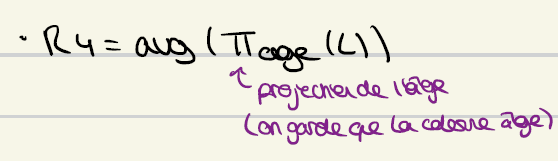
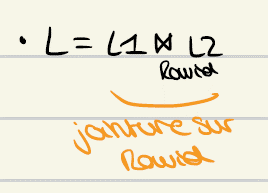
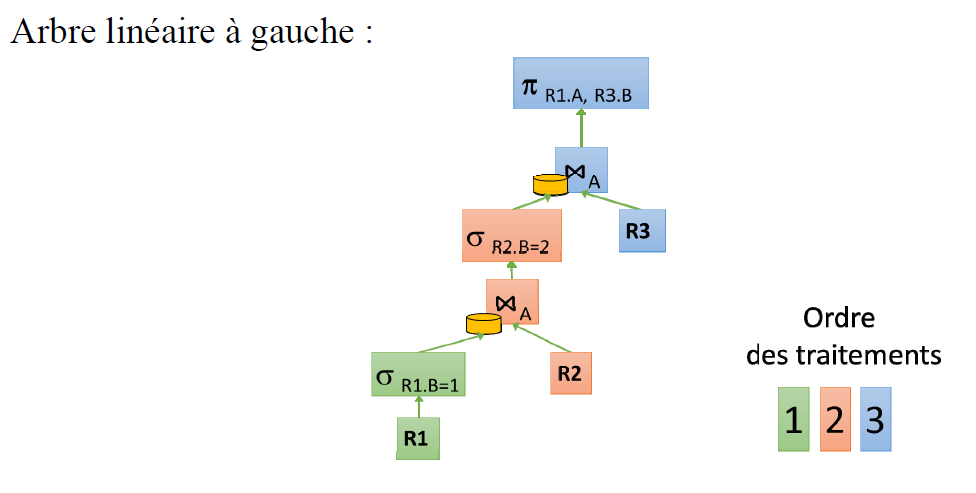
Cheatsheet :

Index :

* La première clé défini le tris de l’index -> important pour le parcourir
* Index couvrant : répondre à une requête sans lire les données, car le résultat de la requête se trouve dans l’index
* Index non plaçant :
* Cet index est-il utilisable pour cette requête ?
  + Oui
    - Atteindre (pred, min-machin)
    - Parcours (max-pred, max machin)
    - Vérifier que le parcours ne contient pas de clés inutiles
  + Non :
    - Le parcourt contient des clés inutiles

Symbole :



Hachage :

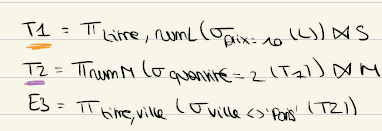
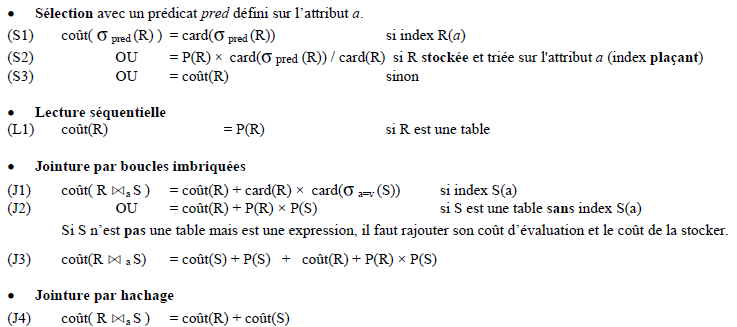
* Un répertoire contient des packets qui contienne des PL valeurs (je pense pointeur)
* Profondeur Globale : = taille du répertoire
* Profondeur Locale : toujours égale à PG au départ, c’est une valeur qu’on vas vite fait bouger et utiliser dans les algos
* Trouver l’ordre de R[A,B] -> Regarder les modulo des valeur à l’intérieur
* Insertion d’une valeur v :
  + On met v dans le packet numéro
  + Si PLi plein :
    - PG = PLi :
      * Doubler le répertoire : R[A,B] -> R[A,B,A,B] (PG = PG + 1)
      * Puis aller au cas suivant :
    - PLi < PG :
      * Créer un nouveau packet Pj
      * Remplacer le deuxième Pi par Pj
        + Si il y a 4 pi : « recopier le remplacement dans le reste du repertoire »
      * Incrémenter les PL de Pi et Pj
      * Repositionner les valeurs de P et la nouvelle valeur v entre Pi et Pj en recalculant les modulos avec la nouvelle valeur de PG
        + Si pi encore plein, refaire l’algo
* Suppression d’une valeur : si un packet i devient vide
  + On peut supprimer un packet vide uniquement si PL = PG, sinon il reste vide
  + Trouver le packet le plus proche ~~qui a le même~~ 
    - Je crois qu’il y a une histoire de suffixe en binaire
    - 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]
  + Le remplacer par ce paquet j
  + Décrémenter le PL du paquet j de 1

Hachage linéaire :

* 3 zones :
  + Une zone de débordement lorsqu’un paquet est plein mais que le taux d’occupation n’est pas dépassé
  + Une zone principale de taille
  + Une zone réservée aux éclatements
    - Quand la taille de cette zone devient plus grand que N on la passe en zone principale : N -> 2N
    - Et on reset p = 0
* Paramètre :
  + N paquet
  + p : prochain paquet à éclater
  + Fonction de hachage tel que
    - On garde en tête et
  + Taux d’occupation :
    - Nombre totale de valeur partout
    - ------------------------------------------------------------------
    - Nombre de cases dans zone principale + extension
    - (on ne compte pas la zone de débordement)
* Accès :
  + Calculer :
  + Si alors lire le paquet
  + Sinon lire le paquet
* Insertion :
  + Tant que le taux d’occupation n’est pas dépassé : insérer et insérer en zone de débordement
  + Dépassement du taux d’occupation
    - Eclater le paquet numéro p
    - Incrémenter p = p + 1
    - Mettre le nouveau paquet dans la zone éclatements
    - Répartir les valeurs entre le paquet éclaté et le nouveau paquet en utilisant la méthode d’accès ci-dessus

Arbre B+

Cout opération :

* Projection = le SELECT final
* Selection = WHERE = filtre
* la jointure sur l’attribue (en particulier l’attribu où S à un index)
* Classiquement sur un arbre linéaire de bas en haut : Jointure → Sélection → Projection
* Expression algèbrique d’un arbre :
  + Arret sur un des traits de l’arbre, écrire avec les en imbriquant tout
  + Ouais tout imbriquer en gros
  + 
  + Si quantité alors
  + **Parfois il faut ajouter le cout du index range scan si on l’a**
* facteur de sélectivité du prédicat  :
  + Je pense que c’est les fractions du dessus sans le cardinal de la table
  + Je pense que
* Suivre les formules à la lettre
  + 
  + P(R) = Nombre de page
  + On peut trouver certaines valeurs qui nous manque par équation !
    - Exemple avec dans l’interro de cours
* Enumérer tous les plans équivalents
  + Penser à la commutativité de la jointure
  + Pas forcément utiliser tous les index disponibles
  + Bon exo 3 TD3 correction Manon
* EXO2 TD3 POTENTIELLEMENT PARTIEL

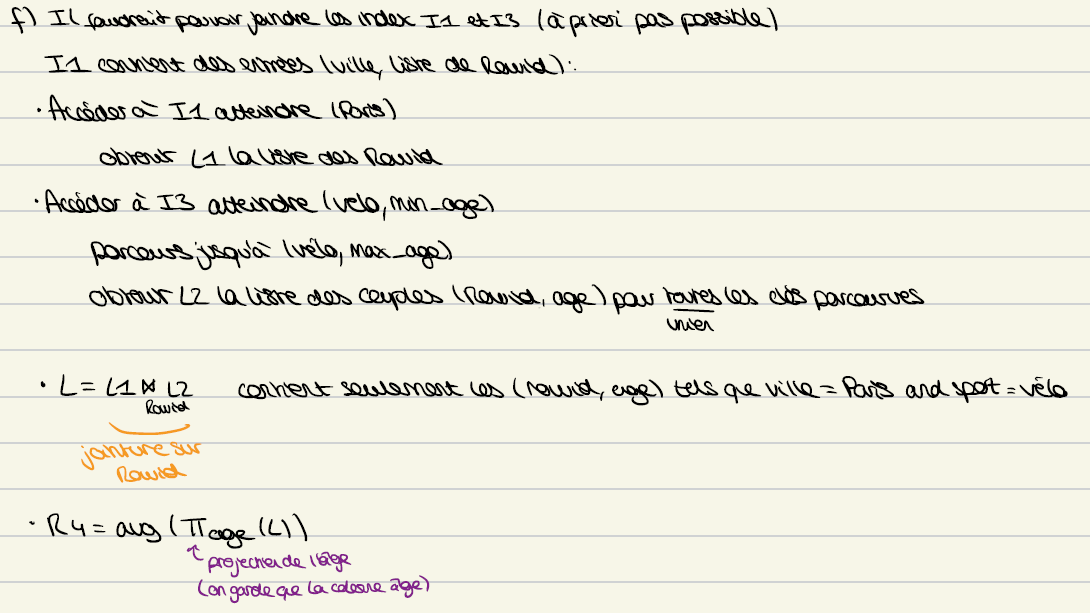
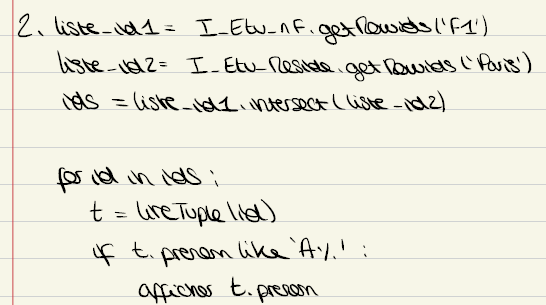
Plans requête SQL

* 4 méthodes d’accès à très faible cout = Nb de page lus = nombre de page stocké par l’index :
  + INDEX RANGE SCAN :
    - cout = (profondeur de l’index – 1) + nombre de row sélectionné (colonne row du plans) \* nombre de page de l’index / nombre de ligne la table
    - Ou plus facile parfois avec un predicat
    - Cout = (profondeur – 1) + nb de page de l’index \* SF(pred)
  + INDEX UNIQUE SCAN
  + INDEX FULL SCAN
  + INDEX SKIP SCAN
* TABLE ACCESS BY ROW ID :
  + Cout plus fort :
    - Cost\_table\_Access\_By\_Rowid(n) = Cost\_Index\_Range\_Scan(n) + n \* CF / num\_rows
    - Num\_rows = card(T) = nombre de ligne de la table
    - n = nombre de row sélectionné (colonne row du plans) = estimé par
    - CF = clustering factor de l’index
  + Prédicat : access("PRIX"<2000)
* Y’a un truc qui estime par
  + cardSelection(T, ATT, V1, V2) = (V2 – V1) / ( borneSup – borneInf de q ) \* num\_rows
* TABLE ACCESS FULL (+ un filter(pred) en général) :
  + Cout == cout lecture sequentiel de la table = nombre de page en théorie
  + En pratique : coût\_lecture\_sequentielle(T) = pages(T) / multiRead + card(T) / cpu
    - avec multiRead = 3,77 et cpu = 58000
* Remplir un plan de requête :
  + 1) estimer le nombre de lignes avec (voir formule plus haut)
  + 2) Avec ce nombre de ligne on peut calculer les couts en fonction de ce qui est utilisé ?

Pseudo-code :

* Index combine : Liste\_id1.intersect(liste\_id 2)
* Filter() : if pred1 and pred2 : affiche(row)
* LIKE ‘A%’ : if t.prenom like ‘A%’
* TABLE ACCESS FULL : for e in Etudiant.tuple()
* INDEX SCAN : for id in I\_FORMA\_NF.getRowId(pred)
* HASH Join :
  + Pas d’intrication de boucle : plusieurs boucle si nécéssaire
  + Voir code dernière page

Rédaction/question cool :

* Exercice 2, question f) 
* Jointure par boucle avec index TD4.1.3
* Hash join TD4.1.5