Scalable Distributed Compact Trie Hashing

Proposé par D.E ZEGOUR

Trie hashing (TH)

- **Le hachage digital (81) est l'une des méthodes les plus rapides pour l'accès au fichiers monoclé, ordonnés et dynamiques.**

- **La technique utilise une fonction de hachage variable représentée par un arbre digital qui pousse et se rétracte en fonction des insertions et suppressions.**

- **Caractéristiques principales :**

✓ **l'arbre réside en mémoire pendant l'exploitation du fichier.**

✓ **6 Octets / case**

✓ **Un accès au plus pour retrouver un article**

Compact trie hashing (CTH)

- **Plusieurs manières de représenter la fonction d'accès en mémoire. ➔ Objectifs : doubler les fichiers adressés pour le même espace mémoire utilisé par la représentation standard.**
- **L'idée : représenter les liens de manière implicite au détriment d'algorithmes de maintenance légèrement plus long.**
- **Consommation : 3 octets par case du fichier.**

  → Ce qui permet d'adresser des millions d'articles avec très peu d'espace mémoire.
- **Intéressante pour un environnement distribué.**

Distributed compact trie hashing

- Nous proposons une distribution de CTH relativement aux propriétés des Sdds :

- ❑ Distribution des cases du fichier sur les serveurs(à raison d'un serveur par case)

- ❑ Pas de site maître

- ❑ Aucun dialogue entre les clients.

Plan à suivre

- **Compact trie hashing**

- **Distribution de la méthode sur plusieurs sites.**
- **Illustration de la méthode**

- **Algorithmes de recherche et insertion**
- **Requête à intervalle et suppression**

- **Variantes**
- **Conclusion**

Compact trie hashing
Mécanisme de construction

1. a ce dx ef sont insérées dans la case 0.

L'arbre : | 0

Collision sur case 0

Clé maximale ='||…'

a ce dx ef h

Séquence de division : d

K = 1 ; I = 0

a
ce
dx
ef

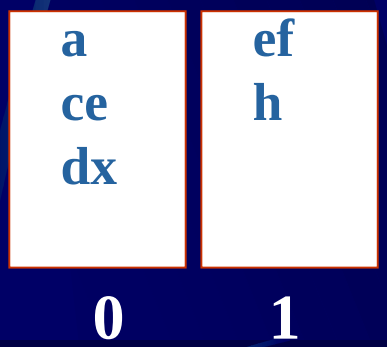
0

Capacité de la case = 4

Compact trie hashing
Mécanisme de construction

2. Après insertion de h

L'arbre : d 0 | 1



Compact trie hashing
Mécanisme de construction

3. Insertion de x y

kx

L'arbre : **d 0 | 1**

0	1
a ce dx	ef h x y

Collision sur case 1

Clé maximale =‘||…’

ef h k x y

Séquence de division = ‘k’

K = 1 ; I = 0

Compact trie hashing
Mécanisme de construction

4. Après insertion de kx

L'arbre : d 0 k 1 | 2

0	1	2
a ce dx	ef h kx	x y

Compact trie hashing
Mécanisme de construction

5. Insertion de fe

L'arbre : d 0 k 1 | 2

0	1	2
a ce dx	ef fe h kx	x y

Collision sur case 1

Clé maximale = ‘k|…’

ef fe h hx k

Séquence de division = ‘h’

K = 1 ; I = 0

Compact trie hashing
Mécanisme de construction

6. Après insertion de hx

L'arbre : d 0 h 1 k 3 | 2

0	1	2	3
a ce dx	ef fe h hx	x y	kx

Compact trie hashing
Mécanisme de construction

6. Insertion de hy

L'arbre : d 0 h 1 k 3 | 2

0	1	2	3
a ce dx	ef fe h hx	x y	kx

Collision sur case 1

Clé maximale =‘h|…’

ef fe h hx hy

Séquence de division = ‘h_’

K = 2 ; I = 1

Compact trie hashing
Mécanisme de construction

7. Après insertion de hy

L'arbre : d 0 h _ 1 4 k 3 | 2

0	1	2	3	4
a ce dx	ef fe h	x y	kx	hx hy

Compact trie hashing
Mécanisme de construction

8. Insertion de yya yyb

L'arbre : d 0 h _ 1 4 k 3 | 2

0	1	2	3	4
a ce dx	ef fe h	x y yya yyb	kx	hx hy

Collision sur case 2

Clé maximale =‘||…’

x y yya yyb yyc

Séquence de division = ‘yya’

K = 3 ; l = 0

Compact trie hashing
Mécanisme de construction

9. Après insertion de yyc

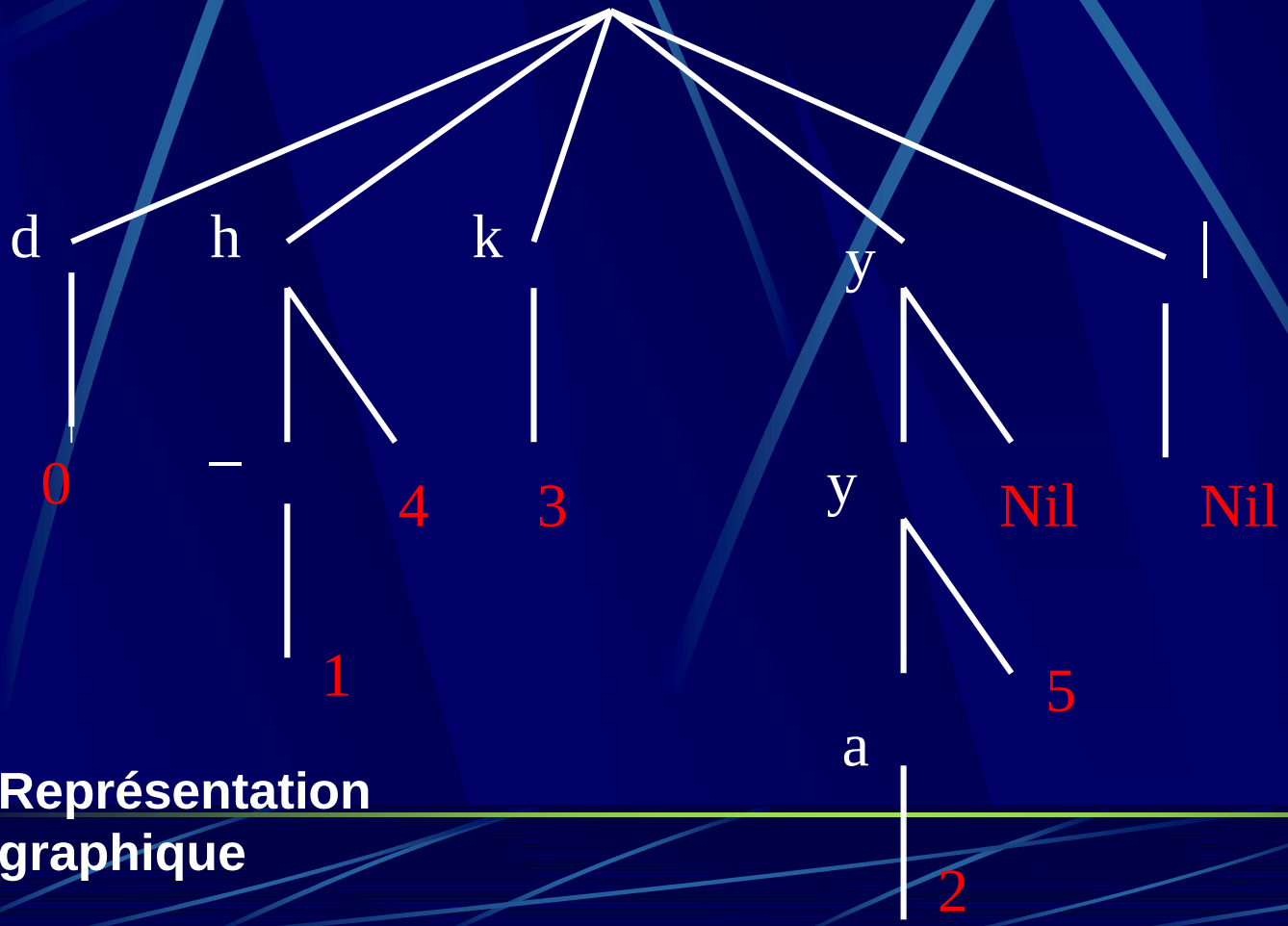
L'arbre : d 0 h _ 1 4 k 3 y y a 2 5 Nil | Nil

0	1	2	3	4	5
a ce dx	ef fe h	x y yya	kx	hx hy	yyb yyc

Compact trie hashing

Mécanisme de construction

ARBRE : d 0 h _ 1 4 k 3 y y a 2 5 Nil | Nil



Représentation graphique

- **La concaténation des digits sur une branche de l'arbre représente la clé maximale de la case figurant dans la feuille**
- **forme préordre**
- **Suite de nœuds internes et externe**

Compact trie hashing
Propriété et Performances

- **Propriété**
- **Les cases sont ordonnées de gauche à droite**

- **Algorithmes en mémoire :**
  - ✓ **Recherche : N/2 en moyenne**
  - ✓ **Insertion : N/2 décalages en moyenne**
  - ✓ **Encombrement : 3 octets / case en moyenne.**

- **algorithmes sur disque :**
  - ✓**1 accès au plus pour retrouver un article**

Scalable Distributed Compact trie hashing: Concepts

- Au niveau de chaque client il y a un arbre digital partiel
- Tout client commence avec un arbre vide (| 0)
- Mise à jour progressive de l'arbre du client.

- Au niveau de chaque serveur il y a
  - ✓ un arbre digital partiel
  - ✓ une case contenant les articles du fichier
  - ✓ une liste d'intervalles [Inf_i, Sup_i] dont le premier correspond au serveur réel et les autres, s'ils existent, aux serveurs virtuels.

- L'arbre digital au niveau du serveur garde la trace de tous les éclatements sur ce serveur.
- L'expansion du fichier se fait à travers les collisions.
- A chaque collision il y a distribution du fichier (du serveur éclaté) sur un nouveau serveur (Scalabilité) .

Distributed Compact trie hashing

Concepts

- **Quand une case éclate , il y a**

✓**Extension de l'arbre du serveur**

✓**Initialisation d'un nouveau serveur avec un arbre vide**

✓**Partage des articles entre l'ancien et le nouveau**

✓**Les intervalles sont mis à jours**

✓**Création éventuelle de plusieurs serveurs virtuels (Nœuds Nil)**

(Le processus d'éclatement est donné plus loin)

- **Initialisation du système**

Initialiser le serveur 0 avec

Case vide ; Intervalle : > λ , <= Λ ; Arbre : | 0

Distributed Compact trie hashing

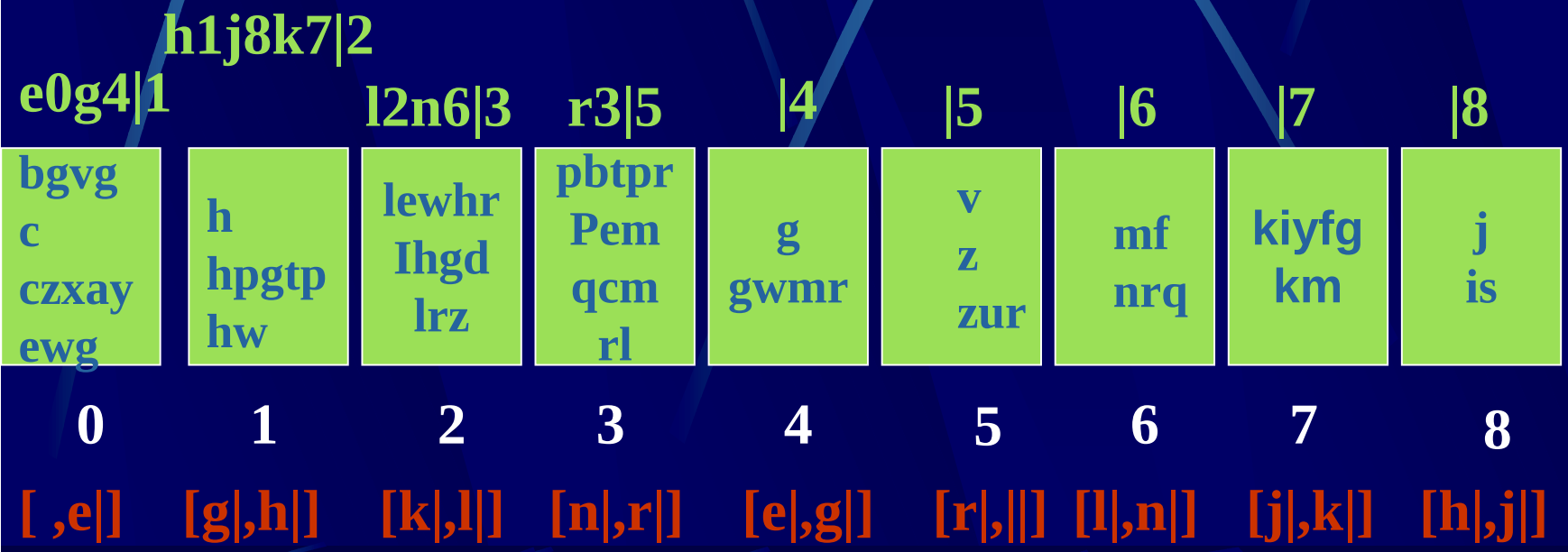
Illustration

Client1 : e 0 g 4 k 1 | 2

Client2 : g 0 k 1 n 2 r 3 | 5

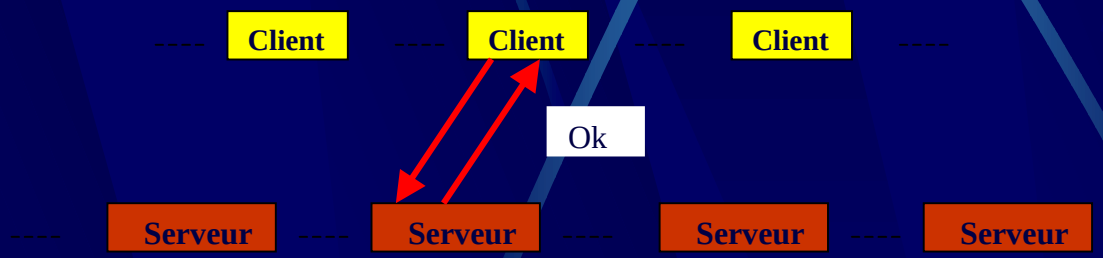
Client3 : g 0 k 1 n 2 | 3

Client4: g 0 k 1 n 2 | 3



Distributed Compact trie hashing
Transformation (Client, Clé) → Serveur

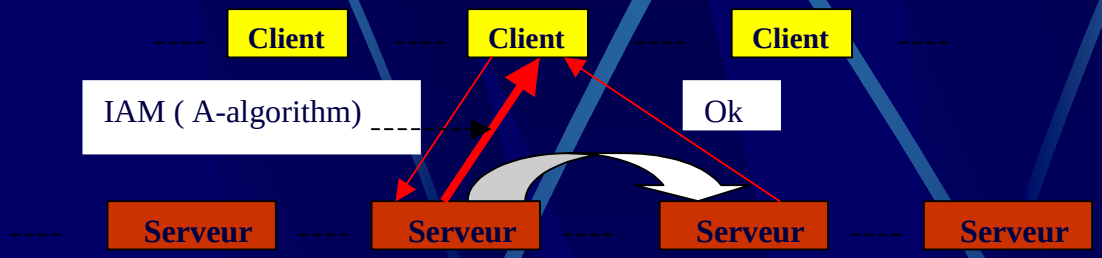
- **Bon adressage**



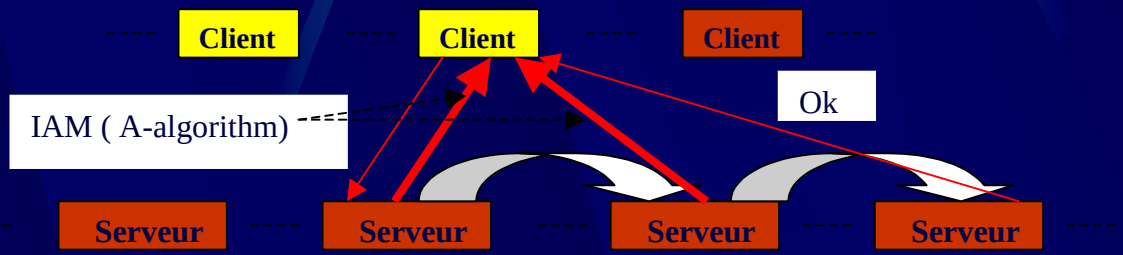
Distributed Compact trie hashing

Transformation (Client, Clé) → Serveur

- **Mauvais adressage (avec 1 forward)**

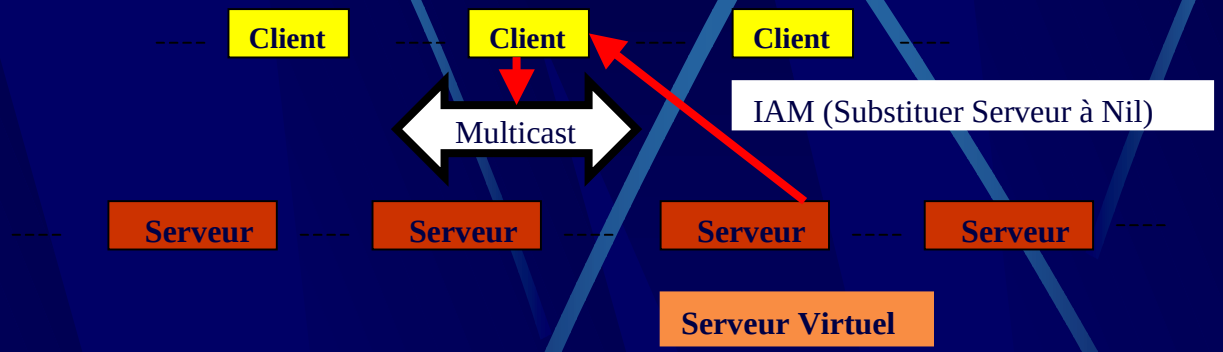


- **Mauvais adressage (avec 2 forward)**

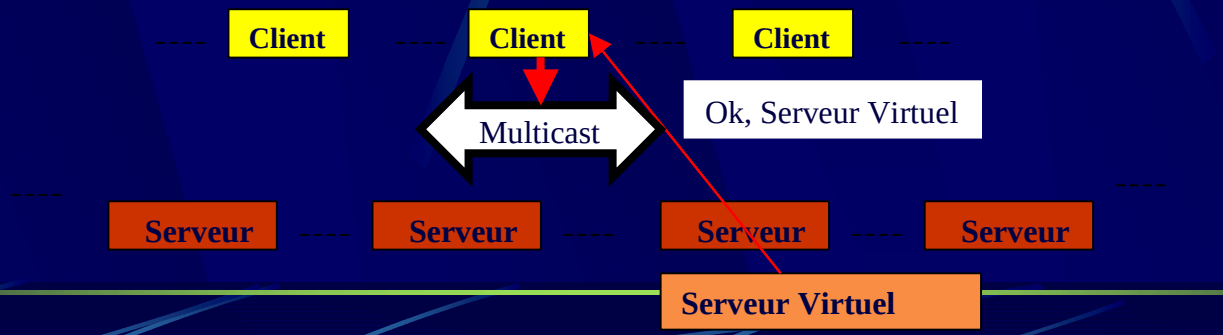


Distributed Compact trie hashing
Transformation (Client, Clé) → Serveur

- **Cas d'un nil et un serveur réel répond**

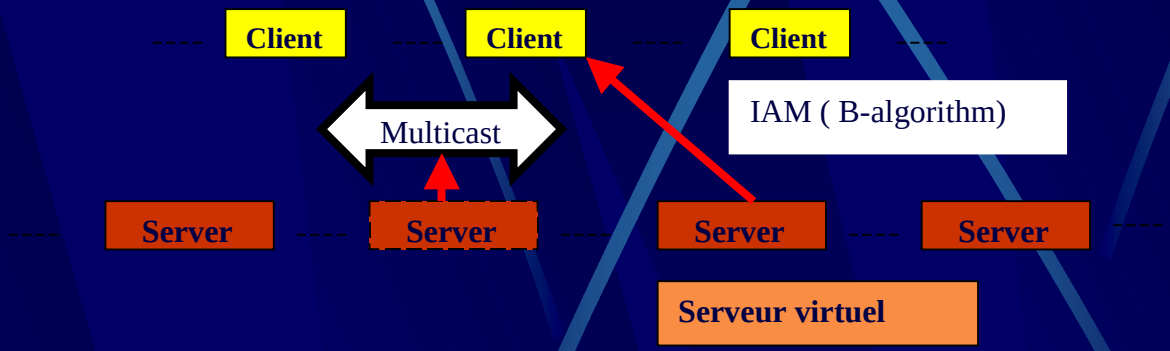


- **Cas d'un nil et un serveur Virtuel répond**

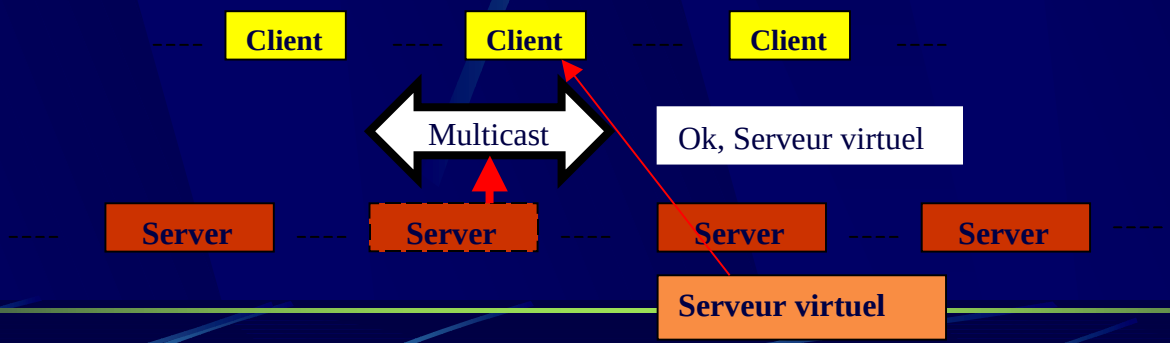


Distributed Compact trie hashing
Transformation (Client, Clé) → Serveur

- **Une impasse est détectée sur un serveur et un serveur réel répond**



- **Une impasse est détectée sur un serveur et un serveur virtuel répond**



Distributed Compact trie hashing
Transformation (Client, Clé) → Serveur

Cas de bouclage

- **Supposons qu'un client avec l'arbre c 0 e t 3 5 | 2 recherche la cle 'h' et supposons que le serveur 2 contient l'arbre w 2 | 7 avec l'intervalle >s, <=w.**

- **Le module de recherche sélectionne le serveur 2. Comme 'h' n'est pas dans l'intervalle de ce serveur, il y a remplacement dans l'arbre du client, ce qui donne**

C 0 e t 3 5 w 2 | 7.

- **La ré application de CTH sur 'h' nous redonne 2 et le processus de recherche rentre dans une boucle infinie.**

Distributed Compact trie hashing

Terminologie

Soit l'arbre : « a r 0 9 b 5 i e g 3 k l 8 4 9 12 k … »

Sous arbres de niveau 0 : « a r 0 9 », « b 5 », « i e g 3 k l 8 4 9 12 », …

« a r 0 9 » est le sous arbre de racine a.

Sous arbre de niveau 1 : « r 0 » , « e g 3 k l 8 4 9 12 », …

Sous arbres de niveau 2 : « g 3 », « k l 8 4 »

Dans le sous arbre « i e g 3 k l 8 4 9 12 » les branches de gauche à droite sont : 'i e g 3', 'i e k l 8', 'i e k 4', 'i e 9' et 'i 12'

On définit les fonctions : First_path et Next_path

Forme d'une branche : b = « $C_0C_1....C_L$ M »

C_i est un digit et M une adresse de serveur

First_digit(b) = C_0 ;Digits(b) = $C_0C_1....C_L$; Leaf(b) = M

Digits(b) est la clé maximale de Leaf(b).

Distributed Compact trie hashing

Algorithme A

Algorithme A : modifier l'arbre du client en fonction de l'arbre du serveur

Soit Cm = $C_0C_1...C_k$ la clé maximale du serveur recherchée (m)

ind_m l' indice de m dans l'arbre

ind_d l'indice de C_0 dans l'arbre

(i) Insérer à la position ind_d tous les sous arbres du serveur de racines strictement inférieures à C_0.

Ind_d et ind_m sont modifiés en conséquence. Si N est le nombre de nœuds de tous les sous arbres, alors (+ N)

Appliquer (ii) ou (iii) selon le cas.

(ii) Cas où C_0 n' existe pas dans l'arbre du serveur

- **déterminer le prochain serveur dans l'arbre du serveur soit e, puis remplacer m par e.**

Distributed Compact trie hashing

Algorithme A

(iii) Cas où C_0 existe dans l'arbre du serveur comme racine d'un sous arbre:

Ni : nombre de nœuds internes qui précédent m dans l'arbre du client

Ne : = Longueur(Cm) - Ni, (le nombre de digits à ignorer dans le serveur

Coté serveur :

dans le sous arbre de racine C_0, déterminer toutes les branches b' des sous arbres de niveau Ne telles que Digits(b') + '\|' <= ($C_{Ne}C_{Ne+1},\ldots C_k$) + '\|' . Soit b'_{Last} la dernière branche.

Coté client :

a) Si Ni = 0, remplacer m par les branches b' trouvées.

b) Soit L la longueur de b'_{last} ; Remplacer C_{Ne} ….C_{Ne+L-2} et le sous arbre de racine C_{Ne+L-1} par les branches b'

Distributed Compact trie hashing

Algorithme A

Client : j 3

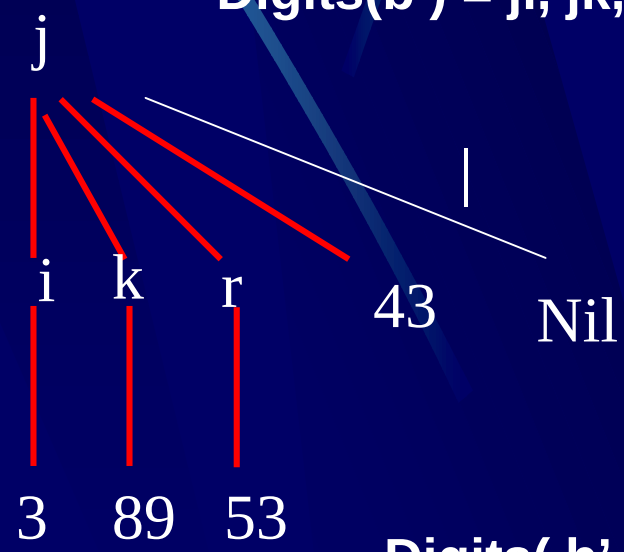
Cm = j ; Ni = 1 ;Ne = 0 ;

Digits(b)= ‘j’

→ Client : j i 3 k 89 r 53 43

Arbre du serveur :
j i 3 k 89 r 53 43

Digits(b’) = ji, jk, jr, j



Digits(b'_{last}) = j

Distributed Compact trie hashing

Algorithme A

Arbre client :

… m _ 56 l 38 t 25 50 …

Cm :=‘mt’
Sous arbre de racine C’$_{Ne}$

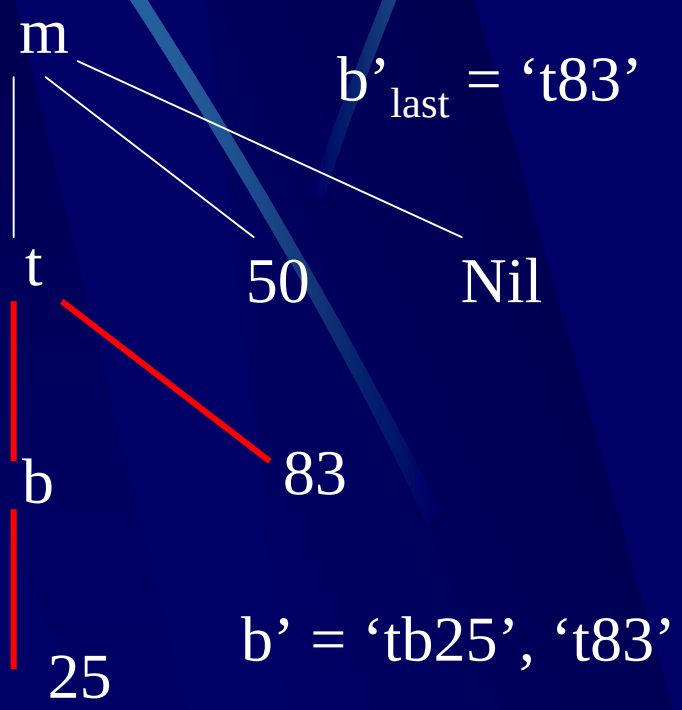
Ni = 1; Ne = 1

b = ‘t25’

Arbre client après

… m _ 56 l 38 t b 25 83 50 …

Arbre du serveur 25 : mtb25 83 50 | Nil



Distributed Compact trie hashing

Algorithme B

Algorithme B : Modifier l'arbre du client selon I (résultat du multicast)et la clé maximale Cmsup du serveur I

(i) déterminer Ni, le nombre de nœuds internes qui précèdent Ind_m. Nombre de nœuds à ignorer de Cmsup : Ne = Long(Cm) - Ni

(ii) L := 0;

Si Ni <> 0, déterminer le nombre de digits communs, soit L, dans Cmsup à partir de Ne+ 1 et dans l'arbre du client commençant à la position ind_m – Ni.

Nombre de nœuds à créer N: = 2 * (Long (Cmsup) - (Ne+ L))

(iii) insérer à la position Ind_m - Ni + L, les digits manquants: Cmsup[Ne + 1 +L], Cmsup[Ne + 1 +L + 1]... Il existe (Long(Cmsup) - (Ne+L).

Suivi par M, suivi par (Long(Cmsup) - (Ne+L) – 1 nœuds Nil.

Distributed Compact trie hashing

Algorithme B

Exemple / Algorithme B

Client avant : … o e 24 h 69 r 18 66 ….

Ni = 0 ; Ne = 1

Cm = or; M = 18

Multicast = 53 ; Borne supérieure = oj

Client après : o e 24 h 69 j 53 r 18 66 ...

Distributed Compact trie hashing

Algorithme B

Exemple / Algorithme B

Client avant : ... x h 46 t 35 25 ....

Ni = 0 ; Ne = 1 ;

Cm = x ; M = 25

Multicast = 78 ; Borne supérieure = xv

Client après : ... x h 46 t 35 v 78 25 ....

Distributed Compact trie hashing

Recherche / Insertion

Appliquer l'algorithme de transformation (Client, cle) → m

Si m est un serveur virtuel (cas du Nil)

(i) Transformer le serveur virtuel en un nouveau serveur réel, N.

(ii) Initialiser ce serveur avec une case contenant la nouvelle clé, un arbre vide et le même intervalle que celui du serveur virtuel

(iii) Ajuster l'arbre du serveur en remplaçant Nil par N.

(iv) Ajuster l'arbre du client en remplaçant Nil par N

Distributed Compact trie hashing

Recherche / Insertion

Si m est un serveur réel

(i)Si x n'est pas dans la case et case non pleine insérer tout simplement x dans la case et l'algorithme se termine.

(ii) Si x n'est pas dans la case et celle-ci est pleine il y a collision.

- ✓ **Éclatement du serveur**

Distributed Compact trie hashing
Recherche/Insertion

Processus d'éclatement :

- **Former la séquence et déterminer la séquence de division Seq**
- **Éclater la case du serveur I en 2 selon Seq**
- **l'ancien serveur I contient les clés <= Seq**
- **le nouveau serveur, soit J, contient le reste**

  Int(I) ←] inf(I) , Seq]

  Int(J) ←] Seq , Min(Sup(I), Seq $_{-1}$]

  l'arbre du serveur nouvellement créé est ' | J'

- **Générer éventuellement des serveurs virtuels vides** (voir plus loin).

- **Modifier l'arbre du serveur éclaté** (voir plus loin)

Distributed Compact trie hashing
Recherche/Insertion

Processus d'éclatement : Générer des serveurs virtuels selon le processus

J ← 1; Sauv:= Min(Seq_{-1}, Sup(m)] // m : Serveur éclaté
Tq j <= (k-i-1) et sauv <= Sup(m)
    Créer un serveur virtuel avec l'intervalle] sauv, $Seq_{-(j+1)}$]
    Sauv := $Seq_{-(j+1)}$
    J:=j+1
Ftq

Seq_{-j} c'est Seq sans les j derniers digits

Distributed Compact trie hashing
Recherche/Insertion

Serveurs virtuels

- **Un traitement particulier est entrepris chaque fois qu'il y a création de nœuds NIL. But : maintenir la partition**
- **Nous devons associer un serveur vide à chaque nœud Nil dont l'intervalle est inclus dans la limite du serveur éclaté (serveur virtuel)**
- **Un serveur virtuel contient uniquement l'intervalle associé au nœud Nil**
- **Les serveurs virtuels peuvent être gardés dans le serveur éclaté et seront utilisés dans les opérations de Multicast/Broadcast**
- **Un serveur virtuel devient un serveur réel quand un article y est inséré.**
- **Le nombre de serveurs virtuels est très faible (5% pour des insertions aléatoires).**

Distributed Compact trie hashing
Recherche/Insertion

Expansion de l'arbre du serveur :

- **Tenir compte du fait que l'on éclate tout en restant dans l'intervalle du serveur éclaté.**

- **Traitement particulier dans le cas ou Sup(m) < Seq_{-1} + '|' : modification de l'arbre en conséquence**

Distributed Compact trie hashing
Recherche/Insertion

Expansion de l'arbre du serveur :

Cas k-i > 1 :

- **Remplacer m par Nil**
- **Si Sup(m) < Seq_{-1} + '|'**

  Considérer $S=d_1d_2…d_j$ la séquence de digits dans Sup(m) de laquelle on élimine le préfixe commun avec Seq_{-1}(premiers digits existant dans Seq_{-1})

- **Insérer à la position Ind_d la séquence C'_{i+1} C'_{i+2} .....C'_k m [$d_1d_2…d_j$]M [$Nil_1Nil_2..Nilj$]Nil_1 Nil_2 .....Nil_{k-l-2}**
- **2(k-i) + 2j nœuds sont ajoutés, M étant le prochain serveur alloué pour le fichier.**

Distributed Compact trie hashing
Recherche/Insertion

Exemple 1

Considérer un serveur (41 par exemple) avec l'arbre vide '| 41' et l'intervalle]'nq' , 'n|'].

Supposons une collision sur ce serveur avec la séquence de division 'nzc'.

L'arbre du serveur devient : 'n z c 41 61 Nil | Nil'.

2 nœuds Nil sont créés.

L'intervalle du serveur 41 devient]'nq', 'nzc'] .

Le nouveau serveur, 61 par exemple, aura l'intervalle]'nzc', 'nz'] .

Bien que, 2 nœuds Nil sont crées, seulement un serveur virtuel est créé avec l'intervalle]'nz', 'n|']. L'autre avec l'intervalle ['n|', '||'], a été éliminé car non inclus dans l'intervalle du serveur éclaté (41)

Distributed Compact trie hashing
Recherche/Insertion

Exemple 2

Considérer un serveur (772 par exemple) avec l'intervalle]'vx', 'vzj'] et l'arbre vide '| 772 '.

Supposons que la séquence de division Seq est 'vy'.

Comme Sup(772) est inférieur à Seq_{-1} (cad. 'v|'), alors

Si on ne fait pas attention, l'arbre devient

'v y 772 1176 | NIL'. 1176 est le nouveau serveur ajouté.

INCOMPATIBILITE !!! (car intervalle du nouveau serveur =] 'vy', 'v '] d'après l'arbre)

Ce qu'il faut faire

l'arbre du serveur 772 est modifié comme suit : 'v y 772 z j 1176 NIL NIL | NIL'. 1176 est le nouveau serveur ajouté.

L'ancien serveur , cad. 772, aura]'vx', 'vy'] comme nouveau intervalle

Le nouveau serveur, 1176, aura comme intervalle]'vy', 'vzj'].

'vzj' étant le Min entre Seq-1 + '|' qui est 'v|' et Sup(772) qui est 'vzj'.

Distributed Compact trie hashing
Requête à intervalle

Requête à intervalle : lister les clés dans [X, Y]

- ***1. Aller au serveur pouvant contenir X (soit M)***

- ***2. Aller vers tous les serveurs S visibles (descendants) de M et lister toutes les clés dans l'intervalle désiré (Récursivement)***

- ***Si non couverture de l'intervalle,***

- *soit utiliser le* Multicast pour avoir le serveur ayant une borne inférieure égale à la dernière borne supérieure (serveur de reprise)

- soit utiliser le serveur Père de M pour avoir le serveur de reprise

- ***Poser M := Serveur de reprise et Reprendre en 2)***

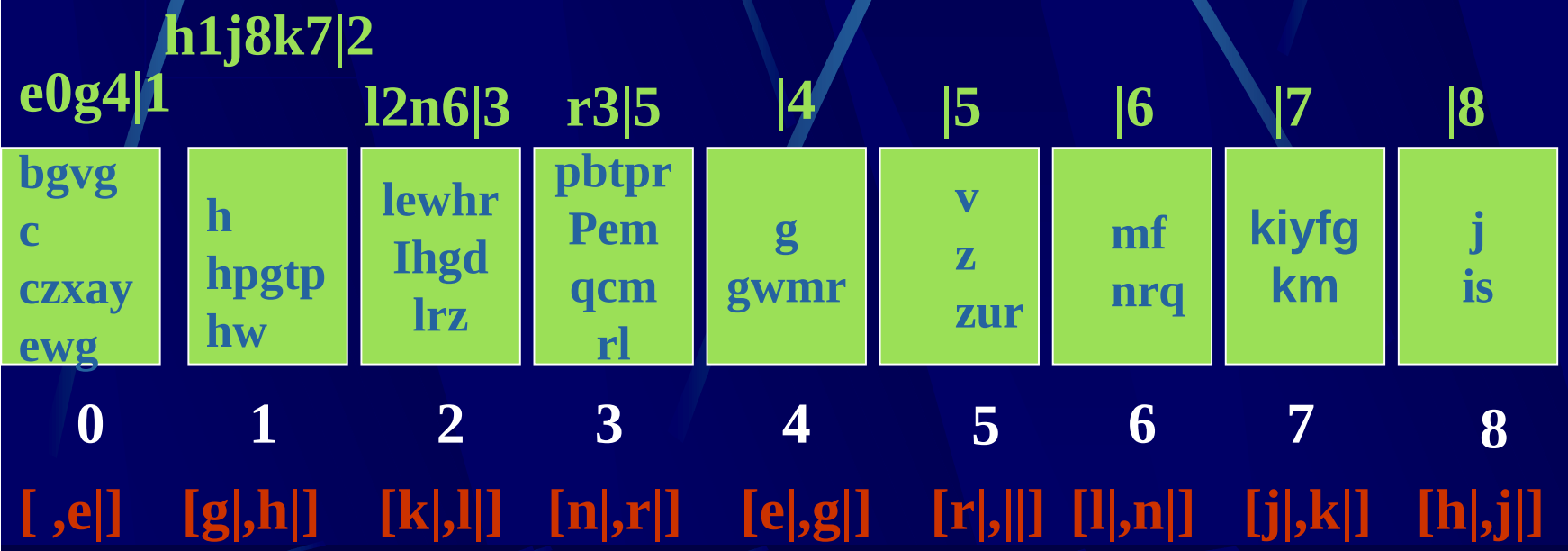
Distributed Compact trie hashing

Suppression

Supprimer ‘h’ par client 2

Client1 : e 0 g 4 k 1 | 2
Client2 : g 0 k 1 n 2 r 3 | 5

Client3 : g 0 k 1 n 2 | 3
Client4: g 0 k 1 n 2 | 3



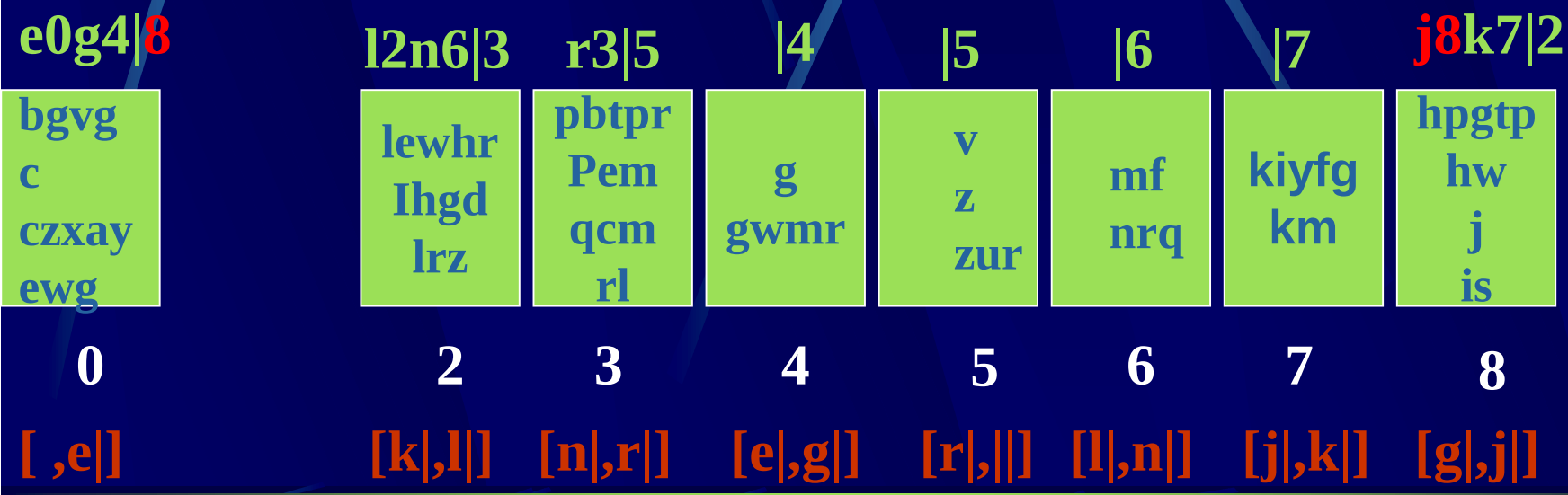
Distributed Compact trie hashing

Suppression

Après suppression de ‘h’

Client1 : e 0 g 4 | 2
Client2 : g 0 n 2 r 3 | 5

Client3 : g 0 n 2 | 3
Client4: g 0 n 2 | 3



Distributed Compact trie hashing
Performance

- **Facteur de chargement : 70 %**
- **Nombre de multicast presque nul et indépendant du nombre d'articles insérés**
- **5 à 6 nœuds sont transférés en moyenne et indépendant du nombre d'articles insérés**
- **Réalisation des requêtes à intervalle et de la suppression sans recours au multicast (si Chaînage vers le père existe)**

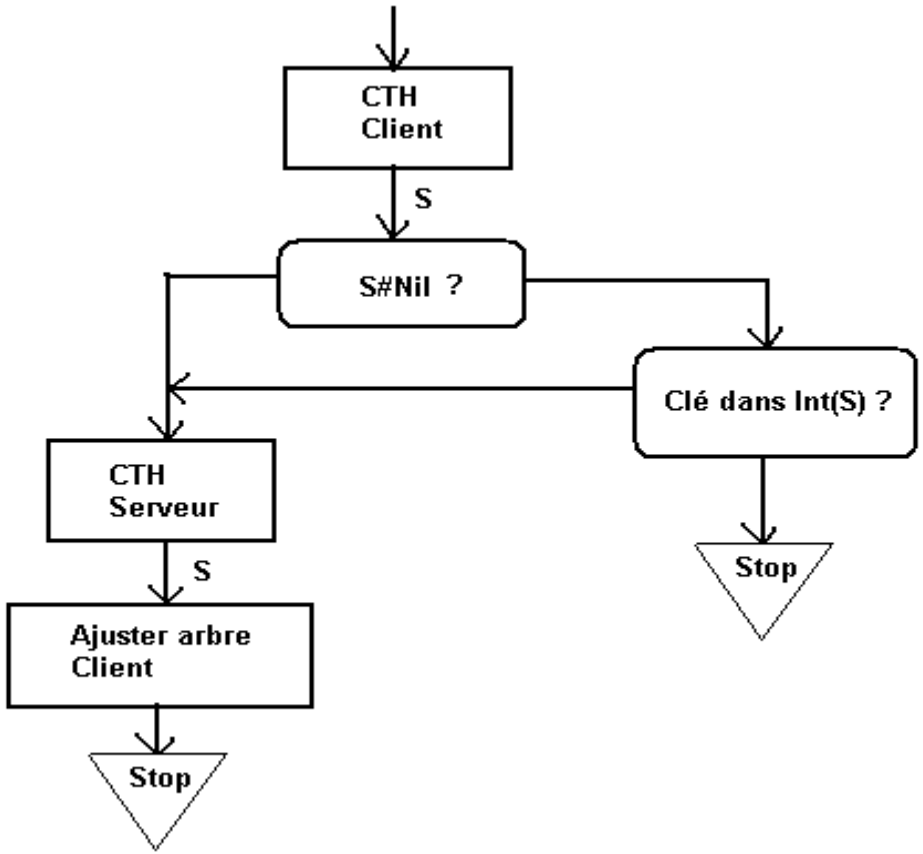
Distributed Compact trie hashing
Variantes

CTH* avec arbre central

- **Pas d'arbre au niveau des serveurs**
- **Arbre réel au niveau d'un serveur**
- **Pas de multicasting.**

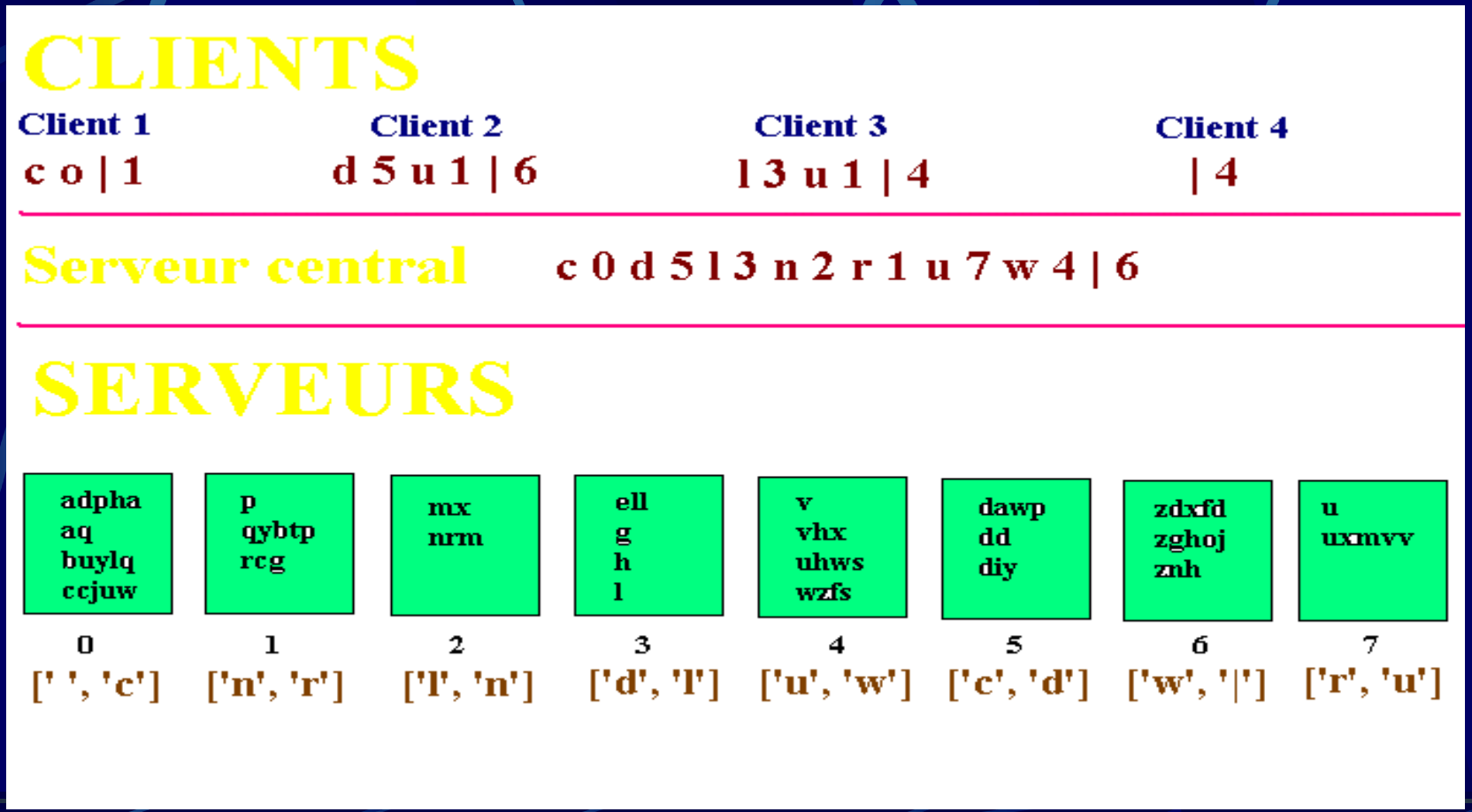
Distributed Compact trie hashing

Variantes



Distributed Compact trie hashing

Variantes



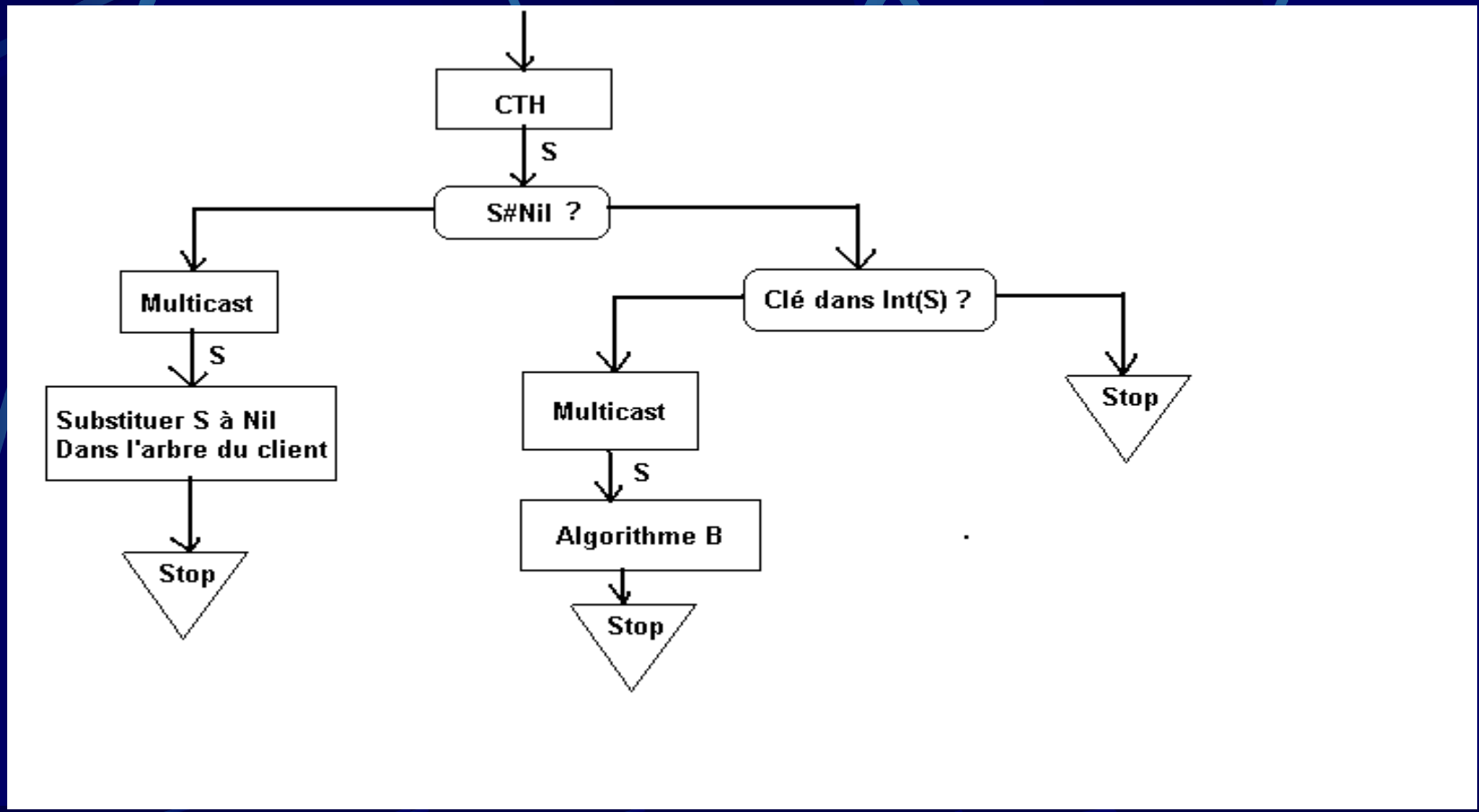
Distributed Compact trie hashing
Variantes

CTH* avec plus de ‘multicast’

- **Pas d’arbre au niveau des serveurs**
- **Pas d’arbre central**

Distributed Compact trie hashing

Variantes



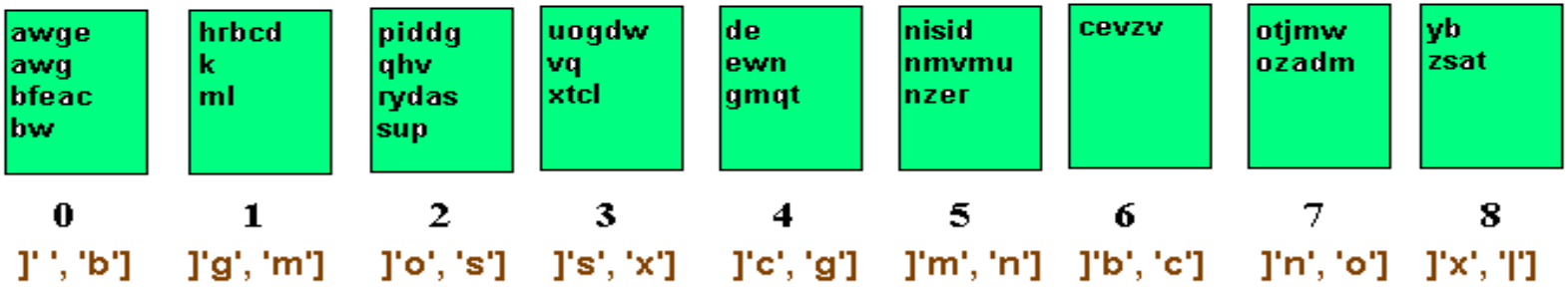
Distributed Compact trie hashing

Variantes

CLIENTS

1	2	3	4
g 0 | 3	o 5 | 0	c 0 g 4 o 1 s 2 | 3	o 1 | 2

SERVEURS



Distributed Compact trie hashing
Conclusion

- **Généralisation de CTH**
- **Préservation de l'ordre des articles : facilite les opérations de parcours séquentiel et de requêtes à intervalle.**
- **Un protocole de communication en cours d'écriture pour le test réel de la méthode (Plate forme Linux)**
- **La méthode de base avec chaînage des cases (comme les arbres B+)**
- **Deux variantes sont proposées**
- **Toutes les variantes de LH* peuvent s'intégrer dans CTH*.**