Systèmes de Fichiers

Hachage et Arbres B

Serge Abiteboul

INRIA

February 28, 2008

Systèmes de fichiers et SGBD

Introduction

Système d'exploitation contient un système de manipulation de fichiers
Système de gestion de bases de données

- Deux niveaux indépendants: logique, physique
- Grande quantité de données
- Recherches complexes: clés
- Offre d'autres fonctionnalités: concurrence, reprise sur pannes, confidentialité...
- En général, n'utilise pas le système de fichiers standard pour des raisons de performance

Pour quelles données ?

Système de fichiers propre au SGBD

- Données brutes
- Information sur le schéma logique (structure des données, autorisation,...)
- Information sur le schéma physique
- Statistiques: tailles des relations, critères de sélectivité, taux de mise-à-jour et requêtes.

Hiérarchie de mémoire

Faire croire: une grande mémoire rapide et peu chère

- prix: de la mémoire pas chère (disque)
- vitesse: de la mémoire rapide

Mémoire principale : semiconducteur (random access memory)

- zone de travail directement adressable
- petite (mega octets), chère, rapide, sensible (crash)
- ordre microsecondes

Mémoire secondaire à accès direct : disques optiques et magnétiques, mémoire flash (clé USB),

- résistante aux pannes système
- moins chère, moins rapide
- milliers de tours/mn
- ordre millisecondes

Mémoire tertiaire à accès indirect : bandes, jukebox optique

- archives, peu chère, très lente

Stockage ailleurs sur le réseau

Système de fichiers

Fichier : séquence d'articles (record/enregistrement)
Articles : composé d'un ou plusieurs champs élémentaires
Information

1. Données
2. Information de format: taille fixe ou variable
3. Deleted: pour des articles supprimés

Bloc = Unité de transfert avec la mémoire secondaire
typiquement une taille fixe 2^9 à 2^{12} octets
Segment = Plusieurs blocs
Fonctions: mises-à-jour de fichier (créer, détruire, lire, insérer, supprimer, modifier)
Gestion des pannes, des accès/confidentialité, de la concurrence, etc.

Article taille fixe

Nombre fixe de champs
Taille des champs fixe
Suppression avec compression $\Rightarrow$ coûteux
Suppression avec bit indiquant que l'article est vide $\Rightarrow$ crée des trous
Insertion : trouver un article vide

Articles de taille variable

1. Différents types d'articles dand même fichier
2. Champs de longueurs variables
3. Champs répétés: e.g., ensembles

Difficile à gérer

1. Suppression: combler les trous
2. Mises-à-jour: la taille d'un article qui change
3. Dépassement de blocs: article de très grosse taille

Solutions (sans miracle): Taille Fixe + débordement

1. réserver taille maximum $\Rightarrow$ place perdue
2. réserver taille faible + gestion blocs de débordement $\Rightarrow$ complexité

Optimisation des ressources

Optimiser les accès au disque

Clustering

- Regroupement des blocs physiques qui contiennent des données souvent utilisées ensembles
- Placement de ces blocs sur le même cylindre physique
- Compromis entre bon placement et trop de perte de place

Cache / Buffeur

- Zone de mémoire principale stockant des blocs
- Essayer de prévoir quels blocs seront utilisés
- Problèmes: stratégie de remplacement des blocs (e.g., LRU = least recently used)

Utiliser des structures de données plus riches que les fichiers en tas $\Rightarrow$

Fichiers logique

Fichiers logiques : Types abstraits de données

Fichier séquentiel

- lire premier/suivant (nom, buffeur, test)
- écrire (nom, buffeur), supprimer(nom,buffeur,test)

Fichier accès direct : utilise une clé[1]

- lire (nom, buffeur, clé, test) - accès par la clé
- écrire (nom, buffeur), supprimer (nom, clé, test)

Fichier séquentiel indexes

- lire premier/ suivant/ direct
- écrire; supprimer

Multiclé : on accède à partir de plusieurs clés

[1] clé: un champ qui identifie exactement chaque enregistrement.

Implantation des fichiers logiques

Critères

- temps moyen de recherche
- temps moyen de suppression
- temps moyen d'insertion
- espace supplémentaire

Typologie des utilisations

- accès à un article
- balayage du fichier
- taux de mises-à-jour

Différentes techniques

- Le tri
- Les index
- Arbre-B
- Hachage
- Hachage dynamique
- Multi-hachage

Fichiers Indexés

Le fichier est trié suivant une clé,
Les articles sont chaînés à l'aide de pointeurs
Insertion :

- dans un article vide du même bloc que l'article précédent l'article à insérer dans l'ordre sur la clé,
- dans un autre bloc en cas d'absence de place

Suppression :

- simple si on ne cherche pas à compacter

Problèmes :

1. Les mises-à-jour sont rapidement compliquées
2. Ne marche bien que si la taille du fichier est à peu près connue.
3. Accès trop lents $\Rightarrow$ index

Les Index

Index: structure annexe permettant d'accélérer certains accès
Index dense : index dans lequel chaque valeur de la clé est représentée
(nécessaire quand le fichier n'est pas trié)
nombre d'entrées dans l'index = nombre d'articles
Index non-dense: Un index non dense impose un fichier trié.

$$\text{Densité} \nearrow \text{efficacité} \nearrow \text{espace} \nearrow$$

Solution : index non dense avec une entrée par bloc

$$\text{Mise-à-jour de la base} \Rightarrow \text{Mise-à-jour des index}$$

Les Index: Opérations

Recherche : parcours de l'index + une lecture
Insertion :

- index dense: insérer la clé dans l'index si elle n'y est pas
- index non dense: pas de changement dans l'index si l'insertion est réalisée dans un bloc existant. Sinon, insérer la première clé du nouveau bloc dans l'index.

Suppression :

- dense: détruire la clé dans l'index si c'est le dernier article avec cette clé
- non dense: la plupart du temps pas de changement

Autres Index

Index primaire sur un attribut clé: un enregistrement au plus pour chaque valeur de la clé
Index secondaire sur un attribut non clé (en général dense)
Index croisé Sur un ensemble d'attributs
Fichier inversé Index sur tous ses attributs
Quand l'index grossit (e.g., index dense)

Stocker l'index dans un fichier

que l'on peut trier et indexer lui même.
Dans chaque bloc, on note la clé la plus petite, et on constitue un tableau

$\Rightarrow$ L'idée des arbres-B

Arbre-B+

Index multiniveau

- résistant aux mises-à-jour (pas de réorganisation)
- coût en temps et en espace

Principe : arbre équilibré
(chemin racine-feuille de longueur constante)

- Les noeuds stockent l'information de l'index,
- les Feuilles pointent sur le fichier,
- noeuds et feuilles correspondent à des blocs disque.

Contenu d'un noeud: $P_0 K_1 P_1 ... P_{N-1} K_N P_n$

- K_i : valeurs des clés $i \leq j \Rightarrow K_i \leq K_j$
- P_i : pointeurs

Contenu d'une feuille:
Bloc de données avec des clés voisines

Arbre-B+ – Suite

Recherche d'une clé b

1. Descente suivant P_i si $K_i \leq B \leq K_{i+1}$.
2. $H = \Theta(log(N))$ accès disque
   (H= profondeur de l'arbre, N nombre de blocs du fichier)

Insertion

1. Recherche du bloc
2. Insertion s'il y a de la place
3. Sinon : éclatement

Suppression : similaire

Problème : équilibrage de l'arbre

Exemple:

- blocs de 512 octets, clé de 6 octets
  $\Rightarrow$ 64 pointeurs par noeud
- 3 niveaux $\Rightarrow$ 128 MEGA OCTETS de fichier
- Si l'arbre est équilibré sinon ...

Arbre-B

Chaque K_i est un pointeur

1. Soit vers un autre noeud
2. Soit vers un bloc de données

Fichiers hachés

Technique

1. Tableau $T[0..n]$ de pointeurs vers des blocs
2. Fonction de hachage H :

$$\{\text{domaine de la clé }\} \Rightarrow [0..n]$$

3. Chaque $T[i]$ est un pointeur vers un bloc de données
4. Un article de clé K se trouve dans le bloc pointe par $T[H(K)]$

Problème: débordement

On chaîne un bloc supplémentaire
Recherche : 1 accès si pas de débordement
Insertion et suppression : naturelles
On vise un taux de remplissage de 60%
Si la taille varie beaucoup, les performances sont mauvaises, on passe au hachage dynamique.

Hachage - Arbre B

Hachage: facile à utiliser; très efficace
Arbre B: plus lourd mais permet des requêtes d'intervalle âge entre 15 et 20 ans qui passent mal avec le hachage

Hachage Dynamique

Fonction de hachage

$$H(K) = b_0 b_1 b_2 ... b_n ...$$

Table de hachage de taille variable :

$$1, 2, 4, ..., 2^n$$

Insertion: quand un bloc déborde,

- On double la taille de la table,
- On éclate le bloc en utilisant le bit de plus de la fonction de hachage.

Suppression: Deux voisins à moitie pleins, on fusionne

Bonnes performances

1. Recherche : 1 accès disque seulement
2. Insertion : max. deux écritures et une lecture
3. Suppression : max. deux lectures et une écriture

Multi-hachage

Hacher une grande table
(e.g. dictionnaire)

1. d mots
2. k fonctions h_i indépendantes dans $[0..N]$
3. tableau *tab* de n bits
   $TAB[j] = 1$ si $h_i(v) = j$ pour v dans le dico et $1 \leq i \leq k$

Probabilité du bit j à zéro
aucun mot d'un dictionnaire de taille d ne positionne un $T[j]$ donné est

$$p = \frac{\alpha^0}{0!} e^{-\alpha} = e^{-\alpha}$$

avec $\alpha = dk/n$.
Probabilité d'acceptation erronée
pour que v arbitraire soit accepté est $(1 - p)^k$.
Ex: commande *spell DE Unix.*
Choix: $d = 25000$, $n = 400000$, $k = 11$
Cela DONNE 0.000458711.

Digression - Moteur de recherche

Indexation de tous les mots: mot $\rightarrow$ URL

Tri pour pouvoir faire efficacement des recherches multi-mots

Importance des pages

Conclusion

Les fichiers sont souvent de taille fixe
Les index ne portent que sur des attributs de taille fixe
Le hachage est très utilisé (la taille est fixée à la création) notamment en recherche d'information (Web)
Le hachage dynamique parfois
Les arbres-B sont très utilisés (ainsi que plusieures variantes) pour les requêtes d'intervalle
Touts les SGBD supportent une forme d'indexation (souvent hachage et arbre-B)
Beaucoup supportent les index secondaires
Les systèmes de fichiers inversés sont utilisés en documentation

Merci