L'informatique des entrepôts de données

Daniel Lemire

SEMAINE 4
Les techniques d'indexation

4.1. Présentation de la semaine

Les entrepôts de données utilisent plusieurs techniques d'indexation. Nous avons abordé à la semaine précédente le problème de la sélection des vues. Les vues sont une forme d'indexation dans la mesure où elles servent à accélérer les requêtes.

Cette semaine, nous allons faire le tour des index plus traditionnels comme les arbres B et les tables de hachage. Cependant, dans le présent cours, on ne vous demandera pas de mettre au point vos propres arbres B ou tables de hachage. Il faut néanmoins en comprendre l'usage dans le contexte des entrepôts de données. C'est un des objectifs de cette semaine.

Pour construire des entrepôts de données, on doit pouvoir indexer des données hétérogènes : il n'y a pas que des tables relationnelles. Nous allons donc voir comment on indexe du texte ou des fichiers XML.

Nous terminerons la semaine avec le problème de l'indexation des jointures. Ce dernier problème ouvrira la porte aux index de la semaine prochaine : les index multidimensionnels.

4.2. Arbres B et tables de hachage

Étant donné une colonne avec n éléments, il est fréquent que l'on souhaite identifier les éléments correspondants à un certain critère. Par exemple, s'il y a une colonne « nom », on peut vouloir accélérer les requêtes du type « nom=Jean ». Dans le pire des cas, il faut visiter

chacun des n éléments de la colonne pour trouver les rangées correspondant à notre requête, mais nous pouvons souvent faire mieux.

Pratiquement tous les moteurs de base de données supportent les arbres B et les tables de hachage. Il faut en connaître les principales caractéristiques.

Les arbres B sont une des structures d'indexation les plus utilisées en informatique. Leurs principales caractéristiques sont un temps de recherche $O(\log n)$ – donc comparable à la recherche binaire – avec un accès aux valeurs dans l'ordre. En Java, la classe `java.util.TreeMap` est un exemple de structure en arbre. Les arbres B sont organisés de manière à tenir compte des caractéristiques des disques de stockage moderne. Il existe différentes variantes comme les arbres B+ ou les arbres B*, mais il n'est pas utile de les distinguer dans ce cours.

L'arbre B est souvent utilisé pour associer des clefs à des valeurs. Les nœuds vont contenir des paires clefs-valeurs. On peut prendre pour un valeur présente dans une colonne (par exemple « Jean ») pour l'associer avec un identifiant (par exemple, l'identifiant d'un enregistrement). Les clefs choisies doivent avoir un ordre (comme par exemple, l'ordre lexicographique). On pourrait mettre en œuvre une table relationnelle avec un arbre B utilisant comme clefs les valeurs de la clef primaire de la table. Les systèmes de fichiers utilisent souvent les arbres B pour associer les noms de fichiers à leur contenu.

A lire : `http://fr.wikipedia.org/wiki/Arbres_B`

La table de hachage est une autre option. En théorie, les tables de hachage sont bien plus rapides que les arbres B, lorsque l'on a beaucoup de données, puisqu'elles permettent une recherche en temps constant $O(1)$ (amorti). Par contre, les tables de hachage ne permettent pas de visiter les valeurs dans l'ordre. Comme les valeurs « proches » (par exemple, « Jean » et « Jeanne ») ne sont pas stockées dans la même région de mémoire au sein d'une table de hachage, les tables hachages peuvent être pénalisées par un accès en mémoire imprévisible, menant à des performances moindres. En Java, la classe `java.util.HashMap` est un exemple de table de hachage.

A lire : `http://fr.wikipedia.org/wiki/Table_de_hachage`

Afin de nous faire une idée, comparons la performance des classes `java.util.TreeMap` et `java.util.HashMap` pour la sélection d'éléments uniques. Considérons le programme Java suivant.

```
import java.util.*;

public class Comparaison {
```

```
  public static void run() {
   for(int N = 32; N<=1048576;N*=2) {
    int[] x = new int[N];
    Random rand = new Random();
    for(int k = 0;k < N; ++k) x[k] = rand.nextInt();
    HashSet<Integer> hs = new HashSet<Integer>();
    for(int k = 0;k < N; ++k) hs.add(x[k]);
    TreeSet<Integer> ts = new TreeSet<Integer>();
    for(int k = 0;k < N; ++k) ts.add(x[k]);
    int howmanyqueries = 1000000;
    int[] whichvalues = new int[howmanyqueries];
    for(int k = 0;k<howmanyqueries; ++k)
      whichvalues[k] = rand.nextInt();
        long aft2 = System.currentTimeMillis();
    for(int t=1;t<10;++t)
      for(Integer wv : whichvalues) hs.contains(wv);
        long aft3 = System.currentTimeMillis();
    for(int t=1;t<10;++t)
      for(Integer wv : whichvalues) ts.contains(wv);
        long aft4 = System.currentTimeMillis();
      System.out.println(N+" "
        +(aft3-aft2)/(1000.0*howmanyqueries)
        + " "+(aft4-aft3)/(1000.0*howmanyqueries));
     }
  }

  public static void main(String [] a) {
    run();
  }
}
```

La Figure 1 illustre la différence de performance entre un arbre et une table de hachage. On observe que la performance de la table de hachage ne décroît pas tellement au fur et à mesure que la structure de données devient plus volumineuse, alors que la performance de la structure de données en arbre se détériore assez rapidement. Lorsque notre ensemble atteint un million d'éléments, la table de hachage est 20 fois plus rapide. Cette différence de performance est cependant peut-être trompeuse. En effet, dans le contexte des entrepôts de données, on sélectionne rarement un seul élément. Par exemple, au lieu de chercher

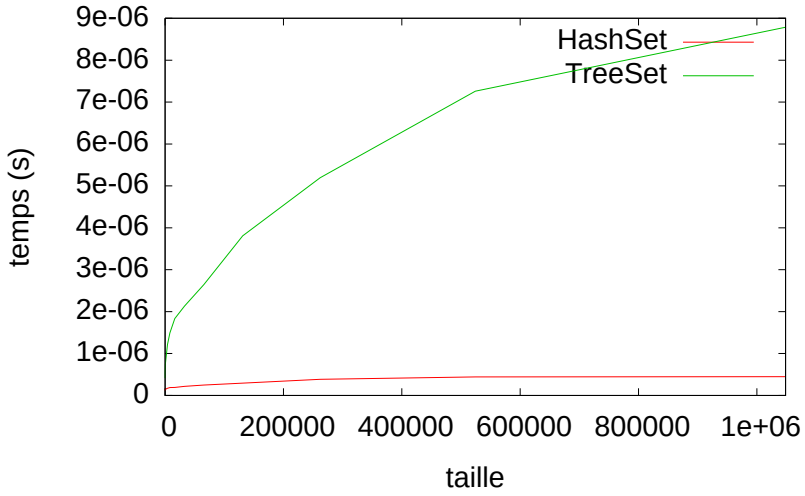

Figure 1. Comparaison entre les classes `java.util.TreeMap` et `java.util.HashMap` pour la sélection (aléatoire) d'éléments

un individu en particulier à partir de son nom, on cherchera plus souvent à sélectionner tous les individus ayant plus de 20 ans. Pour ce type de recherche par plage, la structure en arbre sera préférable.

Par ailleurs, lorsque la structure de données réside sur un disque rigide, l'arbre B est parfois avantagé. En effet, ce qui compte alors n'est plus tellement la quantité de données qui doit être lue, mais plutôt le nombre d'accès aléatoires au disque. En pratique, on peut souvent créer un arbre B qui a une hauteur ne dépassant pas beaucoup deux ou trois nœuds. Ce qui fait que le temps nécessaire pour retrouver une valeur correspond à deux ou trois accès aléatoires au disque. En comparaison, la table de hachage va nécessiter au moins un accès aléatoire, mais souvent deux accès [**1**]. Ainsi, il n'est plus si évident que la table de hachage sera beaucoup plus rapide que l'arbre B lorsque ceux-ci résident sur le disque.

Au total, dans les entrepôts de données, nous utilisons souvent les arbres B, car ils sont plus polyvalents. Cependant, les tables de hachage peuvent être plus rapides.

Il existe de nombreuses variantes de l'arbre B. Les arbres B sont utilisés pour indexer de grands volumes de données hors mémoire, organisant ces données en pages (par ex., faisant 4 Ko) pour optimiser les opérations de disque, mais ils souffrent d'une inefficacité dans la mise en cache. Le Bf-Tree [**?**] introduit une séparation entre les pages de cache et les pages disque, permettant une gestion plus fine avec des mini-pages dans une nouvelle structure de pool de buffer, améliorant

ainsi les performances de lecture et écriture, et se montrant jusqu'à 6 fois plus rapide pour les écritures que les arbres B traditionnels. Des systèmes comme UpscaleDB compressent les clés afin d'améliorer la performance [**3**].

4.3. Indexation du texte

L'indexation du texte passe normalement par la construction d'un *index inversé*. Un index inversé donne, pour un mot donné, la liste des textes dans lesquels il apparaît, et ce, rapidement. Accessoirement, il peut aussi inclure la liste des positions du mot dans les textes. On peut mettre en œuvre un index inversé en utilisant un arbre B ou une table de hachage.

Par exemple, étant donné les textes

- D1 = "La vie est belle"
- D2 = "Belle belle"

Un index inversé indiquant en plus la position des mots prendra la forme suivante :

- la → D1,1
- vie → D1,2
- est → D1,3
- belle → D1,4; D2,1; D2,2

Un index inversé permet de trouver, étant donné un mot, ses positions correspondantes, en temps $O(1)$ si on utilise une table de hachage. En d'autres mots, si on double le nombre de documents, le temps de recherche ne sera pas affecté. Par contre, la mise à jour de l'index, lors de l'ajout ou de la modification d'un document, peut prendre un temps linéaire $O(n)$ dans le pire des cas. Cela signifie que la construction d'un index peut être coûteuse. En d'autres mots, un index inversé est une structure qui privilégie la rapidité de la réponse aux requêtes, mais qui peut coûter cher en temps de construction et de mise à jour.

Comme un index inversé peut contenir autant d'information que le corpus lui-même, son espace de stockage nécessaire peut être encore plus grand que celui requis par le corpus. Par contre, avec des techniques de compression, on obtient généralement un index qui est significativement plus petit que la somme des documents, mais un index occupera toujours au moins 10 % de l'espace requis pour stocker les documents eux-mêmes. La compression de l'index peut améliorer les performances globales en limitant la lecture de données sur le disque.

Avec la classe `java.util.HashMap`, on peut facilement mettre en œuvre un index inversé simple :

```
import java.util.*;

public class IndexInverse {
  HashMap<String, Vector<Integer>> index
  = new HashMap<String, Vector<Integer> >();
  public IndexInverse() {}
  public void ajoute(String mot, int position) {
    if(! index.containsKey(mot))
      index.put(mot,new Vector<Integer>());
    index.get(mot).add(position);
  }

  public Vector<Integer> trouve(String mot) {
    return index.get(mot);
  }

  public static void main(String[] args) {
    String[] D1 =
    new String("La vie est belle").split(" ");
    String[] D2 =
    new String("Belle belle").split(" ");
    IndexInverse ii = new IndexInverse();
    for(String s: D1) {
      ii.ajoute(s,1);
    }
    for(String s: D2) {
      ii.ajoute(s,2);
    }
    System.out.println("Je cherche vie...");
    for(int i : ii.trouve("vie"))
      System.out.println("dans le document: "+i);
    System.out.println("Je cherche belle...");
    for(int i : ii.trouve("belle"))
      System.out.println("dans le document: "+i);
  }
}
```

Lors de l'indexation d'un texte, il faut cependant tenir compte de plusieurs autres stratégies :

- On n'indexe pas tous les mots. Les articles comme le, la, les sont le plus souvent omis. Un ensemble de mots vides de sens s'appelle un *antidictionnaire*.
- Lorsque l'on indexe des mots, il faut tenir compte qu'il y a plusieurs formes. Par exemple, la casse d'un mot peut changer (Orange versus orange), on peut le mettre au pluriel ou le conjuguer (orange, oranges; devoir, devrait). Il faut donc procéder à la *lemmatisation* des mots avant de les indexer (voir `http://fr.wikipedia.org/wiki/Lemmatisation`). Évidemment, la lemmatisation dépend de la langue du texte. Dans le cas de documents suffisamment longs, il n'est pas difficile d'identifier automatiquement la langue (par exemple, la présence de l'article *the* indique que le texte est en anglais).
- Les moteurs de recherche doivent souvent classer les résultats selon leur pertinence. L'approche de pondération la plus utilisée est sans doute le *term frequency-inverse document frequency* (tf-idf) ranking (voir `http://fr.wikipedia.org/wiki/TF-IDF`) qui s'appuie sur le modèle vectoriel (voir `http://fr.wikipedia.org/wiki/Modèle_vectoriel`). Il existe d'autres alternatives, incluant les modèles probabilistes.

4.4. Indexation XML

Plusieurs systèmes de bases de données, tels que IBM DB2, SQL Server ou Oracle, permettent de charger et d'indexer des documents XML. Il existe aussi des moteurs de base de données conçus spécialement pour le XML. On peut ensuite interroger ces bases de données en utilisant XPath or XQuery. En fait, le langage XQuery a été conçu comme un équivalent du SQL pour le XML.

A lire : `http://fr.wikipedia.org/wiki/XPath`

A lire : `http://fr.wikipedia.org/wiki/XQuery`

Le traitement de ces documents peut poser des problèmes de performance. La méthode la plus simple pour indexer un série de documents XML est de les transformer en structures dans lesquelles le logiciel peut naviguer directement sans devoir interpréter les balises XML. Considérons, par exemple, le modèle ORDPATH [**6, 5**]. Dans ce modèle, on commence par associer le nœud-racine avec l'identifiant 1. Le premier nœud contenu dans le nœud-racine se voit attribué l'identifiant 1.1, le second 1.2, etc. On procède ensuite de manière récursive. Le premier nœud dans le premier élément sera noté 1.1.1, le second 1.1.2,

ORDPATH	nom	type	valeur
1	liste	élément	aucune
1.1	temps	attribut	janvier
1.2	bateau	élément	aucune
1.3	bateau	élément	aucune
1.3.1	canard	élément	aucune

Figure 2. Exemple de représentation ORDPATH

etc. Étant donné les identifiants ORDPATH, on peut facilement déterminer si deux nœud sont inclus l'un dans l'autre, ou voisins, etc. La Figure 2 illustre la représentation ORDPATH concernant le document XML suivant :

```
<liste temps="janvier" >
 <bateau />
 <bateau >
  <canard />
 </bateau>
</liste>
```

Après avoir finalement obtenu une table, on peut ensuite l'indexer en utilisant des index conventionnels (arbres B, etc.). Par exemple, un index sur la colonne valeur nous permettra de déterminer rapidement s'il y a un résultat non trivial correspondant à la requête XPath @temps="janvier".

A consulter : article original par O'Neil et al. (en anglais)

(Strictement parlant, on ne devrait attribuer que des identifiants utilisant des nombres impairs au départ, comme 1.1, 1.3, 1.5, etc. O'Neil et al. recommande de réserver les nombres pairs pour de nouveaux nœuds pouvant être insérés.)

4.5. Indexation des jointures

Une opération fréquente en SQL est la jointure. Les jointures sont souvent des opérations relativement coûteuses. Sans index, on peut faire une jointure entre deux tables en temps $O(n \log n)$ avec l'algorithme Sort-Merge. Pour illustrer cet algorithme, considérons deux tables que l'on souhaite joindre sur la colonne nom.

nom	profit
Jean	531.00
Pierre	132.00
Marie	32.00

nom	ville
Jean	Montréal
Marie	Québec
Pierre	Roberval

La première opération consiste à trier la colonne correspondante, dans les deux tables :

nom	profit
Jean	531.00
Marie	32.00
Pierre	132.00

nom	ville
Jean	Montréal
Marie	Québec
Pierre	Roberval

Ainsi, il est facile de voir que le premier enregistrement de la première table correspond au premier enregistrement de la seconde table, que le deuxième et le troisième enregistrements de la première table correspondent au deuxième enregistrement de la seconde table. Une fois que les tables ont été triées, il nous suffit de lire en séquence dans chacune des tables. Si on omet le tri, l'opération peut se réaliser en temps linéaire ($(O(n)$) avec un accès entièrement séquentiel des données. Le tri, quant à lui, peut être effectué avec un algorithme efficace—généralement en temps $O(n \log n)$.

nom	profit	nom	ville
Jean	531.00	Jean	Montréal
Marie	32.00	Marie	Québec
Pierre	132.00	Pierre	Roberval

Au lieu d'utiliser cette dernière technique (Sort-Merge), il existe une autre technique basée sur les tables de hachage, appelée Hash Join, laquelle est parfois préférable pour les tables en mémoire vive [**2**]. Pour chaque valeur de la colonne sur laquelle porte la jointure, on associe les enregistrements de la première table et de la seconde [**7**]. Cette association peut se faire au sein d'une table de hachage. Le temps de traitement est donc linéaire ($O(n)$). Malheureusement, une approche par table de hachage nécessite plusieurs accès aléatoire et ceux-ci peuvent être beaucoup plus coûteux que des accès séquentiels.

nom	enregistrements
Jean	(Jean, 531.00) / (Jean, Montréal)
Marie	(Marie, 32.00) / (Marie, Québec)
Pierre	(Pierre, 132.00) / (Pierre, Roberval)

La jointure peut aussi concerner plusieurs tables [**4**] et, dans ce cas, on peut adopter une approche similaire. La normalisation qui mène à une fragmentation en plusieurs tables est souhaitable dans les applications transactionnelles, car elle accélère les mises à jour. Par contre, dans les entrepôts de données, on vise au contraire à minimiser les jointures qui peuvent être coûteuses.

4.6. Questions d'approfondissement

(a) Vous souhaitez indexer une colonne temporelle de manière à identifier tous les événements qui se sont produits dans une plage de temps, quel type d'index devez-vous utiliser?
(b) Un ingénieur détermine que les performances d'un index en arbre B et un index en table de hachage sont équivalentes lors de la sélection de valeurs uniques. Il indexe un million de rangées. Combien faudrait-il de rangées pour que la performance de la table de hachage soit deux fois supérieure à celle de l'arbre B (en théorie)?
(c) En triant une liste de valeurs, on peut ensuite avoir accès à tout élément de la liste en temps logarithmique par recherche binaire[1]. Cette approche est beaucoup plus simple que l'arbre B, qui pourtant a la même complexité. Quel est donc le bénéfice de l'arbre B?
(d) Est-ce qu'un index inversé peut m'aider à trouver tous les textes contenant un mot se terminant par *ent*?
(e) Est-ce que le représentation ORDPATH peut avoir des inconvénients par rapport au simple stockage du document XML?

4.7. Réponses suggérées

(a) Arbre B.
(b) rchons u tel que $\log_2 u = 2\log_2 1,000,000$. Nous avons $u = 1,000,000^2 = 10^{2\times 6} = 10^{12}$.
(c) L'arbre B peut être mis à jour en temps logarithmique.

[1]La recherche binaire débute en comparant la valeur recherchée avec celle se situant au milieu de la liste. Si cet élément est plus grand que l'élément recherché, ce dernier est à gauche sinon à droite. On répète l'opération.

(p) Probablement pas.

(e) Reconstruire un document XML à partir d'une table ORDPATH prend du temps. Si on a besoin de récupérer les documents XML entiers, et non pas seulement quelques nœuds, il vaut mieux stocker le document XML en entier.

BIBLIOGRAPHIE

1. M. Jensen and R. Pagh. Optimality in external memory hashing. *Algorithmica*, 52(3):403–411, 2008.
2. C. Kim, T. Kaldewey, V. Lee, E. Sedlar, A. Nguyen, N. Satish, J. Chhugani, A. Di Blas, and P. Dubey. Sort vs. Hash revisited: fast join implementation on modern multi-core CPUs. *VLDB*, 2(2):1378–1389, 2009.
3. D. Lemire and C. Rupp. Upscaledb: Efficient integer-key compression in a key-value store using simd instructions. *Information Systems*, 66:13–23, 2017.
4. P. O'Neil and G. Graefe. Multi-table joins through bitmapped join indices. *ACM SIGMOD Record*, 24(3):11, 1995.
5. P. O'Neil, E. O'Neil, S. Pal, I. Cseri, G. Schaller, and N. Westbury. ORDPATHs: insert-friendly XML node labels. In *SIGMOD '04*, pages 903–908. ACM, 2004.
6. S. Pal, I. Cseri, O. Seeliger, G. Schaller, L. Giakoumakis, and V. Zolotov. Indexing XML data stored in a relational database. In *VLDB '04*, 2004.
7. P. Valduriez. Join indices. *ACM Transactions on Database Systems (TODS)*, 12(2):218–246, 1987.