编译原理实验 Project-SQL 查询

21307174 刘俊杰

May 2024

1 题目描述

建立一个简单的教务管理数据库，用于存储学生的考试成绩并支持一些基本的查询操作。

1.1 数据格式

该数据库包含以下五张数据表：

Student(sid, dept, age)

学生信息表：sid 为主键，表示学生 ID；dept 表示学生所在院系名称；age 表示学生的年龄。

Course(cid, name)

课程信息表：cid 为主键，表示课程 ID；name 表示课程名称。

Teacher(tid, dept, age)

教师信息表：tid 为主键，表示教师 ID；dept 表示教师所在院系名称；age 表示教师的年龄。

Grade(sid, cid, score)

成绩信息表：sid 和 cid 分别米自 Student 表和 Course 表的主键（即每条记录中的 sid 和 cid 一定存在于相应的表中，下同），二者一起作为该表的主键；score 表示该学生这门课程的成绩。

Teach(cid, tid) 授课信息表:cid 和 tid 分别来自 Course 表和 Teacher 表的主键，二者一起作为为该表的主键；需要注意的是，一门课程可以有多个老师授课。

注：主键（Primary Key）可以唯一标识表中的每条记录，换言之在一张表中所有记录的主键互不相同。

在上述表中，age 和 score 是 [0,100] 的整数，其余都是长度小于等于 10 的非空字符串。其中三个主键 sid _cid_ 和 tid 由数字 0...9 组成，而 name 和 dept 则由大小写字母组成。

1.2 查询语句

（具体查询语句说明见 SQL 查询作业 PDF）

1.3 结果输出

对每条查询输出一张表，列由 COLUMNS 指定，每行为一条满足筛选条件 (EXPR) 的记录（不同的列间用一个空格分隔)。这里只需要输出所有满足条件的记录，不需要打印表头；查询结果为空则不输出；字符串类型无需输出双引号。若想输出所有的列，可以用 * 代替 COLUMNS；若 TABLES 中只有一张表，此时应输出该表的全部列；如果有第二张表，应输出第一张表的全部列与二张表的全部列的笛卡尔积：表内部的列按照上文定义时给出的顺序排序。

如果 TABLES 仅含一张表，则按该表的顺序依次输出符合条件的记录。若 TABLES 包含两张表（形如 A,B ），则先考虑 A 的顺序，再考虑 B 的顺序 (参考前文中两张表做笛卡尔积的例子)。

1.4 【输入格式】

从标准输入读入数据。

首先依次输入五张表 Student、Course、Teacher、Grade 和 Teach 的数据。

对于第 i 张表，第一行输入一个正整数 N_i，表示该表中记录的个数（亦即行数)。接下来 N_i 行，每行输入一条记录，其中不同列之间用一个空格分隔且字符出字段不会用双引号括起。

然后输入一个正整数 M，表示需要处理的查询语句个数。最后 M 行每行输入一条查询语句，保证每条查询语句均符合上述所有要求。

1.5 【输出格式】

输出到标准输出。

按要求输出每条查询语句的结果。

1.6 【样例 1 输入】

```
3
2019001 law 19
2019002 info 20
2019003 info 19
2
20190001 math
20190002 English
2
2019101 info 49
2019102 info 38
4
SELECT FROM Student
SELECT Student . dept FROM Student
SELECT Student.sid,Grade.sid,dept, score FROM Student,Grade
SELECT FROM Student, Grade WHERE Grade .sid=Student. sid
```

1.7 【样例 1 输出】

```
2019001 law 19
2019002 info 20
2019003 info 19
law
info
info
2019001 2019002 law 100
2019001 2019003 law 0
2019002 2019002 info 100
2019002 2019003 info 0
2019003 2019002 info 100
2019003 2019003 info 0
2019002 info 20 2019002 20190001 100
2019003 info 19 2019003 20190002 0
```

1.8 【样例 1 解释】

第 1-3 行、4-6 行、7-12 行和 13-14 行依次对应四条查询的🄴果。

每条查询最多涉及两张表，每个 expr 中最多一个 COLUMN = COLUMN 类型的 COND，且 $Ni \leqslant 10000$、$M \leqslant 10$ 。此外，所有测试数据保证每条查询的结果均不超过 10000 行。

基础部分

2 初始版本 (pass 4 test cases)

2.1 对表中记录的存储

```
// TABLE根据表名存储不同表的记录
unordered_map<string, vector<vector<string>>> TABLE;

// 列与记录索引的映射
unordered_map<string, int> col_to_index = {
    {"sid", 0}, {"cid", 1}, {"tid", 2}, {"score", 3},
    {"name", 4}, {"dept", 5}, {"age", 6}
};

// 每张表的某个列是否存在
 std :: unordered_map<std::string, std :: vector<int>> col_exist = {
        {"Student", {1, 0, 0, 0, 0, 1, 1}},
        {"Course", {0, 1, 0, 0, 1, 0, 0}},
        {"Teacher", {0, 0, 1, 0, 0, 1, 1}},
        {"Grade", {1, 1, 0, 1, 0, 0, 0}},
        {"Teach", {0, 1, 1, 0, 0, 0, 0}}
    };

// 每张表存在列的索引
std :: unordered_map<std::string, std :: vector<int>> col_id = {
        {"Student", {0, 5, 6}},
        {"Course", {1, 4}},
```

```
        {"Teacher", {2, 5, 6}},
        {"Grade", {0, 1, 3}},
        {"Teach", {1, 2}}
};
```

首先为了实现对输入的记录数据的存储，使用 unordered_map 哈希表 TABLE 来存储，以表名作为键、插入的记录为值，插入的记录用 vector<vector<string 存储。

注意的是这里为了实现方便,每个表的记录都有 6 列,即 sid，cid,tid,score,name,dept,age，，如果该表中某列不存在则用”” 替代，后续如果优化可以对这部分进行优化，减少空间开销

同时用另外一个哈希表 col_to_index 记录该列名与记录索引（第几列）的映射。

2.2 对输入记录的处理

针对不同表的输入记录，将记录插入对应的表中，实现代码如下：

```
// 处理输入记录，存储到TABLE对应的表中
void process_input() {
    for (int i = 0; i < 5; i++) {
        int n;
        cin >> n;
        while (n > 0) {
            n--;
            switch (i) {
                case 0: { // Student(sid,dept,age)
                    string sid, age;
                    string dept;

                    cin >> sid >> dept >> age;
                    vector<string> temp = {sid, "", "", "", "", dept, age};
                    TABLE["Student"].push_back(temp);
                    break;
                }
                case 1: { // Course(cid,name)
                    string cid;
```

```
20                      string name;
21                      cin >> cid >> name;
22
23                      vector<string> temp = {"", cid, "", "", name, "", ""};
24                      TABLE["Course"].push_back(temp);
25                      break;
26                  }
27                  case 2: { // Teacher(tid,dept,age)
28
29                      string dept, tid ,age;
30                      cin >> tid >> dept >> age;
31                      vector<string> temp = {"","",tid,"", "", dept, age};
32                      TABLE["Teacher"].push_back(temp);
33                      break;
34                  }
35                  case 3: { // Grade(sid,cid,score)
36                      string sid , cid , score ;
37                      cin >> sid >> cid >> score;
38                      vector<string> temp = {sid, cid , "", score , "", "", ""};
39                      TABLE["Grade"].push_back(temp);
40                      break;
41                  }
42                  default: { // Teach(cid,tid)
43                      string cid , tid ;
44                      cin >> cid >> tid;
45                      vector<string> temp = {"", cid, tid , "", "", "", ""};
46                      TABLE["Teach"].push_back(temp);
47                      break;
48                  }
49              }
50          }
51      }
52  }
```

2.3 对查询语句的预处理

2.3.1 对字符串的预处理函数

以下是三种不同功能的字符串处理函数，方便后续在对查询语句的与处理中用到：（最主要的是 splitClause 函数，为的是避免划分引号里的分隔符，如 WHERE = " AND " ）

```
1  // 移除字符串中的所有空格
2  string removeSpaces(const string& str) {
3      string result;
4      for (char ch : str) {
5          if (!isspace(static_cast<unsigned char>(ch))) { // 使用isspace检查是否为空格
6              result += ch;
7          }
8      }
9      return result;
10 }
11
12 // 分割字符串的函数，基于提供的分隔符(char)，同时去除无用的空格
13 vector<string> splitString(const string& str, char delimiter) {
14     vector<string> tokens;
15     string token;
16     istringstream tokenStream(str);
17
18     while (getline(tokenStream, token, delimiter)) {
19         tokens.push_back(removeSpaces(token));
20     }
21
22     return tokens;
23 }
24
25 // 分割字符串的函数，使用整个字符串作为分隔符(string)，同时去除无用的空格
26 vector<string> splitString_str(const string& str, const string& delimiters) {
27     vector<string> result;
28     size_t pos = 0;
29     size_t delimiterPos;
```

```
30      // 只要还有分隔符，就继续分割
31      while ((delimiterPos = str.find(delimiters, pos)) != string::npos) {
32          // 如果分隔符前面有文本，添加到结果中
33          if (delimiterPos != pos) {
34              result.push_back(removeSpaces(str.substr(pos, delimiterPos - pos)));
35          }
36          // 更新位置，跳过分隔符
37          pos = delimiterPos + delimiters.length();
38      }
39      // 添加剩余的文本
40      if (pos != str.length()) {
41          result.push_back(removeSpaces(str.substr(pos)));
42      }
43      return result;
44  }
45
46  // 按给定的分隔符划分(但不会划分引号里的分隔符)
47  vector<string> splitClause(const string& str, const string& delimiter) {
48      vector<string> result;
49      bool inQuotes = false;
50      size_t start = 0;
51      size_t delimiterLength = delimiter.length();
52
53      for (size_t i = 0; i < str.length(); ++i) {
54          if (str[i] == '\"') {
55              inQuotes = !inQuotes;
56          }
57
58          if (!inQuotes && i + delimiterLength <= str.length()
59          && str.substr(i, delimiterLength) == delimiter) {
60              string subStr = str.substr(start, i - start);
61              if (subStr.front() != '\"' || subStr.back() != '\"') {
62                  subStr = removeSpaces(subStr);
63              }
64              result.push_back(subStr);
```

```
65                start = i + delimiterLength;
66                i += delimiterLength - 1;
67            }
68        }
69
70        string lastSubStr = str.substr( start );
71        if (lastSubStr.front() != '\"' || lastSubStr.back() != '\"') {
72            lastSubStr = removeSpaces(lastSubStr);
73        }
74        if(lastSubStr.size()>0)//避免空串
75        result.push_back(lastSubStr);
76
77        return result;
78    }
```

2.3.2 对查询语句的预处理

实现代码如下：

```
1   int n; // 查询语句的个数
2       cin >> n;
3       cin.ignore(); // 处理输入n后的换行符
4
5       // 处理查询
6       while (n > 0) {
7           n--;
8           string s;
9           // 读取每个查询
10          while ( getline(cin, s)) {
11              // 对查询语句的不同部分进行分割
12              string attribute_str;
13              string table_str;
14              string where_str;
15              int a = s.find("SELECT");
16              int t = s.find("FROM");
17              int w = s.find("WHERE") == -1 ? s.size() + 1 : s.find("WHERE");
```

```
18              attribute_str = s.substr(a + 7, t - a - 8);
19              table_str = s.substr(t + 5, w - t - 5);
20              where_str = (w == s.size() + 1 ? " " : s.substr(w + 6));
21
22              vector<string> table = splitString(table_str, ',');// 查询的表集合
23              vector<string> attribute = splitString(attribute_str, ',');// 查询的列集合
24              vector<string> where = splitClause(where_str, "AND");// 查询的条件集合
25
26              // 处理查询并输出结果
27              process_queries(table, attribute, where);
28              break;
29          }
30      }
```

① 首先，对每一句查询语句针对不同的部分进行分割：

attribute_str 是 SELECT 到 FROM 之间的子字符串，即查询的列；
table_str 是 FROM 到 WHERE 之间的子字符串，即查询的表；
where_str 是 WHERE 之后的子字符串（若无 WHERE，则该 where_str 为""），即查询的条件。

② 使用前面的字符串处理函数，分别对步骤 ① 中的 attribute_str、table_str 和 where_str，用，来分割 attribute_str 和 table_str 不同列或表并去除空格，用 AND 来分割 where_str 来获得不同的条件并去除空格，得到：

attribute 是查询的列集合；
table 是查询的表集合；
where 是查询的条件集合。

③ 最后将查询语句的预处理结果传入到 process_queries 函数来处理查询并输出结果。

2.4 查询处理并输出结果

分针对 1 张表的查询和针对 2 张表的查询分别处理（后续如果有优化，可以优化该部分，使得可以对任意张表进行处理，不用分情况处理）：

```
// 处理查询信息
void process_queries(const vector<string>& table,
const vector<string>& attribute, const vector<string>& where) {
    if (table.size() == 1) { // 查询一张表
        process_queries_for_one_table(table, attribute ,where);
    }
    else { // 查询两张表
        process_queries_for_two_table(table, attribute ,where);
    }
}
```

2.4.1 对 1 张表的查询处理并输出结果

对一个条件的左右值简化：由于对查询一张表的条件的左右值进行处理，查询内容的列中可去除表名，即 TABLE.COLUMN 只用保留 COLUMN。同时为了处理与字符串的比较，如“1999”，要将其中的双引号去除。

实现代码如下：

```
// 对查询一张表的条件的左右值进行处理
// w : 条件字符串 str: 判别符
vector<string> process_condition_for_one_table(const vector<string>& col,string w, string str) {
    vector<string> t;
    t = splitClause(w, str);
    string x = t[0], y = t[1];
    // 针对一张表的查询 查询内容中可去除表名
    vector<string> xt = splitClause(x, ".");
    vector<string> yt = splitClause(y, ".");
    string xx = xt.size() == 1 ? xt[0] : xt[1];
    string yy = yt.size() == 1 ? yt[0] : yt[1];

    if(col_to_index.count(xx)){
        xx = col[col_to_index[xx]];
```

```
    }
    // 处理引号
    else if (xx[0] == '\"' && xx.back() == '\"') {
        xx = xx.substr(1, xx.size() - 2);
    }
    if(col_to_index.count(yy)){
        yy = col[col_to_index[yy]];
    }
    // 处理引号
    else if (yy[0] == '\"' && yy.back() == '\"') {
        yy = yy.substr(1, yy.size() - 2);
    }
    return {xx, yy};
}
```

处理对 1 张表的查询并输出查询结果：

① 首先遍历查询表中的每一个记录，用 flag 标志记录查询语句中的条件是否成立，成立 flag=1，否则 flag=0。

② 针对每一个记录判断是否满足查询的所有条件约束，先遍历判断所使用的判别符（如 =、!=、>= 等等），用判别符分割查询条件，得到左值和右值的字符串。

③ 调用 process_condition_for_one_table 对左右值进行简化。

④ 对简化后的结果根据判别符判断是否成立，不成立则修改 flag 并退出。

⑤ 若 flag=1 即所有条件都成立，则根据查询语句中的表和列，输出对应的查询结果；否则查询结果为空。

实现代码如下：

```
// 处理对1张表的查询

void process_queries_for_one_table(const vector<string>& table,
const vector<string>& attribute, const vector<string>& where) {
    for (auto& it : TABLE[table[0]]) {
            int flag = 1; // 条件是否成立标志

            // 判断where之后的条件是否成立
```

```
9                for (auto& w : where) {
10                   // 针对不同判别符处理
11                   if (w.find("=") != -1) {// =
12                       vector<string> t = process_condition_for_one_table(it,w, "=");
13                       string xx = t[0], yy = t[1];
14                       if (xx != yy) {
15                           flag = 0;
16                           break;
17                       }
18                   } else if (w.find(">") != -1) {// >
19                       vector<string> t = process_condition_for_one_table(it,w, ">");
20                       string xx = t[0], yy = t[1];
21                       if ( stoll (xx) <= stoll(yy)) {
22                           flag = 0;
23                           break;
24                       }
25                   } else if (w.find("<") != -1) { // <
26                       vector<string> t = process_condition_for_one_table(it,w, "<");
27                       string xx = t[0], yy = t[1];
28                       if ( stoll (xx) >= stoll(yy)) {
29                           flag = 0;
30                           break;
31                       }
32                   }
33               }
34
35               // 如果条件成立,输出查询结果
36               if (flag) {
37                   if ( attribute [0] == "*") { // 输出查询表的所有列
38                       vector<string>output;
39                       for (auto&i:col_id[table [0]]) {
40                               output.push_back(it[i ]);
41                       }
42                       output_strings(output);
43                       cout << endl;
```

```
44                  } else { // 输出查询的列
45                      vector<string>output;
46                      for (int i = 0; i < attribute.size(); i++) {
47                          // 由于是对一张表的查询,可以只关注.之后的列名
48                          string a = attribute[i];
49                          vector<string> t = splitClause(a, ".");
50                          string s;
51                          if (t.size() == 1) s = t[0];
52                          else s = t[1];
53                          output.push_back(it[col_to_index[s]]);
54                      }
55                      output_strings(output);
56                      cout << endl;
57                  }
58              }
59          }
60  }
```

2.4.2 对 2 张表的查询处理并输出结果

对查询 2 张表的条件的左右值进行处理：首先将所有查询条件的左右值形如 TABLE.COLUMN，用 . 分割得到表名和列名，由此可以根据 table1 或 table2 的记录得到对应的值，再将存在的”” 双引号去除，最后返回左值和右值。

实现代码如下：

```
1   // 对查询2张表的条件的左右值进行处理
2   // t:左右值 table:表名集合 it1: table1的记录 it2: table2的记录
3   vector<string> process_condition_for_two_table(const vector<string>& t,
4   const vector<string>& table,const vector<string>& it1,const vector<string>& it2) {
5       string x = t[0], y = t[1];
6       string xx, yy;
7       vector<string> xt = splitString(x, '.');
8       vector<string> yt = splitString(y, '.');
9       if (xt.size() == 1) {
10          if (!col_to_index.count(xt[0])) {
11              xx = xt[0];
```

```
            if (xx[0] == '\"' && xx.back() == '\"') xx = xx.substr(1, xx.size() - 2);
        } else xx = it1[col_to_index[xt [0]]] != "" ?
        it1[col_to_index[xt [0]]] : it2[col_to_index[xt [0]]];
    } else {
        if (xt[0] == table[0]) xx = it1[col_to_index[xt [1]]];
        else xx = it2[col_to_index[xt [1]]];
    }
    if (yt.size() == 1) {
        if (!col_to_index.count(yt[0])) {
            yy = yt[0];
            if (yy[0] == '\"' && yy.back() == '\"') yy = yy.substr(1, yy.size() - 2);
        } else yy = it1[col_to_index[yt [0]]] != "" ?
        it1[col_to_index[yt [0]]] : it2[col_to_index[yt [0]]];
    } else {
        if (yt[0] == table[0]) yy = it1[col_to_index[yt [1]]];
        else yy = it2[col_to_index[yt [1]]];
    }
    return {xx,yy};
}
```

处理对两张表的查询并输出查询结果：

① 首先两重循环遍历查询两张表中的每一个记录，用 flag 标志记录查询语句中的条件是否成立，成立 flag=1，否则 flag=0。

② 针对每一个记录判断是否满足查询的所有条件约束，先遍历判断所使用的判别符（如 =、! =、>= 等等），用判别符分割查询条件，得到左值和右值的字符串。

③ 调用 process_condition_for_two_table 对左右值进行简化，并分出从属哪一张表，进而获得最终的左值和右值。

④ 对简化后的结果根据判别符判断是否成立，不成立则修改 flag 并退出。

⑤ 若 flag=1 即所有条件都成立，则根据查询语句中的表和列（这里同样需要对查询内容从属哪一张表做处理），输出对应的查询结果；否则查询结果为空。

实现代码如下：

```
// 处理对两张表的查询
```

```
2  void process_queries_for_two_table(const vector<string>& table,
3  const vector<string>& attribute, const vector<string>& where) {
4      for (auto& it1 : TABLE[table[0]]) {
5          for (auto& it2 : TABLE[table[1]]) {
6              int flag = 1; // 判断where之后的条件是否成立的标志
7
8              // 判断where之后的条件是否成立
9              for (auto& w : where) {
10                 vector<string> t;
11                 // 针对不同的判别符进行处理
12                 if (w.find("!=") != -1) { // !=
13                     t = splitString_str (w, "!=");
14                     vector<string> temp = process_condition_for_two_table(t,table,it1 , it2 );
15                     string xx = temp[0],yy = temp[1];
16                     if (xx == yy) {
17                         flag = 0;
18                         break;
19                     }
20                 } else if (w.find(">=") != -1) { // >=
21                     t = splitString_str (w, ">=");
22                     vector<string> temp = process_condition_for_two_table(t,table,it1 , it2 );
23                     string xx = temp[0],yy = temp[1];
24                     if ( stoll (xx) < stoll (yy)) {
25                         flag = 0;
26                         break;
27                     }
28                 } else if (w.find("<=") != -1) { // <=
29                     t = splitString_str (w, "<=");
30                     vector<string> temp = process_condition_for_two_table(t,table,it1 , it2 );
31                     string xx = temp[0],yy = temp[1];
32                     if ( stoll (xx) > stoll (yy)) {
33                         flag = 0;
34                         break;
35                     }
36                 } else if (w.find("=") != -1) { // =
```

```
37                  t = splitString_str(w, "=");
38                  vector<string> temp = process_condition_for_two_table(t,table,it1, it2);
39                  string xx = temp[0],yy = temp[1];
40                  if (xx != yy) {
41                      flag = 0;
42                      break;
43                  }
44              } else if (w.find(">") != -1) { // >
45                  t = splitString_str(w, ">");
46                  vector<string> temp = process_condition_for_two_table(t,table,it1, it2);
47                  string xx = temp[0],yy = temp[1];
48                  if (stoll(xx) <= stoll(yy)) {
49                      flag = 0;
50                      break;
51                  }
52              } else if (w.find("<") != -1) {// <
53                  t = splitString_str(w, "<");
54                  vector<string> temp = process_condition_for_two_table(t,table,it1, it2);
55                  string xx = temp[0],yy = temp[1];
56                  if (stoll(xx) >= stoll(yy)) {
57                      flag = 0;
58                      break;
59                  }
60              }
61          }
62
63          // 如果条件成立,输出查询结果
64          if (flag) {
65              if (attribute[0] == "*") { // 输出查询表中的所有列
66                  for (int i = 0; i < 7; i++) {
67                      if (it1[i] != "") {
68                          cout << it1[i] << " ";
69                      }
70                  }
71                  for (int i = 0; i < 7; i++) {
```

```
72                          if (it2[i] != "") {
73                              cout << it2[i] << " ";
74                          }
75                      }
76                      cout << endl;
77                  } else { // 输出查询表中的查询对应的列
78                      for (int i = 0; i < attribute.size(); i++) {
79                          string a = attribute[i];
80                          // 判断 TABLE.COLUMN ,输出对应TABLE的COLUMN
81                          vector<string> t = splitString(a, '.');
82                          if (t.size() == 1) {
83                              string x = it1[col_to_index[t[0]]] == "" ? it2[col_to_index[t[0]]] :
84                              it1[col_to_index[t[0]]];
85                              cout << x << " ";
86                          } else {
87                              string x = t[0] == table[0] ? it1[col_to_index[t[1]]] :
88                              it2[col_to_index[t[1]]];
89                              cout << x << " ";
90                          }
91                      }
92                      cout << endl;
93                  }
94              }
95          }
96      }
97  }
```

3 利用数据库查询优化的版本（pass 9 test cases）

3.1 初始版本的问题

上一个版本在第 5 个测试样例时间超时，所以需要进行时间上的优化，其中最耗时的是两层循环要遍历两张表的所有数据，如果一张表的记录数是 m，另外一张是 n，则时间复杂度为 O(mn)。

3.2 实验思路

可以利用数据库优化查询中的知识，先利用第一张表必须筛选出第一张表中符合要求的记录，再对第二张表做同样的操作，再对两张表筛选过后的结果使用二重循环查找对两张表都有约束的条件下成立的记录。

为了避免空间开销过大，只保存筛选之后的记录在原表中的索引。

3.3 代码实现

3.3.1 对查询的列和条件预处理

由于处理字符串是较为耗时的操作，如果将其放入到二重循环中重复地操作，则会造成时间损耗，故在此对查询的列和条件进行预处理。

```
1  // 对针对两张表的查询的条件与查询的列预处理
2  void preprocess_queries_for_two_table(const vector<string>& table,
3  const vector<string>& attribute, const vector<string>& where){
4      // 对针对两张表的查询的条件预处理
5      for(auto&w:where){
6          // 分割出条件中的左值x 右值y 和 分割符z
7          vector<string>t;
8          string x,y,z;
9          if (w.find("=") != -1) { // =
10             t = splitClause(w, "=");
11             z = "=";
12         } else if (w.find(">") != -1) { // >
13             t = splitClause(w, ">");
14             z = ">";
15         } else if (w.find("<") != -1) {// <
16             t = splitClause(w, "<");
17             z = "<";
18         }
19         x = t[0], y = t[1];
20
21         // 对右值为字符串的情况处理
22         if((y[0] == '\"'&&y.back()=='\"' )){
23             y = y.substr(1,y.size()-2);
```

```
24              vector<string>temp = splitClause(x,".");
25              if(temp.size()==1){
26                  if(col_exist[table[0]][col_to_index[temp[0]]]){
27                      conditions_table1.push_back({x,y,z});
28                  }
29                  else {
30                      conditions_table2.push_back({x,y,z});
31                  }
32              }
33              else {
34                  if(table[0]==temp[0]){
35                      conditions_table1.push_back({temp[1],y,z});
36                  }
37                  else {
38                      conditions_table2.push_back({temp[1],y,z});
39                  }
40              }
41          }
42          // 对右值是数字的情况处理
43          else if(!col_to_index.count(y)&&y.find(".")==-1){
44              vector<string>temp = splitClause(x,".");
45              if(temp.size()==1){
46                  if(col_exist[table[0]][col_to_index[temp[0]]]){
47                      conditions_table1.push_back({x,y,z});
48                  }
49                  else {
50                      conditions_table2.push_back({x,y,z});
51                  }
52              }
53              else {
54                  if(table[0]==temp[0]){
55                      conditions_table1.push_back({temp[1],y,z});
56                  }
57                  else {
58                      conditions_table2.push_back({temp[1],y,z});
```

```
59                  }
60              }
61          }
62          // 对左右值都是表的列处理
63          else{
64              int t1,t2;
65              string c1,c2;
66              // 处理左值
67              vector<string>temp = splitClause(x,".");
68              if(temp.size()==1){
69                  if(col_exist[table[0]][col_to_index[temp[0]]]){
70                      t1 = 0;
71                      c1 = x;
72                  }
73                  else {
74                      t1 = 1;
75                      c1 = x;
76                  }
77              }
78              else {
79                  if(table[0]==temp[0]){
80                      t1 = 0;
81                      c1 = temp[1];
82                  }
83                  else {
84                      t1 = 1;
85                      c1 = temp[1];
86                  }
87              }
88
89              // 处理右值
90              temp = splitClause(y,".");
91              if(temp.size()==1){
92                  if(col_exist[table[0]][col_to_index[temp[0]]]){
93                      t2 = 0;
```

```
94                      c2 = y;
95                  }
96                  else {
97                      t2 = 1;
98                      c2 = y;
99                  }
100             }
101             else {
102                 if(table[0]==temp[0]){
103                     t2 = 0;
104                     c2 = temp[1];
105                 }
106                 else {
107                     t2 = 1;
108                     c2 = temp[1];
109                 }
110             }
111         conditions_table_mutual.push_back(make_pair(t1,c1));
112         conditions_table_mutual.push_back(make_pair(t2,c2));
113         conditions_table_mutual.push_back(make_pair(0,z));
114         }
115     }
116
117     // 对查询的列预处理
118     if(attribute[0]!="*"){
119         for (int i = 0; i < attribute.size(); i++) {
120                 string a = attribute[i];
121                 vector<string> t =  splitClause(a, ".");
122                 int x,y;
123                 if (t.size() == 1) {
124                     y = col_to_index[t[0]];
125                     x = col_exist[table[0]][y]?0:1;
126                 } else {
127                     y = col_to_index[t[1]];
128                     x = t[0] == table[0]? 0:1;
```

```
                }
                Col.push_back({x,y});
            }
    }
}
```

3.3.2 对表进行筛选并查找结果

```
// 处理对两张表的查询
void process_queries_for_two_table_2(const vector<string>& table,
const vector<string>& attribute, const vector<string>& where,const
vector<vector<string>>&table1,const vector<vector<string>>table2) {
    vector<int>index1,index2;
    // 对表二进行筛选
     for(int i =0 ;i<table2.size (); i++){
        int flag = 1;
        for(auto&it:conditions_table2){
            string a ,b,c;
            a = it [0], b = it [1], c = it [2];
            if (!judge(table2[i][col_to_index[a]], b,c)){
                flag = 0;
                break;
            }
        }
        if(flag)index2.push_back(i);
    }
    // 对表一进行筛选
    for(int i =0 ;i<table1.size (); i++){
        int flag = 1;
        for(auto&it:conditions_table1){
            string a ,b,c;
            a = it [0], b = it [1], c = it [2];
            if (!judge(table1[i][col_to_index[a]], b,c)){
                flag = 0;
                break;
```

```
28              }
29          }
30          if(flag)index1.push_back(i);
31      }
32      // 最后对筛选的结果进行二重循环查找
33      for(auto&i1:index1){
34      vector<string>it1 = table1[i1];
35      for(auto&i2:index2){
36          vector<string>it2 = table2[i2];
37          int flag = 1;
38          if(conditions_table_mutual.size()>0){
39          int t1,t2;
40          string c1,c2,z;
41          t1 = conditions_table_mutual[0].first,c1 = conditions_table_mutual[0].second;
42          t2 = conditions_table_mutual[1].first,c2 = conditions_table_mutual[1].second;
43          z = conditions_table_mutual[2].second;
44          c1 = t1==0 ? it1[col_to_index[c1]] : it2[col_to_index[c1]];
45          c2 = t2==0 ? it1[col_to_index[c2]] : it2[col_to_index[c2]];
46          if(!judge(c1,c2,z))flag = 0;
47          }
48          if(flag){
49              vector<string>output;
50              if (attribute[0] == "*") { // 输出查询表中的所有列
51                  for (auto&i:col_id[table[0]]) {
52                          output.push_back(it1[i]);
53                  }
54                  for (auto&i:col_id[table[1]]) {
55                          output.push_back(it2[i]);
56                  }
57                  output_strings(output);
58                  cout << endl;
59              } else { // 输出查询表中的查询对应的列
60                  vector<string>output;
61                  for(auto&it:Col){
62                      int x = it[0], y = it[1];
```

```
63                      string temp = x == 0?it1[y]:it2[y];
64                      output.push_back(temp);
65                  }
66                  output_strings(output);
67                  cout << endl;
68              }
69          }
70      }
71  }
72  }
```

4 利用数据局部性优化的版本 (pass a11 10 test cases)

4.1 上个版本的问题

上一个优化版本中，测试样例过了 8 个，在第 9 个测试样例发生了超时，所以还需进一步在程序的运行时间上进行优化。

在寻找可优化的环节中，我意识到尽管在对两表都有约束的条件的两重循环之前进行了两张表各自的筛选，使得筛选过后结果表的大小减小了．但是如果筛选后的结果表还是太大．还是会发生超时问题。

4.2 实验思路

从操作系统中缓存和数据一致性来看，筛选后的结果表还是太大导致的超时不仅是因为两重循环的时间复杂度搞，还是因为两张表太大导致两层循环的大量数据换出换入缓存、数据在缓存中 miss 带来的延迟太大。

故为了避免大量数据换入换出而带来的数据一致性的问题，我使用较小的数据结构来实现匹配。具体操作是只保留表 1 和表 2 筛选过后的记录中的 pair<id,col>，id 为其索引、col 为该 table[id] 数据对应约束条件的值，两重循环遍历的就不再是筛选结果的表记录的所有列．而是筛选结果的表记录的索引和对应条件的值，这样换入的数据就较小，换出的数据也较小，利用了数据一致性以减少访问慢速主存的时间，提高数据访问速度，优化程序性能。

4.3 代码实现

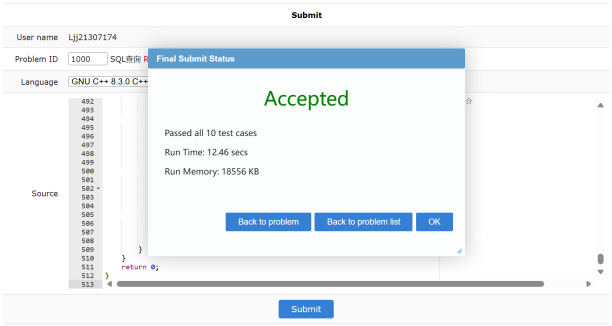
```
// 利用数据库查询优化和数据局部性对2张表查询的处理
void process_queries_for_two_table(const vector<string>& table,
const vector<string>& attribute, const vector<string>& where,const
vector<vector<string>>&table1,const vector<vector<string>>table2) {
    string c1,z;
    //c1两张表中比较的列 z判别符
    if(conditions_table_mutual.size()>0){
    c1 = conditions_table_mutual[0].second;
    z = conditions_table_mutual[2].second;}
    vector<pair<int,string>>t1; // 筛选后的表1记录的索引和对应的列
    vector<pair<int,string>>t2; // 筛选后的表2记录的索引和对应的列
    // 对表1进行筛选
    for(int i =0 ;i<table1.size (); i++){
        int flag = 1;
        for(auto&it:conditions_table1){
            string a ,b,c;
            a = it [0], b = it [1], c = it [2];
            if (!judge(table1[i][col_to_index[a]], b,c)){
                flag = 0;
                break;
            }
        }
        if(flag){
            t1.push_back(make_pair(i,table1[i][col_to_index[c1]]));
        }
    }
    // 对表2进行筛选
     for(int i =0 ;i<table2.size (); i++){
        int flag = 1;
        for(auto&it:conditions_table2){
            string a ,b,c;
            a = it [0], b = it [1], c = it [2];
            if (!judge(table2[i][col_to_index[a]], b,c)){
```

```
                flag = 0;
                break;
            }
        }
        if(flag){
            t2.push_back(make_pair(i,table2[i][col_to_index[c1]]));
        }
    }

    // 利用数据局部性优化(直接用表记录两重循环,大量数据换出换入缓存导致
    miss带来的延迟太大,故用较小数据结构来实现)
    vector<pair<int,int>>tt; // 条件匹配的两个记录在两张表中的索引
    for(auto&i1:t1){
        int i = i1.first;
        string x = i1.second;
        for(auto&i2:t2){
            int j = i2.first;
            string y = i2.second;
            if(!conditions_table_mutual.size()||judge(x,y,z))tt.push_back(make_pair(i,j));
        }
    }

    // 输出查询结果
    for(auto&x:tt){
        int i = x.first,j = x.second;
        if(attribute[0] == "*"){ // 输出查询表中的所有列
            vector<string>output;
        for(auto&i1:col_id[table[0]]){
                output.push_back(table1[i][i1]);
        }
        for(auto&i2:col_id[table[1]]){
                output.push_back(table2[j][i2]);
        }
        output_strings(output);
        cout << endl;
```

```
69      } else { // 输出查询表中的查询对应的列
70          vector<string>output;
71          for(auto&it:Col){
72              int x = it[0], y = it[1];
73              string temp = x == 0?table1[i][y]: table2[j][y];
74              output.push_back(temp);
75          }
76          output_strings(output);
77          cout << endl;
78      }
79      }
80
81  }
```

5 实验结果

测试结果：



测试结果

拓展部分

6 增加 INSERT 功能

6.1 实现过程

```
// 解析 INSERT 语句
void process_insert(const std::string& s) {
    std::string table_name;
    std::string col_names;
    std::string values_names;
    size_t values_pos = s.find("VALUES");
    size_t col_pos = s.find("(");

    // 分割出表名 列名 和 值
    table_name = s.substr(12, col_pos - 12);
    col_names = s.substr(col_pos,values_pos-col_pos);
    values_names = s.substr(values_pos + 7);
    // 去除空格
    table_name = removeSpaces(table_name);
    col_names = removeSpaces(col_names);
    values_names = removeSpaces(values_names);
    // 去除括号
    col_names = col_names.substr(1,col_names.size()-2);
    values_names = values_names.substr(1,values_names.size()-2);

    vector<string>cols = splitClause(col_names,",");
    vector<string>values = splitClause(values_names,",");

    // 插入数据
    vector<string>insert_data(7);
    for(int i = 0;i < cols.size() ;i++){
        insert_data[col_to_index[cols[i]]] = values[i];
    }
    TABLE[table_name].push_back(insert_data);
```

```
30
31 }
```

首先将 INSERT 语句进行分割，得到其表名、插入的列和插入的值。

再将值插入到该表的对应的列中，得到插入的记录。

最后将记录插入表中。

6.2 实现结果

输入:

```
3
2019001 law 19
2019002 info 20
2019003 info 19
2
20190001 math
20190002 English
2
2019101 info 49
2019102 info 38
2
2019002 20190001 100
2019003 20190002 0
1
20190001 2019102
3
SELECT * FROM Student
INSERT INTO Student (sid,dept,age) VALUES (21307174,cs,21)
SELECT * FROM Student
```

输出:

```
2019001 law 19
2019002 info 20
2019003 info 19
2019001 law 19
2019002 info 20
2019003 info 19
```

21307174 cs 21 可以看到 INSERT INTO Student (sid,dept,age) VALUES (21307174,cs,21) 后,Student 表中增加了 21307174 cs 21 的记录。

7 增加 DELETE 功能（条件中只能关联一张表）

7.1 实现过程

```
// 解析 DELETE 语句(条件只能一张表,不能两张表关联)
void process_delete( string s){
    // 对DELETE语句的不同部分进行分割
    string table_str;
    string where_str;


    int t = s.find("FROM");// table_str是FROM到WHERE之间的子字符串,即DELETE的表;
    int w = s.find("WHERE") == -1 ? s.size() + 1 : s.find("WHERE");// where_str是WHERE之后的
    子字符串(若无WHERE,则该where\_str为""), 即DELETE的条件

    table_str = s.substr(t + 5, w - t - 5);
    where_str = (w == s.size() + 1 ? "" : s.substr(w + 6));

    // 分割成集合
    vector<string> table = splitClause(table_str, ","); // 表集合
    vector<string> where = splitClause(where_str, "AND"); // 条件集合

    // 将上一轮的数据清除
    conditions_table1.clear();
    conditions_table2.clear();
    conditions_table_mutual.clear();
    Col.clear();

    for (auto it = TABLE[table[0]].begin(); it != TABLE[table[0]].end(); ) {
    int flag = 1; // 条件是否成立标志

    // 判断where之后的条件是否成立
```

```
29      for (auto& w : where) {
30          // 针对不同判别符处理
31          if (w.find("=") != std :: string :: npos) { // =
32              vector<string> t = process_condition_for_one_table(*it, w, "=");
33              string xx = t[0], yy = t[1];
34              if (xx != yy) {
35                  flag = 0;
36                  break;
37              }
38          } else if (w.find(">") != std :: string :: npos) { // >
39              vector<string> t = process_condition_for_one_table(*it, w, ">");
40              string xx = t[0], yy = t[1];
41              if ( stoll (xx) <= stoll(yy)) {
42                  flag = 0;
43                  break;
44              }
45          } else if (w.find("<") != std :: string :: npos) { // <
46              vector<string> t = process_condition_for_one_table(*it, w, "<");
47              string xx = t[0], yy = t[1];
48              if ( stoll (xx) >= stoll(yy)) {
49                  flag = 0;
50                  break;
51              }
52          }
53      }
54      // 如果条件成立,删除元素
55      if (flag) {
56          it = TABLE[table[0]].erase( it ); // 删除元素并更新迭代器
57      } else {
58          ++it; // 仅在未删除时递增迭代器
59      }
60  }
61  }
```

首先，解析 DELETE 语句，分割出 DELETE 的表名和 DELETE 的条件。

再循环遍历 DELETE 的表中的记录，寻找满足条件的记录。

最后将满足条件的记录从表中删除。

7.2 实现结果

输入: 3

2019001 law 19

2019002 info 20

2019003 info 19

2

20190001 math

20190002 English

2

2019101 info 49

2019102 info 38

2

2019002 20190001 100

2019003 20190002 0

1

20190001 2019102

3

SELECT * FROM Student

DELETE FROM Student WHERE dept = ” info”

SELECT * FROM Student 输出: 2019001 law 19

2019002 info 20

2019003 info 19

2019001 law 19

可以看到 DELETE FROM Student WHERE dept = ” info” 后,Student 表中 dept=info 的学生记录被删除。

8 心得体会

通过本次实验，我对编译原理的理论知识有了更深刻的理解，同时利用其他课程的知识来进行程序的优化，最后成功通过所有测试用例。

本次实验中对查询语句的处理和对特定关键词的理解处理．让我进一步对编译原理的理论知识进行了学习，通过关键词判断、分割语句等环节，复习了编译原理中的相关知识。

此次实验的基础部分中，我也遇到了许多困难，如对查询语句的处理、测试样例超时等等，最终都通过调试和进一步优化解决了。其中让我印象深刻的是为了避免超时而进行的优化操作，这里我运用到了数据库原理和操作系统的知识点来解决了。主要是运用数据库原理中学到的查询优化和操作系统中的数据局部性与缓存来解决并进行优化的，这提示我在日后的学习或者实践中要牢牢掌握之前的基础知识并做到融会贯通，将其运用到我们日后的学习工作中。

同时，在拓展部分，我增加了程序的 INSERT 和 DELETE 的功能，使程序的功能更加丰富，以后可以继续拓展更多的功能或进一步实现优化。

通过这次实验，我不仅加深了对理论知识的理解，更重要的是提升了实际操作能力。从存储数据、处理查询语句字符串、进行查询处理，再到调试和优化查询性能，每一步都需要细心和耐心。这个过程锻炼了我的动手能力和解决问题的能力。