|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **《数据库系统原理》实验报告（6）** | | | | | |
| **题目：miniOB 进阶实验** | | | | | |
| 学号 | 2251745 | 姓名 | 张宇 | 日期 | 2024.5.13 |
| **实验环境：miniOB数据库** | | | | | |
| 实验步骤及结果截图：   1. 开发调试环境准备   （1）启动镜像环境    （2）进行编译检查是否环境搭建成功     1. drop-table题目   （1）题目描述：  a. 删除表，清除表相关的资源。  b. 要删除所有与表关联的数据，不仅包括磁盘中的文件，还包括内存中的索引等数据。  （2）题目测试用例示例：    （3）实现思路：drop table 与create table相反，要清理掉所有创建表和表相关联的资源，比如描述表的文件、数据文件以及索引等相关数据和文件。  sql流转到default\_storge阶段的时候，在处理sql的函数中，新增一个drop\_table的case。  drop table就是删除表，在create table时，会新建一个t.table文件，同时为了存储数据也会新建一个t.data文件存储下来。同时创建索引的时候，也会创建记录索引数据的文件，在删除表时也要一起删除掉。  那么删除表，就需要删除t.table文件、t.data文件和关联的索引文件。  同时由于buffer pool的存在，在新建表和插入数据的时候，会写入buffer pool缓存。所以drop table，不仅需要删除文件，也需要清空buffer pool ，防止在数据没落盘的时候，再建立同名表，仍然可以查询到数据。  如果建立了索引，比如t\_id on t（id），那么也会新建一个t\_id.index文件，也需要删除这个文件。  这些东西全部清空，那么就完成了drop table。  具体代码如下：  在default\_storage\_stage.cpp 中的处理SQL语句的case中增加一个  case SCF\_DROP\_TABLE: {      const DropTable& drop\_table = sql->sstr[sql->q\_size-1].drop\_table; // 拿到要drop 的表      rc = handler\_->drop\_table(current\_db,drop\_table.relation\_name); // 调用drop table接口，drop table要在handler中实现      snprintf(response,sizeof(response),"%s\n", rc == RC::SUCCESS ? "SUCCESS" : "FAILURE"); // 返回结果，带不带换行符都可以    }  break;  在default\_handler.cpp文件中，实现handler的drop\_table接口：  RC DefaultHandler::drop\_table(const char \*dbname, const char \*relation\_name) {    Db \*db = find\_db(dbname);  // 这是原有的代码，用来查找对应的数据库，不过目前只有一个库    if(db == nullptr) {      return RC::SCHEMA\_DB\_NOT\_OPENED;    }    return db->drop\_table(relation\_name); // 直接调用db的删掉接口  }  在db.cpp中，实现drop\_table接口  RC Db::drop\_table(const char\* table\_name)  {      auto it = opened\_tables\_.find(table\_name);      if (it == opened\_tables\_.end())      {          return SCHEMA\_TABLE\_NOT\_EXIST; // 找不到表，要返回错误，测试程序中也会校验这种场景      }      Table\* table = it->second;      RC rc = table->destroy(path\_.c\_str()); // 让表自己销毁资源      if(rc != RC::SUCCESS) return rc;      opened\_tables\_.erase(it); // 删除成功的话，从表list中将它删除      delete table;      return RC::SUCCESS;  }  table.cpp中清理文件和相关数据RC Table::destroy(const char\* dir) {      RC rc = sync();//刷新所有脏页      if(rc != RC::SUCCESS) return rc;      std::string path = table\_meta\_file(dir, name());      if(unlink(path.c\_str()) != 0) {          LOG\_ERROR("Failed to remove meta file=%s, errno=%d", path.c\_str(), errno);          return RC::GENERIC\_ERROR;      }      std::string data\_file = std::string(dir) + "/" + name() + TABLE\_DATA\_SUFFIX;      if(unlink(data\_file.c\_str()) != 0) { // 删除描述表元数据的文件          LOG\_ERROR("Failed to remove data file=%s, errno=%d", data\_file.c\_str(), errno);          return RC::GENERIC\_ERROR;      }      std::string text\_data\_file = std::string(dir) + "/" + name() + TABLE\_TEXT\_DATA\_SUFFIX;      if(unlink(text\_data\_file.c\_str()) != 0) { // 删除表实现text字段的数据文件（后续实现了text case时需要考虑，最开始可以不考虑这个逻辑）          LOG\_ERROR("Failed to remove text data file=%s, errno=%d", text\_data\_file.c\_str(), errno);          return RC::GENERIC\_ERROR;      }      const int index\_num = table\_meta\_.index\_num();      for (int i = 0; i < index\_num; i++) {  // 清理所有的索引相关文件数据与索引元数据          ((BplusTreeIndex\*)indexes\_[i])->close();          const IndexMeta\* index\_meta = table\_meta\_.index(i);          std::string index\_file = index\_data\_file(dir, name(), index\_meta->name());          if(unlink(index\_file.c\_str()) != 0) {              LOG\_ERROR("Failed to remove index file=%s, errno=%d", index\_file.c\_str(), errno);              return RC::GENERIC\_ERROR;          }      }      return RC::SUCCESS;  }  （4）实验结果截图：     1. date题目   （1）题目描述：  A．要求实现日期类型字段。  B．当前已经支持了int、char、float类型，在此基础上实现date类型的字段。  C．date测试可能超过2038年2月，也可能小于1970年1月1号。注意处理非法的date输入，需要返回FAILURE。  D．这道题目需要从词法解析开始，一直调整代码到执行阶段，还需要考虑DATE类型数据的存储。  F．需要考虑date字段作为索引时的处理，以及如何比较大小;  G．这里没有限制日期的范围，需要处理溢出的情况。 - 需要考虑闰年。  （2）题目测试用例示例：    （3）实现思路：  语法上修改支持  需要可匹配date的token词和DATE\_STR值（一定要先于SSS，因为date的输入DATE\_STR是SSS的子集）  语法（yacc文件）上增加type，value里增加DATE\_STR值  [Dd][Aa][Tt][Ee]                     RETURN\_TOKEN(DATE\_T);  // 增加DATE的token，需要在yacc文件中增加DATE\_T的token  {QUOTE}[0-9]{4}\-(0?[1-9]|1[012])\-(0?[1-9]|[12][0-9]|3[01]){QUOTE} yylval->string=strdup(yytext); RETURN\_TOKEN(DATE\_STR);  // 使用正则表达式过滤DATE。需要在yacc文件中增加 %token <string> DATE\_STR  同时，需要增加一个DATE类型，与INTS，FLOATS等含义相同：  // in parse\_defs.h  typedef enum { UNDEFINED, CHARS, INTS, FLOATS, DATES, TEXTS, NULLS } AttrType;  Date的合法性判断  输入日期的格式可以在词法分析时正则表达式里过滤掉。润年，大小月日期的合法性在普通代码中再做进一步判断。  增加新的类型日期枚举  代码里有多处和类型耦合地方（增加一个类型，要动很多处离散的代码，基础代码在这方面的可维护性不好)  TupleRecordConverter：：add\_record时做格式转换，需要按照输出要求，将日期类型数据，转换成合适的字符串。  只要输入的日期不合法，输出都是FAILURE\n。包括查询的where条件、插入的日期值、更新的值等。这里在解析时（parse.cpp）中就可以直接返回错误。  （4）实验结果截图：     1. drop-table题目   （1）题目描述：  A．like 操作符用于在 WHERE 子句中搜索符合一定格式的字段。  B．'%'用于匹配零个到多个任意字符（英文单引号“'”除外），'\_'用于匹配一个任意字符（英文单引号“'”除外）。  C．只考虑char类型的字段即可。  （2）题目测试用例示例：    （3）实现思路：  在词法规则文件中添加能识别LIKE和NOT LIKE的词法规则    在语法规则文件中添加定义和对应的规则  a. TOKEN定义    b. 添加识别规则    c. 在定义文件中添加对应的运算符    d. 在具体执行比较的代码中添加LIKE和NOTLIKE的具体实现    （4）实验结果截图： | | | | | |
| **出现的问题：**   1. **gitee使用** 第一次使用gitee仓库，要拉取代码，还要上传代码提交测试，做的过程中很不习惯 2. **代码阅读**   在修改代码时，因为代码过于复杂，所以阅读代码成了一个大问题 | | | | | |
| **解决方案：**   1. **命令使用**   在多次操作失误后，逐渐也就习惯了，慢慢适应   1. **代码阅读**   我去观看了官方网站的视频和文档，对代码结构有了基础的了解，对于阅读出现困难的部分我使用了chatGPT来进行辅助分析 | | | | | |