Ikko-Database

https://github.com/nikifaets/Ikko-Database

Николай Пашов

7 юни 2020 г.

Съдържание

1 Въведение

1.1 Описание на задачата

Настоящият проект представлява имплементация на база данни, описана по-подробно на следния линк. Базата от данни е изградена от таблици. Поставената задача изисква таблиците да поддържат три типа записи - цяло число (*Int*), дробно число (*Double*), и низов масив (*String*), както и имплементацията на потребителски интерфейс, позволяващ базови операции върху таблиците - четене, променяне, запазване и др. (описани са подробно на горепосочения линк).

1.2 Акценти

1.2.1 Дефиниция на типовете данни

За да се работи удобно и структурирано със заданените типове данни, тяхното представяне трябва да бъде стриктно дфенирано. Това улеснява имплементацията на методи за боравене с тези данни, както и валидирането им. Правилата за записване на трите типа записи в таблиците са следните:

- ***Int*** - съдържа само символи в интервала 0-9. Пример: 123
- ***Double*** - съдържа символи в интервала 0-9 и един символ за дробна запетая - ".". Пример: 123.123
- ***String*** - може да съдържа всякакви символи, обградени в кавички. Кавички и наклонена черта се представят с наклонена черта преди тях. Пример:

```
"This\\That is a \"string\""
```

Ще бъде прочетено от програмата като низ от символи със стойност:

```
This\That is a "string".
```

1.2.2 Архитектура

Тъй като целта на проекта е да позволява разнообразни операции върху таблици с данни, създаването на подходящи модули, които да съдържат данните по време на изпълнение на програмата, е от ключова важност. Ако записите от таблиците се конвертират в удобни за работа програмни единици, то операциите за манипулация на информация стават тривиални. По тази причина е поставен голям акцент върху реализацията на подходящи основни модули, които да позволяват лесната имплементация на по-сложни опреации.

2 Реализация

2.1 Планиране

2.1.1 Файлов формат

Файл, съдържащ таблица се състои от следните задължителни компоненти в показания ред:

- Ред с името на таблицата;
- Ред с имената на колоните;
- Ред, индициращ типа на всяка колона. Типовете *{Int, Double, String}* се обозначават съответно с цифрите {0, 1, 2};

- Редове със записи. Поле с празна стойност се бележи със символния низ *"Null"*.

Примерна таблица може да се види на Таб. 1

NameInt	NameDouble
0	1
123	Null
455	34.14

Таблица 1: Примерна таблица, съдържаща записи от тип *Int* и *Double*, както и един празен запис.

2.1.2 Главни модули

Поради причините, описани в Секция 1.2.2, акцентът на реализацията до този момент бе създаване на надеждни и скалируеми модули за зареждане на данни от таблица по време на изпълнение. Някои по-ключови класове са следните:

- ***Record*** - съдържа базова информация за запис от определен тип. Наследява се от ***RecordInt***, ***RecordDouble***, ***RecordString*** и ***RecordInvalid***. ***RecordInt*** и ***RecordDouble*** не са му преки наследници, а го наследяват през абстрактния клас ***RecordNumber***. Това е така, тъй като имат споделена функционалност, която не е свързана с другите видове записи.

- ***Row*** - съдържа контейнер с обекти от тип ***Record***. Предоставя функционалност за конвертиране в тип *String*, за да се запише във файл, добавяне на колона, както и други методи за манипулация на данни.

- ***Table*** - съдържа контейнер с обекти от тип ***Row***, както и базова информация, касаеща таблица - имена и типове на колони, име на таблица и др. Предоставя функционалност за четене и записване на таблица, манипулация на записите ѝ, както и съответните валидации
- ***Parser*** - предоставя интерфейс, чрез който обекти от клас ***Table*** четат таблица от файл.
- ***Message*** - позволява изписването на съобщения в конзолния екран. Създаден е с цел разделяне на бизнес и вход/изход логиката.
- ***Database*** - Менажира базата от данни. Приема заявки за операции върху таблици, които валидира и изпълнява при възможност.
- ***CLIParser*** - Класът, който се грижи за потребителския интерфейс. Валидира въведения от потребителя вход и го препраща към съответния метод.

2.2 Имплементация

2.2.1 Прочитане и съхраняване на таблица

Във Фрагмент 1 е изобразен методът за четене на таблица от файл. Проследявайки този метод, може да се покаже целият процес за конвертиране на таблицата в програмни единици, удобни за манипулация. Следният анализ е с цел демонстрация на тези програмни единици.

```
1 void Table::read_table(std::string filename){
2
3     std::ifstream table;
4     table.open(filename);
5
6     if(!table.is_open()){
```

```
7
8           Message::FileNotFound(filename);
9           return;
10      }
11
12      std::string line;
13
14      //read table name - first line
15      std::getline(table, line);
16      name = line;
17
18      //read column names - second line
19      std::getline(table, line);
20      col_names = Parser::parse_line_str(line);
21
22      //read type data - third line
23      std::getline(table, line);
24      col_types = Parser::parse_type_data(line);
25
26
27      if(!row_types.size()){
28
29          Message::CorruptedTypeInformation(filename);
30          return;
31      }
32
33      if(row_types.size() != col_names.size()){
34
35          Message::CorruptedTypeInformation(filename);
36          return;
37      }
38
39      //read rows
40      while(std::getline(table, line)){
41
42          Row row(Parser::parse_line(line, row_types));
43
44          if(!validate_row(row)){
```

```
45
46              Message::CorruptedRow(rows.size());
47              rows.clear();
48              return;
49          }
50
51          rows.push_back(row);
52      }
53
54      table.close();
55  }
```

Фрагмент 1: Метод на клас ***Table*** за прочитане на таблица от файл.

Методът започва с опит за четене на файл и валидиране дали той е прочетен успешно. Чете се редът, съдържащ името на таблицата, после - редът, съдържащ имената на колоните в съответния файл. За тази цел се използва статичният метод *parse_line_str* на клас ***Parser***. Той конвентира ред от таблицата в обект от тип *std::vector<std::string>*, съдържащ записа от всяка една колона като тип *String*. Имплементацията му се вижда във Фрагмент 2. В него се използват други помощни методи на клас ***Parser***.

```
1  std::vector<std::string> Parser::parse_line_str(std::string line){
2
3      std::vector<std::string> line_recs;
4
5      for(int i=0; i<line.size(); i++){
6
7          if(is_separator(line[i])) continue;
8
9
10         std::string val = read_until_whitespace(line.substr(i));
11         i += val.size();
12
13         line_recs.push_back(val);
14
```

```
15      }
16
17      return line_recs;
18  }
```

Фрагмент 2: Метод на клас ***Parser*** за конвертиране на ред от таблица във вектор с елементи от тип *String*.

Следва прочитането на реда, съдържащ типовете в таблицата (ред 18-32). Методът *parse_ type_ data* работи по подобен начин като *parse_ line_ str*, само че конвертира реда не във вектор от тип *String*, ами от тип *Type*, който е *enum*, дефиниран в клас ***Record***. Той съдържа типовете, описани в 2.1.1.

Веднага след това се извършва валидация. Методът *parse_ type_ data* ще върне празен вектор ако е имало грешка във валидацията на типовете, която се извършва в метода. Тази валидация е наложителна, тъй като има ясно ограничение за типа на записите в този ред от таблицата, както и стойностите, които те могат да заемат.

Втората валидация е дали броят на типовете в таблицата съответства на броя имена на колони. Ако не съответстват, значи има грешка във файла и той не може да бъде обработен.

Следва същинското прочитане на записите. То се осъществява като всеки следващ ред от таблицата се конвертира в тип ***Row***, използвайки метода *parse_ line* на класа ***Parser***. Методът е изобразен във Фрагмент 3.

```
1  std::vector<Record*> Parser::parse_line(std::string line, std::
       vector<Type> types){
2
3      std::vector<Record*> recs;
4
5      std::vector<std::string> line_str = parse_line_str(line);
6
7      for(int i=0; i<line_str.size(); i++){
```

```
8
9           Record* rec;
10
11          if(!line_str[i].compare(NULL_REC)){
12
13              rec = Caster::type_to_rec(types[i]);
14              rec->set_empty(true);
15
16          }
17
18          else{
19
20              rec = Caster::string_to_rec(line_str[i]);
21
22          }
23
24          if(rec->get_type() == Invalid){
25
26              Message::InvalidRecord(i);
27              return std::vector<Record*>();
28
29          }
30
31          recs.push_back(rec);
32      }
33
34      return recs;
35  }
```

Фрагмент 3: Метод на клас ***Parser*** за конвертиране на ред от таблица във вектор с тип ***Record****.

Първото, което се забелязва, е че той използва метода *parse_line_str*, за да конвертира реда във вектор от тип *String*. След това всеки елемент от този вектор се конвертира в обект от клас ***Record***.

Първата валидация, която се извършва е на ред 11. Тя проверява дали записът е празен (със стойност *Null*). В този случай, връща обект като

запис от типа, който съответства на типа, който беше прочетен преди това от *parse_ type_ data*. Използва се клас ***Caster*** за създаване на клас, наследяващ ***Record*** (вж. Секция 2.1.2). Този клас е имплементиран за подобни преобразувания

Ако записът не е празен, то тогава той се конвертира от тип *String* в обект на някой от класовете, наследяващи ***Record***, описани в 2.1.2. Това се прави също от клас ***Caster***.

Методът *string_ to_ rec* връща обект на клас ***RecordInvalid*** ако записът е невалиден, тоест не съответства на нито една от дефинициите от Секция 1.2.1.

След като редът от таблицата е конвертиран в обект на клас ***Row***, методът *read_ table* прави валидация на целия ред. Това включва проверки като дали броят на записи в реда съответства на броя типове, описани в таблицата преди това; дали редът е празен и др. Ако валидацията е успешна, редът се добавя към вектора, който съхранява всички редове в таблицата.

2.2.2 Съхраняване и манипулация на записи

Съхранявани по този начин, записите от таблиците са удобни за манипулация. Разбиването на данните на по-малки структури като ***Row*** и ***Record*** позволява гъвкавост при работата с данни. Например, добавяне на нови записи може да стане лесно, използвайки класа ***Row***. Такива методи са изобразени във Фрагмент 4.

```
1 void Row::add_empty_record(Type type){
2
3     Record* new_rec = Caster::type_to_rec(type);
4     new_rec->set_empty(true);
5     records.push_back(new_rec);
6 }
7
```

```
8  void Row::add_record(Record* rec){
9
10     records.push_back(rec);
11 }
12
13 void Row::add_record(std::string val){
14
15     Record* new_rec = Caster::string_to_rec(val);
16     records.push_back(new_rec);
17 }
```

Фрагмент 4: Методи на клас ***Row*** за добавяне на нов запис.

2.2.3 Манипулация на данни от таблица

Следва пример как класът ***Database*** приема заявка и я изпълнява върху таблица. Разглежданият метод е *find_ rows_ by_ value*. И Той е интересен, тъй като съдържа доста ключови елементи, които позволяват имплементацията на подобни методи за опериране върху таблици. Той е изобразен във Фрагмент 5.

```
1  void Database::select_rows(std::string name, int col, Record* val){
2
3      Table table = load_table(name);
4      if(!table.is_loaded_correctly()) return;
5
6      std::vector<int> row_idx = table.find_rows_by_value(col, val);
7
8      Presenter::print_rows(table.get_rows(), row_idx);
9  }
```

Фрагмент 5: Методът *select_ rows* на клас ***Database***.

Методът *load_ table()* връща копие на таблица, приемайки нейното име като аргумент. Методът *load_ table* сам обръща името към път във файловата система, от който да бъде прочетена таблицата. След валидация дали таблицата е заредена успешно, векторът *row_ idx* приема ин-

дексите на всички редове, които съдържат запис със стойност, същата като тази на *val*. Накрая редовете от таблицата и индексите се дават на клас ***Presenter***, който отговаря за представянето на таблици в екрана на конзолата.

На Фрагмент 6 може да се види методът *find_rows_by_value*.

```
1  std::vector<int> Table::find_rows_by_value(int column, Record* val){
2
3
4      std::vector<int> found_rows;
5
6      if(column >= col_names.size()){
7
8          Message::WrongNumberOfColumns(column, col_names.size());
9          return found_rows;
10     }
11
12     //check type at current column
13     Type type = col_types[column];
14     if(val->get_type() != type){
15
16         return found_rows;
17     }
18
19     //search all rows for the value
20
21     for(int i=0; i<rows.size(); i++){
22
23         Record* curr = rows[i].get_records()[column];
24
25         if(*curr == *val){
26
27             found_rows.push_back(i);
28         }
29     }
30
31     return found_rows;
```

```
32
33 }
```

Фрагмент 6: Методът *find_rows_by_value*

Най-ключовото за него може би е сравнението между записи. Важен момент от имплементацията на проекта е това сравнение да работи по еднакъв начин, независимо от типа. Това е предизвикателно, защото то трябва да се използва върху обекти от тип ***Record****, обаче класът ***Record*** няма стойност, а само неговите наследници, и то тази стойност е от различен примитивен тип. Във Фрагмент 7 се вижда как изглежда имплементацията на операторът за сравнение в класа ***RecordDouble***. В останалите е аналогична.

```
1 bool RecordDouble::operator == (const Record& other) const{
2
3     if(const RecordDouble* r_double = dynamic_cast<const
      RecordDouble*>(&other)){
4
5         return abs(r_double->get_value() - value) <= COMP_EPS || (
      r_double->is_empty() && empty);
6     }
7
8     return false;
9 }
```

Фрагмент 7: Класът ***RecordDouble*** имплементира виртуалния оператор за сравнение.

За да има това сравнение логика, наследниците трябва да са от еднакъв тип, затова се прави опит за динамично кастване. Ако той е успешен, се извършва сравнение на стойностите на записите, които са с тип *Double*. Празни записи са винаги еднакви, ако са от един и същ тип.

2.2.4 Запазване на таблица

Конвертирането от таблица, представена като обект на клас ***Table*** във файл с формат, описан в Секция 2.1.1 също става лесно. Може да се види във Фрагмент 8.

```
1
2  void Table::save_table(std::string filename){
3
4      std::ofstream table(filename);
5
6      if(!table.is_open()){
7
8          Message::CannotWriteFile(filename);
9          return;
10     }
11
12     //write table name
13     table << name << std::endl;
14
15     //write column names
16     std::string names = col_names_to_str();
17     table << names << std::endl;
18
19     //write types
20     std::string types = types_to_str();
21     table << types << std::endl;
22     //write rows
23     for(int i=0; i<rows.size(); i++){
24
25         std::string row_string = rows[i].to_string();
26         table << row_string;
27
28         if(i < rows.size()-1){
29
30             table << std::endl;
31         }
32     }
```

```
33 }
```

Фрагмент 8: Метод на клас ***Table*** за запазване на таблица във файл.

Всеки компонент се конвертира в тип *String*, след което се записва във файла. Методите за конвертиране поддържат всички възможни типове записи, както и празни такива.

2.3 Съобщения за грешки

С цел успешното ползване на програмата, навсякъде е имплементирана нужната валидация за грешки. Те се обработват и резултират в подходящо съобщениие към потребителя. В тази секция са показани две примерни съобщения за грешки при неуспешно валидиране на таблица при прочитане.

2.3.1 Невалиден запис

С демонстративна цел е създадена невалидна таблица (Таблица 2).

NameInt	NameDouble
0	1
123	321
455	34.14.57

Таблица 2: Примерна таблица, съдържаща записи от тип *Int* и *Double*, с един неправилен запис.

При опит за прочитане, функцията за четене на таблица *read_table* приключва и в конзолата се изписва следното съобщение:

```
$ Invalid record at column 1
```

```
$ Corrupted row with index 1
```

2.3.2 Неправилен брой колони

Друг пример за невалидна таблица е случай с невалиден брой колони. Такава е Таблица 3.

NameInt	NameDouble	n/a
0	1	n/a
123	12.1	n/a
455	34.14	12

Таблица 3: Примерна таблица, съдържаща записи от тип *Int* и *Double* с неправилен брой колони в един от редовете.

При опит за прочитане, функцията за четене на таблица *read_table* приключва и в конзолата се изписва следното съобщение:

```
$ Row has 3 columns, but expected are 2
$ Corrupted row with index 1
```

3 Заключение

В настоящият проект успешно беше имплементирана база от данни, способна да работи успешно с по-малки таблици. Първоначалната архитектура на проекта позволи сравнително лесно да бъдат имплементирани сложни операции за манипулация на данни и дават потенциал проектът

да бъде развиван още. Ключовите решения за реализацията на задачата бяха изготвянето на еднозначен файлов формат, акцентиране върху съставните единици на проекта и внимателно планирана архитектура.

4 Използвани технологии

Целият проект е реализиран на езика C++. Ключови използвани библиотеки са:

- iostream
- vector
- string
- fstream
- algorithm
- unordered_set
- unordered_map
- sstream

Проектът е тестван на Arch Linux 5.5.10. Компилиран е с g++ за C++11.

5 Литература

Много справки, свързани с реализацията на проекта и документацията му са правени в:

- `https://stackoverflow.com/`

- http://www.cplusplus.com/doc/
- https://en.cppreference.com/w/
- https://www.overleaf.com/learn
- https://tex.stackexchange.com/