Catalog模块:

每张表以及表中所有字段的模式信息都由该模块管理，保存在一个单独的文件中。

使用到的结构：

/\* 表信息文件的文件头 \*/

struct table\_head\_t

{

int tableNum; // 现存放的表的数量

};

/\* 字段结构 \*/

struct attr\_t {

string name; // 字段名

bool isPrimary; // 是否主键

int length; // 字段占字节数

attrtype\_t type; // 类型（CHAR, INT）

};

/\* 表的结构 \*/

struct table\_t

{

string name; // 表名

int attrNum; // 字段数

int recordLength; // 一条记录的字节数

attr\_t attributes[MAX\_ATTR\_NUM]; //字段

};

类：

class Catalog

公有函数：

// 返回要求的字段信息

attr\_t findAttr(string tableName, string attrName);

// 返回要求的表的信息

table\_t findTable(string tableName);

// 返回主键的字段信息（然后去找索引）

attr\_t getPrimaryAttr(string tableName);

// 返回表是否存在

bool tableExist(string tableName);

// 返回表是否存在该字段

bool attrExist(string tableName, string attrName);

// 建表

int createTable(table\_t & table);

// 删除表

int deleteTable(table\_t & table);

私有函数：

// 初始化表文件（写入表信息头）

void initTable();

// 文件读写的包装函数

inline void writeTableHead(fstream & fout, table\_head\_t & tableHead);

inline void readTableHead(fstream & fin, table\_head\_t & tableHead);

inline void writeTable(fstream & fout, table\_t & table);

inline void readTable(fstream & fin, table\_t & table);

inline void writeAttr(fstream & fout, attr\_t & attr);

inline void readAttr(fstream & fin, attr\_t & attr);

Index 模块：

可以为一张表的所有记录在一个字段上建立一个索引文件，现在使用的是线性索引。

使用到的结构：

/\* 索引节点 \*/

struct index\_node\_t

{

string value; // 关键码

unsigned offset; // basep 的偏移

index\_node\_t \*nextNode;

// 强烈注意：该指针在写入文件的时候不会写入，在读出文件的时候不会读出其作用只是在 select 时，返回一个索引项的链表（因为有时查询值是一个范围）

};

/\* 索引头 \*/

struct index\_head\_t

{

attr\_t attr; // 做索引的字段

int recNum; // 记录数目

};

类：

Class Index

公有函数：

// 查找关键码为 valueFrom 到 valueTo 的索引项

int selectIndex(string tableName, string indexName, string valueFrom, string valueTo, index\_node\_t \*res);

// 建立索引

int createIndex(string tableName, string indexName, attr\_t & attr);

// 插入关键码为 value 的新索引项

int insertIndex(string tableName, string indexName, index\_node\_t & node);

// 删除关键码为 value 的索引项

int deleteIndex(string tableName, string indexName, string value);

// 更新关键码为 value 的索引项，新关键码为 newValue

int updateIndex(string tableName, string indexName, string value, string newValue);

// 合并多个索引项链表，结果放在 res 中

int mergeIndex(index\_node\_t \*\*list, int listNum, index\_node\_t \*res);

Index();

~Index();

私有函数：

// 二分查找，返回关键码为 value 的节点的起始地址，给 update 使用

inline int biSearch(fstream & fin, int from, int to, string value, attrtype\_t type);

// 二分查找 范围查询之 from 版，返回第一个关键码大于等于 value 的节点的起始地址，给 select 使用

inline int biSearchFrom(fstream & fin, int from, int to, string value, attrtype\_t type);

// 二分查找 范围查询之 to 版，返回最后一个关键码小于等于 value 的节点的起始地址，给 select 使用

inline int biSearchTo(fstream & fin, int from, int to, string value, attrtype\_t type);

// 比较关键码大小

inline bool lessThan(string value\_1, string value\_2, attrtype\_t type);

// 文件读写的包装函数

inline void readAttr(fstream & fin, attr\_t & attr);

inline void writeAttr(fstream & fout, attr\_t & attr);

inline void readHead(fstream & fin, index\_head\_t & head);

inline void writeHead(fstream & fout, index\_head\_t & head);

inline void readNode(fstream & fin, index\_node\_t & node);

inline void writeNode(fstream & fout, index\_node\_t & node);

Interpreter 模块

使用到的结构体：

/\* 条件树的结构 \*/

struct condition\_tree\_t

{

condition\_tree\_t \* left; // 左子树

condition\_tree\_t \* right; // 右子树

logic\_t logic; // 不同条件之间的逻辑关系

oper\_t opName; // ${leftOperand} ${opName} ${rightOperand}

string leftOperand;

string rightOperand;

bool end;

};

/\* 解析器解析命令后的结果 \*/

struct info\_t {

cmd\_t command;

string tableName; // 命令相关的表名

string indexName; // 索引相关操作的索引名

table\_t t; // 命令相关的表信息

std::vector<string> selectedItems; // select 语句中被选择的字段

std::vector<string> selectedTable; // select 语句中被选择的Table

std::map<string, string> insertItems; // map of attribute name & value while inserting

std::map<string, string> updateItems; // map of attribute name & value while updating

condition\_tree\_t \* tree; // 条件树，用于 where 语句

};

使用到的枚举：

/\* 运算符类型 \*/

typedef enum {

EQ, // ==

NE, // <>

GT, // >

LT, // <

GTE, // >=

LTE // <=

} oper\_t;

/\* 逻辑运算 \*/

typedef enum {

AND,

OR

} logic\_t;

/\* 操作类型 \*/

typedef enum {

NONE, // 无操作

CREATE\_TABLE, // 建表

DROP\_TABLE, // 删表

CREATE\_INDEX, // 建索引 - 不急着做

DELETE\_INDEX, // 删索引 - 不急着做

UPDATE, // 更新记录

SELECT, // 查记录

INSERT, // 插入记录

DELETE, // 删记录

QUIT, // 退出程序

HELP // 打印帮助文档 - 不急着做

} cmd\_t;

类：

class Interpreter

公有函数：

// get command from STDIN

bool inputCommand();

// get condition tree

void getCondition(std::vector<string> command, condition\_tree\_t \* condition);

// get the result after interpretion

info\_t getInfo();

// print information for debug

void debug();

Interpreter();

私有函数：

/\*--------------parse different types of commands------------------\*/

bool parseCommand(); // parse the input

bool parseInsert();

bool parseSelect();

bool parseCreate();

bool parseUpdate();

bool parseDelete();

bool parseQuit();

bool parseHelp();

bool parseDrop();

/\*-----------------------------------------------------\*/

// clear all info before rewriting

void clearInfo();

// show details of condition tree

void showConditionTree(condition\_tree\_t \* root);

// destruct the condition tree

void clearTree(condition\_tree\_t \* root);

// make condition tree with given input

condition\_tree\_t \* makeTree(int index);

私有变量：

string input; // command input by user

std::vector<string> command; // parsed command

info\_t info; // infomation interpreted by Interpreter

其他功能：

实现简单查错功能：

1. 括号是否匹配
2. 键入命令类型是否正确 (creat table t ..)
3. 键入变量类型是否支持 (id double)
4. 各命令是否缺少必要参数 （如 select from t; ）

Record模块

**概述：**

Record模块用来直接对文件中的数据进行增、删、改、选的操作。期中，所有的操作均由二进制文件读写函数write或read完成。

类的设计及其功能：

class Record

{

public:

Record(){};

~Record(){};

//在指定文件尾插入一条记录

int Insert(info\_t & insert\_info);

//删除指定的一条记录

void Delete(info\_t & delete\_info, index\_node\_t & index);

//更新一条记录的字段值

void Update(info\_t & update\_info, index\_node\_t & index);

//根据输入参数中的条件树找出所有符合条件的记录，并且返回符合条件记录链表的头指针

record\_t \*Select(info\_t & select\_info);

//输出一条记录中所有的字段值

void Print(record\_t \*record); //输出一个表的所有字段名

void PrintHead(table\_t & table);

private:

/\* 用来某一条记录是否符合条件树的要求，返回值为1则表示符合要求，返回0表示不符合要求，返回-1表示

某条记录不含要求的字段值，不需判断。offset为某条记录首地址偏移 \*/

int Judge(condition\_tree\_t \* tempCondition, int offset, table\_t table, ifstream &input);

/\* 根据表信息和字段名返回字段值类型等信息，0表示整形，1表示字符串，-1表示不存在此字段，Offset表示

所查找字段相对与当前记录首地址的偏移，attriLength表示这个字段值的字节数 \*/

int getInfo(table\_t table, string infoName, int &Offset, int &attriLength);

};

**类的实现：**

1.

int Insert(info\_t & insert\_info);

根据insert\_info的内容能确定表的名字、要插入的字段值等所需数据。首先根据表明确定文件名，用二进制格式打开文件，定位到文件末尾。然后对于这个表中的每个字段，逐个在insert\_info中寻找是否有要插入的对应字段，如果有，则按照规定字节数以二进制格式输入；如果没有相应字段值要插入，则以字符串形式输入“oop”。

2.

void Delete(info\_t & delete\_info, index\_node\_t & index);

根据输入的delete\_info和index可以确定要打开的文件，并且能直接定位到需要删除的记录的首地址偏移（相对于文件头）。之后，逐个对字段输入字符串“oop”。

3.

void Update(info\_t & update\_info, index\_node\_t & index);

Update的实现和Insert基本一致。根据update\_info和index定位到相应的文件和地址偏移，进行和Insert一样的操作。

4.

record\_t \*Select(info\_t & select\_info);

Select函数的实现需要充分利用select\_info中的条件树。基本思路如下：根据select\_info定位到相应文件头，然后逐条记录调用Judge函数，判断是否满足条件树的要求。如果满足要求，就将这一条记录包装成一个record\_t节点，最终形成链表，将record\_t的链表头指针返回。

5.

void Print(record\_t \*record);

Print输出一条记录中所有的字段值。根据输入的record链表，逐个遍历链表，易于实现。

6.

void PrintHead(table\_t & table);

直接根据table的内容输出一个表中所有字段名，易于实现。

7.

int Judge(condition\_tree\_t \* tempCondition, int offset, table\_t table, ifstream &input);

Judge函数根据输入的条件树tempCondition，文件内一条记录的首地址偏移offset，table信息和input流，判断一个表中的某一条记录是否符合条件树的要求。这是一个递归函数，每次返回的是当前两个子节点返回值的“与”或“或”，由条件数中的参数决定。

8.

int getInfo(table\_t table, string infoName, int &Offset, int &attriLength);

getInfo函数根据某个表中的字段名返回这个字段对应字段值的类型（整形或字符串）、字段值的字节数、字段相对于记录首地址的偏移等基本信息，需要被Judge函数调用。

API 模块，调度各个模块的调度器。

class API

{

public:

API();

~API();

// 建表

int createTable();

// 删表

int dropTable();

// 建立索引

int createIndex();

// 查询

int select();

// 插入

int insert();

// 更新

int update();

// 删除数据

int deleteRecord();

// 从标准输入读

int getInput();

// 退出

int exit();

// 帮助

void help();

// 命令类型

cmd\_t commandType();

};

main routine:

int main()

{

API api;

printf("+-------------------------+\n");

printf("| Welcome to use MiniSQL. |\n");

printf("+-------------------------+\n");

while(1)

{

api.getInput();

switch (api.commandType()) {

case NONE:

break;

case CREATE\_TABLE:

api.createTable();

break;

case DROP\_TABLE:

api.dropTable();

break;

case UPDATE:

api.update();

break;

case SELECT:

api.select();

break;

case INSERT:

api.insert();

break;

case DELETE:

api.deleteRecord();

break;

case QUIT:

api.exit();

break;

case HELP:

api.help();

break;

default:

break;

}

}

return 0;

}