Database System projects

Design report

Mini SQL

- Chen Jianyu -

- Duan Fuzheng -

- Zhuang Jingtian -

2017.6.5

§ Chapter 1 - 系统概述

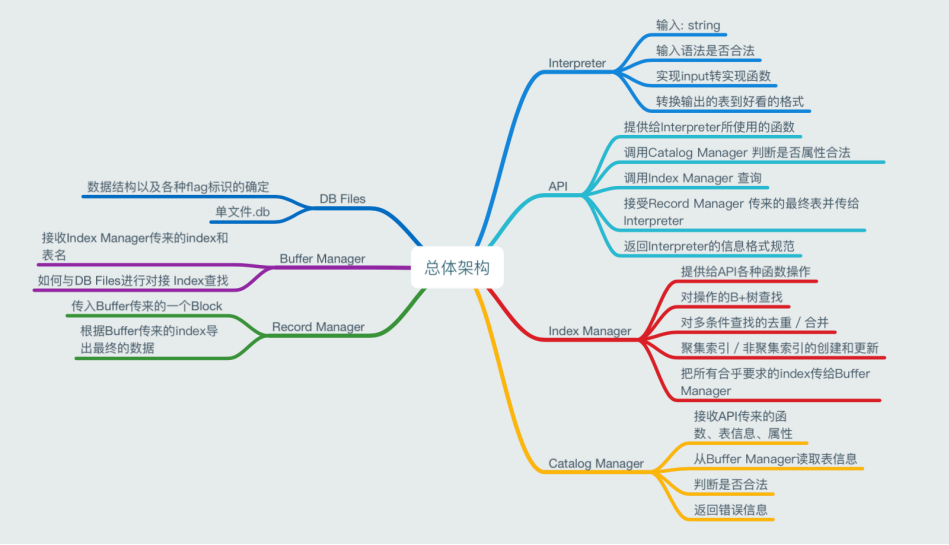
I 任务概述

实验要求设计并实现一个单用户 SQL 数据库，允许用户通过字符界面输入 SQL 语句实现表的建立/删除，索引的建立/删除，表记录的插入/删除/查找等操作。

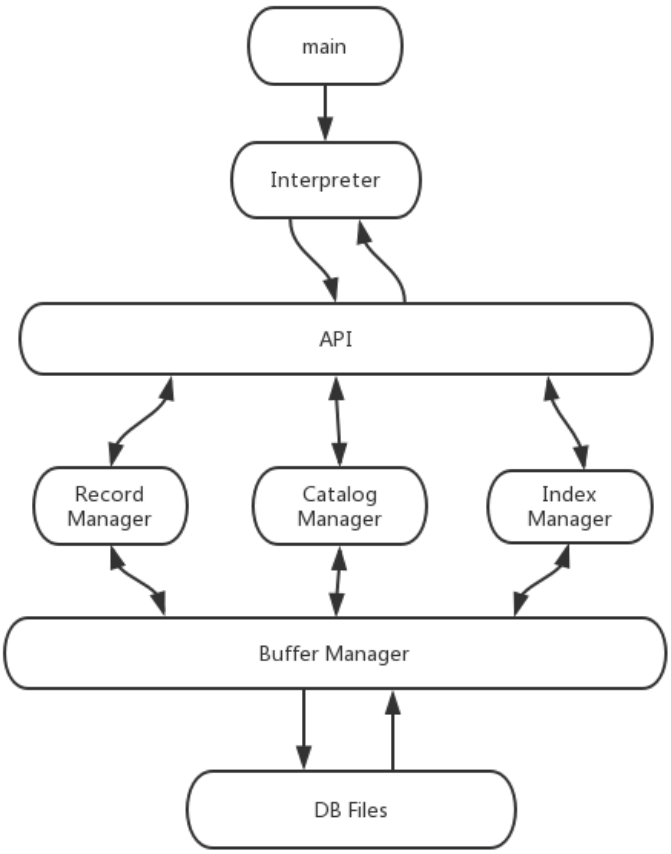
Table 要求支持最多可以定义 32 个属性，各属性可以指定是否为 unique，支持单属性的主键定义，对于表的主属性自动建立 B+ 树索引，对于声明为 unique 的属性可以通过 SQL 语句由用户指定建立/删除 B+ 树索引，并且所有 B+ 树索引都是单属性单值。

数据类型分为 int/float/string 三种，可以通过指定用 and 连接的多个条件进行查询，并且支持等值查询和区间查询、每次一条记录的插入操作、每次一条或多条记录的删除操作。需要支持标准的 SQL 语句，包括创建表、删除表、创建索引、删除索引、选择、插入、删除、退出系统、执行 SQL 脚本。其中每一条 SQL 语句以分号结尾，一条 SQL 语句可以为一行或多行，关键字大小写不敏感。

II 总体框架



总体框架初稿



总体框架结构

III 开发环境

Interpreter/API - 庄景天

IDE Xcode Version 8.3.2 (8E2002)

OS macOS Sierra 10.12.4

Index Manager/Catalog Manager - 段辅正

IDE Visual Studio 2017

OS Windows10 1703

Buffer Manager/Record Manager – 陈建瑜

IDE CLION 2016.03

OS Windows10 1703

§ Chapter 2 – 框架详细设计

I Interpreter

- 1 模块概述 -

Interpreter 是整个系统的最前端部分，承担了所有的输入输出环节，负责接收用户的输入 / 文件导入、词法分析、语法分析、返回结果处理与错误信息输出。

- 2 基本功能 –

*** 输入 / 文件导入**

Interpreter 支持用户以类命令行的形式输入语句和导入文件执行语句，在输入语句状态下，支持分行输入和多条语句同时输入，对于分行未输入完成的语句，会等待至用户输入的第一个分号。多条语句会按分号分隔执行，对于文件导入的执行，报错信息会具体到文件行数。

*** 词法分析**

词法分析采用了 flex 来将 flex 的词法标识文件 sql.l 转为包含正则表达式匹配与其对应的相应模式处理方法的 lex.yy.c 来进行词法分析，它将对 SQL 语句中的关键字(select)、特定符号(>=)、特定格式(int,float,string)等进行匹配。对于关键字和特定格式，返回相应的关键字标号，对于比较类符号，统一返回 COMPARISON 标号，用 subtok 来标记具体的符号类型，对于无法识别的符号，返回 ERROR_STR 。

*** 语法分析**

语法分析采用了 bison 将语法标识文件 sql.y 转为对应的的能对语句模式的匹配识别的 sql.tab.c，生成出的LALR语法分析器可以对输入的SQL语句进行匹配，在匹配成功后以 emit 函数返回一个给 API 的可执行序列，匹配失败后返回并输出 syntax error，bison 使用 flex 返回的关键字标号，并且在 emit 的过程中记录下了执行的相关参数，方便 API 使用。

* 返回结果处理

在 API 执行语句后，如果成功会返回最终对应行的内部序列格式，对其进行格式处理和对齐后，将执行的最终结果输出到屏幕，如果执行过程中 Catalog Manager 报错，将错误信息输出。

* 错误信息输出

对于语法错误，直接通过解析时的 yyerror 进行错误输出，对于 Catalog 报错的情况，Catch 其抛出的错误并且输出错误信息种类，如 table 名不存在/列名不存在等，对于文件执行的情况，报错信息会定位到行数。

- 3 接口设计 -

- void main_loop (istream& input);

Interpreter 作为最顶层的存在，没有需要向下提供的接口，只向 main 函数提供 main_loop 函数，以 cin 作为输入，对 cin 完成所有的执行操作。

- void yy_switch_to_buffer (YY_BUFFER_STATE new_buffer);

将 yyparse 的对象从等待用户输入转换到对 buffer 中内容进行解析。

- YY_BUFFER_STATE yy_scan_string (yyconst char *yy_str);

将 char* 的字符串进行scan操作并返回 YY_BUFFER_STATE 类型。

- int yyparse (void);

yyparse 将会对之前存入 buffer 的 sql 语句进行解析，解析完成后会保存可处理的执行序列，解析错误则会报错。

- void emit(char *s, ...);

yyparse 过程中会调用 emit 函数将当前分析过程的需要执行的序列存入 sql_from_bison，以空格间隔各 API 可接收的操作序列，并以 $ 标记序列的结束，其中...可变参数用以接收来自各节点的参数(如标记select的列数目)，用 va_list ap 的形式格式化写入。

- void yyerror(char *s, ...);

yyerror 会在 sql.tab.c 匹配失败时触发，输出错误信息以及错误行数。

- void clean_sql_from_bison();

初始化用 sql.tab.c 中用来存放可执行序列的字符串 sql_from_bison 。

II API

- 1 模块概述 -

API 是将前端 Interpreter 与Record Manager/Index Manager/Catalog Manager 连接起来的核心部分，它接收来自 Interpreter 处理后的可内部可执行序列，对其进行匹配并调用对应的函数执行语句，返回给 Interpreter 最终的需要输出的序列化的查询结果 / 执行的结果 / 错误信息。

- 2 基本功能 –

* 序列执行

Interpreter 返回的内部序列化可被 API 简单的以类似 switch/case 的方式分步执行，然后调用 Catalog Manager 的函数进行表名/列名进行检查，再调用 Index Manager 提供的函数对需要查找/删除/添加的列进行操作，最后从 Record Manager 获取最终返回的列信息序列到 Interpreter 进行输出。

* 语句分支列表

```
    select:
    select [*,(A),(A,B)] from [table_name] (where) [ A=3 (AND B between
2 AND 3) (OR C=3) ]
    select(name[], table);
    select(name[], where_result);
insert:
    insert into [table_name] values(A,B,C)
    insert(table, values[]);
    insert(table, name[], values[]);
update:
    update [table_name] set A=1,B=2 (where) [A=3 AND B between 2 AND 3
(OR C=3)]
    update(table, name[], values[]);
    update(table, name[], values[], where_result);
delete:
    delete [*] from [table_name] (where) [A=3 AND B between 2 AND 3 OR
C=3]
    delete(table);
    delete(table, where_result);
where_result:
```

```
    where(table, A, "=", B); -> "!=" ">" ">=" "<" "<=" "~"
    where(table, A, "~", B, C);
create:
    create table [table_name] (...)
    create index [index_name] on [table_name] (column_name)
    create(table, name[], type[], primary_key);
    create(index, table, name[]);
drop:
    drop table [table_name]
    drop index [index_name] on [table_name]
    drop(table);
    drop(index,table);
show:
    show tables;
    show status;
exit:
    quit;
    exit;
```

* WHERE处理

在进行 WHERE 操作时，API 还将负责完成 AND/OR 的交并操作，在 WHERE 操作时，会分别对每个单独的逻辑判断向 Index 进行查询，查询结束后对所有返回的Index的集合进行取交/并的操作，并将最后的结果向 Record Manager 进行查询。

* 错误返回

在执行过程中，Catalog Manager 检查过程中可能会报多种错误，throw 出一个供 Interpreter 来 catch 的错误，如表名不存在、列名不存在、名字重复等，将详细的错误信息传递给 Interpreter 来输出。

III Record Manager

- 1 模块概述

Record Manager 承担了系统中的记录管理功能，通过操作 Buffer Manager 来操作数据库存储文件，达到返回查询语句的返回值的目的。

- 2 基本功能

* 表的创建与删除

通过调用 BufferManager 来创建一个空白文件，文件名为tableName.MYD 在查找表的时候看文件名是否存在即可，该检查同样被封装在 BufferManager 中。

* 记录的插入

记录的插入对应的是文件的改变，实际上是先对内存操作，在析构的时候再写回文件。首先调用 BufferManager 取到对应的块（实际上就是最后一块），然后插入到末尾即可。

* 记录的查找（考虑多条件查找）

根据上层提供的 index vector 来依次查找记录，首先调用 BufferManager 取到对应的内存块，再根据 index 算出块内的索引，从而得到正确的结果。

* 实现从 BufferManager 返回的内容中记录的提取与保存

根据上层提供的输出格式把返回内容修改成对应的格式再返回。

* 支持定长记录、支持块内记录

实际上也仅仅支持定长记录，不允许 TEXT 类型的存在，从而解决跨 block 记录的问题。

- 3 接口设计

```
- bool createTable(string tableName);
```

createTable 传入一个表名，会对应的创建一个空的文件，文件名为 tableName.MYD

```
- bool dropTable(string tableName);
```

dropTable 是 createTable 的反操作

```
- int insertIntoTable(TableStruct &ts, char* data);
```

插入一条记录

```
- int deleteRecord(TableStruct &ts, vector<int> &scope, vector<int>
&moved);
```

scpoe 是一个存储了需要删除的元素的 index 的数组， index 从 0 开始，moved 会返回需要更新的索引。moved 的存在的原因是，删除记录的原理为

把最后一条记录移动到当前要删除的记录位置，这样就能既没有空洞又不需要更新大量的索引。moved 中两个为一对，比如 moved[0] 为要更新的索引位置，moved[1] 为新的索引位置，以此类推。

```
- bool selectRecord(TableStruct &ts, vector<int> &scope, vector<char *> &result);
```

scope 为需要选择的记录的 index， index 从 0 开始。result 会返回一个包含了结果集的字符串指针数组，请务必在使用了结果之后将这些指针分别 delete[]。

```
- bool selectAttribute(TableStruct &ts, string col, vector<char *> &values);
```

这是为建立索引提供便利的一个方法，col 为字段名，values 为查询结果，查询结果是按照文件中第几条记录的顺序排列的，请务必在使用了结果之后将指针 delete[]。顺带构造函数可以定义 blockSize。在使用 RecordManager 之前要先初始化 BufferManager ，再将其传入 RecordManager 的构造函数中。

IV Index Manager

- 1 模块概述

Index Manager 负责实现B+树索引的实现，承担了索引的创建和删除，以及承担等值查询，插入删除键值时候的搜索操作

- 2 基本功能

* B+树的建立，删除，维护

Index Manager 支持用户建立 键值分别为 int, float, char[] 三种B+树。并提供添加，删除（软删除）的操作。由于内存的速度远远大于硬盘读取文件的速度，因此一个block对应了B+树中的一个node，以得到最大的效率。

* Index 的创建，维护

Index Manager 提供从block还原index信息，将index中的信息保存到block，新建和删除index的功能。

* 搜索记录

Index Manager 记录了每条记录在block Manager中的位置，支持等值查询，插入、删除功能， 能较快地获取到记录的位置。

- 3 接口设计

```
- IndexInfo& create_index(const std::string& indexName, const std::string& tableName, const std::string& fieldName);
```

根据索引名字，表名，属性名创建索引

```
- void drop_index(const std::string& indexName);
```

根据索引名字删除索引

```
- void insert(const std::string& tableName, byte* key, const BlockPtr& ptr);
```

插入键值

```
- void remove(const std::string& tableName, byte* key);
```

删除键值

```
- bool has_index(const std::string& indexName);
```

根据索引名字检查是否存在

```
- std::vector<BlockPtr> search(const std::string& tableName, int key);
```

根据给出的 key(int) 查找记录所在的位置

```
- std::vector<BlockPtr> search(const std::string& tableName, float key);
```

根据给出的 key(float) 查找记录所在的位置

```
- std::vector<BlockPtr> search(const std::string& tableName, std::string key);
```

根据给出的 key(string) 查找记录所在的位置

```
- std::vector<BlockPtr> search_range(const std::string& tableName, int key_low, int key_high);
```

根据给出的 key(int) 范围查找记录所在的位置

```
- std::vector<BlockPtr> search_range(const std::string& tableName, float key_low, float key_high);
```

根据给出的 key(float) 范围查找记录所在的位置

```
- std::vector<BlockPtr> search_range(const std::string& tableName, std::string key_low, std::string key_high);
```

根据给出的 key(string) 范围查找记录所在的位置

V Catalog Manager

- 1 模块概述

Catalog Manager 负责维护数据库的元数据信息如表的定义信息，属性的定义信息。

- 2 基本功能

*** 表的新建，删除**

Catalog Manager 可以根据提供的信息新建表，根据表的名字删除表。

*** 表的查询**

Catalog Manager 可以根据信息查询表的存在。Catalog 在内存中使用 std::map 来保存表名与表的信息的对应关系，以便得到较快的查询速度

*** 表的维护**

Catalog Manager 可以从 block 中恢复表信息，也可以把表信息写入 block。每一个 block 中保存很多个表的信息，但是不会有一个表跨两个 block 储存。当一个 block 的剩余空间不足以添加一个新的表信息的时候，使用新的 block 来存储。每次程序启动的时候将整个 catalog 文件读入来构建 Catalog Manager，当程序结束时，将 Catalog 中的信息写入一个文件中。

- 3 接口设计

```
-void add_table(TableInfo& table);
```

根据表信息新建表

```
void remove_table(std::string& tableName);
```

根据表名删除表

```
TableInfo& find_table(std::string& table);
```

根据表名查找表

VI Buffer Manager

- 1 模块概述

BufferManager 是整个系统最底层的部分，主要目的是避免上层对文件的直接操作， 同时建立缓存以减少文件 IO， 核心部分为 LRU 近似算法 CLOCK 算法，核心数据结构为一个可以自由定义大小的 Buffer 内存，为了方便的管理各个表的 Buffer，设立了 BufferUnitManager 以适应多表的操作，而 BufferManger 则用来管理 BufferUnitManager。

- 2 基本功能

* 返回指定 index 的内容

首先查看 Buffer 中是否存在指定的 block，若存在则直接返回，若不存在则从文件中读取到 Buffer 内再返回，若 Buffer 已满则按照 Clock 算法的规则替换。

* lock 保证原子性

在一个 Block 被执行写操作时锁定该 Block，防止同时进行不同的读写。

* 缓存

即每个 Block 在被带入内存的时候 flag 设置为 0 ，若被访问到则 flag 设为 1，若执行了编辑操作则 flag 不改变，每次替换时只替换那些 flag 为 1 的 Block。

* edit 标志位确保最少文件 IO

这是一个减少文件 IO 的功能，编辑文件时先在内存中编辑，再给其设置 edited 标志位为 1，在析构时 edited 标志位为 1 的 Block 会被写回去，其余 Block 则跳过。

- 3 接口设计

- bool createFile(string filename);

创建空文件， filename 为文件名。

- bool deleteFile(string filename);

删除文件

- bool readDataFromFile(string filename, int blockIndex, char *readBuffer);

从指定文件的指定 block 读入内存，需要注意的一点是，在 blockSize 确定之后就要一直使用相同的 blockSize 来读取相同的文件，因为 Buffer Manager 有个自动对齐的操作，文件大小会自动对齐到 block 的整数倍， 并且需要读取的内容不可能在两个 block 中，也就是不存在跨区段记录。所以 blockSize 很重要。filename 指定了要读取的文件名，无须担心 buffer 是否建立的问题，如果没有建立的话会自动建立。blockIndex 是需要读取的第几个 block 的意思。readBuffer 由上层使用该接口的提供，读取进来的数据会被放在 readBuffer 之中。

- bool readDatas(string filename, vector<char *> results);

将一个文件中所有的 block 均读进来。

- bool writeDataToFile(string filename, int blockIndex, char *writeBuffer);

顾名思义，向指定 blockIndex 处写入指定数据。大小就是 blockSize。前两个参数同上， writeBuffer 指的是要写的数据对应的内存指针。

- bool lockBlock(string filename, int blockIndex, int lock);

需要时使用，如果 lock 一个 block 那么该 block 不可被写。前两个参数同上， lock 指的是要设定的标志值， 1 的话是 lock， 0 的话 unlock。

- bool deleteLastBlockOfFile(string filename);

这个接口是因为我的设计中，删除一个记录是把最后一条记录给移动到前面去，但是暂时并不删除最后一条记录，只是更新记录的数量，让最后一条记录变得不会被访问而已，在文件最后一个 block 全部被移动到前面之后再删除这个 block，主要目的是减少文件 IO。参数说明同上。

§ Chapter 3 – 综合测试方案

I 基础功能测试

Create 操作

```
create table course
(
    id int,
    name varchar(12) unique,
    score int,
    primary key(id)
);
```

int/float/string 类型上的等值条件查询

```
select * from course_db where id=3150104958;
select * from course_db where score=59;
select * from course_db where name='Clapeysron';
```

int/float/string 类型上的非等值条件查询

```
select * from course_db where id!=3150104958;
select * from course_db where score>=59;
select * from course_db where name<'Clapeysron';
```

int/float/string 类型上的多条件查询

```
select * from course_db where id!=3150104958 and score>50;
select * from course_db where score between 40 and 90;
select * from course_db where id!=3150104958 and score>90 or score<40;
```

非 select *

```
select name,score from course_db where id!=3150104958 and score>50;
```

update 语句

```
update course_db set name='Zhuang Jingtian' where id=3150104958;
update course_db set score=60 where score<60 or score>95;
```

检查约束冲突

```
insert into course_db values(3150104958,'Clapeysron',59);
insert into course_db values(3150104958,'Melody',58);
```

创建 INDEX 并观察执行时间变化

```
select * from course_db where name='Clapeysron';
create index name_index on course_db ( name );
```

```
select * from course_db where name='Clapeysron';
```

检查创建 INDEX 后是否能继续插入

```
insert into course_db values(3150102769,'Melody',58);
```

检查创建 INDEX 后是否能正确 delete

```
delete from course_db where name='Melody';
```

删除INDEX

```
drop index name_index on course_db;
```

检查没有 INDEX 时的 delete

```
delete from course_db where name='Clapeysron';
```

delete 整个表

```
delete from course_db;
```

drop 一个表

```
drop table course_db;
```

表不存在

```
select * from course_db;
```

显示出所有表

```
show tables;
```

显示目前系统情况

```
show status;
```

退出系统

```
exit;
```

II 压力测试

利用实现用随机函数生成的脚本进行巨量数据（1000~10000）的插入，删除操作

```
insert into course_db value(315010****, "******", xx);
delete * from course_db where id = 315010****;
```

向生成并插入的数据中进行搜索操作

```
select * from course_db where id = 315010****;
select * from course_db where name = "******"
```

在已经插入很多数据的情况下新建uquiue属性的索引

```
create index "***" in table "xxxx" on "xxxx";
```

一次性拿出许多数据

```
select * from course_db;
```