SQL - Module 2: Introduction - TD : *plain text database*: la base de données *la plus simple au monde*

Mise en place de la base de données: un programme et un fichier

Avec votre éditeur de texte favori, créer un script bash `db.sh` (ou avec votre langage de script favori)

```
#!/bin/bash

#Une simple fonction de test
greet(){
    echo "Hi $1 !"
}

db_set(){
    echo "$1,$2" >> database
}
db_get(){
    #1. On récupère la ligne qui nous intéresse dans le fichier (par clef)
    #2. On substitue et affiche sur al sortie standard
    #  la clef par une chaîne vide => affiche la valeur
    #3. On n'affiche que la dernière ligne
    #   des résultats qui ont match la clef
    grep "^$1," database | sed -e "s/^$1,//" | tail -n 1
}
```

Rendre le script exécutable et charger les fonctions

```
# se placer dans le répertoire où se trouve votre script avec cd
# rendre le script exécutable
chmod +x db.sh
# charger les fonctions dans l'environnement du shell
source db.sh
# test
greet John
```

Utiliser la base de données

Insérer (**écrire**) des données avec `db_set`

```
# insertion d'une simple chaine de caractères
db_set foo bar
# insertion d'une chaine au format JSON
db_set 1 '{"firstName":"John", "lastName":"Doe"}'
db_set 1 Baz
# insertion d'une chaine de type tableau
db_set 42 "[1,2,3]"
```

Récupérer (**lire**) des données avec `db_get`

```
db_get 42
db_get 1
```

Analyse de ce modèle simple

- très *performant en écriture*: méthode *append to a file* (ajouter à la fin) est ce qu'il y a de plus performant. On parle de *log* function (*append-only*). C'est en gros ce que font les SGBD.
- *très mauvais en lecture*. Les accès disques se font dans le programme `grep`. Un accès disque **coûte cher en temps d'accès** (*Input/Output* ou *I/O*). Les SGBD mettent les données en cache (en mémoire). Quelques ordres de grandeur:
  - Accès disque ~ 10ms pour disques magnétiques et 0.1ms pour disques SSD
  - Accès mémoire RAM ~ de 10-7s à 10-8s (10 à 100 nanosecondes ! On gagne ici un facteur 10000 !)
  - Accès cache processeur ~ 10-7 (10 nanosecondes max)
- très *mauvais en lecture*: augmente linéairement avec le nombre d'enregistrements (de lignes ici) `n`. Pour améliorer ça, on pourrait utiliser un **index**. L'index est une structure de données en arbre et permet d'augmenter les performances de manière *radicale* en lecture. Ajoute cependant un coût supplémentaire en écriture (car il faut maintenir l'index à jour)
- on a fait *des choix forts* sur notre "modèle de données" (comment elles sont représentées dans notre base de données): clef/valeur, chaque écriture sur une nouvelle ligne, pas d'overwrite/update, etc.
- un•e utilisateur•ice du système *doit comprendre le fonctionnement du moteur* pour pouvoir l'utiliser correctement
- sécurité: on peut gérer les droits d'exécution du script `db.sh` et de lecture/écriture du fichier `database` (ok)
- problèmes: performances en lecture, on ne sait pas gérer les accès concurrents (si un autre *user* écrit en même temps ?), gestion de données corrompues, etc.

Conclusion

Implémenter un moteur de base de données *est une tâche complexe* faisant intervenir à la fois de nombreuses **difficultés techniques** (accès, cache, gestion de la mémoire, gestion des pannes, etc.) et **conceptuelles** (modèles et représentation des données).

C’est pour cette raison que ces programmes sont développés depuis plus de 40 ans (énormément de travail et d’argent investis) et que le standard SQL a été créé.

> Ne le faites pas vous même ! (Hors raisons pédagogiques ou si vraiment vous avez une bonne raison de le faire.)

Ressources

- Designing data intensive applications, de Martin Kleppmann, 2017, publié par O’Reilly (Voir Chapitre 3: *Storage and Retrieval*) une référence absolue sur les systèmes d’information et sur les SGBD
- Sed, an introduction and Tutorial, Bruce Barnett, l’éditeur de flux par excellence. Comme l’indique l’auteur de ce site web, sed est un outil merveilleux, mais sa documentation laisse à désirer. Privilégiez ce site pour vous y initier.