SGBD : Bases de données avancées [M3106C]

Hocine ABIR

20 mars 2014

IUT Villetaneuse
E-mail: abir@iutv.univ-paris13.fr

TABLE DES MATIÈRES

CHAPITRE

UN

Optimisation des Requêtes

1.1 Introduction

- Un Système de Gestion de Base de Données est basé sur un langage évolué non-procédural : langage de requêtes (`SQL`), et utilise des *opérateurs* pour évaluer ces reqêtes.
- Une requête :
  - permet la consultation, la mise à jour des données,
  - spécifie le *résultat* que l'on veut obtenir,
  - ne spécifie pas les *chemins d'accès* aux données, ni les *opérateurs* qu'il faut utiliser pour obtenir ce résultat.

Le compilateur de requête est un sous-système du SGBD qui détermine comment :
- accéder aux données,
- exécuter efficacement la requête.

Le problème de trouver un plan d'exécution qui optimise à la fois le coût de :

1. l'accès aux données : Entrées/ Sorties, et
2. l'évaluation de la requête : Mémoire et Processeur

n'a pas de solution générale (c'est à dire pour toutes les requêtes).

Des heuristiques sont alors utilisées pour optimiser les performances : quasi-optimisation.

Ces performances dépendent de plusieurs facteurs. Ceux qui intéressent l'optimisation sont :
- le nombre d'accès disque effectués,
- l'espace mémoire centrale utilisé,
- la durée des opérations (unité centrale)

Tous les compilateurs de requêtes ont une phase d'optmisation qui exploite les chemins possibles aux données : l'organisation physique des données (schéma interne) est un élément important pour l'optmisation des requêtes.

1.2 Importance de l'optimisation

En général, pour évaluer une requête, il y a plusieurs stratégies possibles, correspondant aux différentes translations de la requête en algèbre relationnelle.

Ces translations diffèrent par :

1. *les opérateurs utilisés.*
   Par exemple, les expressions :

(a) $(\sigma^{[Cond1]}R) \cap (\sigma^{[Cond2]}R)$ et

(b) $\sigma^{[Cond1 \wedge Cond2]}R$

sont "équivalentes".

2. *les chemins d'accès aux données.*
   Par exemple, l'opérateur σ dans l'expression $\sigma^{[Cond1 \wedge Cond2]}R$ peut sélectionner les tuples d'une table :
   (a) en effectuant un accès séquentiel à la table R ou
   (b) en utilisant un index défini sur cette table ou
   (c) toute autre méthode d'accès !

Pour illuster ce problème, on considère :
- le schéma de base de données suivant :

```
CREATE TABLE Etudiant(
                NumE int primary key, -- numero d'un etudiant
                NomE varchar        -- nom d'un etudiant
);
CREATE TABLE Matiere(
                NumM int primary key , -- numero d'une matiere
                NomM varchar         -- nom d'une matiere
);
CREATE TABLE Resultat(
                NumE int references Etudiant,
                NumM int references Matiere,
                Note decimal(4,2)    -- une note d'un etudiant dans une matere
);
```

- la requête suivante sur ce schéma de base de données :

"Quels sont les noms des étudiants qui ont obtenu une note inférieure à 10 dans au moins une matière ?"

```
SELECT e.NomE
   FROM Etudiant e, Resultat r
   WHERE e.NumE=r.NumE AND r.Note < 10;
```

- et les deux stratégies ci-dessous pour évaluer cette requête :
  Stratégie 1

$$\Pi^{[e.NomE]}(\sigma^{[e.NumE=r.NumE \wedge r.Note<10]}(e \times r)) \tag{1.1}$$

  Stratégie 2

$$\Pi^{[e.NomE]}((\sigma^{[r.Note<10]}r) \bowtie^{[e.NumE=r.NumE]} e) \tag{1.2}$$

Pour comparer ces deux stratégies, on suppose :
- cardinalité de e : $|e| = 200$ tuples
- cardinalité de r : $|r| = 800$ tuples
- facteur de sélectivité de $\sigma^{[r.Note<10]}$: $s = 0.25$,
- Temps : proportionnel à la somme des tailles des tables intermédiaires c'est à dire $t = O(\sum(|Ti|))$.

On obtient :
- Startégie 1 :

Etapes d'Evaluation	*Complexité*
$T^1 = e \times r$	$\|T^1\| = 160\,000$
$T^2 = \sigma^{[e.NumE=r.NumE \wedge r.Note<10]} T^1$	$\|T^2\| = 200$
$Resultat = \Pi^{[e.NomE]} T^2$	$t = O(160\,200)$

- Startégie 2 :

Etapes d'Evaluation	*Complexité*
$T^1 = \sigma^{[r.Note<10]} r$	$\|T^1\| = 200$
$T^2 = T^1 \bowtie^{[e.NumE=r.NumE]} e$	$\|T^2\| = 200$
$Resultat = \Pi^{[e.NomE]} T^2$	$t = O(400)$

1.3 Stratégie et Complexité

On considère le schéma de base de données suivant :

```
CREATE TABLE fournisseur(
      fo_numero    int primay key,
      fo_nom       varchar,
      fo_categorie  varchar
);
CREATE TABLE projet(
  pr_numero int primay key,
  pr_nom    varchar,
   pr_ville   varchar
);
CREATE TABLE article(
  ar_numero int primay key,
  ar_nom     varchar,
  ar_couleur varchar,
  ar_poids   numeric(10,2)
);
```

```
CREATE TABLE commande(
   co_fournisseur  int
      references   fournisseur (fo_numero),
   co_projet       int
      references   projet (pr_numero),
    co_article     int
      references   article (ar_numero),
   co_quantite     int
);
```

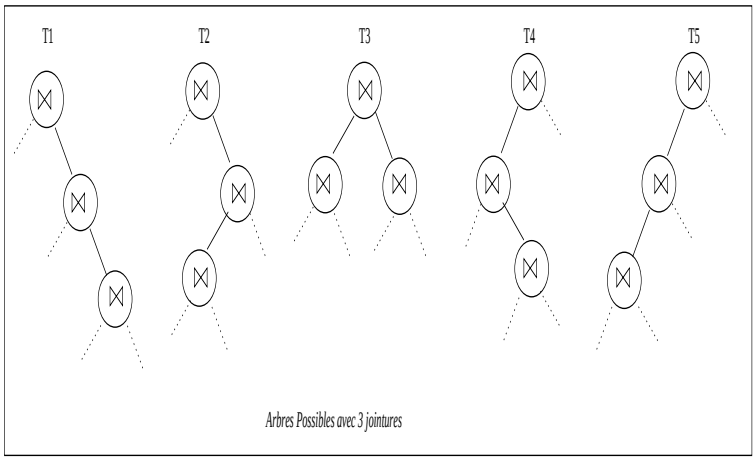
et la requête :

```
   SELECT pr_nom
      FROM projet, commande,
           fournisseur ,  article
      WHERE fo_numero=co.fournisseur
        AND  pr_numero= co_projet
        AND  ar_numero=co_article
        AND    ar_couleur= 'ROUGE'
        AND    fo_nom= 'DUBOIS';
```

1. Nombre de schémas d'arbre algébriques que l'on peut constituer avec n relations est le $(n-1)^{eme}$ nombre de *Catalan* :

$$S^m = \begin{pmatrix} 2m \\ m \end{pmatrix} \frac{1}{m+1} = \frac{2m!}{m!m! \times (m+1)} = \frac{(m+2) \times ... \times (2m)}{m!} \qquad (1.3)$$

   Pour $n = 4$ on a $S^3 = \frac{5 \times 6}{2 \times 3} = 5$



2. Le nombre d'affectation possibles P^n de n relations aux feuilles est :

$$P^n = n! \tag{1.4}$$

Pour $n = 4$ on a $P^4 = 4! = 24$

```
------------------------------
Permutation  1 :  projet commande fournisseur article
Permutation  2 :  projet commande article fournisseur
Permutation  3 :  projet fournisseur commande article
Permutation  4 :  projet fournisseur article commande
Permutation  5 :  projet article commande fournisseur
Permutation  6 :  projet article fournisseur commande
Permutation  7 :  commande projet fournisseur article
Permutation  8 :  commande projet article fournisseur
Permutation  9 :  commande fournisseur projet article
Permutation 10 :  commande fournisseur article projet
Permutation 11 :  commande article projet fournisseur
Permutation 12 :  commande article fournisseur projet
Permutation 13 :  fournisseur projet commande article
Permutation 14 :  fournisseur projet article commande
Permutation 15 :  fournisseur commande projet article
Permutation 16 :  fournisseur commande article projet
Permutation 17 :  fournisseur article projet commande
Permutation 18 :  fournisseur article commande projet
Permutation 19 :  article projet commande fournisseur
Permutation 20 :  article projet fournisseur commande
Permutation 21 :  article commande projet fournisseur
Permutation 22 :  article commande fournisseur projet
Permutation 23 :  article fournisseur projet commande
Permutation 24 :  article fournisseur commande projet
------------------------------
```

Soit au total $5 \times 24 = 120$ arbres algébriques.

3. Chemins d'accès à chaque relation :
   Sequentiel, $B+$ Index, Hash Index, etc
4. Opérateurs internes de jointure :
   Nested Loop, Hash Join, Merge Join ,etc

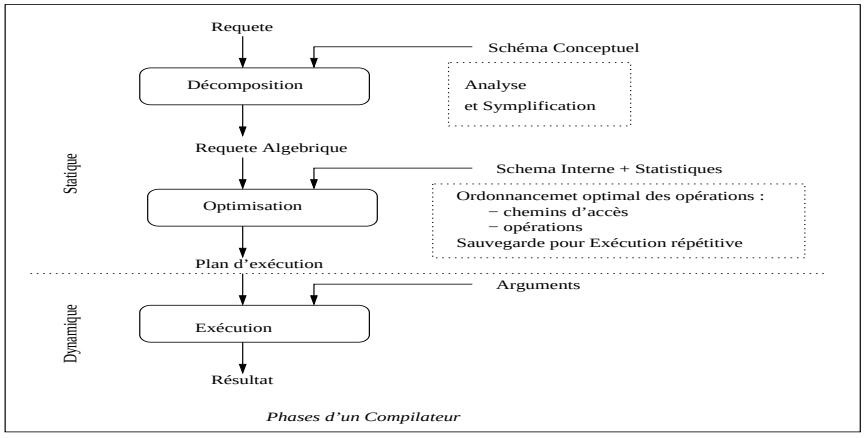
Quand le nombre n de relations de la clause `FROM` est

- peu important ($\leq$ 10), la méthode exhaustive réduit l'espace de recherche en utilisant des heuristiques.
- important ($>$ 10), la recherche exhaustive devient inexploitable. L'espace de recherche reste trop grand malgrès les heuristiques : méthode génétique.

1.4 Différentes Phases de traitement d'une Requête

Le traitement d'une requête peut être décomposé en :

1. Analyse :
   - analyse syntaxique et sémantique de la requête,
   - translation de la requête en langage algébrique sous forme d'un arbre : *arbre algébrique*,
2. Optimisation (phase statique) :
   - générer les différents arbres algébriques équivalents (restructuration),
   - traduit les références conceptuels (relation) en références physiques (fichier),
   - traduit les opérateurs algébriques en opérateurs de base,
   - Evaluer chacun de ces arbres
   - déterminer l'arbre optimale : plan d'exécution
3. Exécution : évalue le plan d'exécution.



1.5 Restructuration

1.5.1 Règles Utilisées

La sélection (σ) et la projection (Π) réduisent la taille de leurs opérandes respectivement horizontalement et verticalement. La restructuration est basée sur cette idée (heuristique) et applique ces opérateurs aussitôt que possible.

Règles de Réduction Horizontale

1. Regroupement des restrictions :

$$\sigma^{[Ap=a \wedge Aq=b]}R \longleftrightarrow \sigma^{[Ap=a]}(\sigma^{[Aq=b]R}) \tag{1.5}$$

2. Restriction et projection :

$$\sigma^{[Ai=a]}(\Pi^{[A1...Ap]}R) \longleftrightarrow \begin{cases} \Pi^{[A1...Ap]}\ (\sigma^{[Ai=a]}R) & si\ i \in [1...p] \\ \Pi^{[A1...Ap]}\ (\sigma^{[Ai=a]}(\Pi^{[A1...Ap,Ai]}R)) & sinon \end{cases} \tag{1.6}$$

3. Restriction et jointure :

$$\sigma^{[Ai=a]}(R \bowtie S) \longleftrightarrow \begin{cases} (\sigma^{[Ai=a]}R) \bowtie S & siAi \in R \\ R \bowtie (\sigma^{[Ai=a]}S) & siAi \in S \end{cases} \tag{1.7}$$

4. Restriction et Union ou différence

$$\sigma^{[Ai=a]}(R \cup S) \longleftrightarrow (\sigma^{[Ai=a]}R) \cup (\sigma^{[Ai=a]}S) \tag{1.8}$$

$$\sigma^{[Ai=a]}(R - S) \longleftrightarrow (\sigma^{[Ai=a]}R) - (\sigma^{[Ai=a]}S) \tag{1.9}$$

Règles de Réduction Verticale

1. Projection et jointure
$\Pi^{[A1..Ap,B1..Bq]}(R \bowtie^{[Ai=Bj]} S) \longleftrightarrow$

$$\begin{cases} (\Pi^{[A1..Ap]}R) \bowtie (\Pi^{[B1..Bq]}S) & siAi \in [1..p]\ et \\ & siBj \in [1..q] \\ \Pi^{[A1..Ap,B1..Bq]}((\Pi^{[A1..Ap,Ai]}R) \bowtie (\Pi^{[B1..Bq,Bj]}S)) & sinon \end{cases} \tag{1.10}$$

2. Projection et Union :

$$\Pi^{[A1..Ap]}(R \cup S) \longleftrightarrow (\Pi^{[A1..Ap]}R) \cup (\Pi^{[A1..Ap]}S) \tag{1.11}$$

Règles d'énumération

1. Commutativité de la jointure

$$R \bowtie S \longleftrightarrow S \bowtie R \tag{1.12}$$

2. Associativité de la jointure :

$$(R \bowtie S) \bowtie T \longleftrightarrow R \bowtie (S \bowtie T) \tag{1.13}$$

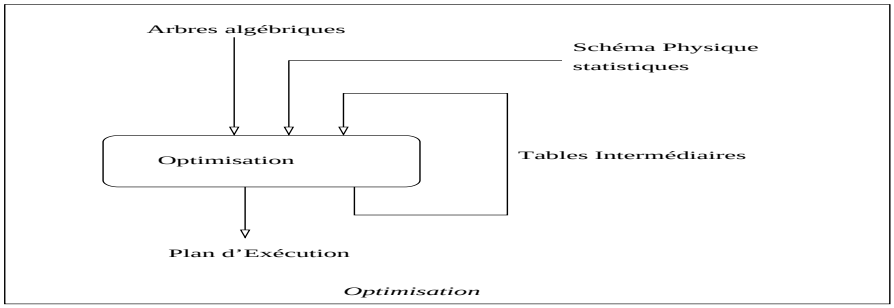
1.5.2 Principe de la Méthode

Etape Heuristique

1. Décomposer chaque restriction ayant une qualification conjonctive en séquence de restrictions en utilisant la règle (1.5),
2. Déplacer chaque restriction vers l'intérieur (feuilles) autant que possible, en utilisant les règles (1.6), (1.7) ,(1.8) et (1.9),
3. Regrouper les restrictions sur les mêmes relations en utilisant la règle (1.5) et supprimer les projections redondantes règle ,
4. Déplacer les projections vers l'intérieur (feuilles) autant que possible en utilisant les règles (1.10) et (1.11).

Etape génération

Enumérer les arbres algébriques en utilisant les règles (1.12) et (1.13).



1.6 Estimation des Coûts des Plans

1.6.1 Paramètres du Coût

Le coût d'un plan d'exécution est déterminé par la somme des coûts des opérations de l'arbre considérées individuellement.

Le coût de chaque opération est composé :
• du coût des accès disques : fonction du nombre E/S ,
• du coût de traitement : fonction du temps UC et de l'espace mémoire utilisé,
Pour estimer le coût d'un plan, trois informations sont importante à chaque estimation du coût d'un opérateur du plan :

1. le coût individuel de chaque opération ,
2. la taille de la table intermédiaire résultante,
3. l'ordre du Résultat : ordonné ou non-ordonné.

1.6.2 Statisques

Pour estimer ce coût, les statistiques suivantes sont nécessaires :

Table

- La cardinalité du domaine de chaque attribut $dom(A)$,
- Le nombre de valeur distinctes de chaque attribut $ndist(A)$,
- La valeur maximale et minimale de chaque attribut $max(A), min(A)$,
- La cardinalité de chaque relation $card(R)$,
- Ordre de la table (cluster).

Index

- Nombre de niveaux,
- Unicite : unique ou nonunique

1.6.3 Le facteur de sélectivité

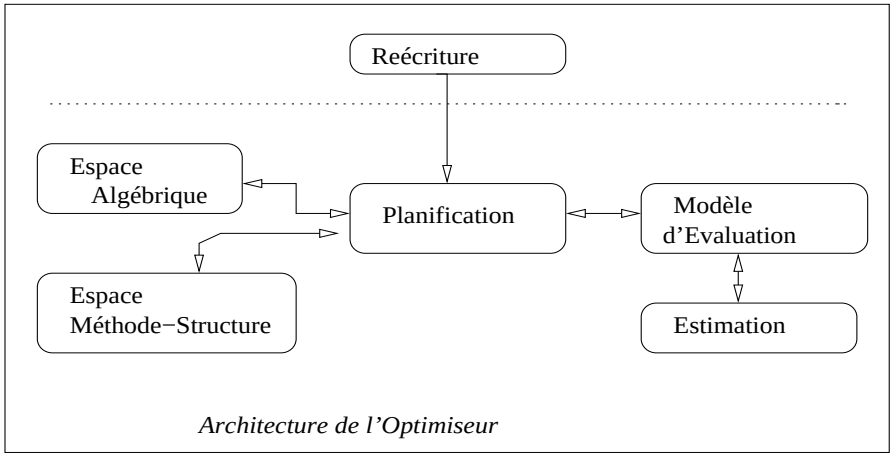
le facteur de sélectivité d'un prédicat est la proportion des tuples qui satisfont ce prédicat.

La cardinalité du résultat d'une opération basée sur un prédicat (sélection, jointure) est une fonction de la cardinalité des opérandes et du facteur de sélectivité s du prédicat.
- sélection : $s * card(R)$
- jointure : $s * card(R) * card(S)$

1.6.4 calcul du facteur de sélectivité s

- $s(A = valeur) = \frac{1}{ndist(A)}$
- $s(A > valeur) = \frac{max(A)-valeur}{max(A)-min(A)}$
- $s(A < valeur) = \frac{valeur-min(A)}{max(A)-min(A)}$
- $s(P \wedge Q) = s(P) * s(Q)$
- $s(P \vee Q) = s(P) + s(Q) - (s(P) * s(Q))$

1.7 Architecture Conceptuel de l'optimiseur



Architecture de l'Optimiseur

1.7.1 Module de Reécriture

Le Module de *Récriture* transforme l'arbre algébrique d'une requête en utilisant uniquement les caractéristiques statiques de la requête, par exemple :
- remplacement des vues,
- application des règles de reécriture.

1.7.2 Module Espace Algébrique

Le module *Espace Algébrique* détermine les différents ordres d'exécution des opérations pour le module de planification en utilisant la méthode décrite au point 1.5.

Pour réduire la taille de l'espace de recherche, ce module utilise 3 $Heuristiques$:

H1 : Les projections (Π) et les selections (σ) sont excécutées à la volée. Les sélections sont effectuées durant les premiers accès de chaque relation. Les projections sont traitées durant la génération du résultat des autres opérateurs.

H2 : Le produit cartésien n'est jamais calculé, sauf si la requête le demande. Les relations sont toujours combinées en utilisant les jointures.

H3 : l'opérande interne (droite) de chaque jointure ne doit jamais être une table intermédiaire (récursivité à gauche). Les JPT (Join Processing Tree)sont des arbres qui permettent de minimiser les coûts des jointures. Cette heuristique a pour objectif de permettre :
- l'utilisation des méthodes d'accès de la relation interne,
- d'exécuter une suite de jointures en pipeline.

1.7.3 Module Espace Méthode-Structure

Les opérateurs suivants sont en général des opérateurs de base :
- Restriction et projection sans élimination de doublets,
- Elimination des doublets,
- Jointure et projection sans élimination des doublets,
- Opérateurs ensemblistes : Union, Différence, Intersection, Produit cartésien,
- Opérateurs de mise à jour : insert, delete, update, ...
- tri
- Fonctions d'aggrégation : count, max, ..

Chaque opérateur de base peut être implémenté par plusieurs algorithmes. Chacun des algorithmes peut :
- être optimal dans certains circonstances ou
- contenir des chemins d'accès particuliers (index, ..)

La stratégie d'exécution dicte :

1. les chemins d'accès aux données,
2. le choix des algorithmes pour effectuer les opérations,
3. l'ordre de ces opérations.

Le module *Espace Méthode-Structure* détermine les différentes implémentations possibles pour une suite d'opérations donnée par le module *Espace Algébrique*. Il produit un plan d'exécution complet. Ce module utilise :
- les methodes disponibles : pour une jointure,par exemple, nested loop, merge scan, hash join.
- les opérations effectuées au vol : par exemple, sélection, projection.
- les méthodes d'accès

1.7.4 Modèle d'Evaluation

Le modèle d'*Evaluation* spécifie les formules arithmétiques qu'il faut utiliser pour estimer des plans d'exécution. A chaque :
- méthode d'accès,
- type de jointure,
- type d'opération de base

est associée une formule d'évaluation. Ces formules sont basées sur la taille du buffer d'E/S, la taille de la relation et la distribution des valeurs.

1.7.5 Module d'Estimation

Le module d'*Estimation* spécifie comment estimer les tailles des relations intermédiaires et les distributions des valeurs des attributs.

1.7.6 Module de Planification

Le module de *Plannification* examine tout les plans d'exécution possibles de chaque arbre algébrique et choisit le moins coûteux. C'est le module principal, il :

- utilise une stratégie de recherche : qui examine l'espace des plans d'exécution d'une façon particulière,
- détermine cet espace par les modules Espace Algébrique et Esapce Méthode-Structure.
- compare les différents plans en utilisant les coûts calculés par les modules Modèle d'Evaluation et Estimation.

1.7.7 Exemple

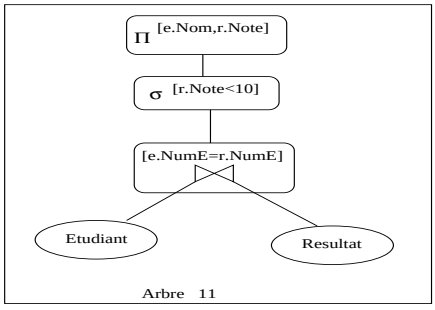
On considère le schéma :

```
CREATE TABLE Etudiant(
                NumE int primary key, -- numero d'un etudiant
                NomE varchar         -- nom d'un etudiant
);
CREATE TABLE Matiere(
                NumM int primary key , -- numero d'une matiere
                NomM varchar          -- nom d'une matiere
);
CREATE TABLE Resultat(
                NumE int references Etudiant,
                NumM int references Matiere,
                Note decimal(4,2)     -- une note d'un etudiant dans une matere
);
```

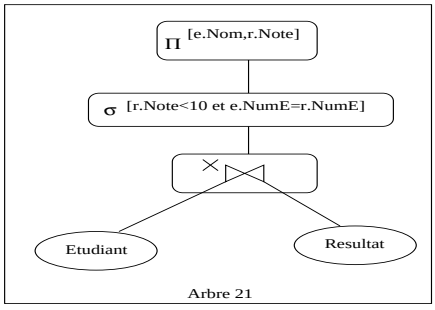
et la requête :

```
SELECT e.NomE
   FROM Etudiant e, Resultat r
   WHERE e.NumE=r.NumE AND r.Note < 10;
```

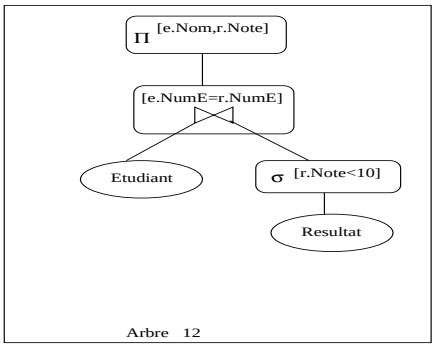
Espace Algébrique



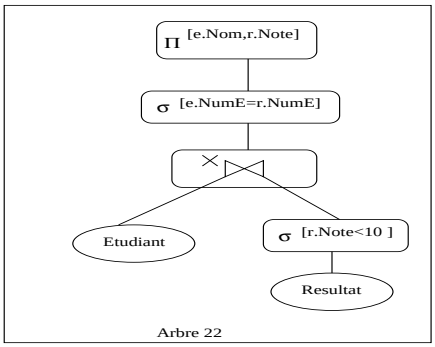
Arbre 11



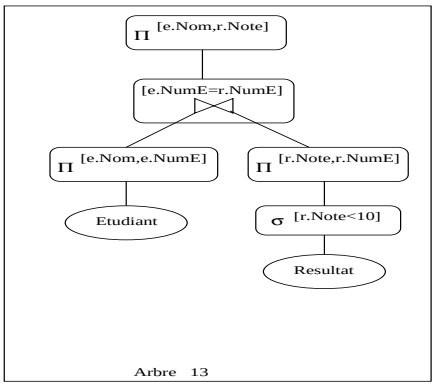
Arbre 21



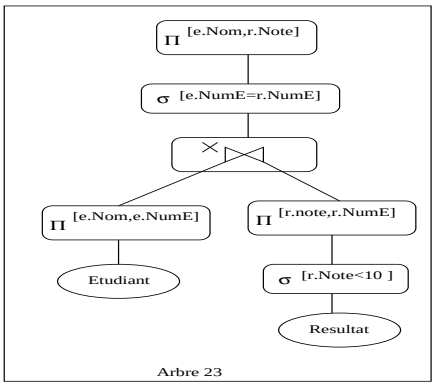
Arbre 12



Arbre 22



Arbre 13



Arbre 23

Soit 6 arbres et leur symétrique donc 12 arbres algébriques !

heuristiques

H1 : L'heuristique (H1) élimine les arbres (11), (12), (21), (22) et leur symétrique donc il reste 4 possibilités.

H2 : L'heuristique (H2) élimine les arbres (21), (22), (23) et leur symétrique donc il reste 2 possibilités : (13) et son symétrique.

H3 : Les deux possibilités restantes sont retenues. Cette heuristique mérite une attention partculière car elle concerne les règles (1.12) et (1.13) qui (à l'opposé des autres règles) multiplient les possibilités :

(a) Enumération des arbres
Le nombre de structure d'arbres que l'on peut constituer avec n relation (en ignorant leur affectation aux feuilles) correspond au $(n-1)^{eme}$ nombre de *Catalan* :

$$S^m = \left(\begin{array}{c} 2m \\ m \end{array} \right) \frac{1}{m+1} = \frac{2m!}{m!m! \times (m+1)} \tag{1.14}$$

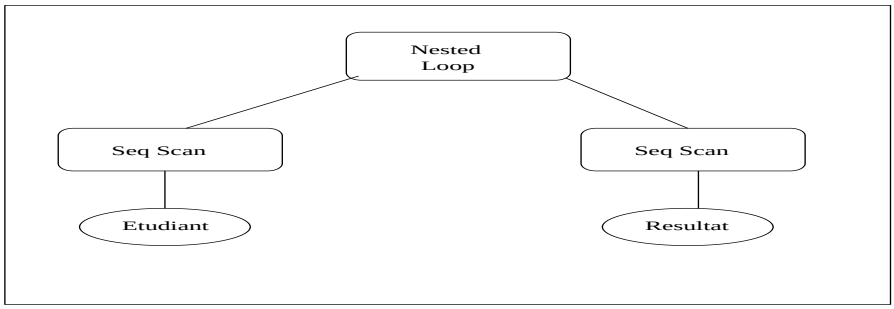
(b) Permutation des relations
Le nombre d'affectation possibles P^n de n aux feuilles est :

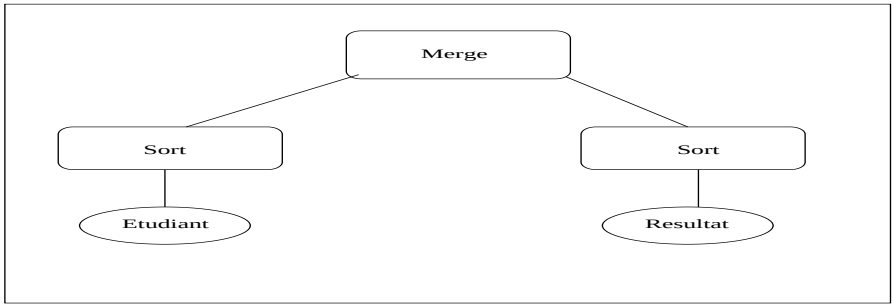
$$P^n = n! \tag{1.15}$$

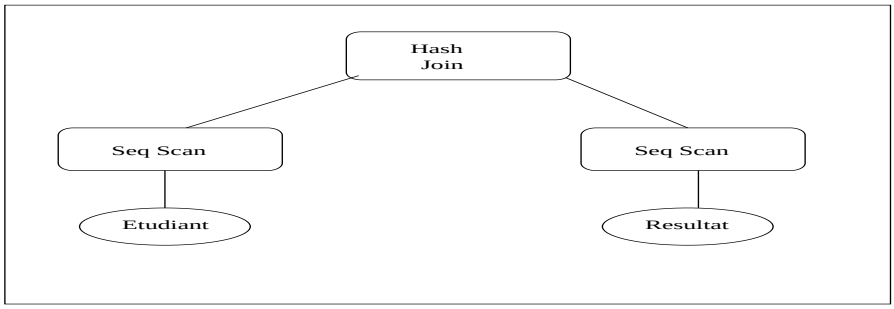
Soit au total, le nombre d'arbres possibles est :

$$B^n = S^{(n-1)} \times P^n \tag{1.16}$$

Si on suppose qu'aucun index n'est défini sur ces tables, on a :







1.8 Stratégie Recherche exhaustive

1.8.1 Caractéristiques générales

La stratégie de recherche exhaustive est :
- basé sur la programmation dynamique,
  - optimisation statique basée sur des statistiques
  - recherche exhaustive avec élimination (pruning) dynamique de certaines alternatives
- Prend en compte l'ordre des résultats

Les solutions possibles pour évaluer une opération (intermédiaire) d'un plan d'exécution peuvent être différenciées par :

1. coût de l'opération
2. ordre du résultat produit par l'opération

Caractériser chaque solution par l'ordre de ses résultats en plus de son coût permet :

1. de partionner les solutions suivant cet ordre
2. de conserver dans chaque classe la meilleure solution produisant le même résultat,
3. d'avoir une optimisation plus globale : le coût de l'opération suivante peut être influencé par l'ordre produit par l'opération précédente plutôt que son coût.

1.8.2 Schéma général

Etape 1 :

Obtenir pour chaque relation de la requête ses différents chemins d'accès. Partionner ces chemins d'accès suivant l'ordre du résultat qu'ils produisent. Estimer le coût des chemins de chaque partition et conserver le chemin de coût faible de chaque partition : on obtient des plans partiels à une seule relation.

Etape 2 :

En utilisant les plans d'accès déterminés à l'étape 1, déterminer pour chaque jointure entre deux relations de la requête, toutes les possibilités d'évaluer cette jointure. Partionner ces possibilités d'évaluation suivant l'ordre du résultat qu'elles produisent. Estimer le coût des possibilités (différents plans) de chaque partition et conserver le plan partiel ayant le plus faible coût dans chaque partition : on obtient des plans partiels à deux relations.

.....

Etape I :

En utilisant les plans partiels à $(I-1)$ relations construits à l'étape (I-1),

- Déterminer les I^{eme} relations de jointure avec les (I-1) relations de chaque plan,
- Déterminer toutes les possibilités d'évaluations de ces jointures, et partionner ces possibilités en fonction de l'ordre du résultat obtenu,
- Estimer le coût des possibilités de chaque partition, et conserver celle de coût faible de chaque partition : on obtient des plans partiels à I relations.

.....

Etape N :

En procédant comme à l'étape précédente, on obtient tous les plans possibles pour évaluer la requête : le meilleur plan est retenu pour évaluer la requête

1.9 Accès Séquentiel *Table Scan*

Méthode

```
r=premier tuple(R)
tant que (r ≠ fin(R)) {
      Sélection sur r,
      Projection de r ; (sans éliminer les doublants)
      r=tuple suivant(r) dans R ;
}
```

Coût

$$E/S = Blk(R)$$

1.10 Implémentations de la Jointure

La jointure est fermement liée au produit cartésien, c'est à dire que chaque tuple du produit cartésien doit être considéré pour établir le résultat.
Les différentes méthodes se classent en 3 groupes :

1. *Nested Loop Join*
2. *Merge Join*
3. *Hash Join*

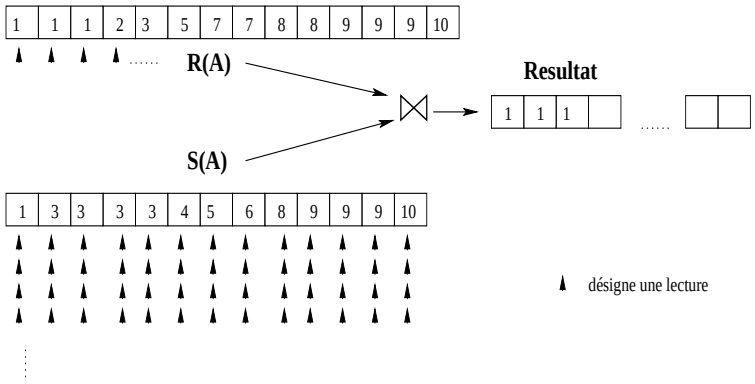
1.10.1 *Nested Loop Join*

Méthode

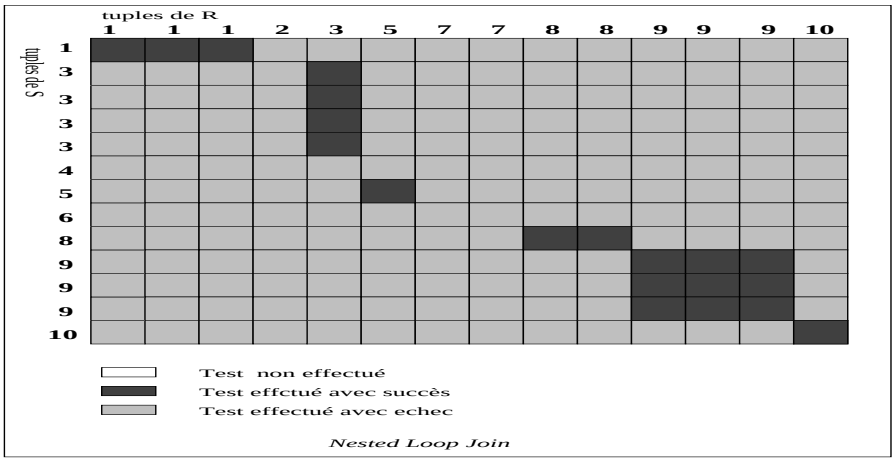
nlj(R,S,i,j)

```
Resultat = ∅ ;
r=premier tuple(R)
tant que (r ≠ fin(R)) {
      s=premier tuple(S) ;
      tant que (s ≠ fin(S)) {
            si (r^i=s^j ) {
                  ajouter tuple (r, s) à Resultat ;
            }
            s= tuple suivant(s) dans S ;
      }
      r=tuple suivant(r) dans R ;
}
```

Exemple



Performance



$$E/S = |R| + |R| \times |S|$$

1.10.2 *Merge Join*

$mj(R,S,i,j)$

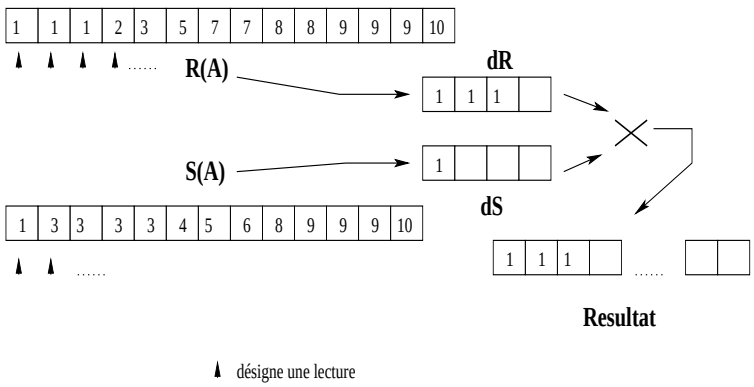
Méthode

- la relation R doit être triée suivant r^i,
- la relation S doit être triée suivant s^j

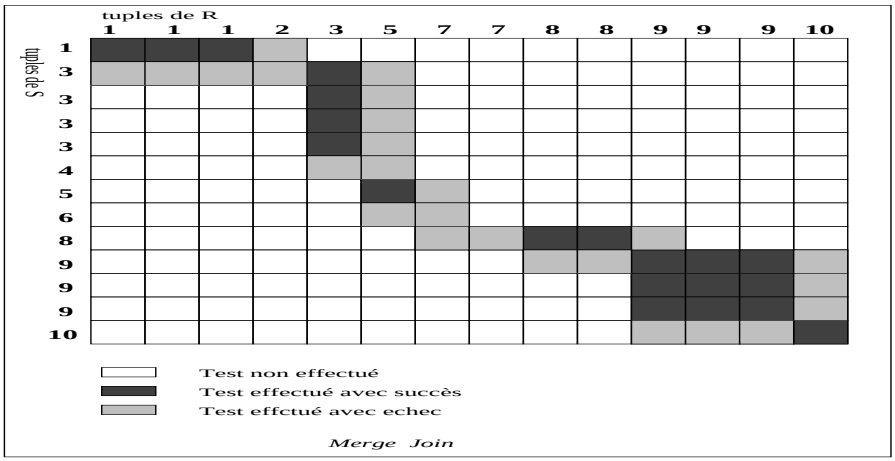
Les tuples ayant la même valeur des attributs de jointure sont regroupés.

```
Resultat = ⊘ ;
r=premier tuple(R) ;
s=premier tuple(S) ;
tant que (r ≠ fin(R) and s ≠ fin(S) ) {
      si (r^i > s^j ) {
            s=tuple suivant(s) dans S ;
      sinon si (r^i < s^j ) {
            r=tuple suivant(r) dans R ;
      sinon {
            ajouter tuple (r, s) à Resultat ;
            s'=tuple suivant(s) dans S ;
            tant que (s' ≠ fin(S) and r^i = s'^j) {
                  ajouter tuple (r, s') à Resultat ;
                  s'=tuple suivant(s') dans S ;
            }
            r'=tuple suivant(r) dans R ;
            tant que (r' ≠ fin(R) and r'^i = s^j) {
                  ajouter tuple (r', s) à Resultat ;
                  r'=tuple suivant(r') dans R ;
            }
            r=tuple suivant(r) dans R ;
            s=tuple suivant(s) dans S ;
      }
}
```

Exemple



Performance



$$E/S \simeq |R| + |S|$$

1.10.3 *Hash Join*

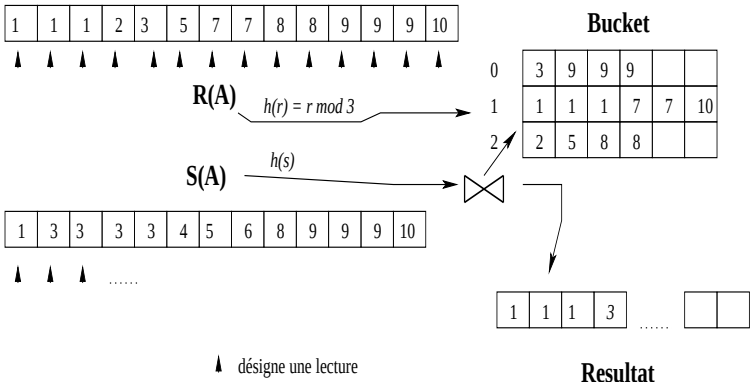
$hj(R,S,i,j)$

Méthode

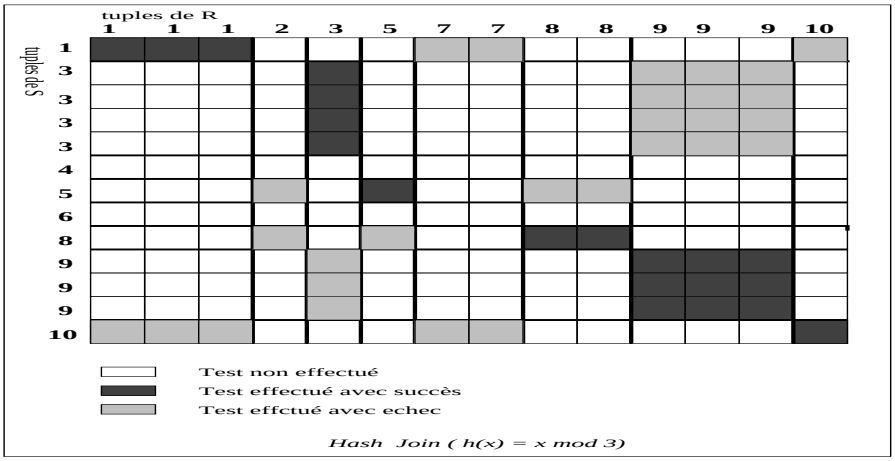
Les tuples ayant la même valeur (ou valeurs simmilaires) des attributs de jointure sont partitionnées par une fonction de hachage h.

```
Resultat = ∅ ;
r=premier tuple(R) ;
tant que (r ≠ fin(R) ) {
      ajouter tuple (r) à Bucket(h(r^i))
      r=tuple suivant(r) dans R ;
} s=premier tuple(S) ;
tant que (s ≠ fin(S)) {
      si (s^j ∈ bucket(h(s^j) ) {
            ajouter tuple (r, s) à Resultat ;
      }
      s= tuple suivant(s) dans S ;
}
```

Exemple



Performance



$$E/S = |R| + |S|$$

1.11 Généralités sur les Méthode d'accès

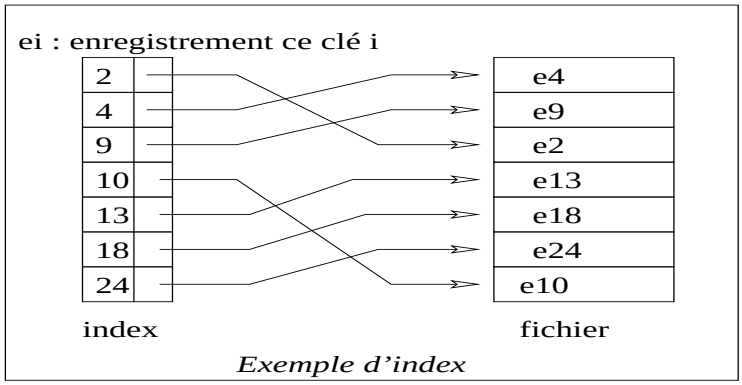
1.11.1 Introduction

Clé

Une *clé* est un sous-ensemble d'attributs (ou champs) d'un objet (ou fichier) qui permet d'identifier (ou de déterminer) un ou plusieurs tuples (ou enregistrements). Une clé permet de décrire les attributs d'un objet sous forme *clé,adresse physique*.

Index

Le principe de base de l'organisation indexée est d'associer à un fichier, une table ayant une entrée pour chaque couple (clé,adresse physique).



Exemple d'index

1.11.2 Caractéristiques d'un index

ordre

Un index de N clés réparties en b clés par bloc peut être :
- *non-ordonné* : nécessite en moyenne

$$\frac{N}{2b} \tag{1.17}$$

  accès disque pour trouver une clé si l'index n'est pas ordonné.
- *ordonné* : nécessite

$$\log [2](\frac{N}{b}) \tag{1.18}$$

  accès disque pour trouver une clé si l'index est ordonné.

Densité

Un index peut être :
- *dense* : il contient toutes les clés des différents enregistrements.
- *non-dense* : il contient seulement certaines clés des différents enregistrements.

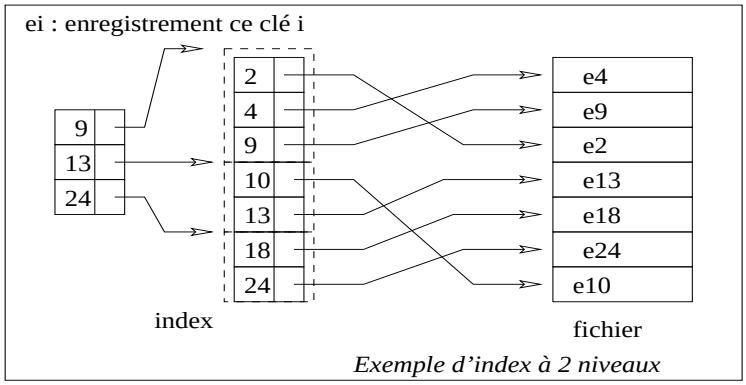
La densité d'un index est le rapport entre le nombre de clés de l'index et le nombre d'enregistrement du fichier. L'index est dit dense si la densité est égale à 1 et non-dense si la densité est inférieure à 1. Si l'index est non-dense, le fichier et l'index doivent être ordonnés.

Partiel(sparse)/total

Un index partiel est un index construit sur une partie choisie des clés.

Niveaux

Un index peut être vu comme un fichier de clés. Si le nombre de clés est important, il devient intéressant de créer un index pour l'index : on obtient alors un index dit *multi-niveaux*.



Exemple d'index à 2 niveaux

1.11.3 Méthodes d'accès sous PostgreSQL

Les méthodes d'accès disponibles sur Postgres :

```
=> select amname as "Méthodes d'Accès" from  pg_am;
 Méthodes d'Accès
------------------
 rtree
 btree
 hash
 gist
(4 rows)
```

Chaque méthode d'accès structure et organise des informations sur les données pour permettre l'accès aux données : ces structures s'appellent des *index*.

```
=> \di
                  List of relations
 Schema |        Name        | Type  | Owner |  Table
--------+--------------------+-------+-------+----------
 public | controle_pkey      | index | abir  | controle
 public | etudiant_pkey      | index | abir  | etudiant
 public | note_et_numero_key | index | abir  | note
 public | note_ridx          | index | abir  | note
(4 rows)
```

La commande `CREATE INDEX` permet de créer et initialiser ces structures :

`CREATE` [`UNIQUE`] `INDEX` *Nom_Index* `ON` *Nom_Table*
[`USING` *Méthode_Accès*] *Colonne* [*Nom_Opérateur*] [, ...])
[`WHERE` *Qualification*]

Le contenu d'un index :
- est entièrement controlée par les méthodes d'accès de l'index.
- décrit une application `clé` $\longrightarrow$ `tid`

Les tuples d'une relation ayant la même clé (valeur) peuvent être regroupés dans un même bloc ou dans des blocs adjacents par la commande `SQL CLUSTER`.

```
=> cluster etudiant_pkey on etudiant;
CLUSTER

=> select ctid,et_nom from etudiant;
 ctid  | et_nom
-------+---------
 (0,1) | ALTINIK
 (0,2) | AZAGBA
 (0,3) | YAPO
(3 rows)
```

1.12 $B+$ Index

Les arbres $B+$ ont été introduit pour implémenter des index à plusieurs niveaux.

1.12.1 Définiton

Un arbre $B+$ d'ordre $m(m \geq 1)$ est un arbre tel que :

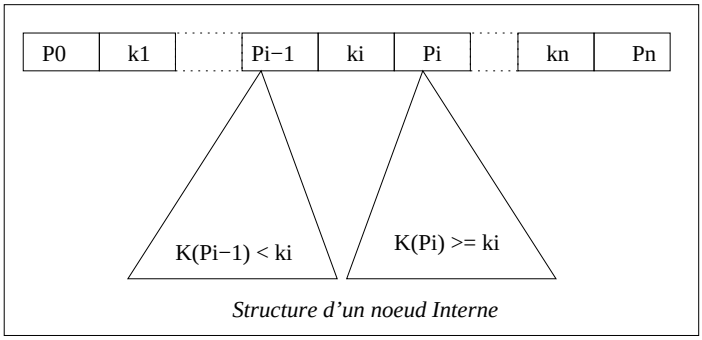
1. toutes les feuilles sont au même niveau (arbre équilibré),
2. la racine a k clés et r descendants tel que

$$\begin{aligned} k &\in [0..2m] \\ r &\in [1..2m+1] \end{aligned} \tag{1.19}$$

3. les autres noeuds ont k clés et n descendants tel que

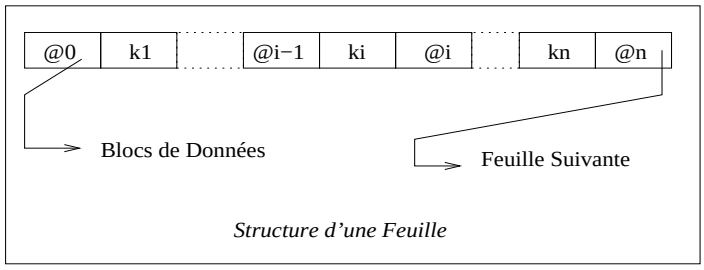
$$\begin{aligned} k &\in [m..2m] \\ n &\in [m+1..2m+1] \end{aligned} \tag{1.20}$$

4. les clés sont ordonnées (ordre partiel) suivant un parcours préfixe, comme illutré par la figure ci-dessous où :
   - Pi : désigne une référence (pointeur)
     - interne (structure de l'arbre) pour les noeuds internes,
     - externe (adresse du bloc physique contenantl'enregistrement de clé ki)
   - K(Pi) : désigne l'ensemble des clés du sous arbre de racine Pi



Structure d'un noeud Interne

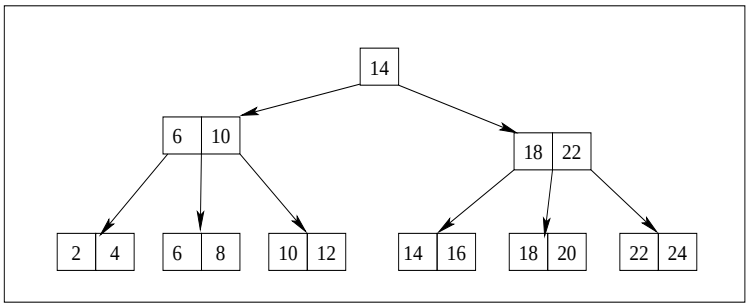
1.12.2 structure d'une feuille



Structure d'une Feuille

1.12.3 Exemple

Arbre B^+ d'ordre x (1 ou 2)

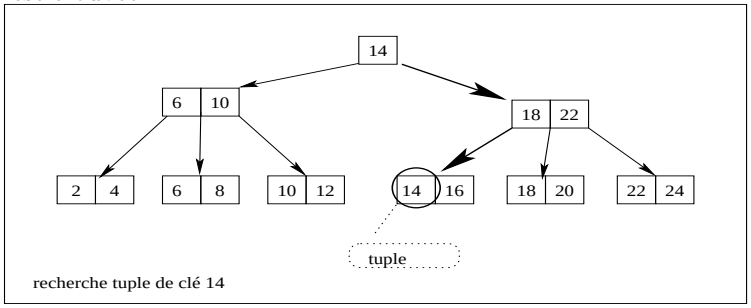


1.12.4 Opérations

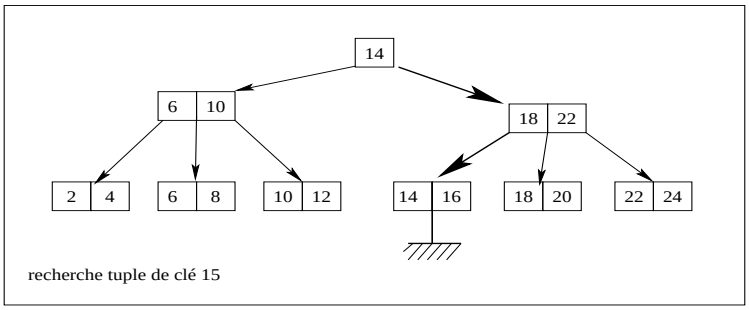
Recherche

La recherche d'un enregistrement de clé k donnée s'effectue par un parcours de l'arbre B^+ depuis la racine vers une feuille :

1. A chaque noeud atteint de l'arbre, la clé k est comparée aux différentes clés contenues dans le noeud en parcourant celles-ci de gauche à droite,
2. Le pointeur retenu, qui permettra d'aller à un noeud fils, est le premier pointeur rencontré qui a à sa droite une clé strictement supérieure à k. Si cette condition n'est pas respectée, on retient le dernier pointeur,
3. La recherche s'achève quand :
   - la clé est trouvée

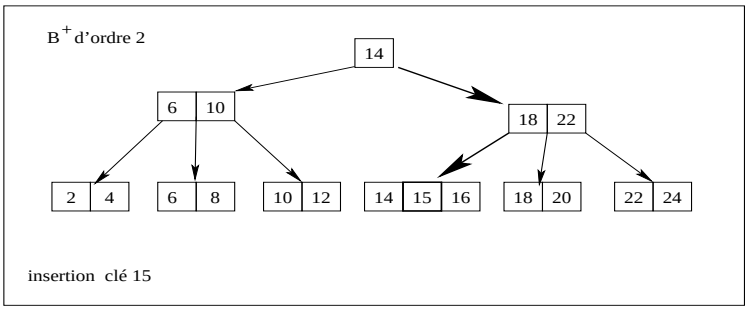


   - ou si le pointeur retenu est nul (en feuille).

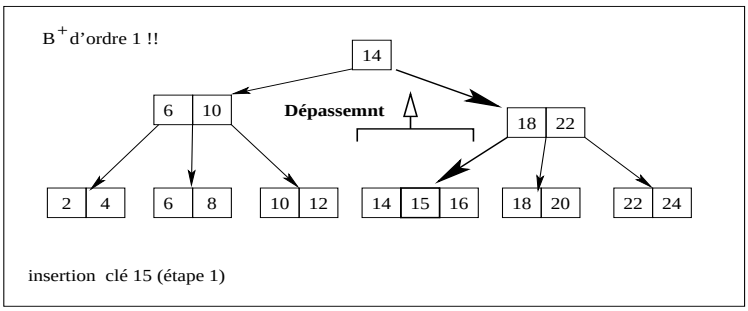


Insertion

1. La feuille f où la clé k doit être insérée à la position p est déterminée en recherchant k dans l'arbre. La clé k est insérée à la position p de la feuille f,



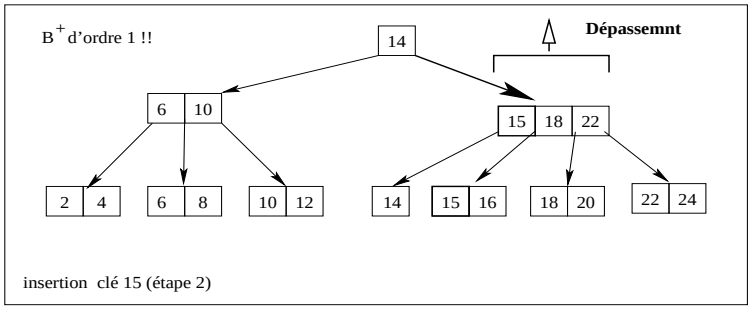
2. Si le nombre de clés de f est supérieur à $2m$ (c'est à dire $2m+1$) :



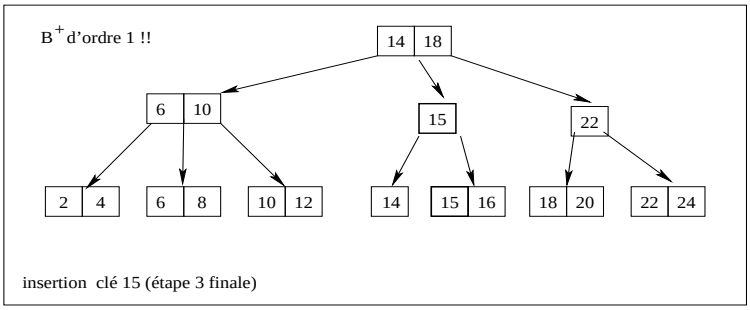
Il y a un dépassement (overflow), le noeud est éclaté (split) comme suit :

(a) Eclater le noeud en deux noeuds suivant les positions $[1..m]$, $m+1$, $[m+2..2m+1]$,

(b) Insérer la clé en position $m+1$ de f dans le noeud père de f,

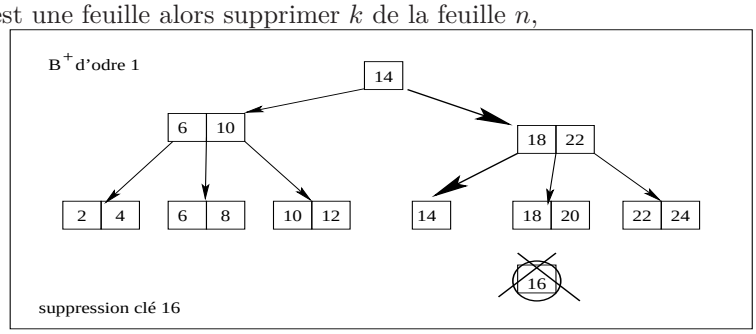


(c) Répéter l'éclatement tant qu'il y a un dépassement (overflow) dans le noeud père.

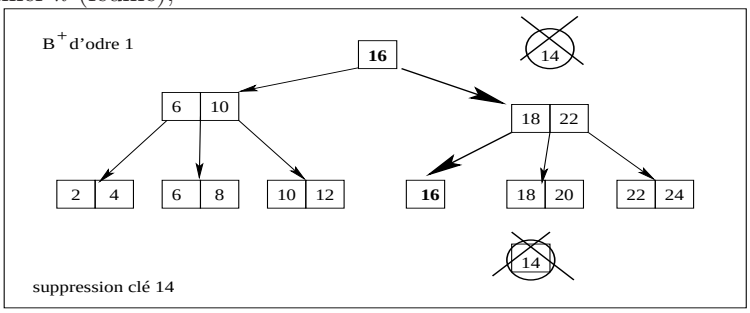


Suppression

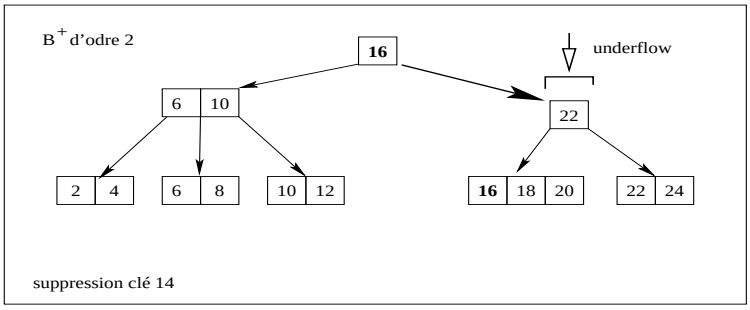
1. Rechercher dans l'arbre B^+, le noeud n où se trouve la clé k à supprimer
2. Si n est une feuille alors supprimer k de la feuille n,



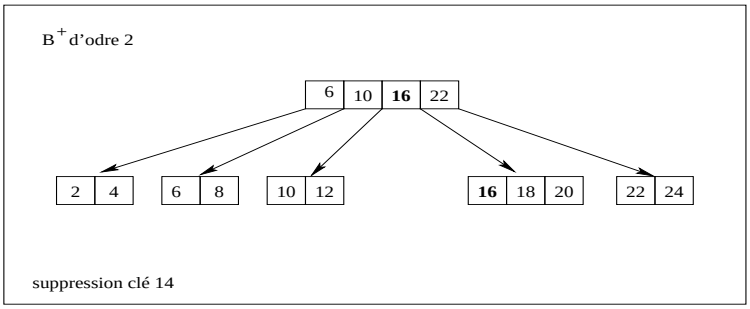
3. Si n n'est pas une feuille alors remplacer k par la clé s successeur de k, supprimer k (feuille),



4. Si le nombre de clés k^n du noeud n est inférieur m (c'est à dire $m-1$) alors il y a underflow, le noeud n est fusionné (merge) comme suit :
   (a) Consulter le frère droite et/ou le frère gauche de n. Soit f celui qui a le minimum de clés.
   (b) Si le nombre de clés k^f de f est égal à m (minimum) (ou $k^f \leq m+1$ si n et f sont des feuilles) alors fusion :
      i. Fusionner le noeud n et le frère f en descendant la clé de partition :

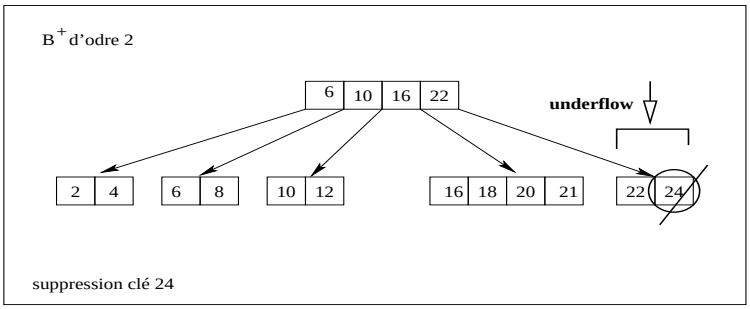


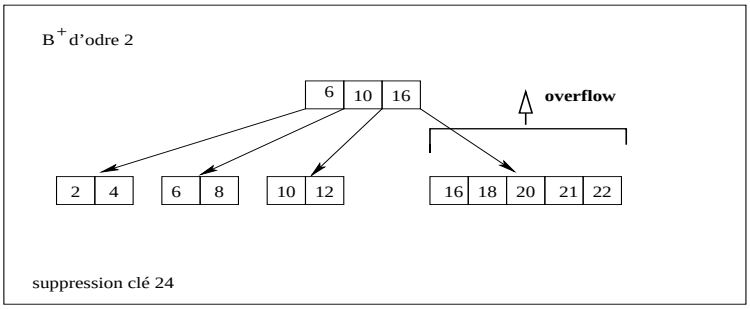
ii. Cette fusion peut engendrer un underflow du pére. Répeter le processus de fusion tant que le père a un underflow : Soit n le père, aller à l'étape 4.



(c) Sinon (on a $k^f > m$ ($k^f > m+1$ si n et f sont des feuilles) aussi $k^f \leq 2m$) fusion puis éclatement :

i. Fusionner le noeud n avec le frère f :





ii. Eclater la feuille obtenue en deux feuilles en remontant dans le noeud père, la clé du milieu : nouvelle clé de partition.