Devoir noté – Hash Join

Laurent Bindschaedler Jean-Cédric Chappelier

du 13 avril 2016, 12:00, au 25 avril 2016, 23:59.

I. Introduction et instructions générales

Ce devoir noté consiste à écrire une version simplifiée d'une jointure par hashing (« *hash join* »), beaucoup utilisée dans l'implémentation de bases de données relationnelles. Nous allons ici l'appliquer à des fichiers CSV (« *comma-separated values* »).

Vous écrirez tout votre code dans un seul fichier nommé `cvs_join.c` sur la base d'un fichier fourni, à compléter.

Indications : Si un comportement ou une situation donnée n'est pas définie dans la consigne ci-dessous, vous êtes libres de définir le comportement adéquat. On considérera comme comportement adéquat toute solution qui ne viole pas les contraintes données et qui ne résulte pas en un crash du programme.

Instructions :

1. Cet exercice doit être réalisé **individuellement !**
   L'échange de code relatif à cet exercice noté est **strictement interdit !** Cela inclut la diffusion de code sur le forum.

   Le code rendu doit être le résultat de *votre propre production*.
   Le plagiat de code, de quelque façon que de soit et quelle qu'en soit la source sera considéré comme de la tricherie (c'est, en plus, illégal et passible de poursuites pénales).

   En cas de tricherie, vous recevrez la note «NA» pour l'entièreté de la branche et serez de plus dénoncés et punis suivant l'ordonnance sur la discipline.

2. Vous devez donc également garder le code produit pour cet exercice dans un endroit à l'accès strictement personnel.

   Le fichier (source !) `cvs_join.c` à fournir pour cet exercice ne devra plus être modifié après la date et heure limite de rendu.

3. Veuillez à rendre du code *anonyme* : **pas** de nom ni numéro SCIPER !

4. Utilisez le site Moodle du cours pour rendre votre exercice.

Vous avez jusqu'au lundi 25 avril, 23:59 (heure du site Moodle du cours faisant foi) pour soumettre votre rendu.
Aucun délai supplémentaire ne sera accordé.

II. Implémentation d'une hashtable

Récupérez le fichier `csv_join.c` fourni sur le site Moodle et regardez-le pour comprendre la structure globale.

Pour commencer, vous devez implémenter une *hashtable* (« table de hachage »). Pour rappel, une *hashtable* est une structure de données qui permet un accès en temps constant ($O(1)$) à l'aide de la « *valeur de hachage* » (*hash value*) d'une clé d'accès :

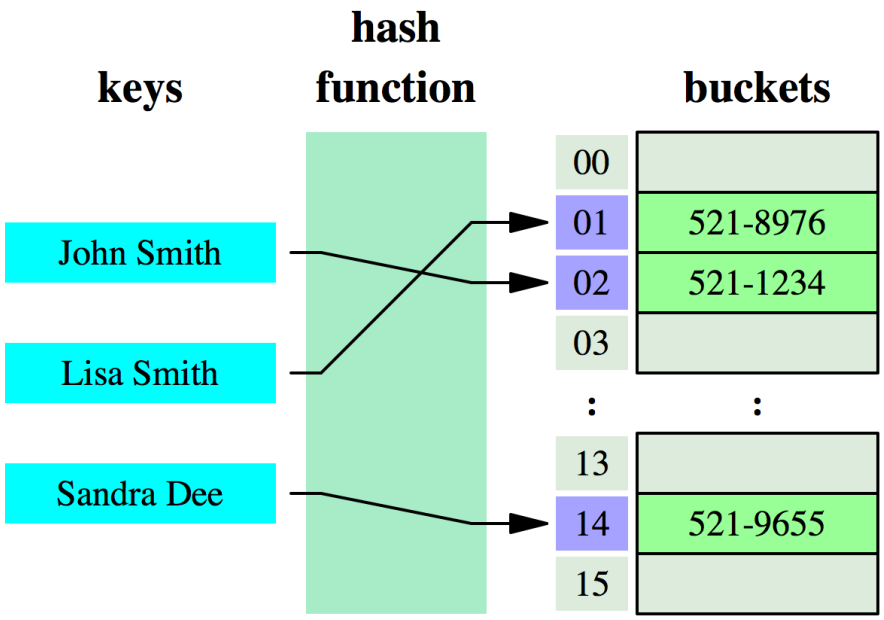

Figure 1: Une hashtable lie une clé à une valeur à l'aide d'une fonction de hashing.

La représentation concrète de la *hashtable* que nous vous proposons d'implémenter dans ce devoir ressemblera à ceci :

Définissez un type `bucket` qui correspond à un contenu de *hashtable* (clé, valeur et autres champs éventuels si nécessaires). Le type de la clé est `const char*` et

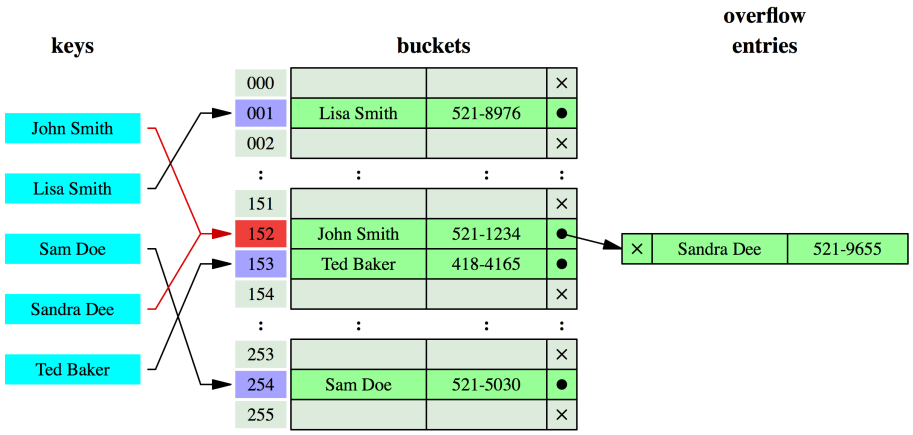

Figure 2: Représentation d'une hashtable avec gestion des collisions par chaînage.

celui de la valeur est `const void*` (pointeur générique). Nous insistons sur les `const` ici (la *hashtable* ne doit pas modifier ses contenus).

Défnissez ensuite un type `Htable` qui correspond à une *hashtable* contenant `size` alvéoles (ou positions) pour des entrées (256 dans la figure ci-dessus) ; `size` est une caractéristique de la *hashtable* (indiquant sa taille en nombre de position possibles ; notez que le vrai nombre d'entrées contenues dans la table peut être différent : certaines positions sont vides (exemple la 002 ci-dessus), certaines clés peuvent partager la même position (exemple « John Smith » et « Sandra Dee » ci-dessus)).

Note : Vous **devez** effectuer des allocations dynamiques de la mémoire associée à la *hashtable* afin d'éviter de gaspiller inutilement de la mémoire. Des points bonus seront attribués pour les implémentations économes. Il va de soi (et on ne le rappellera plus) que toute mémoire allouée dynamiquement doit être libérée proprement.
[Voir également la note au sujet de la propriété, plus bas.]

Définissez ensuite les fonctions suivantes qui permettent d'effectuer un ensemble d'opérations standard sur la *hashtable* :

- `construct_Htable` qui prend une taille en paramètre et retourne un pointeur vers une *hashtable* nouvellement créée (à la bonne taille) ;
- `delete_Htable_and_content` qui libère la mémoire associée à la *hashtable* passée en paramètre, **ainsi que tout le contenu mis dans cette table** (cf remarque sur la propriété ci-dessous) ;
- `add_Htable_value` qui prend une *hashtable* en argument ainsi qu'une clé (`const char *`) et une valeur (`const void *`) et qui stocke la valeur associée à la clé dans la *hashtable*.

Si la clé existe déjà dans la *hashtable*, elle doit être remplacée par la nouvelle valeur. En cas de collision des valeurs de hash pour deux clés différentes, on utilisera une liste chainée pour stocker les paires clé-valeur (cf. fig. ci-dessous) ;

- `get_Htable_value` qui prend une *hashtable* en argument ainsi qu'une clé (`const char *`) et qui retourne la valeur associée (`const void *`) si elle existe ou `NULL` si elle n'existe pas.

NOTE au sujet de la propriété des contenus de la table : les plus avancés d'entre vous se seront sûrment posé la question sur la propriété du contenu de la *hashtable* (i.e. les éléments contenus dans la table doivent ils être « cachés » ou peuvent ils être partagés à l'extérieur ?).

Comme nous n'allons ici utiliser la *hashtable* que localement sur le même contenu que celui utilisé par ailleurs, nous avons choisi de vous proposer ici une implémentation *simple* dans laquelle le contenu de la *hashtable* peut être partagé à l'extérieur de celle-ci (pas besoin de copie, donc).

C'est également pour cela que nous vous demandons d'implémenter une fonction `delete_Htable_and_content` au lieu de simplement `delete_Htable` (qui, en cas d'implémentation partagée comme ici, ne supprimerait pas les contenus partagés) : puisque dans ce devoir le propriétaire de la *hashtable* sera le même que celui du contenu partagé, on peut se permettre que la *hashtable* libère ce contenu partagé.

Tout ceci est bien sûr une version *simplifiée* pour ce devoir, pas tout à fait conforme à la réalisation d'une « vraie » *hashtable* plus professionnelle…
[fin de la note]

Pour calculer le valeur de hash d'une clé (`key` ci-dessous) donnée, pour une taille de *hashtable* donnée (`size`), nous vous fournissons la fonction suivante (déjà présente dans le code fourni) :

```
/** ---------------------------------------------------------------------
 ** Hash a string for a given hashtable size.
 ** See http://en.wikipedia.org/wiki/Jenkins_hash_function
 **/
size_t hash_function(const char* key, size_t size)
{
    size_t hash = 0;
    size_t key_len = strlen(key);
    for (size_t i = 0; i < key_len; ++i) {
        hash += (unsigned char) key[i];
        hash += (hash << 10);
        hash ^= (hash >> 6);
    }
    hash += (hash << 3);
    hash ^= (hash >> 11);
    hash += (hash << 15);
```

```
    return hash % size;
}
```

N'hésitez pas à créer d'autres fonctions utilitaires si nécessaire !

Et n'oubliez pas de tester correctement les fonctionnalités de votre *hashtable* pour plusieurs combinaisons de tailles, clés et valeurs **avant** de passer à l'étape suivante. Cela vous évitera bien des tracas…

III. [Pour information :] Lecture et écriture d'un fichier CSV (code fourni)

Les relations d'une base de données peuvent être vues comme de simples tableaux à deux dimensions. Les fichiers que nous allons utiliser pour les représenter sont au format CSV. Par convention, tous les fichiers CSV traités dans ce devoir contiendront une ligne d'en-tête indiquant à quoi correspondent les valeurs des lignes suivantes. Voici un extrait d'un tel fichier :

```
Code,English Name,French Name
CH,Switzerland,Suisse
FR,France,France
GB,United Kingdom,Royaume-Uni
US,United States,Etats-Unis
...
```

Le type `csv_row` fourni correspond à une ligne d'un fichier CSV.

La fonction `read_row` accepte un fichier (ouvert en lecture) et y lit une ligne. Elle retourne une `csv_row` contenant les éléments de la ligne lue. En cas d'erreur, la fonction `read_row` retourne `NULL`.

La fonction `write_row` accepte un fichier (ouvert en écriture) ainsi qu'une `csv_row` et l'écrit à la suite dans le fichier.

La fonction `write_rows` accepte un fichier (ouvert en écriture) et deux `csv_row`s ; elle les écrit l'une à la suite de l'autre sur une même ligne, séparées par une vrigule.

Ces deux fonctions ont un dernier paramètre, `ignore_index`, contenant l'indice d'une colonne à ne pas dupliquer (ce sera la valeur de la colonne de jointure de la *seconde* relation dans l'utilisation ci-dessous ; ainsi, cette colonne ne sera pas dupliquée dans le fichier résultat).

La fonction `row_element` reçoit une `csv_row` ainsi qu'un index `i` en paramètre ; elle copie et retourne l'élement à la position `i`. En cas d'erreur, cette fonction retourne `NULL`.

Vous utiliserez ces fonctions pour implémenter la fonction suivante (*join*).

IV. Implémentation du hash join

Avant de commencer cette partie, assurez vous d'avoir correctement testé le code écrit dans la partie I.

La jointure (*join*) est une opération fondamentale en base de données. Elle consiste à associer deux relations (ou tables) par le biais d'un lien logique donné par un prédicat. La plupart du temps, le prédicat consiste à joindre les lignes de deux tableaux dont les valeurs d'une certaine colonne sont identiques (le prédicat utilisé est donc l'égalité).

Par exemple, considérez la jointure suivante entre deux tables : une liste des employés d'une entreprise avec l'identifiant du département où ils travaillent et une liste des départements de l'entreprise :

Employé	Département ID
Alice	4
Bob	1
Charles	2
Eve	2
Oscar	1

Département	Département ID
Informatique	1
Vente	2
Service après-vente	3
Marketing	4

On cherche alors à construire la table des employés avec le nom du département dans lequel ils travaillent. Ce problème s'exprime justement par une jointure relationnelle sur la colonne *Département ID*. Pour l'exemple ci-dessus, le résultat de la jointure serait :

Employé	Département
Alice	Marketing
Bob	Informatique
Charles	Vente
Eve	Vente
Oscar	Informatique

Une jointure par hashing n'est pas forcément la meilleure méthode pour calculer la jointure de deux relations. En revanche, si les relations en question sont

gigantesques (plusieurs terabytes), il devient difficile d'effectuer la jointure en mémoire car les tables ne peuvent pas y être stockées en entier.

L'algorithme de jointure par hashing classique a été conçu pour résoudre ce problème. Il fonctionne de la façon suivante.

Pour chaque ligne $r1$ de la première relation :

1. Ajouter $r1$ à la *hashtable*.
2. Si la *hashtable* est pleine ou que la première relation a été entièrement scannée :

- Scanner la seconde relation ligne par ligne ($r2$) et rechercher la colonne de jointure dans la *hashtable*.
- Si la colonne est présente dans la *hashtable*, écrire la jointure des deux lignes ($r1$ suivie de $r2$) à la suite du résultat.
- Remettre la *hashtable* à zéro et continuer au point 1.

NOTES :

1. On considère le cas simplifié où la première relation est plus petite que la seconde. De plus, vous pouvez supposer que la première relation ne contient pas de duplicatas sur la colonne de jointure (chaque élément est unique). Attention, ce n'est pas forcément le cas pour la seconde relation !

2. On considère que la *hashtable* est « pleine » à partir du moment où le nombre d'entrées qu'elle contient est au moins égal à 75% de sa taille (paramètre `HASH_TABLE_LOAD_FACTOR` défini en début de code fourni).

3. Ecrivez l'en-tête *combiné* (des deux relations) à la première ligne du fichier résultat, avant d'écrire le résultat de la jointure.

Consigne : Vous devez *scanner* les relations. Cela veut dire que vous devez lire les fichiers correspondant séquentiellement (potentiellement plusieurs fois dans le cas de la seconde relation ; pour revenir au début d'un fichier `file`, utiliser `fseek(file, 0, SEEK_SET);`) et écrire séquentiellement dans le fichier de résultat. On aura donc à tout moment au maximum la *hashtable* (et son contenu) en mémoire ainsi qu'**une seule** ligne de la seconde relation.

Implémentez l'algorithme de jointure ci-dessus dans une fonction `hash_join` qui prendra en paramètres (regardez son appel dans `main()`) : * les deux fichiers correspondant aux deux relations en entrée ; * le fichier dans lequel écrire le résultat ; * les indexes des colonnes à joindre dans les deux relations (un index par relation ; * la taille mémoire maximale à utiliser pour stocker son contenu (son tableau de « *buckets* »). En cas de succès, cette fonction doit retourner `0`. En cas d'erreur, elle retournera un entier non nul.

V. Test du tout

Vous pouvez tester votre programme complet à l'aide du `main()` fourni et les entrées suivantes (fichiers fournis sur le Moodle du cours) :

- Relation 1 (`countries.csv`) :

```
Code,English Name,French Name
CH,Switzerland,Suisse
FR,France,France
GB,United Kingdom,Royaume-Uni
US,United States,Etats-Unis
```

- Relation 2 (`cities.csv`) :

```
Country Code,City Name,Ranking
CH,Geneva,1
CH,Lausanne,2
CH,Zurich,3
FR,Paris,1
US,San Francisco,1
US,New York,2
US,Los Angeles,3
US,Washington DC,4
US,Seattle,5
US,Minneapolis,6
```

Le résultat attendu devrait être (`join.csv` ; **Attention !** de fournir un nom différent pour ne pas écraser cette référence lors de vos test `;-)`) :

```
Code,English Name,French Name,City Name,Ranking
CH,Switzerland,Suisse,Geneva,1
CH,Switzerland,Suisse,Lausanne,2
CH,Switzerland,Suisse,Zurich,3
FR,France,France,Paris,1
US,United States,Etats-Unis,San Francisco,1
US,United States,Etats-Unis,New York,2
US,United States,Etats-Unis,Los Angeles,3
US,United States,Etats-Unis,Washington DC,4
US,United States,Etats-Unis,Seattle,5
US,United States,Etats-Unis,Minneapolis,6
```

Note : vous pouvez comparer deux fichiers à l'aide de la commande `diff` :

```
diff join.csv mon_output.csv
```