B-Tree

# Présentation

## Intention

Il fallait s’y attendre, garder les situations finales en mémoire est trop volumineux.

La recherche par dichotomie dans un fichier trié est trop lente.

L’utilisation de MySQL se révèle terriblement lente. Et les queries volumineux déclenchent des exceptions de sockets fermés prématurément.

Je me lance dans un stockage arborescent de type B-Tree personnalisé. Je prévois que ce stockage soit permanent sur disque, avec tout de même un cache mémoire des premiers niveaux.

Pour une présentation, voir <https://en.wikipedia.org/wiki/B-tree>

Je veux stocker dans ce fichier toutes les situations finales (à une symétrie près) d’une taille donnée et leurs situations gagnantes associées.

## Rappels

Pour une valeur donnée du paramètre NF, disons *nf*, toutes les situations finales ont *nf* + 1 pierres.

On se limite à l’étude des deux tableaux dits « classique » et « français ». Le nombre de cases du rectangle englobant ces plateaux est 7x7=49, numérotées de 0 à 48 de gauche à droite depuis le haut du rectangle englobant. On peut donc stocker les indices de ces pierres dans un tableau de bytes (octets non signés) triés par indices croissants.

Les situations gagnantes des deux plateaux étudiés ne dépassent pas le nombre de 8 (aux symétries près). Aussi nous ajoutons à chaque situation finale un octet dont chaque bit représente l’une des situations gagnantes. Par construction de la situation finale, ses situations gagnantes sont exactement celles qui sont atteignables depuis la situation finale.

Donc une situation finale est représentée par exactement *nf* + 2 octets.

Le dual de chaque situation finale est une situation de départ, et le flag désigne alors les situations initiales duales des situations gagnantes à partir desquelles on peut atteindre cette situation de départ.

## Vocabulaire

Conventions choisies (puisqu’il y a divergence dans la littérature) :

* Clé : situation stockée dans le B-Tree. Chaque clé est un tableau de *nf* + 2 octets. L’arbre ne contient pas de doublons.
* Nœud : structure stockant des clés et des liens vers d’autres nœuds, ses enfants. Chaque nœud sauf un est l’enfant de exactement un nœud appelé son parent.
* **Ordre : nombre maximal d’enfants que peut avoir un nœud** 
  + Cet ordre est ≥ 2. S’il vaut 2, il s’agit d’un arbre binaire.
* Racine : Seul nœud de l’arbre qui n’a pas de parent.
* Feuille : nœud n’ayant aucun enfant

On rappelle que les feuilles d’un B-Tree sont par construction toutes à une même distance de la racine (voir mécanismes d’insertion et de suppression).

Nous choisissons un ordre du B-Tree, typiquement 25 = 32, qui est le nombre maximal d’enfants que peut avoir un nœud. Le nombre maximum de clés qu’on peut stocker dans un nœud est 25 – 1 =31.

Une valeur de l’ordre telle que 2k – 1 optimise la recherche dichotomique à l’intérieur d’un nœud plein.

## Comparaison des situations

Il faut comparer deux situations finales entre elles pour pouvoir ordonner cet arbre. On obtient le tableau des *nf* + 1 indices des pierres d’une situation triés par indices croissants, et deux situations S1 et S2 sont comparées de la manière suivante :

S[i] représente l’indice d’ordre *i* dans le tableau des *nf* + 1 indices triés, alors

S1 < S2 si et seulement si il existe une valeur *i* tel que S1[*i*] < S2[*i*] et pour tout *j* < *i*, S1[*j*] == S2[*j*].

Donc la situation la plus petite est celle qui présente la première case vide quand on les parcourt de gauche à droite depuis le haut du rectangle englobant.

# Optimisations

## Nombre de situations dans un nœud

Lors de la recherche, il s’agit de déterminer la position d’une situation dans l’ensemble des situations d’un nœud donné, afin de déterminer l’opération à effectuer.

Les situations du nœud étant triées, il est envisagé d’effectuer une recherche dichotomique à l’intérieur du nœud. Pour que cette recherche soit la plus efficace possible, il est utile qu’à chaque étape elle se fasse dans un ensemble impair de situations.

* , recherche finale, comparaison de la situation traitée avec une des situations du nœud
* , lors de la recherche dans un nœud, il est préférable que celui-ci ait un nombre impair de situations. Ainsi le choix de la situation du milieu est équilibré, et s’il n’y a pas égalité, la recherche se poursuit dans un ensemble de taille identique, quel que soit le résultat.

Après résolution, on obtient

Cette optimisation est illusoire lors de la création de l’arbre, l’expérience montrant que statistiquement, les nœuds ne sont remplis au plus qu’à hauteur de 70 %

Mais on peut envisager, lorsque l’arbre est définitivement construit, de créer un arbre qui n’est plus exactement un B-Tree (les chemins de la racine à une feuille ne seront pas tous de même longueur), et où un maximum de nœuds seront saturés.

## Arbre saturé

On rappelle que par construction un B-Tree est tel que tous les chemins de la racine à une de ses feuilles ont même longueur.

Il est saturé si tous ses nœuds sont saturés, i/e. qu’ils contiennent chacun le maximum de clés.

## Nombre maximal de clés

On suppose l’arbre saturé.

Calcul du nombre de nœuds suivant la profondeur de l’arbre.

L’ordre de l’arbre est noté .

Soit *Ln* le nombre de nœuds à la profondeur *n*.

: c’est la racine.

: chaque nœud a *O* enfants.

Et donc

Soit *Nn* le nombre de nœuds au maximum à la profondeur *n* .

Et donc

Le nombre de clés dans un nœud saturé étant , le nombre de clés maximal dans un arbre saturé jusqu’à la profondeur *n* est

## Nombre minimal de clés

On verra que pour un ordre O > 2, seul le nœud racine peut avoir (si l’arbre n’est pas vide) un nombre de clés égal à 1.

Si O = 2i, chaque nœud autre que la racine a au moins i – 1 = (O-1)/2 clés.

Si O = 2i + 1, chaque nœud autre que la racine a au moins i = (O-1)/2 clés.

Donc dans la situation la plus défavorable, on a

* A la profondeur 0, une racine avec une seule clé, donc deux enfants
* A la profondeur 1, 2 nœuds, chacun avec (O-1)/2 clés, et (O-1)/2 + 1 enfants
* Aux profondeurs suivantes, au bout de chaque lien de la profondeur précédente, on a un nœud exactement de la même composition.

On est donc en présence de deux arbres d’ordre (O-1)/2 + 1 de profondeur n-1, plus le nœud racine qui contient une clé.

## Compactage

Dans un premier temps, on calcule les entrées du B-Tree, entrées qu’on insère au fur et à mesure pour obtenir un arbre dont les nœuds ne sont pas saturés. Une fois ce calcul effectué, on souhaite reconstruire l’arbre de manière que le maximum de nœuds soient saturés.

On détermine la profondeur nécessaire de l’arbre compacté :

Soit *N* le nombre de clés total, *p* la profondeur nécessaire et suffisante, alors *p* doit vérifier :

Alors l’idée est de placer *O* – 1 clés dans la racine, et de distribuer les nœuds restant dans les O enfants, chaque enfant recevant « à peu près » le même nombre de clés, soit *1 + (N – O + 1)/O*, soit (*N – O + 1)/O*.

Puis on prend pour chaque enfant cette quantité attribuée pour calculer la distribution de ses clés.

Avec cette méthode, lorsqu’on arrive à une feuille, il se peut qu’elle soit presque vide…

Imaginons *O=6*, et *N=64*

S’il y a clés, l’arbre sera de profondeur 1, et tous les nœuds seront saturés.

Mais s’il y a 36 nœuds, cela nécessitera une profondeur 2.

On place 5 clés dans la racine, il reste 31 nœuds à répartir dans les 6 sous-arbres. Le premier sous-arbre contiendra 6 clés, les autres en contiendront 5.

Donc le 1er sous-arbre nécessite deux niveaux (profondeur 1), les autres un seul.

6

12

18

24

30

1

2

3

4

5

0

31

32

 33

 34

 35

…

Ce n’est plus un B-Tree. Pourquoi pas, après tout, on n’y effectuera plus d’insertions, uniquement des lectures.

# Organisation

On stocke des situations de *nf* + 1 pierres, nécessitant, en ajoutant le champ de bits des situations gagnantes associées, *nf* + 2 octets.

On stocke ces informations dans un fichier contenant chacun des nœuds placés les uns à la suite des autres. On décide qu’il y aura au maximum *ordre* situations dans chaque nœud. Le nœud *racine* est en tête du fichier. Chaque *enfant* est repéré par un offset, entier non signé sur 4 octets. Cet offset est le n° d’ordre du nœud dans le fichier (donc il faut multiplier par la taille de chaque nœud pour connaitre l’offset exprimé en nombre d’octets du début du nœud.

Le nœud est organisé ainsi :

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Child 0 | Situation 0 | Child 1 | Situation 1 | … | Child *ordre* – 2 | Situation *ordre* – 2 | Child *ordre* - 1 |

Mais il faut envisager que le nœud ne soit pas complet. On ajoute donc un octet en début de cette liste pour stocker le nombre d’enfants que le nœud possède. Donc les nœuds ont tous la même taille,

Les données d’un nœud sont écrites de gauche à droite dans le tableau.

# Recherche

On rappelle qu’on gère un arbre de situations toutes distinctes les unes des autres.

Soit à rechercher si la situation S est dans l’arbre.

En partant de *racine*,

* Si S est plus grande que toutes les situations du nœud courant, on retrouve le dernier indice d’enfant du nœud, donc celui de Child *ordre* – 1 si le nœud est plein.
* Sinon Si on ne trouve pas S dans le nœud, on retrouve l’indice de la situation juste supérieure à S.
  + Si le nœud est une feuille, la situation cherchée est absente de l’arbre
  + Sinon, si elle est présente dans l’arbre, elle est alors dans le sous-arbre qui débute par l’enfant pointé par le lien à l’indice déterminé ci-dessus.

# Insertion

## Présentation

Notre arbre a un ordre O donné

Rappels : on note

* le résultat de la division euclidienne (division entière) de O ≥ 2 par 2,
* son complément

Si O est pair,

Si O est impair,



|  |  |
| --- | --- |
| O | ordre de l’arbre, nombre maximal de liens dans un nœud |
| [0 .. O-1] | intervalle des indices des liens |
| NbK = O-1 | nombre maximal de clés dans un nœud |
| [0 .. O-2]=[0 .. NbK-1] | intervalle des indices des clés |

On reçoit une clé K qu’on veut insérer (si elle n’est pas déjà présente) dans l’arbre

* Arbre vide
  + Ne possède qu’un seul nœud, la racine, vide.
  + Celle-ci n’a pas d’enfants
  + On place K en début de racine
  + Celle-ci a maintenant pour nombre actuel de clés : 1
  + Elle n’a pas d’enfants
* Autres cas
  + A tout moment, si K est trouvé, l’algorithme s’interrompt immédiatement
  + On considère alors les situations où K est absent de l’arbre
  + On commence par rechercher K dans la racine (qui n’est donc pas vide)
  + idxInfK : On détermine l’indice de la dernière clé inférieure à K
  + S’il n’y en a pas (K est < à la 1ère clé du nœud), –1 🡪 idxInfK
  + Si le nœud n’est pas une feuille, on continue alors dans le nœud pointé par le lien d’indice idxInfK+1
  + Sinon, soit F cette feuille
    - Si F n’est pas saturée
      * On décale d’une position vers la droite les clés à partir de idxInfK+1
      * On inscrit K à l’indice idxInf+1
      * Inutile de s’occuper des liens, la feuille n’en a pas
    - Si elle est saturée : démarche théorique (dans la pratique, on optimise les opérations)
      * On insère comme ci-dessus K dans F, qui possède maintenant une clé de trop
      * On retire de F la clé Pivot située en son milieu
      * On crée une nouvelle feuille newF
      * On déplace les clés > Pivot dans newF
      * On remonte au parent P la clé Pivot et newF
      * L’objectif est d’insérer Pivot et newF dans le parent
        + Et si le parent est lui aussi saturé, le scinder en deux
        + La difficulté supplémentaire est la gestion de ce nœud qui remonte lors de la scission de l’enfant.
      * Si le père saturé est la racine, on crée deux nœuds dans lesquels on transfère pour moitié les clés (et liens) de la racine, la racine ne conserve que le pivot dont les enfants sont ces deux nœuds.

## Optimisation de la scission

|  |  |
| --- | --- |
| O | ≥ 2, l’ordre peut être pair ou impair |
| N | Nœud saturé |
| NbKeys | =O – 1 : Nombre de clés dans N, indicées de 0 à O – 2 |
| NbChildren | 0 ou O, indicés de 0 à O – 1 |
| K | Clé à insérer dans N |
| idxK | Indice de 0 à NbKeys de l’emplacement théorique où doit être inséré (ou ajouté) K |
| Child | Null ou sous-arbre de clés > K remontant d’une scission précédente |
| NN | Nouveau nœud issu de la scission |
| Kup | Valeur remontant au parent, peut être K ou une ancienne clé de N |

On utilise la division entière : (2n)/2 = (2n+1)/2 = n

Après scission

* N contiendra O/2 clés, indicées de 0 à O/2 – 1
* NN contiendra O – 1 – O/2 clés, indicées de 0 à O – 2 – O/2
  + Quelles que soient la parité de O et la valeur de idxK.
  + Le nombre cumulé de clés de N et NN reste O – 1, puisque on insère une clé K, mais qu’une clé Kup est extraite pour remonter dans le parent.
* N non Leaf contiendra O/2 + 1 children, indicés de 0 à O/2
* Si N non leaf, NN contiendra O – O/2 children, indicés de 0 à O – 1 – O/2

Trois cas sont à envisager suivant la valeur de idxK :

|  |  |
| --- | --- |
| IdxK | Etat après scission |
| idxK = O/2 | Kup=K  NN contient de 0 à O–2–O/2 les O–1–O/2 clés de N auparavant indicées de O/2 à O–2  Child est placé en début des liens de NN  NN contient de 1 à O–1–O/2 les O–1–O/2 liens de N auparavant indicés de O/2+1 à O–1  N conserve ses O/2 premières clés indicées de 0 à O/2–1  N conserve ses liens indicés de 0 à O/2 |
| idxK < O/2 | K et Child sont dans N  Kup=N.Keys[O/2 – 1]  NN contient de 0 à O–2–O/2 les O–1–O/2 clés de N auparavant indicées de O/2 à O–2  NN contient de 0 à O–1–O/2 les O–O/2 liens de N auparavant indicés de O/2 à O–1  Les O/2–1–idxK clés d’indice idxK à O/2–2 de N sont décalées de 1 case à droite  Les O/2–2–idxK liens d’indice idxK+1 à O/2–1 de N sont décalés de 1 case à droite  Keys[idxK]=K  Children[idxK+1]=child |
| idxK > O/2 | K et Child sont dans NN  Kup=N.Keys[O/2]  NN contient de 0 à idxK–2–O/2 les idxK–1–O/2 clés de N auparavant indicées de O/2+1 à idxK–1  NN contient de 0 à idxK–1–O/2 les idxK–O/2 liens de N auparavant indicés de O/2+1 à idxK  NN.Keys[idxK–1–O/2]=K  NN.Children[idxK–O/2]=child  NN contient de idxK–O/2 à O–2–O/2 les O–1–idxK clés de N auparavant indicées de idxK à O–2  NN contient de idxK+1–O/2 à O–1–O/2 les O–1–idxK liens de N auparavant indicés de idxK+1 à O–1 |

## Algorithme non récursif

|  |  |
| --- | --- |
| Ordre | nombre maximal de liens d’un nœud, indicés de 0 à Ordre – 1 |
| Ordre-1 | nombre maximal de clés d’un nœud, indicées de 0 à Ordre – 2 |
| Leaf | nœud sans liens |
| Saturé | nœud avec le maximum de clés, et donc si ce n’est pas une Leaf, le maximum de liens. |

**bool Add(K) :**

// Soit à insérer la clé K dans l’arbre, renvoie false sans insérer si K déjà présente

If Root.NbKeys == 0

Root.Key[0] = K

return true

Stack stack

Node node = Root

for( ;; )

if(node.Find(K, out int idx))

return false

// node : noeud actuel, idx : indice au-delà duquel insérer la clé

stack.Push(node, idx)

if(node.Leaf)

break

node = node.Children[idx+1]

// K est absente, et node est une Leaf, en fin de stack

Node child = null // En cas de split

for( ; ! stack.IsEmpty ; )

(Node node, int idx) context = stack.Pop()

if(node.NbChildren < Order)

Insert(context.node, context.idx, K, child)

break

Split(node, idx, ref K, ref child)

if(node==Root)

// On tient à conserver la même @ pour Root

// donc on ne se contente pas de créer un nouveau Root

BuildNewRoot(K, child)

return true

**bool Node.Find(clé K, out int idx) :**

recherche dichotomique de K dans Node

renvoie true si trouvé, et idx est alors son indice

sinon renvoie false, et idx est l’indice de la plus grande clé < K, ou -1 si K est < à toute clé de Node

**void Insert( Node node, int idx, clé K, Node child) :**

node.Keys.InsertAt(idx+1, K)

if(child != null)

node.Children.InsertAt(idx+2, child)

**void Split(Node node, int idx, ref clé K, ref Node child) :**

// Voir paragraphe [Optimisation de la scission](#_Optimisation_de_la)

**void BuildNewRoot(clé K, Node droite) :**

Node gauche = new Node()

gauche.Keys.AddRange(Root.Keys)

Root.Keys.Clear()

Root.Keys.Add(K)

if( !Root.Leaf)

gauche.Children.AddRange(Root.Children)

Root.Children.Clear()

Root.Children.Add(gauche)

Root.Children.Add(droite)

# Suppression

## Feuille ou non ?

Les clés étant situées dans tous les types de nœuds, la clé K à supprimer peut être dans une feuille ou non. Si elle n’est pas dans une feuille, elle est dans un nœud N avec enfants, et il existe dans l’arborescence G à gauche de cette clé K une plus grande clé K- < K. C’est celle située dans la feuille F à l’extrémité du chemin partant de l’enfant de gauche de K puis en restant à droite. Supprimons K- de F, remplaçons K par K- dans N. L’ordre est respecté, le nombre de clés de N aussi. Seul le nombre de clés de la feuille à l’extrémité du chemin suivi a diminué de 1 unité. Tout se passe comme si on avait supprimé une clé dans cette feuille et non dans N.

K

K-

K-

## Dépeupler – Repeupler

Maintenant, on a vu qu’ajouter une clé à un nœud saturé amenait à couper ce nœud en deux, chacun conservant la moitié au moins des clés. Inversement, supprimer une clé d’une feuille peut la dépeupler en deçà de cette moitié. On s’astreint à ne conserver que des nœuds au moins à moitié remplis, donc on va équilibrer cette région de l’arbre. La feuille dépeuplée F a un parent P, grâce auquel elle est connectée à au moins une feuille ou même deux autres feuilles FG et FD de part et d’autre. Si l’une d’elles (par exemple la feuille de droite FD) a suffisamment de clés, on lui emprunte sa clé d’extrémité qu’on remonte dans P, et la clé chassée de P est ajoutée à F, qui est de nouveau peuplé à moitié. Evidemment, si la feuille est aussi la racine, on ne peut effectuer cette opération puisque la racine n’a pas de parent.

K-

K+

K+

K-

Cela reste vrai si le nœud dépeuplé n’est pas une feuille, mais a lui-même des enfants.

K-

K+

K+

K-

b

c

d

e

b

c

d

e

## Fusionner

On examine maintenant le cas où FG et FD sont aussi pauvres en clés. On choisit l’un d’eux, par exemple FG, et on va reconstituer un nœud composé de FG, N, et de la clé qui les sépare, et qu’on retire du parent. Cela réduit le nombre de clés du parent, qu’on essaie donc d’équilibrer de la même manière, par transfert de clé ou fusion de nœuds.

2

4

1

0

3

a

b

c

d

e

f

4

f

1

0

2

3

a

b

c

d

e