MiniSQL设计说明书

浙江大学**2021~2022**学年春夏学期《数据库系统》夏学期大程报告

一、总体概述

1.1 实验目的

1. 设计并实现一个精简型单用户SQL引擎MiniSQL，允许用户通过字符界面输入SQL语句实现基本的增删改查操作，并能够通过索引来优化性能。
2. 通过对MiniSQL的设计与实现，提高系统编程能力，加深对数据库管理系统底层设计的理解。

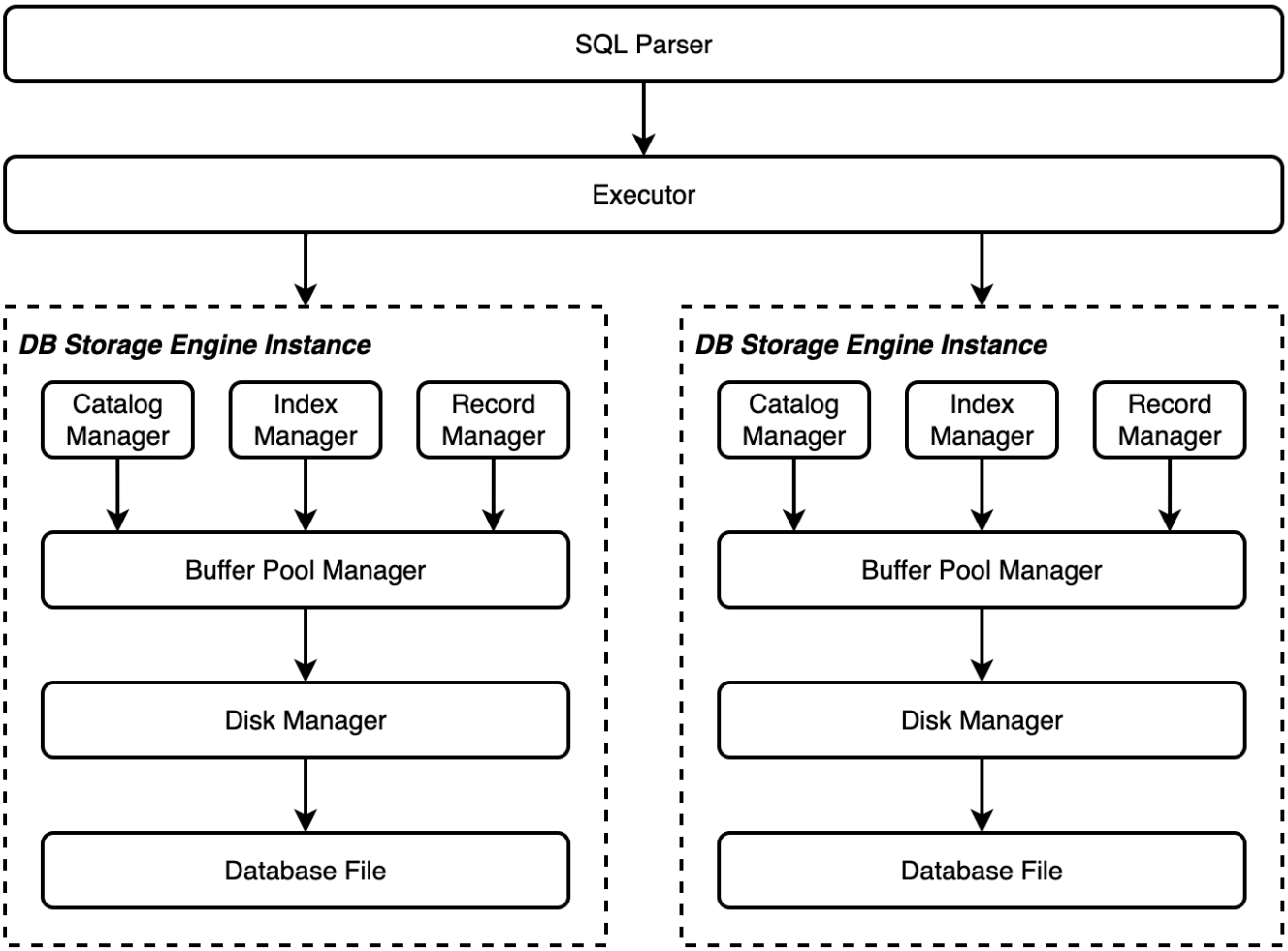
1.2 实验需求

1. 数据类型：要求支持三种基本数据类型：`integer`，`char(n)`，`float`。
2. 表定义：一个表可以定义多达32个属性，各属性可以指定是否为`unique`，支持单属性的主键定义。
3. 索引定义：对于表的主属性自动建立B+树索引，对于声明为`unique`的属性也需要建立B+树索引。
4. 数据操作: 可以通过`and`或`or`连接的多个条件进行查询，支持等值查询和区间查询。支持每次一条记录的插入操作；支持每次一条或多条记录的删除操作。
5. 在工程实现上，使用源代码管理工具（如Git）进行代码管理，代码提交历史和每次提交的信息清晰明确；同时编写的代码应符合代码规范，具有良好的代码风格。

二、系统架构与模块概述

2.1 系统架构示意图

- 在系统架构中，解释器`SQL Parser`在解析SQL语句后将生成的语法树交由执行器`Executor`处理。执行器则根据语法树的内容对相应的数据库实例（`DB Storage Engine Instance`）进行操作。
- 每个`DB Storage Engine Instance`对应了一个数据库实例（即通过`CREATE DATABSAE`创建的数据库）。在每个数据库实例中，用户可以定义若干表和索引，表和索引的信息通过`Catalog Manager`、`Index Manager`和`Record Manager`进行维护。目前系统架构中已经支持使用多个数据库实例，不同的数据库实例可以通过`USE`语句切换（即类似于MySQL的切换数据库），在初步实现时，可以先考虑单个数据库实例的场景，在单个实例跑通后再支持多个实例。



2.2 系统模块概述

2.2.1 Disk Manager

- Database File（DB File）是存储数据库中所有数据的文件，其主要由记录（Record）数据、索引（Index）数据和目录（Catalog）数据组成（即共享表空间的设计方式）。与书上提供的设计（每张表通过一个文件维护，每个索引也通过一个文件维护，即独占表空间的设计方式）有所不同。共享表空间的优势在于所有的数据在同一个文件中，方便管理，但其同样存在着缺点，所有的数据和索引存放到一个文件中将会导致产生一个非常大的文件，同时多个表及索引在表空间中混合存储会导致做了大量删除操作后可能会留有有大量的空隙。在本实验中，为了方便同学们实现，我们采取共享表空间的设计方式，即将所有的数据和索引放在同一个文件中。学有余力的同学可以额外尝试使用独占表空间的设计方式进行设计。
- Disk Manager负责DB File中数据页的分配和回收，以及数据页中数据的读取和写入。

2.2.2 Buffer Pool Manager

- Buffer Manager 负责缓冲区的管理，主要功能包括：

1. 1. 根据需要，从磁盘中读取指定的数据页到缓冲区中或将缓冲区中的数据页转储（Flush）到磁盘；
   2. 实现缓冲区的替换算法，当缓冲区满时选择合适的数据页进行替换；
   3. 记录缓冲区中各页的状态，如是否是脏页（Dirty Page）、是否被锁定（Pin）等；
   4. 提供缓冲区页的锁定功能，被锁定的页将不允许替换。

- 为提高磁盘 I/O 操作的效率，缓冲区与文件系统交互的单位是数据页（Page），数据页的大小应为文件系统与磁盘交互单位的整数倍。在本实验中，数据页的大小默认为 4KB。

2.2.3 Record Manager

- Record Manager 负责管理数据表中记录。所有的记录以堆表（Table Heap）的形式进行组织。Record Manager 的主要功能包括：记录的插入、删除与查找操作，并对外提供相应的接口。其中查找操作返回的是符合条件记录的起始迭代器，对迭代器的迭代访问操作由执行器（Executor）进行。
- 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。不要求支持单条记录的跨页存储（即保证所有插入的记录都小于数据页的大小）。堆表中所有的记录都是无序存储的。
- 需要额外说明的是，堆表只是记录组织的其中一种方式，除此之外，记录还可以通过顺序文件（按照主键大小顺序存储所有的记录）、B+树文件（所有的记录都存储在B+树的叶结点中，MySQL中InnoDB存储引擎存储记录的方式）等形式进行组织。学有余力的同学可以尝试着使用除堆表以外的形式来组织数据。

2.2.4 Index Manager

- Index Manager 负责数据表索引的实现和管理，包括：索引（B+树等形式）的创建和删除，索引键的等值查找，索引键的范围查找（返回对应的迭代器），以及插入和删除键值等操作，并对外提供相应的接口。
- B+树索引中的节点大小应与缓冲区的数据页大小相同，B+树的叉数由节点大小与索引键大小计算得到。

2.2.5 Catalog Manager

- Catalog Manager 负责管理数据库的所有模式信息，包括：

1. 1. 数据库中所有表的定义信息，包括表的名称、表中字段（列）数、主键、定义在该表上的索引。
   2. 表中每个字段的定义信息，包括字段类型、是否唯一等。
   3. 数据库中所有索引的定义，包括所属表、索引建立在那个字段上等。

- Catalog Manager 还必需提供访问及操作上述信息的接口，供执行器使用。

2.2.6 Executor

- Executor（执行器）的主要功能是根据解释器（Parser）生成的语法树，通过Catalog Manager 提供的信息生成执行计划，并调用 Record Manager、Index Manager 和 Catalog Manager 提供的相应接口进行执行，最后通过执行上下文将执行结果返回给上层模块。

2.2.7 SQL Parser

- 程序流程控制，即"启动并初始化 → '接收命令、处理命令、显示命令结果'循环 → 退出"流程。
- 接收并解释用户输入的命令，生成命令的内部数据结构表示，同时检查命令的语法正确性和部分语义正确性，对正确的命令生成语法树，然后调用执行器层提供的函数执行并显示执行结果，对不正确的命令显示错误信息。

三、内部数据形式及各模块提供的接口

3.1 DISK AND BUFFER POOL MANAGER

3.1.1 buffer pool manager

3.1.1.1 设计思想

buffer pool manager 负责在内存中管理从磁盘中创建和读取的磁盘页，通过对其进行高效的缓冲区管理以实现高效的数据读写。其使用高效的替换策略：当内存空间不足的时候，bpm首先从其维护的一个自由空间数据中查询是否有空闲的空间(free_list)，如果其有空闲的空间，那么直接返回一个空闲的空间，如果其没有空闲的空间，那么使用lru替换策略，从已有的内存页面中选择出一个最迟更新的内存页面，检查其状态（如果为脏页，将其写回磁盘），并将新的自由空间返回给bpm使用。当内存空间被其他模块释放的时候，并不是直接将其delete，而是将其存入replacer或者 free_list 中，以减少数据库运行过程中运行的delete和new指令的数量，加快数据库运行速度。

3.1.1.2 主要函数实现

3.1.1.2.1 获取磁盘页

在获取磁盘页的时候，首先查询在内存中是否已经有该页面的缓冲内容，如果有，那么将其 pin_count加一后返回，如果没有，那么使用上述替换策略获得一个空闲的内存页，使用disk_manager读取对应的磁盘页面到该内存页中，将pin_count置位后返回结果。

3.1.1.2.2 释放内存页（但是没有delete）

在其他模块使用内存页完毕之后，将对应的pin_count减一，并检查其pin_count是否为0，如果是，那么将其添加到bpm的replacer中，表示页面已经使用完毕，能够被替换，同时视 is_dirty的状态决定是否将页面标记为脏页

3.1.1.2.3 刷写磁盘页面

调用该函数之后，不论内存页脏页与否，都将其刷写回磁盘中。

3.1.1.2.4 创建新的内存页面

本函数主要供bpm内部调用，首先在free_list中查询是否有空闲的页面，如果没有的话再去replacer中获取一个空闲的页面，并在空闲页中写入足够的元信息之后使用返回新的内存页。

3.1.1.2.5 删除已经分配的内存页

如果对应的磁盘页面是脏页，那么将其刷写回磁盘，将该页面添加到free_list中

3.1.2 lru replacer

3.1.2.1 设计思想

设计思想：为了实现对于内存页的高效管理，本模块使用lru替换策略进行新空闲页面的存取。总体设计思想如下

1. 使用list这一能够在双端进行快速添加和删除的数据结构进行空闲磁盘页的存取
2. 使用一个vector存储各个内存页面的状态以及其在空闲页数组中对应的位置（实际上存储的是一个迭代器，以避免访问list造成的O(n)时间复杂度。

3.1.2.2 主要函数实现

3.1.2.2.1 获取空闲页

根据lru替换策略，直接从维护的空闲内存页中的最后一个记录返回，并将其在对应的数组和维护迭代器的数组中删除。

3.1.2.2.2 删除空闲页（Pin函数）

从维护迭代器的数组中获取对应的迭代器，并使用该迭代器删除对应的 list 中的数据。

3.1.2.2.3 添加空闲页 （Unpin函数）

根据lru替换策略，向维护内存空闲页的list头添加数据，并将新的迭代器添加到维护迭代器的数组中。

3.1.3 disk manager

3.1.3.1 设计思想

该模块主要实现了对于数据库文件的管理，包括磁盘页面读取，写入，逻辑页和物理页面的映射等等，并使用bitmap位图进行空闲页面的管理。值得注意的是，本程序实现中使用了一个缓存数组保存各个位图页的数据，避免重复读取磁盘增加的性能开销。

3.1.3.2 主要函数

3.1.3.2.1 读取磁盘页

将输入的逻辑页id映射为物理页id，并调用已经实现的读取函数读取数据。

3.1.3.2.2 写入磁盘页

将内存页写入逻辑页id对应的物理页中。

3.2 RECORD MANAGER

3.2.1 Row&Column&Schema

3.2.1.1 设计思想

这三个模块主要实现的功能类似，均为自身维护的数据的序列化和反序列化，便于写入磁盘和从磁盘中加载原有的信息，不同的是，Row维护自身的各个列的元信息，Column维护Schema中各个列的元信息。
值得注意的是，我们再实现中向schema添加了两个数据存储表的主键和unique的值，便于在插入的过程中辨别插入的值是否违反主键约束和唯一性约束。

3.2.1.2 主要函数

3.2.1.2.1 序列化

将各自维护的数据通过memcpy写入缓冲区（buf）中，如果需要序列化的数据是一个字符串，那么首先写入其长度，之后再将对应的字符串内容写入缓冲区中。

3.1.1.2.2 反序列化

这里借鉴raii的思想，按照和序列化相反的顺序进行反序列化。如果反序列化的数据是一个非字符串数据，那么直接按照其大小进行反序列化；否则首先读取其大小，再读取对应的字符串。这里也使用一个`magic_number`进行反序列化正确与否的判断。

3.2.2 Table_Heap

3.2.2.1 设计思想

Table_heap主要负责表中元组的相关操作，包括插入，删除，更新，获取等，实现对于元组的高效管理。

3.2.2.2 主要函数

3.2.2.2.1 插入元组

在分配的磁盘页较少的时候，使用first fit策略，在已有的所有磁盘页中查询能够容纳对应元组大小的第一个页，将该元组插入该页面的空闲位置中；在磁盘页较多的时候，使用next-fit策略，查询最后一个磁盘页能否容纳对应元组，如果可以，那么直接插入，否则直接创建新的页，将对应的记录插入。

3.2.2.2.2 删除元组

按照元组中存储的元信息，将处于指定磁盘页和制定slot位置的元组删除。

3.2.2.2.3 更新元组

按照元组中存储的元信息，将处于指定磁盘页和制定slot位置的元组更新。

3.2.2.2.3 获取元组

按照元组中存储的元信息，将处于指定磁盘页和制定slot位置的元组读取并返回。

3.2.3 table_iterator

3.2.3.1 设计思想

Table_iterator是table_heap的一个迭代器，能够从表的第一个记录线性移动到最后一个记录，为catalog, execute Engine等上层模块提供遍历表的接口

3.2.3.2 主要函数

3.2.2.2.1 Begin

获取对应于表的第一个元组的迭代器。

3.2.2.2.2 End

返回一个存储无效信息的迭代器，以判断遍历是否结束。

3.3 b+树相关

3.3.1 b+树介绍

B+树是B-树的变体，也是一颗多路搜索树。一棵m阶的B+树主要有这些特点：

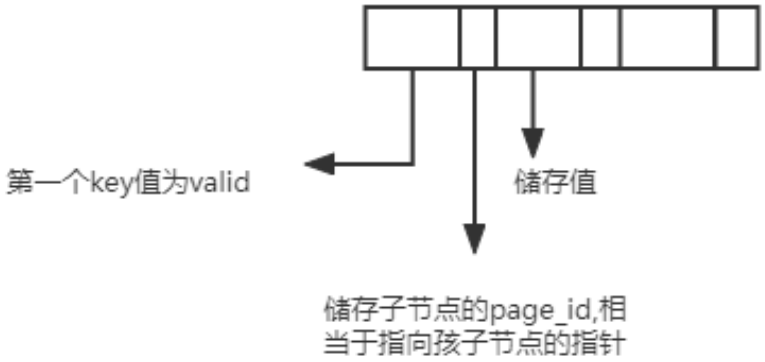
- 每个结点至多有m个子女;
- 非根节点关键值个数范围：⌈m/2⌉ - 1 <= k <= m-1
- 相邻叶子节点是通过指针连起来的，并且是关键字大小排序的。

3.3.2 数据结构介绍

而在本程序中，我们定义了B+树节点的类，并且分别定义了中间节点与叶节点两个子类作区分。

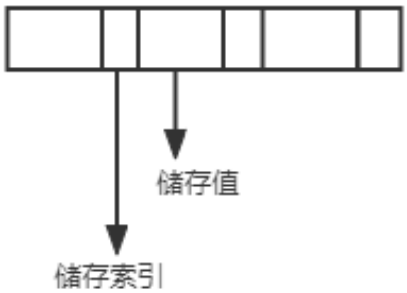
1. 中间节点：

   中间结点`BPlusTreeInternalPage`不存储实际的数据，它只按照顺序存储m个键和$m+1$个指针（这些指针记录的是子结点的`page_id`）。由于键和指针的数量不相等，因此我们需要将第一个键设置为INVALID，也就是说，顺序查找时需要从第二个键开始查找。在任何时候，每个中间结点至少是半满的（Half Full）。当删除操作导致某个结点不满足半满的条件，需要通过合并（Merge）相邻两个结点或是从另一个结点中借用（移动）一个元素到该结点中（Redistribute)来使该结点满足半满的条件。当插入操作导致某个结点溢出时，需要将这个结点分裂成为两个结点。

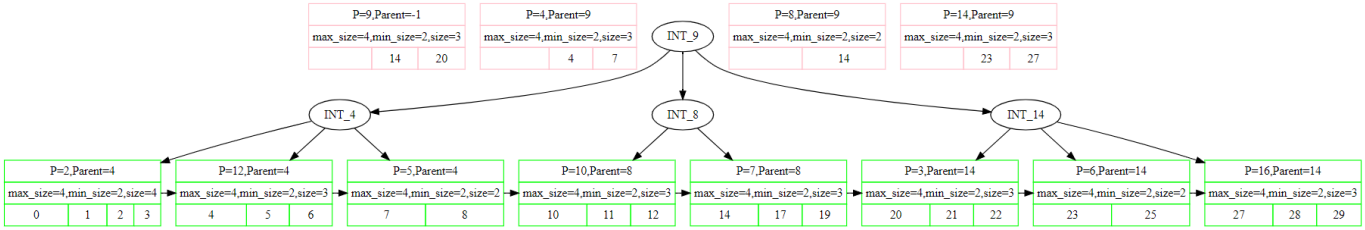


2. 叶子结点

叶结点`BPlusTreeLeafPage`存储实际的数据，它按照顺序存储m个键和m个值，其中键由一个或多个`Field`序列化得到，在`BPlusTreeLeafPage`类中用模板参数`KeyType`表示；值实际上存储的是`RowId`的值，它在`BPlusTreeLeafPage`类中用模板参数`ValueType`表示。叶结点和中间结点一样遵循着键值对数量的约束，同样也需要完成对应的合并、借用和分裂操作。



合法b+树



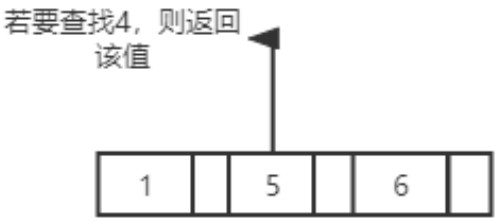
3.3.3具体操作

3.3.3.1查找操作

1. 对于每个节点的查找，我们采用了顺序查找

```
 int size = GetSize();
  ValueType result;
  int i;
  for (i = 1; i < size; i++) {
    if (comparator(key, array_[i].first) < 0) {
      result = array_[i - 1].second;
      break;
    }
  }
  if (i == size) result = array_[i - 1].second;
  return result;
```

找到第一个大于该节点的位置



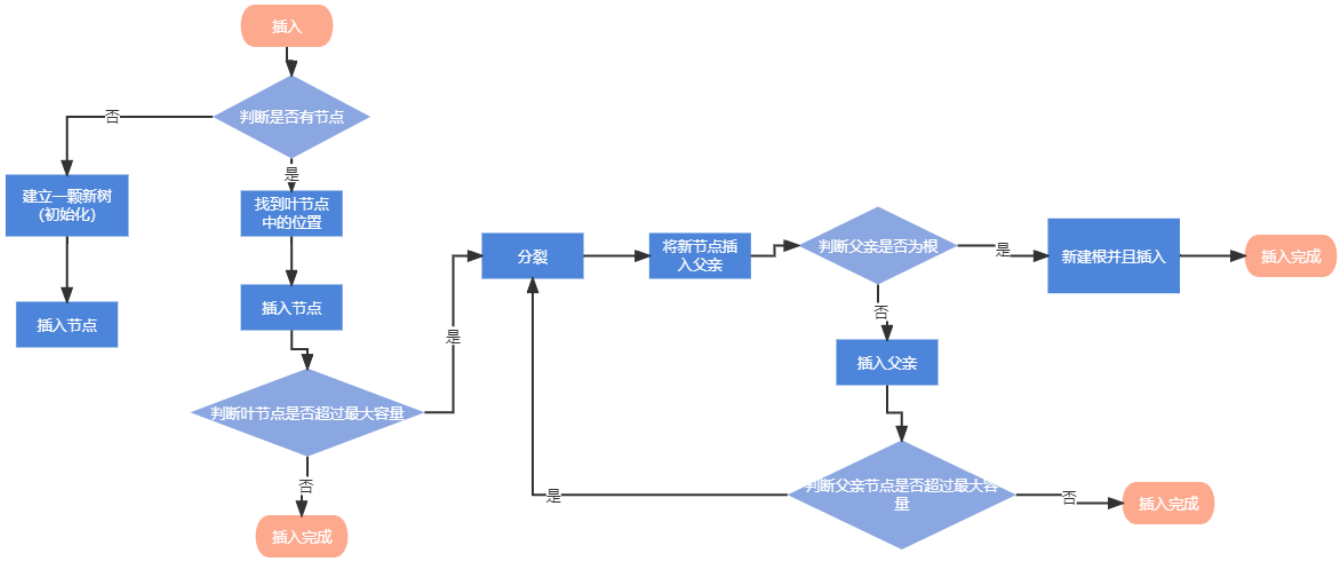
2. 对于整棵树的查找，我们从根节点开始递归，一直查找到叶子节点为止。

```
INDEX_TEMPLATE_ARGUMENTS
Page *BPLUSTREE_TYPE::FindLeafPage(const KeyType &key, bool leftMost) {
  Page *page = buffer_pool_manager_->FetchPage(root_page_id_);  // now root page is pin
  BPlusTreePage *node = reinterpret_cast<BPlusTreePage *>(page);
  while (!node->IsLeafPage()) {
    InternalPage *internal_node = reinterpret_cast<InternalPage *>(node);
    page_id_t next_page_id = leftMost ? internal_node->ValueAt(0) : internal_node-
>Lookup(key, comparator_);
    Page *next_page = buffer_pool_manager_->FetchPage(next_page_id);  //
next_level_page pinned
    BPlusTreePage *next_node = reinterpret_cast<BPlusTreePage *>(next_page);
    buffer_pool_manager_->UnpinPage(node->GetPageId(), false);  // curr_node unpinned
```

```
    page = next_page;
    node = next_node;
  }
  return page;
}
```

当该节点不是叶子结点时，则不断调用查找函数，返回第一个比查找值大的数，之后调用其对应的page_id，进入其孩子节点。最终可以找到值对应的索引。

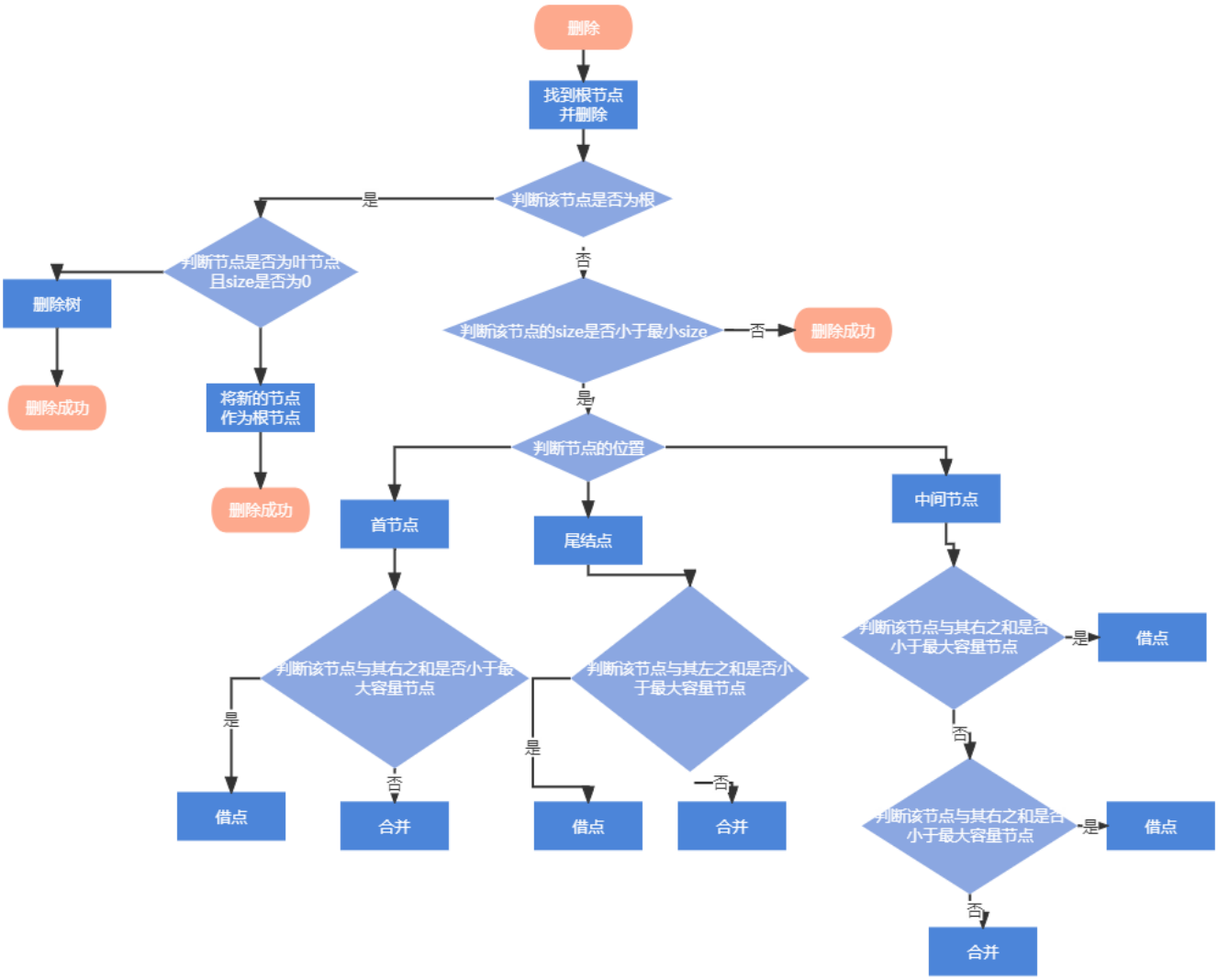
3.3.3.2插入操作



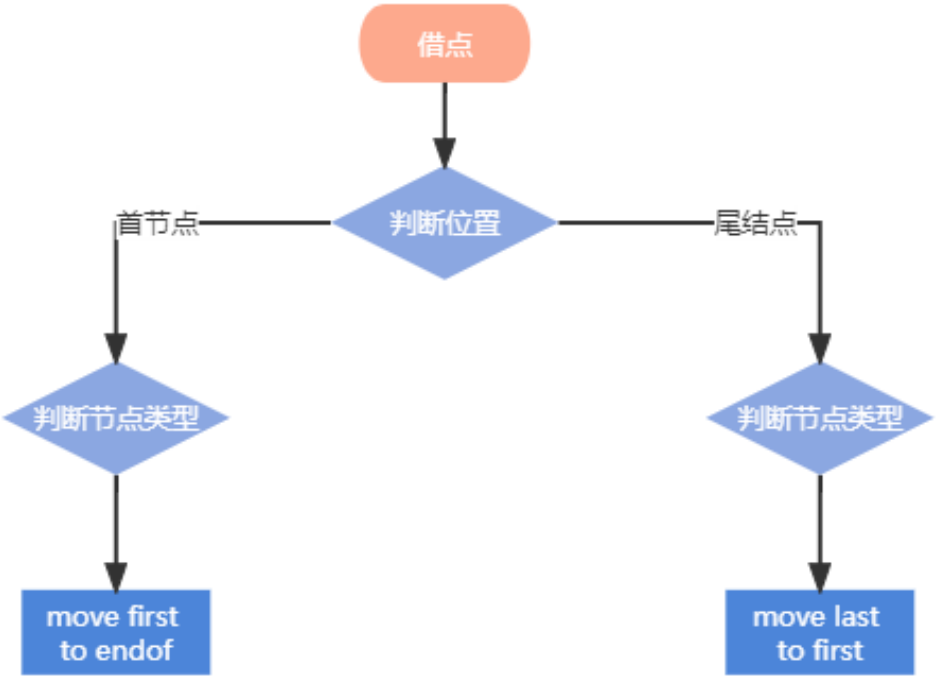
插入操作接口

```
  // Insert a key-value pair into this B+ tree.
  bool Insert(const KeyType &key, const ValueType &value, Transaction *transaction =
nullptr);

  void StartNewTree(const KeyType &key, const ValueType &value);

  bool InsertIntoLeaf(const KeyType &key, const ValueType &value, Transaction
*transaction = nullptr);

  void InsertIntoParent(BPlusTreePage *old_node, const KeyType &key, BPlusTreePage
*new_node,
                        Transaction *transaction = nullptr);
```

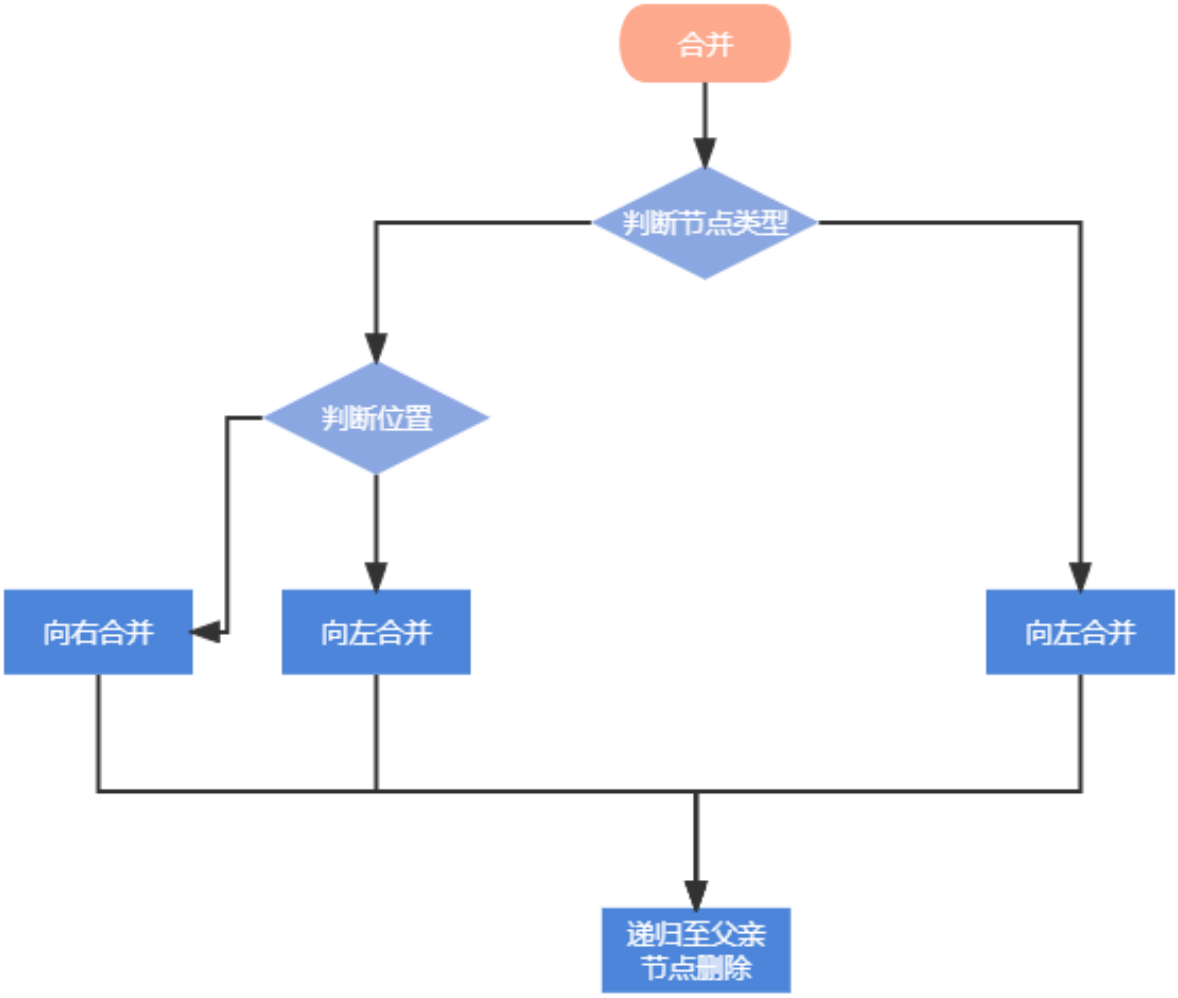
3.3.3.3删除操作


删除
找到根节点
并删除
判断该节点是否为根
是
否
判断节点是否为叶节点
且size是否为0
删除树
删除成功
将新的节点
作为根节点
删除成功
判断该节点的size是否小于最小size
否
删除成功
是
判断节点的位置
首节点
尾结点
中间节点
判断该节点与其右之和是否小于最大容量节点
是
否
借点
合并
判断该节点与其左之和是否小于最大容量节点
是
否
借点
合并
判断该节点与其右之和是否小于最大容量节点
是
借点
否
判断该节点与其右之和是否小于最大容量节点
是
借点
否
合并

1. 借点操作


借点
判断位置
首节点
尾结点
判断节点类型
判断节点类型
move first
to endof
move last
to first

2. 合并操作



NOTE:

1. 删除操作中只有合并会想根节点递归，但并不影响查询，这是因为查找时查找的是第一个比该值大的点。
2. 在合并操作中，由于函数**MOVEALLTO**是由节点合并至另一节点尾部，所以向左合并时目标节点为本节点，而向右合并则为邻居结点

函数接口

```
  void Remove(const KeyType &key, Transaction *transaction = nullptr);
template <typename N>
  N *Split(N *node);

  template <typename N>
  bool CoalesceOrRedistribute(N *node, Transaction *transaction = nullptr);

  template <typename N>
  bool Coalesce(N **neighbor_node, N **node, BPlusTreeInternalPage<KeyType, page_id_t,
KeyComparator> **parent,
                int index, Transaction *transaction = nullptr);
```

```
  template <typename N>
  void Redistribute(N *neighbor_node, N *node, int index);

  bool AdjustRoot(BPlusTreePage *node);

  void UpdateRootPageId(int insert_record = 0);
```

3.4 Catalog Manager

3.4.1 相关介绍

Catalog Manager 负责管理和维护数据库的所有模式信息，包括：

- 数据库中所有表的定义信息，包括表的名称、表中字段（列）数、主键、定义在该表上的索引。
- 表中每个字段的定义信息，包括字段类型、是否唯一等。
- 数据库中所有索引的定义，包括所属表、索引建立在那个字段上等。

3.4.2 设计思想

上述模式信息在被创建、修改和删除后可以被持久化到数据库文件中。此外，Catalog Manager还为上层的执行器Executor提供了公共接口以供执行器获取目录信息并生成执行计划。

3.4.3 函数接口与具体操作

实现与Catalog相关的元信息的序列化和反序列化操作。

与之前RecordManager中的序列化和反序列化类似，通过魔数MAGIC_NUM来确保序列化和反序列化的正确性。

在CatalogManager、TableInfo、IndexInfo中都通过各自的heap_维护和管理与自身相关的内存分配和回收。

在实现目录、表和索引元信息的持久化后，实现整个CatalogManager类后，CatalogManager类应具备维护和持久化数据库中所有表和索引的信息。

主要函数实现：

初始化：从文件中读取所有模式信息并插入到哈希表中。

存储：程序关闭时将哈希表中所有模式信息写回到文件中。

读取信息：大量的 get_xx（） 接口帮助其他模块在仅知道表名/索引名等有限的信息时查询到其他有关联的信息。

插入：插入表时直接插入到哈希表，插入索引时需要在哈希表和索引对应的那张表中写入索引信息。

删除：删除表时直接从哈希表中删除。删除索引时需要在哈希表和索引对应的那张表都删除索引信息。

```
void CatalogMeta::SerializeTo(char *buf)
  //  static constexpr uint32_t CATALOG_METADATA_MAGIC_NUM = 89849;
  //  std::map<table_id_t, page_id_t> table_meta_pages_;
  //  std::map<index_id_t, page_id_t> index_meta_pages_;
  // ASSERT(false, "Not Implemented yet");
  // 写入magic num
  CatalogMeta *CatalogMeta::DeserializeFrom(char *buf, MemHeap *heap)
  // ASSERT(false, "Not Implemented yet");
```

```
  // 检测magic num
    uint32_t CatalogMeta::GetSerializedSize()
CatalogMeta::CatalogMeta() {}

CatalogManager::CatalogManager(BufferPoolManager *buffer_pool_manager, LockManager
*lock_manager,
                               LogManager *log_manager, bool init)
  CatalogManager::~CatalogManager() {
  FlushCatalogMetaPage();
  delete heap_;
}
  dberr_t CatalogManager::CreateTable(const string &table_name, TableSchema *schema,
Transaction *txn,
                                    TableInfo *&table_info)
    dberr_t CatalogManager::GetTable(const string &table_name, TableInfo *&table_info)
    dberr_t CatalogManager::GetTables(vector<TableInfo *> &tables)
    dberr_t CatalogManager::CreateIndex(const std::string &table_name, const string
&index_name,
                                    const std::vector<std::string> &index_keys,
Transaction *txn,
                                    IndexInfo *&index_info)
    dberr_t CatalogManager::GetIndex(const std::string &table_name, const std::string
&index_name,
                                 IndexInfo *&index_info)
    dberr_t CatalogManager::GetTableIndexes(const std::string &table_name,
std::vector<IndexInfo *> &indexes)
    dberr_t CatalogManager::DropTable(const string &table_name)
    dberr_t CatalogManager::DropIndex(const string &table_name, const string
&index_name)
    dberr_t CatalogManager::FlushCatalogMetaPage()
    dberr_t CatalogManager::LoadTable
    dberr_t CatalogManager::LoadIndex(const index_id_t index_id, const page_id_t
page_id)
    dberr_t CatalogManager::GetTable
```

```
IndexMetadata *IndexMetadata::Create(const index_id_t index_id, const string
&index_name, const table_id_t table_id,
                                     const vector<uint32_t> &key_map, MemHeap *heap)
  uint32_t IndexMetadata::SerializeTo(char *buf)
  uint32_t IndexMetadata::GetSerializedSize()
  uint32_t IndexMetadata::DeserializeFrom(char *buf, IndexMetadata *&index_meta,
MemHeap *heap)
```

```
uint32_t TableMetadata::SerializeTo(char *buf)
  uint32_t TableMetadata::GetSerializedSize()
  uint32_t TableMetadata::DeserializeFrom(char *buf, TableMetadata *&table_meta,
MemHeap *heap)
```

3.5 SQL EXECUTOR

3.5.1 设计思想

为上层模块提供封装好的各种sql功能函数，方便执行sql语句，具体内容在主要函数的实现中。

3.5.2 主要函数

3.5.2.1 ExecuteCreateDatabase

这个函数主要执行数据库创建的工作。首先从抽象语法树中获取数据库的名称，判断无重复数据库名之后用数据名创建DBStorageEngine类的实例，并将数据库名写入存储所有数据库名称的文件 .minisql_meta中，以持久化存储和在退出minisql之后重新载入对应的数据库信息。

3.5.2.2 ExecuteShowDatabases

这个函数负责输出所有数据库的名称，由于在实现中我将所有数据库的名称存储在 .minisql_meta文件中，所以这里只需要遍历该文件并输出数据库名称即可。

3.5.2.3 ExecuteUseDatabase

这个函数的实现逻辑比较简单，首先从抽象语法树中获取数据库名，之后判断数据库名是否合法（即判断其是否存在该数据库，使用 sys 中的stat函数实现），之后将current_db_修改即可。

3.5.2.4 ExecuteShowTables

这个函数负责输出当前数据库所有的数据表名称。调用的时候首先检查current_db_是否被正确置位，如果没有则输出错误信息后返回；否则在对应的catalog_meta中获取所有数据表名并输出。

3.5.2.5 ExecuteCreateTable

这个函数负责在当前数据库中创建新的表。调用的时候首先检查current_db_是否被正确置位，如果没有则输出错误信息后返回；否则遍历抽象语法树，创建对应的schema（值得注意的是，由于在接下来的select函数中输入的抽象语法树中并没有对int类型的数据和float类型的数据做区分，而是统一使用kNodeNumber类型，所以这里直接将输入的数值类型统一用float进行存储），将主键和unique键的信息存入schema之后调用catalog_manager中的createTable函数进行表的创建，返回结果。

3.5.2.6 ExecuteDropTable

这个函数负责在当前数据库中删除表。调用的时候首先检查current_db_是否被正确置位，如果没有则输出错误信息后返回；否则从抽象语法树中获取要删除的表的信息，在调用catalog_manager中的dropindex函数将该表的所有索引删除之后调用catalog_manager中的droptable函数将该表本身删除。

3.5.2.7 ExecuteShowIndexes

这个函数负责在当前数据库中输出所有的索引。调用的时候首先检查current_db_是否被正确置位，如果没有则输出错误信息后返回；否则从catalog_manager中获取各个表对应的索引并输出其名称。

3.5.2.8 ExecuteCreateIndex

这个函数负责在当前数据库中创建索引。调用的时候首先检查current_db_是否被正确置位，如果没有则输出错误信息后返回；否则遍历抽象语法树，获得要创建的索引在表中对应的列的位置和名称，调用catalog_manager的createIndex创建对应的索引。

3.5.2.9 ExecuteDropIndex

这个函数负责在当前数据库中删除索引。调用的时候首先检查current_db_是否被正确置位，如果没有则输出错误信息后返回；否则遍历抽象语法树，获得要删除的index的名称，调用catalog_manager中的dropindex函数进行索引的删除。

3.5.2.10 get_columns_by_condition

这是我自己添加的功能函数，也是模块中的一个核心函数，供insert，update等多个模块调用。其功能为根据输入的抽象语法树中的条件查询节点在指定的数据表中进行查询，并返回所有符合条件的元组的rowid vector， 调用的时候其会首先检查对应的查询语句是否为空，如果是那么就使用table iterator遍历对应的数据表，将所有的rowid存储到结果中并返回；否则从抽象语法树条件查询的节点一路向下进行类似递归的查询，将各个比较语句和连接语句进行存储，在遍历结束之后查看当前查询查询是否能够使用索引，如果是那么调用index中的scankey函数进行查询，否则也是用迭代器对表的每一个元组进行检查，判断其是否满足对应的连接条件和查询条件。如果是，那么将其添加到返回的结果中，否则不添加，这样最后返回的就是存储着所有满足条件的rowid的vector。

3.5.2.11 ExecuteSelect

这个函数负责在当前数据库中进行查询。调用的时候首先检查current_db_是否被正确置位，如果没有则输出错误信息后返回；否则从抽象语法树中获得表名等信息之后调用get_columns_by_condition函数获取所有符合条件的rowid的vector，并输出对应结果。

3.5.2.12 ExecuteInsert

这个函数负责在当前数据库中进行插入。调用的时候首先检查current_db_是否被正确置位，如果没有则输出错误信息后返回；否则遍历抽象语法树先创建对应的row，然后使用建立在表上的主键索引和唯一键索引判断插入的数据是否违反了唯一性，如果不违反唯一性，那么将对应的记录插入堆表中，并更新索引信息后返回。

3.5.2.13 ExecuteDelete

这个函数负责在当前数据库中进行删除。调用的时候首先检查current_db_是否被正确置位，如果没有则输出错误信息后返回；否则从抽象语法树中获得表名等信息之后调用get_columns_by_condition函数获取所有符合条件的rowid的vector，并将其逐个从堆表和索引中删除后返回结果。

3.5.2.14 ExecuteUpdate

这个函数负责在当前数据库中进行更新。调用的时候首先检查current_db_是否被正确置位，如果没有则输出错误信息后返回；否则从抽象语法树中获得表名等信息之后调用get_columns_by_condition函数获取所有符合条件的rowid的vector，并将其逐个在堆表和索引中更新。

3.5.2.15 ExecuteExecfile

这个函数负责执行对应的sql语句。调用的时候就是逐个从文件中读入字符，直到读取到标志着结束的分号之后调用助教提供的解析函数进行语法树解析，然后调用execute函数进行执行。

四、系统测试

accounts.txt, accounts1.txt, accounts2.txt 各有10000个数据
basic.txt 包含建库的基本操作，可以直接execfile
cselect.txt 包含查询的基本测试，需要手动输入
dreopen.txt 用于测试数据持久性，需要手动输入

```
[       OK ] CatalogTest.CatalogTableTest (114 ms)
[ RUN      ] CatalogTest.CatalogIndexTest
[       OK ] CatalogTest.CatalogIndexTest (111 ms)
[----------] 3 tests from CatalogTest (226 ms total)

[----------] 4 tests from BPlusTreeTests
[ RUN      ] BPlusTreeTests.BPlusTreeIndexGenericKeyTest
[       OK ] BPlusTreeTests.BPlusTreeIndexGenericKeyTest (59 ms)
[ RUN      ] BPlusTreeTests.BPlusTreeIndexSimpleTest
[       OK ] BPlusTreeTests.BPlusTreeIndexSimpleTest (59 ms)
[ RUN      ] BPlusTreeTests.SampleTest
[       OK ] BPlusTreeTests.SampleTest (121 ms)
[ RUN      ] BPlusTreeTests.IndexIteratorTest
[       OK ] BPlusTreeTests.IndexIteratorTest (59 ms)
[----------] 4 tests from BPlusTreeTests (300 ms total)

[----------] 1 test from PageTests
[ RUN      ] PageTests.IndexRootsPageTest
[       OK ] PageTests.IndexRootsPageTest (0 ms)
[----------] 1 test from PageTests (0 ms total)

[----------] 4 tests from TupleTest
[ RUN      ] TupleTest.ColumnSerializeDeserializeTest
[       OK ] TupleTest.ColumnSerializeDeserializeTest (0 ms)
[ RUN      ] TupleTest.SchemaSerializeDeserializeTest
[       OK ] TupleTest.SchemaSerializeDeserializeTest (0 ms)
[ RUN      ] TupleTest.FieldSerializeDeserializeTest
[       OK ] TupleTest.FieldSerializeDeserializeTest (0 ms)
[ RUN      ] TupleTest.RowTest
[       OK ] TupleTest.RowTest (0 ms)
[----------] 4 tests from TupleTest (0 ms total)

[----------] 2 tests from DiskManagerTest
[ RUN      ] DiskManagerTest.BitMapPageTest
[       OK ] DiskManagerTest.BitMapPageTest (2 ms)
[ RUN      ] DiskManagerTest.FreePageAllocationTest
[       OK ] DiskManagerTest.FreePageAllocationTest (120 ms)
[----------] 2 tests from DiskManagerTest (122 ms total)

[----------] 1 test from TableHeapTest
[ RUN      ] TableHeapTest.TableHeapSampleTest
[       OK ] TableHeapTest.TableHeapSampleTest (570 ms)
[----------] 1 test from TableHeapTest (570 ms total)

[----------] Global test environment tear-down
[==========] 18 tests from 9 test suites ran. (1222 ms total)
[  PASSED  ] 18 tests.
```

1.execfile "basic.txt"; //此时建立五个数据库并在db0中建立account

```
create database db0;
create database db1;
create database db2;
create database db3;
create database db4;
show databases;
use db0;
create table account(
  id int,
  name char(16) unique,
  balance float,
  primary key(id)
);
```

```
minisql > execfile "basic.txt";
create database db0 success
create database db1 success
create database db2 success
create database db3 success
create database db4 success
db0
db1
db2
db3
db4
total 5 databases
switching to db: db0
operation ok
time cost: 0.264879 second
```

2.execfile "accounts.txt";
3.execfile "accounts1.txt";
4.execfile "accounts2.txt"; //分三次插入30000条数据

```
minisql > execfile "accounts.txt";
operation ok
time cost: 4.87645 second
minisql > execfile "accounts1.txt";
operation ok
time cost: 4.6683 second
minisql > execfile "accounts2.txt";
operation ok
time cost: 6.27083 second
```

5.手动输入cselect.txt中的测试

```
insert into account values(3300015, "name99999", 745.432); // 检验主键
insert into account values(9999999, "name20016", 60.985); // 检验unique

select * from account where id = 12500011;
select * from account where balance = 462.55;
select * from account where name = "name56789";  // 时间大
select * from account where id <> 12500013;
select * from account where balance <> 66.66;
select * from account where name <> "123";
select id, name from account where balance = 666.64 and id = 12500008;
select id, name from account where balance <= 200 and balance >= 100;

create index idx01 on account(name); // 建索引
```

```
insert into account values(32001, "name45678", 666.66);
select * from account where name = "name56789";  // 时间变小
select * from account where name = "name45678";  // 时间变小

drop index idx01;  // 删索引
select * from account where name = "name56789";  // 时间变大
select * from account where name = "name45678";    // 时间变大

insert into account values(32001, "name45678", 666.66);
update account set balance = 0 where name = "name45678";  // 更新
select * from account where name = "name45678";

delete from account where name = "name45678"; // 单个删除
delete from account where balance = 666.66;
select * from account where balance = 666.66;

delete from account where balance < 900 and balance > 0;
select * from account;

drop table account;
show tables;
show indexes;
execfile "accounts.txt"; // 重新插入，检验数据持久性
create index idx01 on account(name); //重建索引，检验数据持久性
quit;
```

```
minisql > insert into account values(3300015, "name99999", 745.432);
;
Minisql parse error at line 2, col 1, message: syntax error
syntax error
DB_FAILED
time cost: 0 second
```

```
minisql > insert into account values(9999999, "name20016", 60.985);
DB_UNIQUE_KEY_COLLISION ERROR
time cost: 0.000258 second
```

```
minisql > select * from account where id = 12500011;
12500011 name11 539.080017
total 1 records
operation ok
time cost: 0.000376 second
```

```
minisql > select * from account where balance = 462.55;
12500021 name21 462.549988
total 1 records
operation ok
time cost: 0.290398 second
```

```
minisql > select * from account where name = "name56789";
total 0 records
operation ok
time cost: 0.000184 second
```

```
3309982 name29982 827.958008
3309983 name29983 260.399994
3309984 name29984 961.346985
3309985 name29985 272.626007
3309986 name29986 752.466980
3309987 name29987 689.893982
3309988 name29988 599.422974
3309989 name29989 977.940979
3309990 name29990 656.434021
3309991 name29991 194.507996
3309992 name29992 369.778015
3309993 name29993 440.291992
3309994 name29994 794.166992
3309995 name29995 479.506012
3309996 name29996 849.770020
3309997 name29997 646.247009
3309998 name29998 858.072021
3309999 name29999 885.867004
total 29999 records
operation ok
time cost: 0.978009 second
```

```
3309982 name29982 827.958008
3309983 name29983 260.399994
3309984 name29984 961.346985
3309985 name29985 272.626007
3309986 name29986 752.466980
3309987 name29987 689.893982
3309988 name29988 599.422974
3309989 name29989 977.940979
3309990 name29990 656.434021
3309991 name29991 194.507996
3309992 name29992 369.778015
3309993 name29993 440.291992
3309994 name29994 794.166992
3309995 name29995 479.506012
3309996 name29996 849.770020
3309997 name29997 646.247009
3309998 name29998 858.072021
3309999 name29999 885.867004
total 30000 records
operation ok
time cost: 0.787623 second
```

```
3309983 name29983 260.399994
3309984 name29984 961.346985
3309985 name29985 272.626007
3309986 name29986 752.466980
3309987 name29987 689.893982
3309988 name29988 599.422974
3309989 name29989 977.940979
3309990 name29990 656.434021
3309991 name29991 194.507996
3309992 name29992 369.778015
3309993 name29993 440.291992
3309994 name29994 794.166992
3309995 name29995 479.506012
3309996 name29996 849.770020
3309997 name29997 646.247009
3309998 name29998 858.072021
3309999 name29999 885.867004
total 30000 records
operation ok
time cost: 0.922644 second
```

```
minisql > select id, name from account where balance = 666.64 and id = 12500008;
12500008 name8
total 1 records
operation ok
time cost: 0.316262 second
```

```
3309902 name29902
3309909 name29909
3309910 name29910
3309912 name29912
3309927 name29927
3309931 name29931
3309935 name29935
3309942 name29942
3309946 name29946
3309947 name29947
3309950 name29950
3309954 name29954
3309955 name29955
3309964 name29964
3309968 name29968
3309976 name29976
3309991 name29991
total 3022 records
operation ok
time cost: 0.386496 second
```

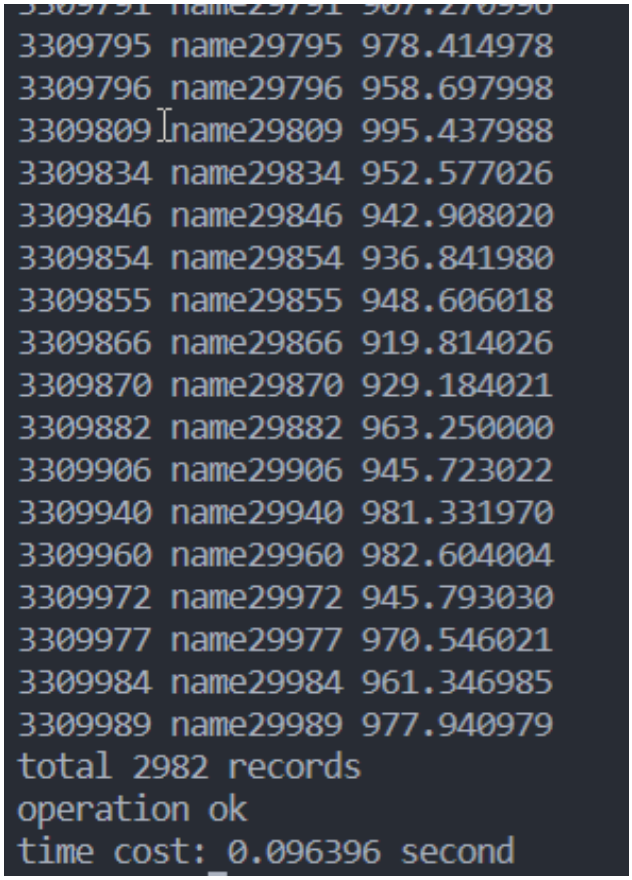
```
minisql > create index idx01 on account(name);
operation ok
time cost: 4.10783 second
```

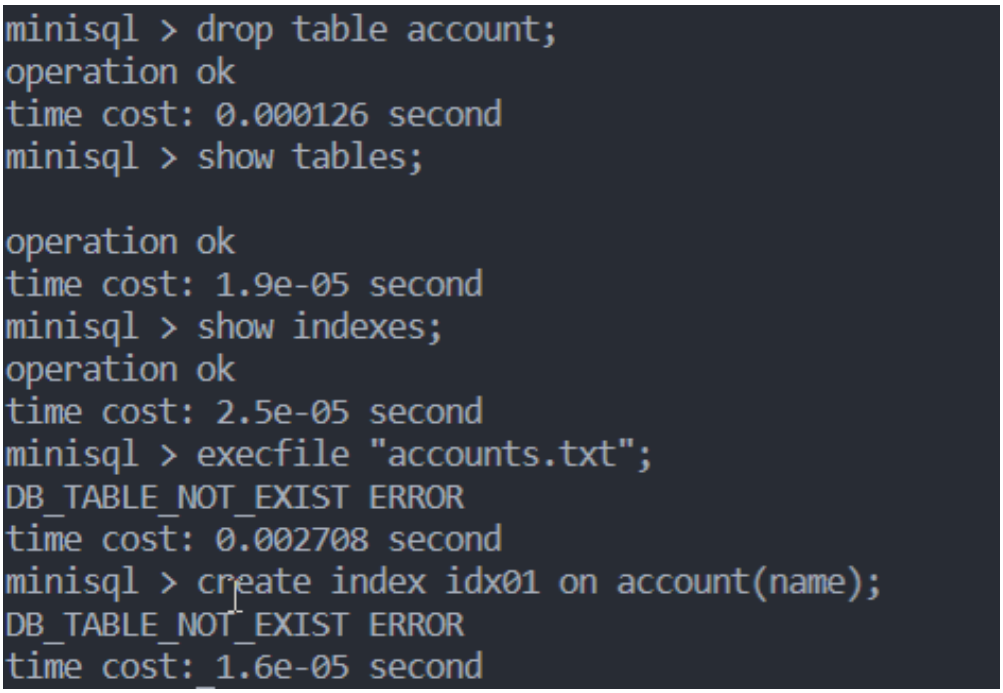
```
minisql > insert into account values(32001, "name45678", 666.66)
operation ok
time cost: 0.000654 second
minisql > select * from account where name = "name56789";
total 0 records
operation ok
time cost: 0.000213 second
minisql > select * from account where name = "name45678";
32001 name45678 666.659973
```

```
minisql > drop index idx01;
operation ok
time cost: 6.9e-05 second
minisql > select * from account where name = "name56789";
total 0 records
operation ok
time cost: 0.37037 second
minisql > select * from account where name = "name45678";
32001 name45678 666.659973
total 1 records
operation ok
time cost: 0.334581 second
```

```
minisql > insert into account values(32001, "name45678", 666.66);
DB_PRIMARY_KEY_COLLISION ERROR
time cost: 7.7e-05 second
minisql > update account set balance = 0 where name = "name45678";
1  rows affected
operation ok
time cost: 0.353097 second
minisql > select * from account where name = "name45678";
32001 name45678 0
total 1 records
operation ok
time cost: 0.333821 second
```

```
minisql > delete from account where name = "name45678";
1  rows affected
operation ok
time cost: 0.329903 second
minisql > delete from account where balance = 666.66;
0  rows affected
operation ok
time cost: 0.322326 second
minisql > select * from account where balance = 666.66;
total 0 records
operation ok
time cost: 0.326037 second
```





6.手动输入dreopen.txt中的测试检验数据持久性

```
show databases;
use db0;
select * from account; // 重开操作，检验持久性
show indexes;
```