

《数据库系统概论》课程项目 实验报告

蒋昊迪 2020012328

一、整体设计

1. 技术选型

项目采用 Rust 语言进行编写，并使用了一些 crates.io 中的第三方依赖库。

2. 模块设计

模块设计基本上参照实验指导书，分为文件管理模块、记录管理模块、索引管理模块、SQL 解析模块和查询处理模块。

与实验指导书不同的是，系统管理模块在代码实现中并没有单独作为一个模块，这是因为系统数据在存储上采用了和普通数据相类似的模式，同时对修改查询语句的支持在功能上也比较类似于查询处理模块。因此系统管理模块的内容被拆分合并进了记录管理模块和查询处理模块。

在本实验报告中，为了更好的说明系统管理模块的功能和实现，依然将其作为单独的模块中进行叙述。

3. 进度管理

在实验进度上，我基本将开发任务分配到了整个学期。同时，我并没有严格地按照模块进行开发，而是在完成文件管理和记录管理的基本设计后，同时实现了前端的基本框架。这样在之后的开发中可以以 SQL 语句为导向进行开发，从而能够及时发现错误，及时调整设计。

4. 实现功能

我完成了基本功能中所要求的所有功能，另外还包括附加功能中的如下功能：

- SELECT 中的 LIKE，聚合检索，LIMIT，OFFSET 以及嵌套查询
- 联合主外键
- 多列索引

二、具体设计

1. 文件管理模块

代码实现位于 `filesystem/` 目录下。

由于采用 Rust 进行开发，因此无法使用已经提供的文件管理模块。因此，我参考 C++ 文件管理模块实现了 Rust 版本的文件管理模块，因此绝大多数功能（如缓存分配和管理）和 C++ 版本类似。但是我也进行了些许调整：

1. 为了方便实现，不会根据文件名来判定是否打开相同的文件，转交由上层模块来保证不会重复打开相同文件。相应地也调整了文件关闭时对缓存处理的逻辑；
2. 调整了获取缓存页的接口，区分只读和读写，这样可以更好地利用 Rust 的安全特性，同时实现更为精准的写回策略。

`FileManager` 用于文件管理，`BufPageManager` 为实际的对外接口，主要负责页的读写以及缓存的管理。`PageBuffer` 储存一页数据，实现了整数（`i32/u32` 类型）、浮点数（`f32`）及字符串的二进制读写接口，在后续的模块设计中也基于这些接口实现了一条记录、一个 B+ 树结点的读写。

2. 记录管理模块

这一模块主要实现了记录的储存。代码位于 `record/` 目录下。

实际数据采用页式存储，每一页大小为 8 KiB，其中前 32 B 作为表头，记录总页数、每页元素数等信息。之后为一 Bitmap 记录每一个 slot 是否被使用，数据采用定长类型进行存储。可

以根据表的元数据计算出每一项所需的空間（包括记录是否有效的 Bitmap 和数据本身）计算出每一页可以储存的数据。

在文件存储上，所有数据存于同一目录下，每一张表一个文件，我采用的命名方式为 `data.{database}.{table}.table`，这样可以保证不发生重复。

`RecordManager` 实现了对单张表记录读写的管理。提供了数据插入、删除、查找、修改的接口。而在此基础上，我还基于 Rust 迭代器的特性，封装了对一张表数据遍历的迭代器。这在后续查询管理模块会详细介绍。

3. 索引管理模块

索引模块代码位于 `index/` 目录下。

我实现了 B+ 树的索引模块设计。页的设计基本和普通记录保持一致，前 32 Bytes 保存元信息，包括总页数、结点前驱、后继、本页元素数量等信息，另外一部分信息（如索引列数量）由外部传入。读取时，一次性读入一页所有元素，并将内容写入结构体中。在发生修改后，重新写回对应位置。

B+ 树的实现基本按照其设计实现，实现了插入、删除以及查找功能。由于每一页中项数不会太多，因此内部采用顺序查找。单个元素，为 (Key, Value) 的形式，其中 Key 为索引项，支持由多列组成索引，进行顺序存储，Value 为被索引元素在记录中的位置（可以表示为 (页号, 槽号) 的二元组），每次查询到该元素后需要再在记录中获取其余列的信息。

主键和普通索引的命名方式为 `pki.{database}.{table}.table` 和 `idx.{database}.{table}.{id}.table`。

另外，为了和普通的遍历统一接口，也实现了基于索引的迭代器。在使用索引查询时，需要给出查询的上下界，首先定位到下界所在的叶节点，然后逐个遍历，直至超过上节点范围。

为了验证 B+ 树实现的正确性，我在单元测试中测试了 1000000 条记录的增删，并定时遍历整棵树确认所有结构的正确性，并在这一大规模测试中发现了不少 bug。

4. 查询管理模块

代码位于 `query/` 目录下。

查询模块是较为核心的模块，下面分非 SELECT 语句和 SELECT 语句进行介绍。

(1) 增删改操作

对于增删改操作，主要需要考虑的便是操作的合法性问题。

对于 INSERT 操作，在检查完数据的合法性（包括数据类型、主外键约束）之后，需要向记录以及所有的索引项中添加相应的项。在发生错误时，不进行回滚，但停止语句的执行。

对于 UPDATE 和 DELETE 操作，首先解析 WHERE 语句，检查合法性并筛选出符合条件的项。之后检查主外键约束——这里包含三个部分，自身主键约束，自身外键约束，和其它表的外键约束。由于这里有可能一次性操作多行数据，因此需要对操作的原子性进行一定的处理，也即先进行数据检查，再操作数据。对于 UPDATE 操作还需要关注修改数据自身相互之间对完整性约束的影响。

对于主外键约束检查以及 WHERE 语句的筛选，我也在绝大多数能够使用索引的地方应用索引加速。

(2) SELECT 语句

SELECT 语句的结构变化为所有语句中最为丰富的。因此，我为 SELECT 语句设计了查询树结构。SELECT 语句读入后会首先生成抽象语法树，接着在对抽象语法树进行遍历的过程中解析出查询树的各部分结构，并组装成查询树结构。之后会对查询树进行两次遍历：第一遍遍历检查语句合法性，如列是否存在及数据类型是否匹配；而第二遍遍历则开始实际进行查询过程。

在查询过程中，如前面所说，我使用流水线策略，借助于 Rust 中提供的迭代器 (Iterator trait) 来实现数据的传送，迭代器的每一项即代表一条结果。具体来说，在第二遍遍历的查询过程中，我们会先访问结点的孩子得到迭代器（以及对应的数据列等其它信息），对得到的结果进

行处理得到新的迭代器，并返回至上层。在根处我们会收集得到的数据并返回至前端。

根据已经实现的 SELECT 语句类型，我实现了如下几种查询树结点（定义位于 `query/types.rs` 文件）：

表选择结点 SelectTable 该结点确定查询的表，参数只有表名。该结点会调用记录管理模块或索引管理模块获得对应的遍历迭代器并返回给上层。

在判定何时调用索引时，该结点会利用由父亲结点传下来的 WHERE 语句信息，并与存在的索引信息进行比较，如果满足索引条件则采用对应的索引遍历迭代器（并提供对应的上下界信息）¹。

列选择结点 SelectColumn 列选择结点用于选择输出哪些列的信息，同时完成聚合查询的功能。

在第一遍遍历时，需要确认该结点的所有列均在查询结果中存在且无二义。在第二遍遍历时，则需要根据筛选列的信息从每一条结果中截取对应的列进行返回（这可以采用 Iterator trait 的 `map` 方法实现）。

对于聚合查询来说，除了截取对应的列之外，还需要对结果进行聚合（使用 `fold` 方法实现），因此对于每一个聚合选项（SUM, MAX 等）都要确定初始值，聚合函数等信息，并在此基础上实现统一的聚合形式。

条件选择结点 FilterRow 条件选择结点即 WHERE 子句中的内容，完成对行的筛选。对于一个筛选条件，可以较为容易地使用 Iterator 的 `filter` 方法实现。但是不同的 WHERE 子句在实现时也有各自需要注意的细节：

对于必做项目中的 `col op col` 和 `col op val`，需要在第一遍遍历时检查列的存在性以及左右两侧类型是否匹配。而在构造 `filter` 函数时只需要将对应的列进行比较即可。

对于模糊查询 `col LIKE pattern`，需要检查对应的列是否为 VARCHAR 类型。而由于模糊匹配的语义本身较为简单，因此实现上没有引用第三方库，而是手动构建自动机并进行匹配（代码位于 `util/regex.rs`）。

对于嵌套查询，只需要将子查询的内容构建成一棵子树并递归查询，检查运行结果是否只有一列（这可以在第一遍遍历时检查），对于 `op` 型查询，还需要检查运行结果是否只有一行（这只能在第二遍遍历时检查）。

表连接结点 JoinTable 表连接结点处理二表连接的情况。²对于一般性的二表连接，我采用嵌套循环的方式进行连接。

而对于实验中要求的 `A.x=B.y` 型连接，我则采用了排序归并连接的方式。需要注意的是，实验指导书上给出的伪代码并没有包含连接列中有重复元素的情况，因此为了保证正确性，需要对这一情况进行特殊处理。

输出限制结点 LimitOffset 由于 Iterator 中包含与 LIMIT 和 OFFSET 等效的 `take` 和 `skip` 方法，因此该结点的实现较为简单。

目前对于查询树的生成没有实现任何优化，结点的生成顺序按照（从根向下）`LimitOffset`, `SelectColumn`, `FilterRow`, `JoinTable`, `SelectTable` 的顺序进行组合。

¹索引遍历时的上下界都是取等的，这样可以简化实现，同时对于不取等的情形也会在后续筛去而不影响正确性

²在理论上可以进行多表连接，但是并没有对此进行优化，因此运行效率无法满足需求。

5. 系统管理模块

系统管理主要为元数据的存储及修改查询，代码位于 `record/` 和 `query/` 目录下。

(1) 元数据储存

对于元数据我采用了和普通数据一样的数据管理模式，即保存于若干数据表中。例如，列信息会以下面的形式保存³：

```
1 CREATE TABLE COLS(  
2     DB_NAME VARCHAR(32), -- 数据库名  
3     TB_NAME VARCHAR(32), -- 数据表名  
4     COL_NAME VARCHAR(32), -- 列名  
5     PROPERTY INT, -- 主键，数据类型等信息  
6     ORDER INT); -- 排序信息
```

外键信息和索引信息会以下面的形式存储：

```
1 CREATE TABLE FK(  
2     DB_NAME VARCHAR(32), -- 数据库名  
3     FK_NAME VARCHAR(32), -- 外键名  
4     TB_FROM_NAME VARCHAR(32), -- 引用数据表名  
5     TB_TO_NAME VARCHAR(32) -- 被引用数据表名  
6     COL_FROM_NAME VARCHAR(32), -- 引用数据列  
7     COL_TO_NAME VARCHAR(32) -- 被引用数据列  
8     ID INT); -- 排序信息  
9  
10 CREATE TABLE IDX(  
11     DB_NAME VARCHAR(32), -- 数据库名  
12     TB_NAME VARCHAR(32), -- 数据表名  
13     IDX INT -- 自增 ID  
14     COL_NAME VARCHAR(32), -- 列名  
15     PROPERTY INT, -- 主键，数据类型等信息  
16     ORDER INT); -- 排序信息
```

这些信息统一保存在 `global.*.table` 中。

(2) 元数据的修改

由于数据和元数据采用统一表示，对于元数据的管理的基本逻辑和普通数据类似，但是边界条件相对较多。

对于数据库的增删，需要考虑重名、不存在等情况。在删除时需要一并删除相应的数据。

对于数据表的增删，同样需要考虑重名、不存在等情况，另外对于主外键还需要检查对应列是否存在及是否合法。

对于索引的增删，需要考虑重复和列不存在等情况。在添加索引的时候，我采用将记录逐条插入的方式。

主外键的增删相对较为复杂，此时需要考虑多个表之间的主外键关系以及已有数据的检查，包括插入主键时的重复性检查，删除主键时的外键检查，以及删除外键时的主键存在性检查等。由于此时涉及的数据相对较多，因此需要使用索引进行加速。

³在形式上，数据库名和表名可以形成外键，但是这里依然还是在代码中手动管理这些关系

(3) 元数据的使用

在读取元数据后，我们会将其保存于相应的结构体 `TableMeta` 中，用于后续对数据的读取和处理中。

6. SQL 解析模块与前端设计

前端的设计比较简单，输入部分展示当前数据库名称，并实现了简单的跨行输入（以 ; 为分界符）。在输出部分，使用 `comfy-table` 库对表格结果进行美化，同时输出行数以及运行时间。代码位于 `manager.rs`。

对于输入的 SQL 解析，`antlr` 官方并没有支持对 Rust 的文法解析生成器支持，但是有第三方实现⁴。自动生成的代码位于 `parser/` 目录下。

在完成文法解析并生成 AST 之后，再通过一遍扫描解析出每一条 SQL 语句的类型，并将类型和相应的参数传递到对应的处理模块中进行进一步处理。其中对于绝大多数的指令参数均为几个数据或线性列表，但是 `SELECT` 语句相对较为复杂，我在这一阶段便将它组成为上面所提到的查询树。

三、接口设计

1. 文件管理模块

文件管理模块使用 `BufPageManager` (`filesystem/buf_page_manager.rs`) 对外提供服务，主要接口如下：

```
1  /// 新建对象
2  pub fn create(c: usize, r: T) -> Self
3  /// 打开文件
4  pub fn open(&mut self, path: &str) -> Result<usize, Error>
5  /// 关闭文件
6  pub fn close(&mut self, fd: usize) -> Result<(), Error>
7  /// 获取只读页
8  pub fn fetch_page_for_read(
9      &mut self,
10     fd: usize,
11     page: u64
12 ) -> Result<&PageBuffer, Error>
13 /// 获取读写页
14 pub fn fetch_page_for_write(
15     &mut self,
16     fd: usize,
17     page: u64
18 ) -> Result<&mut PageBuffer, Error>
```

2. 记录管理模块

记录管理模块使用 `RecordManager` (`record/table.rs`) 对外提供服务，主要接口如下：

```
1  /// 新建对象
2  pub fn new(
3      path: &str,
4      buf: &Rc<RefCell<BufPageManager<LRUReplacer>>>>,
```

⁴`antlr-rust` 库

```
5     meta: &'a TableMeta
6 ) -> Result<RecordManager<'a>, Box<dyn Error>>
7 /// 设置默认值
8 pub fn set_default_value(
9     &mut self,
10    data: &DataEntry
11 ) -> Result<(), Box<dyn Error>>
12 /// 插入元素
13 pub fn insert(&mut self, data: &DataEntry) -> Result<Pos, Box<dyn Error>>
14 /// 按位置查找元素
15 pub fn find(&self, pos: &Pos) -> Result<Option<DataEntry>, Box<dyn Error>>
16 /// 更新元素
17 pub fn update(&self, pos: &Pos, data: &DataEntry) -> Result<(), Box<dyn
    Error>>
18 /// 删除元素
19 pub fn delete(&self, pos: &Pos) -> Result<(), Box<dyn Error>>
20 pub fn delete_where<T>(&self, f: T) -> Result<i32, Box<dyn Error>>where
21     T: Fn(DataEntry) -> bool,
22 /// 生成迭代器
23 pub fn iter(&self) -> RecordIterator<'_>
```

另外, RecordIterator 和 RecordIntoIterator 为基于 RecordManager 实现的迭代器, 二者的差别在于生命周期管理方式。二者均实现了 Iterator trait。

元数据记录管理位于 TableMetaManager(record/column.rs), 主要接口如下:

```
1 /// 新建对象
2 pub fn new(
3     path: &str,
4     buf: &Rc<RefCell<BufPageManager<LRUReplacer>>>,
5     database: &str
6 ) -> TableMetaManager
7 /// 读取元数据
8 pub fn read_table(&self, table_name: &str) -> Result<TableMeta, Box<dyn
    Error>>
9 /// 查找引用该表的外键
10 pub fn find_fk_constraint(
11     &self,
12     table_name: &str
13 ) -> Result<Vec<(String, ForeignKey)>, Box<dyn Error>>
14 pub fn list_table(&self) -> Result<Vec<Vec<DataValue>>, Box<dyn Error>>
15 /// 创建数据表
16 pub fn create_table(
17     &self,
18     table_name: &str,
19     meta: &TableMeta
20 ) -> Result<(), Box<dyn Error>>
```

```
21 /// 删除数据表
22 pub fn delete_table(&self, table_name: &str) -> Result<(), Box<dyn Error>>
23 /// 更新数据表
24 pub fn update_table(
25     &self,
26     table_name: &str,
27     meta: &TableMeta
28 ) -> Result<(), Box<dyn Error>>
```

3. 索引管理模块

索引管理模块通过 IndexManager (index/manager.rs) 提供服务，主要接口如下：

```
1 /// 新建对象
2 pub fn new(
3     path: &str,
4     buf: &Rc<RefCell<BufPageManager<LRUReplacer>>>,
5     meta: &TableMeta,
6     idx: &IndexOptions
7 ) -> Result<IndexManager, Box<dyn Error>>
8 /// 插入元素
9 pub fn insert(
10     &mut self,
11     key: &Vec<DataValue>,
12     value: &Pos
13 ) -> Result<(), Box<dyn Error>>
14 /// 删除元素
15 pub fn remove(
16     &mut self,
17     key: &Vec<DataValue>,
18     value: &Pos
19 ) -> Result<bool, Box<dyn Error>>
20 /// 查找元素（仅给出索引值）
21 pub fn find_key(
22     &self,
23     key: &Vec<DataValue>
24 ) -> Result<Option<Pos>, Box<dyn Error>>
25 /// 查找元素（给出一行所有值）
26 pub fn find(&self, key: &Vec<DataValue>) -> Result<Option<Pos>, Box<dyn Error>>
```

另外，IndexIterator 为基于 IndexManager 实现的迭代器，实现了 Iterator trait。

4. 查询解析模块

QueryBuilder (query/builder.rs) 实现了由 AST 树转化为查询指令的功能，接口如下：

```
1 /// 新建对象
2 pub fn new() -> QueryBuilder
```

```
3  /// 执行解析
4  fn visit(
5      &mut self,
6      node: <Self::Node as ParserNodeType<'input>>::Type
7  ) -> Self::Return
```

QueryRunner (query/runner.rs) 实现了指令的执行，接口如下：

```
1  /// 新建对象
2  pub fn new(
3      dbt: &Rc<RefCell<DatabaseTable>>,
4      buf: &Rc<RefCell<BufPageManager<LRUReplacer>>>,
5      path: &str
6  ) -> QueryRunner
7  /// 运行指令
8  pub fn run_cmd(&mut self, query: Query) -> Result<QueryResult, Box<dyn
9      Error>>
10 /// 获取当前运行数据库
11 pub fn get_db(&self) -> Option<String>
```

SelectTree 为查询树的基本数据类型，SelectTreeVisitor 为用于遍历查询树的 trait，接口如下：

```
1  type Return;
2  /// 遍历查询树结点的函数
3  fn visit_select_table(&'a mut self, node: &SelectTable) -> Self::Return;
4  fn visit_select_column(&'a mut self, node: &SelectColumn) -> Self::Return;
5  fn visit_filter_row(&'a mut self, node: &FilterRow) -> Self::Return;
6  fn visit_join_table(&'a mut self, node: &JoinTable) -> Self::Return;
7  fn visit_offset_limit(&'a mut self, node: &OffsetLimit) -> Self::Return;
8  /// 访问子结点
9  fn visit(&'a mut self, node: &SelectTree) -> Self::Return;
```

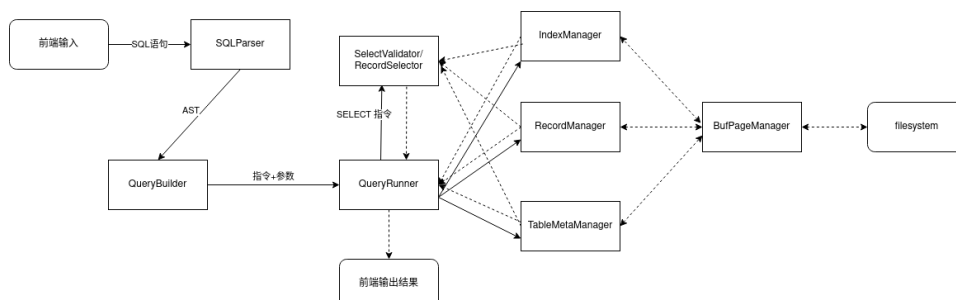
SelectValidator (query/select_validator.rs) 和 RecordSelector (query/record_selector.rs) 均实现了这一 trait。用于实现两遍对查询树的遍历。

5. SQL 解析模块

SQLParser 为 antlr 自动生成的文法解析器，其使用方法见 manager.rs。

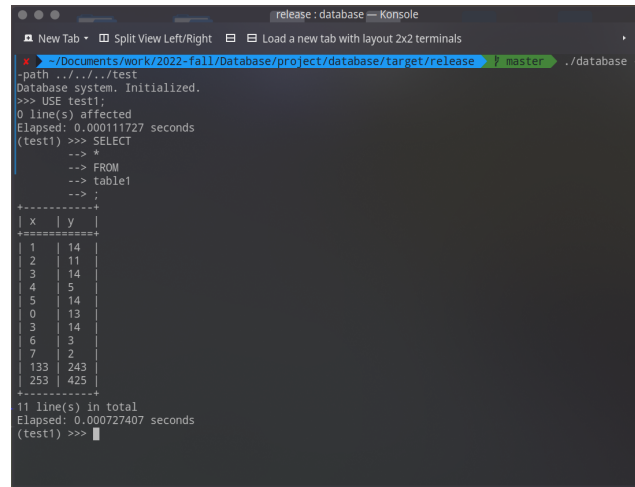
6. 数据通路

数据处理流程如下（实线代表指令，虚线代表数据）：



四、实验结果

实现功能见第一部分。运行截图如下：



```
release : database - Konsole
New Tab  Split View Left/Right  Load a new tab with layout 2x2 terminals
~/Documents/work/2022-fall/Database/project/database/target/release  master  ./database -
-path ~/../././test
Database system. Initialized.
>>> USE test1;
0 line(s) affected
Elapsed: 0.000111727 seconds
(test1) >>> SELECT
--> +
--> FROM
--> table1
--> ;
+-----+
| x | y |
+-----+
| 1 | 14 |
| 2 | 11 |
| 3 | 14 |
| 4 | 5 |
| 5 | 14 |
| 0 | 13 |
| 3 | 14 |
| 6 | 3 |
| 7 | 2 |
| 133 | 243 |
| 253 | 425 |
+-----+
11 line(s) in total
Elapsed: 0.000727407 seconds
(test1) >>>
```

五、参考文献

1. 课程组提供实验指导书。
2. Rust 标准库相关文档。<https://doc.rust-lang.org/std/index.html>