Stockage et Accès aux Méga-données

Introduction à la gestion des méga-données

3V des méga-données : Volume, Vélocité, Variété
Défis : Passage à échelle, distribution des données, distribution des traitements
Cloud pour le big data : location des services informatiques (qualité, élasticité)

- **Architecture** : Infrastructure (DataCeneter : Iaas) < Plateforme (Services API : Paas) < Logiciel (application finales : Sais).

Index non plaçant, composés, couvrants, Arbre B+

Objectifs des SGBD

- **Cohérence intégrée des données :** cohérence de transaction et intégrité, partage, **performance d'accès,** sécurité.
- **Indépendance des données :** logique (organisation conceptuelle), physique (stockage).

Stockage

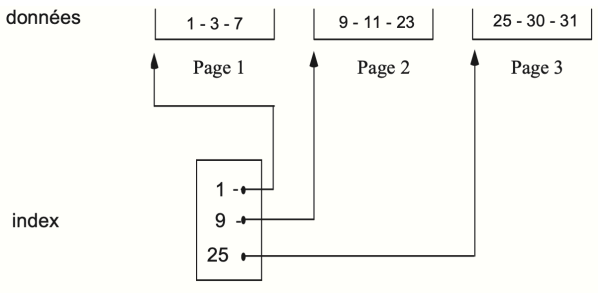
- Données stockées sur un support persistant.
- **Page :** unité de stockage (taille fixe, 8Ko le plus souvent).
- **Opérations : lecture et écriture des pages**
- **Enregistrement :** donnée stockée, ligne d'une table
- **ROWID :** adresse d'un enregistrement = (nomFichier, numéro de page, position)
- **Coût d'une opération :** durée, temps de lecture et d'écriture des pages, dépend de la méthode d'accès

Organisation séquentielle

- **Non trié :** + mise à jour - parcours complet pour toutes les requêtes
- **Trié** : - maintenance difficile + parcours raccourci car données triées

Organisation indexée

- **Clé**
  - **Clé de l'index :** 1 ou plusieurs attributs
  - Stockage en fonction de la clé : regroupements, stockage contiguë des enregistrements ayant la même clé
- **Index plaçant :** définition de l'organisation à la création de la table
  - **Tri** : *create table … organization index*
  - **Regroupement** : *create cluster*
  - Pas plus d'un index plaçant par table
- **Index plaçant non dense :** index occupant moins de place avec des données triées.
- **Index dense** : contient toutes les valeurs de la clé



Index plaçant non dense

- **Index non plaçant (secondaire) :** données stockées sans être triées ou données triées selon un autre attribut que l'index primaire.
  - Create index <nom> on <table>(<attrs>) : create index indexAge on Personne(age)
  - **Entrée d'index : clé => ROWID**
    - **Unique :** attribut indexé satisfait une contrainte d'unicité : clé primaire, unique, …
    - **(Clé, ROWID) :** un seul enregistrement par valeurs
    - **(Clé, liste des ROWID) :** plusieurs enregistrements par valeur
- **Accès par index**
  - Évaluation de sélection : égalité, intervalle, inégalité, comparaison préfixe (*like 'ch%'l)*
  - Table de hachage : égalité uniquement
- **Index couvrant une requête :** possible d'évaluer la requête sans lire les données => tous les attributs de la requêtes sont indexés
  - Index plaçant non dense => jamais couvrant, parce qu'il ne contient pas toutes les valeurs de l'attribut indexé
- **Index composé :** clé : concaténation des attributs
  - **Sélection préfixe** : (A1), (A1, A2), … (A1…An) : parcours latéral de toutes les feuilles de l'index
- **Directives des index**
  - **index**(table index) : select /*+ index(Personne IndexAge) */ * from Personne where age >= 18
  - **no_index**(table index), **index_combine**(table index1 index2)

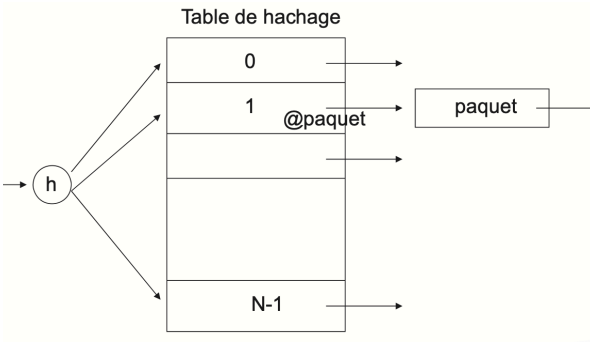
Arbre B+ : Index hiérarchisé

- **Efficacité :** arbre peu profond, équilibré, suffisamment compact
- **Coût d'accès :** proportionnel à la longueur du chemin : nombre de noeuds lus/écrits
- **Ordre d :** capacité d'un noeud de l'arbre
- **Noeuds :**
  - **Racine :** point de départ d'une recherche : $1 \leq n \leq 2d$
  - **Noeud intermédiaire + feuilles :** $d \leq n \leq 2d$
  - **Feuilles :** toutes les clés de l'index pour lesquels il existe un enregistrement
- **Degré sortant :** la racine et les noeuds intermédiaires -> n + 1.
- Nombre max de clés dans les feuilles : $2d(2d+1)^{(p-1)}$
- **Chainage des feuilles**
  - **+** Requêtes avec intervalles
  - **Chainage double :** requêtes avec inégalité
  - **Avantages :** parcours d'un seul chemin
- **Insertion :**
  - Insérer si y'a de la place. Si pas de place -> éclatement
  - Éclatement de feuille -> (d+1) et d
  - Éclatement de noeud intermédiaire : -> d, 1 inséré dans le parent, d dans le nouveau noeud
- **Suppression :**

  - Si feuille pleine, ok. Sinon, redistribution avec feuille du même parent.
  - Si redistribution impossible -> fusion de feuilles. -> application récursive.
- **Avantages :** régularité, lecture séquentielle rapide, accès rapide.
- **Inconvénients :** suppression -> trous, index non plaçant -> lecture des enregistrements non continus, taille d'index parfois importante
- **Arbre B+ distribué**

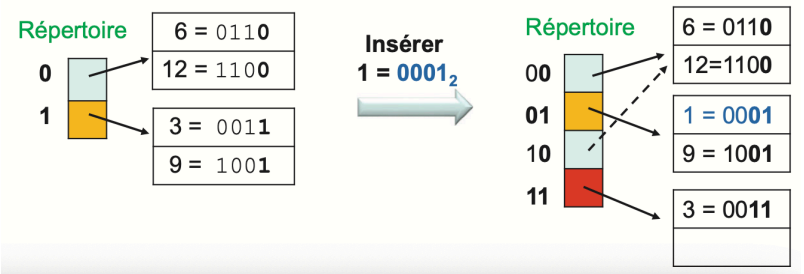
Index par hachage

- **Hachage statique :**
  - Fichier de taille fixe -> obtenir une distribution uniforme des données.
  - **Paquets :** groupe d'articles
  - **Fonction de hachage sur la clé :** donne l'adresse de l'article
    - Conversion en nombre entier, modulo, pliage de la clé, …
    - Bien choisir la fonction de hachage, sinon saturation
  - **Indirection : table de hachage :** H(k) -> position d'une cellule -> adresse d'un paquet. + Suppression d'un paquet
  - **Avantages :** recherche égalité, bonne quand les données évoluent peu
  - **Extension :** autorisation des débordements
  - **Techniques de débordement**
    - **Adressage ouvert :** si paquet plein -> article placé dans le paquet suivant -> mémoriser les paquets avec débordement
    - **Chainage : paquet logique :** chainage d'un paquet de débordement à un paquet plein
      - Trop de débordement -> plus d'intérêt
    - **Rehachage :** paquet plein -> nouvelle fonction de hachage



Hachage statique

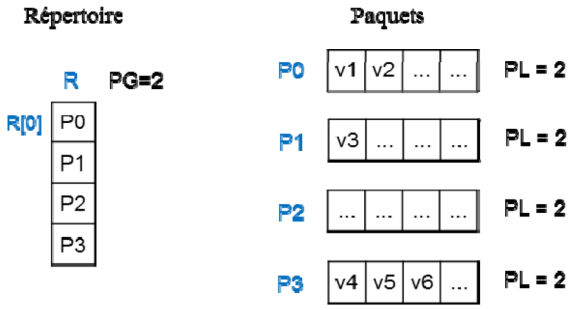
- **Hachage dynamique**
  - Fichier grandissant progressivement
  - **Techniques : hachage extensible, hachage linéaire**
- **Hachage extensible**
  - Ajout de niveau d'indirection. Jamais de débordement
  - **Répertoire :** arbre à préfixe (hiérarchie basée sur le suffixe), liste de pointeurs vers les paquets
  - **Profondeur globale** : Retrouver une entrée. $h_{PG}(v) = v \,\%\, 2^{PG}$ -> lecture de la case $h_{PG}(v)$.



Hachage dynamique

- Pas de répartition de toutes les valeurs de la table quand on l'agrandit.

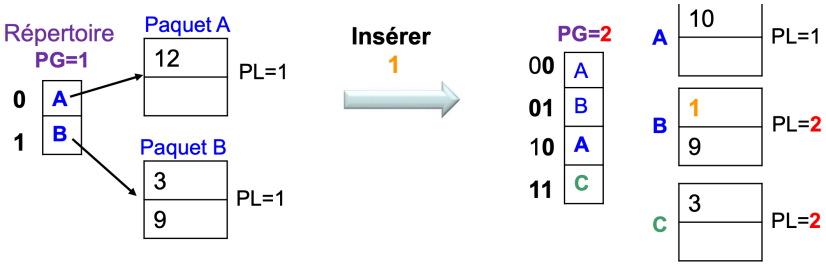
- **Profondeur locale** : Indique pour chaque paquet s'il peut éclater rapidement, sans agrandir le répertoire.
- **Notations :**
  - **Répertoire :** $R[P_0, P_1, \dots, P_k]$
  - P_i **:** nom du paquet
  - $PG = pg$: profondeur globale -> k cases : $k = 2^{pg}$
  - **Paquet :** $P_i(v_j, \dots)$
  - v_j **:** valeur contenue par le paquet
  - $PL = pl$: profondeur locale



Hachage extensible

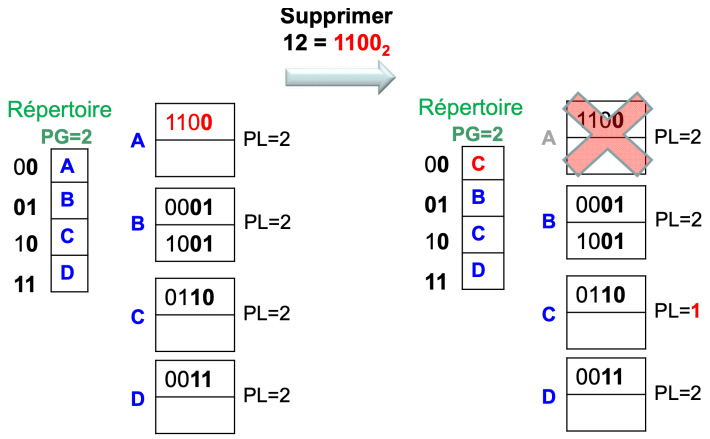
- **Initialisation :** N : nombre de valeurs, p : capacité d'un paquet
  - $k = 2^{PG} \geq N/p$; $PL = PG$

- **Insertion**
  - **Paquet non plein :** insertion
  - **Paquet plein et** $PL_i < PG$**:** (1) éclater Pi, (2) créer Pj, ajouter l'adresse de Pj dans Pi, (3) incrémenter les valeurs des PL de Pi et Pj, (4) répartir les valeurs entre Pi et Pj
  - **Paquet plein et** $PL_i = PG$**:** (1) recopier les k premières cases dans les k nouvelles, (2) incrémenter PG, (3) appliquer le cas précédent



Hachage extensible - Insertion

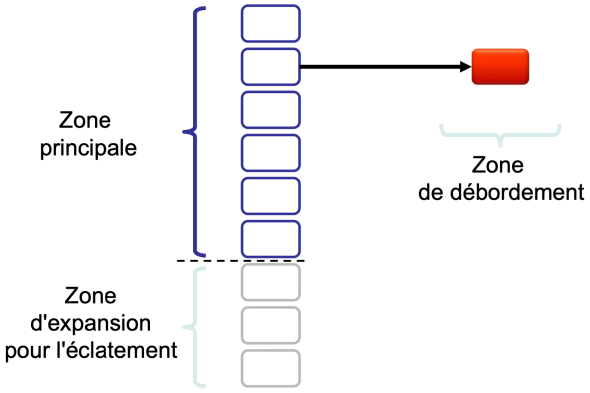
- **Suppression :** Si le paquet devient vide et $PL_i < PG$, on le laisse vide. Sinon :
  - Fusionner Pi et Pj (suffixe de longueur PG-1 en commun en base 2)
    - PG = 3 : R[0] et R[4], R[1] et R[5], …
  - Supprimer Pi, le décrocher du répertoire, mettre Pj à la place avec PL -= 1
  - Si pour tous les paquets PL < PG : diviser le répertoire par 2. PG -= 1.



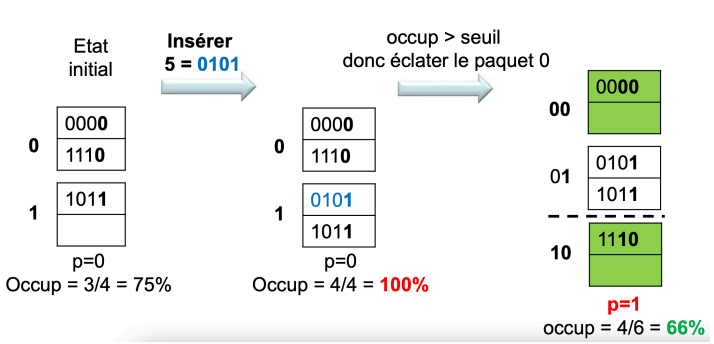
Hachage extensible - Suppression

- **Avantages :** accès en un seul bloc <-> répertoire tient en mémoire, modification progressive de la table de hachage
- **Inconvénients :** peut ne plus tenir en mémoire, peu de fichiers en mémoire -> répertoire inutilement gros

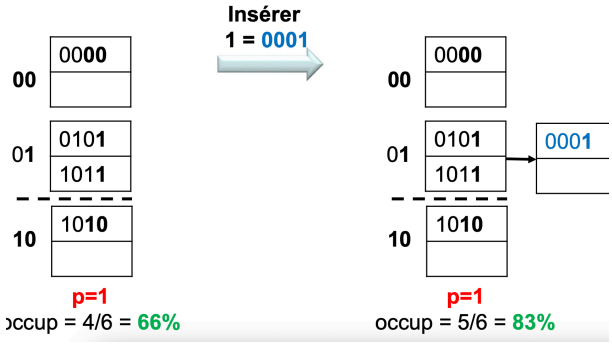
- **Hachage linéaire**
  - Garantie : Nombre moyen d'enregistrements par page < seuil.
    - Taux d'occupation moyen d'un paquet < 80%
    - Ajout des paquets au fur et à mesure, éclatement dans l'ordre
      - Marquer le prochain paquet à éclater : p
      - Eclater par série de N éclatements, puis 2N, …
  - + Pas besoin de répertoire, + plus rapide que le hachage extensible.
  - - Débordement temporaire, - débordement permanent possible
  - **Initialisation :** N paquets (0, … N-1)
  - **Fonctions de hachages :** N, 2N, 4N, …
    - $h_0(x) = x \,\%\, N$ et $h_i(x) = x \,\%\, (2^i N)$
  - **Taux d'occupation : V/C**
    - V: nombre total de valeurs dans tous les paquets des 3 zones
    - C: nombre de cases dans les paquets à accès direct : zones principale et expansion
  - **Insertion :** exemple
    - N = 2 seuil = 90% h0(x) = x%2 h1(x) = x%4 p = 0
    - Les paquets éclatés et les nouveaux paquets utilisent la fonction de hachage de niveau supérieur.
    - Les paquets qui n'ont pas éclaté conservent la fonction initiale.



Zones - Hachage linéaire



Insertion - Hachage linéaire



Insertion - Hachage linéaire (2)

- **Accès :**
  - Calculer $k = h_i(v)$ et $k' = h_{i+1}(v)$
  - Si $k >= p$ -> lire k, sinon lire k'.

Opérations relationnels (Cours 3 & 4)

Notions (slides 5)
- **Unité de mesure :** la page.
- **Opérations :** sélection, projection, jointure, tri
- Coût E/S >> Coût CPU : lire sur disque coûte plus cher que accéder en mémoire.

Pipeline et Matérialisation (slides 6 - 12)
- **Evaluation en pipeline :** évaluation **sans lecture des données** dans la base.
  - **Opérande** = données en entrée = résultat d'autres opérations ≠ table.
  - **Matérialisation :** écriture temporairement sur disque pour évaluation.
  - Pipeline -> opérande non matérialisée
  - **Sortie progressive :** produite par *consommation* des opérandes = traitement à la volée
  - **Avantages :** pas de matérialisation -> moins coûteux
- **Opération unaire : op =** $\sigma_{pred}(e)$, $\pi_{attrs}(e)$ **:** si $e \neq table : cout(op) = cout(e)$
- **Opération binaire :**
  - **Deux branches en pipeline :** Union avec doublons, fusion de listes triées
  - **Une branche en pipeline :** jointure entre une petite et une grande relation
- **Opération n-aire :** coût = somme du coût des opérandes
- **Evaluation itérative en pipeline :**
  - **Pipeline :** opération itérative, calcul du résultat tuple par tuple
  - **Avantages :**
    - chaque opérateur produit son résultat à la demande de son père
    - Contrôle du flux des n-uplets intermédiaires.

Implémentation des opérateurs (slides 13 - 14)

- **Modèle d'itérateur :**
  - **Interface :** *open(), nextTuple(), close()*
  - Implémentation en pipeline ou avec matérialisation
  - **Avantages :** facilitation de l'exécution d'un plan et calcul progressif du résultat de manière interactive
- **Génération dynamique du code :**
  - + : Optimisation tardive avec informations plus récentes sur les ressources disponibles
  - - : temps de compilation de requête
- **Requête SQL** -> plan = arbre -> résultat : itération depuis la racine de l'arbre

Algorithmes des opérateurs relationnels

- **Parcours séquentiel d'une table** (slide 16)
  - **Select * from R :** TABLE ACCESS FULL
  - **Nb de pages** : $page(R)$; **Taille d'une page** en o : T_{page} ;
  - **Nb de nuplets dans une page** : $\frac{T_{page}}{largeur(R)}$
  - **Cardinal de R :** $card(R) = page(R) * (\frac{T_{page}}{largeur(R)})$
  - **Coût de R :** $cout(R) = c * page(R)$
    - c < 1 (ex : c = 0.27) si les pages à lire sont contingues ; c = 1 sinon

- **Sélection** (slides 17 - 21)
  - $\sigma_{p(A)}(\dots)$ **:** prédicat sur l'attribut A
  - E : expression composée -> **coût(op) = coût(E)**
  - T : **table** et l'attribut A **non indexé** -> **coût(op) = page(T)**
  - **Sélection par index non plaçant :**
    - *for* ***rowid*** *in* ***IdxA.getRowIds(p(A)) :*** *result.append(* ***R.getTuple(rowid)*** *)*
    - **Index :**
      - **Traverser l'index : (1a) : atteindre une feuille de l'index**
        - $C_{index} = 0$ si index en mémoire
        - $C_{index} = hauteur - 1$
      - **Lire les rowids :** (1b)
        - **Index unique scan :** $C_{rowid} = 0$ si les feuilles de l'index contiennent les rowids
        - **Index range scan :** $C_{rowid} = n * \frac{card(\sigma_{p(A)})}{card(R)}$ avec n le nombre de pages contenant les rowids
    - **Lire les tuples associés aux rowids :** (2)
      - **getTuple(rowID) :** voir **Table Access by Rowid**

- $cout(\sigma_{NPP(A)}(R)) = C_{index} + C_{rowid} + \frac{card(\sigma_{P(A)}(R)) * CF}{card(R)}$
- **CF : Clustering factor**
  - Indique dans quelle mesure l'arrangement des tuples dans une page dépend de l'attribut indexé
  - Déterminé à partir des entrées *consécutives* de l'index
    - **CF faible :** tuples insérés avec des valeurs croissantes de l'attribut indexé
    - **CF élevé :** tuples insérés indépendamment de l'attribut indexé
  - page(R) < CF < card(R)
  - [v1, v2] : plage de valeurs consécutives : N rowids, P pages à lire
    - Nombre moyen de rowid par page : 1 < N/P < nb tuples par page
  - $CF = \frac{card(R)}{\frac{N}{P}}$
- **<u>Sélection par index plaçant</u> :**
  - *for* ***page P*** *in* ***IdxA.getPages(p(A))*** *: for* ***tuple t*** *in* ***P :*** *result.append(* ***t*** *)*
    - On suppose que tous les tuples satisfont p(A)
  - **Traverser l'index :** obtention de la première page indexée
    - $C_{index} = 0$ si l'index tient en mémoire ou <u>hachage linéaire</u>
    - $C_{index} = 1$ sinon et index par <u>hachage extensible</u>
    - $C_{index} = hauteur - 1$ pour les <u>arbres B+</u>
  - **Lire les pages :** lire une fraction de la table
  - $cout(\sigma_{PP(A)}(R)) = SF(p(A)) * page(R) + C_{index}$
  - **SF :** facteur de sélectivité du prédicat
- **<u>Sélections complexes</u> :** plusieurs prédicats
  - **Attributs indexés :**
    - **AND : Intersection des adresses :** vecteur binaire + ET logique
    - **OR : Union des adresses :** vecteur binaire + OU logique

- **<u>Projection</u>** (slides 22)
  - **Sans doublon : $\pi_{attrs}(R)$: select distinct attrs from R**
    - $\pi_{attrs}(R)$ tient en mémoire ou R est sans doublon : **coût(op) = coût(R)**
    - **Sinon :** tri de R selon les attributs ou par hachage sur disque
  - **Avec doublon : coût(op) = coût(R)**

- **<u>Jointure par boucles imbriquées</u>** (slides 23 - 29)
  - **<u>Boucles imbriquées</u> :**
    - $R \bowtie_{R.a=S.a} S$: **S table**
    - *for* ***r*** *in* ***R*** *: for* ***s*** *in* ***S*** *: if* ***r.a = s.a*** *: result.append(* ***(r, s)*** *)*
    - **Avantages** : permet d'évaluer d'autres conditions de jointure
    - **Inconvénients** : relecture de S pour chaque tuple
    - $cout(R \bowtie_{R.a=S.a} S) = cout(R) + page(R) * page(S)$

- **Boucles imbriquées par blocs :**
  - **M + 2** pages de R tiennent en mémoire -> on peut en charger **M**
  - Réduction du nombre d'accès à S -> itération par blocs de M pages de R -> jointure de S avec les blocs
  - *for **bloc Br of R** : for **tuple s** in **S :** for **tuple r** in **Br :** if **r.a = s.a :** result.append(**(r, s)**)*
  - $cout(R \bowtie_{R.a=S.a} S) = cout(R) + \frac{page(R)}{M} page(S)$
- **Boucles imbriquées avec matérialisation :** $\bowtie_{Mat}$
  - S : sous expression et non une table -> matérialiser S **avant** le calcul de la jointure
    - Evaluer S : coût(S)
    - Stockage de S : page(S)
    - Jointure par boucles imbriquées entre R et la matérialisation de S
  - $cout(R \bowtie_{Mat:R.a=S.a} S) = cout(S) + page(S) + cout(R \bowtie_{R.a=S.a} S)$
- **Boucles avec index**
  - Index sur l'attribut a de S
  - $cout(R \bowtie_{Ind:R.a=S.a} S) = cout(R) + card(R) * cout(\sigma_{a=v}(S))$
  - **Cas particulier** : **a clé de S :** $cout(\sigma_{a=v}(S)) = 1$
  - **Cas général** :
    - **Index non plaçant :**
      - for **r** in **R :** for **i** in **IdxSa.getRowIds(r.a) : s = S.getTuple(i) ;** result.append(**(r, s)**)
      - $cout(\sigma_{a=v}(S)) = C_{rowid} + \frac{card(\sigma_{a=v}(S)) * CF}{card(S)}$
    - **Index plaçant :**
      - for **r** in **R :** for **Ps** in **IdxSa.getPages(r.a) :** for **s** in **Ps :** if **r.a = s.a :** result.append(**(r, s)**)
      - $cout(\sigma_{a=v}(S)) = SF(a = v) * page(S)$

- **Jointure par tri-fusion** (slides 30-31)
  - **Tri : Sort join :** tri de R et S sur l'attribut de jointure
    - $cout(R \bowtie_{T:R.a=S.a} S) = 2(page(R) + page(S)$
  - **Fusion : Merge join :** fusionner les résultats triés
    - $cout(R \bowtie_{F:R.a=S.a} S) = page(R) + page(S)$
  - $cout(R \bowtie_{TF:R.a=S.a} S) = 3(page(R) + page(S)$
  - **Améliorations** :
    - Conserver des morceaux triés de R et S sans fusionner.
    - Fusion quand le nombre de paquets restants pour R et S (PR + PS) < k
    - Fusion en une seule étape des paquets

- **Jointure par hachage** (slides 32-34)
  - $page(R) \geq page(S)$

- **Principe :** Hacher S selon la clé (1) Itérer sur R pour joindre sur la clé (2)
- **<u>S tient en mémoire</u> :**
  - **(1)**
    - **map = Hashmap <A, List<Tuple>>**
    - for **s** in **S : L = map.getOrCreate(S.a) ; L.add(s)**
  - **(2)** for **r** in **R : L = map.get(r.a) ;** for **s** In **L :** result.append(**(r, s)**)
  - $cout(R \bowtie_{H:R.a=S.a} S) = cout(R) + cout(S)$
- **<u>Hachage externe</u> : S <u>ne tient pas</u> en mémoire**
  - Algorithme **Grace Hash Join**
  - **Taille de la mémoire :** k+1 pages
  - **Hacher S puis R sur disque** :
    - Répartition récursive des données de S dans k paquets jusqu'à avoir des paquets de moins de k pages
      - S_i contient les tuples tels que $h(S.a) = i$
    - Idem pour R
    - $e = inf(log_k(page(S)))$
    - $cout = 2e(page(R) + page(S))$
  - **Chargement des paquets :** jointure des paquets de S avec ceux de R
    - $cout = page(R) + page(S)$
  - $cout(R \bowtie_{HExt:R.a=S.a} S) = (2e + 1)(page(R) + page(S))$

- **<u>Jointure n-aire</u>** (slides 35)
  - Evaluation de n jointures binaires
  - Parcours de l'arbre de jointure en profondeur d'abord
  - Evaluation des opérations en remontant

- **<u>Tri externe</u>** (slides 37-40)
  - k pages tiennent en mémoire
  - Tri de blocs puis fusions de blocs
  - **Tri :** lire R, créer des paquets de k pages triés : $paquets(R) = \frac{page(R)}{k}$
    - **Coût :** lecture + matérialisation : $2page(R)$
  - **Fusion :**
    - Fusion des paquets par groupes de k : page vide -> nouvelle page du prochain paquet
    - On obtient des paquets triés de taille k^2
    - Coût d'une étape : $2page(R)$
    - Continuer jusqu'à ce que $\frac{page(R)}{k^s} \leq 1$
    - $s = sup(log_k(page(R)))$
  - **Matérialisation :** $cout(tri(R)) = 2s * page(R)$
  - **Sinon :** $cout(tri(R)) = (2(s-1)+1) * page(R)$
  - **E expression :** $cout(tri(E)) = 2(s-1) * page(E) + cout(E)$

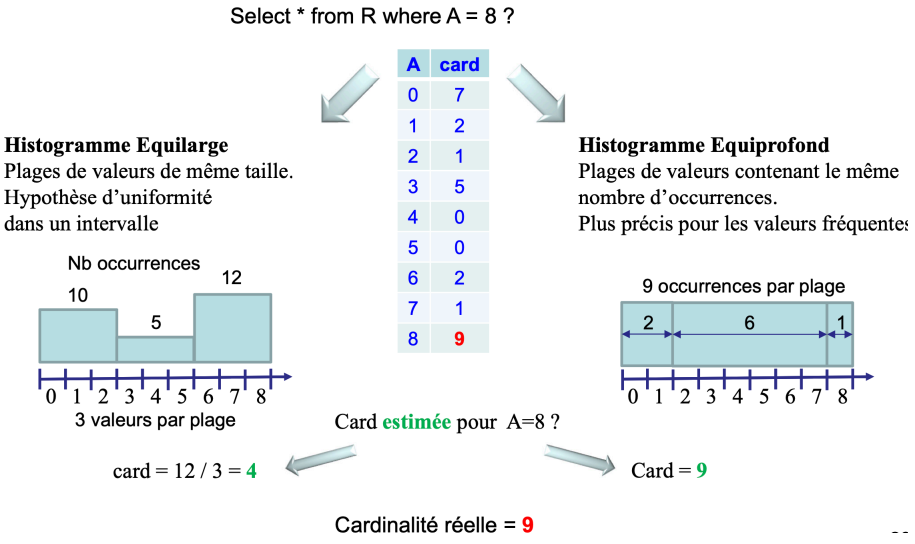
- **Autres opérations** (slides 41)
  - **Group by :** hachage, tri
  - **a IN (sous-requête) :**
    - Jointure par boucles imbriquées
    - Si la sous-requête ne dépend pas de la requête principale : jointure pr hachage ou par matérialisation

---

Traitement et optimisation de requêtes

Optimisation d'une requête

- **Etapes :** Simplification (2), Normalisation (3), Restructurations (4)
- Avant : Analyse (1)
- **Normalisation** : analyse lexicale et syntaxique, mise de la requête sous forme normale (OR -> union, AND -> jointure ou sélection)
- **Simplification** : tautologies ou non, redondance, règles d'intégrité
- **Heuristiques**
  - Opérations manipulant moins de données sont les plus rapides
  - Commencer par traiter les opérations les plus sélectives (projection, sélection) avant les jointures
- **Estimation du coût** : coût I/O + coût CPU
- **Histogrammes**



Histogramme

Facteur de sélectivité

- **Egalité** : $SF = \frac{|valeurs|}{D(R,A)}$ $D(R,A)$: nombre de valeurs distinctes de A
- **Inégalité** : $SF = \frac{|valeurs|}{max(A) - min(A)}$

    - $A < v : |valeurs| = v - min(A)$
    - $A > v : |valeurs| = max(A) - v$
    - $v1 \leq A \leq v2 : |valeurs| = v2 - v1$
    - $(A \leq v1) or (A \geq v2) : |valeurs| = max(A) - v2 + v1 - min(A)$
- $SF(AND(p1,p2)) = SF(p1) * SF(p2)$
- $SF(OR(p1,p2)) = SF(p1) + SF(p2) - SF(AND(p1,p2))$
- $SF(NOT(p)) = 1 - SF(p)$

Cardinalité des opérations

- **Cardinalité des opérations :**
- **Sélection :** $card(\sigma_{pred}(R)) = SF(pred) * card(R)$
- **Projection** : $card(\pi_A(R)) \leq card(R)$ (égalité si A unique)
- **Produit cartésien :** $card(RxS) = card(R) * card(S)$
- **Union :** $card(R \cup S) \in [max(card(R), card(S)); card(R) + card(S)]$
- **Différence :** $card(R - S) \in [0; card(R)]$
- **Jointure naturelle :** $card(R \bowtie_{R.a=S.a} S) = card(S)$ a : clé primaire de R
- **Jointure entre deux clés étrangères** :
    - A clé primaire de T, clé étrangère sur R et sur S
    - $card(R \bowtie_{R.a=S.a} S) = \dfrac{card(R) * card(S)}{D(T, A)}$
- **Jointure entre deux sélections** :
    - $card(\sigma_{p1}(R) \bowtie_A \sigma_{p2}(S)) = SF(p1) * SF(p2) * card(R \bowtie_A S)$

Règles de transformation

- Commutativité des opérations : produit cartésien, jointure, union
- Associativité des opérations binaires
- Idempotence des opérations unaires
- Distributivité de la sélection et la projection avec les opérateurs binaires

Stratégies de recherche

- **Déterministe :**
    - Construction des plans à partir des relations de base
    - Programmation dynamique : largeur d'abord
    - Excellent jusqu'à 5-6 relations
- **Aléatoire :**
    - Recherche optimale autour d'un point de départ donné
    - Réduit le temps d'optimisation au profit du temps d'exécution
    - Meilleur avec plus de 5-6 relations