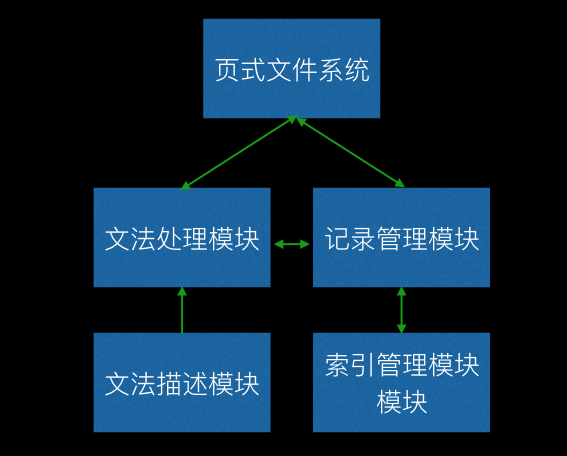
数据库大作业 总结报告

计65 赖金霖 2016011377

一、系统架构设计

我实现的数据库系统架构如上，其中箭头表示数据的流向。用户的输入首先进入文法描述模块，通过flex+bison的工具处理成程序理解的操作，传给文法处理模块。文法处理模块的功能是具体执行每一条语句，如果是系统操作就直接通过页式文件系统修改配置文件，如果是记录相关操作则通过记录管理模块进行操作。记录管理模块实际上是管理每张表及其索引，索引管理模块不直接访问页式文件系统，而通过记录管理模块间接访问。这样做的好处是，给文法处理模块一个统一的接口，并且对记录的操作能很方便地同步体现在索引上（如保证插入、删除、修改之后同步保存）。

二、各模块详细设计

1、文法描述模块

这部分包括lexer.l和Parser.y两个文件，其中lexer.l描述了定义了每个词语（区分大小写），Parser.y描述了每个SQL语句的语法。一条语句先被分成许多词语，然后被Parser识别成语法，最终被翻译成一个使用visitor设计模式的类，类的成员变量为语句所需要的变量，类中还有accept成员函数表示执行这一条语句。

这一部分一开始有一个的严重问题，Parser在处理一些很长的语句（如插入几万条记录）时复杂度退化到O(n^2)。对于反复的文法，有左递归和右递归两种实现方法，左递归的复杂度为O(n^2)，而右递归的复杂度为O(n)。我遇到的问题是左递归很慢，实现右递归后程序报错，经分析后我发现右递归需要一个和输入同长的符号栈支持。问题是首先我的语法树上每个结点需要的空间都很大，其次bison的默认配置中符号栈很小。我把每个结点改成指针类型（Parser.y每一条语法都要增加一行new semvalue()），虽然程序变得丑陋，但是节省了栈空间，然后我通过

#define YYMAXDEPTH 1000000

#define YYINITDEPTH 800000

的命令修改了默认栈大小，使得很长的指令也能右递归实现。

这一部分的实现有许多细节：

·Parser不需要像编译器那样遍历语法树，其实只要在语句级的语法结点建立类，然后在Statement级的语法结点访问accept函数，就可以满足语句解析的要求。

·Lexer和Parser生成的都是c语言程序，但我为了方便使用new语句及c++模板库，仍然用g++编译。

·parsertestmain.cpp是很简短的主程序代码，作用是把输入完全交由yyparse函数处理，因此如果用文件输入，执行完需要手动Ctrl+C退出。

·我对浮点数、日期的处理都相当tricky。浮点数由数字+’.’+数字组成，因此还不能识别科学记录法，但已经支持现有的数据集。日期最多8位，我用一个int存储，十进制下第5~8位位年，3~4位为月，1~2位为日，这么做可以直接对日期比较大小，因此支持在日期上建立索引。日期格式为数字+’-‘+数字+’-’+数字，还可以从字符串转化（去掉字符串两端的’，匹配前面的数字格式）。

·对于字符串，我用

'(\\'|[^'])\*'

的正则表达式匹配，满足了转义字符的要求。

·为了支持退出，我增加了EXIT;命令，表示退出数据库。

2、文法处理模块

这一部分主要是handler.cpp（最大的一个文件）和handler.h文件。在handler.cpp中，定义了每种语句类的constructor和accept函数，以及这些函数用到的辅助函数。

文法分为系统文法和记录文法两种。在每一条系统文法执行前，有一个checkglobalconfig()函数被调用，作用是检查目录下是否有.config文件，如果没有则建立。这一文件的作用是存储所有数据库名，用于SHOW DATABASES;语句。下面对所有语句的具体实现进行说明：

CREATE DATABASE db;被分解成三步：建立同名文件夹、建立db/.config文件、在.config文件中增加数据库。其中db/.config文件存储了表的信息，第一页存储表的名字和详细信息的位置（在第几页），之后每一页表示一张表的详细信息，包括表的名字和每一列的类型、大小等。

DROP DATABASE db;只需要删除文件夹、并删除.config文件中的信息就行了。需要注意的是我通过用.config文件中最后一条数据库信息覆盖被删除的数据库信息来删除，因此假如当前数据库显示为db1,db2,db3，删除db1之后显示会变成db3,db2。

USE DATABASE db;程序执行这一条语句时从db/.config读出表的列表，并对每一张表，在RecordManager里打开表文件，再识别并读出所有索引。

SHOW TABLES;程序切换数据库时直接读取了表的信息，所以这一条语句直接输出存储的内容。

CREATE TABLE …;程序执行建表操作分两步：一是检查语法是否合法，比如PRIMARY KEY是否存在、FOREIGN KEY是否合法、COLUMN NAME是否重复等；二是计算出所有列信息并存于db/.config中。程序获取这些列信息有两个函数GetColumn和GetColumns，分别表示从文件中获取一个表的某一列信息（类型、位置等）和一个表的所有列信息。

DROP TABLE …;删除表只需要删除相关文件并删除db/.config内的信息。

DESC tb;调用之前的GetColumns函数，直接输出信息即可。

INSERT INTO …;插入记录分两步：检查合法性、插入记录。程序除了检查类型是否匹配外，还会检查FOREIGN KEY是否存在，PRIMARY KEY是否重复。当所有待插入记录看上去合法之后，程序才依次执行插入。这里有一个实现上的问题，我预想的日期格式（yyyy-mm-dd）和orderDB不一样，所以在orderDB插入日期时是把STRING cast成DATE，日期合法性无法在前一阶段检查，所以在插入过程中如果检查到日期不合法，程序就会退出，舍去之后的记录。

DELETE FROM …;删除涉及到对WHERE的解析，有多种实现方法。我的方法是先检查语义合法性（条件是否可比之类的），然后从WHERE中找到一条区间操作（大于、小于之类的），如果区间操作的列上有索引，就可以用索引提前筛选出一些记录，如果没有索引，就只能获取全部记录。在这之后，程序循环遍历所有记录，对每条记录判断是否符合WHERE的限制，如果符合就加入待删除列表，最后统一删除。

UPDATE …;UPDATE相当于先DELETE后INSERT，UPDATE SET需要经过INSERT语句中的检查，WHERE限制的筛选方式和DELETE语句类似，最后统一进行UPDATE。这里有一个细节，如果UPDATE了PRIMARY KEY，需要检查UPDATE的数量是否＞1，如果＞1，会造成重复的PRIMARY KEY，我这里的处理是直接退出报错，一条也不更改。

SELECT …;SELECT语句首先检查SELECTOR和TABLE LIST是否合法，然后根据WHERE里的区间操作和是否有索引对每一张表都筛出一部分记录或者全部记录（与DELETE里类似）。和DELETE不同的是，SELECT需要进行一个递归，我用WHILE循环模拟这一递归，实现表之间的笛卡尔积。这个递归可能很慢，我有一个小的优化，递归的每一层枚举的是一张表的元素，如果低层的值确定了，高层的记录范围可能会因为WHERE语句缩小，因此在低层确定枚举的记录之后，我对高层记录进行了一次筛选（如果有INDEX），快速提高了一些场景下多张表连接的速度（如用FOREIGN KEY SELECT）。

CREATE INXEX …;我做的是对相邻4byte支持索引。这样能直接加速INT,FLOAT,DATE的查询速度，看上去无法加速CHAR。但实际上我的实现中CHAR前4byte存储长度，用INDEX筛选并不是错误的，只是范围略粗糙。

DROP INDEX …;记录管理模块识别是否有INDEX是靠是否有对应的文件判断的，所以直接删除文件就能删除INDEX。

3、记录管理模块

记录管理模块有三大功能：文件管理、记录管理、索引管理。

文件管理调用了页式文件系统，把对一张表的操作包装成函数，向文法处理模块提供接口。文法处理模块对一张表进行一次操作时，会依次调用记录管理模块的一些函数，但此时结果没有存到文件里，只有当文法处理模块调用saveFile函数时，修改才被反映到文件里。

记录管理是把对一张表的操作翻译成对文件的操作，文件维护了记录数量、记录长度、页数量等信息，插入操作是插入到末尾，删除操作是用末尾覆盖被删除记录，查找函数filterRecord需要提供checker。我的记录实际存储长度比定义长度多4，作用是多一个内部的PRIMARY KEY方便查找，我在打开文件时会自动给这一个字段添加索引。

索引管理有四个函数：createIndex, hasIndex, findbyIndex, deleteIndex。功能如字面意思，其中findbyIndex是有索引之后filterRecord函数的替代品。

4、索引管理模块

索引管理模块总体上是一个B+ Tree。每个结点都可以变成一个CHAR\*，存于文件，而文件里的CHAR\*也能翻译成B+ Tree Node。B+ Tree维护的是值->存储位置的映射，在查询时还要依赖记录管理模块把对应位置的记录提取出来，不过能因此能省下不少空间。B+ Tree内部还维护了存储位置->值的映射（存储位置通常很小，因此很容易维护），方便了许多操作。

三、接口说明

1、文法描述模块

这部分没有什么接口，主程序调用yyparse()函数，就把输入全部交给文法描述模块了。需要注意的是语法树结点的semvalue类特别大（为了统一维护所有语法），最好使用指针。

2、文法处理模块

文法处理模块是频繁调用其他模块函数的一个模块，对外接口在handler.h内。其中有Type、Value、Set、Column、Selector、Expr、WhereItem、Field类，分别表示文法里的类型（INT、FLOAT之类的）、值（具体值）、对Column的更改、对Column的描述、Select的列、右值、Where项、列信息。而每条SQL语句都被翻译成Handler类的子类，定义也在handler.h中，使用相应的constructor建立handler之后，调用accept就可以在程序内部执行相应的语句。

3、记录管理模块

记录管理模块有一个RecordFilter类，用于filterRecord判断一条记录是否需要返回。它有五个已经实现的子类：EqualFilter、GreaterFilter、LowerFilter、RegexFilter、TransparentFilter，功能如字面所示。

此外，这一模块的函数有：

void createFile(const char\*, int)：根据名字和记录长度创建表。

void deleteFile(const char\* )：根据名字删除一张表。

int openFile(const char\*)：打开一张表，返回表的编号。

void saveFile(int)：根据表编号存储一张表。

bool closeFile(int)：根据表编号关闭一张表。

void closeAll()：关闭所有表。

int insertRecord(int,const char\*)：根据表编号插入一条记录，返回记录编号。

int findRecord(int,int)：根据表编号和记录编号查找一条记录在文件中的位置。

bool deleteRecord(int, int)：根据表编号和记录编号删除一条记录。

bool renewRecord(int,int,char\*)：根据表编号和记录编号更新一条记录。

std::vector<char\*> filterRecord(int,int,int,const char\*,RecordFilter\*)：根据表编号、查找区间左端点、查找区间右端点、给checker传入的字符串、checker来筛选记录，其中每条记录的前4byte存储记录编号。

uint parseIndex(const char\*)：这个函数和索引无关，功能是从filterRecord的一条结果获取这条记录的编号（前4byte变成int）。

int getRecordLength(int)：根据表编号获取记录长度。

bool hasIndex(int,int)：根据表编号和索引编号判断一张表是否有某个记录（记录编号为列左端点/4）。所有的Index操作都要考虑内部的PRIMARY KEY，比如第一列的INDEX编号为1，左端点为4，右端点为8。

int createIndex(int,int,int)：根据表编号、列左端点、列右端点建立索引，返回索引编号。部分出于稳定存储空间的考虑（否则B+ Tree结点大小不一定），程序只实现了右端点-左端点=4的索引。

void deleteIndex(int,int)：根据表编号、索引编号删除索引。

std::vector<char\*> findbyIndex(int,int,int,int)：根据表编号、索引编号、查找下界、查找上界用索引筛选出一些记录。

4、索引管理模块

BTreeNode类是一个B+Tree的结点，它有tobuffer()函数，用于把自己变成buffer类型。

BTree类是实际的B+Tree，有以下函数：

std::vector<uint\*> tobuffers()：把整棵树转成buffer类型。

void insertdata(int,int)：插入一条数据（值->位置）。

void changedata(int,int)：根据旧位置获取值，然后改成新位置（输入旧位置、新位置，把值->旧位置改成值->新位置）。

std::vector<int> finddata(int,int)：根据值的区间（左闭右开），返回位置列表。

int findexact(int)：根据值查找一个位置。

void deletedata(int)：根据位置删除数据。

void buildtree(std::vector<uint\*>& inp)：从buffer类型建立B+Tree。

四、实现功能及实验结果

我实现了所有基础功能，包括数据库管理、表管理、记录操作。除此之外的扩展功能有创建删除索引、三张及以上表的连接。

对最终的大数据集，在对四张表的id建立索引之后，插入所需的时间在数秒之内。对于如下的语句

SELECT customer.name,food.name,orders.quantity FROM orders,customer,food WHERE customer.id=orders.customer\_id AND food.id=orders.food\_id AND orders.date='2017/11/11';

也能在数秒之内出解。

我的运行环境是macOS Mojave 10.14.2，直接make然后运行main就能执行了。如果要从文件输入，可以用./main < file命令。