目前实现了一个简单的数据库存储引擎，具有的功能

1. 可持久化
2. B+树索引
3. 页级缓存
4. 页的回收复用
5. SQL解析
6. ~~（即将完成）大型记录存储~~
7. （粗糙版）事务并发与日志

持久化的实现

要实现持久化，就是要定制从内存到外存之间的转换协议，我的做法是针对一个表（内存）对应一个文件（外存）

文件是单一的，但内部需要分页，正常人的做法是分页大小为4K字节的倍数，比如 `InnoDB` 采用16K，我选择8192字节，实测相对快一点

每个页都需要分配一个 `page id` ，由于页是固定大小的，很容易在文件中定位，比如 `page_id*8192`

既然分页了，那就需要确定需要怎样、多少的页类型来足以描述整个表，现在只谈表头页（ `head page` ）和记录页（ `record page` ）

表头页

表头页用于描述一个表的信息（其实也没啥信息）

在目前没怎么开始具体设计的时候，很容易构造出一个草稿

```
 ----------------------------------
| MAGIC | FIRST_FREE |  TABLE_INFO |
 ----------------------------------
|←          8192 Bytes            →|
```

这是一个像样的表头页，简单描述以下信息

- `magic` ： 用于校验它是否为 `head page`
- `first free` ： 第一个空闲的 `page id` ，这涉及到页的复用和分配
- `table info` ： 文本形式的表信息，记录各个 `column` 、 `key attribute` 的细节

设定每个表对应的表头页的 `page id` 为 `0` ，可以很方便的在首次访问就能得到表的描述信息以构造内存中所需描述的对象

细节：表头页加载后就单独常驻内存，这样可不拖慢整个系统的性能

记录页

记录页（数据页）则用于描述行记录，单个记录页可描述多个行数据，能描述多少个取决于不同行记录的大小

那么该怎么描述，才能得知这个页能否存入这条记录，这个页怎样才能$O(1)$获取数据?

这里采用分槽页（ `slotted page` ）来解决这两个问题

```
              →-----------------------------------
             ↑                                    ↓
 ----------------------------------------------------·---------------------------------
| MAGIC | END_FREE | PTR_CNT | PTR[0] | PTR[1] | .. | REC[N-1] | REC[N-2] | .. | REC[0] |
 ----------------------------------------------------------------------------·-------
                     = N        ↓                                           ↑
|←          FIXED           →|    →-------------------------------------------
```

这里显得稍微复杂，逐一解释

- `magic`： 校验它是 `record page` 的魔数
- `end free`： 空闲区域（`free space`）的结尾偏移量，总是指向最后的 `ptr` 指向的记录 `rec` 的第一个字节
- `ptr_cnt`： `ptr` 的个数，使得实际存在的 `ptr` 范围为$[0,ptr \ cnt)$
- `ptr[i]`：记录指针，指向某个记录的第一个字节
- `rec[i]`：真正的行记录

可以看出，前三个域的大小是已知固定的，而指针个数等于记录个数，是动态变化的

一开始 `end free` 肯定是偏移量为8192，表示$[ptr[ptr \ cnt],end \ free)$是一段唯一、连续可用的区域，很简单就能算出一条记录能否插入，无需逐一尝试各种碎片段

每次插入记录，就分配出一个新的指针，由于有 `ptr cnt`，可以很容易的知道新的指针 `ptr` 开始分配的地方，而记录 `rec` 则从当前的 `end free` 往前移需要的字节数再填充，完成后 `end free` 和 `ptr` 都指向该记录的首字节（其实 `end free是` 可以省略的，但是方便）

而删除记录则相对麻烦点，比如记录在各个 `rec` 域的中间，删除等价于把（位置在左边的）前面的记录都后移覆盖到当前记录的最后一个字节，修改 `end free` 和改动位置的记录对应的 `ptr`，维持 `free space` 总是中间一段连续区域的性质，以提高空间利用率。虽然逐个偏移记录看着有点慢，但是利用 `memmove` 性能也不差， `ptr` 的修改也不涉及到很多字节数目，整体不亏（虽然还残留有问题，但后面我会指出）

行记录

现在到了设计记录页中 `rec` 内部的阶段，他们有可能是整型，整型可能4字节、8字节，甚至可能是字符串，多长也不知道，因此需要记入足够的信息

简化问题，假定单一的记录不会特别大，8K足以满足所有单一记录的要求，可以这样来设计

```
 ------------------------------------------------------------
| (998244) | (28,7) | (10^12+7) | (35,5) | "kiseki" | "jojo" |
 ----------.--------.-----------.--------.----------.--------
 0          4        12          20       28         35
|←                FIXED                  →|
```

第一个为4字节整型，我们直接记入其数值，第三个长整型同理

而第二、第四个列为可变长的字符串，可以记入 `(offset,length)` 以避免接下来的逐个偏移（PS.这里的 `length` 是算入结束符的）

这样的好处是左半边的大小是固定的，这在 `column` 的信息中已经决定好了，这样可方便记录任意变长类型

至于不同列的类型判断的问题，还是要看表头信息

B+树索引

数据页已经设计好了，那就要设计怎样找到数据页的索引协议，

这里使用了常规的B+树索引，我的实现是非聚簇的，因为实在是太方便了

（聚簇索引想了下也不需要太大改动，就是我认为可能需要两套不同的平衡因子还有数据和键值杂糅在一起的设计容易出bug）

这个就是查询、增加、删除的操作实现（所有操作均基于页）

至于外存上的页面设计， `bplus page` 大概如下所示

```
 ------------------------------------------------------------------
| MAGIC | PARENT | KEY_CNT | CHILD_CNT | --KEYS-- | --CHILDREN-- |
 ------------------------------------------------------------------
|←               FIXED                →|
```

由于前面各段都是固定的，平衡因子 `b` 很容易通过简单的式子在编译时求出最优解

而对于表头页，我们需要加一个小的段来表示它的 `root` 的 `page id`

我在实现上妥协的一点是，对于内部节点，由于采用和叶子节点相同的结构， `children` 中的儿子指针是8字节的 `offset<<32 | page_id` 的形式实现， `offset` 是用于给叶子节点定位到记录页的第 `offset` 个指针，这会带来一定的浪费，因为内部节点只需要 `pageid` 即可，但它实在是太方便了

另外，删除操作真正的实现我觉得很复杂，虽然写了一套，但后来还是改成了 `LAZY` 操作，直接把叶的儿子节点置0，这样不仅方便而且还减少调整树的形态的开销，并且避免了前面记录页的删除操作带来的管理混乱）

页级缓存

其实挺简单的，就是个 `LRU` 的淘汰置换，不考虑全表扫描等致命操作，效率还是可以的，有一个小的优化是可以用数组映射数值较小的 `page_id` ，而少部分交给 `map` 进行额外映射

在具体的实现上，可能比较麻烦的地方标记脏的操作，我的做法是额外分一个类，用 `read_op` 和 `write_op` 来对 `page_id` 寻址（从缓存中获得）和分配页面类，每一次写操作的寻址都标记脏，而读操作则直接分配一个 `const` 限定的页强制限制写操作， `bplus_tree` 不得持有任何一个 `page_id` 而必须通过该类来访问节点，当然还有一些读缓存和写缓存之类的小优化，不细说了

页的回收复用

如果一个页是确实要删除，那么与其让它变成空洞文件，不如回收利用，这里需要表头页来进行配合

一个 `free page` 是极其简单的结构，只有 `magic` 和 `next_page`

我们通过表头的 `first free` ，可以直接拿到要复用的页，并把 `first free` 置为当前复用页的 `next_page`

看到这应该能想出完整方案了，就是一个插头的过程，不细说

另外再聊一个没啥用处的优化，就是对于大部分的页，可以只初始化（覆盖）几个或十几个字节就能完成复用，而不用强行8K都要初始化，看看前面的固定区域都是非常小的，道理挺简单

最后，我在实际过程中还是把这个复用给砍了，因为既然采用 `LAZY` 的策略，那还需要什么回收，可以等到重建过程直接整理碎片即可（然而没写）

SQL解析

这个交给SQLite完成，它提供了 `Virtual Table` 机制，基本上都要看官方文档

剩下需要实现的就是面向接口编程，提供迭代器给SQLite即可

其实还是要考虑如何断定能否使用索引、常规数据类型的转换、隐式rowid的创建等琐碎问题，这里是我写的最磨蹭的地方

事务

目前的事务使用的是简单的一次封锁，虽然比较简单，但我觉得并不比两阶段封锁（2PL）要差

原因：

1. 最坏情况下，2PL和一次封锁复杂度一致（比如一个写操作上来就把根给互斥锁掉了），甚至更差（锁的开销也很大）
2. 繁琐，需要考虑死锁检测/预防，需要相当程度的运行时成本（比如每添加一条边就要判环测试，或者搞个定时器来检测），写起来也麻烦（比如锁了根，但是由于插入操作导致不断向上递归分裂结点，最后根变成了个新分配的，还得接着封锁blabla）
3. 一次封锁下事务对应的日志肯定是连续的，就是两个commit之间的日志肯定是同一个事务，这在后面的恢复工作中带来极大的便利（2PL应该也行，但没这个好使）

至于为啥不写更好的MVCC那肯定是太难实现了。。。

该模型下不需处理事务冲突级别的回滚（写互斥了，也不存在死锁冲突，因此不用abort），只针对崩溃恢复（突然掉电，不正常结束直接kill，其实点个叉干掉shell也算的）进行处理

用到的日志分为三种： `undo` / `redo` / `commit` ，为了方便连 `start` 标记都省了，并且粒度是页

- `undo` ：负责记录撤回用的日志，保证可回滚到该事务任意写操作前的状态
- `redo` ：负责重演，保证 `commit` 后确实执行

- `commit`：表示一个事务正常的结束

日志的记录方式都是统一的：三种日志不管任何事务全部放在同一个 `[表文件名]log` 文件中，每次添加都用 `append` 方式追加到文件末尾，每次追加都是 `[日志页][内容页]` 的形式，通过日志页中记录的事务信息来决定内容页该怎么用

在追加的过程中，如果用u/r/c来简写前面三种日志，可以确定事务肯定是这种形式xxxxxcxxxxxcxxxxc，x表示任意的u或r，描述同一个页的u肯定在r之前，而c与c之间肯定是描述同一事务的日志

维护日志的关键过程

- 什么时候添加 `undo`：第一次写操作前，并强制刷新磁盘
- 什么时候添加 `redo`：最后一次写操作后，commit前
- 什么时候添加 `commit`：事务结束时
- `commit` 后的维护：把事务带来的修改更新到全局的已提交缓冲，新的事务从该缓冲中访问并拷贝为私有的页

如果是只读操作（不存在事务状态），对页的访问则直接访问已提交缓冲，否则访问事务私有页

（这部分显得有点多此一举，因为我本来是想写多版本控制的。。）

恢复过程：

1. 每次崩溃恢复都会在建立新连接时（进程首次访问/创建）执行
2. 顺序遍历整个 `log` 文件，找出所有commit标记，依顺序处理
3. 所有存在commit标记的都执行redo日志操作，抛弃undo日志
4. 不存在commit（也就是后缀部分，xxxcxxxc[xxx崩掉来不及写]的形式）则执行redo操作，抛弃redo日志
5. 因为都是按照顺序的事务来处理的，在单个 `xxxxxc` 过程中无论是 `undo` 还是 `redo` 都可任意顺序执行，因为日志记录的单位是页，也就是同一事务中的日志其实是相互独立的，又因为内容页直接记录的就是页，直接刷新覆盖到原来的表文件对应的位置就可以了

一些细节：

1. 页的大小改为4K，即可满足文件系统写入时的原子性
2. 其实过程需要系统调用 `fsync`，但这个只有unix api才支持，没有一个好的跨平台方法（我的设计用的 `FILE*` 而不是 `fd`），暂时只用了不太严格的 `fflush`，起码用户态的崩溃是没问题的
3. `undo` 必不可缺，但 `commit` 时直接把要更新的 `redo` 页写到磁盘的表文件也不是不行，但这会牺牲磁盘的顺序读，这也是为什么把日志全部放在一个文件并用追加形式的原因

（其实实现上的琐碎事比上面的多得多）

参考

《数据库系统实现》

《数据库系统概念》