数据库大作业报告

李晨昊 2017011466

2020-1-12

目录

1 简介

本项目使用 rust 实现。执行 `cargo run --bin db --release` 运行数据库 repl，执行 `cargo test -p tests --release` 进行测试，执行 `make` 进行代码覆盖率测试。要求 nightly 版本的 rust 编译器，版本越新越好；为了执行代码覆盖率测试，需先安装 `cargo-tarpaulin`(安装方法为 `cargo install cargo-tarpaulin`) 和 `pycobertura`(安装方法为 `pip install pycobertura`)，并且装有 makefile 中指定的浏览器。

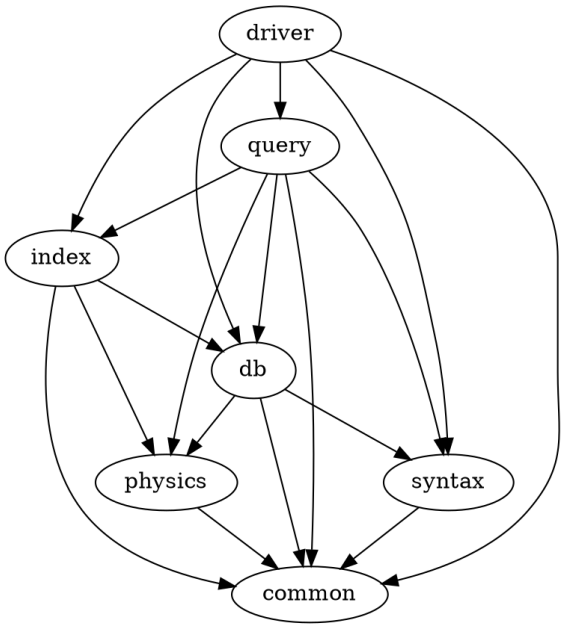
截至目前为止，rust 代码总行数为 3580。

实现了以下的额外功能：

- 简单的查询优化
- 多表连接：理论上支持任意多表的连接 (只要性能和空间允许)
- 聚集查询：支持 `avg`，`sum`，`max`，`min`，`count` 关键字，对 `select` 的结果进行聚集
- 模糊查询：支持 `like` 关键字，包括通配符`%` 和 `_`，支持转义字符
- 日期数据类型
- `unique` 约束，`check` 约束
- `insert` 支持指定要插入数据的列，`update` 的 `set` 子句支持复杂的表达式
- 简单的命令行着色

2 系统架构设计

整个项目按照 rust 的 crate 为边界划分逻辑，crate 间的依赖关系为：



自底向上。每个模块的功能为：

- `common`：提供一些公用的结构，例如错误处理的机制
- `physics`：定义数据库各种页的结构
- `syntax`：解析 sql 语句，基于**我写的** parser generator lalr1
- `db`：定义数据库的核心接口，以及实现一些不需要用到索引的修改操作
- `index`：实现基于 B+ 树的索引，以及实现一些需要用到索引的修改操作
- `query`：实现四种查询解析
- `driver`：顶层接口和可执行文件

3 各模块详细设计

3.1 记录管理部分

我没有使用提供的页式文件系统的代码，也没有自己实现页式文件系统相关的逻辑 (如 lru 缓存，脏页写回等)。我的数据库与硬盘的交互完全依赖于 os 提供的内存映射 (mmap)。为此我使用了memmap-rs库，虽然它是跨平台的，不过不同平台上的内存映射的实现策略并不完全一样，我目前只在 linux 平台上测试过，因此我也不能保证在其它平台上能正常工作。

本质上来说直接依赖于 os 的 mmap 一定是比依赖于 libc 提供的 read 和 write 更高效的实现方式，因为 libc 的 read 和 write 一定依赖于 os 提供的 read 和 write，而 os 内部对于文件的读写也有相关的缓存。直接使用 mmap 的话去掉了这些中间层。更重要的是，当一个磁盘页已经在内存中时对它的读写完全是零额外开销的，不需要像 libc 的 read 一样至少要进行一次内存复制，脏页的写回也可以尽量延后到 os 认为需要写回为止，这就让我的实现灵活了很多。

这样实现的一个缺点在于没有精准的控制写回，所以没有办法实现事务管理和错误恢复等高级功能。不过这也是相对而言的，因为 libc 提供的 read 和 write 也没有相关的保障 (例如，libc 的 write 有自己内部的缓存，而且即使强制要求不使用 libc 的缓存，os 内部也有缓存，总之不能保证写回操作一定被执行了)，所以只是说我的实现没有做到最好，不是说这个实现在这一点上不如使用了提供的代码的实现。

数据库由两个文件组成，一个用来存储所有元信息和所有定长字段，另一个用来存储所有变长字段 (即 `varchar` 字段)。前一个中所有存储都是按页对齐的，而后一个中则完全没有页的概念，直接像访问普通内存一样访问内存映射的文件。

每种页的物理布局在 `physics` crate 中描述。值得注意的是 rid 的实现，它用于在前一个文件中定位字段。使用一个 32 位无符号整数来描述 rid，其中高 23 位表示页号，低 9 位表示页内编号，因此前一个文件的最大容量为 $2^{23} * 8192\text{B} = 64\text{GB}$，每个数据页最多有 $2^9 = 512$ 个数据项。定长数据在存储的时候，前若干字组成一个位数组，记录每一项是否为空。每个数据

字段的偏移量都设计成让它满足这种数据类型的对齐要求 (如一个 `int` 字段一定按 4 字节对齐)。

后一个文件中因为没有页的概念，所以直接用一个 32 位无符号整数来进行寻址即可，寻址的粒度 (即变长字段空间分配的粒度) 是 32B，所以后一个文件的最大容量为 $2^{32} * 32\text{B} = 128\text{GB}$。空间的分配采用简单的 first-fit 内存分配算法 (虽然这里分配的不是内存，但逻辑是一样的)。因为实现的复杂性，`test` crate 中专门设计了对变长字段空间分配的测试，用很大规模的输入数据测试了空间分配的正确性。

3.2 索引管理部分

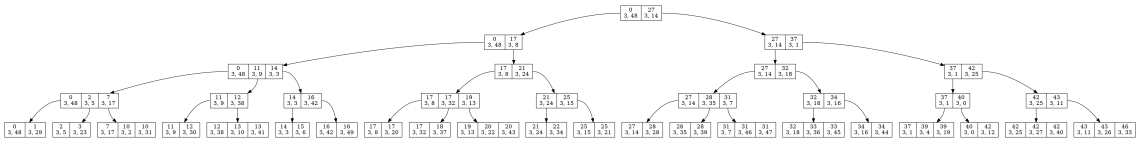
索引是通过 B+ 树来实现的。目前实现的 B+ 树的键的类型是 (*被索引的关键字的值*，*被索引的关键字的* `Rid`) 的二元组，而**没有值类型**，即相当于一个 (不允许重复的)set。这样其实丧失了 B+ 树的一个优点，即在非叶节点中可以不存储值从而增加节点的容量。不过其它优点还是得到保留了的，如所有值都在叶节点中出现，并且叶节点用链表串起来，便于范围查询。

B+ 树的实现没有什么可以仔细解释的地方，就是完整的实现了所有相关的逻辑，包括插入时节点的分裂和删除时节点的合并。此外对于树中的搜索还做了一点优化，在一个节点内部搜索时可以使用二分查找，由于每个页内的表项数目往往是较大的，因此二分查找的确有一定的加速效果。

B+ 树的索引不支持 `varchar` 类型，而支持其它所有定长类型。这样的多态性是利用 rust 的常量泛型参数和宏实现的 (c++ 中有完全类似的两个概念)，具体实现可以参考代码。值得注意的是，rust 的常量泛型参数这个特性目前尚未稳定，因此需要较新版本的编译器才能支持，编译器的版本问题我已经在上一次报告中说过了。

因为 B+ 树实现的复杂性，`test` crate 中专门设计了对索引模块的测试，用很大规模的输入数据测试了索引的正确性。

我使用 `graphviz` 绘制了一张 B+ 树的图形 (见 `index` crate 中的 `print_dot` 函数)，这里存储的元素类型是整数，并且手动设定了一个很小的分支因子：



3.3 系统管理部分

如之前所说，我实现的数据库基本上是单文件的，只是额外分出一个文件来存储变长字段。在这样的实现下，创建数据库和删除数据库即是创建文件和删除文件；创建表和删除表不涉及文件系统操作，与各种查询操作的实现没有本质区别。

主键和外键约束的实现很大程度上依赖于索引，所以我设定相关的数据约束都不能作用于 `varchar` 类型，因为我的索引实现不支持它。单一主键的唯一性我采用查询索引来实现，复合主键的唯一性我采用散列表来实现，这个散列表是在修改数据时临时构造的，所以可能会有较大的开销。外键要求一定要引用一个保证了唯一性的字段，包括标注 `unique` 的字段和单一主键。**不支持命名外键，不支持复合外键**。

alter 表操作我实现了删除一个列和在末尾添加一个列。一般来说这是一个相当昂贵的操作，我的实现中，需要先把所有的新格式的数据都插入数据库，然后再删除全部的旧数据，然后更新全部索引。

3.4 查询解析部分

我实现的 sql 语法与 `parser` 文法规则文档中要求的基本一致，但也有一定的区别，支持的语法略少一些，例如不支持命名外键，不支持复合外键等。**parser 层面支持的语法，底层的实现也支持，反之则不支持**。

因为我的lalr1并不支持大小写不敏感的选项 (准确来说这应该算 lexer 的功能，lexer 也是我写的，re2dfa)，所以所有 sql 关键字都需要手动设定大小写均接受，写的有点难受。我实在是不能理解为什么 sql 要设计成大小写不敏感的。

parser 得到的字符串还不能直接输入给数据库，需要考虑转义字符。一般的字符串中 sql 不像其他语言一样使用\来转义，而是用''来表示无法输入的'，且不支持任何其他的转义字符。作为 `like` 的参数的字符串用_，\% 和\\来表示字符 _，% 和\。

`query` crate 中实现了 select，insert，delete，update 四种查询。查询的输入信息是 ast 节点。下面简单讲一下主要的技术点。

各种查询 (`insert` 除外) 都用到的一个基础设施是单个表的过滤。这一步我会尝试使用索引来优化：对于列和常数的比较，可以直接通过遍历 B+ 树的特定区间来代替。此外在单个表的过滤中我支持了模糊查询，底层是使用正则表达式来实现的。

`select` 中，首先收集所有只涉及单个表的判断条件，使用它们对各自的表先过滤一遍，得到表的个数个结果列表。然后使用跨两个表的判断条件来进行两个列表的组合，相比于暴力的二重循环 + 判断，我做了一点优化，如果一个条件比较的左右项类型相同，而且比较条件不是`!=`，则可以使用排序 + 二分搜索来进行组合。得到完整的结果之后可以进行聚集操作。经测试对于多个表的连接仍然能有不错的性能。

`insert` 中，首先结合指定要插入数据的列 (如果有给出的话)，列的默认值和插入值给出实际插入的数据，然后申请一个临时的缓冲区，将所数据先写入这个缓冲区，同时进行完整性检查，这主要包括 `notnull` 约束的检查，唯一性约束的检查 (包括单主键，复合主键，`unique` 约束三种情形) 和 `check` 约束的检查，确认没有任何错误之后，申请新的一行的空间，将缓冲区中的内容写入这一行。为了实现复合主键约束的检查，在开始插入之前先构造一个散列

表，插入的同时查询和维护散列表，保证不出现重复的多主键，其余的唯一性约束的检查可以借助索引进行。

`delete` 中，直接遍历一遍单个表的过滤结果进行删除即可。删除的同时需要删除对应的索引，为了避免在 B+ 树上进行迭代的同时修改树，`delete` 不会采用索引优化。执行删除的时候可能破坏参照完整性，默认使用 restrict 策略，即如果发现当前被删除的列中包含被外键引用的字段，则拒绝删除并报错。

`update` 中，结合了 `insert` 的完整性检查的逻辑 (复用了 `insert` 中的代码) 与 `delete` 类似的参照完整性检查之外。除此之外，在 `set` 子句中我实现了比其他查询的 `where` 子句更加复杂的表达式的运算，包括基本的四则运算，逻辑运算 (支持短路)，`like` 等，以及它们的任意组合。

parser generator 生成的 lexer 和 parser 可以独立使用，得益于这种封装，我在 `driver` crate 中使用这个 lexer 为命令行的 repl 实现了一个非常简易的语法高亮，最后的结果还算比较漂亮：

```
→  db git:(master) ✗ cargo run --bin db
    Finished dev [unoptimized + debuginfo] target(s) in 0.04s
     Running `target/debug/db`
Database repl by MashPlant. Enter sql statement separated by semicolon.
>> CREATE TABLE PART (
..   P_PARTKEY INT,
..   P_NAME VARCHAR(55),
..   P_MFGR CHAR(25),
..   P_BRAND CHAR(10),
..   P_TYPE VARCHAR(25),
..   P_SIZE INT,
..   P_CONTAINER CHAR(10),
..   P_RETAILPRICE DECIMAL,
..   P_COMMENT VARCHAR(23),
..   PRIMARY KEY(P_PARTKEY)
.. );
```

当然，也没有做什么太复杂的处理，所以也不会有什么其他神奇的功能。对于这个着色的兼容性也不做保证，只能说在一般的 linux 控制台里应该是能正常显示的。

4 主要接口说明

4.1 记录管理部分

主要是 `physics` crate 中的一些 struct：

- `DbPage`：数据库的第 0 页，记录数据库的元信息，基本上是一个 `TablePage` 的位置的数组
- `TablePage`：记录一张表的元信息，基本上是一个 `ColInfo` 的数组
- `ColInfo`：记录一列的元信息

- `ColFlags`：用位域紧凑地存储一列的约束信息
- `IndexPage`：索引页的内容
- `CheckPage`：存储 `check` 约束的内容
- `DataPage`：定长数据页的内容
- `FreeLobSlot`：变长数据文件中用于内存分配，组成空闲链表
- `VarcharSlot`：定长数据 (8 字节)，指向变长数据文件中的一个位置
- `Rid`：指向定长数据文件中的一个位置

4.2 索引管理部分

`index` crate 中：

- `Index` struct：供上层使用的索引接口，使用底层的 `IndexPage` 实现增删查的功能
- `handle_all!` macro：便于处理不同的数据类型的索引
- `alter` module：实现一些需要用到索引的修改操作，包括以下函数：
  - `add_col`：增加一列
  - `drop_col`：删除一列
  - `add_foreign`：增加 foreign 约束
  - `add_primary`：增加 primary 约束
  - `drop_primary`：删除 primary 约束
  - `create_index`：为一列创建索引

4.3 系统管理部分

主要是 `Db` struct，使用底层的 `DbPage` 等维护数据库的元信息，包括以下函数 (有一定省略)，其中一部分是对外的接口 (以 ast 节点作为输入)，一部分是对内的接口 (以页的引用等作为输入)：

- `create`：创建数据库 (constructor)
- `open`：打开数据库 (constructor)
- `lit2ptr`：将值写入指定内存
- `data2lit`：将内存中的原始数据读出，得到值
- `get_page`：获取页的引用
- `alloc_page`：申请页
- `dealloc_page`：释放页
- `get_data_slot`：获取定长数据槽的引用
- `alloc_data_slot`：申请定长数据槽
- `dealloc_data_slot`：释放定长数据槽
- `get_lob`：获取变长字段的引用

- `alloc_lob`：申请变长字段
- `dealloc_lob`：释放变长字段
- `create_table`：创建表
- `foreign_links_to`：获取所有指向一张表中的字段的外键
- `get_tp`：按名字搜索表，获取表的引用
- `record_iter`：获取一张表的迭代器，用于迭代它的所有记录
- `alloc_index`：申请索引的树根 (但不负责将现有的字段插入索引页)
- `drop_index`：释放索引 (经错误检查后调用 `dealloc_index`)
- `dealloc_index`：释放索引的整棵树
- `drop_foreign`：删除 foreign 约束
- `rename_table`：重命名表
- `drop_table`：释放表 (经错误检查后调用 `drop_list`)
- `drop_list`：释放表的所有内容
- `show_table`：输出表的元信息

4.4 查询解析部分

`parser` crate 中将 sql 输入解析为 ast 节点，后者输入给 `query` crate，其中处理四种查询：`delete`，`insert`，`select`，`update` 各一个 module。其中 `select` 中用 `SelectResult` struct 专门定义了输出格式。

`driver` crate 中把所有组建整合起来。`Eval` struct 表示一个运行环境，它只是一个 `Db` struct 的简单封装，包括以下函数：

- `exec_all`：执行 sql 文本
- `exec`：执行单条 sql 语句 (输入是 ast 节点)，返回字符串或者错误信息
- `select`：执行单条 select 语句 (输入是 ast 节点)，返回 `SelectResult` 或者错误信息

`cli.rs` 中定义主函数，`main.rs` 用于我开发时的测试。

5 实验结果

因为不支持复合外键，所以不能直接运行给出的测试样例。不过这也是唯一一个不支持的地方，去掉对应的约束语句之后即可正常执行，**读入小数据集的全部数据耗时约 0.7 秒**。

代码覆盖率测试结果如下，可以看出基本上做到了全部覆盖，不过还有两点需要说明：

1. 现有的 rust 代码覆盖率测试工具都没有提供分支覆盖率的测试，所以这个测试结果并不能完全证明我的代码的可靠性
2. 因为 `cargo-tarpaulin` 这个工具自己的一些 bug，一些宏定义的代码显示为未覆盖，所以实际上代码覆盖率还应该再高一些

Filename	Stmts	Miss	Cover	Missing
driver/src/lib.rs	39	1	97.44%	35
tests/src/index.rs	30	0	100.00%	
tests/src/integrate.rs	184	0	100.00%	
tests/src/lob.rs	34	1	97.06%	52
query/src/delete.rs	18	0	100.00%	
query/src/filter.rs	19	0	100.00%	
query/src/insert.rs	98	10	89.80%	111, 127-132, 139-141
query/src/lib.rs	7	0	100.00%	
query/src/predicate.rs	75	13	82.67%	26, 46-50, 62-63, 71, 78-79, 81, 103, 106
query/src/select.rs	173	4	97.69%	39, 54, 172, 292
query/src/update.rs	104	8	92.31%	13, 19, 21, 52, 141, 149-158
syntax/src/ast.rs	22	2	90.91%	167, 177
syntax/src/lib.rs	10	0	100.00%	
syntax/src/parser.rs	11	3	72.73%	18-20
index/src/alter.rs	190	3	98.42%	125, 271-272
index/src/cmp.rs	21	2	90.48%	15-16
index/src/iter.rs	43	2	95.35%	50, 57
index/src/lib.rs	127	2	98.43%	52, 169
db/src/alter.rs	63	3	95.24%	33, 42, 55
db/src/db.rs	224	9	95.98%	48, 60, 63, 87, 92, 109-110, 167, 232
db/src/iter.rs	13	0	100.00%	
db/src/lib.rs	31	0	100.00%	
db/src/lob.rs	42	0	100.00%	
db/src/show.rs	43	0	100.00%	
physics/src/data_page.rs	6	0	100.00%	
physics/src/db_page.rs	6	0	100.00%	
physics/src/index_page.rs	8	0	100.00%	
physics/src/rid.rs	5	2	60.00%	16-17
physics/src/table_page.rs	33	0	100.00%	
common/src/errors.rs	5	3	40.00%	90, 95-96
common/src/ty.rs	53	6	88.68%	71, 73-75, 93, 112
common/src/unsafe_helper.rs	17	0	100.00%	
TOTAL	1754	74	94.70%	

6 小组分工

我独立完成了所有的工作。

7 参考文献

基本没有参考什么特别的资料，主要是零散地在网上查询关于 sql 语法和规范的定义。基础架构方面一定程度上参考了CS346 Spring 2015。