· Projet de C++PR72

Le contexte du projet est une base de données gérée par votre programme. La base de données est constituée de plusieurs tables, chacune gérée par des fichiers portant le nom de la table gérée et une extension en fonction du rôle du fichier (structure, index, contenu). La base de données est dans un répertoire portant son nom, et chaque ensemble de fichiers d'une table est dans un sous-répertoire du même nom, contenu par le répertoire de la base de données elle-même.

Le programme qui gère la base de données gère les arguments suivants :

- -d suivi d'un nom, le nom de la base de données à ouvrir/créer
- -l suivi d'un chemin, le chemin vers le répertoire parent de la base de données (i.e. le répertoire qui contient celui de la base de données)

Pour plus de clarté, le programme est décomposé en plusieurs parties correspondant chacune à un aspect de la gestion de la base de données.

Les sections suivantes définissent les différents éléments du projet, qui seront les critères d'évaluation du projet. Ce document est actuellement incomplet et sera mis à jour dans les jours à venir.

Une base de données est un outil permettant la gestion de données structurées ou non. Dans ce projet, nous nous intéressons à créer une base de données simplifiée manipulée par un sous-ensemble de SQL.

Pour vous permettre de mener à bien ce projet, vous devez comprendre comment les données seront stockées par la base de données, quelles sont les commandes SQL que vous devrez implémenter, et comment faire l'interface entre le contenu de la base de données sur votre support de stockage et le résultat du parsing SQL.

Des fichiers avec les définitions du code vous sont fournis avec ce README afin de garantir que la structure de votre programme est correcte.

o Principe du programme

Le programme que vous devez réaliser fournira sous une forme simplifiée les moyens d'interagir avec une base de données. L'interface de l'utilisateur avec la base de données étant faite avec une variante du langage SQL, vous serez amenés à implémenter les 4 étapes suivantes :

- Parse : cette étape consiste à parcourir la requête SQL sous forme de texte, pour en extraire les différents éléments et les stocker dans une représentation en mémoire (sous forme de structures de données). Cette étape s'assure que la syntaxe de la requête est correcte.
- Check : cette étape consiste à vérifier que la sémantique de la requête est correcte (tables et champs qui existent, types des données, etc.)
- Expand : cette étape complète les éventuelles données manquantes dans la requête (par exemple : transformation du caractère * en l'ensemble des champs de la table, etc.)

- Execute : la requête, maintenant analysée, vérifiée et complète, peut être exécutée. Il s'agit à cette étape de lire/écrire les données nécessaires sur le support de stockage contenant la base de données.

o Représentation de la base de données sur le support de stockage

Pour les besoins de ce projet, nous considérons qu'une base de données est stockée dans un répertoire qui lui est propre et qui porte son nom, i.e. la base de données inventaire est stockée dans un répertoire nommé "inventaire". Une base de données est constituée de tables, qui sont des ensembles structurés de données de même nature. Dans le répertoire de la base de données, chaque table dispose de son propre répertoire. Ce répertoire de table contient 3 ou 4 fichiers relatifs à cette table : le fichier de définition, le fichier d'index, le fichier de contenu et le cas échéant, le fichier de clé.

Le fichier de définition

Ce fichier contient la description des champs de la table. La liste des champs est définie par un champ par ligne avec son type, défini par l'énumération `field_type_t` (c.f. la définition des types utilisés par le programme). Ce fichier permet de connaître l'ordre et le type des champs stockés par la table.

Une ligne définissant un champ est écrite de la manière suivante : `N nom` où `N` est le numéro de type de champ issu de l'énumération `field_type_t`, et `nom` est le nom du champ.

§ Le fichier d'index

Le fichier d'index est composé d'enregistrements de taille fixe. Chaque enregistrement est composé de 3 champs :

Active	Offset	Length

dont voici la signification :

- active : ce champ est défini sur un octet (type uint8_t). Il a pour valeur zéro si cet index n'est pas actif (il peut donc être réutilisé), et une valeur différente de zéro dans le cas contraire. Une valeur de zéro signifie que le contenu correspondant doit être ignoré en lecture et peut être réutilisé en écriture.
- offset: ce champ est défini sur 4 octets (type uint32_t). Il définit la position de ce champ dans le fichier de contenu. Il définit la position en octets à partir du début du fichier.
- length: ce champ est défini sur 2 octets (type uint16_t). Il définit la taille de l'enregistrement courant. Cette taille sera toujours la même pour une table donnée (donc chaque ligne d'index a une valeur length identique, pour simplifier l'accès aux données)

Ce fichier est utile pour accéder aux enregistrements de la table.

§ Le fichier de contenu

Ce fichier contient les données de la table. Chaque enregistrement est stocké dans l'ordre de la définition de la table (obtenue en lisant son fichier de définition). Les données de type float, int et primary key sont stockées sous leur forme binaire. Les chaînes de caractères sont stockées intégralement (il s'agit de tableaux de longueur fixe). Tous les enregistrements d'une table ont donc une longueur identique.

Par exemple, avec les valeurs données ci dessous (longueur des chaînes de caractères égale à 150), les enregistrements d'une table définie par un int (type SQL), un float et deux text auront une taille de 316 octets, avec accès à l'entier au premier octet de l'enregistrement courant, accès au nombre réel au neuvième entier, accès à la première chaîne de caractères au 17ème octet et accès à la seconde chaîne de caractères au 167ème octet.

Le fichier de clé

Si un champ de la table est défini du type `primary key`, un quatrième fichier est créé : il contient une valeur binaire d'un `unsigned long long` initialisé à `1` et incrémenté à chaque insertion d'un enregistrement dans la table, lorsqu'aucune valeur pour cette clé n'est spécifiée. Dans le cas contraire, la valeur est mise à jour avec la valeur maximale de ce champ dans la table, incrémentée de `1`;

§ Exemple

L'exemple ci dessous montre l'arborescence d'une base de données nommée `db` et contenant deux tables : `une_table` et `another_table`. La table `une_table` comporte un champ de type `primary key` alors que la table `another_table` n'a que des champs `text`, `int` ou `float` :

```
db
├── another_table
│   ├── another_table.data
│   ├── another_table.def
│   └── another_table.idx
└── une_table
    ├── une_table.data
    ├── une_table.def
    ├── une_table.idx
    └── une_table.key
```

o Langage SQL simplifié

Interagir avec une base de données se fait notamment avec le langage SQL (Structured Query Language). Dans ce projet, vous serez amenés à manipuler les requêtes (simplifiées) suivantes :

- CREATE TABLE
- INSERT
- SELECT
- DELETE
- UPDATE
- DROP TABLE
- DROP DATABASE ou DROP DB

Une requête SQL se termine toujours par un caractère `';'`.

§ Création d'une table

La structure d'une commande création de table est la suivante : `CREATE TABLE table_name (field1_name field1_type, ...fieldN_name fieldN_type);`

Le nom d'une table, et les noms de champs obéissent aux même règles que les noms de variables en C. Les types des champs seront les suivants :

- int pour un entier (représenté par un long long)
- primary key pour un entier non signé (unsigned long long) avec incrémentation automatique.
- float pour un nombre flottant (type double dans l'implémentation)
- text pour du texte.

§ Suppression d'une table

La suppression d'une table se fait avec la commande SQL suivante : DROP TABLE table_name;. Elle permet de supprimer la table nommée table_name.

§ Suppression d'une base de données

Le principe est similaire à celui de la suppression d'une table : on supprime une BDD avec la commande SQL suivante : DROP DATABASE db_name;.

§ Insertion d'un enregistrement dans une table

La commande SQL utilisée est la suivante : INSERT INTO table_name (field1, ... fieldN) VALUES (value1, ... valueN);

Cette commande ajoute à la table nommée table_name les valeurs value1 à valueN dans les champs field1 à fieldN (donc le champ field1 aura la valeur value1, le champ field2 aura la valeur value2 et ainsi de suite). Les valeurs sont encadrées par des quotes simples quand il s'agit de chaines de caractères.

§ Sélection d'enregistrements à partir d'une table

La commande SQL utilisée est la suivante : SELECT * FROM table_name WHERE condition; ou SELECT field1, ... fieldN FROM table_name [WHERE condition];

Cette commande affiche l'ensemble des champs des enregistrements de la table satisfaisant à la clause WHERE. Si cette dernière est absente, l'ensemble des enregistrements de la table sont

affichés. Seuls les champs listés entre les mots-clé SELECT et FROM sont affichés. L'étoile * est un méta-caractère qui indique que l'on souhaite afficher tous les champs de la table.

§ Suppression d'un enregistrement dans une table

La commande SQL utilisée est la suivante : DELETE FROM table_name [WHERE condition];

Cette commande supprime tous les enregistrements de la table correspondant à la clause WHERE. En l'absence de cette dernière, toute le contenu de la table est supprimé (mais pas la table elle-même).

§ Modification d'un enregistrement dans une table

La commande SQL utilisée est la suivante : UPDATE table_name SET field1=value1, ..., fieldN=valueN [WHERE condition];

Cette commande affecte les nouvelles valeurs définies après SET à l'ensemble des enregistrements correspondant à la clause WHERE. En l'absence de cette dernière, tous les enregistrements sont modifiés.

§ Clauses WHERE

Dans les 3 requêtes suivantes, il peut être nécessaire de filtrer des champs pour les visualiser ou les modifier. C'est le rôle de la clause facultative WHERE (le fait qu'elle ne soit pas requise est matérialisée ci dessous par des [ ] de part et d'autre de la clause WHERE).

Cette clause est composée du mot-clé WHERE suivi d'un ensemble de conditions. Nous nous contenterons ici de ne combiner ensemble soit que des AND, soit que des OR. Une condition sera donc de la forme :

- WHERE field1=value1 pour une clause WHERE sur un seul champ.
- WHERE field1=value1 AND ... AND fieldN=valueN pour une clause WHERE nécessitant que toutes les conditions soient remplies.
- WHERE field1=value1 OR ... OR fieldN=valueN pour une clause WHERE nécessitant qu'au moins une des conditions soit remplie.

Pour gérer des clauses WHERE, vous devrez implémenter la fonction suivante : int create_filter_from_sql(char *where_clause, s_filter *filter). La fonction va lire la requête, en extraire le nom de la table, et construire un filtre stocké dans la variable pointée par filter.

o Structure du projet

Ce projet sera structuré suivant une hiérarchie de classes permettant un flux simple depuis la saisie du SQL jusqu’à son exécution si toutes les étapes sont valides.

SQL

Votre programme principal contiendra une boucle en attente de la saisie de la commande de l’utilisateur. Dans le cas où la commande est *exit*, le programme se termine. Dans tous les autres cas, le programme va essayer de parser l’entrée utilisateur comme du SQL. Pour cela, vous pouvez écrire une classe mère purement virtuelle nommée sql_query, de laquelle hériteront des classes

correspondant à chaque requête définie ci-dessus : select_query, update_query, etc. La classe sql_query définit l'interface suivante :

+ parse(std::string user_sql): void

+ check() : void

+ expand() : void

+ execute() : void

Ces méthodes lèvent des exceptions si leurs opérations ne peuvent pas aboutir (syntaxe SQL incorrecte, incohérence de la requête avec la structure de table/base, impossibilité d'étendre la requête, impossibilité de l'exécuter). De cette façon, il vous est possible de faire quelque chose comme suit dans votre boucle de programme principal :

```
string sql;
cin >> sql;
if (sql == "exit")
      break;
try {
      unique_ptr<sql_query> query = query_factory::generate_query(sql) ;
      query->check();
      query->expand();
      query→execute();
} catch (unknown_sql_exception &e) {
      /* Do something */
} catch (sql_exception &e) {
      /* Do something */
} catch (check_exception &e) {
      /* Do something */
} catch (expand_exception &e) {
      /* Do something */
} catch (execute_exception &e) {
      /* Do something */
} catch (exception & e) {
}
```

Chaque requête aura ses propres attributs en fonction de son type (nom de la base de données pour drop database, nom de la table pour les autres, ainsi que les listes de champs, de types, etc.). Les attributs peuvent être également définis par des classes lorsqu'ils sont composés de plusieurs informations et/ou ils peuvent prendre des types de valeurs variables (il est alors possible d'utiliser soit une hiérarchie de classes, soit des templates).

L'exemple de code contient également une factory de requête qui sera chargée d'appeler les fonctions de parsing de chaque type de requête possible jusqu'à en trouver une qui aboutisse à un parsing correct, ou jusqu'à ce que toutes les requêtes possibles aient été testées (il est alors possible de lancer une exception de type *unknown_sql_exception*.

Gestion des tables

Cette section définit comment les tables et les bases de données vont être stockées. Concernant la base de données, c'est simple : il s'agit d'un répertoire du même nom que la base, qui contiendra toutes ses tables. Chaque table sera composée d'un répertoire à son nom, contenant 3 à 4 fichiers (voir plus haut pour la description détaillée). La hiérarchie de classes sera centrée sur les fichiers. Une classe mère table*_file* fournira les fonctions de base (tester si un fichier existe, l'ouvrir et le

fermer). Chaque type de fichier héritera ensuite de cette classe de base pour implémenter les comportements spécifiques.

La hiérarchie résultante devrait s'approcher de ce qui suit :

Classe db_info

- current_db_path : std::string

+ setDbPath(const std::string&) : void

+ getDbPath(const std::string&) : std::string

Classe table_file

Propose *a minima* les membres suivants :

```
- source_file: std::string
+ exists(): bool
+ open(): void
+ close(): void
```

Open et close soulèvent des exceptions en cas de problème.

Classe definition_file

Hérite de table_*file*. La définition de table contient, en sus des membres de table_*file*, les membres suivants :

```
+ get_table_definition(): table_definition
+ write_table_definition(const table_definition &def): void
```

Classe index_file

Hérite de table_*file*. L'index de table contient, en sus des membres de table_*file*, les membres suivants :

```
struct index_entry __attribute__((__packed__)) {
      bool is_active;
      uint32_t position ;
      uint16_t length;
};
+ write_index_entry(const index_entry &entry, uint32_t offset): void
+ get_index_entry(uint32_t position): index_entry
```

Notez bien le __attribute__((__packed__)) de la structure qui permet de ne pas aligner (avec padding) les champs. Cela permet d'écrire et lire directement une structure de ce type dans le fichier d'index correspondant.

Dans la méthode write_index_entry, l'offset est la position du fichier en octets depuis le début, où sera écrit l'index.

Classe content_file

Hérite de table_*file*. Le contenu de table contient, en sus des membres de table_*file*, les membres suivants :

#include <cstdint>

```
+ write_record(const std::vector<uint8_t> &record, uint32_t offset): void
+ read_record(uint16_t length, uint32_t offset): std::vector<uint8_t>
```

write_record écrit la représentation binaire d’un enregistrement de table à l’offset du fichier de contenu (offset obtenu par lecture de l’index de table)

read_record lit length octets à partir de l’offset et les retourne dans un vecteur d’octets.

Classe key_file

Hérite de table*_file*. La clé de table contient (si la table contient un champ de type primary key), en sus des membres de table*_file*, les membres suivants :

```
+ get_next_key(): uint64_t
+ update_key(uint64_t last_value): void
```

get_next_key lit la prochaine valeur de clé depuis le fichier de clé et retourne cette valeur.

update_key met à jour la valeur de la clé dans le fichier à une valeur égale à l’entier long non signé immédiatement supérieur à *last_value*.

Classes additionnelles

Vous serez amenés à définir des classes de données pour faciliter votre développement. Celles ci ne sont pas définies ici car elles dépendront fortement de votre façon de résoudre les problèmes posés par ce projet, et varieront donc d’un groupe à l’autre.

La cohérence et la pertinence de vos choix entreront dans l’évaluation du projet.