Fichiers et indexation



Les données numériques sont une des composantes essentielles des programmes informatiques.

- Il s'agit par définition des *informations qui doivent être conservées entre deux exécutions*.
- Avec l'augmentation des capacités de stockage et leur mise en réseau, les quantités de données conservées ont considérablement augmenté au cours des dernières décennies.

Dans le cadre de ce cours, nous aborderons :

- la question du stockage de ces données sur un support informatique (Fichiers, bases de données),
- ainsi que les méthodes permettant de consulter et mettre à jour régulièrement ces données.

1 Données et fichiers

1.1 Transport et flux de données

- La transmission des données entre programmes nécessite l'ouverture d'un canal de communication
  - entre client et serveur
  - par lequel transitent les données (les requêtes et les réponses).
- Le transport est géré
  - par le système d'exploitation (lorsque les données transitent au sein d'un même ordinateur)
  - ainsi que par des routeurs (lorsque les données transitent d'un ordinateur à l'autre sur le réseau).
- Au niveau du client,
  - les réponses en provenance du serveur sont organisées sous la forme d'une liste,
  - qu'on appelle un **flux de données**.



La notion de flux de données signifie que les réponses sont lues dans un ordre fixe, telles qu'elles ont été écrites au niveau du serveur. On parle de lecture à accès **séquentiel** (par opposition à la lecture à accès aléatoire).

Formats d'échange

Les principaux formats d'échange de données sont :

- csv
- json
- xml



TODO

Exemples :

- Clients
- Cours de l'Euro
- codes postaux
- codes postaux

1.2 Conservation des données

La conservation des données repose principalement sur la **structure de stockage**,

- définissant la manière dont les données sont physiquement stockées sur le support,
- autrement dit la méthode de **rangement** de la série de mesures :
  - fichiers,
  - répertoires,
  - index,
  - etc...
- reposant sur des supports de stockage (ou mémoires) :
  - mémoire centrale,
  - mémoires secondaires (disque dur, CD-ROM, memoire flash (SSD), etc...).

Trame et bloc de données

Un jeu de valeurs encodé et stocké sur un support informatique est appelé un "**enregistrement**",

- conservé sous la forme d'une **trames de données**.
- Une trame peut obéir à un format *textuel* ou *binaire*

Encodage binaire :

- Définition d'une trame, en général de taille fixe.
- Chaque rubrique occupe un nombre d'octets déterminé, afin que chaque jeu de données occupe la même place en mémoire.
- L'utilisation de trames de taille fixe facilite le stockage et la conservation des données sur les supports magnétiques (disque dur, etc...)
- Les données sont transmises dans un format numérique (type) identique à celui utilisé en mémoire centrale.



** Trame de données **



Dubois	Martine	29, rue du Verger, Orléans	22
15 octets	15 octets	40 octets	4 octets

Encodage textuel :

- Le jeu de données est codé dans un format descriptif (contenant à la fois les valeurs et une description des données : types, attributs, ...).
- Ce format facilite la transmission d'un programme à un autre (format "plat") mais est moins propice au stockage.
- La "sérialisation" est l'opération qui consiste à encoder des données sous la forme d'un texte brut (codage ASCII ou utf8), en "perdant" le moins possible d'information.
- Des exemples de formats textes standards sont donnés en 2.1.3 Structures de données.

Tableaux statiques

Bloc de données

Un **bloc de données** correspond à une série d'enregistrements obéissant au même **format**:

- Chaque enregistrement est de taille identique;
- L'encodage des données est le même.

La structure de base servant à stocker à une série d'enregistrements est le tableau de données à 2 dimensions :

- Tableau de données (data frame) = intitulé de colonnes + liste de lignes
  - Intitulé de colonne = nom de l'attribut
  - Une ligne = un enregistrement

Structure sous-jacente : tableau à 2 dimensions (ou matrice de données)



Dubois	Martine	29, rue du Verger, Orléans	22
Gilbert	Jonas	8, rue des Fleurs, Blois	23
Dalban	Pierre	13, av. du Général, Privas	22
...	...	...	...
Manoukian	Marianne	55, place des Bleuets, Aubagne	24

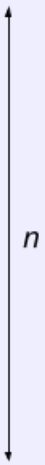


Tableau statique

Un bloc de données obéit formellement à une structure de type **tableau statique**:

- Un tableau statique est une structure séquentielle **de taille fixe**, constituée de N "cases"
- Les cases peuvent être "libres" ou "occupées".



Le tableau statique correspond à l'organisation séquentielle fondamentale des mémoires informatique :

- taille limitée du support matériel
- données accessibles grâce à leur **adresse** (position dans le tableau)
- problème de rangement (il faut conserver une information sur la position des données déjà enregistrées) ⇒ organisation *logique* des données sur le support.



- Dans un **tableau statique** T, la position des cases reste fixe au cours de l'utilisation :
  - `T[i]` désigne toujours la même zone mémoire
  - Analogie : les pages d'un cahier



- Dans un **tableau dynamique** (liste python par exemple), la position des cases varie au cours de l'utilisation :
  - `T[i]` ne désigne pas toujours la même zone mémoire
  - Analogie : briques de lego, puzzle glissant



Structure de stockage

Les mémoires informatiques sont des structures statiques. Une fois les données sauvegardées sur le disque, il est difficile d'insérer ou de supprimer des éléments.

La difficulté consiste à définir une *organisation logique* du tableau permettant de gérer efficacement un tel ensemble (qui peut être de grande taille) :

- savoir où enregistrer les données
- savoir comment les retrouver

On parle de *structure de stockage*. Une telle structure doit permettre :

- d'ajouter des données
- d'effacer des données
- d'accéder rapidement à une case particulière

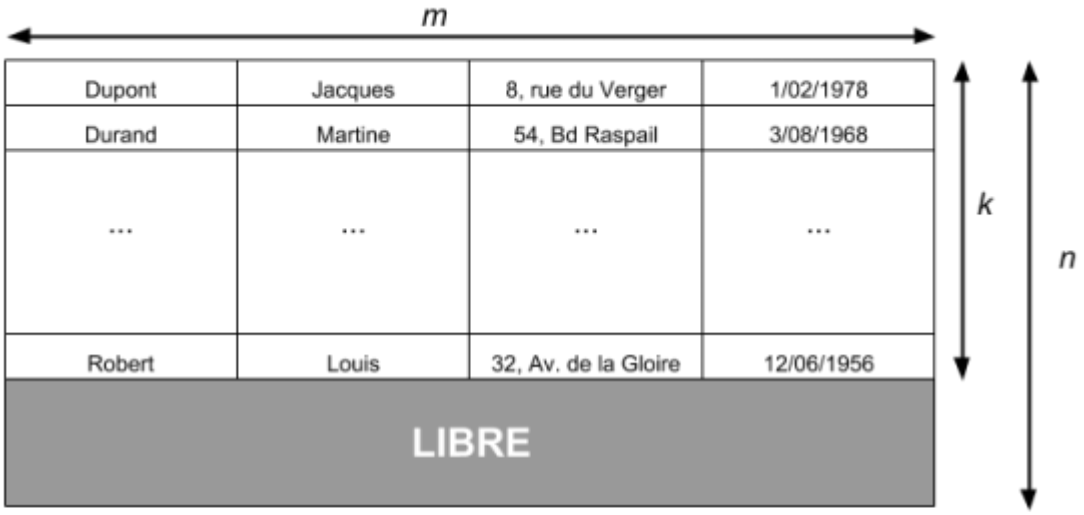
Stockage dense

On considère un ensemble de k données stockés dans un tableau de données statique T de N cases (avec $0 \leq k \leq N$).

Remarques :

- Les cases du tableau sont numérotées de 0 à $N-1$
- Les données sont de type quelconque mais chaque case ne peut contenir qu'une donnée
- Si i est un indice de case, $T[i]$ désigne le contenu de la case
- N est fixé mais k varie en fonction du nombre de données stockées

**Propriété ** : les données sont stockés dans les k premières cases du tableau. Ainsi, les cases de à $k-1$ sont occupées et l'indice k désigne la première case libre.



Opérations fondamentales :

- insertion de données : O(1)
- recherche de données : O(k)
- suppression de données : O(k)

Stockage distribué

On conserve une table des cases libres

- **Index bitmap** :
  - Chaque bit correspond à une case (libre/occupé).
  - Lors de l'écriture d'une nouvelle donnée (allocation), on fait passer le bit de 0 à 1
  - Lorsque la donnée est effacée, le bit correspondant passe de 1 à 0

$$B = \underset{0}{0}{10010100100.....}\underset{n-1}{01}$$

- **Table d'allocation**:
  - On considère un tableau de taille n dans lequel $k < n$ cases sont occupées.
  - Chaque *bloc de données* d est indexée par l'adresse $i < n$ donnant sa position dans le tableau
  - On connaît également sa taille m.
  - La *table d'allocation* donne pour chaque bloc :
    - sa position i
    - le nombre m de cases occupées

Stratégies d'allocation

Stratégie par bloc: on alloue des blocs composés de plusieurs cases (S'il existe des blocs libres consécutifs, ils sont fusionnés en un seul)

On souhaite :

- minimiser le nombre de blocs libres
- maximiser la taille des blocs libres

- Plus proche choix : la liste des blocs libres est parcourue jusqu'à trouver un bloc de la taille demandée (ou sinon, le premier bloc de taille supérieure, qui est alors découpé en deux blocs).
  - first fit : le premier bloc suffisamment grand pour les besoins
  - best fit : le plus petit bloc qui ait une taille au moins égale à la taille demandée
  - worst fit : le plus grand bloc disponible (qui est donc découpé)



**exercice **

Écrire un algorithme permettant d'insérer une donnée d de taille m dans le premier bloc libre disponible (pensez à mettre à jour la table d'allocation B).

Autre stratégie (allocation rapide): On alloue des blocs de 1,2,4,8,...2K pages. Pour une taille donnée 2i-1 < n < 2i, on commence par chercher les blocs de taille 2i, puis 2i+1, ... jusqu'à 2K , en divisant ces blocs le cas échéant.

Problème des stratégies par bloc: s'il y a trop de données, on obtient des blocs de taille 1 –> nécessité de réorganiser (défragmenter)

1.3 Fichiers et répertoires

La mémoire secondaire n'est pas directement accessible (les programmes n'ont pas la possibilité d'aller écrire directement sur le disque). Le système d'exploitation assure ainsi l'indépendance du programme par rapport à l'emplacement des données, au travers d'instructions d'entrée/sortie spécialisées.

Pour que les programmes puissent écrire sur le disque, on introduit des objets intermédiaires : les fichiers. Un fichier est caractérisé par (nom, emplacement (volume, arborescence), droit d'accès, taille,...). Il s'agit d'une entité logique. Tout programme utilisant un fichier passe par le système d'exploitation qui, à partir des informations, détermine la localisation des données sur le support.



A retenir :

- Un fichier est une référence vers un ou plusieurs blocs de données, enregistrés sur un support physique.
- Un fichier est caractérisé par son descripteur, constitué de son nom, son chemin d'accès, ses droits d'accès (lecture/écriture/exécution) selon les utilisateurs, sa position sur le support, sa taille, etc...

- La gestion des fichiers est une des fonctions essentielles des systèmes d'exploitation :
  - possibilité de traiter et conserver des quantités importantes de données
  - possibilité de partager des données entre plusieurs programmes.



** Opérations de base ** :

- *Ouverture* : initialisation d'un **flux** en lecture ou en écriture
- *Lecture* : consultation des lignes l'une après l'autre (à partir de la première ligne), dans l'ordre où elles ont été écrites sur le support
- *Ecriture* : ajout de nouvelles données à la suite ou en remplacement des données existantes

Volume

Le volume est le support sur lequel sont enregistrées les données. On parle de mémoire secondaire (Disque dur, disquette,CD-ROM, etc...). Un volume est divisé en pistes concentriques numérotées de 0 à n (par ex n = 1024). Chaque piste supporte plusieurs enregistrements physiques appelés secteurs, de taille constante (1 secteur = 1 page).



Page (ou secteur)

Les pages sont les unités de base pour la lecture et l'écriture. une page est une zone contiguë de la mémoire secondaire qui peut être chargée en mémoire centrale en une opération de lecture. Taille standard : une page = 1-2 ko.

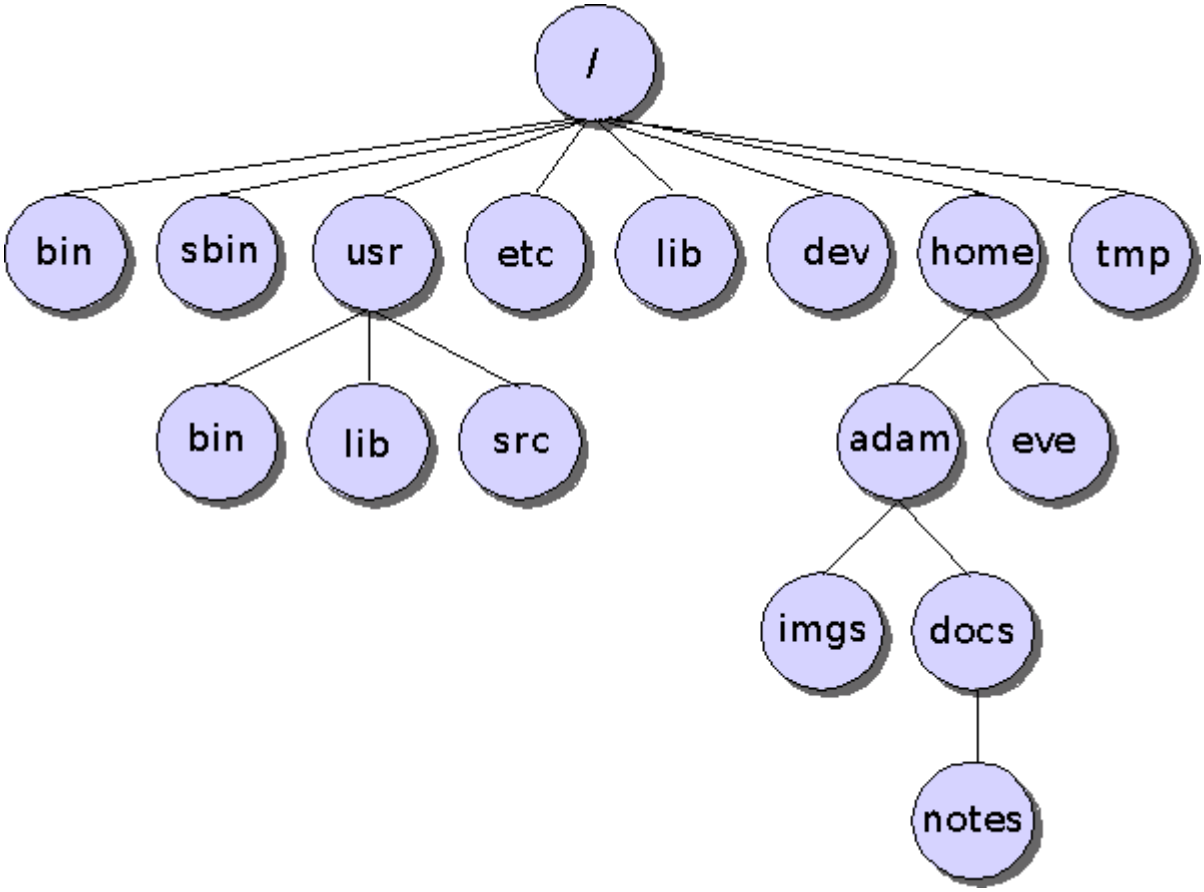
La mémoire secondaire est donc organisée comme un tableau de pages : (T[0],...,T[L-1]), où L est le nombre de pages. Chaque page fait m octets. Chaque page peut être libre ou occupée.

** Répertoires ** Chaque volume possède un numéro appelé le label du volume. Tous les descripteurs de fichiers sont regroupés dans une table des matières appelée Répertoire (Directory).

Remarque : cette table des matières est en fait un fichier dont le descripteur est contenu dans le label du volume.

Organisation hiérarchique :

- Lorsque le nombre de fichiers est élevé, les descripteurs de fichiers sont classés dans plusieurs répertoires, organisés sous une forme arborescente.
- Le répertoire de plus bas niveau hiérarchique est appelé racine → chemin d'accès (path)



Consultation des données : lecture d'un fichier (Read)

Méthode traditionnelle pour le traitement de fichiers de petite taille. La consultation des données nécessite d'ouvrir une "communication" entre le programme et les fichiers. Ce canal de communication permet de recevoir un flux de données.

Pour établir la communication, il faut connaître : le "chemin d'accès" aux données (disque dur local) l'"adresse" des données (lorsqu'il s'agit de données stockées sur un serveur distant)

L'opération d'ouverture de fichier initialise un descripteur de fichier, qui sert à désigner (dans le programme) le fichier sur lequel on travaille, et d'accéder au contenu du flux.

Ouverture simple:



Python

```
f = open('/chemin/vers/mon/fichier.dat','r')
```

Le deuxième argument représente le mode d'ouverture, ici 'r' représente une ouverture en mode lecture.

Ouverture avec test :

Il est important de vérifier que cette opération d'ouverture s'effectue correctement avant de poursuivre le programme (nombreuses possibilités d'erreur : fichier effacé, erreur de nom, pas de droits de lecture,...). On utilise une instruction de test spécifique pour vérifier que l'ouverture du fichier s'est correctement effectuée, de type `try...catch...` (essaye ... sinon ...) permettant de prévoir une action de secours lorsqu'une une opération "risquée" échoue.



Python

```
try :
  f = open('/chemin/vers/mon/fichier.dat','r')
except IOError:
  print "Erreur d'ouverture!"
```

Lorsque l'opération d'ouverture est réalisée avec succès, le flux de données devient accessible en lecture (les premières lignes du fichier sont chargées en mémoire et une tête de lecture se positionne sur le premier caractère de la première ligne). Il ne reste plus qu'à lire les données.

La consultation des données s'effectue séquentiellement à l'aide de l'opérateur de lecture `readline`. Chaque appel à cet opérateur charge les nouvelles données et déplace la tête de lecture vers les données suivantes. Cette opération peut être effectuée plusieurs fois jusqu'à atteindre la fin de fichier.



Algorithmes de bas niveau (niveau système d'exploitation)

Remarque : Lecture/Ecriture

Les opérateurs lire_ligne (readline) et écrire_ligne (write) ne travaillent pas directement sur les données du fichier. Les données du fichier sont chargées en mémoire centrale dans une mémoire "tampon". L'objet f servant à décrire le fichier a comme attributs :

- t : table des pages,
- i : numero de la page courante,
- p : tableau d'octets de la page courante (mémoire tampon),
- j : position dans la page courante (tête de lecture).

Lors des opération de lecture, la mémoire tampon est mise à jour au fur et à mesure qu'on avance dans la lecture du fichier par des opérations de lecture sur le disque. En général, plusieurs pages sont chargées en avance. Lors d'une opération d'écriture, la mémoire tampon reçoit les nouvelles données à écrire. Ces données sont effectivement écrites sur le disque lorsque le tampon est suffisamment rempli ou lors de l'opération de fermeture. Au moment de l'écriture effective, le système d'exploitation fait appel à un opérateur d'allocation pour choisir le "meilleur" bloc où stocker les données.

Si on suppose que les données sont rangées sous la forme d'un tableau de lignes, chaque opération

de lecture consiste à consulter une ligne du tableau, et à positionner la tête de lecture sur la ligne suivante.



Exemples : lecture de données texte : chaque opération de lecture lit tous les caractères jusqu'au caractère "fin de ligne".

__ 1. Lecture d'une ligne unique :__

Python

```
s = f.readline()
```

__ 2. Lecture de toutes les lignes (la lecture s'effectue dans une boucle) + affichage de la ligne:__

Python

```
for s in f :
  print s
```

Enregistrement des données : sauvegarde dans un fichier (Write)

L'opération de sauvegarde des données est l'opération complémentaire de la lecture. De nouvelles données doivent être enregistrées sur le disque dur en vue d'une consultation future. Le format de sauvegarde peut être de type texte ou de type binaire. Nous présentons ici la sauvegarde des données dans des formats texte.

Comme dans le cas de l'opération de lecture, il faut au préalable définir dans le programme un descripteur de fichier servant de point d'entrée pour les opération d'écriture. On effectue ce qu'on appelle une ouverture en mode "écriture".

Python

```
try :
  f = open('monfichier.dat','w')
except IOError:
  print "Erreur d'ouverture!!"
```

On notera qu'il existe en python (ainsi qu'en C, C++, ...) plusieurs modes d'ouverture exprimés par le deuxième argument de la fonction open. On retiendra le mode 'w' (création d'un nouveau fichier vide) et le mode 'a' (ajout de nouvelles données à la suite d'un fichier déjà existant).

La sauvegarde dans le fichier s'effectue à l'aide d'un opérateur d'écriture. Dans le cas des chaînes de caractères, l'opérateur d'écriture sauvegarde ces données à la suite des données déjà écrites.

Python

```
f.write("Bonjour!\n")
```

La sauvegarde de données nécessite d'effectuer un choix sur le mode d'encodage, obéissant en général à une norme bien précise (csv, json, xml, etc...). Voir section 1.3.

Une fois les opérations de lecture ou d'écriture terminées, il est nécessaire de fermer le fichier. L'opération de fermeture assure que les données sont effectivement enregistrées sur le disque (et non simplement stockées dans la mémoire tampon - voir section XXX).

Voir aussi :

- Gestion des fichiers sous Unix

2 Index et Données



**Rappel **

Une *donnée informatique* est un élément d'information ayant subi un encodage numérique

- Consultable/manipulable/échangeable par des programmes informatiques
- Possibilité de la conserver sur un support de stockage numérique (CD-ROM, disque dur, SSD, ...)
    - Les informations peuvent être stockés dans un fichier (ex : fichier csv).
- Un jeu de valeurs encodé et enregistré est appelé un *enregistrement*

Pour une gestion efficace des données, il est nécessaire de pouvoir identifier chaque enregistrement de façon unique.

L'indexation des données repose sur un principe simple d'étiquetage consistant à attribuer une étiquette différente à chaque enregistrement.

1. Cette étiquette peut être une suite de caractères arbitraires, un entier, ou un descripteur explicite. On parle en général de **clé** ou d'**identifiant** pour désigner cette étiquette.
2. Il existe un ordre total dans le domaine de valeurs des clés, permettant d'effectuer des opérations de tri sur les données a partie de la valeur de leur clé.

2.1 Définitions et propriétés

- L'**indexation** des données consiste à attribuer à chaque donnée distincte un **identifiant** unique.
- On parle également de *clé* de l'enregistrement:

On peut représenter l'opération d'indexation sous la forme d'une fonction. Si d est le jeu de valeurs, $k(d)$ désigne l'identifiant de ce jeu de valeurs.

Unicité/spécificité

L'indexation suppose l'existence d'une bijection entre l'ensemble des données et l'ensemble des clés, permettant d'assurer l'unité et la spécificité de la clé



- soient d_1 et d_2 deux enregistrements,
- Unicité :
  - si $d_1 = d_2$, alors $k(d_1)=k(d_2)$.
- Spécificité:
  - si $k(d_1) = k(d_2)$, alors $d_1=d_2$.

Efficacité

L'existence d'un identifiant unique pour chaque jeu de valeurs d permet la mise en œuvre d'une *recherche par identifiant* (ou recherche par clé).

La recherche par identifiant repose sur une fonction d'adressage I qui à tout identifiant k associe sa position (entrée) i dans un tableau de données: $I : k \rightarrow i$. Ainsi si k est l'identifiant servant à la recherche, l'extraction des informations se fait en 2 étapes:

- $i = I(k)$ (lecture de l'index des données)
- $d = D[i]$ (lecture des données elles mêmes)



La lecture de l'index repose sur le parcours d'une liste $$L = ((k_1, i_1), (k_2, i_2), ..., (k_N, i_N))$$ telle que $k_1 < k_2 < ... < k_N$, de telle sorte que la recherche s'effectue en O(log N) (recherche dichotomique).

Compacité

L'identifiant doit en pratique tenir sur un nombre d'octets le plus petit possible pour que la liste L puisse être manipulée en mémoire centrale. Autrement dit, il faut que :

- $|k| << |d|$

pour que :

- $|L| << |D|$



Un identifiant entier, codé sur 4 octets, permet d'indexer jusqu'à $2^{4 \times 8} \simeq 4\times10^9$ données différentes.

2.2 Utilisation

** Définir un ordre sur les données **

La présence d'un identifiant permet de définir un ordre total sur les données :

- ordre sur les entiers (identifiant entier)
- ordre alphabétique (identifiant texte)
- ordre *ASCII*bétique (chaîne de caractères quelconque)

** Lier les données **

Dans le cas des bases de données, l'identifiant constitue une *référence* vers les jeux de valeurs des tableaux de données. Une référence permet de *lier* les données d'une table aux données d'une autre table.

Exemple :

- Artistes
- Albums
- Pistes

- Pour chaque album de la table des albums, l'identifiant d'artiste (ici un numéro) permet de lier l'album avec l'artiste (ou groupe) correspondant.
- Pour chaque piste de la table des pistes, l'identifiant d'album permet de lier la piste à l'album correspondant (et donc à l'artiste correspondant par transitivité)



Exercice : donner le nom du groupe qui interprète la piste '*Paradise City*'.

** Structure d'ensemble **

L'index définit l'*appartenance* d'une donnée à un ensemble.

Soit $\mathcal{E}$ un *ensemble* de données indexées : $$ \mathcal{E} = \{d_1, d_2, ..., d_K\} $$ On a l'équivalence : $$ d \in \mathcal{E} \Leftrightarrow k(d) \in I $$

Ainsi, il ne peut y avoir de doublon car $\forall d$:

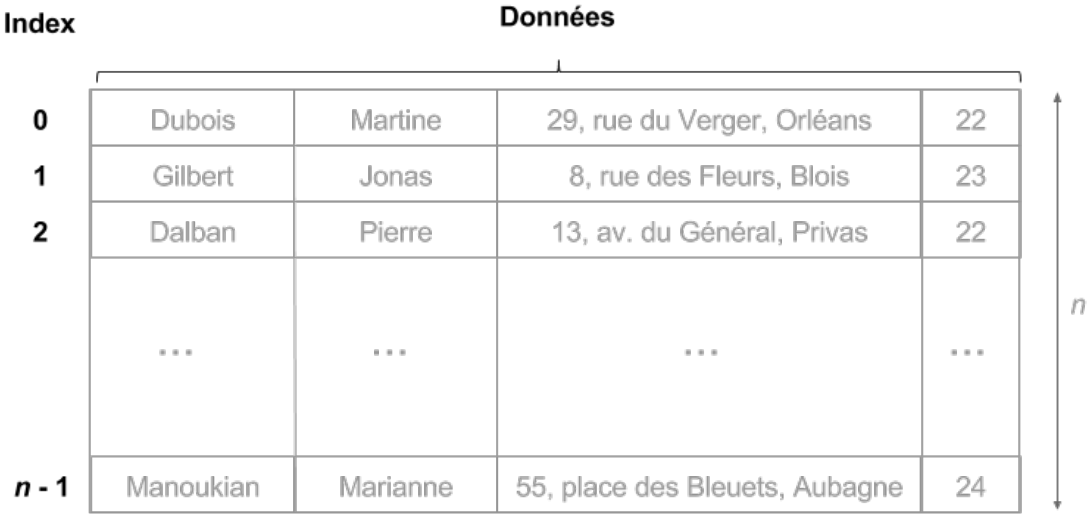
- $k(d)$ est unique
- $i = I(k(d))$ est unique ssi $d \in \mathcal{E}$ et indéfini sinon.

3 Exemples d'indexation des données

3.1 Adressage des tableaux

L'exemple le plus simple d'indexation est celui fourni par les **numéros de case** d'un tableau.

- Soit D un tableau de n lignes
- le numéro $i < n$ est à la fois l'identifiant et l'*entrée* (ou *adresse*) de la ligne $D[i]$



3.2 Maintenance centralisée d'un index

Dans le cas général, l'identifiant n'est pas égal à l'entrée!

On sépare donc l'index k de l'entrée i:

- k est l'identifiant (ou clé) de la donnée d. Il s'agit d'une valeur numérique quelconque.
- i est l'entrée de la donnée d, correspondant à sa position dans le tableau de données.



Lors de l'ajout de nouvelles données, il est nécessaire de définir une méthode permettant d'assurer:

- l'*intégrité* de l'index
- l'unicité de l'identifiant

Il existe différentes méthodes permettant d'assurer l'intégrité de l'index:

- Le programme maintient une liste triée des identifiants déjà utilisées. Lors de l'ajout d'une nouvelle donnée, il s'assure que l'identifiant n'est pas déjà présente dans la liste.
  - Coût :
    - $O(n)$ en mémoire
    - $O(\log n)$ pour l'ajout
- Dans le cas où les identifiants sont des numéros (entiers), il est possible d'utiliser un compteur qui s'incrémente à chaque nouvelle insertion.
  - Coût :
    - $O(1)$ en mémoire
    - $O(1)$ pour l'ajout



Exemples d'indexation centralisée :



- numéro INE (étudiants)
- numéro URSSAF (sécurité sociale)
- numéro d'immatriculation (véhicules)
- numéro de compte en banque
- code barre
- etc.

3.3 Indexation pseudo-aléatoire : les fonctions de hachage

Utilisation d'une *fonction de hachage* :

- qui "calcule" la valeur de l'identifiant à partir des valeurs du jeu de valeurs à insérer.
- La fonction de hachage doit être telle que la probabilité de choisir le même identifiant pour deux données différentes soit extrêmement faible.

Attribution d'un identifiant arbitraire entre 0 et n-1

- Etape 1 : **transcodage** binaire des données
  - `i = int.from_bytes(d, byteorder='big')`
  - avantage : les données différentes ont un code entier différent
  - mais : |i| = |d|

```
s = 'paul.poitevin@centrale-marseille.fr'
d = bytes(s, 'ascii')
i = int.from_bytes(d, byteorder='big')
print("i =", i)
```

donne :

```
i =
852806586114976881510056778471769180697471859150728801241505293627173168229624211058
```

- Etape 2 : **réduction** du code
  - Méthode 1 : le modulo n (reste de la division entière par n)
    - `k = H(i) = i mod n`
    - Avantage :
      - $|k| << |d|$
    - Inconvénient:
      - deux données différentes peuvent avoir le même code
      - ce codage revient à sélectionner les bits de poids faible
      - deux données proches ou très similaires auront un index proche ou similaire : si `j = i + 1`, `H(j) = H(i) + 1` (presque toujours)
      - —> il faut prendre n *premier*
    - $n = 2^{32} = 4294967296$:
      - H_code(`paul.poitevin@centrale-marseille.fr`) = 1697539698
      - H_code(`martin.mollo@centrale-marseille.fr`) = 1697539698
    - $n = 67280421310721$ (premier):

        - H_code(`paul.poitevin@centrale-marseille.fr`) = 36148249469507
        - H_code(`martin.mollo@centrale-marseille.fr`) = 65330032132071
    - Méthode 2 : combiner produit et modulo – soient m et n deux nombres premiers entre eux
        - `k = H(i) = (i * m) mod n`
        - Avantage :
            - $|k| << |d|$
            - deux entiers proches donneront auront des codes très différents : si `j = i + 1`, `j * m - i * m = m`
        - Inconvénient :
            - deux données différentes peuvent avoir le même code
                - le produit `i * m` peut etre coûteux à calculer

- Méthode 3 : Hachage cryptographique :
    - Le hachage cryptographique est construit de telle sorte qu'il soit très difficile de trouver un entier `j != i` tel que `H(i) = H(j)`.
    - Un tel code est appelé une "signature".
        - Exemples :
            - MD4
            - SHA

Exemple

Le gestionnaire de bases de données MongoDB utilise une indexation des données par clé cryptographique.

4 Généralisation : multi-indexation



TODO

5 Structures de données pour l'indexation

5.1 Liste triée



TODO

5.2 Index bitmap



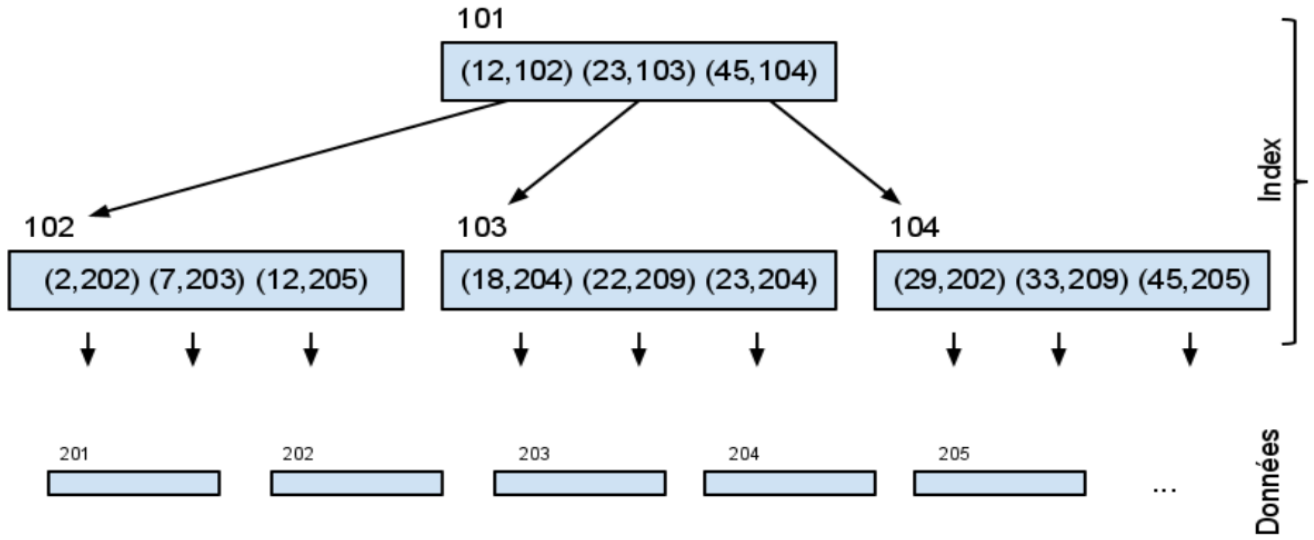
TODO

5.3 Les B-arbres

Que faire lorsque l'index ne tient pas en totalité dans la mémoire centrale ?

- L'index est découpé en "blocs"
- Les blocs sont organisés sous la forme d'un arbre (B-arbre = "*Balanced Tree*")

Considérons un index composé de couples (clé, numero de page). On suppose que l'index est également découpé en pages, chaque page d'index contenant au maximum b clés. Si l'index comporte beaucoup de pages, il est intéressant de l'organiser hiérarchiquement en pages et sous-pages, selon une organisation appelée "B-arbre" (*Balanced Tree*):



- chaque noeud de l'arbre contient une liste croissante de couples (clé, numéro d'une sous-page)
- chaque clé est dupliquée dans sa sous-page :
  - les clés contenues dans la sous-page sont inférieures ou égales à elle,
  - les clés contenues dans la sous-page sont strictement supérieures à celles de la sous-page précédente.
  - les feuilles contiennent des couples (clé, numéro de la page du tableau de données)

<!-- <note> **algo : lecture de l'Index** * paramètre d'entrée : k I ← charger la racine de l'arbre tant que I n'est pas une feuille : k' ← première clé de I tant que k > k' k' ← clé suivante I ← charger sous-page de k' </note> -->

Remarque : Pour que l'accès aux données soit efficace,

- il faut que l'arbre soit le moins profond possible : arbre "équilibré".

<!-- * Dans ce cas, * chaque noeud a ''b'' fils, * et la profondeur de l'arbre est de l'ordre de logb(N). * Pour charger la page contenant le tuple cherché, * il faut donc logb(N) + 1 lectures sur disque. * En pratique, il existe des algos permettant d'assurer que chaque noeud contient entre b/2 et b clés (sauf la racine). -->



Voir :

- http://fr.wikipedia.org/wiki/Arbre_B
- http://en.wikipedia.org/wiki/B-tree)

From:
https://wiki.centrale-med.fr/informatique/ - **WiKi informatique**

Permanent link:
https://wiki.centrale-med.fr/informatique/tc_info:2020_cm_index

Last update: **2021/01/27 21:46**