 – v1
A ≤ v1 or A ≥ v2	max(A) - v2 + v1 – min(A)

Facteur de sélectivité
Sélection sur plusieurs attributs

- Le prédicat est **composé** de plusieurs prédicats:
  - $pred_A$, $pred_B$, ...
  - Exple: $pred_{age}$: age > 20 $pred_{Ville}$: ville='Paris'

- Conjonction: $pred_A$ AND $pred_B$
  - SF($pred_A$ AND $pred_B$) = SF($pred_A$) * SF($pred_B$)
  - Exple:
    - Pred : age > 20 **AND** ville = 'Paris'
    - SF(pred) = SF(age > 20) * SF(ville='Paris')

- Disjonction entre 2 termes : $pred_A$ OR $pred_B$
  - SF($pred_A$ OR $pred_B$) = SF($pred_A$) + SF($pred_B$) - SF($pred_A$) * SF($pred_B$)
  - Exple:
    - Pred : age >20 **OR** ville = 'Paris'
    - SF(pred) = SF(age > 20) + SF(ville = 'Paris') – SF(age > 18) * SF(ville = 'Paris')

- Rmq: peut se généraliser pour *n* prédicats
  - Disjonction n-aire: *(n-1)* disjonctions binaires

Facteur de sélectivité
Sélection avec **négation**

- Négation: complément à 1
- SF (not (pred)) = 1 – SF(pred)

- Exemple:
  - jourFermeture = 'Dimanche'
    - SF(jourFermeture = 'Dimanche') = 1/7

  - jourFermeture <> 'Dimanche'
    - SF(jourFermeture <> 'Dimanche') = 1 – 1/7 = 6/7

Cardinalité des opérations

Projection

$card(\Pi_A(R)) <= card(R)$ (égalité si A est *unique*)

Produit cartésien

$card(R \times S) = card(R) \cdot card(S)$

Union

borne sup. : $card(R \cup S) = card(R) + card(S)$

borne inf. : $card(R \cup S) = max\{card(R), card(S)\}$

Différence

borne sup. : $card(R–S) = card(R)$

borne inf. : 0

Cardinalité d'une jointure naturelle

- Jointure naturelle entre 2 tables R et S sur l'attribut A
  - A est clé "primary key" de R(A, ...)
    - donc card(R) / D(R, A) = **1**
  - A est clé étrangère "foreign key" dans S (...., A*, ...)
  - domaine(S.A) ⊆ domaine(R.A)
  - Pour chaque tuple de S, il existe **un et un seul** tuple de R

- $card(R \bowtie_A S) = card(S) . 1$

- Exple :
  - **Etu**(<u>nomE</u>, age) **Note**(nomE*, codeUE*, note) **UE**(<u>codeUE</u>, niveau, titre)
  - 100 Etudiants, 600 notes, 30 UE
  - $card(Etu \bowtie_{nomE} Note) = card(Note) = 600$
  - $card(Note \bowtie_{codeE} UE) = card(Note) = 600$
  - $card(Etu \bowtie_{nomE} Note \bowtie_{codeE} UE) = card(Note) = 600$

Cardinalité d'une jointure entre deux clé étrangères

- Cas d'une jointure $R \bowtie_A S$ entre 2 clés étrangères
  - Entité(A, ...), R(..., A*, ...) , S(..., A*, ...)
  - dom(R.A) ⊆ dom(Entité.A) donc D(Entité, A) ≥ D(R, A)
  - dom(S.A) ⊆ dom(Entité.A) donc D(Entité, A) ≥ D(S, A)

- $card(R \bowtie_A S) = card(R) * card(S) / D(Entité, A)$

- Exple :
  - **Etu**(nomE, age) et D(Etu, nomE) = **100**
  - **InscritSport**(sport, nomE*, annee)
    - D(InscritSport, nomE) = 20
  - **Résa** (codeLivre*, nomE*, date, bibliothèque, durée)
    - D(Resa, NumE) = 50
  - 100 Etudiants, 40 InscritSport, 200 Résa de livres
  - $card(InscritSport \bowtie_{nomE} Résa) = 40 * 200 / \mathbf{100} = 80$

Cardinalité d'une jointure entre 2 sélections

- Jointure entre deux expressions contenant des sélections
- Réécrire la jointure :
  - Exprimer la jointure avant la sélection
  - Commutativité de la composition des opérations :
    - jointure(sélection, sélection) = sélection(sélection(jointure))

$$card(\ \sigma_{p1}(R) \quad \bowtie_A \quad \sigma_{p2}(S)\ ) =$$

$$SF(p1) * SF(p2) * card(R \bowtie_A S)$$

Espace de recherche

- Caractérisé par les plans “équivalents” pour une même requête
  - ceux qui donnent le même résultat
  - générés en appliquant les règles de transformation vues précédement
- Le coût de chaque plan est en général différent
- L'ordre des jointures est important

Règles de transformation

- Commutativité des opérations binaires
  - $R \times S \equiv S \times R$
  - $R \bowtie S \equiv S \bowtie R$
  - $R \cup S \equiv S \cup R$
- Associativité des opérations binaires
  - $(R \times S) \times T \equiv R \times (S \times T)$
  - $(R \bowtie S) \bowtie T \equiv R \bowtie (S \bowtie T)$
- Idempotence des opérations unaires
  - $\Pi_{A'}(\Pi_{A''}(R)) \equiv \Pi_{A'}(R)$
  - $\sigma_{p_1(A_1)}(\sigma_{p_2(A_2)}(R)) \equiv \sigma_{p_1(A_1) \wedge p_2(A_2)}(R)$
    - où $R[A]$ et $A' \subseteq A$, $A'' \subseteq A$ et $A' \subseteq A''$

Règles de transformation

- Commutativité de la sélection et de la projection (si proj. des attr. sél.)
- Commutativité de la sélection avec les opérations binaires

$$\sigma_{p(A)}(R \times S) \equiv (\sigma_{p(A)}(R)) \times S$$

$$\sigma_{p(A_i)}(R \bowtie_{(A_j,B_k)} S) \equiv (\sigma_{p(A_i)}(R)) \bowtie_{(A_j,B_k)} S$$

$$\sigma_{p(A_i)}(R \cup T) \equiv \sigma_{p(A_i)}(R) \cup \sigma_{p(A_i)}(T)$$

où A_i appartient à R et T

- Commutativité de la projection avec les opérations binaires

$$\Pi_C(R \times S) \equiv \Pi_{A'}(R) \times \Pi_{B'}(S)$$

$$\Pi_C(R \bowtie_{(A_j,B_k)} S) \equiv \Pi_{A'}(R) \bowtie_{(A_j,B_k)} \Pi_{B'}(S)$$

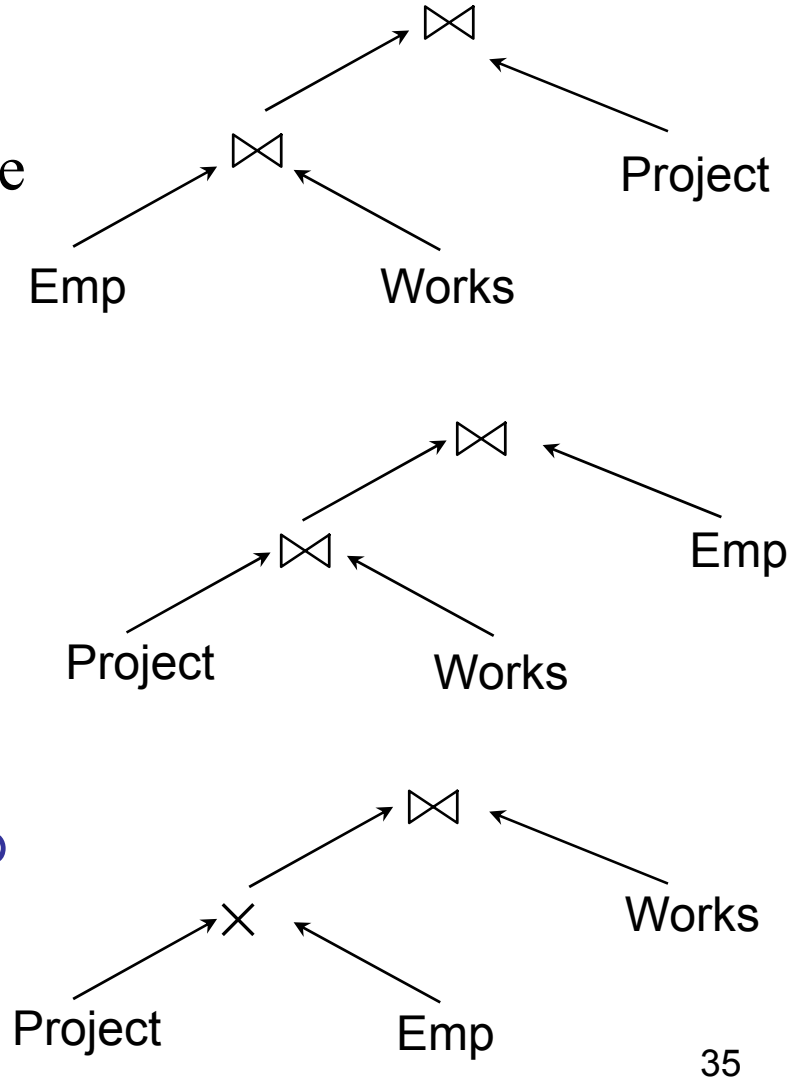
$$\Pi_C(R \cup S) \equiv \Pi_C(R) \cup \Pi_C(S)$$

où $R[A]$ et $S[B]$; $C = A' \cup B'$ où $A' \subseteq A$, $B' \subseteq B$, $A_j \subseteq A'$, $B_k \subseteq B'$

Ordre des jointures

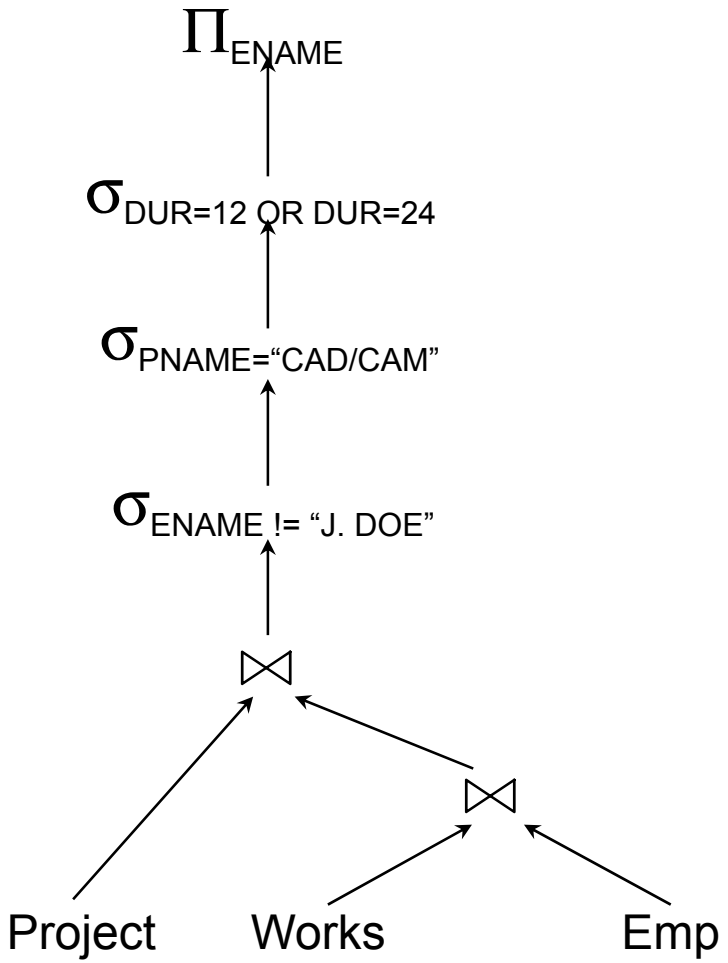
- Avec *N* relations, il y a O(***N!***) ordres de jointures équivalents qui peuvent être obtenus en appliquant les règles de *commutativité* et d'*associativité*

```
SELECT    Ename,Resp
FROM      Emp, Works,Project
WHERE     Emp.Eno=Works.Eno
AND       Works.PNO=Project.PNO
```

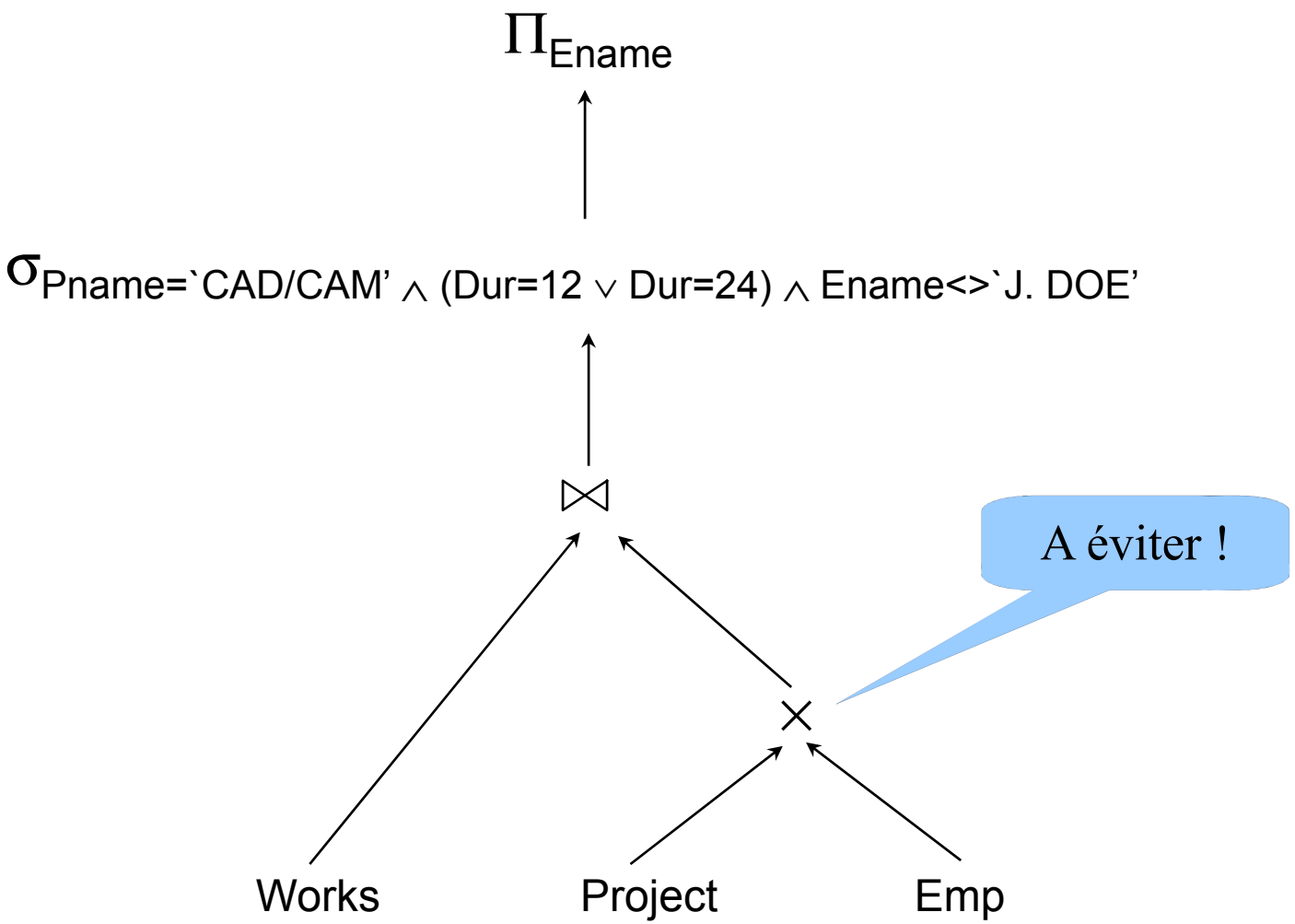


Exemple

```
SELECT Ename
FROM Project p, Works w,
     Emp e
WHERE w.Eno=e.Eno
AND   w.Pno=p.Pno
AND   Ename<>`J. Doe'
AND   p.Pname=`CAD/CAM'
AND   (Dur=12 OR Dur=24)
```

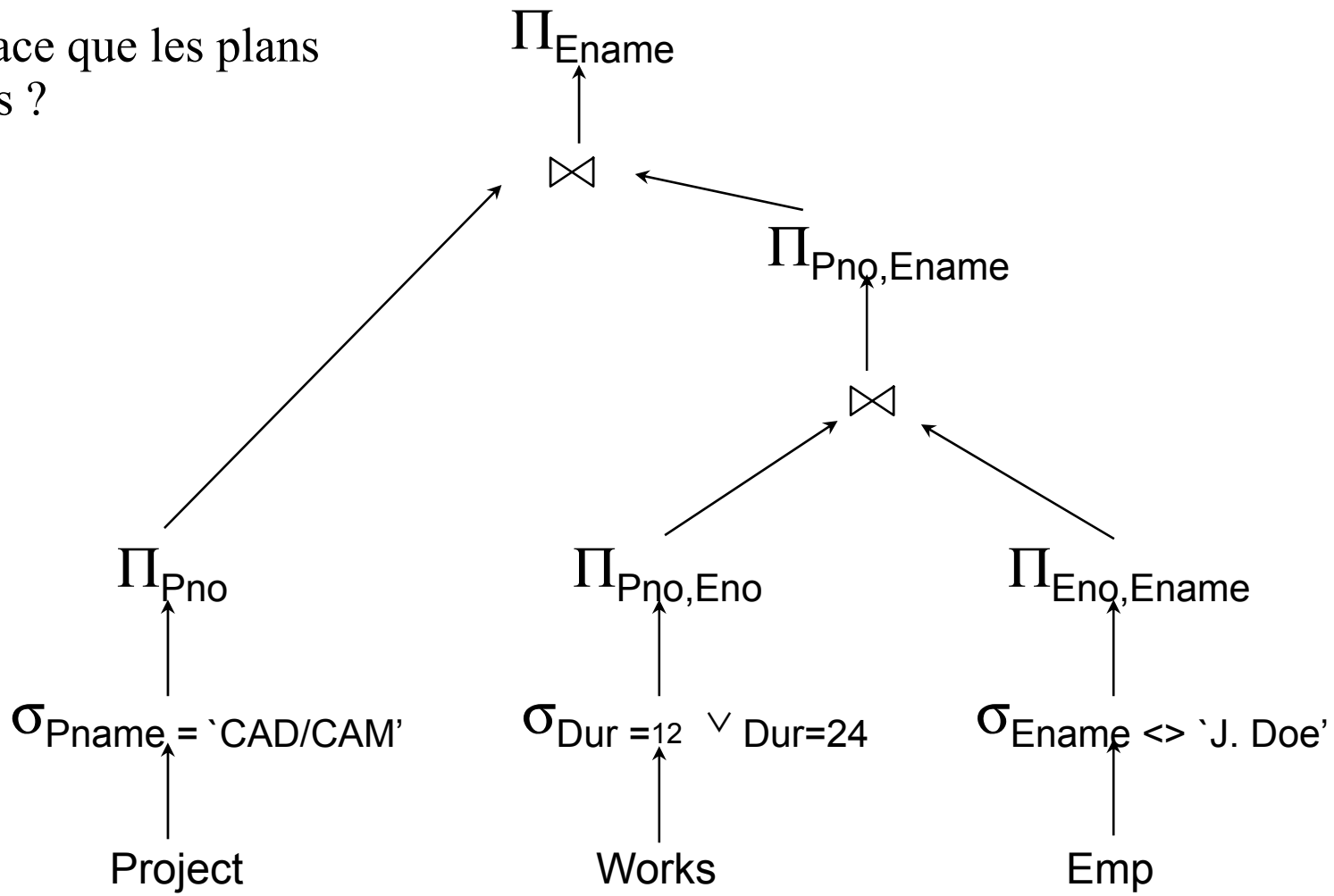


Plan équivalent

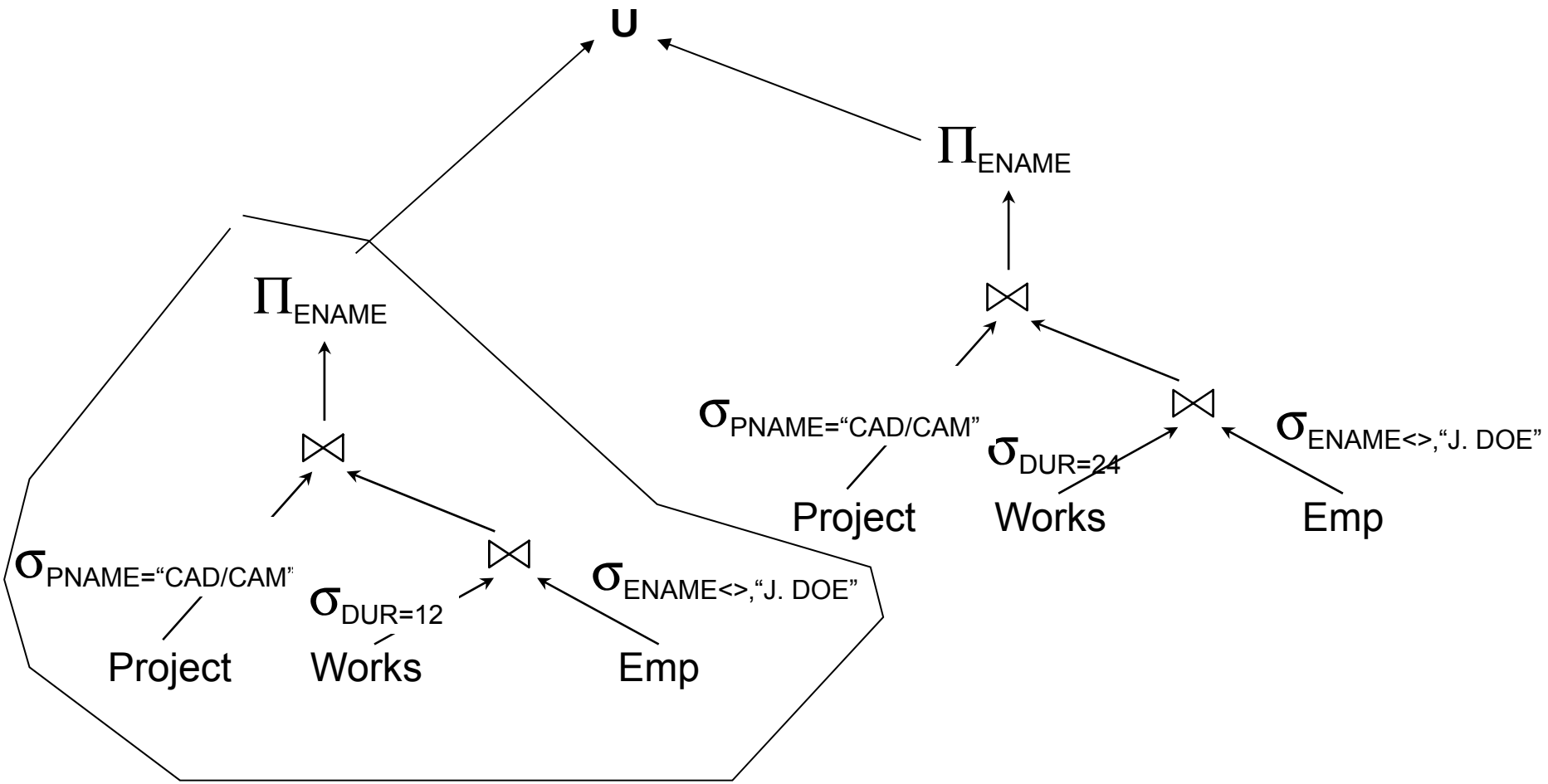


Autre plan équivalent

Plus efficace que les plans précédents ?



Encore un autre plan équivalent

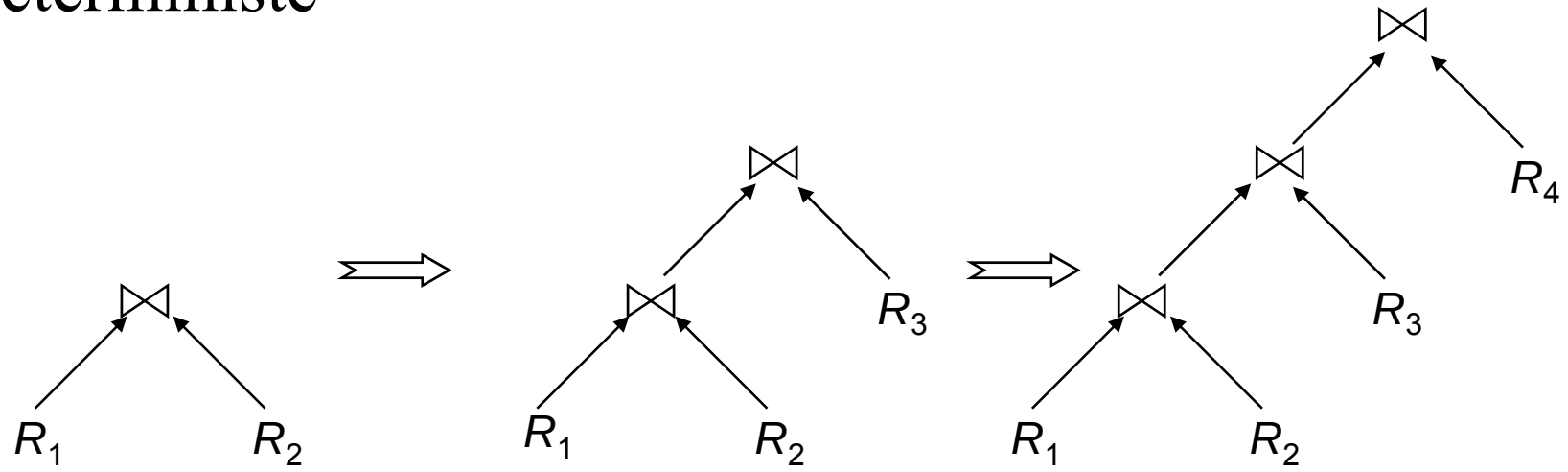


Stratégie de recherche

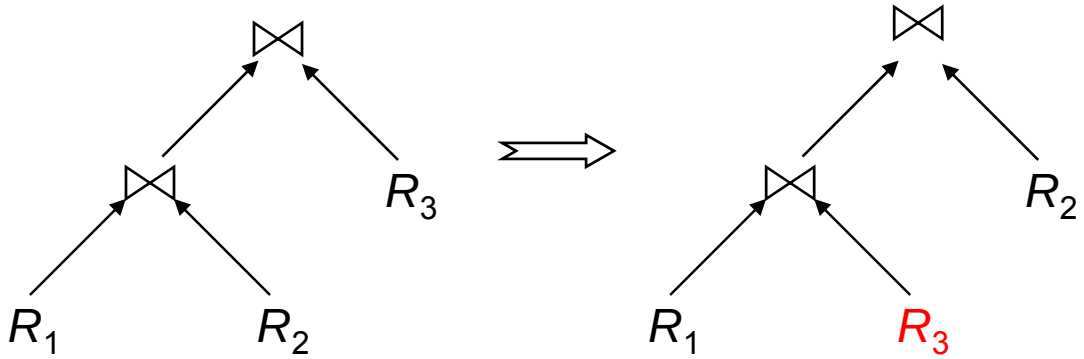
- Il est en général trop coûteux de faire une recherche exhaustive
- Déterministe
  - part des relations de base et construit les plans en ajoutant une relation à chaque étape
  - programmation dynamique: largeur-d'abord
  - excellent jusqu'à 5-6 relations
- Aléatoire
  - recherche l'optimalité autour d'un point de départ particulier
  - réduit le temps d'optimisation (au profit du temps d'exécution)
  - meilleur avec > 5-6 relations
    - **recuit simulé (simulated annealing)**
    - **amélioration itérative (iterative improvement)**

Stratégies de recherche

- Déterministe



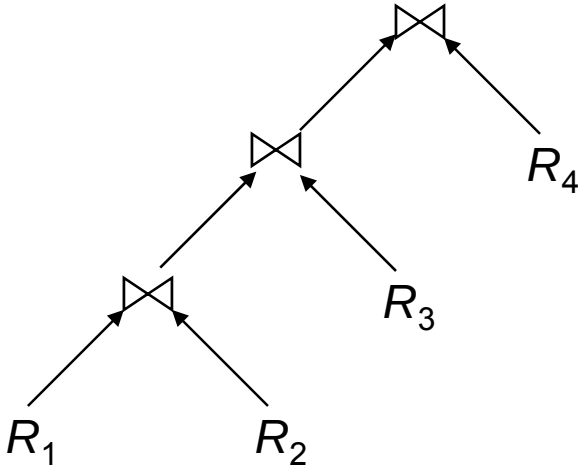
- Aléatoire



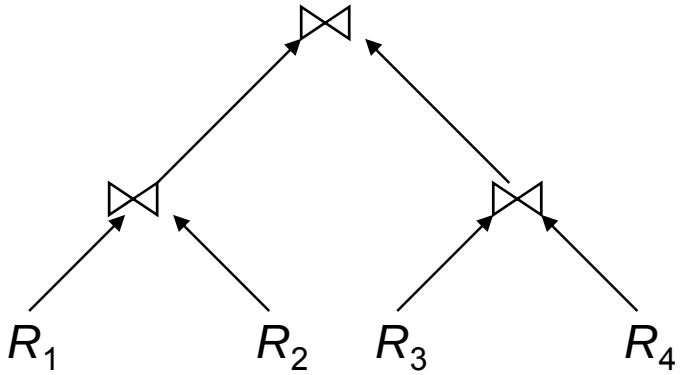
Algorithmes de recherche

- Limiter l'espace de recherche
  - heuristiques
    - par ex. appliquer les opérations unaires avant les autres
    - Ne marche pas toujours (perte d'index, d'ordre)
  - limiter la forme des arbres

Arbre linéaire



Arbre touffu



42

Génération de plan physique

- Sélection :
  - Commencer par les conditions d'égalité avec un index sur l'attribut
  - Filtrer sur cet ensemble de n-uplets ceux qui correspondent aux autres conditions
- Jointure
  - Utilisation des index, des relations déjà triées sur l'attribut de jointure, présence de plusieurs jointures sur le même attribut
- Pipelines ou matérialisation

Conclusion

- Point fondamental dans les SGBD
- Importance des métadonnées, des statistiques sur les relations et les index, du choix des structures d'accès.
- L'administrateur de bases de données peut améliorer les performances en créant de nouveaux index, en réglant certains paramètres de l'optimiseur de requêtes (voir TME)

Conclusion

- Un SGBD doit transformer une requête déclarative en un programme impératif :
  - Plan d'exécution
  - Algèbre
- Calculer les tailles des résultats intermédiaire donne une idée du coût d'un plan mais..
  - Comment mettre en œuvre les opérateurs ?
  - Comment enchaîner les opérateurs ?
  - Comment trouver le meilleur plan en fonction de ce qui précède

Réponses dans les prochains cours

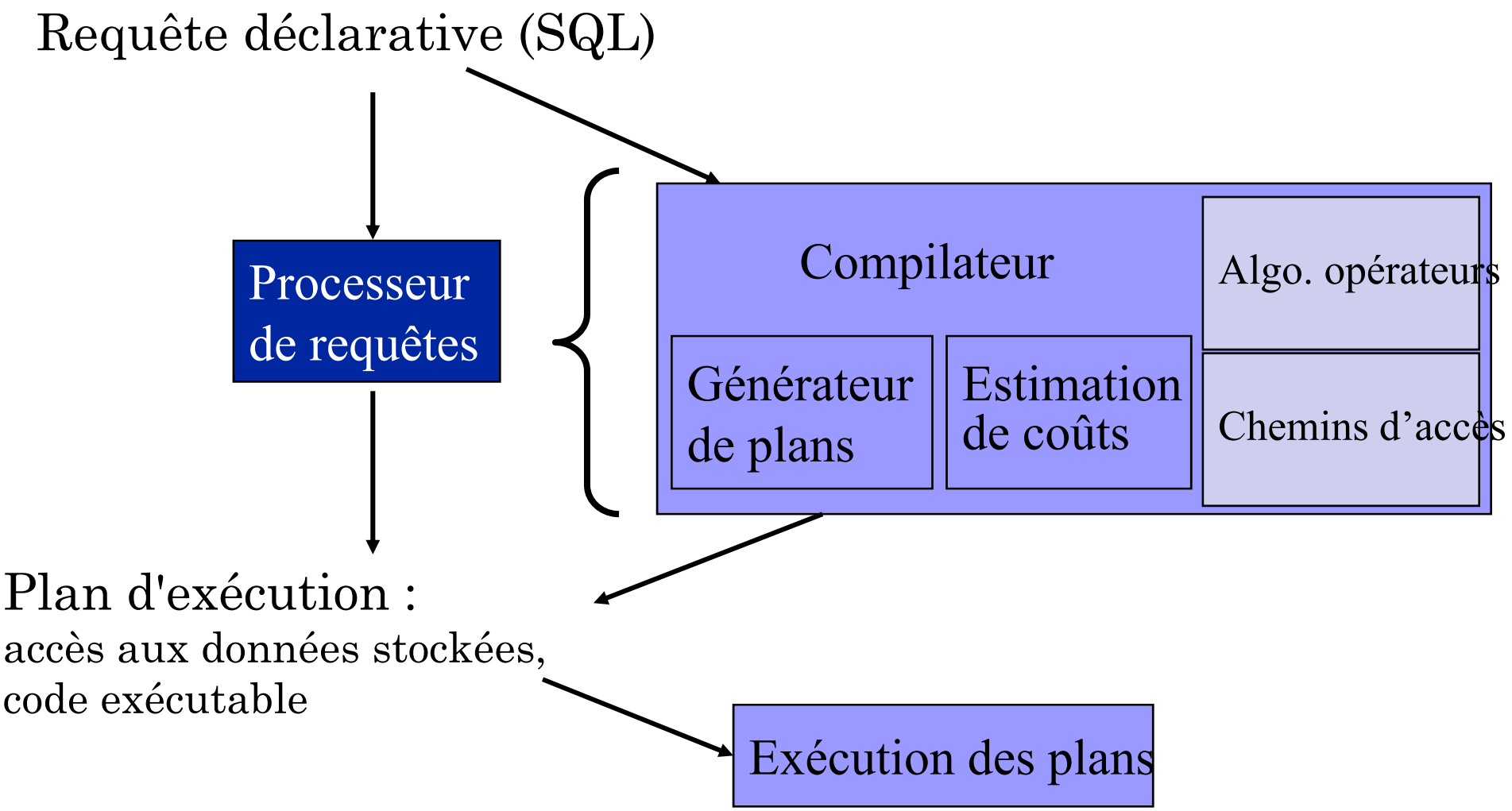
4I803
Cours 4

Opérateurs relationnels
Implémentation et coût

Plan

- Rappel et Objectif
- Notion de pipeline
- Tri
- Sélection
- Projection
- Jointure
- Autres

Traitement des requêtes (rappel)



Objectif

- Comprendre les **algorithmes** qui évaluent les opérateurs relationnels
- Quantifier les accès aux données nécessaire pour évaluer une opération
  - Unité de mesure : la page
  - Les opérations principales sont :
    - sélection, projection, jointure, tri, ...
- Disposer d'un modèle (i.e., des formules) pour déterminer le **coût** d'une opération en termes d'accès aux données
- Hypothèses : coût E/S >> coût CPU
  - Lire un nuplet à partir d'une page de données stockée sur disque dure beaucoup plus longtemps que de calculer un nuplet à partir de données déjà en mémoire.

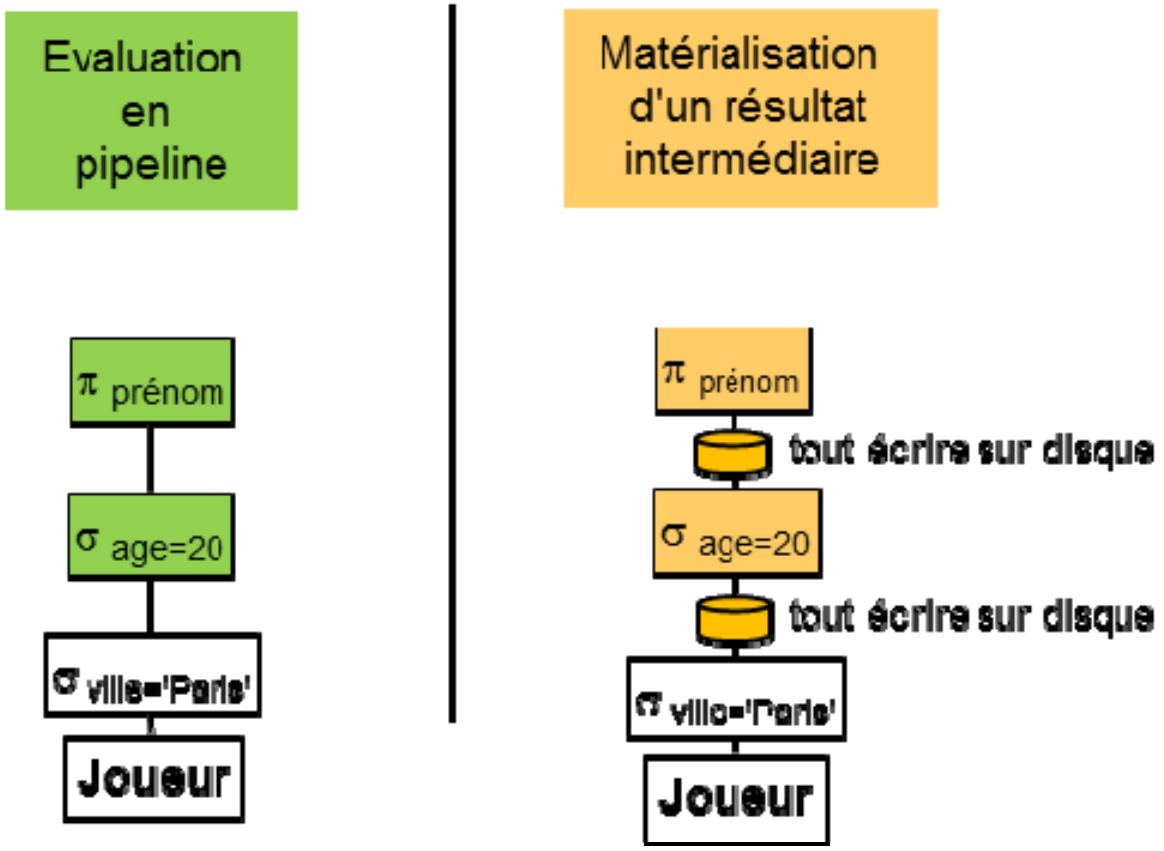
Implémentation des opérateurs

- Il existe plusieurs algorithmes *physiques* possibles pour un opérateur *logique*
  - comprendre les différentes variantes
- Détailler les **étapes** de l'algorithme physique
- Faire la distinction entre
  - Etape impliquant un accès aux données
  - Etape sans accès aux données

Evaluation en pipeline d'une opération

- Une opération est évaluée en **pipeline**
  - Si elle est évaluée **sans** lire aucune donnée stockée dans la base
    - Chaque opérande (données en entrée) doit être le résultat d'une autre opération
    - Opérande ≠ table
    - Opérande **non** matérialisée : jamais écrite temporairement sur disque avant d'évaluer l'opération
  - On « consomme » les opérandes pour « produire » la sortie progressivement
- Pipeline = traitement à la volée
- Avantage : moins couteux car pas de matérialisation

Pipeline vs. matérialisation



- Rmq: pas de pipeline pour la première sélection car accès aux données stockées.

Opération unaire évaluée en pipeline

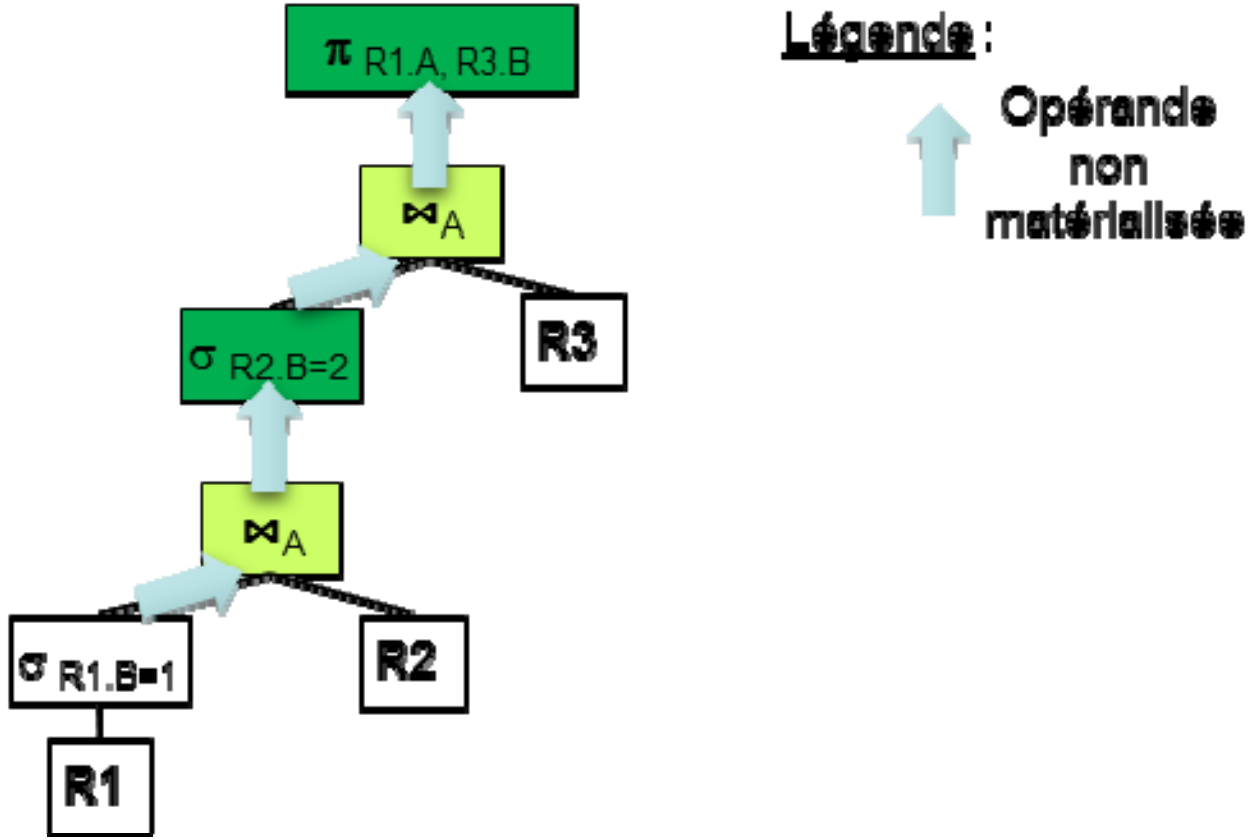
- Une opération unaire a une opérande *e*
  - Exples : $\sigma_{prédicat}(e)$ $\pi_{attributs}(e)$
- Si l'opérande *e* est une expression composée d'au moins une opération, c-à-d, si $e \neq$ table alors
  - coût($\sigma_{prédicat}(e)$) = coût(*e*)
  - coût($\pi_{attributs}(e)$) = coût(*e*)

Opération binaire évaluée en pipeline

- Deux branches en pipeline
  - Union avec doublons
    - opération non relationnelle: UNION ALL en SQL
  - Fusion de listes déjà triées
- Une branche en pipeline
  - Jointure entre une "petite" et une "grande" relation
- Coût d'une opération *n-aire* en pipeline

  = somme du coût de ses opérandes

Opération binaire évaluée en pipeline



- "Flux" de nuplets "remontant" sur la branche principale.
- Evaluation en pipeline possible des jointures si R2 et R3 ont préalablement été "chargés" en mémoire.

Evaluation itérative en pipeline

- Une opération en pipeline est **itérative**
  - Parcours itératif des nuplets de l'opérande pour calculer, un par un, les nuplets du résulat.
    - Le 1er nuplet du résultat dépend seulement des *m* premiers éléments de l'opérande
    - Le 2ème nuplet du résultat dépend seulement des *n* éléments suivants de l'opérande
    - … ainsi de suite jusqu'au dernier nuplet du résultat
- Avantage :
  - Chaque opérateur produit son résultat à la demande de son père
  - **Contrôle** du flux des nuplets intermédiaires

Implémention des opérateurs

- Modèle d'Itérateur
  - Interface commune à tous les opérateurs :
    - méthodes open(), nextTuple(), close()
    - nextTuple() invoque récursivement nextTuple() des opérandes
  - Permet une implémentation en pipeline ou avec matérialisation
  - Avantages :
    - Facilite l'exécution d'un plan : le plan est lui-même un itérateur
    - Permet de calculer progressivement le résultat de manière interactive
      - latence réduite pour produire les n premiers tuples du résultat
- Génération dynamique de code
  - Avantages :
    - optimisation tardive avec information plus récente sur les ressources disponibles (mémoire)
    - pas d'appels imbriqués de méthodes nextTuple()
  - Inconvénient: temps pour compiler la requête

Algorithmes des opérateurs relationnels

- Diapos suivantes
  - Parcours séquentiel
  - Tri
  - Sélection
  - Projection
  - Jointure
  - ...

Parcours séquentiel d'une table

- Requête:
  - Select * from R

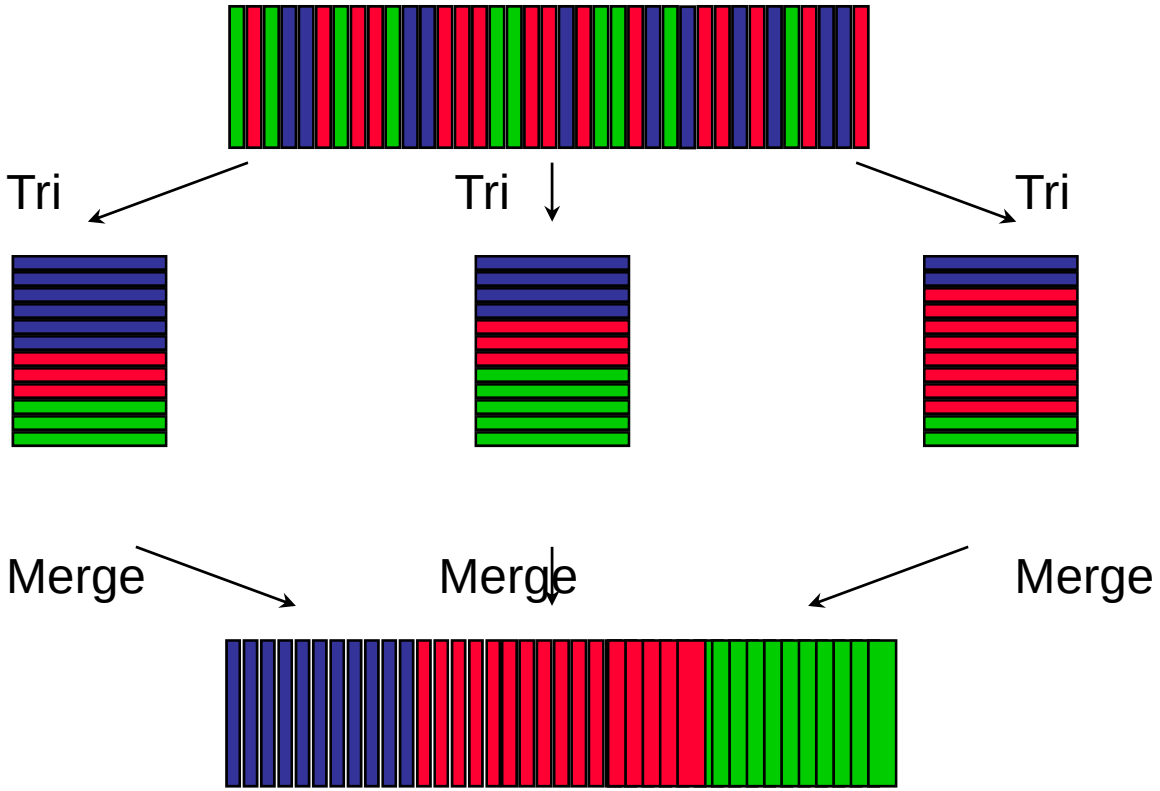
- *R* stockée dans *page(R)* pages du disque
  - Taille d'une page en octets : T_{page}
  - Nombre de nuplets de R dans une page :
    - T_{page} / largeur(R)
  - page(R) = card(R) / (T_{page} / largeur(R))

- Parcours séquentiel : *Table Access FULL*
  - Coût(R) = page(R) . *c*
    - avec *c* < 1 si les pages à lire sont contigües (exple c=0,27 en TME)
    - sinon *c* = 1

Tri externe

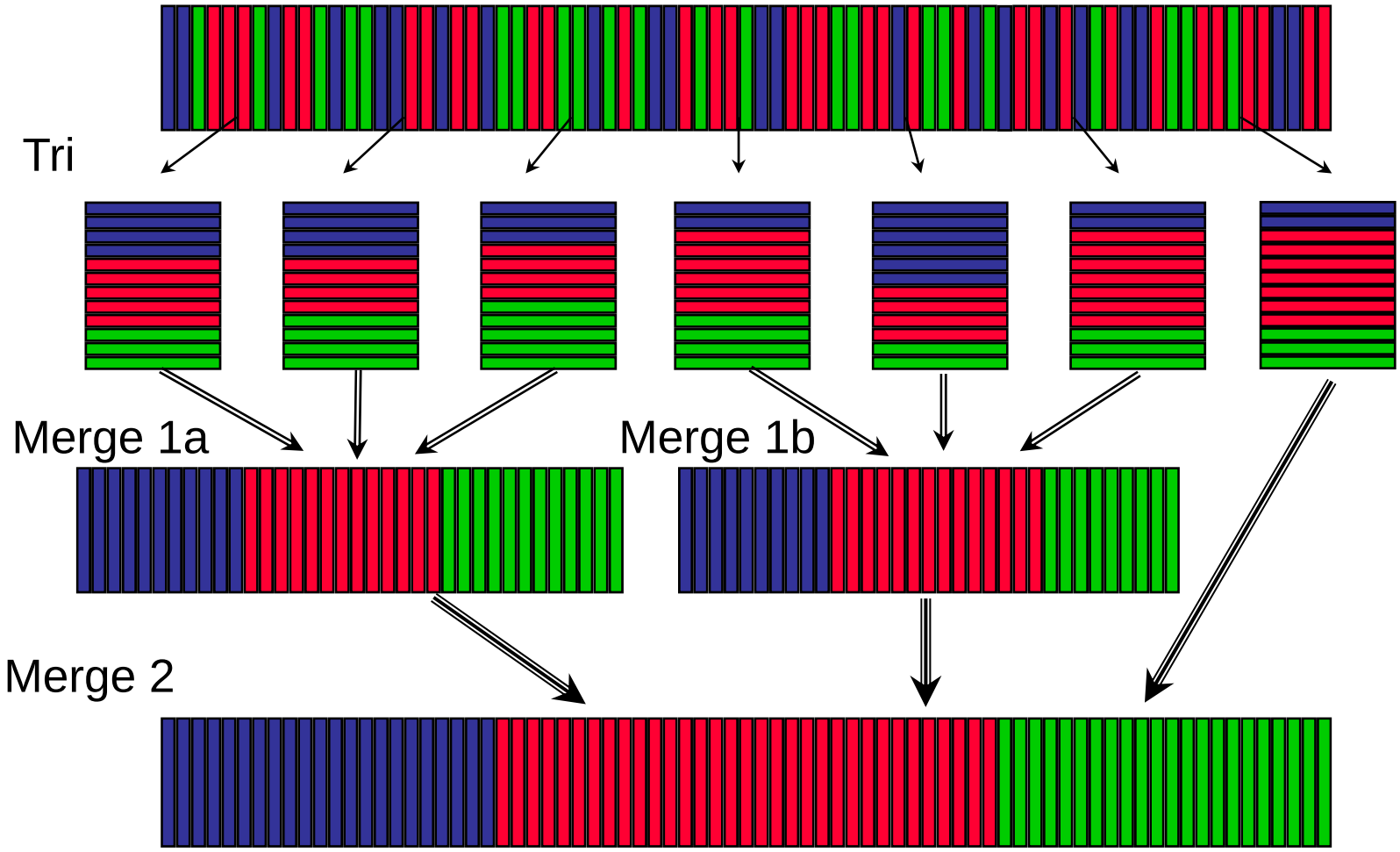
- Hypothèse : k pages tiennent en mémoire
- Algorithme en **s** étapes : tri de blocs puis fusions de blocs
- Etape n°1 : tri
  - Lire R pour créer des paquets triés de k pages chacun
  - Nombre de paquets obtenus : page(R)/k
  - Coût de l'étape de tri (lecture + matérialisation) : 2.page(R)
- Puis étapes n° 2, 3, ..., *s* : fusion
  - Fusion
    - Charger la première page de k paquets et les fusionner
    - Dès qu'une page est vide, charger la suivante du même paquet
    - Dès que les k premiers paquets ont été fusionné, fusionner les k paquets suivants
    - On obtient des paquets triés de taille k^2 pages
    - Coût d'une étape : lire et matérialiser toutes les données : 2.page(R)
    - Nombre de paquets : page(R)/k^2
  - Continuer jusqu'à obtenir **un seul** paquet, on a donc :
    - **page(R) / k^s ≤ 1**

Tri externe : Fusion en 1 seule étape



Ref: CS460 Vassilis.C

Fusion en plusieurs étapes



Ref: CS460 Vassilis.C

Tri externe (suite)

- Nombre d'étape s tq : $k^s \geq$ page(R) :
  $s = \lceil \log_k(\text{page}(R)) \rceil$

- Coût total des s étapes :
  - Lorsque le résultat final du tri est matérialisé
  - Coût(tri(R)) = 2 . page(R) . S

- Si on ne matérialise **pas** le résultat de la dernière étape
  - Dernière étape = seulement lire R
    - Exple : R trié sera affiché ou transmis à une autre opération.
  - Coût(tri(R)) = 2 . page(R) . (s - 1) + page(R)

- Si E est une expression composée (E ≠ table)
  - Tri fait en pipeline : la première lecture de E consiste à évaluer E et le stocker. Le dernier résultat n'est pas matérialisé.
    - Coût(tri(E)) = [coût(E) + page(E)] + [2.page(E).(s-2)] + [page (E)]
  - ⇔ Coût(tri(E)) = coût(E) + 2 . page(E) . (s - 1)

Sélection : σ

- Sélection : $\sigma_{p(A)}$ (...)
  - avec p(A) est un prédicat dépendant de l'attribut A

- Si E est une expression composée
  - Coût($\sigma_{p(A)}$ (E)) = coût(E)

- Si T est une table et l'attribut A n'est pas indexé
  - Coût($\sigma_{p(A)}$ (T)) = *page(T)*

Sélection par index **non** plaçant: : σ_{NP}

- Soit *IdxA* l'index non plaçant sur R.A, et *p(A)* le prédicat de sélection

```
foreach rowid i ∈ IdxA.getRowIds( p(A) ) do
        r = R.getTuple(i)
        add r in result
```

1. Traverser l'index : **`getRowIds( predicat )`**
   - Atteindre la feuille v de l'index (prédicat A=v) et atteindre les feuilles consécutives (prédicat d'inégalité)
     - C_{index}= 0 si l'index tient en mémoire
     - Sinon C_{index} = hauteur de l'arbre + (nombre de feuilles atteintes - 1)
     - Avec Oracle : Index Range Scan ou Index Unique Scan (si A est unique)
   - Lire les **`rowid`** des nuplets
     - $C_{rowid} = \lceil card(\sigma_{p(A)}(R)) / \text{nombre de rowId par page} \rceil$
     - C_{rowid} = 0 si A est unique car son rowId est stocké dans la feuille
2. **Lire une page par rowId : `getTuple( rowID )`**
   - Oracle: Table Access By Rowid

- Coût($\sigma_{NP\ p(A)}(R)$) = C_{rowid} + $card(\sigma_{p(A)}(R)) \cdot 1$

Sélection par index **plaçant** : σ_P

- Soit *IdxA* l'index plaçant sur R.A, et *p(A)* le prédicat de sélection

```
foreach page P_R ∈ IdxA.getPages( p(A) ) do
  foreach tuple r ∈ P_R do
      add r in result
```
on suppose que tous les r de Pr satisfont p(A)

1. Traverser l'index pour obtenir l'adresse de la première page indexée
   - C_{index}= 0 l'index tient en mémoire ou pour le hachage linéaire
   - Sinon
     - C_{index}= 1 pour le hachage extensible
     - C_{index}= hauteur de l'arbre – 1 pour un arbre B+
2. Lire les pages "pleines" de tuples du résultat = **lire une fraction de la table**

- Coût ($\sigma_{P\ p(A)}$ (*R*)) = $\lceil$ page(R) * **SF(p(A))** $\rceil$
  - Rappel : SF(p(A)) est le facteur de sélectivité du prédicat p(A)

Sélections complexes

- Sélections complexes sur une table contenant plusieurs prédicats
- Lorsque plusieurs attributs sont indexés séparément
  - Conjonction (AND)
    - Intersection d'adresses de nuplets (rowid)
    - Vecteur binaire puis ET logique
  - Disjonction (OR)
    - Union d'adresses de nuplets
    - Vecteur binaire puis OU logique

Projection : π

- Projection sans doublons : $\pi_{Attributs}(R)$
  - R est une relation
  - Correspond au "Select distinct Attributs"
  - si $\pi_{Attr}(R)$ tient en mémoire ou si R est sans doublons
    - Coût($\pi_{Attr}(R)$) = coût(R)
  - si $\pi_{Attr}(R)$ ne tient **pas** en mémoire. Deux possibilités :
    - En triant
      - Lire R pour matérialiser R trié selon *Attributs*, puis lire R trié
    - OU en hachant
      - Lire R et la hacher sur disque, insérer uniquement si nouvelle valeur pour « Attributs » puis lire chaque paquet
- Projection SQL avec doublons : Select Attributs
  - Opération **non** relationnelle
  - Coût($proj_{Attr}(R)$) = coût(R)

Jointures: ⋈

- Diapos suivantes :
- Boucles imbriquées
  - simple
  - par bloc
  - avec index
- Par hachage
- Par Tri fusion
- ...

Boucles imbriquées : $\bowtie$

- On suppose que S est une table
- Jointure notée R $\bowtie_{R.a=S.a}$ S
  - Itérer sur les nuplets résultant de R

```
foreach tuple r ∈ R  do
  foreach tuple s ∈ S do
      if r.a=s.a then add <r,s> in result
```

- Avantage :
  - Algo général, permet d' évaluer tout prédicat de jointure
  - Exple (théta jointure) : R $\bowtie_{R.a > S.a}$ S
- Inconvénient :
  - Relire S pour chaque tuple de R
- Amélioration :
  - Relire S pour chaque **partie** de R : voir diapo suivante

Boucles imbriquées par blocs : ⋈

- On suppose que M+2 pages de R tiennent en mémoire
  - On peut "charger" M pages de R en mémoire
- Possibilité de **réduire** le nombre d'accès à S
  - Itération principale sur R par blocs de M pages
  - Puis joindre chaque nuplet de S avec le bloc courant
- Algo :

```
foreach Br of R do
  foreach tuple s ∈ S do
    foreach tuple r ∈ Br,
      if r.a=s.a then add <r,s> in result
```

- Coût(R $\bowtie_{A.a=B.a}$ S) = coût(R) + page(R)/M . page(S)
  - Rmq, en TD: M = 1 page

Boucles imbriquées avec matérialisation : $\bowtie_{Mat}$

- Sert lorsque S est une sous-expression : S ≠ table
  - Matérialiser S **avant** de calculer la jointure
- Jointure notée R $\bowtie_{Mat\ R.a=S.a}$ S
  - Etape préliminaire
    - Evaluer S : coût(S)
    - Stocker son résultat dans S_{Mat} Coût pour écrire S_{Mat} : page(S)
  - Jointure par boucles imbriquées entre R et S_{Mat}

Coût(R $\bowtie_{Mat\ R.a=S.a}$ S) = coût(S) + page(S) +coût(R $\bowtie_{R.a=S.a}$ S)

  - Exple pour M=1
    - Coût(R $\bowtie_{Mat\ R.a=S.a}$ S) = coût(S) + page(S) + coût(R) + page(R) . page(*S*)

Jointure par boucles avec index (1)

- Jointure
  - par boucles imbriquées et
  - index sur l'attribut *a* de S
    - Évite de parcourir S entièrement pour chaque bloc de R

$$\text{Coût}(R \bowtie_{\text{Ind R.a=S.a}} S) = \text{coût}(R) + \text{card}(R) \,.\, \text{coût}(\sigma_{a=v}(S))$$

- Cas particulier de la jointure sur clé
  - si l'attribut *a* est une clé de S, alors $\text{coût}(\sigma_{a=v}(S)) = 1$

- Cas général :
  - le terme $\textit{coût}(\sigma_{a=v}(S))$ dépend du type d'index

Jointure par boucles avec index (2)

- Jointure avec index **non plaçant** *IdxSa* sur S.a
  - En supposant que l'index tient en mémoire : $C_{index} = 0$

```
foreach tuple r ∈ R do
     foreach rowid i ∈ IdxSa.getRowIds(r.a) do
        s = S.getTuple(i)
        add <r,s> in result
```

  - coût($\sigma_{a=v}(S)$) = C_{rowid} + card($\sigma_{a=v}(S)$)
  - donc coût($R \bowtie_{Ind\ R.a=S.a} S$) = coût(R) + card(R) . [C_{rowid} + card($\sigma_{a=v}(S)$)]

- Jointure avec index **plaçant** sur S.a
  - En supposant que tous les *s* de *Ps* satisfont s.a = p.a

```
foreach tuple r ∈ R do
     foreach page Ps ∈ IdxSa.getPages(r.a) do
        foreach tuple s ∈ Ps do
            add <r,s> in result
```

  - coût($\sigma_{a=v}(S)$) = $\lceil$ page(S) * SF(a=v) $\rceil$
  - donc coût($R \bowtie_{Ind\ R.a=S.a} S$) = coût(R) + card(R) . $\lceil$ page(S) * SF(a=v) $\rceil$

Jointure par tri puis fusion (1)

- Trier *R* et *S* sur l'attribut de jointure : Sort(join)
  - voir tri externe
- Fusionner les relations triées : Merge join
- Amélioration pour réduire le nombre de lectures et écritures
  - Laisser R triée en plusieurs morceaux sans les fusionner. Idem pour S.
  - On peut fusionner directement les 2 relations dès que le nombre de paquets restant dans les 2 relations est inférieur à k
    - Trier R en P_R paquets, et trier S en P_S paquets, tq : $P_R + P_S < k$
  - Fusion des Pr paquets de R avec les Ps paquets de S en **une** seule étape
- Exemple : jointure entre 2 tables R et S
  - Page(R)=6000, page(S)=3000, k=100 pages en mémoire
    - Coût du tri = 2. (page(R) + page(S)
    - On obtient 60 blocs de R et 30 blocs de S soit un total de 90 blocs
    - Il suffit de lire les blocs pour les fusionner : coût = page(R) + page(S)
    - Bilan: coût($R \bowtie_{TF\ R.a=S.a} S$) = 3 (page(R) + page(S))

Jointure par fusion (2) algorithme détaillé

- Principe : itérer progressivement sur R et S
- S'il y a plusieurs tuples de r (ou s de S) pour une valeur de a, produire toutes les paires (r,s)
  - R={(1,b) (7,z) (7,b) (7,c)} S = {(6,e) (7,e) (7,a) (9,i)}
  - Le résultat contient 6 tuples: 7ze 7be 7ce 7za 7ba 7ca
- Algo :
  - Initialiser r et s
    - r ← premier tuple de R, s← premier tuple de S
  - tant que r et s existent
    - si r.A = s.A alors
      - tmpR← r
      - continuer d'itérer sur R pour ajouter dans tmpR les tuples tq R.A = S.A
      - tant que s existe et s.A = r.A
        - » Pour chaque t dans tmpR, ajouter (s,t) dans le résultat
        - » S← suivant(S)
    - si r.A < s.A alors r ← suivant(R) sinon si s.A < r.A suivant(S)

Jointure par hachage (1)

- Hypothèse : R est plus grande que S page(R) ≥ page(S)
- Principe : traitement en 2 étapes
  1) Lire S pour la hacher selon la clé de jointure
  2) Itérer sur les tuples de R et jointure sur clé
- S tient en mémoire créer une hashmap en mémoire

```
1) init: Map : hashmap<A, List<Tuple>>
   foreach tuple s ∈ S do
       -- créer la liste L et l'associer à S.a si elle n'existe pas
       List L ← Map.getOrCreate(S.a)
       add s in L
2) foreach tuple r ∈ R do
     L ← Map.get(R.a)
     foreach tuple s ∈ L add (r,s) in result
```

Coût(R $\bowtie_{H\ R.a=S.a}$ S) = coût(S) + coût(R)

Jointure par hachage (2)

- Hachage externe si S ne tient **pas** en mémoire.
  - Hacher S sur disque
    - les pages deviennent des 'paquets' répertoriés dans une table de hachage T.
    - T[h(v)] → paquet contenant tous les tuples de S tq S.a=v
  - Création de T: ajouter le coût pour lire et compléter un paquet
    - (1 lecture + 1 écriture) par tuple de S, donc ajouter **2* card(S)**
  - Lecture de T pendant l'itération sur R: ajouter le coût d'accès à T
    - une lecture par tuple de R, donc ajouter **card(R)**

$$\text{Coût}(R \bowtie_{\text{HExt } R.a=S.a} S) = \text{coût}(S) + \mathbf{2 * card(S)} + \text{coût}(R) + \mathbf{card(R)}$$

Jointure n-aire

- Evaluer n jointure binaires
- Parcourir l'arbre de jointure en profondeur d'abord
- Evaluer les opérations en remontant
  - depuis l'opération la plus à gauche
- Exple: $(R \bowtie_a S) \bowtie_b T$ et jointure par boucles imbriquées

```
- foreach tuple r ∈ R  do
    • foreach tuple s ∈ S do
        - if r.a=s.a then
            » foreach tuple t ∈ T do
                » if r.b=t.b then
                    » add <r,s,t> in result
```

- Exple: $(R \bowtie_a S) \bowtie_b T$ et jointure par hachage

```
- foreach tuple r ∈ R  do
    • add (a,r) in hashMap M1
- foreach tuple s ∈ S do
    • Lr = M1.get(s.a)
    • foreach r ∈ Lr
        - add (b ,(r,s)) in hashMap M2
- foreach tuple t ∈ T do
    • Lrs = M2.get(t.b)
    • foreach (r,s) ∈ Lrs
        - add <r,s,t> in result
```

Order by

- Voir tri externe

Autres opérations

- Group By avec agrégation
  - Hachage ou tri

- a IN (*sous-requête*)
  - Evaluer la sous-requête pour chaque valeur de a
    - Algorithme général de type "boucles imbriquées"
  - Lorsque la sous-requête ne dépend **pas** de la requête principale
    - Evaluer une (semi) jointure : Requête principale $\ltimes$ sous-requête
      - Charger la sous requête en mémoire : voir algo de type $\bowtie_{H}$

      Ou
      - Matérialiser la sous-requête : voir algo de type $\bowtie_{Mat}$

Perspectives

- Nombreuses autres variantes pour implémenter les opérateurs relationnels
- Evaluation en parallèle d'un opérateur
- Tri externe en parallèle
  - Sort benchmark : voir le site sortbenchmark.org
    - 2009: 500 GO/mn, 2013: 1,4 TO/mn,
    - 2014: 4,3 TO/mn http://sortbenchmark.org/ApacheSpark2014.pdf
    - 2015: 16 TO/mn
    - 2016: **44** TO/mn http://sortbenchmark.org/TencentSort2016.pdf
    - Coût 'vert' du tri:
      - » prix $ par TO/mn, énergie consommée par TO/mn