数据库系统概论 项目报告

徐潇悦 2020011089

January 2024

1 Introduction

这是清华大学 2023 秋数据库系统概论课程的项目报告。在本项目中，我实现了一个基础的数据库系统（称为 TestDB），支持数据管理、数据库和数据表管理等功能。

2 系统架构设计

TestDB 包含以下模块，每个模块对应代码/src 中的一个子文件夹。

```
.
├── parser : antlr4 生成的文法解析器
├── filesystem : 课程提供的文件管理系统
├── db_manage : 解析器、系统管理
├── table : 单数据表管理
├── record_manage : 记录管理
├── index : 索引管理
└── frontend : 读入交互指令
```

模块之间的依赖关系入课程文档给出的大致相同。

例如，当一条 insert 文件传入 TestDB 后，会经历 parse → DBVisitor(db_manage) → DBManager(db_manage) → TableManager(table) → RecordFile(record_manage) → IndexManager(index) → IndexFile(index) 这样的过程。

3 各模块详细设计

3.1 parser

该模块由 antlr4 根据课程提供的 SQL.g4 文法自动生成。

3.2 filesystem

该模块来自课程提供的 filesystem 文件，包含文件管理和缓存管理。

3.3 db_manage

该模块主要包含两个子模块 DBVisitor 和 DBManager。

3.3.1 DBVisitor

继承了 antlr4 生成的 SQLBaseVisitor 类，重写了其中的若干 visit 函数，负责对不同类型的命令进行解析。DBVisitor 包含一个 DBManager 对象，解析命令后通过调用 DBManager 的成员函数转化为数据库操作。

3.3.2 DBManager

数据库系统的顶层管理结构，提供了若干函数接口，并处理数据库和数据表的增删改工作。

- 数据库管理：DBManager 将一个 <db_name> 数据库视为./data 文件夹下的一个 <db_name> 子文件夹，通过创建、删除、遍历文件夹来进行数据库的增删查功能。同时 DBManager 维护了一个 using_table_name 来记录当前窗口正在使用的数据库名称，支持数据库的切换功能。
- 数据表管理：每个 <db_name> 数据库的 <tb_name> 数据表对应./data/<db_name> 下的 <tb_name> 子文件夹，DBManager 通过创建、删除、遍历文件夹来进行数据表的增删查功能。当 using_table 存在时，DBManager 使用一个 <name, TableManager> 的字典来维护当前打开的数据表。默认切换到数据库时会关闭上一个数据库所有的表并打开当前数据库所有表。

- 数据表命令：处理涉及具体数据表的命令，比如在表中增删查改数据、修改索引 schema 等，DBManager 会在字典中查找目的表是否存在，然后调用 TableManager 对象的成员函数，将命令转化成数据表操作。

涉及多张表的操作都由 DBManager 先处理成单表操作然后汇总。比如双表 Join 查询，DBManager 会先收集针对单表的 condition，交给单表获取对应行，然后把得到的行进行综合判断处理。

3.4 table

table 包含 4 个子模块。

3.4.1 TableManager

TableManager 是单个数据表管理的顶层模块，维护了 IndexManager、Re cordFile、MetaFile 对象。

- 创建表：DBManager 创建表时调用此构造函数。TableManager 会保存传入的 Schema，转告给 IndexManager 和 RecordFile，并且调用 MetaFile 保存表的元信息。
- 打开表：DBManager 打开表时调用此构造函数。TableManager 会调用 MetaFile 读取表的元信息，并且转告给 IndexManager 和 RecordFile。
- 数据增删查改：TableManager 对传入的 data 进行检查后，传给 RecordFile 进行具体操作。
- 索引增删查改：直接调用 IndexManager 的成员函数。如果索引没有命名，TableManager 将自动生成名称，比如 A 列上的索引叫做 index__on__A__。

3.4.2 MetaFile

MetaFile 负责数据表元信息的管理。MetaFile 在/data/<db_name>/<tb_name>文件夹下建立一个/meta 文件夹，并在 schema.txt 中记录 schema 信息。

3.4.3 Schema

Schema 负责管理表的列信息，包括列名、数据类型、主键和外键约束、是否允许 null、列的默认值。同时 Schema 文件提供了 FieldData 类，这是

数据库操作的基本数据单元，为 INT、FLOAT、VARCHAR 提供了一个统一的数据类，记录了数据的具体值和类型、是否 null。

3.4.4 SlotPage

SlotPage 提供了页面上定长槽的管理。SlotPage 维护了一个 bitmap，记录页面中槽的使用情况，并且提供了若干接口函数，支持向 buff_ptr 指向的页槽写入或者删除一条数据。SlotPage 也提供了从页面起始处的地址向第 i 个槽地址的转换。

3.5 record_manage

record_manager 包含 3 个模块。(代码中的 RecordSlot 是废弃模块，它实际上没有被使用。)

3.5.1 RecordFile

RecordFile 是记录管理的顶层结构，组织了若干 RecordPage。处理命令时，RecordFile 会根据需要遍历各个 page。RecordPage 的记录是通过链表实现的，每个页面会记录 prev 和 next 信息，RecordFile 记录了链表的头和尾。除此之外，RecordFile 也要求页面维护了一个非满页面的链表，这样就不需要在插入时遍历所有页面来找到空槽位。

RecordFile 的页面链表 head&tail、非满页面链表 head&tail 都记录在/meta 文件夹下的 record.txt 里面。打开数据表时 RecordFile 读取 record.txt 文件来获取信息，并且在修改数据后写入 record.txt。

RecordFile 还包含 RecordSchema 和 RecordPageSchema 对象，这是为了在创建或打开一个 RecordPage 时告诉它页面和槽的结构信息。

3.5.2 RecordPage

RecordPage 是记录文件页面的管理结构。RecordPage 包含一个 SlotPage 对象，用来管理槽操作。RecordPage 通过 RecordSchema 来增删查改槽位中的信息，通过 RecordPageSchema 来读取和修改 page_id、next_page、next_empty_page 等信息。

3.5.3 RecordSchema

RecordSchema 主要包含两个类，用来记录页面和槽的结构。

在创建表时，RecordFile 根据 Schema 得到页面的结构安排，比如 page_id_offset（page id 在页面的位置到页头的 offset）（由于我把页面的 bitmap 记录在页的最前面，槽的大小会影响 page_id_offset 这些信息），并得到每个槽的结构安排，比如槽的第 i 条数据相对于槽头的 offset。这些信息组织成 RecordSchema 和 RecordPageSchema，并且记录在/meta/record.txt 中。

- RecordSchema: 槽文件读取的重要结构，记录了各列的 offset 以及类型等信息。RecordSchema 可以通过 set_read 和 set_write 来设置要读取哪些列的信息。调用 RecordSchema::read 会使它从参数地址位置读数据到 result_data 对象，然后可以通过访问 result_data 对象将数据取出。
- RecordPageSchema: 页文件读取的重要结构，记录了页头各条目的 offset。RecordSchema 可以结合传入的页头地址和自己保存的 offset 读出页面上的信息 (除了 bitmap 和 free_slots，这两个由 SlotPage 管理)。

3.6 index

index 主要包含 3 个模块。

3.6.1 IndexManager

由于一个表上可能建有多个索引，我使用 IndexManager 来进行管理。<index_name> 名称的索引对应于/<tb_name>/index 文件夹下的 <index_name>.db 文件，其元信息存储在/<tb_name>/meta/index 文件夹下的同名.txt 文件中。

IndexManager 处理索引的创建和删除作用。它保存了若干 IndexPageSchema 和 SlotSchema 对象来记录索引页面的结构，工作方式和 RecordSchema 相似。

IndexManager 包含一个 IndexFile 的 vector，用来维护当前打开的所有索引文件。

```
// manage databases
void create_db(const std::string &db_name);
void drop_db(const std::string &db_name);
std::vector<std::string> show_db();
void use_db(const std::string &db_name);

// manage table
void create_table(const std::string &tb_name, const Schema & schm);
void drop_table(const std::string &tb_name);
void describe_table(const std::string &tb_name);
void show_tables();
void show_indexes();

// data
int load_from_file(const std::string filename, const std::string tb_name, const std::string divider);
void insert_data(std::string tb_name, std::vector<std::vector<FieldData>> data_list);
void select(bool, std::vector<std::string> column, std::string table, std::vector<Constrain> cons);
void delete_data(std::string tb_name, std::vector<Constrain> cons);
void update_data(std::string tb_name, std::vector<std::string> colums, std::vector<FieldData> new_data, std::vector<Constrain> cons);
void join_select(std::vector<std::string> tables, std::vector<std::string> columns, std::vector<std::string> names, std::vector<Constrain> cons);

// index
void add_index(std::string tb_name, std::string index_name, std::vector<std::string> column_name);
void drop_index(std::string tb_name, std::string index_name);

// pk
void drop_pk(std::string tb_name);
void add_pk(std::string tb_name, std::vector<std::string> columns);
```

图 1: DBManager

3.6.2 IndexFile

IndexFile 是一个 B+ 树组织，文件的管理方式和 RecordFile 比较像。

3.6.3 IndexPage

IndexPage 包含 LeafPage 和 InternalPage 两种，页面管理方式和 RecordPage 类似。

3.6.4 IndexSchema

IndexSchema 的作用和 RecordSchema 类似。

3.7 frontend

frontend 对应/src 下的 frontend.cpp，负责读用户的指令并且转交给 parser。

4 主要接口说明

由于前面的介绍比较详细，我直接贴出了主要接口的代码截图。

```cpp
/* data */
int insert_data(const std::vector<FieldData> & data);
void insert_batch_data(const std::vector<std::vector<std::string>> & data);
void select(bool , const std::vector<std::string> &  cols, const std::vector<Constrain> & cons);
void delete_data(const std::vector<Constrain> & cons);
void update_data(const std::vector<std::string> & columns, const std::vector<FieldData> & new_data , const std::vector<Constrain> cons);

/* index */
void add_index(std::string index_name, std::vector<std::string> column_names);
void drop_index(std::string index_name);

/* pk */
void drop_pk();
void add_pk(std::vector<std::string> columns);
```

图 2: TableManager

```cpp
/* data */
void insert_data(std::vector<FieldData> data);
std::vector<RID> insert_batch_data(std::vector<std::vector<FieldData>> );
void select(std::vector<int> select_ids, std::vector<int> compare_ids, std::vector<Constrain> cons);
void delete_data(std::vector<int> comp_ids, std::vector<Constrain> cons);
void update_data(std::vector<int> set_ids, std::vector<FieldData> new_data,  std::vector<int> compare_ids, std::vector<Constrain> cons);
```

图 3: RecordFile

```cpp
/* Create Index */
void insert_data(const std::vector<FieldData> new_data, const RID & rid);
void create_index_on(std::string, bool, const Schema & table_schema , const std::vector<std::string> & fields);

/* Meta File */
void write(const std::string metadir);
void read(const std::string metadir);

/* Open and Close */
void open();
void close();

void drop_index(std::string index_name);
```

图 4: IndexManager

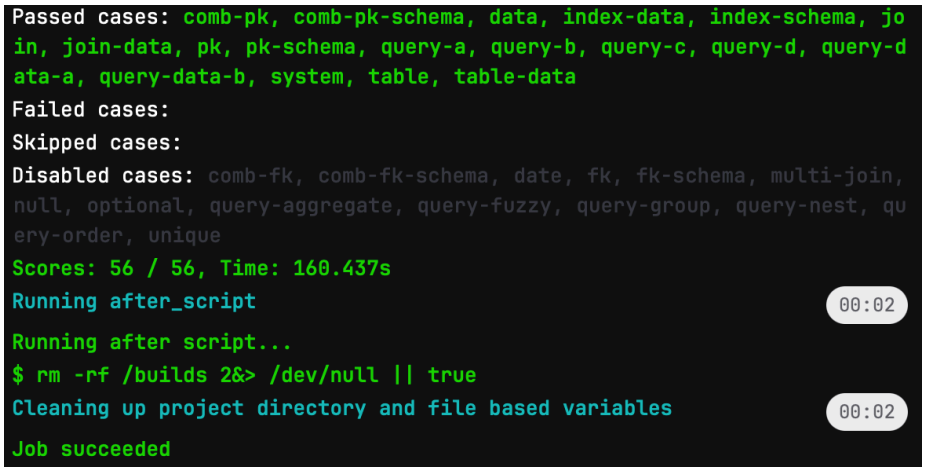

图 5: Result

5 实验结果

除了 foreign key 的约束，我实现了全部的基础功能。CI 运行的结果如 5。

6 小组分工

本组是单人队，所有工作都由我单人完成。

7 参考文献

- 使用了 antlr4 生成文法 parser。生成代码在/src/parser 中，同时为了在 ci 上运行还包含了对应的 runtime 代码，在/linux_runtime 中。
- 页式文件系统使用了课程文档提供的 filesystem，在/src/filesystem 中。
- frontend 里面 parser 的使用参考了课程文档的教学。
- 其他内容都由本人自己从 0 编写。