Описание классов запросов

Попель Мария

Виды запросов и синтаксис

Возможные названия полей

В расписании присутствуют поля с названиями `group`, `month`, `day`, `lessonNum`, `subject`, `teacherName`, `room`. Соответственно номер группы, месяц, день, номер аудитории и номер пары записываются целыми числами или диапазоном, например, `210-215`. Название предмета или фамилия преподавателя записываются одним словом – для них возможен поиск по маскам, но невозможно применение логических операций или диапазонов. Доступные логические операции: `=`, `>`, `<`, `!=`, `>=`, `<=`, `~`.

SELECT

Запрос имеет вид: `SELECT fldName_1=val_1 ... fldName_n=val_n end`

Вместо знака `=` можно поставить `>=`, `<=`, `>`, `<`, `!=`, `~` - последний знак соответствует маске. Для маски верны следующие типы записи: `Абв*` - вместо `*` можно вставить *ОДИН любой знак*; `Абв^` - вместо `^` можно вставить любое количество (кроме 0) любых символов.

Каждая тройка запроса вводится без пробелов: название поля, логический оператор, значение. Сами тройки перечисляются через пробел без прочих знаков. В конце каждого запроса должна стоять запись `end`.

RESELECT

Структура запроса имеет ту же логику, что и запрос типа `SELECT`. Но теперь отбор записей происходит из ранее отобранных с помощью `SELECT` записей. Если ранее не производилось отбора записей с помощью запроса `SELECT`, сервер выдаст ошибку `No previous selection to reselect from`.

PRINT

Запрос имеет вид: `PRINT fldName_1 ... fldName_n sort fldName ASC end`. Вместо `ASC` – сортировка по возрастанию, можно написать `DESC` – сорти-

ровка по убыванию. Запрос будет выводить перечисленные поля с соответствующими им значениями из ранее определенной с помощью `SELECT` или `RESELECT`. Если между `PRINT` и `sort` не указывать названия полей, то будет выведена вся выбранная до того таблица. Если предыдущим запросом не был `SELECT` или `RESELECT`, то программа выдаст ошибку `No previous selection to print`.

INSERT

Запрос имеет вид: `INSERT fldName_1=val_1 ... fldName_n=val_n end`. В качестве полей должны быть перечислены возможные названия полей. Они могут быть указаны не все – тогда значениям неуказанных полей будет присваиваться 0 или в зависимости от типа поля. Здесь логическим символом может быть ТОЛЬКО знак `=`.

Этот запрос добавляет новую запись в базу данных, если она не пересекается по дате, номеру пары и аудитории или именем преподавателя с другой уже существующей записью. В случае пересечения – исходная база данных не будет изменена.

UPDATE

Запрос имеет вид: `UPDATE fldName_11=val_11 ... fldName_n1=val_n1 WHERE fldName_12=val_12 ... fldName_n2=val_n2 end`. Здесь до ключевого слова `WHERE` перечисляются поля, которые нужно исправить, с новыми значениями. В этой части запроса из логических символов возможен только знак `=`. После ключевого слова `WHERE` записываются условия, по которым отбираются записи для изменения. Здесь уже работает логика отбора как в запросе `SELECT`, т.е. возможны любые из имеющихся логических операций.

DELETE

Запрос имеет вид: `DELETE fldName_1=val_1 ... fldName_n=val_n end`. Здесь в запросе после ключевого слова перечисляются условия, по которым идет отбор полей для удаления. Отбор происходит аналогично случаю запроса `SELECT`. Если не было отобрано ни одного запроса, сервер выдаст сообщение: `Nothing was deleted.`

Общие сообщения

В общем случае, если выборка возвращает пустое множество значений, выдается сообщение: `Нет подходящих записей`.

После выполнения запросов типа SELECT, RESELECT, PRINT - в файл scheduleout.txt выводится изначальный запрос. После выполнения запросов

типа INSERT, UPDATE, DELETE - в этот же файл выводится измененненная база данных (расписание).

Запуск сервера и клиента

Вводим `make`, затем `./server`. Потом в новом окне консоли запускаем клиент `./client localhost 1234`. Далее следовать появившимся инструкциям.

Комментарии к .h-file

Класс ошибок

Переменные: `code_` - код ошибки, `mess_` - сообщение об ошибке.

```
class Exception {
private:
    int code_;
    std::string mess_;
public:
    Exception(int c, std::string m) : code_(c), mess_(m) {}
    int getCode() const { return code_; }
    std::string getMessage() const { return mess_; }
};
```

Класс записей

Переменные: `day_` - день, `month_` - месяц, `lesson_` - номер пары, `room_` - номер аудитории, `subject_name_` - название предмета, `teacher_` - фамилия преподавателя, `group_` - номер группы.

Содержит функции для вывода каждой ячейки и функцию для вывода записи в строку (`toString`).

```
class Entry {
private:
    int day_;
    int month_;
    int lesson_;
    int room_;
    char subject_name_[128];
    std::string teacher_;
    int group_;

public:
```

```
    Entry(int day, int month, int lesson, int room, const char*
        red↪ subjName, std::string teacher, int group)
        : day_(day), month_(month), lesson_(lesson), room_(room),
            red↪ teacher_(teacher), group_(group) {
        strncpy(subject_name_, subjName, sizeof(subject_name_));
    }

    bool operator==(const Entry& other) const {
        return day_ == other.day_ &&
               month_ == other.month_ &&
               lesson_ == other.lesson_ &&
               room_ == other.room_ &&
               strcmp(subject_name_, other.subject_name_) == 0 &&
               teacher_ == other.teacher_ &&
               group_ == other.group_;
    }

    int getDay() const { return day_; }
    int getMonth() const { return month_; }
    int getLesson() const { return lesson_; }
    int getRoom() const { return room_; }
    const char* getSubjectName() const { return subject_name_; }
    std::string getTeacher() const { return teacher_; }
    int getGroup() const { return group_; }

    void setDay(int day) { day_ = day; }
    void setMonth(int month) { month_ = month; }
    void setLesson(int lesson) { lesson_ = lesson; }
    void setRoom(int room) { room_ = room; }
    void setSubjectName(const char* subjName) {
        strncpy(subject_name_, subjName, sizeof(subject_name_));
    }
    void setTeacher(std::string teacher) { teacher_ = teacher; }
    void setGroup(int group) { group_ = group; }

    std::string toString() const {
        std::stringstream ss;
        ss << "Date:␣" << day_ << "." << month_
           << ",␣Lesson:␣" << lesson_
           << ",␣Room:␣" << room_
           << ",␣Subject:␣" << subject_name_
           << ",␣Teacher:␣" << teacher_
           << ",␣Group:␣" << group_;
        return ss.str();
    }
};
```

Класс информации о клиенте и сессии

Переменные: `socket` – номер сокета, `previousSelection` – множество записей предыдущей сессии клиента.

```
struct ClientInfo {
    int socket;
    std::vector<Entry*> previousSelection;

    ClientInfo() : socket(0), previousSelection() {}
};
```

Структура для вывода результата

Переменные: `entry` – запись, `message` – информация о результате, `error` - ошибки.

```
struct result {
    std::vector<Entry> entry;
    std::string message;
    Exception error;

    result() : entry(), message(), error(0, "") {}

    void addEntry(const Entry& ent) { entry.push_back(ent); }
    void addMessage(const std::string& mes) { message = mes; }
    void addError(const Exception& err) { error = err; }
};
```

Новые структуры для обозначения ключевых слов, названия полей, логических операций и т.д.

```
typedef enum { SELECT, RESELECT, ASSIGN, INSERT, UPDATE, DELETE,
    red↪ PRINT } ComType;
typedef enum { DAY, MONTH, LESSON_NUM, ROOM, SUBJNAME,
    red↪ TEACHERNAME, GROUP, NONE_FIELD } Field;
typedef enum { ASC, DESC } Order;
typedef enum { LT, GT, EQ, LT_EQ, GT_EQ, NEQ, LIKE } BinOp;
```

Класс условий

Переменные: `field_` - название поля, `operation_` - логическая операция, `value_` - значение в поле.

```
class Cond {
private:
    Field field_;
    BinOp operation_;
    Value value_;
public:
    Cond(Field fld, BinOp optn, Value val) : field_(fld),
        red↪ operation_(optn), value_(val) {}
    Field getField() const { return field_; }
    BinOp getOperation() const { return operation_; }
    Value getVal() const { return value_; }
};
```

Класс запроса

Переменные: `command_` - тип запроса, `query_` - запрос, `parsers` – вектор парсинговых функций.

```
class Query {
private:
    ComType command_;
    std::string query_;
    using parser_fn = Query* (*)(const std::string& query);
    static std::list<parser_fn> parsers;
public:
    Query(const std::string& query) : command_(SELECT), query_(
        red↪ query) {}

    virtual ~Query() = default;
    virtual void parse() = 0;
    ComType getCommand() const { return command_; }
    static Query* do_parse(const std::string& query);
    static void register_parser(parser_fn p);
    static void clear_parsers();
protected:
    void setCommand(ComType command) { command_ = command; }
    virtual const std::string& getQueryString() const { return
        red↪ query_; }
};
```

Класс запроса типа обработки запроса Select

Переменные: `condition_` - вектор условий в тройках запроса, `fields_` - вектор полей в тройках запроса.

```
class SelectingQuery : virtual public Query {
public:
    SelectingQuery(const std::string& query) : Query(query),
        red↪ condition_(), fields_() {}
    void parse() override;
    void parseCond();
    const std::vector<Cond>& getConditions() const { return
        red↪ condition_; }
    const std::vector<Field>& getFields() const { return fields_;
        red↪ }
    bool parseTriple(const std::string& triple);
private:
    std::vector<Cond> condition_;
    std::vector<Field> fields_;
};
```

Класс запроса типа обработки запроса Assign

Переменные: `values_` - вектор пар значений и соответствующих им полей.

```
class AssigningQuery : virtual public Query {
public:
    AssigningQuery(const std::string& query) : Query(query),
        red↪ values_() {}
    void parse() override;
    const std::vector<std::pair<Field, Value>>& getValues() const
        red↪ { return values_; }
protected:
    bool parseAssigningTriple(const std::string& triple);
private:
    std::vector<std::pair<Field, Value>> values_;
};
```

Класс запроса типа обработки запроса Update

```
class UpdateQuery : public SelectingQuery, public AssigningQuery
    red↪ {
public:
```

```
    UpdateQuery(const std::string& query) : Query(query),
        red↪ SelectingQuery(query), AssigningQuery(query) {}
    void parse() override;
};
```

Класс запроса типа обработки запроса Insert

```
class InsertQuery : public AssigningQuery {
public:
    InsertQuery(const std::string& query) : Query(query),
        red↪ AssigningQuery(query) {}
    void parse() override;
};
```

Класс запроса типа обработки запроса Print

Переменные: `fields_` - вектор полей в тройках запроса, `sort_fields_` - вектор пар полей и порядка сортировки.

```
class PrintQuery : public Query {
private:
    std::vector<Field> fields_;
    std::vector<std::pair<Field, Order>> sort_fields_;
public:
    PrintQuery(const std::string& query) : Query(query), fields_()
        red↪ , sort_fields_() {}
    void parse() override;
    const std::vector<Field>& getFields() const { return fields_;
        red↪ }
    const std::vector<std::pair<Field, Order>>& getSortFields()
        red↪ const { return sort_fields_; }
};
```

Класс запроса типа обработки запроса Delete

```
class DeleteQuery : public SelectingQuery {
public:
    DeleteQuery(const std::string& query) : Query(query),
        red↪ SelectingQuery(query) {}
    void parse() override;
};
```

Вспомогательные функции для парсинга

```
Field parseField(const std::string& fieldStr);
BinOp parseBinOp(const std::string& opStr);
Value parseValue(const std::string& valueStr);
```

Класс расписания

Переменные: `schedule_` - расписание, индексы по типам полей.

Функции: `checkTimeCross` – проверка пересечения записей, `saveToFile` – сохранение в файл, `deleteEntries` – удаление значений по набору условий, `buildIndexes()` – построение индексов, `loadFromFile` – загрузка из файла базы данных.

```
class Schedule {
private:
    std::vector<Entry*> schedule_;
    std::unordered_map<std::string, std::vector<int>>
        red↪ teacher_index_;
    std::unordered_map<int, std::vector<int>> group_index_;
    std::unordered_map<int, std::vector<int>> room_index_;
    std::unordered_map<int, std::vector<int>> lesson_index_;
    std::unordered_map<std::string, std::vector<int>>
        red↪ subject_index_;

public:
    Schedule();
    Schedule(FILE* fin);
    ~Schedule();
    bool checkTimeCross(int day, int month, int lesson, std::
        red↪ string teacher, int room, int group);
    void addEntry(Entry* entry);
    void deleteEntry(Entry* entry);
    void updateEntry(int day, int month, int lesson, int room,
        red↪ Entry* newEntry);
    std::vector<Entry*> select(const std::vector<Cond>& crit);
    std::vector<Entry*> reselect(const std::vector<Entry*>&
        red↪ selected, const std::vector<Cond>& crit);
    void print(const std::vector<Entry*>& entries);
    void saveToFile(const std::string& filename);
    std::vector<Entry*> deleteEntries(const std::vector<Cond>&
        red↪ crit);

    void buildIndexes();
```

```
    void loadFromFile(const std::string& filename);
};
```

Класс базы данных

Переменные: `schedule_` - расписание, `clientSessions` – пары информации о клиенте, `startQuery` – основная функция, запускающая обработку запроса.

```
class DataBase {
private:
    Schedule* schedule_;
    std::unordered_map<int, ClientInfo> clientSessions;
public:
    DataBase(FILE* fin);
    ~DataBase();
    DataBase(const DataBase&) = delete;
    DataBase& operator=(const DataBase&) = delete;
    result startQuery(int clientSocket, std::string& query);
    Schedule* getSchedule() const { return schedule_; }
};
```

Функции сравнения

Функции сравнения целых чисел, строк, смеси целых чисел и строк.

```
bool compareInts(int val1, int val2, BinOp operation);
bool compareStrings(const std::string& str1, const std::string&
    red↪ str2, BinOp operation);
bool compareIntStr(const std::string& str1, const std::string&
    red↪ str2, BinOp operation);
```