MiniSQL 总体设计报告

管嘉瑞 张峻瑜

时间：2022 年 6 月 4 日

Contents

1 MiniSQL 系统概述

1.1 背景

1.1.1 编写目的

- 运用本学期所学数据库知识，设计并实现一个精简版单用户 SQL 引擎 MiniSQL，允许用户通过字符界面输入 SQL 语句来实现基本的增删改查操作，并且能够通过索引来优化查询性能。
- 了解数据库系统中各个模块的实现与优化方法，了解前沿数据库的大致架构。
- 提高系统编程能力，加深对于数据库底层设计的理解。

1.1.2 项目背景

数据库管理系统（DBMS）是相互关联的数据的集合，也是一组访问这些数据的程序。数据集合（通常称为数据库）包含与企业相关的信息。DBMS 的主要目标是提供一种既方便又高效的数据库信息的方法。

数据库管理已从专业的计算机应用程序演变为几乎所有企业的中心部分，因此，有关数据库的知识系统已成为计算机科学的重要组成部分。DBMS 的开发也是计算机技术领域中尤为基础也尤为重要的部分，对于数据库基础知识的学习与掌握，是十分必要的。

本学期，我们学习了数据库系统的相关知识。为了解数据库底层各个模块的实现细节与相互关系、加深对于数据库系统的理解，本项目将基于 C++ 实现一个简单的数据库管理系统，

1.2 功能描述

1.2.1 基本数据读写功能

MiniSQL 支持基本的数据库操作，包括建立数据库、建表、插入行、删除行、更新行，也包括删除表、删除数据库，并且支持条件查询（单条件或多条件）。数据类型方面，MiniSQL 支持 int(整型), float(浮点型), char(字符串型) 三种数据库类型，其中 char 类型需要指定长度，每种数据类型都可以为 null。

MiniSQL 支持执行文件。Execfile 将逐句执行文件中的 SQL 语句。如果语句语法错误或执行失败，则会中止执行文件。

本组实现的 MiniSQL 在原有框架上进行了较多优化（如堆表优化、索引重构、memHeap 优化等），使得数据库操作速度非常快。

1.2.2 索引及约束

MiniSQL 支持 B+ 树唯一索引。索引有两种类型：AUTO 索引和自定义索引。其中 AUTO 索引是数据表建立之初就建立起来的，包括主键 AUTO 索引：对于声明为主键的 (单或多) 键值，会在其上自动建立以 _AUTO_PRI_ 为前缀的索引；还包括唯一 AUTO 索引：对于声明为 unique 的 (单) 键值，会在其上自动建立以 _AUTO_UNIQUE_ 为前缀的索引。除了自定义索引外，允许用户在不存在重复键值的键上建立自定义索引，便于快速查询。

本项目中索引跟唯一性约束的关系如下：

1. 索引不允许重复，只要在键值上建了索引，键值必是唯一的。如果某种操作 (如插入或更新行)，会导致索引键上有重复，则会被拒绝 ([Rejection])

2. 对于声明为 primary key 或 unique 的列，自动建立索引，保证唯一性。

3. 对于没有声明为 primary key 或 unique 的列，不会自动建索引，插入或更新过程中也没有重复限制。但如果要在这样的列上建立索引，首先会报警告 ([Warning])，提升用户确保列上无重复键值。然后将已有列逐行插入索引中，若中途发现键值重复，则会报 Error，并丢弃这个没有建成的索引。否则建索引成功，并且该键值与 unique 键保持相同性质。

MiniSQL 支持删除索引 (drop index)，不过需要注意的是，删除 AUTO 索引是不被允许的，只能删除自定义索引。

索引键上允许插入 null 值 (但是同样不可重复，最多插一个)。但对于主键索引，有约束使得不允许插入 null 值 (如果是 multi key，任何一列都不可为 null)。以上设定均参考 mySQL 的设定。

支持建立多键索引和多键上的重复检查，但目前只有单键索引会用于加速查询。

索引可以用于加速条件查询。对于条件查询 (不只是 select，也包括 delete 和 update 中的条件筛选)，如果没有索引，需要线性扫描堆表来逐个检查每行是否符合条件，速度较慢。用索引可以快速找到需要的行。索引加速查询既可以用于等值查询也可以用于范围查询，其中范围查询的方式是找到小于等于目标值的最大键值在索引中的迭代器，然后通过迭代器按某种方向的遍历实现范围查询。目前索引加速查询只用于单条件查询，多条件查询与索引的结合涉及到执行计划的优化，有待实现。

另外，在程序程序上我们对索引进行了重构和去模板化，使得其对上层封装性提升、性能提高。

1.2.3 异常处理

为了增强系统的鲁棒性，我们实现了部分异常处理功能。

首先是异常输入方面，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)。

另一方面是中途强制退出的处理。我们发现如果在程序运行时因各种原因 (包括 ctrl+Z 或 C 退出，运行时错误等) 退出，buffer pool 中的脏页不会写回磁盘，从而导致了数据丢失。主流数据库采用日志来解决此类问题。我们利用 signal 函数接受中断信号，并调用 quit_flush 函数，使得脏页全部写回磁盘再退出，避免数据丢失。这种方法虽然无法解决全部情况的中断退出，但是成本低、多数时有效，对提升系统稳定性有重要作用。

1.3 运行环境和配置

本项目部分参考了 CMU-15445 BusTub 框架，并做了一些修改和扩展。
本项目基于 C++ 17 开发，Linux 平台。

1.3.1 编译 & 开发环境

- `apple clang`: 11.0+ (MacOS)，使用 gcc –version 和 g++ –version 查看
- `gcc&g++` : 8.0+ (Linux)，使用 gcc –version 和 g++ –version 查看
- `cmake`: 3.16+ (Both)，使用 cmake –version 查看

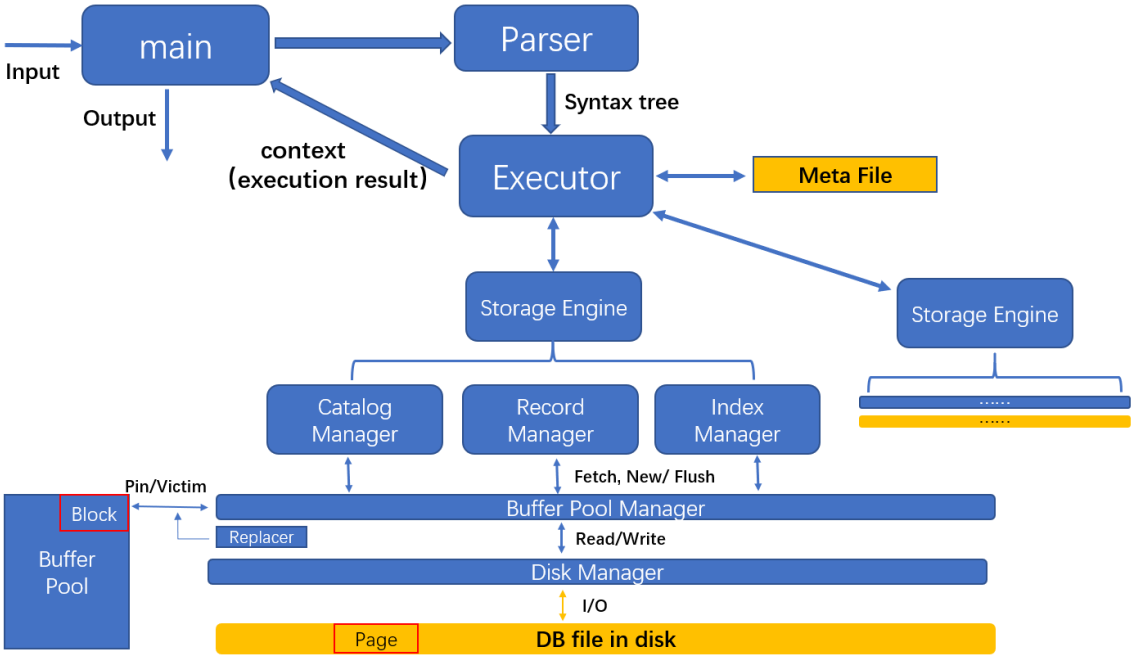
1.3.2 构建步骤

```
// bash or zsh
mkdir build
cd build
cmake ..
make -j
// 可执行文件将会被输出到 /build/bin/main
```

1.4 参考资料

2 MiniSQL 系统结构设计

2.1 总体设计



参考原框架与教材知识，MiniSQL 分为以下部分：

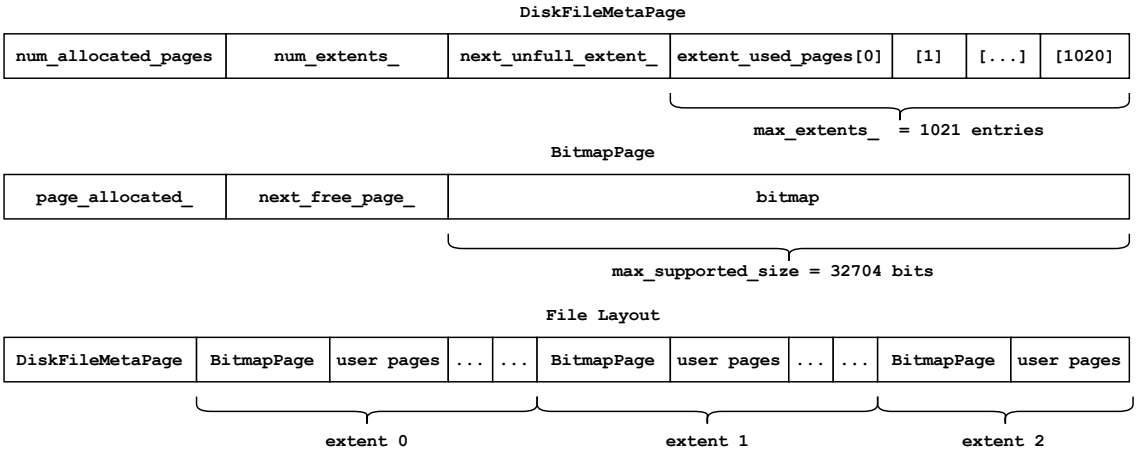
1. Parser：负责读取用户输入并转换为语法树
2. Executor：读取语法树，生成并优化执行计划（在本项目中，可以利用索引进行查询，但无进一步执行计划的生成）。调度 CatalogManager,RecordManager,IndexManager 对 SQL 语句进行执行。同时，Executor 也负责数据库的选择、创建、删除，并把相关信息存储在 Database Meta File 中。每个数据库对应一个 DBStorageEngine。
3. CatalogManager：存储并管理数据库的元信息、表以及索引的信息、创建与删除，为 Executor 提供 IndexManager 以及 RecordManager
4. RecordManager：堆表的管理，记录的插入、删除、更新。
5. IndexManager：索引的管理，记录的插入、删除、更新。
6. BufferPoolManager：实现用户页的缓存，加速页的写入、读取。
7. DiskManager：管理磁盘上物理文件的读写。负责将页写入磁盘或者从磁盘读出。

2.2 DiskManager 模块

本项目中，每个数据库的数据各自存储在单一文件之中，文件名为 {数据库名}.db。该文件被划分为若干个页 (page), 每页大小为 4096 字节。每个页被物理页号 (physical_page_id) 唯一标识, 同时其中的用户页具有供上层模块标识用的逻辑页号 (logical_page_id)。页的分配、释放、读取与写入由 DiskManager 模块统一管理。

- 用户页: 上层模块通过 DiskManager 申请的页。
- 系统页: 除过用户页之外的页。这些页被 DiskManager 用于存储关于磁盘、页管理的原信息。
- 物理页号：DiskManager 从磁盘写入/读出页时使用的页号，物理页号与页一一对应。
- 逻辑页号：为了隐藏 DiskManager 的实现细节，系统页对上层模块不可见，只有用户页能被上层模块访问。每个用户页被逻辑页号唯一标识。在用户页中，逻辑页号从 0 开始连续增长。

.db 文件的文件布局如下：



为了加速页的分配与删除，本项目不仅仅对用户页采用了缓存，也对系统页采用了缓存。系统页的缓存由 DiskManager 单独管理，管理的函数为：

- `ReadPhysicalMetaPage(page_id_t physical_page_id)`
  1. 从 buffer 中寻找该页
  2. 若找到，则直接从内存中读取
  3. 若未找到，则从磁盘中读取，并通过 `replacer_` 替换掉 buffer 中的某页。

4. 若被替换的页为 dirty，则把该页写入磁盘

- `WritePhysicalMetaPage(page_id_t physical_page_id)`

1. 从 buffer 中寻找该页
2. 若未找到，将该页存入 buffer，写入数据，并标记为 dirty
3. 若找到，直接写入数据，并标记为 dirty

`DiskManager` 模块的接口以及实现如下：

- `void ReadPage(page_id_t logical_page_id, char *page_data)`：从文件中读取 `logical_page_id` 标识的用户页，并把页数据写入 page_data 指向的内存中。

1. 调用 `IsPageFree()`，若该页未被分配，抛出错误。
2. 若该页未被分配,调用 `ReadPhysicalPage()` 读取数据。`ReadPhysicalPage` 的实现如下：
   (a) 调用 `MapPageId()`, 根据逻辑页号计算出 `extent_id` 对应的位图页与物理页号
   (b) 调用 `ReadPhysicalMetaPage()`，读取位图页
   (c) 若位图页中该页对应的 `bit` 为 0，抛出异常
   (d) 否则，将该页的数据读入内存

- `void WritePage(page_id_t logical_page_id, const char *page_data)`：将 `page_data` 指向的 4096 字节写入 logical_page_id 所标识的用户页对应的文件块中。

1. 调用 `IsPageFree()`，若该页未被分配，抛出错误。
2. 调用 `MapPageId()`, 根据 logical_page_id 计算 physical_page_id。
3. 调用 `ReadPhysicalPage()`，将数据读入内存。

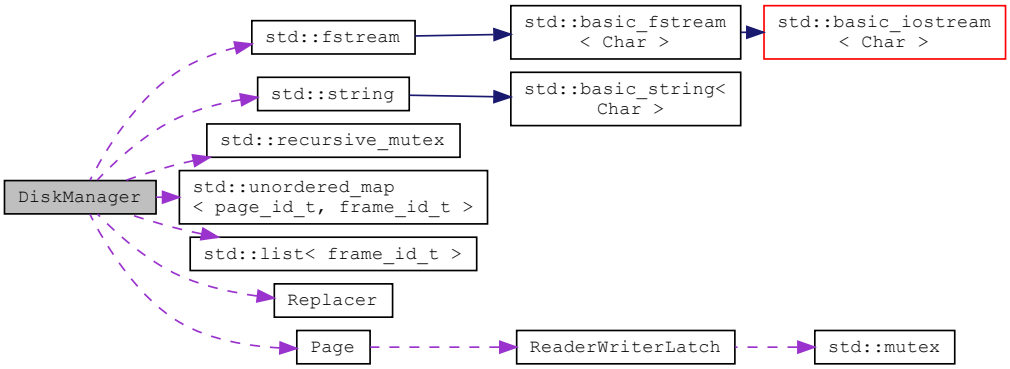
- `page_id_t AllocatePage()`：申请分配一个用户页，返回新分配用户页的逻辑页号。

1. 根据 `next_extent_id_`,找到对应 extent 的位图页。若大于 `GetMaxExtentNum()`，抛出异常。
2. 从位图页中获取 `next_free_page`，若大于 `GetMaxSupportedSize()`，抛出异常。
3. 调用 `BitmapPage::AllocatePage`，分配一个页，并保存分配的页号。
4. 更新 `BitmapPage::next_free_page_`。
5. 更新 `next_unfull_extent_`。

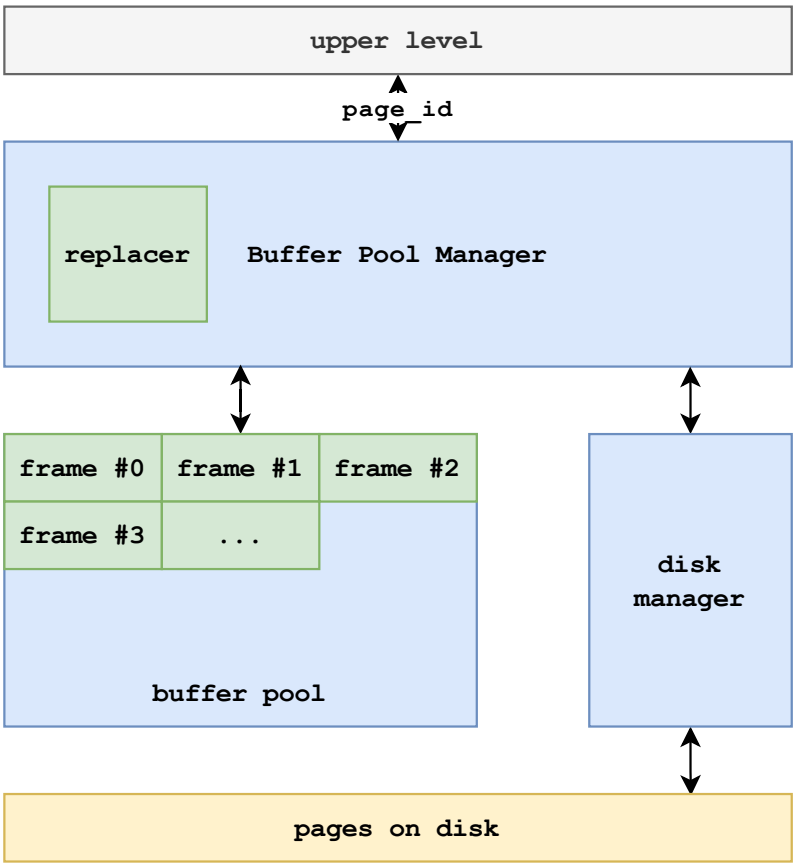
6. 返回分配的页号。

- `void DeAllocatePage(page_id_t logical_page_id)`：释放被 logical_page_id 所标识的用户页。
  1. 调用 `IsPageFree`，若未被分配，抛出错误。
  2. 调用 `ReadPhysicalMetaPage`，读入位图页。
  3. 调用 `BitmapPage::DeAllocatePage`，释放页，若失败则抛出错误。
  4. 更新 `BitmapPage::next_free_page_`。
  5. 更新 `next_unfull_extent_`。
- `bool IsPageFree(page_id_t logical_page_id)`: 查询被 logical_page_id 所标识的用户页是否已经被分配。
  1. 调用 `FetchPhyscicalMetaPage`，获取对应的位图页。
  2. 根据位图页中对应的位，返回分配情况。
- `void Close()` ： 关闭 DiskManager, 保存相关信息，释放文件句柄。
  1. 将全部 buffer 中的 dirty 页写入磁盘。
  2. 将 `DiskMetaPage` 写入磁盘。
  3. 释放 buffer 的内存。

该模块的依赖关系图如下：

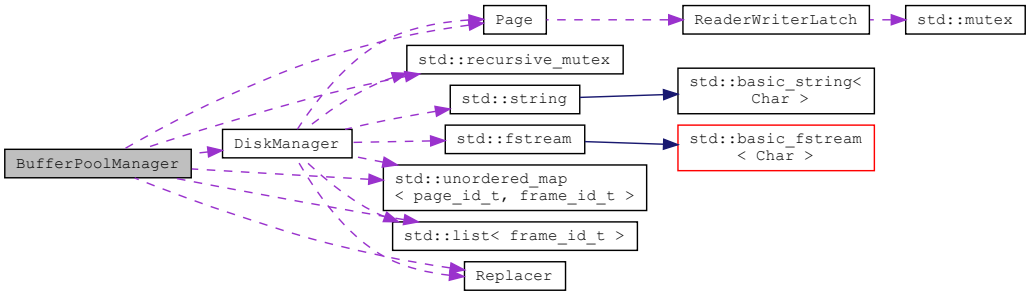


2.3 BufferPoolManager 模块



为了提升从磁盘读/写页的速度，BufferPoolManager 对所有用户页进行了缓存处理，这大大加快了上层模块获取/写入页数据的速度。

该模块的依赖关系图如下：



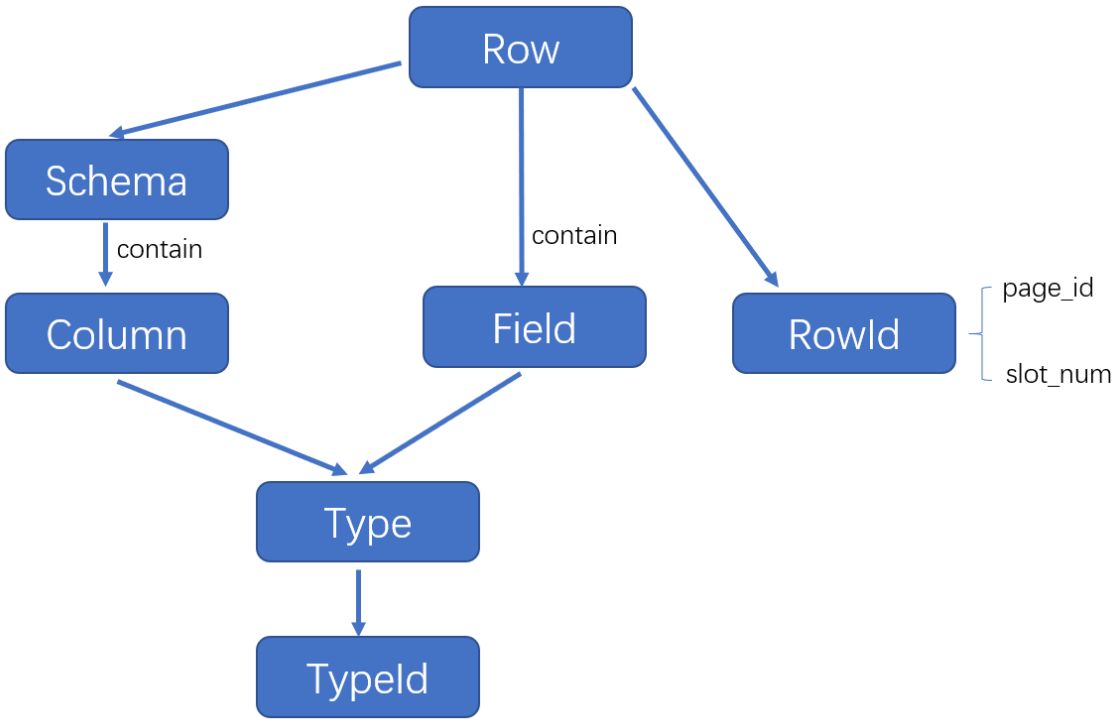
该模块的接口及实现如下：

- `Page * FetchPage(page_id_t LogicalPageId)`：获取 LogicalPageId 标识的用户页对应的 Page 对象。
  1. 检查该页是否在缓存中。
  2. 若不在，从 free_list 中获取或调用 replacer 获取要替代的缓存页，并把目标页读入该缓存页。
  3. 目标页引用计数加一。
  4. 返回目标页对应的 Page 对象。
- `bool UnPinPage(page_id_t LogicalPageId,bool dirty)`：引用计数减一。
- `page_id_t NewPage(page_id_t & pageId)`：申请分配一个用户页。
  1. 调用 `DiskManager::AllocatePage`，申请分配一个新的用户页。
  2. 将该页加入缓存，引用计数加一。
- `bool DeletePage(page_id_t pageId)`：删除一个用户页。
  1. 检查该页是否在缓存中。
  2. 若在，从缓存和 replacer 中移除该页，将对应的 frame 加入 free_list
  3. 调用 `DiskManager::DeAllocatePage`，释放该页。
- `FlushPage(page_id_t pageId)`：手动将该页写入磁盘。
- `FlushAll`：将全部为 dirty 的页写出磁盘。
- `IsPageFree`：查询该页是否已被分配。

2.4 RecordManager 模块

2.4.1 整体设计

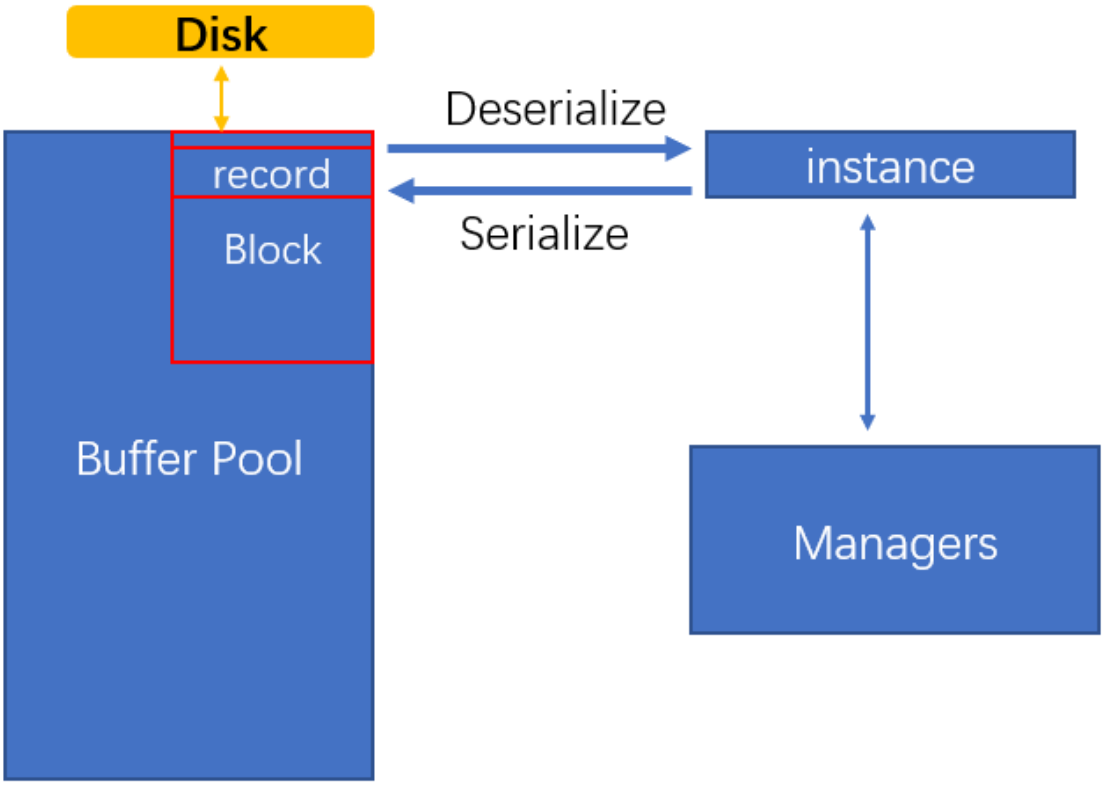
Record Manager 管理的对象是数据表中记录。一条记录 row 由若干个 field 构成，field 是存储数据的单元，由数据类型 (TypeId). 其中 schema 记录表中每一个 row 的结构，由若干个 column 构成。Column 是每一列的元信息，除了记录对应 filed 的类型外，还要记录列名、下标、是否唯一、是否可为 null 等，如果是字符串类型还要记录最大长度。某种程度上来说 field 是一条 row 在一个 column 上的实例化。



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。

2.4.2 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。

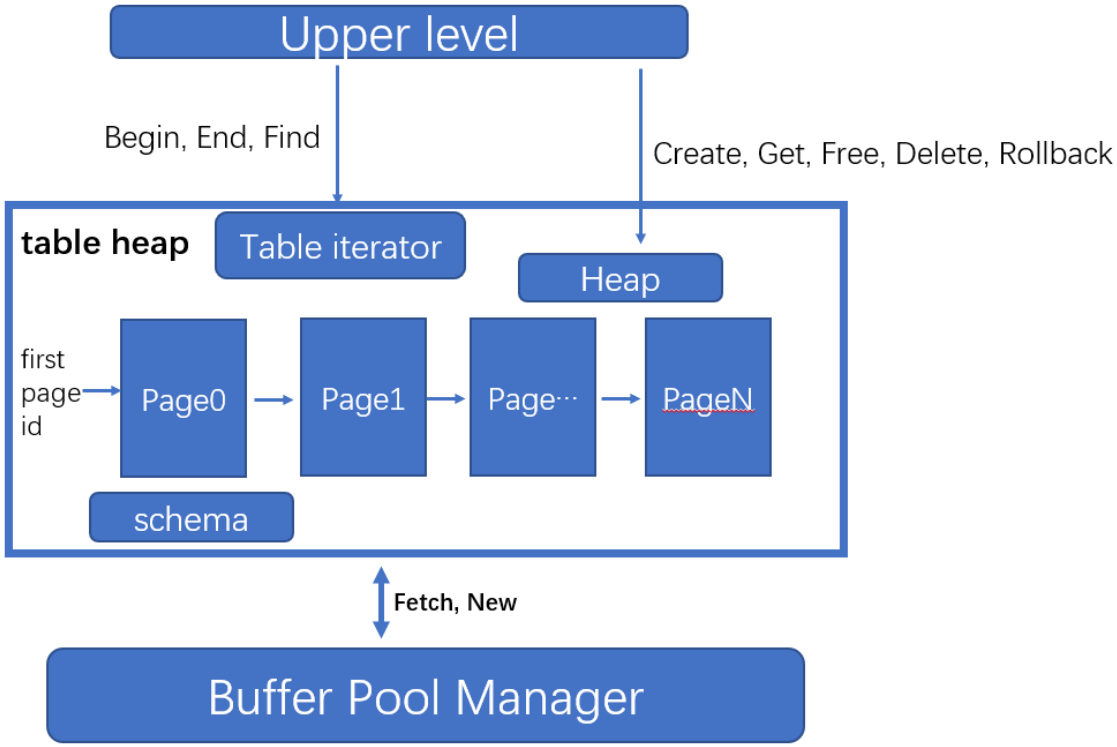
序列化的过程较为简单，只需要构成对象的成员通过提供的 MACH_WRITE 宏写入字节流即可，而反序列化只需要在确定头指针的情况下按照与序列化相同的顺序利用 MACH_READ 宏读出数据并生成对象即可。需要注意的是，当内部成员有重要的元信息时，如字符串的长度，则需要将这类元信息也进行序列化，使得反序列化时有办法知道每步操作的内存范围。

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。

2.4.3 堆表的实现与优化

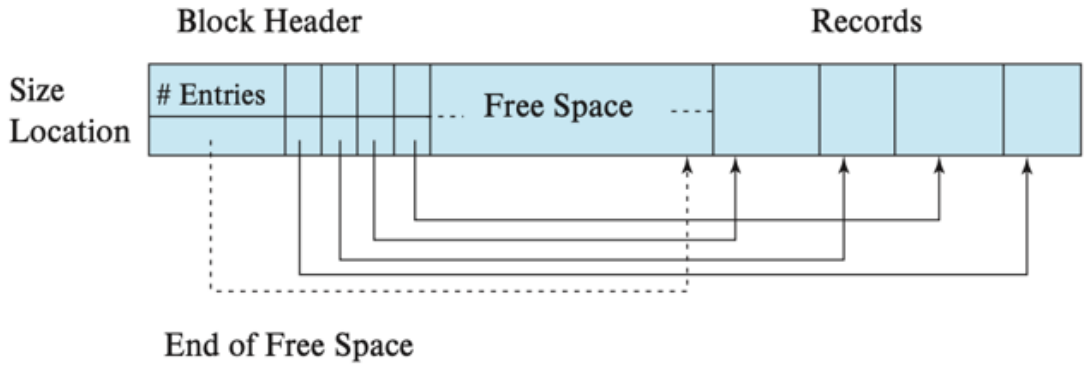
堆表 (table heap) 是 MiniSQL 非常重要的组成部分，是真正存储表内容的数据结构，含有一个表的全部数据。

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)。

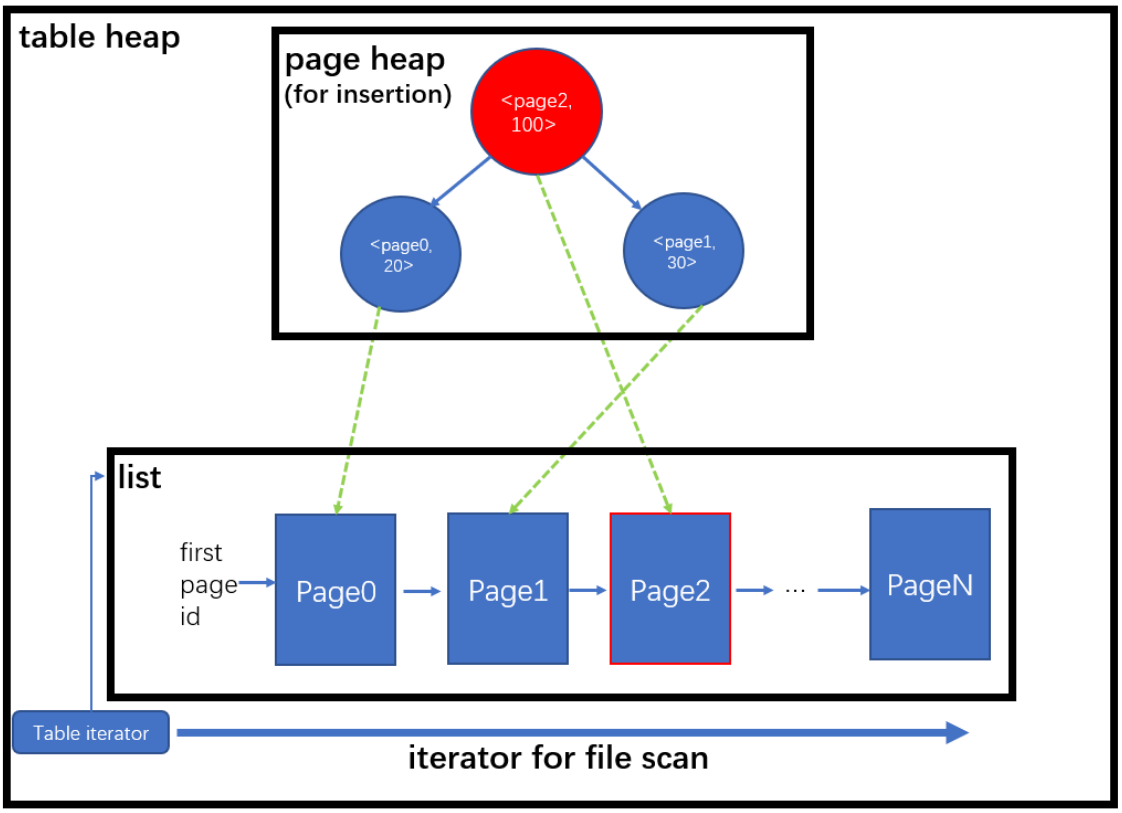
对于每个 table_page 的内部，除了包含记录之外，还包含上下页的位置信息、可插入位置（freepointer）信息等等，并且位于固定位置。因此，每个记录（row 的序列化后的结果），在页中从右向左依次排列，每条记录的 RowId 分为 page_id 和 slot_num 两部分，分别为记录所在的页号和槽号，槽号即是一条记录在一个表中的位置标识。在一个页内，可以通过 slot_num 定位记录的位置。



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。

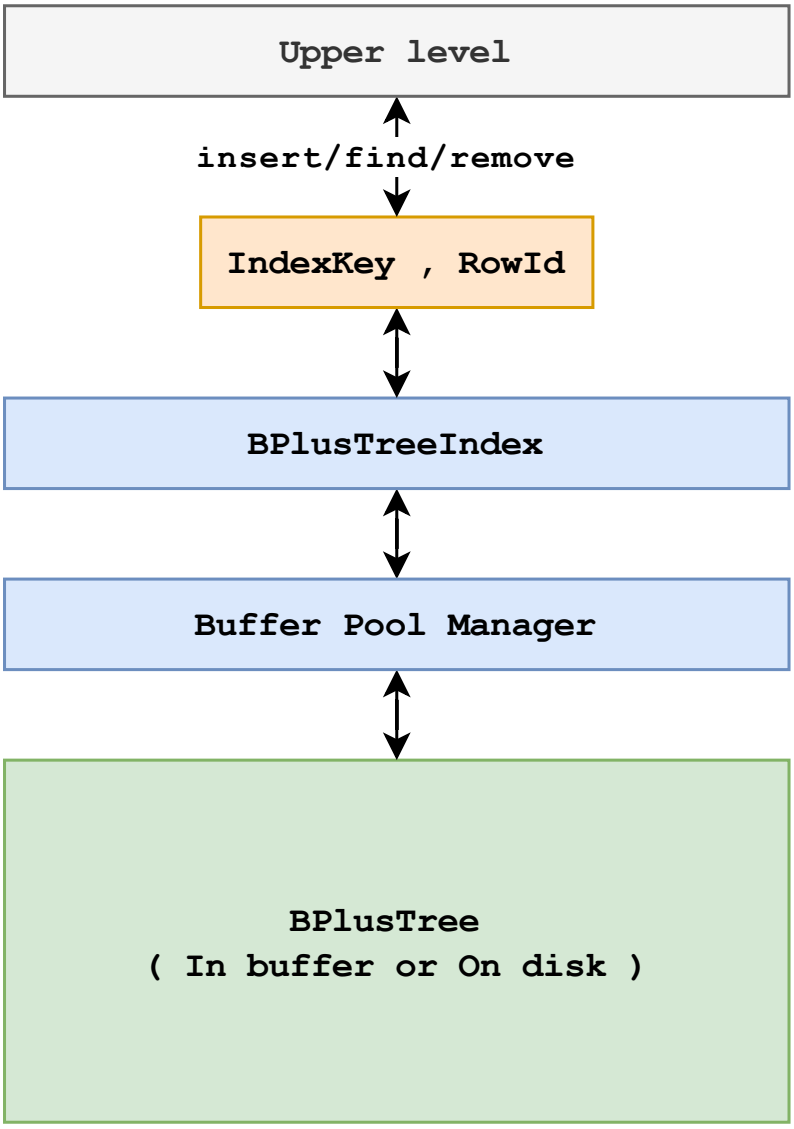
一个自然的想法是在堆表中记录当前最后一页，或者是上一次插入页的 page_id, 然后只需要检查这一页是否有剩余空间，有则插入，无则在这一页后创建新页即可。这种策略虽然可以减少复杂度、避免重复读磁盘，但是不能保证堆表空间利用的效率，因为如果在插入的中途穿插删除操作，并且删除的是位于链表中间位置的数据页的记录，那么这些页中删除留下的空间就无法被利用，堆表空间只增不减，同样不可接受。



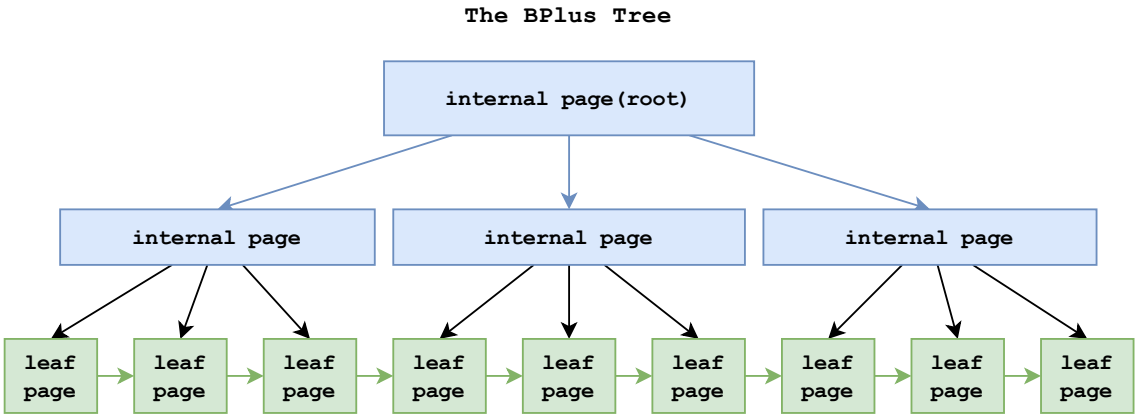
受到“堆表”这一名字的启发，我们决定使用**最大堆的数据结构来管理堆表的插入**。在堆表中，我们维护了一个名为 page_heap_ 的最大堆，堆的节点中存有页号 page_id 和对应页的剩余空间大小（节点结构 <page_id, free_space>），根据剩余空间大小排序。每次插入时的策略非常简单，只需要查看堆顶页的剩余空间，如果剩余空间大于插入记录的空间，则直接插入到这一页中，否则新建一个数据页，插入新记录，并将页号和剩余空间作为新节点插入堆中。**这种策略可以保证每次插入最多访问两个数据页，并且保证每次都将插入剩余空间最大的数据页中，保证了空间利用效率**。当然付出的代价是插入、删除、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。

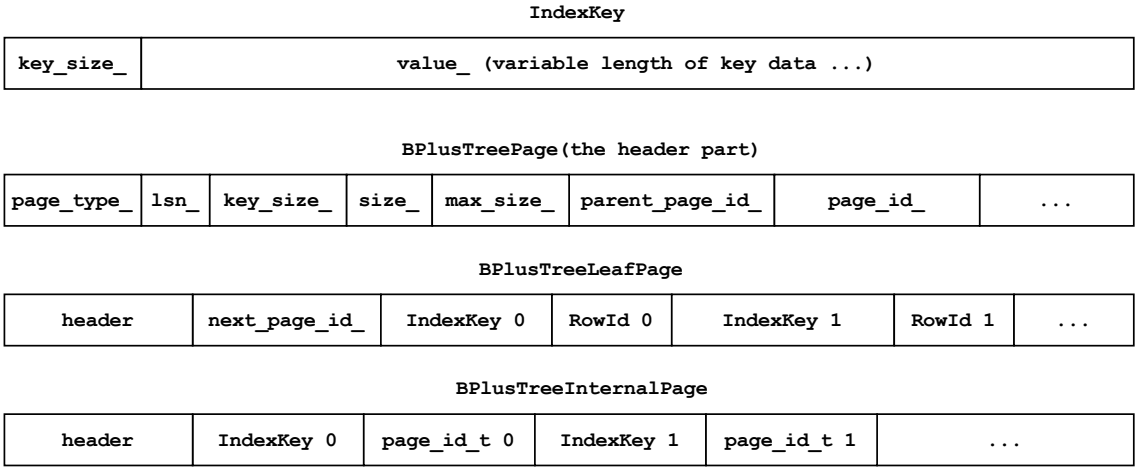
2.5 IndexManager 模块



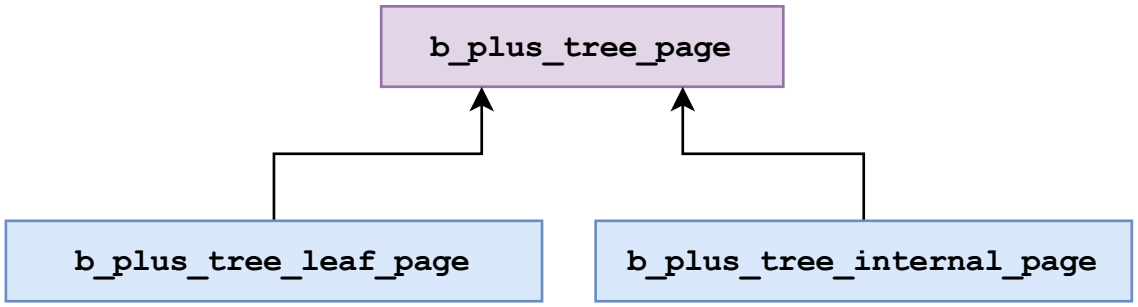
IndexManager 负责管理索引项的添加、删除与访问。由于索引本身特殊的数据结构，给定某个键，我们能够快速的从索引中获取该键对应记录的 RowId，从而快速从堆表中取得该行记录。本项目实现了在**任意长度**键上建立几乎**零额外空间消耗**（也就是，除了存储键本身的值与 RowId 所需的空间之外，几乎没有额外的空间消耗）的 B+ 树索引。在一个表的一个索引之中，包含了若干个索引项。每个索引项由两部分构成：IndexKey 与 RowId。IndexKey 存储了键值，RowId 存储了该索引项对应记录在堆表中的位置。本模块的功能就是实现建立 B+ 树索引、删除、添加、访问索引项的功能。



一颗 B+ 树由若干个叶子节点 (BPlusTreeLeafPage) 与内部节点 (BPlusTree InternalPage) 构成。内部节点按照增序存储了若干个子树的最小键值，以及子树的根节点页号。索引项以增序存储在叶子节点之中。每个内部节点或者叶子节点都单独占用一整个用户页。对于不同的索引，其 IndexKey 所占空间不同，因此每个内部节点与叶子节点所能存储的最大项数也不同，且在程序运行期间动态计算决定。以下是 IndexKey、BPlusTreePage、BPlusTreeLeafPage 与 BPlusTreeInternalPage 的内存布局。



BPlusTreePage、BPlusTreeLeafPage 与 BPlusTreeInternalPage 的继承关系如下：



值得注意的是，本模块摒弃了原框架中的模板设计，而是采用了单一的类来实现 BPlusTreeIndex，该实现方法有诸多好处，详见” 设计亮点与 Bonus” 章节。

本项目未采用 GenericKey 以及 GenericComparator，而是使用自己实现的 IndexKeyComparator 来进行 IndexKey 之间的比较。原理与 GenericComparator 相同，但在内存管理方式上略有区别，详见” 设计亮点与 Bonus” 章节。

本模块提供的接口以及实现如下：

- `BPlusTreeIndex::InsertEntry(const Row &key,RowId rowid)`：插入一个索引项。
  1. 将 row 序列化为 Indexkey.
  2. 调用 `BPlusTree::Insert` 函数，尝试插入该 key。
  3. 若插入失败，则代表已有重复键值，返回 False.
  4. 否则，返回 true.
  5. `BPlusTree::Insert` 函数实现如下：
     (a) 调用 `BPlusTree::InternalInsert()`，插入键值，并获取可能的“分裂页”。分裂页指的是，插入子节点导致节点分裂而产生的新页。
     (b) 若有重复，返回 false。否则：
     (c) 若有分裂页，创建新的树根，并将两个子树连接到树根上。
     (d) 返回 true.
     (e) `BPlusTree::InternalInsert` 函数实现如下：
        i. 若该页为叶子节点，执行 2。否则执行 6。
        ii. 若有重复，则返回：重复。
        iii. 将 IndexKey 与 RowId 组合成 BLeafEntry，并二分查找，插入该项。
        iv. 若页已满，则申请新页，并对半分配两页所含的索引键值。
        v. 返回：节点中第一个键值、新分配的页、无重复。
        vi. 二分查找，找到要插入索引项应当处于的子节点 x，调用 `InternalInsert(x)`。
        vii. 若有重复，则返回：重复。
        viii. 若无新创建的页，则返回：节点中第一个键值、无重复。
        ix. 否则，将新页插入该节点。
        x. 若页已满，则申请新页，并对半分配两页所含的索引键值。
        xi. 返回：节点中第一个键值、新分配的页、无重复。
- `BPlusTreeIndex::RemoveEntry(const Row &key)`：删除一个索引项。
  1. 将 row 序列化为 Indexkey。
  2. 调用 `BPlusTree::Remove`。

3. `BPlusTree::Remove` 的实现如下：
4. (a) 调用 `BPlusTree::InternalRemove()`，移除索引项。
   (b) 若返回未找到，则返回 false。
   (c) 若某个子节点的子节点数变为 0，则删除该子节点。
   (d) 若仅有一个子节点，则删除根节点，并把子节点作为新的根节点。
   (e) `BPlusTree::InternalRemove()` 的实现如下：
      i. 若该节点为叶子节点，执行 2。否则，执行 4。
      ii. 二分查找，尝试删除。
      iii. 若未找到，返回错误，否则，返回该节点的 size 与第一项的键值。
      iv. 该节点为内部节点，二分查找找到要下一步搜索子节点。
      v. 调用 `BPlusTree::InternalRemove()`。
      vi. 若未找到，返回错误。
      vii. 若子节点的 size 小于 `GetMinSize()`，尝试将子节点与其伙伴节点合并，或者从其伙伴节点中调换一个索引项到子节点中。
      viii. 返回该节点的 size 与第一项的键值。

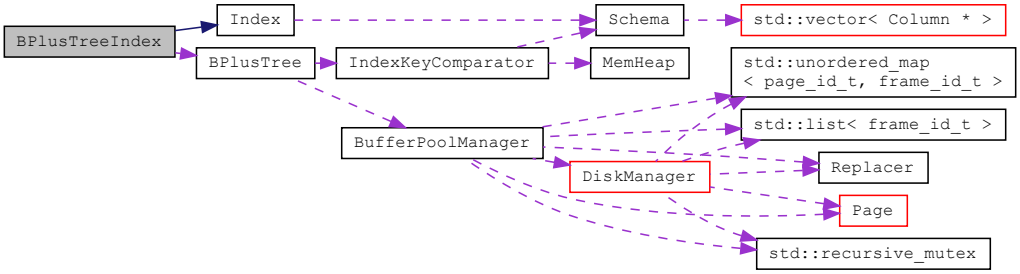
- `BPlusTreeIndex::Scankey`：读取一个索引项。
  1. 沿着 B+ 树自顶向下递归查找。在每个节点内部，采用二分查找算法。
- `BPlusTreeIndex::GetBeginIterator`：获取首迭代器。该迭代器的键值从最小或指定索引项开始递增，到尾迭代器结束。
  1. 二分查找。
- `BPlusTreeIndex::FindLastSmallerOrEqual(const Row & key)`：获取最后一个小于等于 key 的索引项的迭代器。
  1. 二分查找。
- `BPlusTreeIndex::GetEndIterator`：获取尾迭代器。

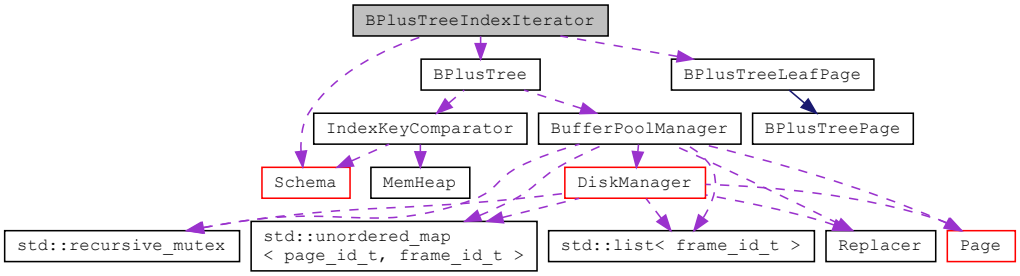
不同于 TableHeap 的迭代器，B+ 树的迭代器本身并不存储任何实际数据。这是由于 B+ 树迭代器解引用之后所得的数据与实际存储在内存中的数据一致，因此无需分配额外的空间。BPlusTreeIterator 解引用之后，返回的就是实际存储在叶子节点之中的索引项。迭代器成员如下：

- BPlusTree* tree：Iterator 所在的 B+ 树。
- BPlusTreeLeafNode *：Iterator 所在的叶子节点。在 Iterator 创建直到 End 或者析构期间，该节点对应的 page 始终被 pin.

- Schema * ：用于 BPlusTreeIteraotr::IsNUll()，判断该迭代器对应的索引项键值是否为空。

类关系图如下：





2.6 CatalogManager 模块

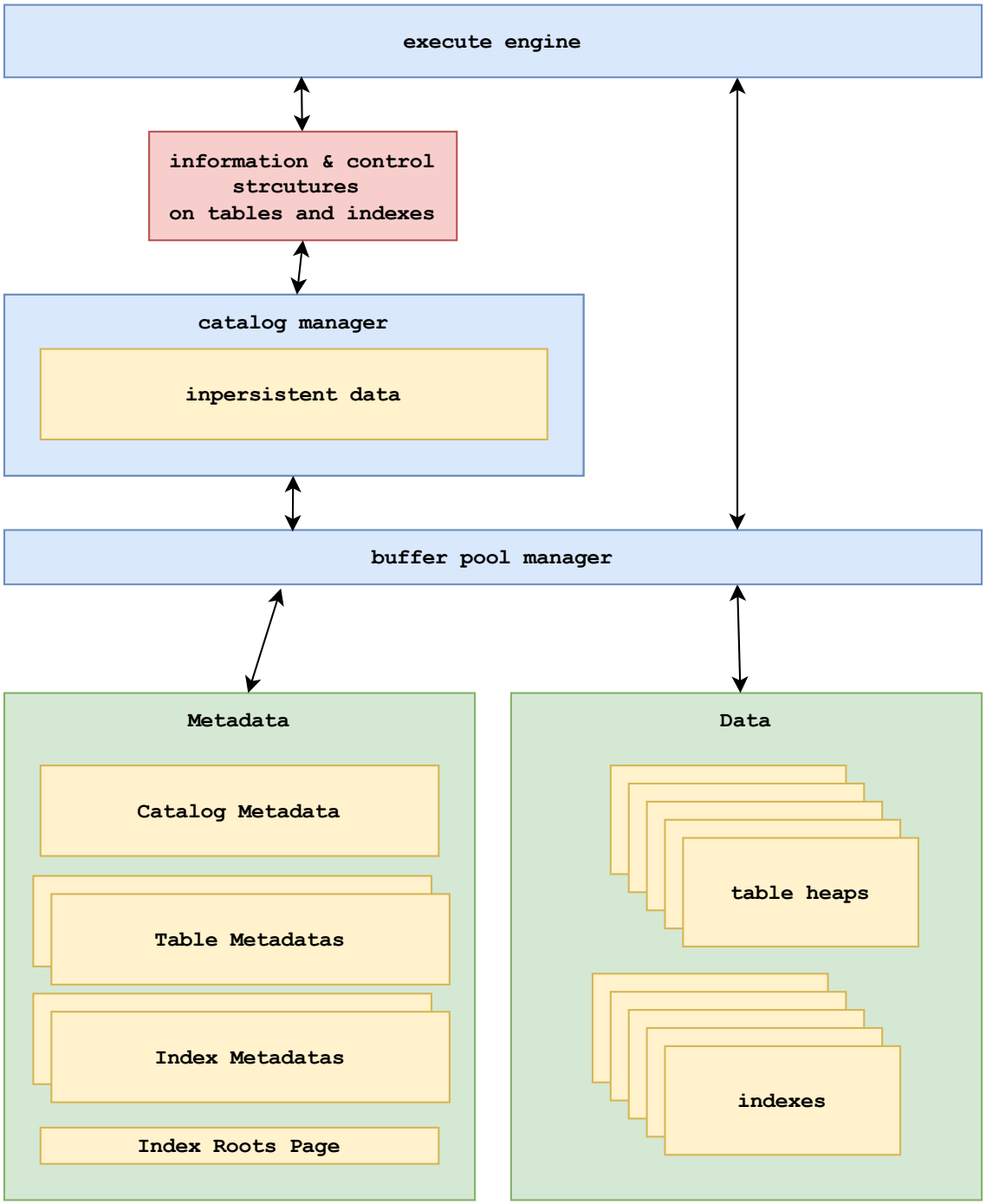
CatalogManager 管理了所有表、索引以及整个数据库的元信息，并把这些元信息与对应的数据结构关联起来，并为 Executor 提供查询、获取元信息与数据结构的接口。

具体的做法如下：ExecuteEngine 可以经由 CatalogManager 凭借 TableName , IndexName 获取表与索引的信息，也就是：IndexInfo 与 TableInfo。

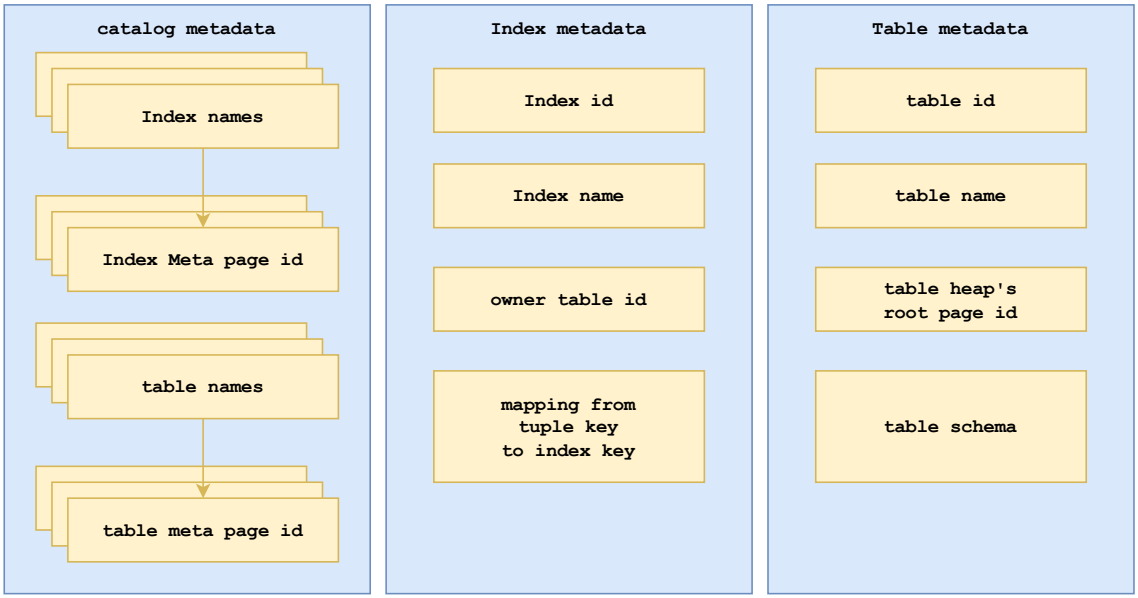
IndexInfo 中包含了 Index 的元信息：IndexMetaData，并且 IndexMetaData 直接存储在磁盘中。同理，TableInfo 包含了 Table 的元信息：TableMetaData。除此之外，IndexInfo 也包含了指向 BPlusTreeIndex 的指针，TableInfo 包含了指向 TableHeap 的指针。Executor 可以通过这些来判断一个索引或者表是否存在、列与索引的定义、以及操纵索引与表中实际存储的数据。

此外，CatalogManager 还管理了整个数据库中哪些表、哪些索引，以及索引的根节点信息。所有索引的根节点信息存储在一个单独的名为 `INDEX_ROOTS_PAGE` 的用户页中。

以上介绍的相互关系可以用下图表示。



CatalogMetaData 、IndexMetaData、TableMetaData 的结构如下图所示：



TableInfo 、IndexInfo 的数据成员如下：

```
1
2  class IndexInfo {
3   private:
4    IndexMetadata *index_meta_;
5    Index *index_;
6    TableInfo *table_info_;
7    IndexSchema *key_schema_;
8    MemHeap *heap_;
9  };
10
11 class TableInfo {
12  private:
13   TableMetadata *table_meta_;
14   TableHeap *table_heap_;
15   MemHeap *heap_; /** store all objects allocated in table_meta and table
         heap */
16 };
```

- CreateTable(const std::string &table_name, TableSchema *schema, Transaction *txn, TableInfo *&table_info) ： 创建表。
  1. 检查表是否重名，若重名，返回错误。
  2. 为表分配 root page 与 TableMetaPage
  3. 创建 TableMetaData 对象
  4. 创建 TableInfo 对象

5. 写入对应的页中
6. 更新 CatalogMeta 与 CatalogManager

- GetTable(const std::string &table_name, TableInfo *&table_info);

1. 检查表是否存在，若不存在，返回错误
2. 从 TableInfo 的 map 中取出 TableInfo *

- GetTables(std::vector<TableInfo *> &tables) const;

1. 遍历 TableInfo 的 map，逐个存入数组。

- `CreateIndex(const std::string &table_name, const std::string &index_name, const std::vector<std::string> &index_keys, Transaction *txn, IndexInfo *&index_info);`

1. 检查表是否存在，若不存在，返回错误。
2. 检查索引是否重名，若重名，返回错误。
3. 检查列是否在表中存在，若不存在，返回错误
4. 为索引分配 index root page 与 index meta page
5. 创建 IndexMetadata 对象
6. 创建 IndexInfo 对象
7. 更新 CatalogMeta 与 CatalogManager
8. 返回 IndexInfo * 对象的指针

- `GetIndex(const std::string &table_name, const std::string &index_name, IndexInfo *&index_info) const;`
- `GetTableIndexes(const std::string &table_name, std::vector<IndexInfo *> &indexes) const;`

1. 查询 map<std::string, map<std::string, index_id_t»，返回对应的全部 IndexId

- `DropTable(const std::string &table_name);`

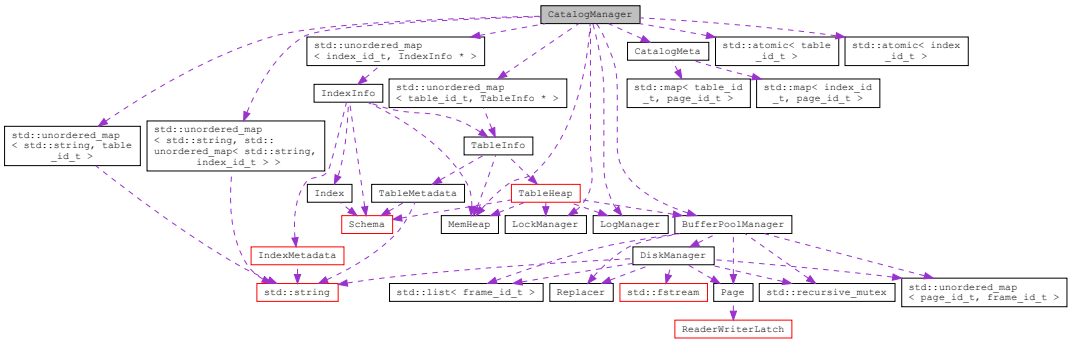
1. 检查表是否存在，若不存在，返回错误。
2. 调用 DropIndex，删除表上的全部索引
3. 调用 TableHeap->FreeHeap，删除堆表中的全部数据。
4. 释放 TableMetaPage。
5. 析构 TableHeap、TableMetaData、TableInfo 对象。

6. 更新 CatalogManager 与 CatalogMeta

- `DropIndex(const std::string &table_name, const std::string &index_name);`

1. 检查索引是否存在，若不存在，返回错误。
2. 调用 Index->Destroy，删除索引中的全部数据。
3. 释放 IndexMetaPage。
4. 析构 BplusTreeIndex,IndexMetaData,IndexInfo 对象。
5. 更新 IndexRootsPage。
6. 更新 CatalogManager 与 CatalogMeta。

类关系图如下：



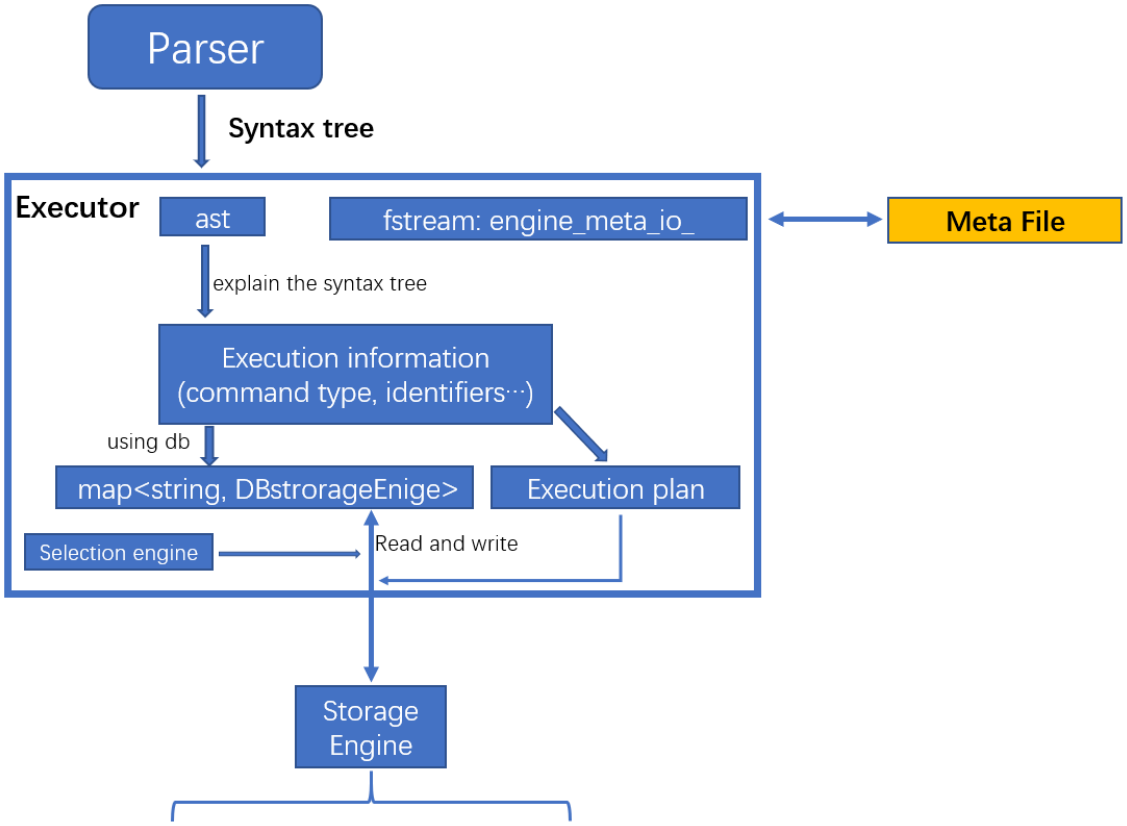
2.7 Parser 模块

Parser 模块的主要功能是根据用户的输入生成语法树 (syntax tree), 并将语法树的根节点作为参数传给 executor。语法树包含了一条 SQL 语句的全部可用信息，executor 根据语法树生成执行计划并输出。如果语法错误则输出错误信息。

Parser 模块由框架给出，今后可做更多改进，如支持 not null 声明、逻辑运算符优先级判断和括号，指定表的 drop index 等。

2.8 Executor 模块

2.8.1 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。

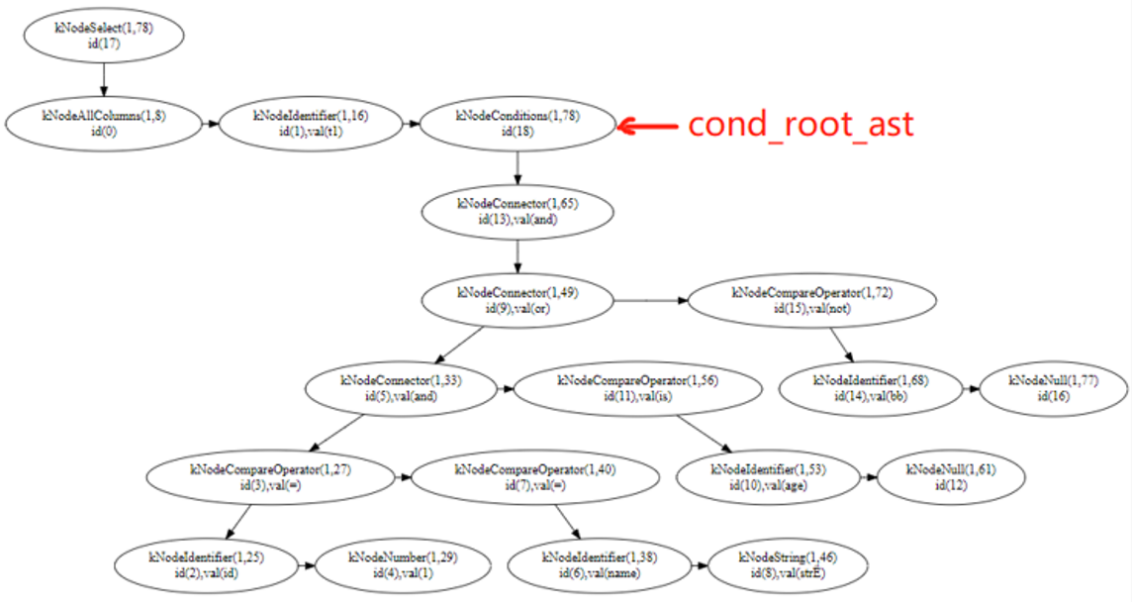
ExecuteEngine 对外接口只有 Execute 这个公共函数，接受的是 parser 生成的语法树根节点。另外，ExecuteContext 结构体可以起到向上层（主函数）返回执行结果的作用，目前主要作用是返回输出的文本字符串以及是否退出。

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。

2.8.2 条件筛选子模块

```
1 // my added member function (critical part)
2 // cond_root_ast: the root node for the Condition Node in syntax tree
3 // tinfo: the current selected table info
4 // iinfos: the indexes info of current table
5 // rows: receive the result
6 dberr_t ExecuteEngine::SelectTuples(const pSyntaxNode cond_root_ast,
      ExecuteContext *context, TableInfo *tinfo,
7                                    vector<IndexInfo *> iinfos,
8                                    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。

2.8.3 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

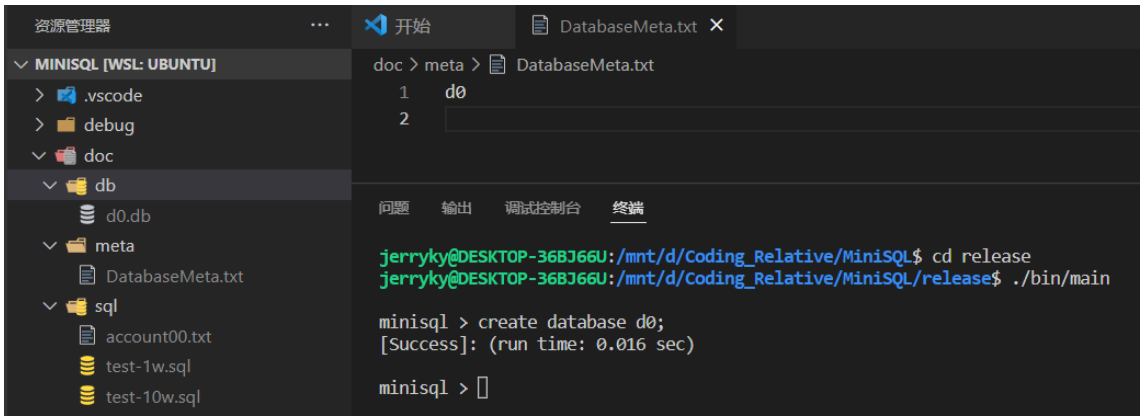
3. 关闭文件
4. 其中若文件中途有命令无效，则中途停止，不继续执行

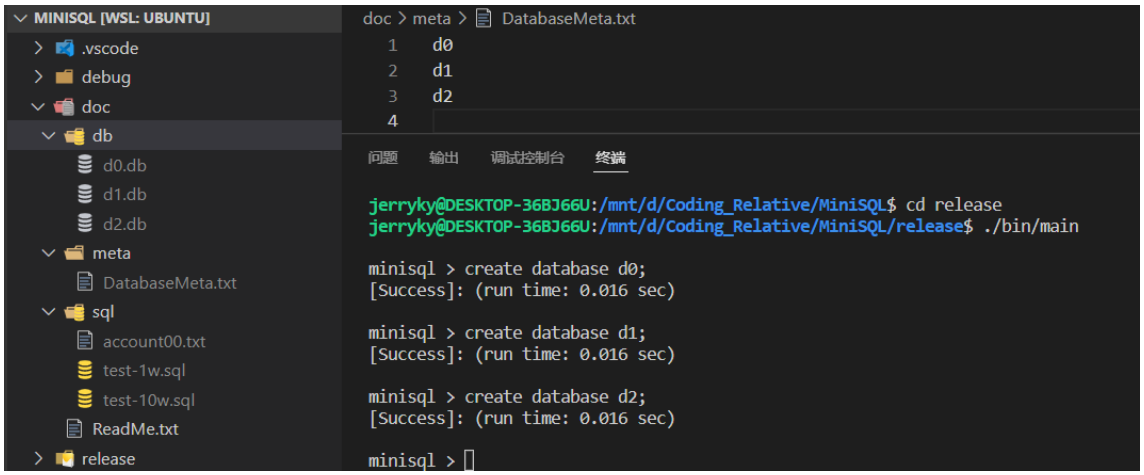
- `ExecuteQuit`：
  1. 将 context 的 flag_quit_ 标记为 true，代表结束

3 测试方案和测试样例

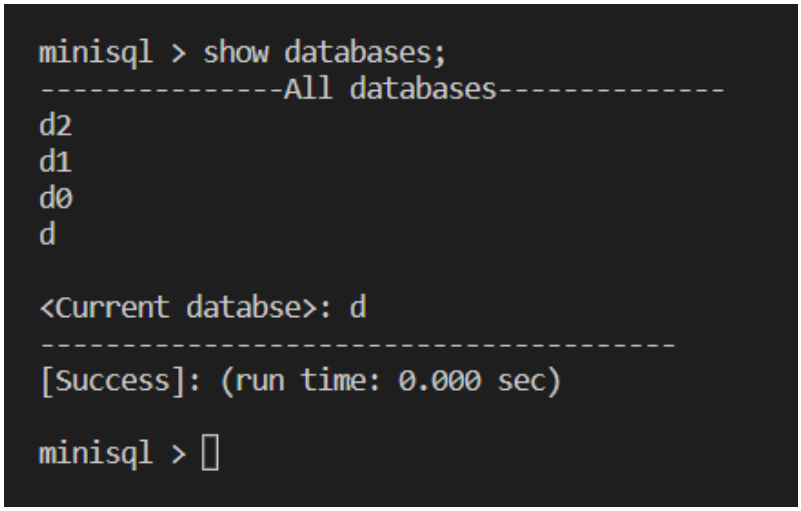
3.1 基本 SQL 语句测试

3.1.1 数据库的创建与显示



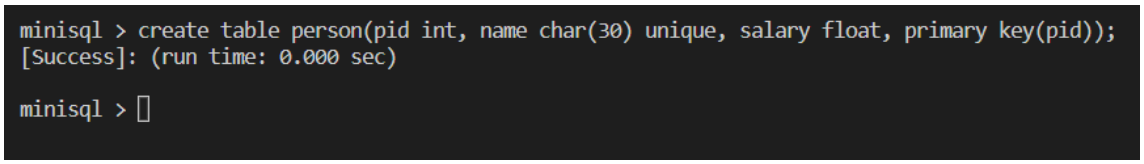


建表完成后，发现/doc/db 目录下多出 d0.db 的文件，并且元文件 DatabaseMeta.txt 中新出一行。创建多个数据库同理。



可以通过 show databases 查看当前所有数据库和在用数据库。

3.1.2 建表与显示



如图，建立一个以 int 型 pid 为主键，声明为 unique 的字符串型 name，float 型 salary，为三列的 person 数据表，建表成功。

```
minisql > create table t(a int);
[Warning]: Created table has no primary key!
[Success]: (run time: 0.000 sec)

minisql > show tables;
---------------All tables----------------
++++++++++++++++++++++++++
<Table name>
t
<Columns>
a               int
(1 columns in total)
<Row number>
0
<Indexes>
(No index)

++++++++++++++++++++++++++
<Table name>
person
<Columns>
pid             int
name            char(30)
salary                  float
(3 columns in total)
<Row number>
0
<Indexes>
_AUTO_UNIQUE_person_name_
_AUTO_PRI_person_pid_
(2 indexes in total)

++++++++++++++++++++++++++

(2 tables in total)
-------------------------------------
[Success]: (run time: 0.000 sec)
```

再建一个没有主键的表，会抛出警告提示没有主键，但允许建表。然后输入 show tables，可以查看所有表的信息。我们对 show tables 进行了改进，不仅是罗列表明，还要查看每个表的一些基本属性，如列名、列属性，当前行数，建在上面的索引名等。

```
minisql > update person set salary = 3000.6 where pid = 1;
[Note]: Using index "_AUTO_PRI_person_pid_" to select tuples!
(1 rows updated)
[Success]: (run time: 0.000 sec)

minisql > select * from person;
Table: person
pid  name  salary
1  Tom  3000.60010
2  John  1023.09998
(2 rows selected)
[Success]: (run time: 0.000 sec)

minisql > delete from person where name = "John";
[Note]: Using index "_AUTO_UNIQUE_person_name_" to select tuples!
(1 rows deleted)
[Success]: (run time: 0.000 sec)

minisql > select * from person;
Table: person
pid  name  salary
1  Tom  3000.60010
(1 rows selected)
[Success]: (run time: 0.000 sec)
```

3.1.3 插入记录与简单筛选

```
minisql > insert into person values(1, "Tom", 2000.5);
[Success]: (run time: 0.000 sec)

minisql > insert into person values(2, "John", 1023.1);
[Success]: (run time: 0.000 sec)

minisql > select * from person;
Table: person
pid  name  salary
1  Tom  2000.50000
2  John  1023.09998
(2 rows selected)
[Success]: (run time: 0.000 sec)
```

如图，执行简单的插入语句，显示插入成功，并可以通过 select * 语句验证表内容。

3.1.4 更新与删除记录

```
minisql > execfile "account00.txt";
```

如图，执行更新、删除语句，并 select 验证，可以发现数据已经被修改。这里 [Note] 提示用户目前使用索引进行查找，这里使用的是自建的 AUTO 索引

3.1.5 执行文件

为了更好测试其他功能,先验证执行文件功能。文件 account00.txt 位于/doc/sql 目录下，含有建表并插入 1 万行，建立索引的操作。

```
[Execute]: insert into account values(12509998, "name9998", 623.28);
[Success]: (run time: 0.000 sec)

[Execute]: insert into account values(12509999, "name9999", 86.27);
[Success]: (run time: 0.000 sec)

(10001 commands in file executed successfully)
[Success]: (run time: 0.641 sec)

minisql >
```

```
minisql > select * from account;
```

每条被执行的记录会得到显示，并在最终输出成功执行的总语句条数和总用时（可以看到速度很快）。使用 select 语句验证执行成功。

```
12509998  name9998  623.28003
12509999  name9999  86.27000
(10000 rows selected)
[Success]: (run time: 0.062 sec)
```

```
minisql > select balance, id from account where id <= 12501000 and balance > 995 and name <> "name555";
[Note]: Multiple conditions!
Table: account
balance  id
998.60999  12500186
998.88000  12500257
999.34998  12500554
995.59998  12500669
998.53003  12500686
995.15997  12500813
999.53003  12500884
996.03003  12500968
(8 rows selected)
[Success]: (run time: 0.000 sec)
```

（1 万行无法截全）

3.1.6 复杂筛选

简单的全选操作以及在插入更新删除同时进行了展示。下面还需要进行多条件、投影操作的复杂筛选验证。对于刚刚通过文件生成的 1 万行表 account，我们执行复杂筛选语句。

```
minisql > insert into account values(12510000, "name10000", null);
[Success]: (run time: 0.000 sec)

minisql > select * from account where balance is null or balance > 999.8;
[Note]: Multiple conditions!
Table: account
id  name  balance
12502438  name2438  999.84003
12508413  name8413  999.91998
12510000  name10000  null
(3 rows selected)
[Success]: (run time: 0.016 sec)
```

可以看到按要求输出了对应结果，包括结果总数。

3.1.7 null 有关

null 的插入和筛选同样有效，在 account 表上举例。

```
minisql > select * from account where balance <= 0.3;
Table: account
id  name  balance
12501466  name1466  0.17000
12509109  name9109  0.24000
12509512  name9512  0.06000
(3 rows selected)
[Success]: (run time: 0.016 sec)
```

可以看到 balance 为 null 的一行成功插入并被成功筛选。注意 null 在范围查找中不会被比较，如此时对 balance 进行范围查找，无论是小于大于还是不等于某个数字，null 一行均不会被选中，只有 is 和 not 运算符才对 null 有效。

```
minisql > show indexes;
--------------All indexes on tables---------------
Table name: account:
_AUTO_PRI_account_id_

Table name: t:
(No index)

Table name: person:
_AUTO_UNIQUE_person_name_
_AUTO_PRI_person_pid_

-------------------------------------------------
[Success]: (run time: 0.000 sec)
```

3.2 索引功能测试

3.2.1 AUTO 索引的自动建立

MiniSQL 中，数据库会对声明为 unique 和 primary key 的列自动建立 AUTO 类型的索引。通过 show indexes 可以查看所有索引。

```
minisql > select * from account where name = "name5678";
Table: account
id  name  balance
12505678  name5678  235.94000
(1 rows selected)
[Success]: (run time: 0.016 sec)

minisql > create index idx_name on account(name);
[Warning]: Creating index on a field without uniqueness declaration. Make sure no duplicate keys!
[Success]: (run time: 0.078 sec)

minisql > select * from account where name = "name5678";
[Note]: Using index "idx_name" to select tuples!
Table: account
id  name  balance
12505678  name5678  235.94000
(1 rows selected)
[Success]: (run time: 0.000 sec)
```

可以看到之前所建立的表的 AUTO 索引。

3.2.2 单键索引的建立与加速查询

为了加速通过 account 的 name 进行查询，我们可以在 name 上建立索引，并比较查询速度。

```
minisql > select * from account where name >= "name9994";
[Note]: Using index "idx_name" to select tuples!
Table: account
id  name  balance
12509994  name9994  340.82001
12509995  name9995  181.25999
12509996  name9996  788.87000
12509997  name9997  119.38000
12509998  name9998  623.28003
12509999  name9999  86.27000
(6 rows selected)
[Success]: (run time: 0.000 sec)
```

可以看到，同样是查询一条记录，不用索引耗时 0.016 秒（线性扫描堆表），而，建立索引后，通过索引查询会给出提示，并且速度极快。同样，也可以利用索引进行范围查询。

```
minisql > create index idx_name_id on account(id,name);
[Warning]: Creating index on multiple-field key. Make sure no duplicate keys!
[Success]: (run time: 0.078 sec)
```

索引会定位到比较值”name9994”，然后通过索引迭代器返回后面的值。即使”name9994”不存在，索引也可以通过返回仅次于目标值的迭代器加速范围查询。

3.2.3 多键索引的建立

允许建立多键索引，但因为多条件查询执行计划较复杂，MiniSQL 暂未实现利用多键索引加速查询的功能。但是可以用其约束唯一性。

3.2.4 索引的删除

MiniSQL 支持删除自定义索引。

```
minisql > drop index idx_name_id;
[Success]: (run time: 0.000 sec)

minisql > drop index idx_name;
[Success]: (run time: 0.000 sec)

minisql > select * from account where name = "name5678";
Table: account
id  name  balance
12505678  name5678  235.94000
(1 rows selected)
[Success]: (run time: 0.016 sec)
```

可以看到删除 name 上的索引后，查询速度又变慢了。(但是过多索引会提高维护成本，降低表内容更改的效率)

MiniSQL 不允许删除 AUTO 索引，这是出于维护限制约束的需要。

```
minisql > drop index _AUTO_PRI_account_id_
;
[Rejection]: Can not drop an AUTO key index!
[Failure]: SQL statement executed failed!
```

3.3 限制约束测试

3.3.1 索引键值唯一性限制

对于声明为主键或 unique 的列，或自定义索引的列，其上不允许有重复键值。如果插入、更新操作会导致出现重复键值，会被拒绝。创建一个简单的表 t(a, b, c, d) 为例。

```
minisql > create table t(a int, b int unique, c int, d int, primary key(a));
[Success]: (run time: 0.000 sec)

minisql > insert into t values(1, 1, 1, 1);
[Success]: (run time: 0.000 sec)

minisql > insert into t values(1,2,2,2);
[Rejection]: Inserted row may cause duplicate entry in the table against index "_AUTO_PRI_t_a_"!
[Failure]: SQL statement executed failed!

minisql > insert into t values(2,1,2,2);
[Rejection]: Inserted row may cause duplicate entry in the table against index "_AUTO_UNIQUE_t_b_"!
[Failure]: SQL statement executed failed!
```

可以看到第二、三次插入分别因为违反主键唯一性、唯一键唯一性而被拒绝。同样，可以验证更新时的限制和多键索引限制。

```
minisql > insert into t values(2,2, 2, 2);
[Success]: (run time: 0.000 sec)

minisql > select * from t;
Table: t
a  b  c  d
1  1  1  1
2  2  2  2
(2 rows selected)
[Success]: (run time: 0.000 sec)

minisql > create index idx_cd on t(c,d);
[Warning]: Creating index on multiple-field key. Make sure no duplicate keys!
[Success]: (run time: 0.000 sec)

minisql > update t set c = 1, d = 1 where a = 2 and b = 2;
[Note]: Multiple conditions!
[Rejection]: Updated row may cause duplicate entry in the table against index "idx_cd"!
[Failure]: SQL statement executed failed!
```

可以看到在 c，d 上建立多值索引后，如果更新导致出现重复键值，同样会被拒绝。

3.3.2 自定义索引的检查

前面说过 MiniSQL 支持自定义索引，但需要保证键值唯一性。
首先，如果自定义的索引跟已存在的所有存在等价关系，则会报警告。

```
minisql > create index idx_b on t(b);
[Warning]: Index being created is equivalent to the existed index "_AUTO_UNIQUE_t_b_" !
[Success]: (run time: 0.000 sec)
```

再一个，如果建立的索引上有重复键值会被拒绝。

```
minisql > insert into t values(3,3,3,1);
[Success]: (run time: 0.000 sec)

minisql > select * from t;
Table: t
a  b  c  d
1  1  1  1
2  2  2  2
3  3  3  1
(3 rows selected)
[Success]: (run time: 0.000 sec)

minisql > create index idc_c on t(c);
[Warning]: Creating index on a field without uniqueness declaration. Make sure no duplicate keys!
[Success]: (run time: 0.000 sec)

minisql > create index idx_d on t(d);
[Warning]: Creating index on a field without uniqueness declaration. Make sure no duplicate keys!
[Error]: Can not create index on fields which already have duplicate values!
[Failure]: SQL statement executed failed!
```

可以看到 c 上无重复键值，自定义索引只会提示警告，而若有重复键值，则会报错，不会建立这个索引。

3.3.3 主键非 null 限制

类比 mySQL 的设定，MiniSQL 同样不允许主键的列中含有 null 值。

```
minisql > insert into t values(null, 4,4,4);
[Rejection]: Can not assign "null" to a field of primary key while doing insertion!
[Failure]: SQL statement executed failed!
```

3.4 系统稳定性测试

3.4.1 简单的异常输入处理举例

输入的字符串长度不符合要求：

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

筛选的列名不存在：

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

3.4.2 异常中断退出时数据库信息的保持

为了防止段错误、强制退出等操作导致 buffer pool 中的页没有及时 flush 回磁盘导致数据丢失，我们通过中断信号函数来强制 flush。在插入过程中中途退出，发现已经插入的数据不会丢失。

```
minisql > execfile "account00.txt";

[Execute]: create table account(
  id int,
  name char(16),
  balance float,
  primary key(id)
);
[Success]: (run time: 0.000 sec)

[Execute]: insert into account values(12500000, "name0", 514.35);
[Success]: (run time: 0.000 sec)
```

```
[Execute]: insert into account values(12500118, "name118", 374.38);
[Success]: (run time: 0.000 sec)

[Execute]: insert into account values(12500119, "name119", 83.72);
[Success]: (run time: 0.000 sec)

^Z


[Exception]: Forced quit!
```

```
[Exception]: Forced quit!
jerryky@DESKTOP-36BJ66U:/mnt/d/Coding_Relative/MiniSQL/release$ ./bin/main

minisql > use d0;
[Success]: (run time: 0.000 sec)

minisql > select * from account;
Table: account
id  name  balance
12500000  name0  514.34998
12500001  name1  103.14000
12500002  name2  981.85999
```

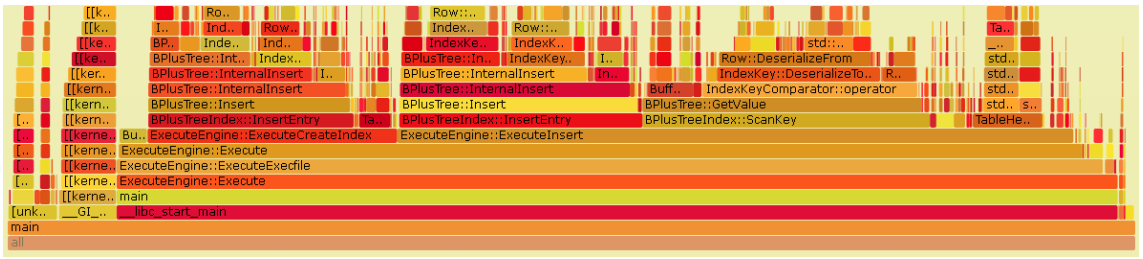
当然，这种中断处理方式有局限性，且配合事务使用较好，目前只是雏形。未来用日志实现会更好。

3.5 性能分析

由于 Release 模式下无法获取堆栈调用信息，且较多函数被内联优化，本小节的火焰图均在 Debug 构建模式下测得。同时由于 perf 的存在，实际运行时间会略小于所测得时间。

3.5.1 Insert

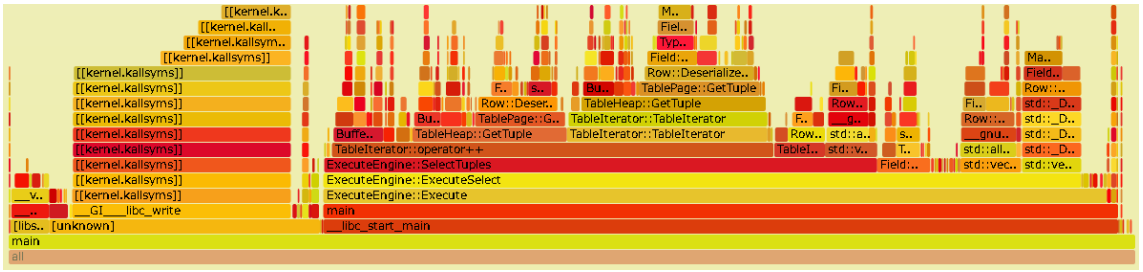
```
create database d;
use d;
execfile "test-10w.sql";
quit;
```



时间：23.11s(Debug) 9.49s(Release)

3.5.2 Select，无条件

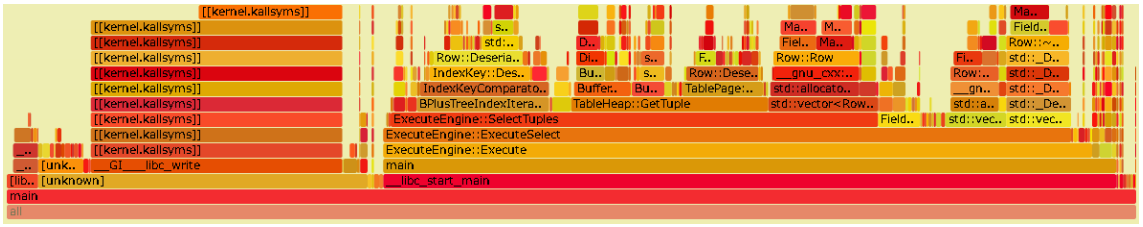
```
use d;
select * from person;
quit;
```



时间：0.821s(Debug) 0.281s(Release)

3.5.3 Select，单条件，有索引

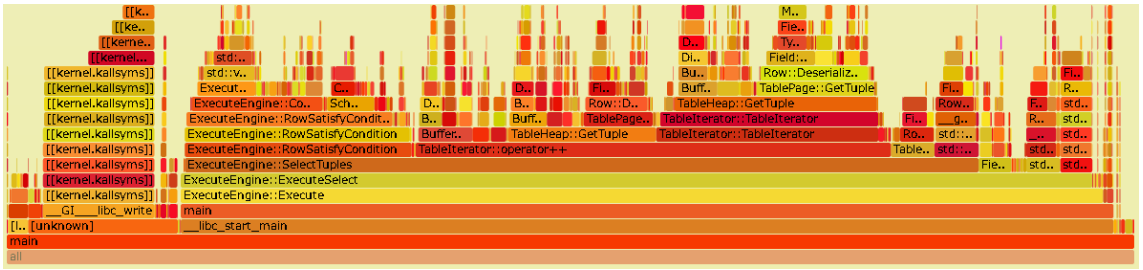
```
use d;
select * from person where pid > 50000;
quit;
```



时间：0.342s(Debug) 0.254s(Release)

3.5.4 Select，多条件，有索引

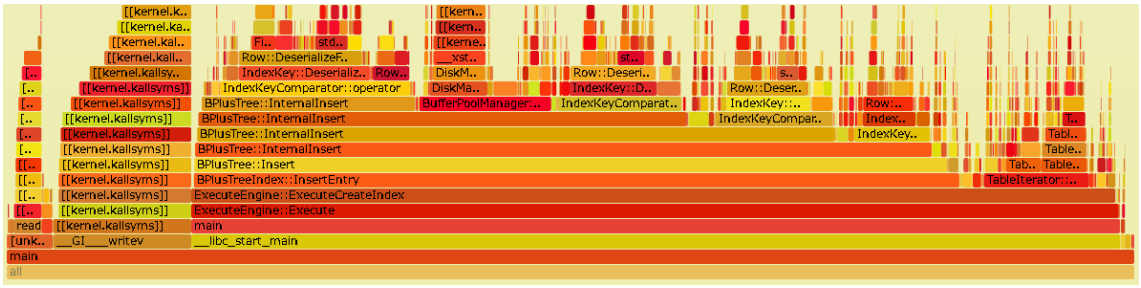
```
use d;
select * from person where pid > 25000 and age < 75000;
quit;
```



时间：0.770s(Debug) 0.141s(Release)

3.5.5 已有表 Create Index

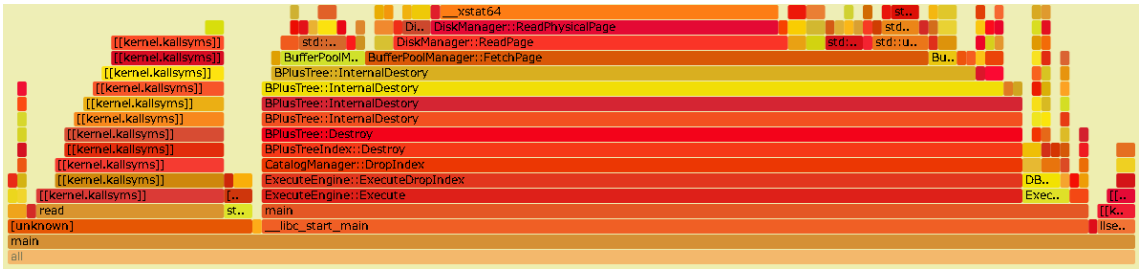
```
use d;
create index idx_temp on person(identity);
quit;
```



时间：5.285s(Debug) 2.198s(Release)

3.5.6 Drop Index

```
use d;
drop index idx_temp;
quit;
```



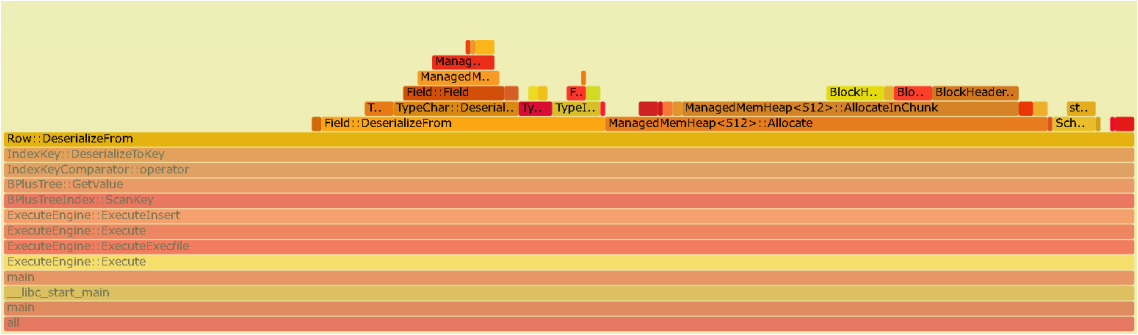
时间：0.079s(Debug) 0.069s(Release)

3.5.7 Drop Table

```
use d;
drop table person;
quit;
```

时间：0.221s(Debug) 0.207s(Release)

3.5.8 Row::DeserializeFrom



3.5.9 总结

程序运行 90% 的时间都花在了行的序列化与反序列化上。

4 设计亮点与 Bonus

4.1 堆表的插入优化

框架提出的线性扫描插入方式效率低、与 LRU 策略相悖，我们使用最大堆的数据结构优化，大大提升了插入，尤其是连续插入的速度。具体原理见第 2.4.3 一节。

4.2 Replacer 的优化与多种 Replacer 的实现

4.2.1 LRUReplacer 的优化

在普遍的实现中，LRUReplacer 的算法如下：

- 对于每个 frame, 维护 lastUsedTime, 代表该 frame 上一次被访问的时间。
- 维护一个 list, 包含了可以被替换的 fid
- `UnPin(frame_id_t fid)`：
  1. 将所有在 list 中的 fid 对应的 lastUsedTime 加 1
  2. 将 fid 对应的加入 list，并设置 lastUsedTime 为 0
  3. 时间复杂度：$O(N)$
- `Pin(frame_id_t fid)`：
  1. 将 fid 从 list 中移除。

2. 时间复杂度：$O(1)$

- `Victim()`：

  1. 遍历所有 list 中的 fid
  2. 找到 lastUsedTime 最大的 fid
  3. 从 list 中移除 fid，并返回 fid
  4. 时间复杂度：$O(N)$

由于 `Unpin` 和 `Pin` 函数的使用极为频繁，因此 `Unpin` 的复杂度为 $O(N)$ 是不可接受的。因此，本项目放弃原框架中 `unordered_set` 实现，改用数组实现，并提供了以下等价的优化算法：

- 对于每个 frame, 维护 lastUsedTime, 代表该 frame 上一次被访问的时间。
- 维护一个 list, 包含了可以被替换的 fid
- 维护 min_time, 表示所有 lastUsedTime 中的最小值。
- `UnPin(frame_id_t fid)`：

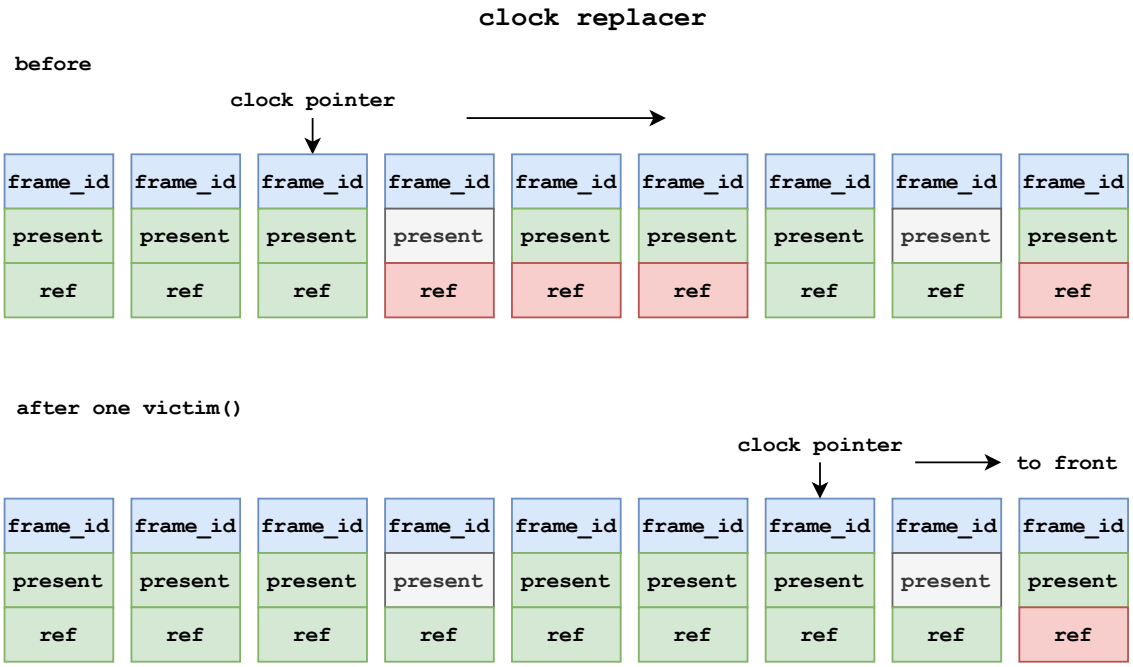
  1. 将 fid 对应的加入 list，并设置 lastUsedTime 为 min_time - 1
  2. 若发生溢出，则把所有的 lastUsedTime 加 1。
  3. 均摊时间复杂度：$O(1)$

- `Pin` 与 `Unpin` 原理与之前的大致相同。

4.2.2 ClockRepalcer 的实现

Clock Replacer, 顾名思义，像钟表一样的 replacer。Clock replacer 维护一个环形数组，其中的 Clock pointer 在每次 victim() 被调用时，循环地向后寻找第一个未被访问并且 present 的项，并返回该项。

简单原理图如下，ref 表示是否已被访问。每次 clock pointer 扫描到一项时，就将 ref 设置为 false。



后续的测试标明,clock replacer 的性能与 lru replacer（优化后）相当，都远大于优化之前的 lru replacer, 这是由于 clock replacer 的 unpin 、pin 操作都是 $O(1)$ 的。

具体实现如下:

- `Victim`:
  1. 从 clock pointer 向后遍历所有项，若达到结尾则返回开头。
  2. 寻找第一个 ref 为 false 且 present 的项，同时把经过的项的 ref 设置为 false。
- `Unpin`:
  1. 将该项对应的 ref 设为 true
  2. 将该项对应的 present 设为 true
- `Pin`:
  1. 将该项对应的 present 设为 false

4.3 Index 的重构和去模板化

在原项目框架中，`BPlusTreeIndex`、`BPlusTreeIndex` 为模板类，接受三个模板参数 `KeyType`，`ValueType`，`KeyComparator`，功能分别为:

- `KeyType`： 键的类型

- `ValueType`：值的类型
- `KeyComparator`：用于比较键的类

这种设计存在多种缺点。其中部分缺点是：

1. 浪费空间。`KeyType` 只支持特定长度的 key（在原框架中，这些长度分别是 4,8,16,32,64）。这意味着对于长度略大于 2 的次幂的那些 key，将会有接近一半的存储空间被浪费。
2. 灵活性差。key 具体需要多少长度，在运行期间是未知的，这就要求我们动态地根据 key 的长度对模板类进行实例化，破坏了封装性与代码的整洁。其次，模板类不可能实例化所有可能的长度，因此能够支持的 key 长度存在上限。
3. 代码膨胀。如果要对 Keysize 进行更加细粒度的划分，将不可避免的实例化大量的模板类，这将造成巨幅的代码膨胀。

当然，这种设计也存在一些优点，比如代码编写比较容易。

因此，本项目放弃了模板的设计，改用单一的 `BPlusTree` ，`BPlusTreeIndex` ，`BPlusTreePage` 类实现 B+ 树的操作，提高了灵活性，且节省了空间。具体做法是：

1. 将 keysize 作为成员保存在 `BPlusTreePage` 中。该成员在 `BPlusTreePage` 被创建时初始化。
2. 将 max_size 作为成员保存在 `BPlusPage` 中，代表该 Page 最多能保存的 Entry 数量。该成员在 `BPlusTreePage` 被创建时初始化。
3. 将 `BPlusTreeLeafPage` 的 `ValueType` 固定为 `RowId`
4. 将 `BPlusTreeInternalPage` 的 `ValueType` 固定为 `page_id_t`
5. 将 `KeyType` 固定为 `IndexKey`，`IndexKey` 的成员如下：

```
1 struct IndexKey{
2   uint8_t keysize;
3   char value[0]; // 柔性数组
4 }
```

6. GenericComparator 同时也去掉模板参数，变为 IndexKeyComparator。由于 IndexKey 中包含了 keysize 成员，key 的长度已知，因此可以进行比较。

`BPlusPage` 成员如下：

```
1  class BPlusTreePage{
2    // ...
3    private:
4     IndexPageType page_type_;
5     lsn_t lsn_;
6     int key_size_; //键长度
7     int size_; //当前键数
8     int max_size_; //最大键数
9     page_id_t parent_page_id_;
10    page_id_t page_id_;
11 };
```

相关内存布局如下：

IndexKey

key_size_	value_ (variable length of key data ...)

BPlusTreePage(the header part)

page_type_	lsn_	key_size_	size_	max_size_	parent_page_id_	page_id_	...

BPlusTreeLeafPage

header	next_page_id_	IndexKey 0	RowId 0	IndexKey 1	RowId 1	...

BPlusTreeInternalPage

header	IndexKey 0	page_id_t 0	IndexKey 1	page_id_t 1	...

4.4 内存管理机制的多种优化

4.4.1 内存管理机制的优化

基于原框架中 MemHeap 的思想，本项目进一步优化了对内存的管理。

在本项目中，涉及到大量类对象的动态创建与销毁，内存管理是一个十分重要的方面。这是由于：

- 如果对动态创建的对象的生命周期管理不当，很容易造成内存泄露，尤其是在处理万级别的数据量时，使得程序无法持续运行。
- 大量临时对象的创建与销毁，若采用原始的内存分配方式，将导致极大的性能开销。
- 原有的 SimpleMemHeap 性能不足。

基于 MemHeap 的内存管理方式，我们提出以下两个概念：

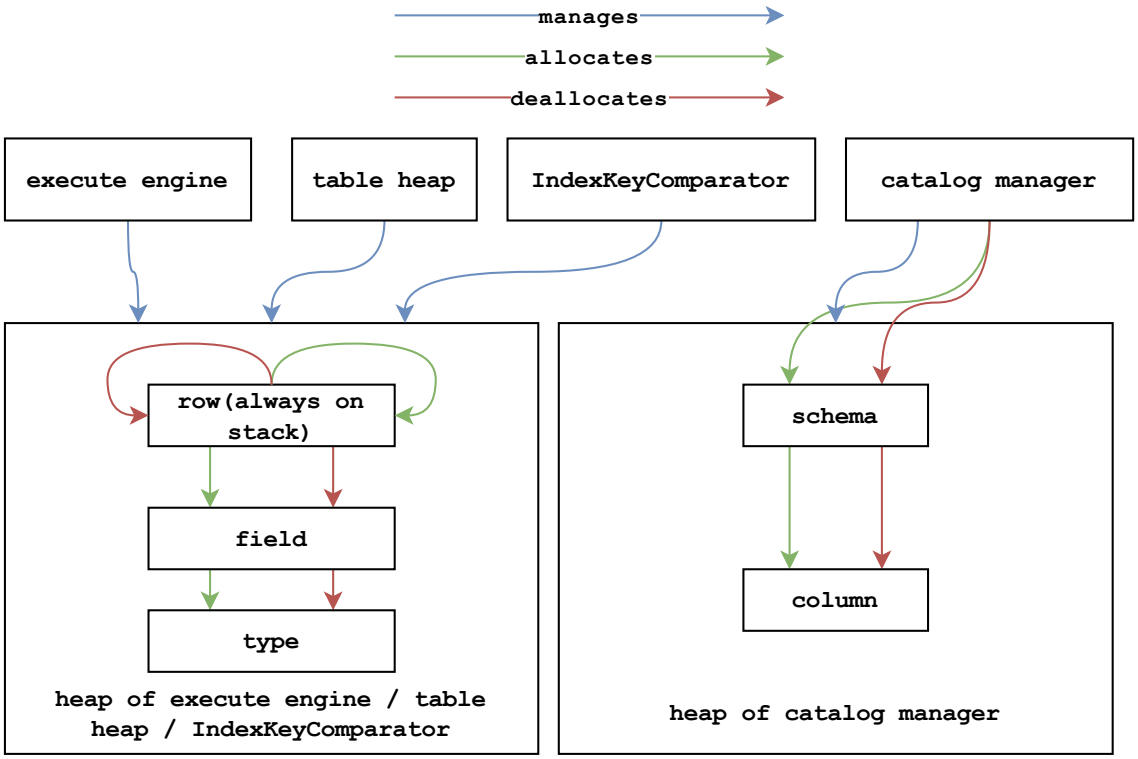
- “主对象”：成员包括 MemHeap，在 heap 上分配或者销毁自身的成员，且负责 MemHeap 的创建与销毁。
- “从对象”：成员包括 MemHeap，在 heap 上分配或者销毁自身以及自身的生源，但无权创建或者销毁 Memheap 的对象。
- 对于无需堆方式内存管理的对象，不在考虑范围之内。

之所以有这样的划分，是由于 memheap 的创建与销毁具有一定的性能开销。若 memheap 也由临时对象维护，那么频繁的创建与删除 memheap（比如原框架中 row 对象）将占用大量时间，且无法复用已分配的内存。因此，只有那些能够在内存中驻留较长时间的对象，才能对 memheap 进行管理。

基于以上概念，我们做出以下划分：

- 主对象：`ExecuteEngine`,`___Manager`,`TableHeap`,`IndexKeyComparator`
- 从对象：`Schema`,`Column`,`Row`,`Field`,`Type`

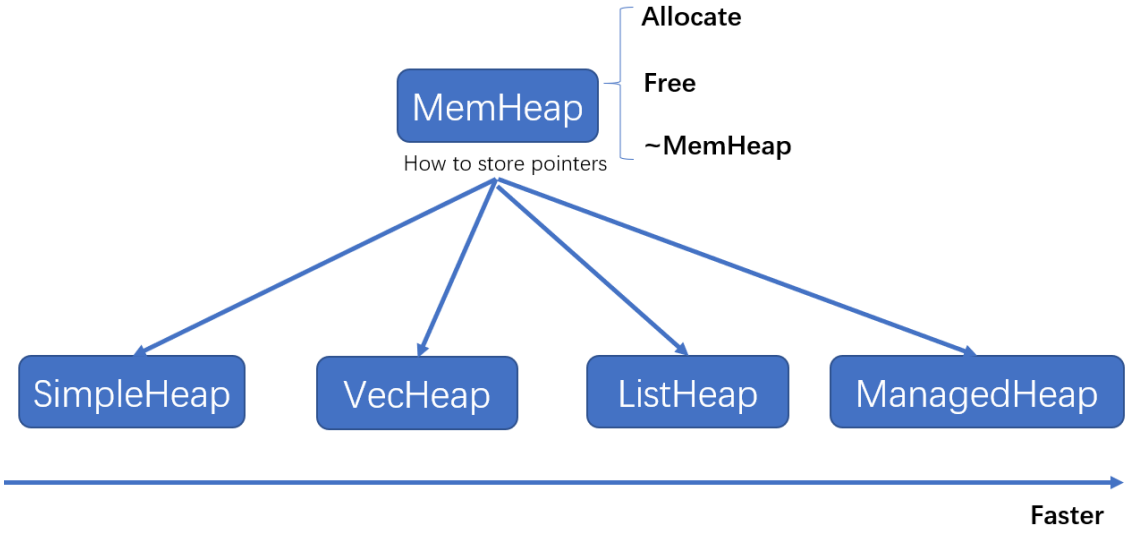
这样一来，便形成了如下的内存管理关系图：



最终，本项目实现了 0 内存泄露。（仅有的内存泄漏来自于 googletest，此外，还修复了 parser 模块中部分内存泄露。）

当然，每次在比较 key 的时候都对 Row 进行 deserialize，以及 row 对象的频繁创建与销毁，依然占据了除去文件读写以外 80% 的时间。在一个数据库系统之中，是否应该对于同一个 record 创建如此多的临时拷贝，并且通过反序列化的方式来进行比较，我认为是一个值得思考的问题。

4.4.2 高效内存池的实现



此外，本项目实现了一个高效快速的内存池：ManagedHeap。

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）

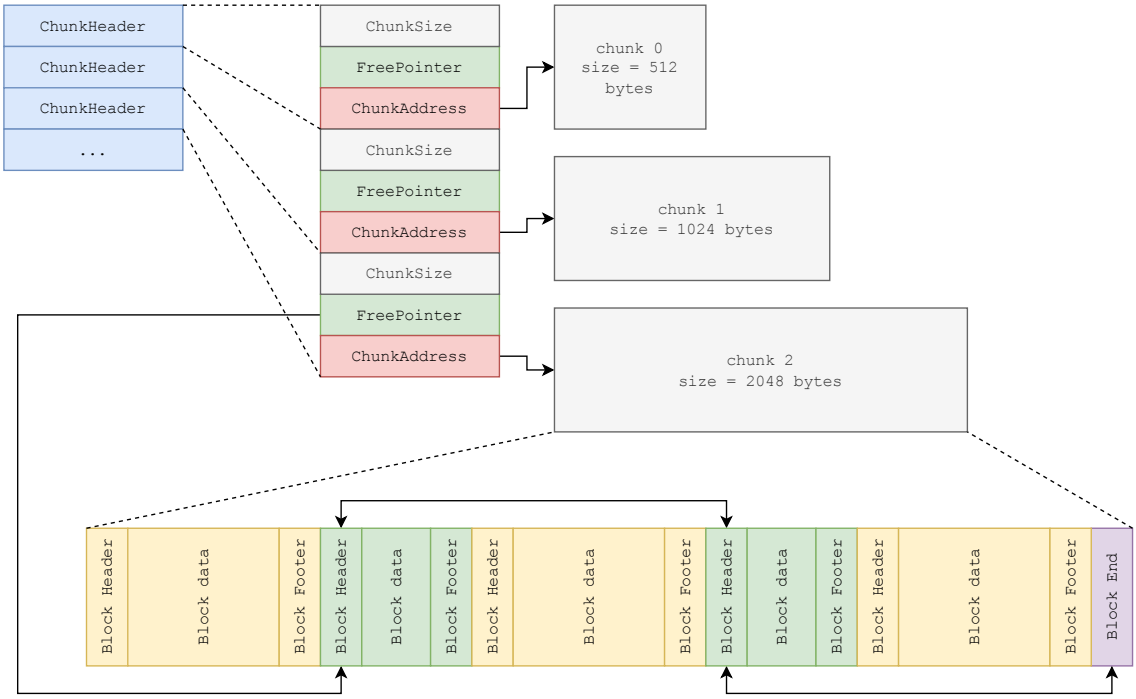
ListHeap：但是 VecHeap 的性能仍不是很理想，其 push_back 依然占据了程序执行的大部分时间（约 60% 80%），后来我们发现，由于在 MiniSQL 中大部分 Allocate 的场景是 row 的 MemHeap 去维护它的每个 field 的空间，而 field 的数量通常不会很大（个位数，或几十个），在这种较少的数据量下，使用 STL 提供的 vector 或 set 来维护可能显得臃肿而没必要。因此我们手写了简单的链表用来存储指针，使得插入复杂度为 O(1)，而删除则需要遍历。在这种轻量级的数据结构下，速度得到了大幅提升，debug 模式下插入带索引的 10 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

ManagedHeap：

设计思想

在数据库运行的过程中，会有大量临时的 Row 对象与 Field 对象不断被生成或者销毁。对于这些临时的对象，无需再重新调用 operaotr new –delete 或者 malloc - free 进行内存的申请和释放，这些 system call 本身的性能开销就比较大。只需要重复利用之前已经被申请过的内存即可。ManagedHeap 中先前被分配的内存在被释放时，该内存块不再被返还给操作系统，而是被 ManagedHeap 标记为 free，后续内存的分配可以直接复用这块内存，大大降低了内存申请与释放的开销。

总体架构



ManagedHeap 由若干个 Chunk 组成，每个 Chunk 代表一块较大的内存。相邻 Chunk 的大小以 2 为倍数倍增。在每个 Chunk 中，内存被线性地从前往后划分为若干个 Block，每个 Block 的状态为已分配或空闲。在每个 Block 中，首部和尾部各自维护了一个完全相同的 BlockHeader，记录了该 Block 的分配信息，其中包括：

- 该 Block 的大小
- 该 Block 是否已被分配
- 若该 Block 未被分配，记录上一个 free Block 相对 chunk 起始的偏移
- 若该 Block 未被分配，记录上一个 free Block 相对 chunk 起始的偏移

ChunkHeader 数组维护了每一个 Chunk 的信息，其中包括：

- Chunk 起始地址

- Chunk 大小，以字节计
- 该 Chunk 中第一个 free block

这样设计的优点在于，不仅能通过链表的结构快速的获取到空闲的块，还能根据某一个块的地址快速获取到相邻块的信息，从而完成块的分割或者合并。

ManagedHeap 的分配与释放实现如下：

- `Allocate(size)`：
  1. 计算能够容纳 size 大小的第一个 chunk。
  2. 从该 chunk 的第一个 free block 开始，沿着链表查找，若未找到则从下一个 chunk 中继续，直到找到满足大小的 free block。
  3. 若该 block 的大小显著大于 size，则对 block 进行分割，将新产生的空闲 block 加入 free list 中。
  4. 将该 block 从双向链表中删除，并对 chunk 的 free pointer、链表中前后 Block 头中的指针进行更新。
- `Free(void * p)`：
  1. p1 = p - sizeof(BlockHeader)，计算得控制块的地址。
  2. 检查魔数，判断该块内存是否由本 heap 分配。若否，抛出错误。
  3. 将该 chunk 的 free pointer 设置为 p1, 并把该块标记为 free。
  4. 将该块的相邻的也为 free 的 block 与自身合并，并更新相关的链表指针。

容易看到，`ManagedHeap` 无论是 `Allocate` 还是 `Free` 都只需要 $O(1)$ 的时间复杂度。并且在打开 release 优化之后，ManagedHeap 的运行速度得到了更大幅度的提升。采用 ManagedHeap 之后，插入带索引的 10 万行记录时间减少到了 10 秒左右。

4.5 中断的处理

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);
```

具体测试已经在测试一节给出。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，以后可以用日志来恢复数据。

5 框架建议

- 索引迭代器机制值得商讨，模板化导致上层需要本不应该知道的额外信息 (B+ 树的节点大小) 才能访问索引，导致封装性不好 (模板化 B+ 树是否有必要？)
- 任何值（包括非 NULL）与 NULL 对比皆返回 kNULL，不方便测试，NULL 和 NULL 相比可多返回一种结果
- catalog 的 createindex 并没提供索引类型有关的参数
- drop index 只指定 index 名、不指定表名，略不合理，应该指定表名
- column 有 nullable 成员，解释器也应该支持 not null 声明。
- int 和 float 的 type 也可以提供 GetData，便于获取数据。
- 每张表自身都不知道其索引信息，必须贼 executor 里手动同步，是否合理? 能否让 index 依赖于 table?
- 框架中应增加管理数据库文件信息的接口（在 ExecuteEngine 里）。

6 分组与设计分工

本组成员（2 人）
姓名：张峻瑜 学号：3200102610
姓名：管嘉瑞 学号：3200102557

本系统的分工如下：

管嘉瑞：实现 record 的序列化、堆表的维护和优化。buffer pool 中 lru_replacer 的初步实现。实现 executor 部分，进行对各个模块的接口对接。实现 VecHeap 和 ListHeap。进行测试。

张峻瑜：实现 DiskManager,BufferPoolManager,CatalogManager。实现 Clock Replacer，lur_replacer 的优化，IndexManager 模块的重构与去模板化，内存管理的优化，ManagedHeap 的实现。