MiniSQL 个人详细报告

G15 3200102557 管嘉瑞

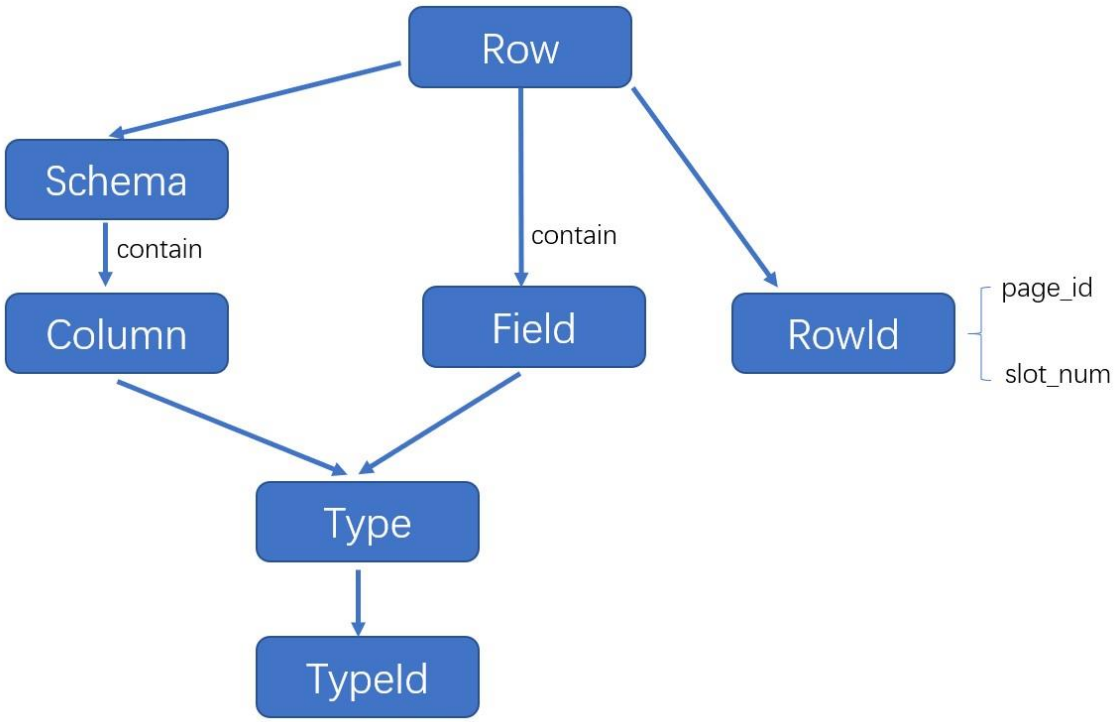
1. 个人分工说明

本人在本次实验中主要负责如下几个部分：buffer pool 模块中 LRU Replacer 的初步实现，Record Manager 模块的实现（包括记录的正反序列化、堆表的实现与优化），Executor 的实现和模块对接，MemHeap 的简单优化（包括 VecMemHeap 和 ListMemHeap），异常处理（包括异常输入处理和强制退出时的数据保存）。此外还进行了一些测试和修复。测试和修复部分不在报告内重复展示，LRU Replacer 已经做过优化在总体报告中已有详细说明。本详细报告中绝大部分内容在总体设计报告中均有体现

2.RecordManager 模块

2.1 整体设计

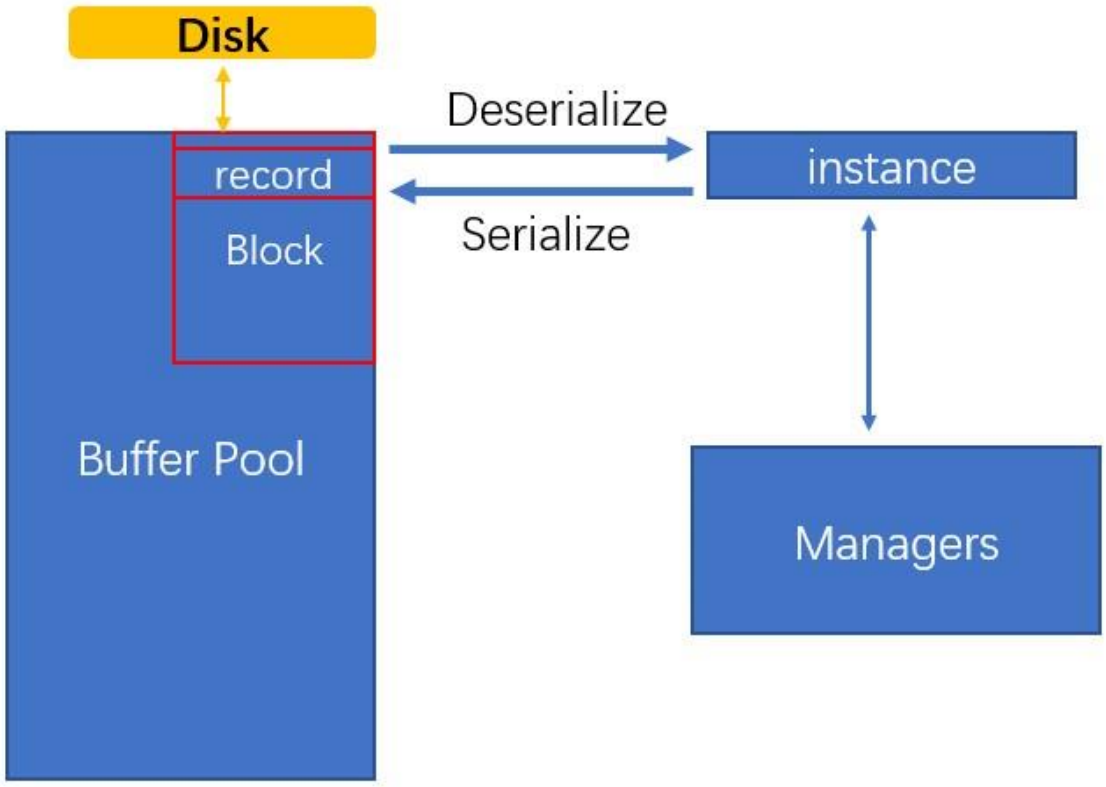
Record Manager 管理的对象是数据表中记录。一条记录 row 由若干个 field 构成，field 是存储数据的单元，由数据类型 (TypeId). 其中 schema 记录表中每一个 row 的结构，由若干个 column 构成。Column 是每一列的元信息，除了记录对应 filed 的类型外，还要记录列名、下标、是否唯一、是否可为 null 等，如果是字符串类型还要记录最大长度。某种程度上来说 field 是一条 row 在一个 column 上的实例化。



这些 record 维护的类都需要进行持久化，给上层提供将其写入及读出 buffer pool 的接口，即序列化与反序列化。

在 MIniSQL 中，所有的记录以堆表（Table Heap）的形式进行组织。堆表是由多个数据页构成的链表，每个数据页中包含一条或多条记录，支持非定长记录的存储。本项目中不考虑单条记录的跨页存储，多个表也不会共享数据页，但是一个表可能占用多个数据页。

3.2 Record 的正反序列化



对于上述提到的 record 对象，我们需要将其序列化成字节流的方法，以写入数据页中。与之相对，为了能够从磁盘中恢复这些对象，我们同样需要能够提供一种反序列化的方法，从数据页的 char* 类型的字节流中反序列化出我们需要的对象。序列化和反序列化操作是将数据库系统中的对象（包括记录，也包括后续的索引、目录等）进行存储格式转化的过程。根本目的是实现数据的持久化，使其有办法读写磁盘。

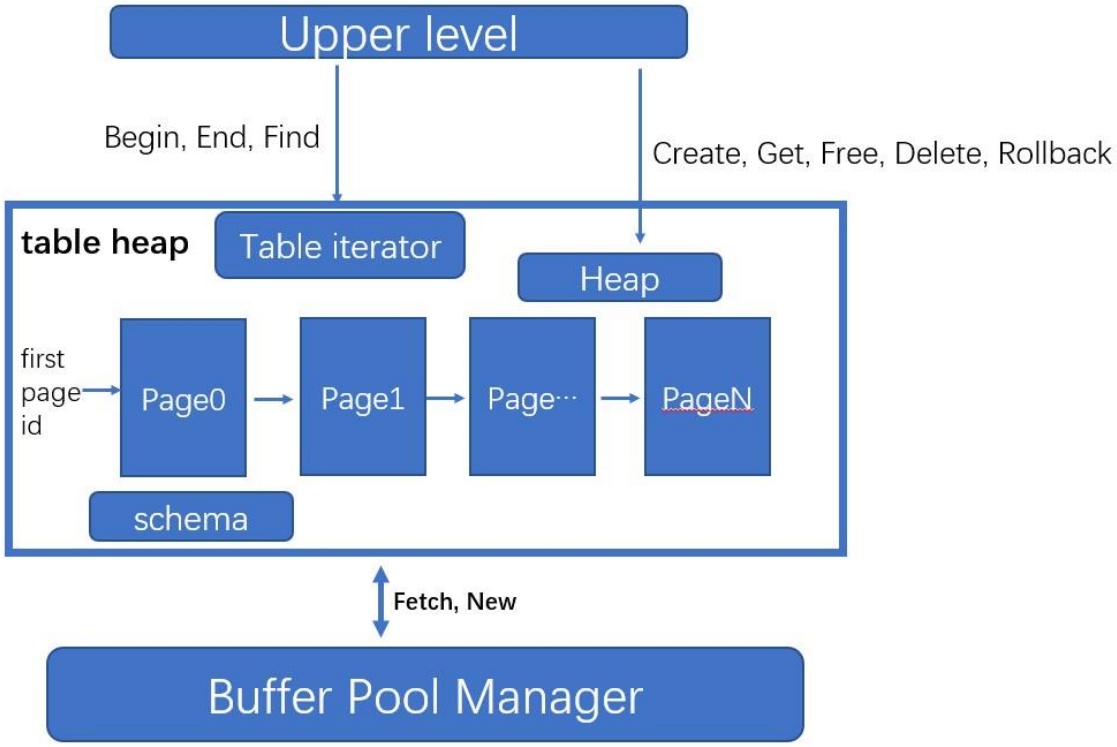
序列化的过程较为简单，只需要构成对象的成员通过提供的 MACH_WRITE 宏写入字节流即可，而反序列化只需要在确定头指针的情况下按照与序列化相同的顺序利用 MACH_READ 宏读出数据并生成对象即可。需要注意的是，当内部成员有重要的元信息时，如字符串的长度，则需要将这类元信息也进行序列化，使得反序列化时有办法知道每步操作的内存范围。

同时，也可以在对象内加入静态成员魔数（MAGIC_NUM），用以在反序列化时验证正确性。

3.3 堆表的实现与优化

堆表（table heap）是 MiniSQL 非常重要的组成部分，是真正存储表内容的数据结构，含有一个表的全部数据。

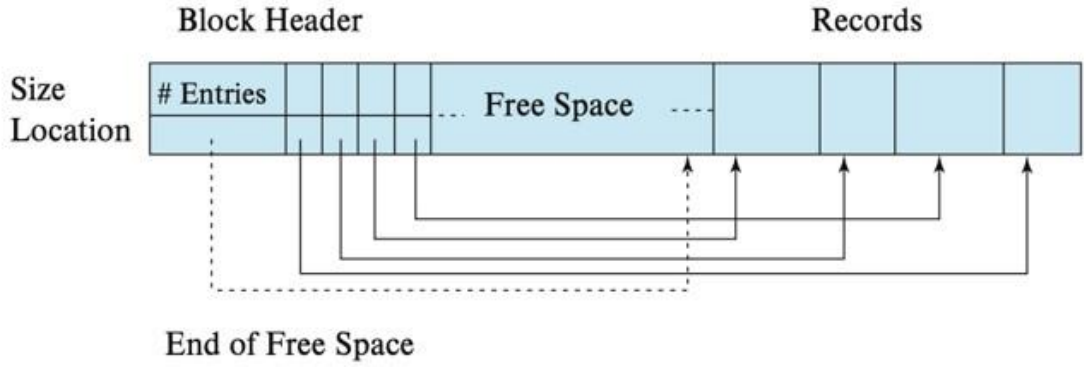
对外部，table heap 需要提供读写数据表的接口，包括对 row 的查找，插入、更新、删除，以及数据表的销毁。另外，还需要对上层提供迭代器，使得上层可以按存储顺序访问堆表的所有记录。注意由于堆表内部可以访问的只有数据页中序列化的结果，并没有真正的 row 对象，所以迭代器解引用返回的 row 对象要额外分配空间生成，这里是在迭代器内部生成并自动回收。



对内部实现而言，一个堆表由若干个页（page）构成，页本是一个广泛的概念，在堆表存储中用 page 的子类 table_page 表示。不同页之间以链表的方式相连（这里

的链表之间的”指针”指的是也中记录的上下页的 page_id，buffer pool manager 可以负责根据 page_id 获取页)。访问数据表时，数据表占用不止一个数据页，则可能需要跳页访问。(注意整体设计图中的 page 并不是真正的页数据，而是存在 table heap 内的 page_id)。

对于每个 table_page 的内部，除了包含记录之外，还包含上下页的位置信息、可插入位置（freepointer）信息等等，并且位于固定位置。因此，每个记录（row 的序列化后的结果），在页中从右向左依次排列，每条记录的 RowId 分为 page_id 和 slot_num 两部分，分别为记录所在的页号和槽号，槽号即是一条记录在一个表中的位置标识。在一个页内，可以通过 slot_num 定位记录的位置。

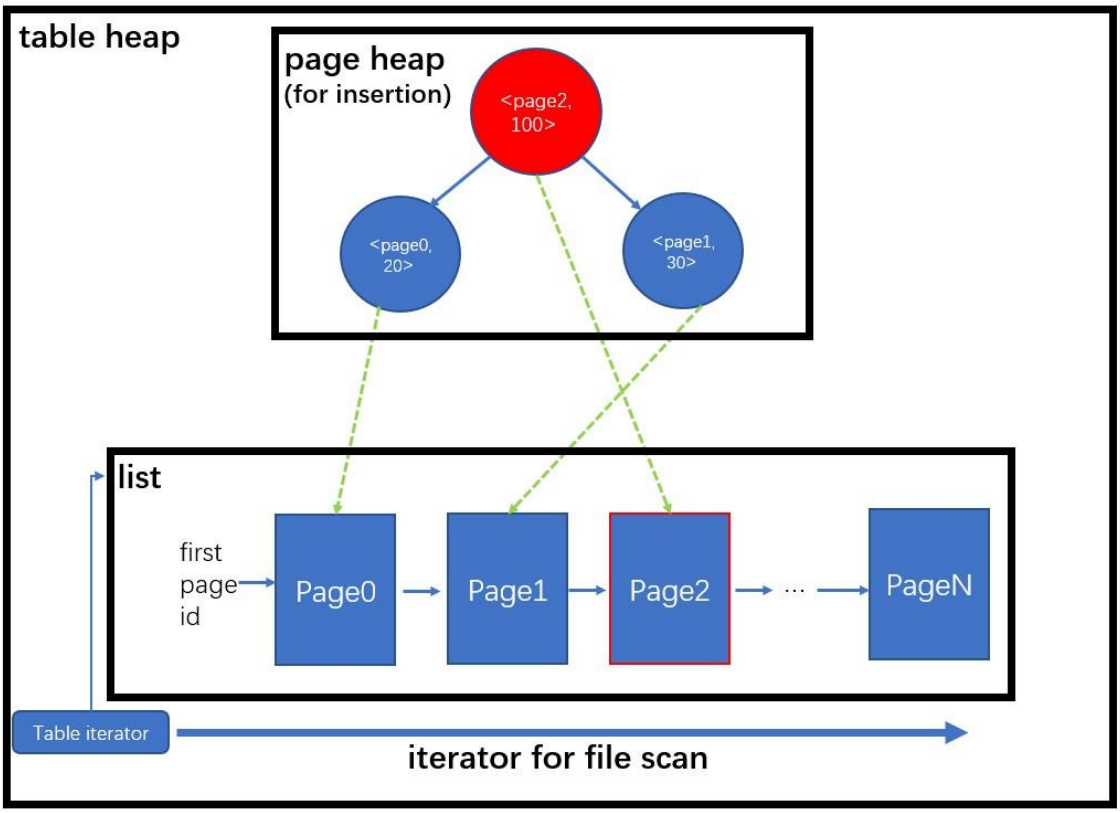


对于堆表的插入，框架给出的策略是线性遍历，即从第一页的开始，逐页分析该页是否有可供插入新记录的剩余空间（插入时可以知道 row 序列化的长度，而页的剩余空间可以通过 freepointer 的位置计算获得)，遇到第一个剩余空间足够的页时，就将其记录插入。如果最后一页空间依然不足，则利用 buffer pool 新建一个页加入尾部，并将新纪录插入（不考虑单记录所占空间大于整个数据页的情况)。

但是我们在实际操作中发现，**线性策略会导致严重的性能问题**，因为要遍历每一个页，才能插入到正确的位置。且不说算法复杂度高，这种策略与 buffer pool 中的 LRU 策略理念相悖，试想这样的场景：buffer pool 内共有 1024 个数据页，并假设初始为空。向一个新建的堆表中不断插入等长记录，使得堆表刚好占有 buffer pool 中的 1024 个数据页，并且插入第 N 条记录，使最后一页 page1023 刚好被插满。此时插入第 N+1 条记录，由于要从 page0 开始遍历到 page1023 逐个检查页是否有空间，

此时 LRU 替换器认为 page0 最远被使用。而当插入第 N+1 条记录时，由于发现 page1023 空间不足，需要分配新页，此时 buffer pool 已被 page0 page1023 占满，replacer 会替换掉最远访问的 page0，换成新页 page1024。此时还好，只替换了一页（注意此时最远被访问的页是 page1）。但当插入第 N+2 条记录时，策略依然是从 page0 开始逐页检查剩余空间，而 page0 已经不再内存池中，必须加载进来，并替换掉最远访问的 page1（此时最远被访问的是 page2），而在发现 page0 空间不足后，还要检查 page1 是否有空间，有需要牺牲掉 page2 把 page1 重新加载回来，以此类推，直到把 page1023 牺牲掉，载入 page1024，检查 page1024，发现 page1024 有剩余空间，才能完成正常插入。所以可以发现，**仅仅是插入第 N+2 条记录这一条记录，就在内存池中替换了 1024 个数据页**！而且在继续插入过程中还会继续存在这种现象，由于磁盘读写速度慢，这种现象完全不可接受。

一个自然的想法是在堆表中记录当前最后一页，或者是上一次插入页的 page_id，然后只需要检查这一页是否有剩余空间，有则插入，无则在这一页后创建新页即可。这种策略虽然可以减少复杂度、避免重复读磁盘，但是不能保证堆表空间利用的效率，因为如果在插入的中途穿插删除操作，并且删除的是位于链表中间位置的数据页的记录，那么这些页中删除留下的空间就无法被利用，堆表空间只增不减，同样不可接受。



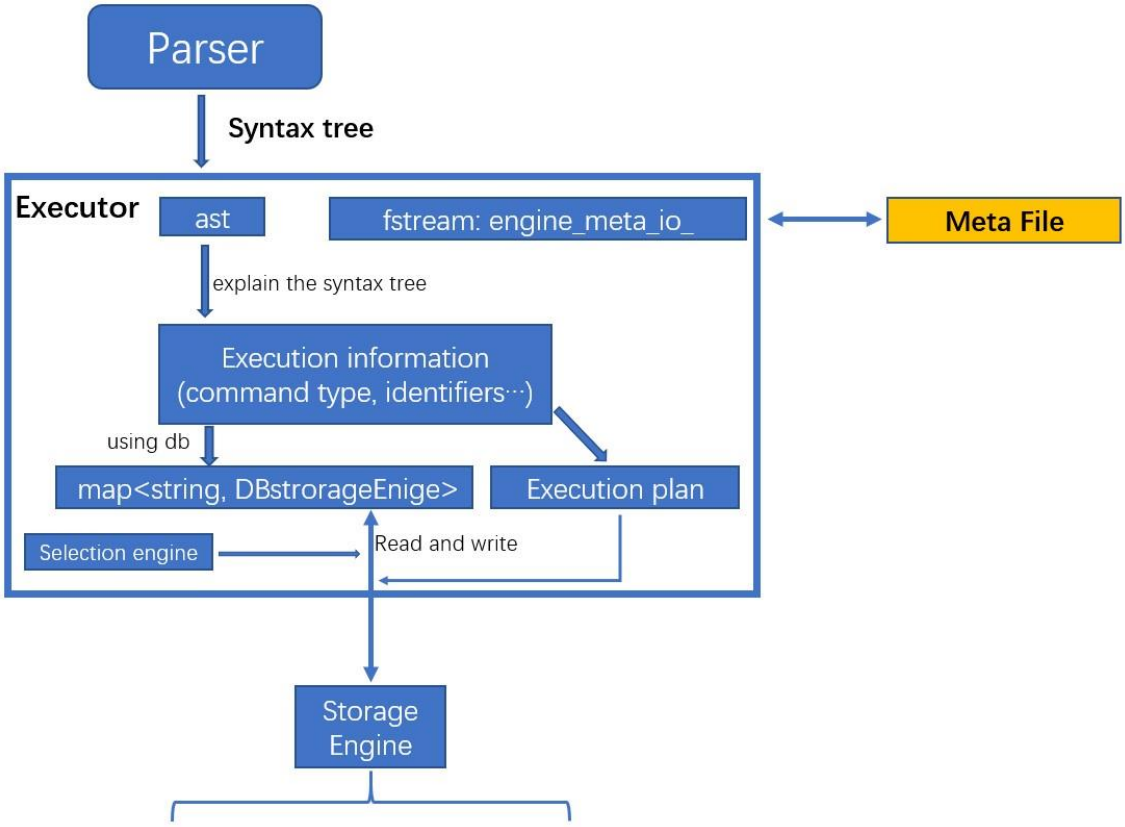
受到“堆表”这一名字的启发，我们决定使用**最大堆的数据结构来管理堆表的插入。**在堆表中，我们维护了一个名为 page_heap_ 的最大堆，堆的节点中存有页号 page_id 和对应页的剩余空间大小（节点结构 <page_id, free_space>），根据剩余空间大小排序。每次插入时的策略非常简单，只需要查看堆顶页的剩余空间，如果剩余空间大于插入记录的空间，则直接插入到这一页中，否则新建一个数据页，插入新记录，并将页号和剩余空间作为新节点插入堆中。**这种策略可以保证每次插入最多访问两个数据页，并且保证每次都将插入剩余空间最大的数据页中，保证了空间利用效率。**当然付出的代价是插入、删除、修改记录时页的剩余空间会改变，可能新增节点，需要改变堆的结构。当然，由于堆表数据页的数量通常不会太多，通常是千或万级，而维护的堆节点结构简单，所以相比于遍历页花费的时间和其他磁盘读写操作，维护堆的时间几乎可以忽略不计。实际中也发现采用这种改进后，插入 10w 行的速度几乎快了 4~5 倍。

用最大堆的方式来管理堆表插入不意味着抛弃堆表的链表结构，每个数据页依然会记录其相邻的数据页的 page_id，只不过链表使用的场景不是插入，而是线性查询。这

就涉及到迭代器：堆表会对外提供迭代器（通过 Begin，End，Find 函数获得），用于遍历堆表的每一条记录，解引用可以获得对应记录的 row。注意堆表本身不含有 row 对象，而是序列化后的结果，因此 row 对象的空间由迭代器生成并维护。拥有迭代器使用权的是上层的 executor，executor 可以通过迭代器线性扫描表的每一条记录(row 对象的形式)，并进行相应操作。

4. Executor 模块

4.1 整体架构



Executor 是 MiniSQL 框架中的第二层，也是跟各个模块对接的关键层。主要功能是根据 parser 层生成的语法树获取 SQL 语句信息，并根据 Catalog 提供的信息生成执行计划，通过 Record Manager、Index Manager 和 Catalog Manager 提

供的相应接口进行相应操作，对数据库进行读写，返回执行结果（如输出文本信息）给上层模块。

ExecuteEngine 内部的成员变量 engine_meta_file_name_ 记录的是数据库的元信息文件名，在/doc/meta 下，默认名为”DatabaseMeta.txt”，记录的是当前所有数据库的 db 文件名（目录/doc/db 下）。ExecuteEngine 维护了一个从数据库名到存储引擎的 map，在构造函数中 ExecuteEngine 会根据从元文件中读取到的数据库文件名，插入并用 db 文件名初始化每个存储引擎。在 CreateDatabase 或 DropDatabase 时会插入或删除对应的存储引擎，并增加或删除对应的 db 文件以及元文件中的对应信息。

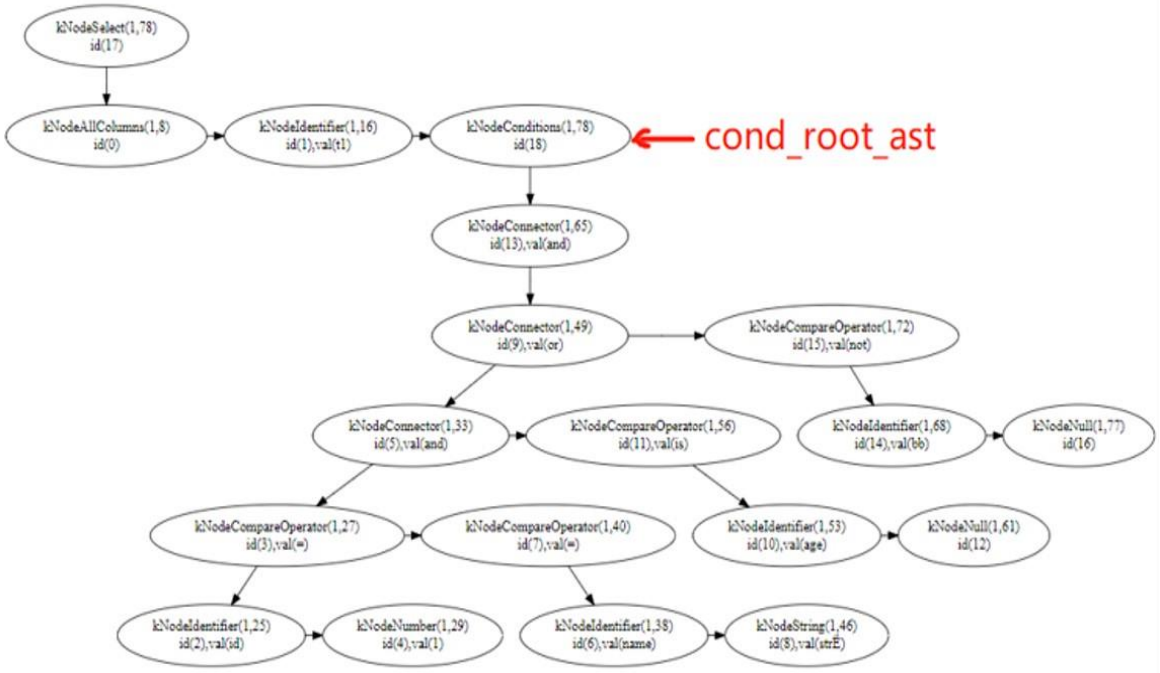
同时，ExecuteEngine 内含有当前使用的数据库名，在执行具体语句时，会现根据当前数据库名从 map 中获取存储引擎，然后利用这个存储引擎对该数据库文件进行操作。ExecuteEngine 对外接口只有 Execute 这个公共函数，接受的是 parser 生成的语法树根节点。另外，ExecuteContext 结构体可以起到向上层（主函数）返回执行结果的作用，目前主要作用是返回输出的文本字符串以及是否退出。

ExecuteEngine 对外接口只有 Execute 这个公共函数，接受的是 parser 生成的语法树根节点。另外，ExecuteContext 结构体可以起到向上层（主函数）返回执行结果的作用，目前主要作用是返回输出的文本字符串以及是否退出。ExecuteEngine 在析构时会对每个存储引擎的 buffer pool 调用 FlushAll，写回所有脏页避免数据丢失。析构会在 mainSelectTupe 函数结束，或强制退出、 quit_flush 被调用时执行。

4.2 条件筛选子模块

```
// my added member function (critical part)
// cond_root_ast: the root node for the Condition Node in syntax
tree
// tinfo: the current selected table info
// iinfos: the indexes info of current table
// rows: receive the result
dberr_t   ExecuteEngine::SelectTuples(const   pSyntaxNode
   cond_root_ast,   ExecuteContext   *context,   TableInfo
   *tinfo, vector<IndexInfo *> iinfos, vector<Row> *rows)
```

在很多语句，如 select、delete、update 中，都涉及到数据表的行筛选，这是 executor 中最重要的部分之一，复用性高，所以单独包装成一个函数。函数有五个参数，其中 cond_root_ast 为条件根节点，是语法树中类型为 kNodeCondition 的节点，所有跟条件有关的信息都在以它为根的子树中。Tinfo 和 iinfos 是表信息和其上的所有索引的信息，由具体的 execute 函数生成并传入，SelectTuples 只需要负责根据表和索引信息，以及条件根节点，把选出的 row 通过第五个参数：Row 的向量指针返回即可（将选出的结果尾加）。



筛选行的总体执行计划是:

	等值条件	范围条件
单条件无索引、多条件(目前暂不支持利用索引加速多条件查询)	利用堆表迭代器线性扫描，利用 RowSatisfyCondition 函数对语法树进行递归，判断是否符合条件。	
单条件且条件列上有索引	利用索引找到对应键的迭代器，通过迭代器获得 RowId，根据 RowId 直接获取 row。	利用索引提供的 FindLastSmallOrEqual 函数获得小于等于目标值的最大键的迭代器，利用 B+ 树的迭代器向前或后遍历。

其中如果没有条件列上的索引，或者筛选条件为多条件（标志是存在类型为 kNodeConnector 的节点），则使用线性扫描（filescan）的方式，利用堆表迭代器遍历每个 row，逐个 row 判断是否符合条件。对于单个 row 是否符合条件，我们利用一个 RowSatisfyCondition 函数，对语法树进行递归，判断一个 row 是否符合条件。

对于单条件且条件列上有索引的情况，如果是等值条件，只需要利用索引的 GetBeginIterator 找到目标键的迭代器，解引用得到 RowId，然后利用 RowId 直接生成 row 并返回即可。

如果是单条件有索引，且是在条件列上的范围查询，则利用索引提供的 FindLastSmallOrEqual 函数返回小于等于该目标键值（如 select * from account where id >= 1 中的 1）的最大键的迭代器，然后根据范围条件遍历得到 RowId，再得到 row 即可。其中根据范围条件遍历的过程中需要注意的细节比较多，如相等与否的判断等，需要特别小心。

4.3 各类语句具体执行方式

- ExcuteCreateDatabase :
    1. 获取数据库名，检查数据库名是否已经存在
    2. 建立数据库文件，插入 ExecuteEngine 的存储引擎 map
    3. 在数据库元文件上增加一行新表名
    4. 全部写回磁盘（数据库元信息比较重要，必须马上写回）
- ExecuteDropDatabase :
    1. 获取数据库名，检查数据库名是否已经存在
    2. 将该数据库存储引擎从 map 内删掉
    3. 删掉数据库文件
    4. 删掉元文件对应行
- ExecuteUseDatabase :
    1. 获取数据库名，判断是否存在
    2. 将 current_db_ 设为要使用的数据库名
- ExecuteShowTables :
    1. 遍历当前数据库存储引擎中的每一个 table，利用 table 元信息，输出表信息，包括表名、列属性、行数、索引信息等。
    2. 将该数据库存储引擎从 map 内删掉
    3. 删掉数据库文件
- ExecuteCreateTable :
    1. 判断是否有在用数据库，判断表明是否已经存在
    2. 根据语法树字符串中生成表的 schema

3. 利用 catalog 的 CreateTable 接口建表
4. 如果有主键，在其上建主键 AUTO 索引
5. 如果有 unique 键，在其上键 AUTO 索引

- ExecuteDropTable :
  1. 判断表是否存在，如果存在，调用 catalog 提供的 DropTable 接口，索引的删除会在其内部自动进行。
- ExecuteShowIndexes :
  1. 获取当前数据库
  2. 遍历每个表的索引索引并输出索引名（包括 AUTO 索引）
- ExecuteCreateIndex :
  1. 获取表
  2. 检查自定义索引名是否符合要求（不能与 AUTO 索引命名格式相同）
  3. 根据语法树获得索引的列名向量
  4. 检查表和表中索引名是否存在
  5. 检查是否有等价索引并给出 Warning
  6. 检查唯一性限制
  7. 利用 Catalog 的 CreateIndex 接口建立索引
  8. 初始化索引，将表中已有数据插入索引中。此时如果发现重复键导致插入失败则撤回建索引操作并报错。
- ExecuteDropIndex :
  1. 获取数据库
  2. 检查是否在删除 AUTO 索引，如果是则拒绝

3. 遍历每个表的每个索引，删除该名称的索引。(这里框架不太合理，应该是指定表来删)

- ExecuteSelect :

1. 获取表
2. 利用 SelectTuples 筛选 row
3. 做投影（生成新 row）
4. 输出（表名 + 列名 + 每行的数据）

- ExecuteInsert :

1. 获取表
2. 根据语法树和表信息生成插入的 row
3. 检查限制约束：包括索引唯一约束和主键非 null 约束
4. 插入行，同时更新索引

- ExecuteDelete :

1. 获取表
2. 利用 SelectTuples 筛选 row
3. 删除 row 并删除索引键

- ExecuteUpdate :

1. 获取表
2. 利用 SelectTuples 筛选 row
3. 保存行名和更新的目标值
4. 生成要新 row
5. 对于每个新 row，检查是否会导致约束错误 (索引不唯一，主键含 null)
6. 新表，同时更新所有被影响的索引键

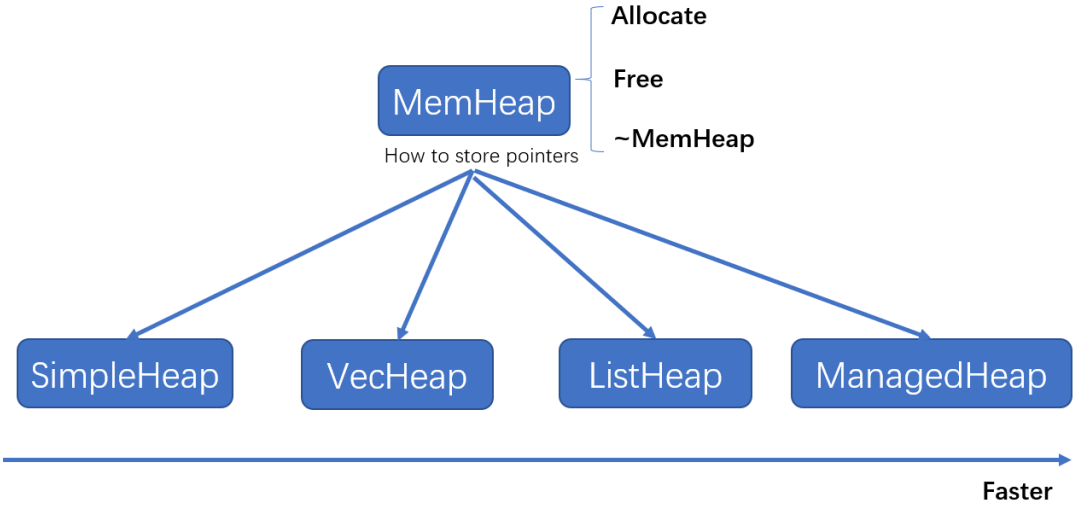
- ExecuteExecfile :

    1. 打开目标 sql 文件
    2. 模仿主函数流程，读取每条指令，解释生成语法树，并执行
    3. 关闭文件
    4. 其中若文件中途有命令无效，则中途停止，不继续执行

- ExecuteQuit :

1. 将 context 的 flag_quit_ 标记为 true，代表结束

5. MemHeap 的简单优化



MemHeap 用于更有效地管理内存分配和回收，析构会自动释放分配的内存空间，尽量避免内存泄露的问题，也能减少频繁malloc 对性能的影响。MemHeap 内部维护了已经分配空间的所有的指针集合，并在free 时进行删除释放，析构时全部删除释放。在MiniSQL 中主要为堆表、row、heap，catalog、executor 所需要生成的对象分配空间，其分配空间的操作（Allocate 函数）在程序执行的过程在被调用的频率极高（其中大部分是row 的heap 为每个field 分配空间），在实际性能测试（perf 分析）的过程中发现其对性能的影响极大，占用了主要的时间（优化前，debug 模式插入带索引的10 万行约300 秒）

VecHeap：在分析中我们发现SimpleHeap 内部通过维护一个unordered_set 来存储指针，这种数据结构在查找时速度较快。但我们发现这样会导致插入的复杂度较高，而free 往往是析构时一并执行的。相比于单个free 操作，单个allocate的操作要频繁的多，所以我们将unordered_set 改成了vector，即VecHeap，发现速度提升了（从300 秒加快至200 秒左右）

ListHeap：但是VecHeap 的性能仍不是很理想，其push_back 依然占据了

程序执行的大部分时间（约60% 80%），后来我们发现，由于在MiniSQL 中大部分Allocate 的场景是row 的MemHeap 去维护它的每个field 的空间，而field 的数量通常不会很大（个位数，或几十个），在这种较少的数据量下，使用STL 提供的vector 或set 来维护可能显得臃肿而没必要。因此我们手写了简单的链表用来存储指针，使得插入复杂度为O(1)，而删除则需要遍历。在这种轻量级的数据结构下，速度得到了大幅提升，debug 模式下插入带索引的10 万行的时间加快到了约34 秒。

说明：在后来我们增加了较多的free操作，导致VecHeap和ListHeap的速度有所降低，且STL在release模式下性能可以接受。但我们最终使用了ManagedHeap，查找、删除速度都较快，在总体设计报告中有详细说明。

6. 异常处理

6.1 异常输入处理

异常输入方面，我们对建表时输入的字符串长度格式、各种操作的匹配格式有着严格的检查并会提示错误信息（如筛选不存在的列名，插入的行数与表行数不匹配等）。

```
minisql > create table bad_t(a int, b char(2.5));
[Error]: Illegal input for char length!
[Failure]: SQL statement executed failed!
```

```
minisql > select bad_col from t;
[Error]: Column "bad_col" not exists!
[Failure]: SQL statement executed failed!
```

6.2 中断处理

实际测试中发现如果中途强制退出（如ctrl+Z/C，或运行时错误），会导致内存池中的页没有及时flush 回磁盘，所以我们接受中断退出信号并调用quit_flush函数来在中断结束前写磁盘。避免数据丢失。

```
void quit_flush(int sig_num)
{
  cout<<"\n[Exception]: Forced quit!"<<endl;
  delete engine;//deconstruction will flush all dirty pages in buffer pool back to disk
  exit(-1);
}
```

```
int main(int argc, char **argv) {

  signal(SIGHUP, quit_flush);
  signal(SIGTERM, quit_flush);
  signal(SIGKILL, quit_flush);
  signal(SIGABRT, quit_flush);
  signal(SIGALRM, quit_flush);
  signal(SIGPIPE, quit_flush);
  signal(SIGUSR1, quit_flush);
  signal(SIGUSR2, quit_flush);
  signal(SIGTSTP, quit_flush);
  signal(SIGSEGV, quit_flush);
```

当然这种处理方式有局限性，比如SIGKILL出现时其实无法继续让程序处理中断，另外这种方法配合事务使用更好，不然容易导致数据库一致性受损。目前只是初步解决方案，以后可以用**日志（log）**来恢复数据。

7. 个人总结

在MiniSQL的实现中，我更加深入地理解了数据库的底层原理，认识到了除了算法、数据结构，对磁盘存储的读写管理也是优化性能的关键。特别是在堆表的优化中我体会到了堆这种数据结构的优越性、适配LRU策略的方法。课上的理论只是和实践编程感觉还是蛮不一样的，需要解决各种实际问题。在这次项目中，我的框架理解能力、模块对接能力得到了很大提升，收获很大，最后看到MiniSQL成功运行、能处理大量数据，成就感很足。十分感谢助教们提供的优秀框架。