MiniSQL数据库系统设计报告

开发时间：

2022.5.01——2022.6.29

开发人员：

- 姓名： 蒋思超 学号：3200104372 专业：计算机科学与技术
- 姓名： 王晨雨 学号：3200102324 专业：计算机科学与技术
- 姓名： 王婕　 学号：3200102315 专业：计算机科学与技术

指导老师：庄越挺

摘要

数据库系统在日常生活中应用广泛，了解其底层具体的设计细节对于数据库系统的开发和优化有至关重要的作用。在MiniSQL项目设计中，我们实现了SQL引擎的精简型单用户版本的基本实现，展现了数据库的底层设计的具体细节，涉及到了磁盘，缓冲区的管理，记录，索引，目录，执行器的设计等，成功实现了基本的查询索引等功能。

关键词：数据库；SQL；底层设计

目录

第1章 项目背景

1.1 项目的提出

数据库管理系统在日常应用中使用频率很高，如何深入理解数据库管理系统的底层设计，对于开发数据库相关软件有着重要意义。自主开发实现一个精简型单用户SQL引擎MiniSQL可以加深对数据库系统底层设计的了解程度，同时切实的研究开发，更助于理解该系统的运行架构，了解系统中仍旧存在的需要优化的方面，便于未来对数据库系统的进一步开发和优化。

1.2 项目任务

MiniSQL要求设计并实现一个精简型单用户SQL引擎MiniSQL。通过运行MiniSQL，用户可以通过字符界面输入SQL语句实现数据库的基本的增删改查操作。MiniSQL中通过索引来实现性能的优化。

具体需实现的要求如下：

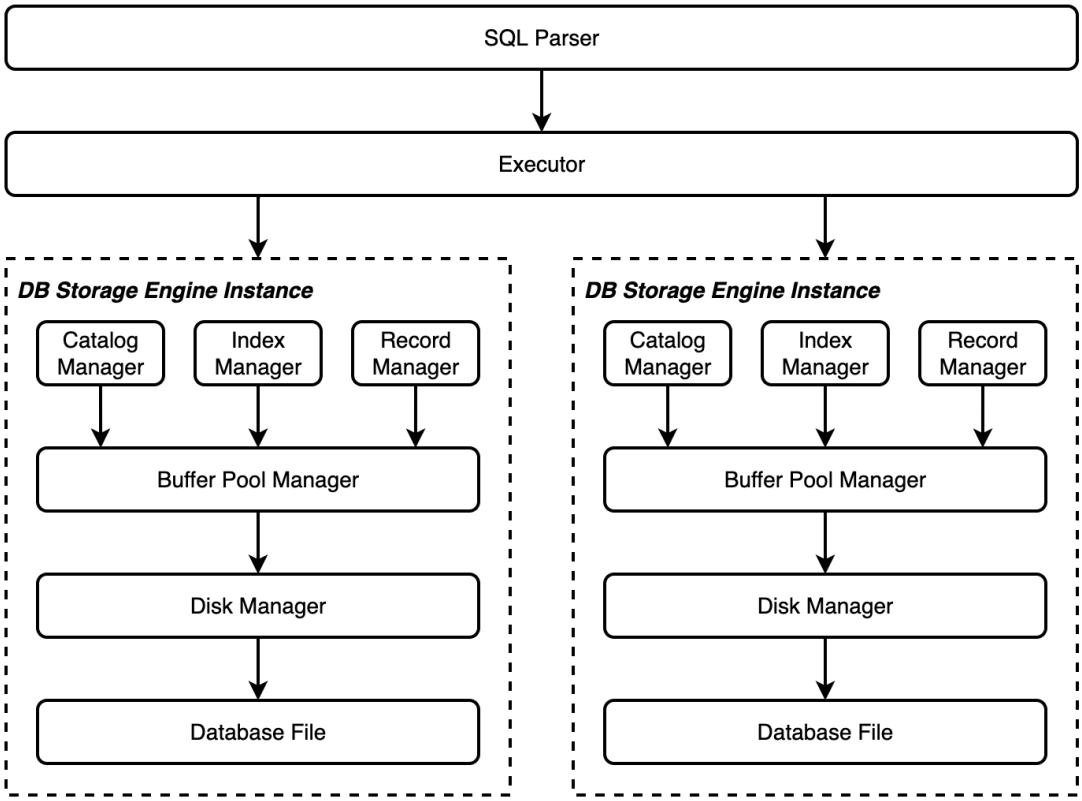
1. MiniSQL可以支持`integer，char(n)，float`三种基本数据类型的操作。
2. 一个表可以定义多达32个属性，各属性可以指定是否为`unique`，支持单属性的主键定义。
3. 索引定义：对于表的主属性自动建立B+树索引，对于声明为`unique`的属性也需要建立B+树索引。
4. 数据操作: 可以通过`and`或`or`连接的多个条件进行查询，支持等值查询和区间查询。支持每次一条记录的插入操作；支持每次一条或多条记录的删除操作。

在实现MiniSQL工程的过程，使用Git进行代码管理，同时编写的代码页需要符合代码规范。

第2章 产品架构概述

2.1 系统架构概述

- 在系统架构中，解释器`SQL Parser`在解析SQL语句后，将生成的需要完成的操作交给执行器，执行器对相应的数据库实例进行操作。每个实例的表的内容和索引在定义后由`Catalog Manager`、`Index Manager`和`Record Manager`进行维护，并进一步保存在缓冲区，之后写入磁盘的相应文件中。当任务传达后，通过`Catalog Manager`、`Index Manager`和`Record Manager`等找到需要操作（或创建）的数据库实例后进行操作，最后再记录在缓存中，后写入文件中。系统架构中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。



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）。我们采用共享表空间的设计方式。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。
- 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. 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。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。

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. 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。

第3章 具体模块的工作实现

3.1 DISK AND BUFFER POOL MANAGER

3.1.1 需求分析

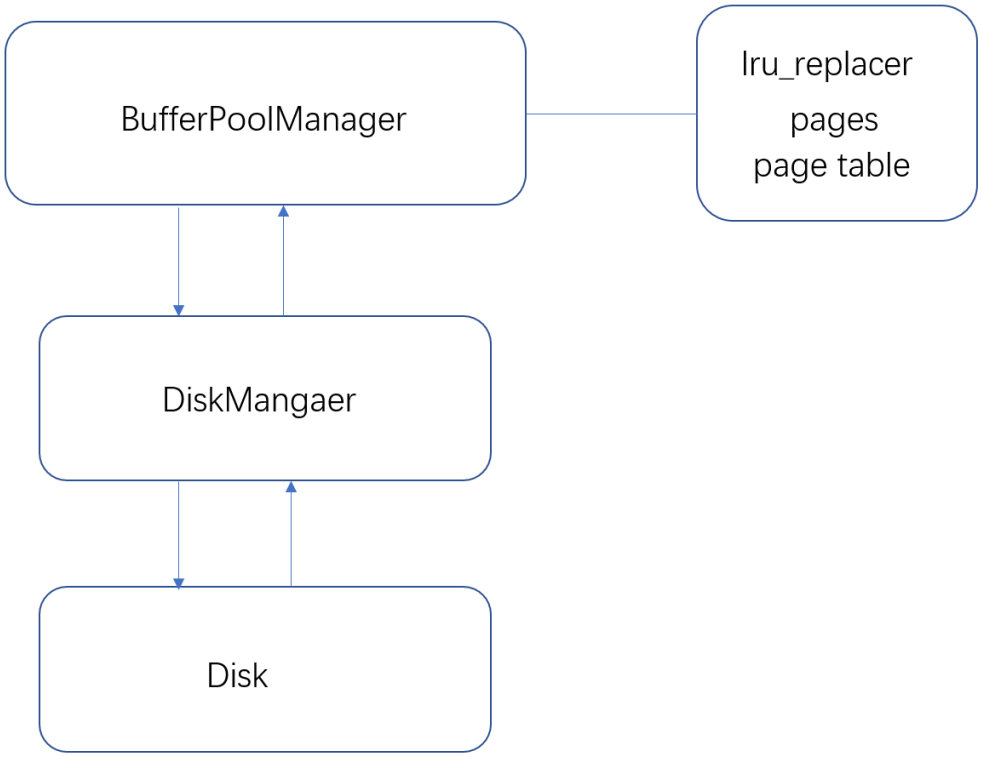
在MiniSQL的设计中，`Disk Manager`和`Buffer Pool Manager`模块位于架构的最底层。

`Disk Manager`主要负责数据库文件中数据页的分配和回收，以及数据页中数据的读取和写入。

`Buffer Pool Manager`中的操作对数据库系统中其他模块是透明的。系统的其它模块可以使用数据页唯一标识符`page_id`向`Buffer Pool Manager`请求对应的数据页。

`Buffer Pool Manager`维护一个数据页的内存池，通过逻辑页号与`Disk Manager`交互，读取或写出对应的数据页。

3.1.2 架构设计



`DiskManager`负责与硬盘进行文件交互，包括元数据信息和数据信息。`BufferPoolManager`和`DiskManager`使用逻辑页号进行交互，`DiskManager`隐藏了元数据信息。

3.1.3 实现细节

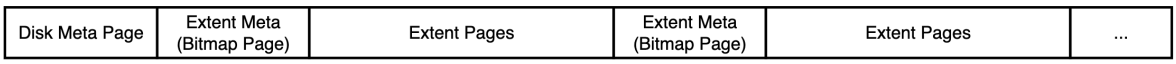
Page 类

```
class Page {
  // There is book-keeping information inside the page that should only be
relevant to the buffer pool manager.
  friend class BufferPoolManager;

public:
  DISALLOW_COPY(Page)

  /** Constructor. Zeros out the page data. */
  Page() { ResetMemory(); }

  /** Default destructor. */
  ~Page() = default;

  /** @return the actual data contained within this page */
  inline char *GetData() { return data_; }

  /** @return the page id of this page */
  inline page_id_t GetPageId() { return page_id_; }

  /** @return the pin count of this page */
  inline int GetPinCount() { return pin_count_; }
```

```
  /** @return true if the page in memory has been modified from the page on
disk, false otherwise */
  inline bool IsDirty() { return is_dirty_; }

  /** Acquire the page write latch. */
  inline void WLatch() { rwlatch_.WLock(); }

  /** Release the page write latch. */
  inline void WUnlatch() { rwlatch_.WUnlock(); }

  /** Acquire the page read latch. */
  inline void RLatch() { rwlatch_.RLock(); }

  /** Release the page read latch. */
  inline void RUnlatch() { rwlatch_.RUnlock(); }

  /** @return the page LSN. */
  inline lsn_t GetLSN() { return *reinterpret_cast<lsn_t *>(GetData() +
OFFSET_LSN); }

  /** Sets the page LSN. */
  inline void SetLSN(lsn_t lsn) { memcpy(GetData() + OFFSET_LSN, &lsn,
sizeof(lsn_t)); }

protected:
  static_assert(sizeof(page_id_t) == 4);
  static_assert(sizeof(lsn_t) == 4);

  static constexpr size_t SIZE_PAGE_HEADER = 8;
  static constexpr size_t OFFSET_PAGE_START = 0;
  static constexpr size_t OFFSET_LSN = 4;

private:
  /** Zeroes out the data that is held within the page. */
  inline void ResetMemory() { memset(data_, OFFSET_PAGE_START, PAGE_SIZE); }

  /** The actual data that is stored within a page. */
  char data_[PAGE_SIZE]{};
  /** The ID of this page. */
  page_id_t page_id_ = INVALID_PAGE_ID;
  /** The pin count of this page. */
  int pin_count_ = 0;
  /** True if the page is dirty, i.e. it is different from its corresponding
page on disk. */
  bool is_dirty_ = false;
  /** Page latch. */
  ReaderWriterLatch rwlatch_;
};
```

在MiniSQL中文件读写最小单位为页，在实现中为4KB。

disk file 组织形式



文件具体如上图所示，每一块`Extent Pages`由一个`bitmap`维护管理。`bitmap`记录了页的使用情况。所有的`bitmap`由`DiskMetaPage`维护，由此形成一个二级结构，使得一个数据库文件能够储存更多的数据。

disk manager 类

```
class DiskManager {
public:
  explicit DiskManager(const std::string &db_file);

  ~DiskManager() {
    if (!closed) {
      Close();
    }
  }

  /**
   * Read page from specific page_id
   * Note: page_id = 0 is reserved for disk meta page
   */
  void ReadPage(page_id_t logical_page_id, char *page_data);

  /**
   * Write data to specific page
   * Note: page_id = 0 is reserved for disk meta page
   */
  void WritePage(page_id_t logical_page_id, const char *page_data);

  /**
   * Get next free page from disk
   * @return logical page id of allocated page
   */
  page_id_t AllocatePage();

  /**
   * Free this page and reset bit map
   */
  void DeAllocatePage(page_id_t logical_page_id);

  /**
   * Return whether specific logical_page_id is free
   */
  bool IsPageFree(page_id_t logical_page_id);

  /**
   * Shut down the disk manager and close all the file resources.
   */
  void Close();

  /**
```

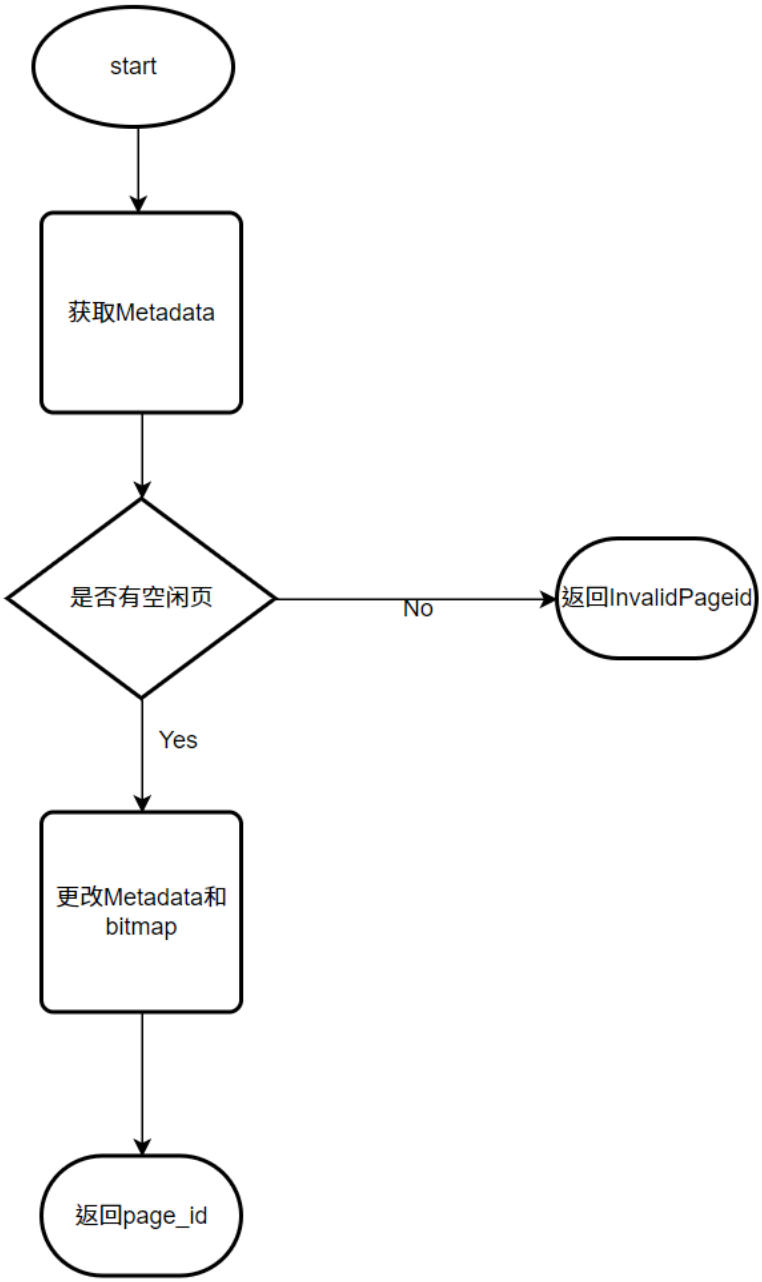
```
    * Get Meta Page
    * Note: Used only for debug
    */
   char *GetMetaData() {
     return meta_data_;
   }

   static constexpr size_t BITMAP_SIZE =
BitmapPage<PAGE_SIZE>::GetMaxSupportedSize();

private:
   /**
    * Helper function to get disk file size
    */
   int GetFileSize(const std::string &file_name);

   /**
    * Read physical page from disk
    */
   void ReadPhysicalPage(page_id_t physical_page_id, char *page_data);

   /**
    * Write data to physical page in disk
    */
   void WritePhysicalPage(page_id_t physical_page_id, const char *page_data);

   /**
    * Map logical page id to physical page id
    */
   page_id_t MapPageId(page_id_t logical_page_id);

private:
   // stream to write db file
   std::fstream db_io_;
   std::string file_name_;
   // with multiple buffer pool instances, need to protect file access
   std::recursive_mutex db_io_latch_;
   bool closed{false};
   char meta_data_[PAGE_SIZE];
   //a simple buffer pool, use a bitmap, if this bitmap is not we want, update
it.
   char cur_bitmap_[PAGE_SIZE];
   uint32_t extent_id_=0;

};
```

重要成员函数：

```
page_id_t AllocatePage();//新分配一个数据页，且返回其逻辑页号
```

该函数首先获得`Metadata`和查看是否存在空闲页，若有则修改对应的`Metadata`和`bitmap`,返回逻辑页号，否则分配失败，返回无效页号。

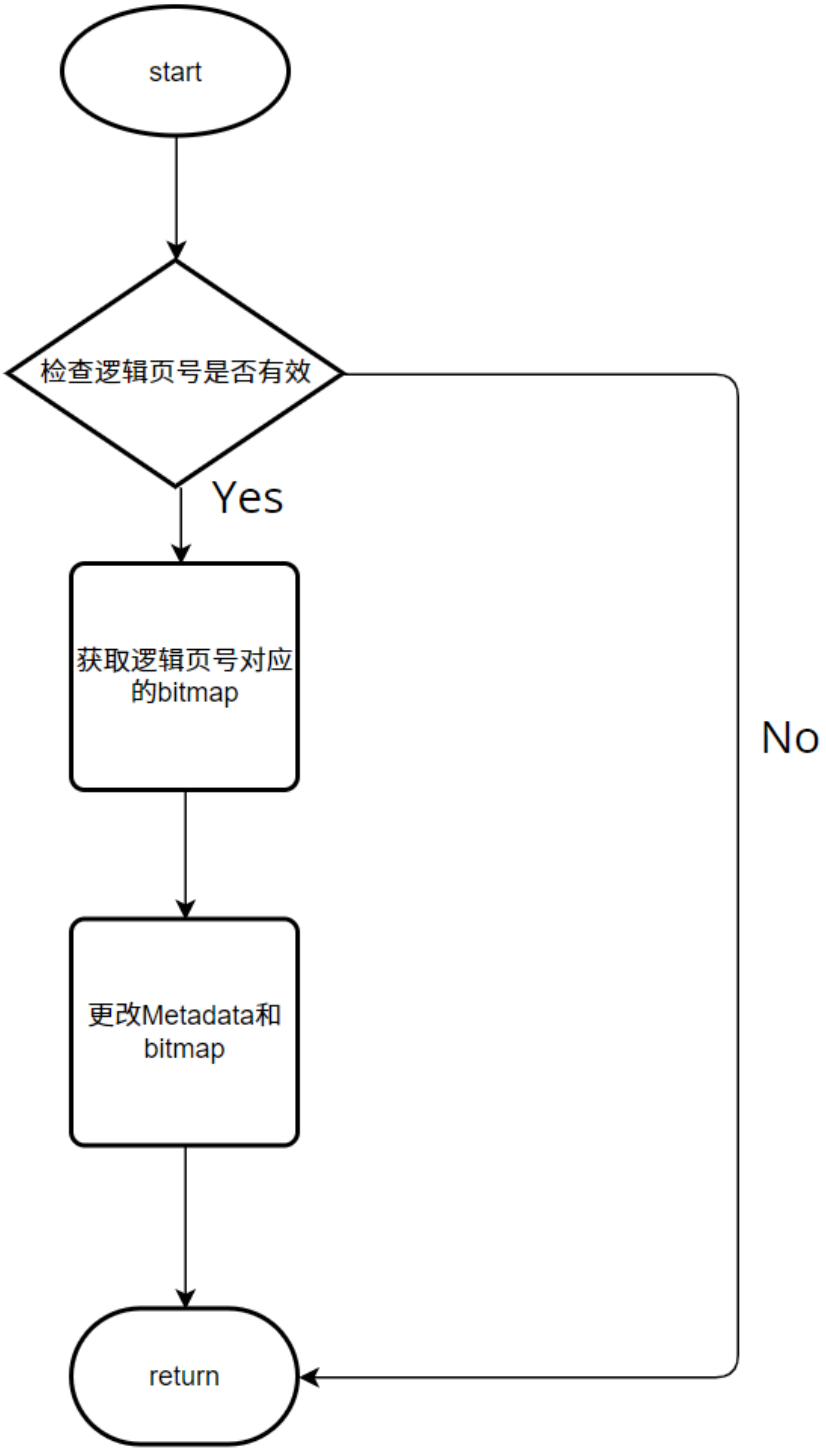
流程图如下：




```
void DeAllocatePage(page_id_t logical_page_id);//释放一个数据页，其位置由逻辑页号给出
```

该函数首先检查逻辑页号是否有效，若有效则修改对应的`Metadata`和`bitmap`，否则直接返回。

流程图如下：



LRU replacer 类

```
class LRUReplacer : public Replacer {
public:
  /**
   * Create a new LRUReplacer.
   * @param num_pages the maximum number of pages the LRUReplacer will be
required to store
   */
  explicit LRUReplacer(size_t num_pages);

  /**
   * Destroys the LRUReplacer.
   */
```

```
  ~LRUReplacer() override;

  bool Victim(frame_id_t *frame_id) override;

  void Pin(frame_id_t frame_id) override;

  void Unpin(frame_id_t frame_id) override;
  /*void Access(frame_id_t frame_id):
   if a frame_id is accessed, move it to the begin of the lru_list
  */

  size_t Size() override;

private:
  // add your own private member variables here
 /*lru_list, always replace the last element of the list*/
 list<frame_id_t> lru_list;
 size_t max_size;
 /*frame_id mapping iterator, to quickly find where the element is*/
 unordered_map<frame_id_t, list<frame_id_t>::iterator> id_map_it;
 void Access(frame_id_t frame_id);
};
```

`LRUreplacer` 用于实现 `buffer pool manager` 中的替换策略。

`lru_list` 维护了一系列可以被替换出内存池的页，每次需要换出时，总是选择链表末尾的页。一旦一个页被释放（Unpin）后，则将其置于链表的头。上层模块可以Pin某个页，使得此页从链表中移除，从而不会被替换。

BufferPoolManager 类

```
class BufferPoolManager {
public:
  explicit BufferPoolManager(size_t pool_size, DiskManager *disk_manager);

  ~BufferPoolManager();

  Page *FetchPage(page_id_t page_id);

  bool UnpinPage(page_id_t page_id, bool is_dirty);

  bool FlushPage(page_id_t page_id);

  Page *NewPage(page_id_t &page_id);

  bool DeletePage(page_id_t page_id);

  bool IsPageFree(page_id_t page_id);

  //bool FlushAllPages();

  bool CheckAllUnpinned();

private:
  /**
```

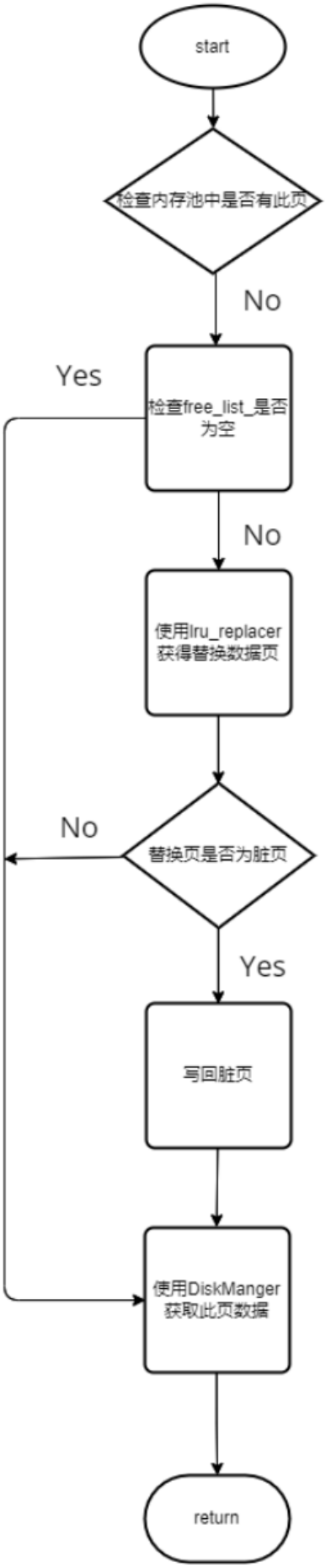
```
   * Allocate new page (operations like create index/table) For now just keep an
increasing counter
   */
  page_id_t AllocatePage();

  /**
   * Deallocate page (operations like drop index/table) Need bitmap in header
page for tracking pages
   */
  void DeallocatePage(page_id_t page_id);


private:
  /*frame_id is the index of pages, page_id is logical page id*/
 /*buffer_pool_manager is the friend class of Page*/
  size_t pool_size_;                                      // number of pages
in buffer pool
  Page *pages_;                                           // array of pages
  DiskManager *disk_manager_;                             // pointer to the
disk manager.
  std::unordered_map<page_id_t, frame_id_t> page_table_;    // to keep track of
pages
  Replacer *replacer_;                                    // to find an
unpinned page for replacement
  std::list<frame_id_t> free_list_;                       // to find a free
page for replacement
  recursive_mutex latch_;                                 // to protect shared
data structure
};
```

`BufferPoolManager` 维护个一个内存池 `pages_`，在此内存池中进行页的替换。`page_table_` 维护了页的逻辑页号和页在内存池中的位置。`free_list_` 维护了内存池中空余的页，当 `free_list_` 为空时，则需要进行替换。替换的页号由 `replacer` 给出。

重点成员函数：

```
Page *FetchPage(page_id_t page_id);//返回对应page_id逻辑页号的页的指针
```

流程图如下：




```
bool UnpinPage(page_id_t page_id, bool is_dirty);
```

释放某一页。当某一页的`pin_count`为0时，则意味这此页可被换出。

```
Page *NewPage(page_id_t &page_id);
```

申请一个新的数据页，调用`disk_manager`中的`AllocatePage()`函数实现。

```
bool DeletePage(page_id_t page_id);
```

删除某一页。若此页的`pin_count`为0，调用`disk_manager`中的`DeAllocatePage(page_id_t logical_page_id)`函数，否则删除失败。

3.2 RECORD MANAGER

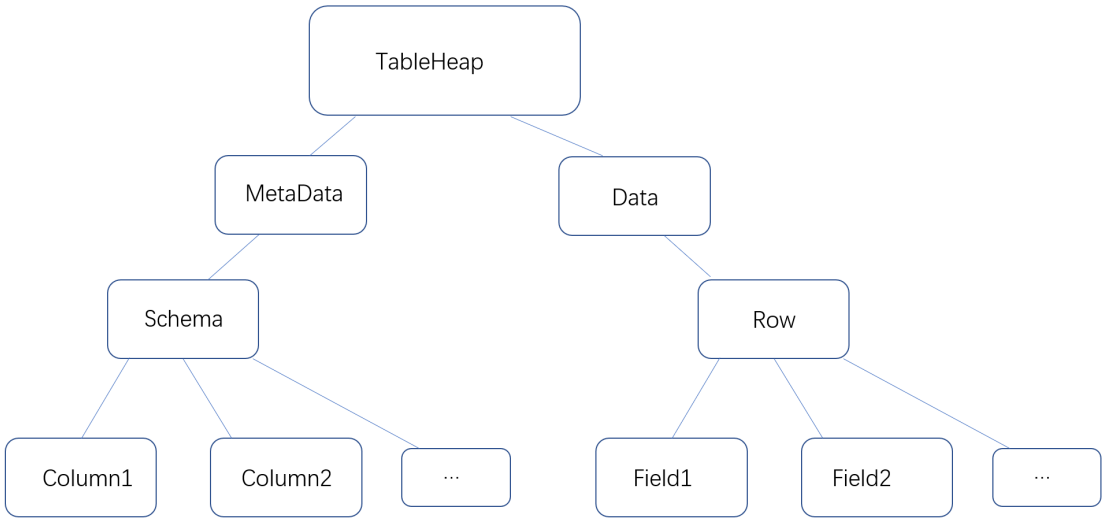
3.2.1 需求分析

在MiniSQL的设计中，`Record Manager`负责管理数据表中所有的记录，它能够支持记录的插入、删除与查找操作，并对外提供相应的接口。需求如下：

`Record Manager`需要将`Row`、`Field`、`Schema`和`Column`对象在硬盘上持久化，使得数据库数据能够长久保存。

`Record Manager`模块使用堆表管理记录，需要实现堆表的插入、删除、查询以及堆表记录迭代器的相关的功能。

3.2.2 架构设计



`Schema`记录了表的元信息，由多个`Column`组成。`Row`记录了数据信息，由多个`Field`组成。此外`TableHeap`提供迭代器和对外接口。

3.2.3 实现细节

Row 类

```
class Row {
  /*this friend class is added by me*/
  friend class TableIterator;

 public:
  /**
```

```
   * Row used for insert
   * Field integrity should check by upper level
   */
  explicit Row(std::vector<Field> &fields) : heap_(new SimpleMemHeap) {
    // deep copy
    for (auto &field : fields) {
      void *buf = heap_->Allocate(sizeof(Field));
      fields_.push_back(new(buf)Field(field));
    }
  }

  /**
   * Row used for deserialize
   */
  Row() = delete;

  /**
   * Row used for deserialize and update
   */
  Row(RowId rid) : rid_(rid), heap_(new SimpleMemHeap) {}

  /**
   * Row copy function
   */
  Row(const Row &other) : heap_(new SimpleMemHeap) {
    if (!fields_.empty()) {
      for (auto &field : fields_) {
        heap_->Free(field);
      }
      fields_.clear();
    }
    rid_ = other.rid_;
    for (auto &field : other.fields_) {
      void *buf = heap_->Allocate(sizeof(Field));
      fields_.push_back(new(buf)Field(*field));
    }
  }

  virtual ~Row() {
    delete heap_;
  }

  /**
   * Note: Make sure that bytes write to buf is equal to GetSerializedSize()
   */
  uint32_t SerializeTo(char *buf, Schema *schema) const;

  uint32_t DeserializeFrom(char *buf, Schema *schema);

  /**
   * For empty row, return 0
   * For non-empty row with null fields, eg: |null|null|null|, return header
size only
   * @return
   */
```

```
  uint32_t GetSerializedSize(Schema *schema) const;

  inline const RowId GetRowId() const { return rid_; }

  inline void SetRowId(RowId rid) { rid_ = rid; }

  inline std::vector<Field *> &GetFields() { return fields_; }

  inline Field *GetField(uint32_t idx) const {
    ASSERT(idx < fields_.size(), "Failed to access field");
    return fields_[idx];
  }

  inline size_t GetFieldCount() const { return fields_.size(); }

private:
  Row &operator=(const Row &other) = delete;

private:
  RowId rid_{};
  std::vector<Field *> fields_;    /** Make sure that all fields are created by
mem heap */
  MemHeap *heap_{nullptr};
  //static constexpr uint32_t ROW_MAGIC_NUM = 200209;
};
```

`Row` 是堆表中记录的单元。成员 `rid_` 维护了在堆表中的位置。`fields_` 维护了此纪录的各个属性值。`heap` 用于内存的分配，防止内存泄漏。

重要成员函数：

- `SerializeTo(*buf, *schema)`
  - 依次将 `Row` 中 `field` 的个数，`bitmap`，以及各个 `field` 的内容序列化，写入 `buf` 指向的内容中，其中调用已经完成的 `field` 的 `SerializeTo` 函数。
- `DeserializeFrom(*buf, *schema)`
  - 将 `buf` 指向的内容依次解释为 `field` 的个数，`bitmap` 以及 `field` 的序列化数据。其中调用已经完成的 `field` 的 `DeserializeFrom` 函数。
- `GetSerializedSize(*schema)`
  - 返回此 `row` 序列化后的字节偏移量。

Column 类

```
class Column {
  friend class Schema;

public:
  /*this constructor have no length parameter, it is not char type*/
  Column(std::string column_name, TypeId type, uint32_t index, bool nullable,
bool unique);
  /*this constructor have a length parameter, it is char type*/
  Column(std::string column_name, TypeId type, uint32_t length, uint32_t index,
bool nullable, bool unique);
  /*other column copy*/
```

```
  Column(const Column *other);

  std::string GetName() const { return name_; }

  uint32_t GetLength() const { return len_; }

  void SetTableInd(uint32_t ind) { table_ind_ = ind; }

  uint32_t GetTableInd() const { return table_ind_; }

  bool IsNullable() const { return nullable_; }

  bool IsUnique() const { return unique_; }

  TypeId GetType() const { return type_; }

  uint32_t SerializeTo(char *buf) const;

  void SetUnique() { unique_ = true; }

  uint32_t GetSerializedSize() const;
  /*deserialize is a static function*/
  static uint32_t DeserializeFrom(char *buf, Column *&column, MemHeap *heap);

private:
  static constexpr uint32_t COLUMN_MAGIC_NUM = 210928;
  std::string name_;
  TypeId type_;
  uint32_t len_{0};       // for char type this is the maximum byte length of
the string data,
  // otherwise is the fixed size
  uint32_t table_ind_{0}; // column position in table
  bool nullable_{false};  // whether the column can be null
  bool unique_{false};    // whether the column is unique
};
```

`Column` 维护了表的某一列的元信息。`name_` 是这一列的属性名。`type_` 是此列的类型名。`len_` 只在 `type_` 为字符串时有意义，标识了最大字符串长度。`table_ind_` 记录了此列在表中的位置。`nullable_` 和 `unique_` 分别标识此列是否可以为 `null` 和是否唯一。

重要成员函数：

- `SerializeTo(*buf)`
  - 依次将 `Column` 中的 `COLUMN_MAGIC_NUM`,`name_.length()`,`name_`,`type_`,`len_`,`table_ind_`,`nullable_`,`unique_` 写入 `buf` 指向的内容。
- `DeserializeFrom(*buf, *&column, *heap)`
  - 将 `buf` 指向的内容依次解释为 `COLUMN_MAGIC_NUM`,`name_.length()`,`name_`,`type_`,`len_`,`table_ind_`,`nullable_`,`unique_`。其中需要验证 `COLUMN_MAGIC_NUM`。
- `GetSerializedSize()`
  - 返回 `Column` 序列化的字节偏移量

Schema 类

```
class Schema {
public:
  explicit Schema(const std::vector<Column *> columns) :
columns_(std::move(columns)) {}

  inline const std::vector<Column *> &GetColumns() const { return columns_; }

  inline  Column *GetColumn(const uint32_t column_index) const { return
columns_[column_index]; }

  dberr_t GetColumnIndex(const std::string &col_name, uint32_t &index) const {
    for (uint32_t i = 0; i < columns_.size(); ++i) {
      if (columns_[i]->GetName() == col_name) {
        index = i;
        return DB_SUCCESS;
      }
    }
    return DB_COLUMN_NAME_NOT_EXIST;
  }

  inline uint32_t GetColumnCount() const { return static_cast<uint32_t>
(columns_.size()); }

  /**
   * Shallow copy schema, only used in index
   *
   * @param: attrs Column index map from index to tuple
   * eg: Tuple(A, B, C, D)  Index(D, A) ==> attrs(3, 0)
   */
  static Schema *ShallowCopySchema(const Schema *table_schema, const
std::vector<uint32_t> &attrs, MemHeap *heap) {
    std::vector<Column *> cols;
    cols.reserve(attrs.size());
    for (const auto i : attrs) {
      cols.emplace_back(table_schema->columns_[i]);
    }
    void *buf = heap->Allocate(sizeof(Schema));
    return new(buf) Schema(cols);
  }

  /**
   * Deep copy schema
   */
  static Schema *DeepCopySchema(const Schema *from, MemHeap *heap) {
    std::vector<Column *> cols;
    for (uint32_t i = 0; i < from->GetColumnCount(); i++) {
      void *buf = heap->Allocate(sizeof(Column));
      cols.push_back(new(buf)Column(from->GetColumn(i)));
    }
    void *buf = heap->Allocate(sizeof(Schema));
    return new(buf) Schema(cols);
  }
```

```
  /**
   * Only used in table
   */
  uint32_t SerializeTo(char *buf) const;

  /**
   * Only used in table
   */
  uint32_t GetSerializedSize() const;

  /**
   * Only used in table
   */
  static uint32_t DeserializeFrom(char *buf, Schema *&schema, MemHeap *heap);

private:
  static constexpr uint32_t SCHEMA_MAGIC_NUM = 200715;
  std::vector<Column *> columns_;   /** don't need to delete pointer to column
*/
};
```

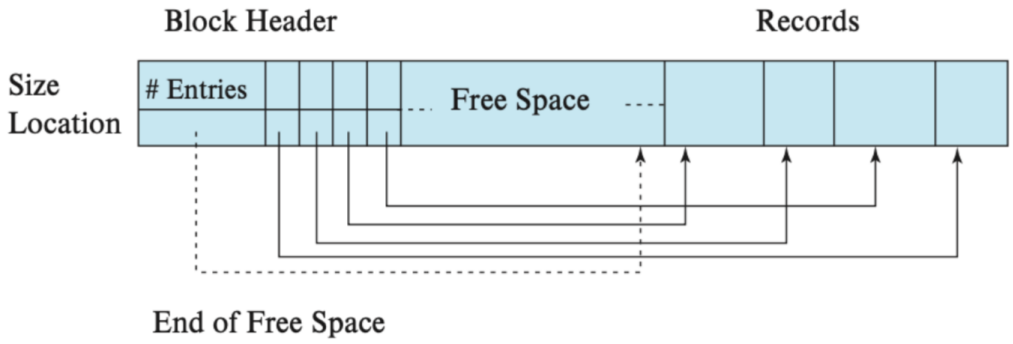
`Schema` 维护了一系列 `Column` 作为表的元信息。

重要成员函数

- `SerializeTo(*buf)`
  - 依次将 `SCHEMA_MAGIC_NUM`，`columns_.size()`，以及各个 `Column` 内容序列化，写入 `buf` 指向的内容，其中调用 `Column::SerializeTo(*buf)`。
- `DeserializeFrom(*buf, *&schema, *heap)`
  - 依次将 `buf` 指向的内容解释为 `SCHEMA_MAGIC_NUM`，`columns_.size()` 以及各个 `column` 的序列化数据，其中调用 `Column::DeserializeFrom(*buf, *&column, *heap)` 且须验证 `SCHEMA_MAGIC_NUM`。
- `GetSerializedSize()`
  - 返回 `Schema` 序列化的字节偏移量

TableHeap 类

堆表结构如下图所示：



在堆表中，页和页以链表的形式组织在一起。页的头部数据保存了本页页号，上一页的页号以及下一页的页号。堆表页的结构如下图所示：

```
/**
* Basic Slotted page format:
*  -----------------------------------------------------
*  | HEADER | ... FREE SPACE ... | ... INSERTED TUPLES ... |
*  -----------------------------------------------------
*                                ^
*                                free space pointer
*
*  Header format (size in bytes):
*  ----------------------------------------------------------------------------
*  | PageId (4)| LSN (4)| PrevPageId (4)| NextPageId (4)| FreeSpacePointer(4) |
*  ----------------------------------------------------------------------------
*  ---------------------------------------------------------------
*  | TupleCount (4) | Tuple_1 offset (4) | Tuple_1 size (4) | ... |
*  ---------------------------------------------------------------
**/
```

堆表的类定义：

```
class TableHeap {
  friend class TableIterator;

public:
  static TableHeap *Create(BufferPoolManager *buffer_pool_manager, Schema
*schema, Transaction *txn,
                           LogManager *log_manager, LockManager *lock_manager,
MemHeap *heap) {
    void *buf = heap->Allocate(sizeof(TableHeap));
    return new(buf) TableHeap(buffer_pool_manager, schema, txn, log_manager,
lock_manager);
  }

  static TableHeap *Create(BufferPoolManager *buffer_pool_manager, page_id_t
first_page_id, Schema *schema,
                           LogManager *log_manager, LockManager *lock_manager,
MemHeap *heap) {
    void *buf = heap->Allocate(sizeof(TableHeap));
    return new(buf) TableHeap(buffer_pool_manager, first_page_id, schema,
log_manager, lock_manager);
  }

  ~TableHeap() {}

  /**
   * Insert a tuple into the table. If the tuple is too large (>= page_size),
return false.
   * @param[in/out] row Tuple Row to insert, the rid of the inserted tuple is
wrapped in object row
   * @param[in] txn The transaction performing the insert
   * @return true iff the insert is successful
   */
  bool InsertTuple(Row &row, Transaction *txn);

  /**
   * Mark the tuple as deleted. The actual delete will occur when ApplyDelete is
called.
   * @param[in] rid Resource id of the tuple of delete
```

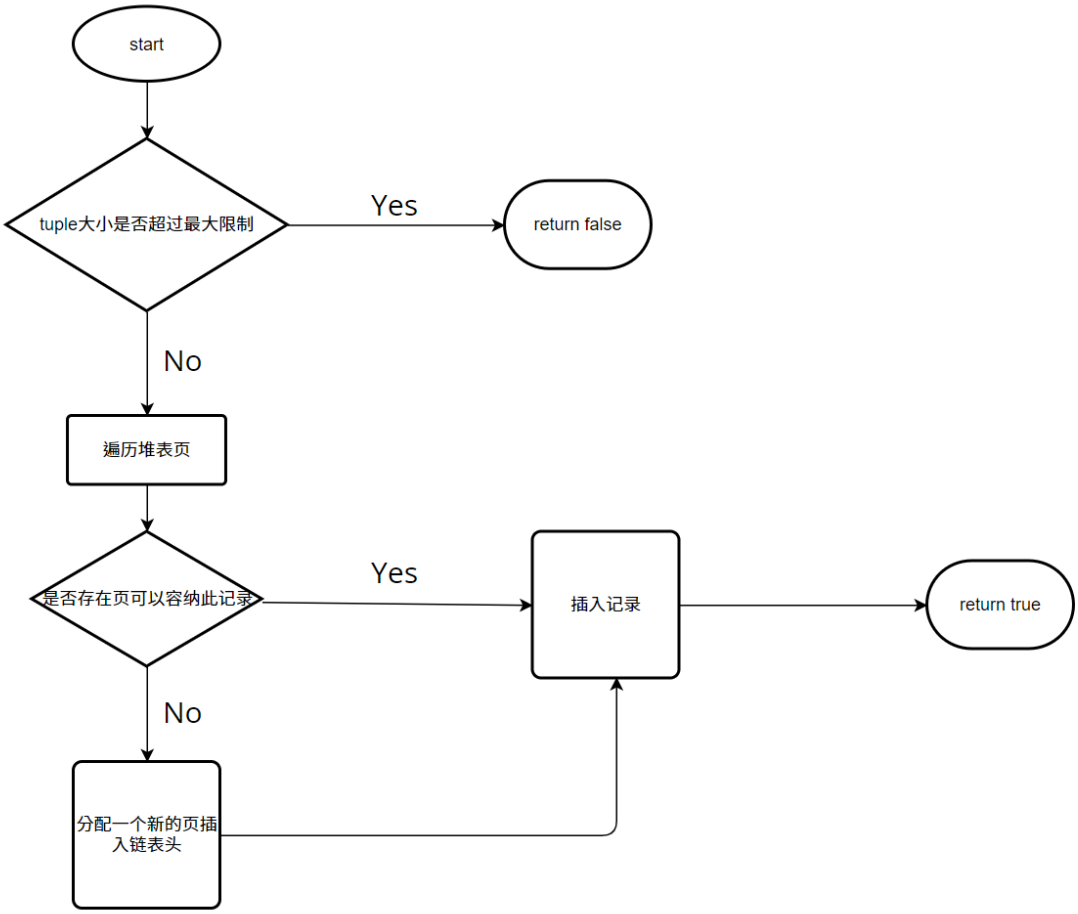
```
   * @param[in] txn Transaction performing the delete
   * @return true iff the delete is successful (i.e the tuple exists)
   */
  bool MarkDelete(const RowId &rid, Transaction *txn);

  /**
   * if the new tuple is too large to fit in the old page, return false (will
delete and insert)
   * @param[in] row Tuple of new row
   * @param[in] rid Rid of the old tuple
   * @param[in] txn Transaction performing the update
   * @return true is update is successful.
   */
  bool UpdateTuple(Row &row, RowId &rid, Transaction *txn);

  /**
   * Called on Commit/Abort to actually delete a tuple or rollback an insert.
   * @param rid Rid of the tuple to delete
   * @param txn Transaction performing the delete.
   */
  void ApplyDelete(const RowId &rid, Transaction *txn);

  /**
   * Called on abort to rollback a delete.
   * @param[in] rid Rid of the deleted tuple.
   * @param[in] txn Transaction performing the rollback
   */
  void RollbackDelete(const RowId &rid, Transaction *txn);

  /**
   * Read a tuple from the table.
   * @param[in/out] row Output variable for the tuple, row id of the tuple is
wrapped in row
   * @param[in] txn transaction performing the read
   * @return true if the read was successful (i.e. the tuple exists)
   */
  bool GetTuple(Row *row, Transaction *txn);

  /**
   * Free table heap and release storage in disk file
   */
  void FreeHeap();

  /**
   * @return the begin iterator of this table
   */
  TableIterator Begin(Transaction *txn);

  /**
   * @return the end iterator of this table
   */
  TableIterator End();

  /**
   * @return the id of the first page of this table
```

```
   */
  inline page_id_t GetFirstPageId() const { return first_page_id_; }

private:
  /**
   * create table heap and initialize first page
   */
  explicit TableHeap(BufferPoolManager *buffer_pool_manager, Schema *schema,
Transaction *txn,
                     LogManager *log_manager, LockManager *lock_manager) :
          buffer_pool_manager_(buffer_pool_manager),
          first_page_id_(INVALID_PAGE_ID),
          schema_(schema),
          log_manager_(log_manager),
          lock_manager_(lock_manager){
  };

  /**
   * load existing table heap by first_page_id
   */
  explicit TableHeap(BufferPoolManager *buffer_pool_manager, page_id_t
first_page_id, Schema *schema,
                     LogManager *log_manager, LockManager *lock_manager)
          : buffer_pool_manager_(buffer_pool_manager),
            first_page_id_(first_page_id),
            schema_(schema),
            log_manager_(log_manager),
            lock_manager_(lock_manager) {}

private:
  BufferPoolManager *buffer_pool_manager_;
  page_id_t first_page_id_;
  Schema *schema_;
  [[maybe_unused]] LogManager *log_manager_;
  [[maybe_unused]] LockManager *lock_manager_;
};
```

关键成员函数：

- `InsertTuple`：

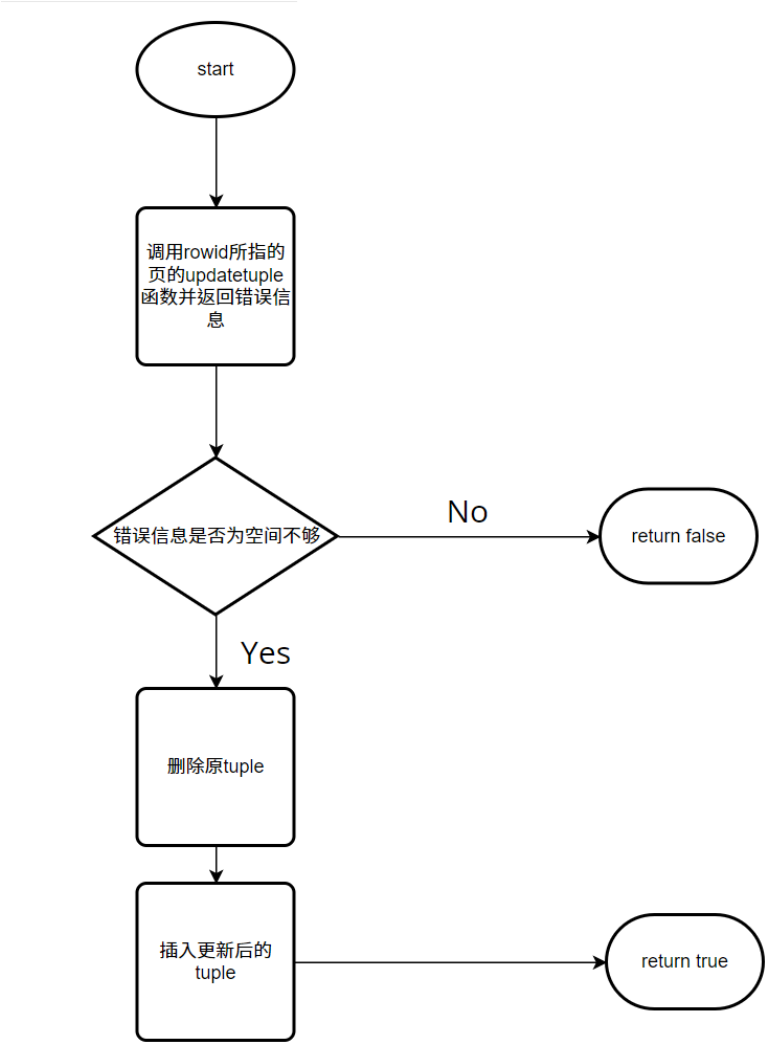
  流程图如下：



使用`first fit`策略找到第一个可以容纳此记录的数据页，若不存在则分配一个新页插入链表表头。插入此记录。

- `UpdateTuple`：

  流程图如下：



TableIterator 类

```
class TableIterator {

public:
  // you may define your own constructor based on your member variables
  //explicit TableIterator();
  TableIterator() = delete;

  explicit TableIterator(Row row, TablePage *table_page, TableHeap *table_heap,
Transaction *txn);

  explicit TableIterator(const TableIterator &other);

  ~TableIterator();

  inline bool operator==(const TableIterator &itr) const { return
row_.GetRowId() == itr.row_.GetRowId(); }

  inline bool operator!=(const TableIterator &itr) const { return !(itr ==
(*this)); }

  const Row &operator*();
```

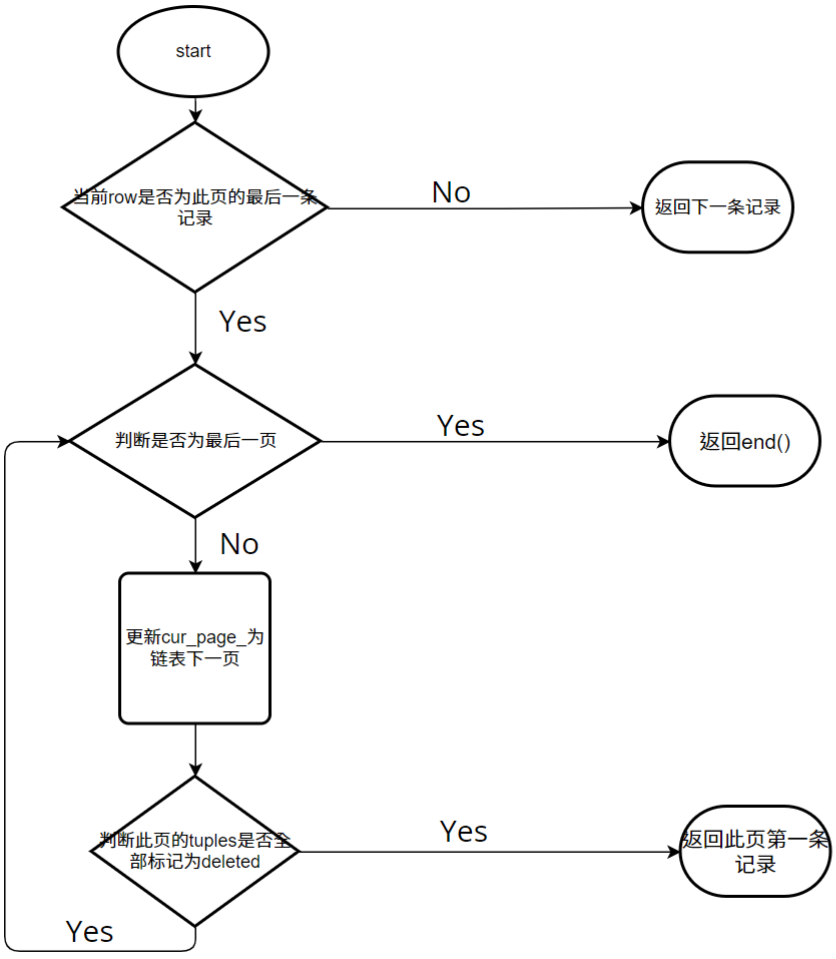
```
  Row *operator->();

  TableIterator &operator++();

  TableIterator operator++(int);

private:
  // add your own private member variables here
  Row row_;
  TablePage *cur_page_;
  TableHeap *table_heap_;
  Transaction *txn_;
};
```

堆表的迭代器维护`row_`来保存所指向的记录内容，维护`cur_page_`当前数据页的指针。

重要的成员函数：

- `TableIterator &operator++();`

流程图如下图所示：



使用重载`++`遍历所有堆表中的所有记录。迭代器的实现给上层模块提供了接口。

3.3 INDEX MANAGER

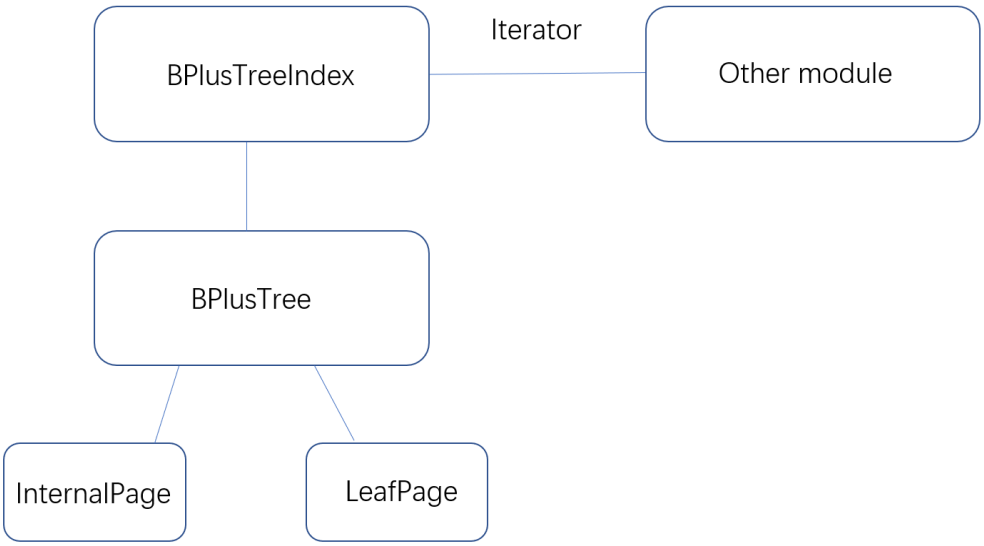
3.3.1 需求分析

在MiniSQL的设计中，仅仅使用遍历堆表线性查找记录效率低效。因此我们需要实现索引根据键值快速查找。

`Index Manager`负责数据表索引的实现和管理，包括：索引的创建和删除，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。

具体实现时，我们选择b+树的数据结构构建索引。

3.3.2 架构设计



`BPlusTreeIndex`提供了迭代器和接口供上层调用。`BPlusTreeIndex`自身维护一颗b+树以及索引信息。

3.3.3 实现细节

B+树数据页

BPlusTreePage 类

```
class BPlusTreePage {
public:
  bool IsLeafPage() const;

  bool IsRootPage() const;

  void SetPageType(IndexPageType page_type);

  int GetSize() const;

  void SetSize(int size);

  void IncreaseSize(int amount);

  int GetMaxSize() const;

  void SetMaxSize(int max_size);
```

```
  int GetMinSize() const;

  page_id_t GetParentPageId() const;

  void SetParentPageId(page_id_t parent_page_id);

  page_id_t GetPageId() const;

  void SetPageId(page_id_t page_id);

  void SetLSN(lsn_t lsn = INVALID_LSN);

private:
  // member variable, attributes that both internal and leaf page share
  [[maybe_unused]] IndexPageType page_type_;
  [[maybe_unused]] lsn_t lsn_;
  [[maybe_unused]] int size_;
  [[maybe_unused]] int max_size_;
  [[maybe_unused]] page_id_t parent_page_id_;
  [[maybe_unused]] page_id_t page_id_;
};
```

在b+树的实现中，我们将一个数据页作为b+树的一个结点。`BPlusTreePage` 作为 `BPlusTreeInternalPage` 和 `BPlusTreeLeafPage` 的父类，提供了两种数据页一般性的数据内容，如此页类型（为叶结点还是内部结点），此页页号 `page_id_`,父页页号 `parent_page_id_`，以及所容纳的键值对个数和上限。

BPlusTreeInternalPage 类

```
INDEX_TEMPLATE_ARGUMENTS
class BPlusTreeInternalPage : public BPlusTreePage {
public:
  // must call initialize method after "create" a new node
  void Init(page_id_t page_id, page_id_t parent_id = INVALID_PAGE_ID, int
max_size = INTERNAL_PAGE_SIZE);

  KeyType KeyAt(int index) const;

  void SetKeyAt(int index, const KeyType &key);

  //here I add a new function SetValueAt(int index, const ValueType& value);
  void SetValueAt(int index, const ValueType &value);


  int ValueIndex(const ValueType &value) const;

  ValueType ValueAt(int index) const;

  ValueType Lookup(const KeyType &key, const KeyComparator &comparator) const;

  void PopulateNewRoot(const ValueType &old_value, const KeyType &new_key, const
ValueType &new_value);

  int InsertNodeAfter(const ValueType &old_value, const KeyType &new_key, const
ValueType &new_value);
```

```
  void Remove(int index);

  ValueType RemoveAndReturnOnlyChild();

  // Split and Merge utility methods
  void MoveAllTo(BPlusTreeInternalPage *recipient, const KeyType &middle_key,
BufferPoolManager *buffer_pool_manager);

  /*buffer_pool_manager here is to change those child's parent_id*/
  void MoveHalfTo(BPlusTreeInternalPage *recipient, BufferPoolManager
*buffer_pool_manager);
  //void MoveHalfTo(BPlusTreeInternalPage *recipient);

  void MoveFirstToEndOf(BPlusTreeInternalPage *recipient, const KeyType
&middle_key,
                        BufferPoolManager *buffer_pool_manager);

  void MoveLastToFrontOf(BPlusTreeInternalPage *recipient, const KeyType
&middle_key,
                         BufferPoolManager *buffer_pool_manager);

private:
  void CopyNFrom(MappingType *items, int size, BufferPoolManager
*buffer_pool_manager);

  void CopyLastFrom(const MappingType &pair, BufferPoolManager
*buffer_pool_manager);

  void CopyFirstFrom(const MappingType &pair, BufferPoolManager
*buffer_pool_manager);

  MappingType array_[0];
};
```

内部结点中维护了一个键值对数组，即键值-指针，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，每个内部结点至少是半满的（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。

内部结点中重要的成员函数：

- `Lookup(key, comparator)`:查找对应某个key的值，在具体实现时为指针，即`page_id`。使用二分查找实现。
- `MoveHalfTo(*recipient, *buffer_pool_manager)`:将其中一半的`pair`转移到`recipient`中，用于分裂。
- `MoveFirstToEndOf(*recipient, &middle_key, *buffer_pool_manager)`:将此页第1个元素移动到`recipient`末尾。用于删除时的重新分配。
- `MoveLastToFrontOf(*recipient, &middle_key, *buffer_pool_manager)`:将此页最后1个的元素移动到`recipient`末尾。用于删除时的重新分配。

- `MoveAllTo(*recipient, const &middle_key, *buffer_pool_manager)`：将此页中的所有元素移动到`recipient`中。用于删除时的合并操作。

BPlusTreeLeafPage 类

```
INDEX_TEMPLATE_ARGUMENTS
class BPlusTreeLeafPage : public BPlusTreePage {
public:
  // After creating a new leaf page from buffer pool, must call initialize
  // method to set default values
  void Init(page_id_t page_id, page_id_t parent_id = INVALID_PAGE_ID, int
max_size = LEAF_PAGE_SIZE);

  // helper methods
  page_id_t GetNextPageId() const;

  void SetNextPageId(page_id_t next_page_id);

  KeyType KeyAt(int index) const;

  int KeyIndex(const KeyType &key, const KeyComparator &comparator) const;

  const MappingType &GetItem(int index);

  // insert and delete methods
  int Insert(const KeyType &key, const ValueType &value, const KeyComparator
&comparator);


  /*I add a int& index to store the return index*/
  bool Lookup(const KeyType &key, ValueType &value, const KeyComparator
&comparator) const;

  int RemoveAndDeleteRecord(const KeyType &key, const KeyComparator
&comparator);

  // Split and Merge utility methods
  void MoveHalfTo(BPlusTreeLeafPage *recipient);

  void MoveAllTo(BPlusTreeLeafPage *recipient);

  void MoveFirstToEndOf(BPlusTreeLeafPage *recipient);

  void MoveLastToFrontOf(BPlusTreeLeafPage *recipient);

private:
  void CopyNFrom(MappingType *items, int size);

  void CopyLastFrom(const MappingType &item);

  void CopyFirstFrom(const MappingType &item);

  page_id_t next_page_id_;
  MappingType array_[0];
};
```

和内部节点类似，叶节点中按照顺序存储m个键和m个值。此外叶节点通过链表的形式连接。`next_page_id_`存放下一页的页号。这种组织形式有利于b+树数据的遍历，对后续模块的范围查找提供了接口支持。

叶节点重要成员函数：

- `MoveHalfTo(*recipient)`:将其中一半的`pair`转移到`recipient`中，用于分裂。
- `MoveFirstToEndOf(*recipient)`:将此页第1个元素移动到`recipient`末尾。用于删除时的重新分配。
- `MoveLastToFrontOf(*recipient)`:将此页最后1个的元素移动到`recipient`末尾。用于删除时的重新分配。
- `MoveAllTo(*recipient)`: 将此页中的所有元素移动到`recipient`中。用于删除时的合并操作。

叶节点的在分裂合并时和内部节点稍有不同，内部节点还需要使用`buffer_pool_manager`维护儿子的`parent_page_id`的更改。

BPlusTree 类

```
class BPlusTree {
  using InternalPage = BPlusTreeInternalPage<KeyType, page_id_t, KeyComparator>;
  using LeafPage = BPlusTreeLeafPage<KeyType, ValueType, KeyComparator>;

public:
  explicit BPlusTree(index_id_t index_id, BufferPoolManager
*buffer_pool_manager, const KeyComparator &comparator,
                     int leaf_max_size = LEAF_PAGE_SIZE, int internal_max_size =
INTERNAL_PAGE_SIZE );

  // Returns true if this B+ tree has no keys and values.
  bool IsEmpty() const;

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

  // Remove a key and its value from this B+ tree.
  void Remove(const KeyType &key, Transaction *transaction = nullptr);

  // return the value associated with a given key
  bool GetValue(const KeyType &key, std::vector<ValueType> &result, int&
position, page_id_t &leaf_page_id, Transaction *transaction = nullptr);

  INDEXITERATOR_TYPE Begin();

  INDEXITERATOR_TYPE Begin(const KeyType &key);

  INDEXITERATOR_TYPE End();

  // expose for test purpose
  Page *FindLeafPage(const KeyType &key, bool leftMost = false);

  // used to check whether all pages are unpinned
  bool Check();
```

```
  // destroy the b plus tree
  void Destroy();

  void PrintTree(std::ofstream &out) {
    if (IsEmpty()) {
      return;
    }
    out << "digraph G {" << std::endl;
    Page *root_page = buffer_pool_manager_->FetchPage(root_page_id_);
    BPlusTreePage *node = reinterpret_cast<BPlusTreePage *>(root_page);
    ToGraph(node, buffer_pool_manager_, out);
    out << "}" << std::endl;
  }

private:
  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);

  //In Destroy function
  void DestroyPage(BPlusTreePage *page);

  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);

  /* Debug Routines for FREE!! */
  void ToGraph(BPlusTreePage *page, BufferPoolManager *bpm, std::ofstream &out)
const;

  void ToString(BPlusTreePage *page, BufferPoolManager *bpm) const;

  // member variable
```

```
  index_id_t index_id_;
  page_id_t root_page_id_;
  BufferPoolManager *buffer_pool_manager_;
  KeyComparator comparator_;
  int leaf_max_size_;
  int internal_max_size_;
};
```

使用`BPlusTree`实现b+树。

成员说明：

- `index_id_`：b+树所属索引的id
- `root_page_id_`:b+树根结点页号，初始化为`INVALID_PAGE_ID`
- `comparator_`：用于比较键值
- `leaf_max_size_`：叶节点最大键值对个数
- `internal_max_size_`：内部节点最大键值对个数

b+树插入算法：

伪代码如下：

```
procedure insert(value K, pointer P)
if (tree is empty) create an empty leaf node L, which is also the root
else InsertIntoLeaf(K,P)
end
```

```
procedure InsertIntoLeaf(value K, pointer P)
Find the target leaf L
if(L can hold one more value ) insert pair(K,P)
else begin
Split the leaf L as L, L1
insert pair(K,P) into L or L1
InsertIntoParent(L,L1)
end
end
```

```
procedure InsertIntoParent(node N, node N1)
if(N is the root)
    then begin
        create a new node R containing the pointer to N and N1, and the values.
        make N, N1 as R's children
        make R as the bplustree's root
        return
    end
Let P=N's parent
if(P can hold one more child)
    then make N1 as P's child
else begin
    Split P as P,P1
    make N1 as P or P1's child
    InsertIntoParent(P,P1)
end
end
```

在真正实现b+树时，我们采取维护内部节点第一个键值的做法。

b+树删除算法

伪代码如下：

```
procedure remove(value K, pointer P)
if(the tree is empty) then return
Find the target leaf L that should contains  (K,P)
delete the pair(K,P) in L
update the value in L's parent up towards root if neccessary
if(the size of L is less than the minimum size)
    then CoalesceOrRedistribute(L)
end
```

```
procedure CoalesceOrRedistribute(Node N)
if(N is the root and N has only one child)
    then adjust the root to the child of N
else begin
    Let L1 = the previous or next child of parent(N)
    if(L can borrow a value from L1)
        then Redistribute(L,L1)
    else if(L1 and L can be fit into a node)
        then begin
        Coalesce(L1,L)
        if(the size of the parent of L is less than the minimum size)
            then CoalesceOrRedistribute(L's parent)
        end
end
end
```

b+树索引迭代器

```
INDEX_TEMPLATE_ARGUMENTS
class IndexIterator {
public:
  // you may define your own constructor based on your member variables
  using LeafPage = BPlusTreeLeafPage<KeyType, ValueType, KeyComparator>;

  explicit IndexIterator(LeafPage *target_leaf, int index, BufferPoolManager
*buffer_pool_manager);

  explicit IndexIterator(page_id_t leaf_page_id, int position, BufferPoolManager
*buffer_pool_manager);

  ~IndexIterator();

  /** Return the key/value pair this iterator is currently pointing at. */
  const MappingType &operator*();

  /** Move to the next key/value pair.*/
  IndexIterator &operator++();
```

```
  /** Return whether two iterators are equal */
  bool operator==(const IndexIterator &itr) const;

  /** Return whether two iterators are not equal. */
  bool operator!=(const IndexIterator &itr) const;

private:
  // add your own private member variables here
  LeafPage* target_leaf_;
  int index_;
  BufferPoolManager *buffer_pool_manager_;//for unpin the page and fetch page
};
```

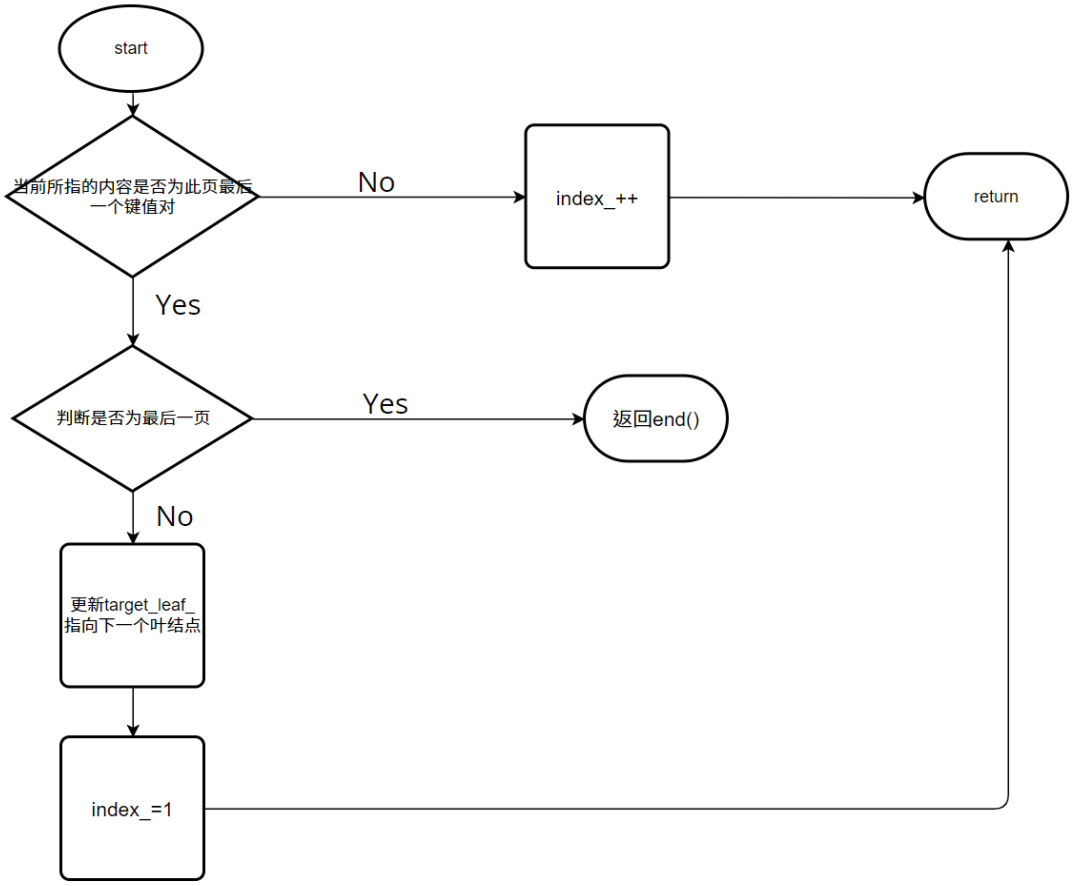
成员说明:

- `target_leaf_`: 叶节点指针
- `index_`:此迭代器所指内容在叶节点中的下标

重要操作符重载:

- `IndexIterator &operator++()`

流程图如下:



3.4 CATALOG MANAGER

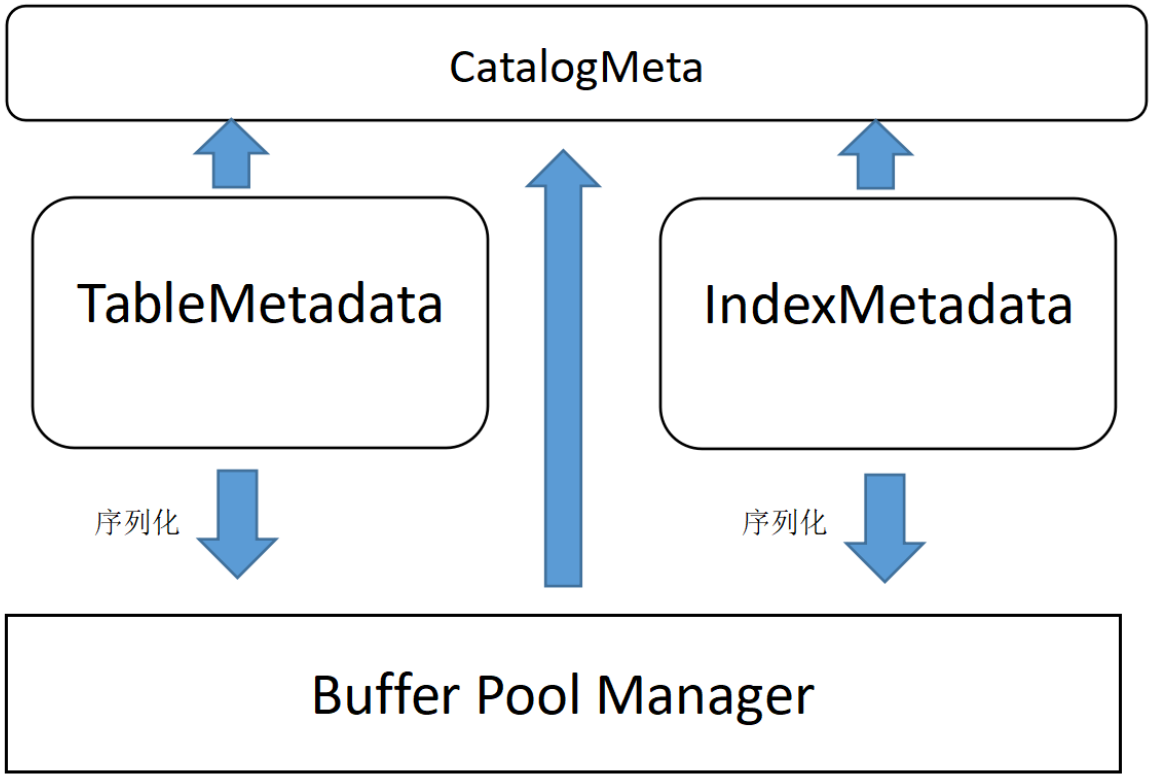
3.4.1 需求分析

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。

这些模式信息在被创建、修改和删除后还应被持久化到数据库文件中。此外，Catalog Manager还需要为上层的执行器Executor提供公共接口以供执行器获取目录信息并生成执行计划。

3.4.2 架构设计



3.4.3 实现细节

table类中的设计细节

TableMetadata类

```
class TableMetadata {
  friend class TableInfo;//table信息存在内存中的形式

public:
  uint32_t SerializeTo(char *buf) const;//序列化

  uint32_t GetSerializedSize() const;//求解大小

  static uint32_t DeserializeFrom(char *buf, TableMetadata *&table_meta, MemHeap
*heap);//反序列化

  static TableMetadata *Create(table_id_t table_id, std::string table_name,
page_id_t root_page_id, TableSchema *schema,
```

```
                                vector<Column> primary_key, MemHeap *heap);  //创
建一个元信息

  inline table_id_t GetTableId() const { return table_id_; }

  inline std::string GetTableName() const { return table_name_; }

  inline uint32_t GetFirstPageId() const { return root_page_id_; }

  inline Schema *GetSchema() const { return schema_; }

  uint32_t GetPrimaryKeyCount() const { return primarykey.size(); }

  inline const std::vector<Column> &GetPrimaryKey() const { return primarykey; }

 private://初始化
  TableMetadata() = delete;

  TableMetadata(table_id_t table_id, std::string table_name, page_id_t
root_page_id, TableSchema *schema,
                vector<Column> primary_key);

private:
  static constexpr uint32_t TABLE_METADATA_MAGIC_NUM = 344528;//检验序列化的
  table_id_t table_id_;
  std::string table_name_;
  page_id_t root_page_id_;
  Schema *schema_;
  std::vector<Column> primarykey;
}
```

TableInfo类

```
class TableInfo {
public:
  static TableInfo *Create(MemHeap *heap) {
    void *buf = heap->Allocate(sizeof(TableInfo));
    return new(buf)TableInfo();
  }

  ~TableInfo() {
    delete heap_;
  }

  void Init(TableMetadata *table_meta, TableHeap *table_heap) {
    table_meta_ = table_meta;
    table_heap_ = table_heap;
  }

  inline TableHeap *GetTableHeap() const { return table_heap_; }

  inline MemHeap *GetMemHeap() const { return heap_; }

  inline table_id_t GetTableId() const { return table_meta_->table_id_; }
```

```
  inline std::string GetTableName() const { return table_meta_->table_name_; }

  inline Schema *GetSchema() const { return table_meta_->schema_; }

  inline vector<Column> GetPrimaryKey() const { return table_meta_->primarykey;
}

  inline page_id_t GetRootPageId() const { return table_meta_->root_page_id_; }

  //inline vector<Column> GetPrimarykey() const { return primarykey; }

 // inline void CreatePrimarykey(vector<Column> &primarykey) { this->primarykey
= primarykey; }

 //inline vector<Column> GetPrimaryKey() const { return primarykey; }

  //inline void CreatePrimaryKey(vector<Column> &primarykey) { this->primarykey
= primarykey; }

  inline void SetRootPageId() {
    this->table_meta_->root_page_id_ = this->table_heap_->GetFirstPageId();
  }

  //I add this function
  inline TableMetadata *GetTableMeta() { return table_meta_; }

private:
  explicit TableInfo() : heap_(new SimpleMemHeap()) {};

private:
  TableMetadata *table_meta_;
  TableHeap *table_heap_;
  MemHeap *heap_; /** store all objects allocated in table_meta and table heap
*/
  //vector<Column> primarykey;
};
```

TableMetadata类维护了表的相关信息，包括表的id，name，在数据页中存放的首页id，表的内容以及它的主键的信息等，表的这些元信息都通过序列化的形式写到数据页中。在需要使用到相关数据时通过反序列操作得到结果。TableInfo类是表在内存中的表现形式，当调用表时，通常都是通过TableInfo来指向所需要操作的表，它维护了表的元信息和操作对象。

重要的成员函数

`TableMetadata::SerializeTo(*buf)`

- 依次将table_id,*table_name*,root_page_id,schema_以及primarykey序列化，其中schema_调用Schema::SerializeTo(*buf)序列化，并写入buf指向的内容。

`TableMetadata::GetSerializedSize()`

- 返回TableMetadata序列化的字节偏移量。

`TableMetadata::DeserializeFrom(*buf,*&table_meta, *heap)`

- 将buf指向的内容依次反序列化成table_id,*table_name*,root_page_id,schema_调用

Schema::DeserializeFrom(*buf*,&schema, *heap),以及primarykey。

indexes类中的设计细节

IndexMetadata类

```
class IndexMetadata {
  friend class IndexInfo;

public:
  static IndexMetadata *Create(const index_id_t index_id, const std::string
&index_name,
                               const table_id_t table_id, const
std::vector<uint32_t> &key_map,
                               MemHeap *heap);

  uint32_t SerializeTo(char *buf) const;

  uint32_t GetSerializedSize() const;

  static uint32_t DeserializeFrom(char *buf, IndexMetadata *&index_meta, MemHeap
*heap);

  inline std::string GetIndexName() const { return index_name_; }

  inline table_id_t GetTableId() const { return table_id_; }

  uint32_t GetIndexColumnCount() const { return key_map_.size(); }

  inline const std::vector<uint32_t> &GetKeyMapping() const { return key_map_; }

  inline index_id_t GetIndexId() const { return index_id_; }

private:
  IndexMetadata() = delete;

  explicit IndexMetadata(const index_id_t index_id, const std::string
&index_name,
                         const table_id_t table_id, const std::vector<uint32_t>
&key_map) :
      index_id_(index_id),
      index_name_(index_name),
      table_id_(table_id),
      key_map_(key_map){
  }

private:
  static constexpr uint32_t INDEX_METADATA_MAGIC_NUM = 344528;
  index_id_t index_id_;
  std::string index_name_;
  table_id_t table_id_;
  std::vector<uint32_t> key_map_;  /** The mapping of index key to tuple key */
};
```

IndexInfo类

```
class IndexInfo {
public:
  static IndexInfo *Create(MemHeap *heap) {
    void *buf = heap->Allocate(sizeof(IndexInfo));
    return new(buf)IndexInfo();
  }

  ~IndexInfo() {
    delete heap_;
  }
  //初始化参数，并且通过meta_data算出别的
  void Init(IndexMetadata *meta_data, TableInfo *table_info, BufferPoolManager
*buffer_pool_manager) {
    meta_data_ = meta_data;
    table_info_ = table_info;
    key_schema_ = key_schema_->ShallowCopySchema(table_info->GetSchema(),
meta_data->GetKeyMapping(), heap_);
    index_ = CreateIndex(buffer_pool_manager);
    // Step1: init index metadata and table info
    // Step2: mapping index key to key schema
    // Step3: call CreateIndex to create the index
  }

  inline Index *GetIndex() { return index_; }

  inline std::string GetIndexName() { return meta_data_->GetIndexName(); }

  inline IndexSchema *GetIndexKeySchema() { return key_schema_; }

  inline MemHeap *GetMemHeap() const { return heap_; }

  inline TableInfo *GetTableInfo() const { return table_info_; }

  inline IndexMetadata *GetIndexMeta() const { return meta_data_; }

 private:
  explicit IndexInfo() : meta_data_{nullptr}, index_{nullptr},
table_info_{nullptr},
                         key_schema_{nullptr}, heap_(new SimpleMemHeap()) {}

  Index *CreateIndex(BufferPoolManager *buffer_pool_manager) {
    using INDEX_KEY_TYPE = GenericKey<32>;
    using INDEX_COMPARATOR_TYPE = GenericComparator<32>;
    using BP_TREE_INDEX = BPlusTreeIndex<INDEX_KEY_TYPE, RowId,
INDEX_COMPARATOR_TYPE>;
    void *buf = heap_->Allocate(sizeof(BP_TREE_INDEX));
    return new (buf) BP_TREE_INDEX(meta_data_->GetIndexId(), key_schema_,
buffer_pool_manager);
  }

private:
  IndexMetadata *meta_data_;//元信息（其他三个均由反序列化后的元信息生成）
  Index *index_;//索引对象
```

```
  TableInfo *table_info_;//表格信息
  IndexSchema *key_schema_;//索引模式
  MemHeap *heap_;
};
```

IndexMetadata类维护了索引的相关信息，包括索引的id，name，索引所在表的id及索引与数据内容的对应map等，索引的这些元信息都通过序列化的形式写到数据页中。在需要使用到相关数据时通过反序列操作得到结果。IndexInfo类是索引在内存中的表现形式，当调用索引时，通常都是通过IndexInfo来指向所要操作的索引，它维护了索引的元信息和操作对象。

重要的成员函数

`IndexMetadata::SerializeTo(*buf)`

- 依次将IndexMetadata中的index_id,*index_name*,table_id,*key_map*写入buf指向的内容。

`IndexMetadata::GetSerializedSize()`

- 返回IndexMetadata序列化的字节偏移量。

`IndexMetadata::DeserializeFrom(*buf,*&index_meta, *heap)`

- 将buf指向的内容依次反序列化成index_id,*index_name*,table_id,*key_map*。

`IndexInfo::Init(*index_meta_data,&table_info,*buffer_pool_manager)`

- 通过index_meta_data,table_info,调用Schema::ShallowCopySchema(table_info->GetSchema(),meta_data->GetKeyMapping, heap*)得到key_schema*来初始化IndexInfo。
  调用CreateIndex(buffer_pool_manager)来创建一个索引对象。

Catalog类中的设计细节

CatalogMeta类

```
class CatalogMeta {
  friend class CatalogManager;

public:
  void SerializeTo(char *buf) const;

  static CatalogMeta *DeserializeFrom(char *buf, MemHeap *heap);

  uint32_t GetSerializedSize() const;

  inline table_id_t GetNextTableId() const {
    return table_meta_pages_.size() == 0 ? 0 : table_meta_pages_.rbegin()->first;
  }

  inline index_id_t GetNextIndexId() const {
    return index_meta_pages_.size() == 0 ? 0 : index_meta_pages_.rbegin()->first;
  }

  static CatalogMeta *NewInstance(MemHeap *heap) {
    void *buf = heap->Allocate(sizeof(CatalogMeta));
    return new(buf) CatalogMeta();
```

```
  }

  /**
   * Used only for testing
   */
  inline std::map<table_id_t, page_id_t> *GetTableMetaPages() {
    return &table_meta_pages_;
  }

  /**
   * Used only for testing
   */
  inline std::map<index_id_t, page_id_t> *GetIndexMetaPages() {
    return &index_meta_pages_;
  }

private:
  explicit CatalogMeta();

private:
  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_;
};
```

我们需要一个数据页和数据对象`CatalogMeta`来记录和管理这些表和索引的元信息被存储在哪个数据页中。`CatalogMeta`的信息将会被序列化到数据库文件的第`CATALOG_META_PAGE_ID`号数据页中（逻辑意义上），`CATALOG_META_PAGE_ID`默认值为0。

重要的成员函数如下：

`CatalogMeta::SerializeTo(*buf)`

- 依次将CatalogMeta中记录的两个map，分别是用来记录表的id与它所在的数据页和索引的id与它所在的数据页，序列化到数据库文件中的第CATALOG_META_PAGE_ID号数据页中。

`CatalogMeta::GetSerializedSize()`

- 返回CatalogMeta序列化的字节偏移量。

`CatalogMeta::DeserializeFrom(*buf,*&index_meta, *heap)`

- 将buf指向的内容依次反序列化成两个map：table_meta_pages,index_meta_pages。

CatalogManager类

```
class CatalogManager {
public:
  explicit CatalogManager(BufferPoolManager *buffer_pool_manager, LockManager
*lock_manager,
                          LogManager *log_manager, bool init);

  ~CatalogManager();

  MemHeap *GetHeap() { return heap_; }
```

```
  dberr_t CreateTable(const std::string &table_name, TableSchema *schema,
std::vector<Column> primary_key,
                      Transaction *txn, TableInfo *&table_info);

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

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

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

  dberr_t GetIndex(const std::string &table_name, const std::string &index_name,
IndexInfo *&index_info) const;

  dberr_t GetTableIndexes(const std::string &table_name, std::vector<IndexInfo *>
&indexes) const;

  dberr_t DropTable(const std::string &table_name);

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

private:
  dberr_t FlushCatalogMetaPage() const;

  dberr_t LoadTable(const table_id_t table_id, const page_id_t page_id);

  dberr_t LoadIndex(const index_id_t index_id, const page_id_t page_id);

  dberr_t GetTable(const table_id_t table_id, TableInfo *&table_info);

private:
  [[maybe_unused]] BufferPoolManager *buffer_pool_manager_;
  [[maybe_unused]] LockManager *lock_manager_;
  [[maybe_unused]] LogManager *log_manager_;
  [[maybe_unused]] CatalogMeta *catalog_meta_;
  [[maybe_unused]] std::atomic<table_id_t> next_table_id_;
  [[maybe_unused]] std::atomic<index_id_t> next_index_id_;
  // map for tables
  std::unordered_map<std::string, table_id_t> table_names_;
  std::unordered_map<table_id_t, TableInfo *> tables_;
  // map for indexes: table_name->index_name->indexes
  [[maybe_unused]] std::unordered_map<std::string,
std::unordered_map<std::string, index_id_t>> index_names_;
  [[maybe_unused]] std::unordered_map<index_id_t, IndexInfo *> indexes_;
  // memory heap
  MemHeap *heap_;
};
```

CatalogManager类维护和持久化数据库中所有表和索引的信息。在数据库初次打开时初始化所有数据，在后 续打开时加载所有表和索引的信息。同时该类还需对上层模块提供对指定数据表和指定数据索引的操作方式。

重要的成员函数如下：

`CatalogManager::CatalogManager(BufferPoolManager *buffer_pook_manager,LockManager *lock_manager,LogManager *log_manager,bool init)`

- 在第一次调用时初始化创建变量catalog_meta_，在后续调用时通过在缓冲池中获取数据页反序列化catalog_meta。*通过得到的catalog_meta*反序列化得到所有的tableinfo，indexinfo，并且更新关于表和索引的两个map。

`CatalogManager::~CatalogManager()`

- 将更改过的新的表的数据和索引的数据重新序列化到内存池中。

`CatalogManager:CreateTable(const string &table_name,TableSchema *schema,std::vector<Column>primary_key,Transaction *txn,TableInfo *&table_info)`

- 创建一个新的表，创建这个表的元信息，并将它序列化到内存池中，同时创建一个指向表的信息的指针，并且在CatalogManager中的map中更新相关的值。

`CatalogManager:GetTable(const string &table_name, TableInfo *&table_info)`

- 在记录表信息的两个map里寻找所需要的表的信息，并且将指向它的表信息的指针返回。

`CatalogManager:GetTables(vector<TableInfo *>&tables) const`

- 找到多张表的信息并返回。

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

- 先检查是否存在一个在该表里的相同的指针。通过指向索引信息的指针创建一个key_map，并且创建索引的元信息，序列化到内存池中，更新指向表的信息的指针，并且在CatalogManager中的map中更新相关的值。最后根据表的内容和前面得到的内容创建这个新的索引。

`CatalogManager::GetIndex(const std::string &table_name,const string &index_name,IndexInfo *&index_info)`

- 在记录index信息的两个map里寻找需要的索引的信息并返回。

`CatalogManager::GetTableIndexs(const std::string &table_name,vector<IndexInfo *>&indexs)`

- 找到多个索引的信息并返回。

`CatalogManager:DropTable(const string &table_name)`

- 删除表的所有信息及它包含的索引的信息

`CatalogManager:DropIndex(const string &table_name,const string &index_name)`

- 删除某个表中的相关索引信息

`CatalogManager:FlushCatalogMetaPage()const`

- 更新存放catalog序列化的数据页的信息

`CatalogManager:LoadTable(const table_id_t table_id,const page_id_t page_id)`

- 更新catalog数据中记录表的id与之对应序列化的数据页的id的map

`CatalogManager:LoadIndex(const index_id_t index_id,const page_id_t page_id)`

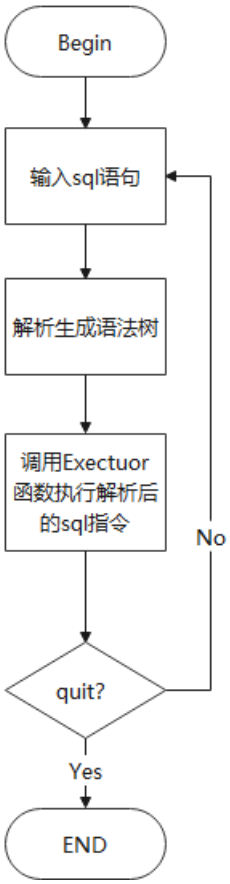
- 更新catalog数据中记录索引的id与之对应序列化的数据页的id的map

3.5 Executor

3.5.1 需求分析

`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

3.5.2 架构设计



3.5.3 实现细节

`ExecuteEngine`**类**

```
class ExecuteEngine {
public:
  ExecuteEngine();

  ~ExecuteEngine() {
    for (auto it : dbs_) {
      delete it.second;
    }
  }
  /**
   * executor interface
```

```
   */
  dberr_t Execute(pSyntaxNode ast, ExecuteContext *context);
private:
  dberr_t ExecuteCreateDatabase(pSyntaxNode ast, ExecuteContext *context);
  dberr_t ExecuteDropDatabase(pSyntaxNode ast, ExecuteContext *context);
  dberr_t ExecuteShowDatabases(pSyntaxNode ast, ExecuteContext *context);
  dberr_t ExecuteUseDatabase(pSyntaxNode ast, ExecuteContext *context);
  dberr_t ExecuteShowTables(pSyntaxNode ast, ExecuteContext *context);
  dberr_t ExecuteCreateTable(pSyntaxNode ast, ExecuteContext *context);
  dberr_t ExecuteDropTable(pSyntaxNode ast, ExecuteContext *context);
  dberr_t ExecuteShowIndexes(pSyntaxNode ast, ExecuteContext *context);
  dberr_t ExecuteCreateIndex(pSyntaxNode ast, ExecuteContext *context);
  dberr_t ExecuteDropIndex(pSyntaxNode ast, ExecuteContext *context);
  dberr_t ExecuteSelect(pSyntaxNode ast, ExecuteContext *context);
  dberr_t ExecuteInsert(pSyntaxNode ast, ExecuteContext *context);
  dberr_t ExecuteDelete(pSyntaxNode ast, ExecuteContext *context);
  dberr_t ExecuteUpdate(pSyntaxNode ast, ExecuteContext *context);
  dberr_t ExecuteTrxBegin(pSyntaxNode ast, ExecuteContext *context);
  dberr_t ExecuteTrxCommit(pSyntaxNode ast, ExecuteContext *context);
  dberr_t ExecuteTrxRollback(pSyntaxNode ast, ExecuteContext *context);
  dberr_t ExecuteExecfile(pSyntaxNode ast, ExecuteContext *context);
  dberr_t ExecuteQuit(pSyntaxNode ast, ExecuteContext *context);
  CmpBool Travel(TableInfo *currenttable, TableIterator &tableit, pSyntaxNode
root);
  CmpBool TravelWithoutIndex(TableInfo *currenttable, TableIterator &tableit,
pSyntaxNode root);
  dberr_t NewTravel(DBStorageEngine *Currentp, TableInfo *currenttable,
                            pSyntaxNode root, vector<RowId> *result);
private:
  bool isRecons;
  [[maybe_unused]] std::unordered_map<std::string, DBStorageEngine *> dbs_;  /**
all opened databases */
  [[maybe_unused]] std::string current_db_;  /** current database */
};
```

其中`isRecons`变量用来记录是否已经将内存中已有的database进行重建。

构造函数和析构函数

```
// 将isRecons初始化为false，只有第一条语句时需要检查内存并且重建先前的database
ExecuteEngine::ExecuteEngine() { isRecons = false; }
// 释放当前已经打开的数据库
ExecuteEngine::~ExecuteEngine() {
    for (auto it : dbs_) {
      delete it.second;
    }
}
```

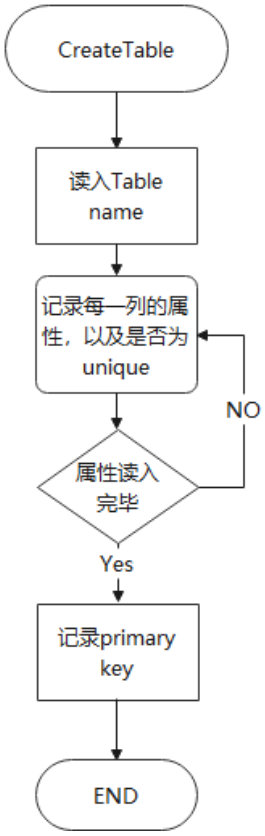
Database相关函数

- `dberr_t ExecuteEngine::ExecuteCreateDatabase`
  - 先在`databasefile.txt`找是否已经有同名数据库
  - 建立数据库，插入`unordered_map dbs_`中，记录当前的数据库
- `dberr_t ExecuteEngine::ExecuteDropDatabase`

    - 在`dbs_`中找到该数据库，删除
    - 在文件`databasefile.txt`中删除
- `dberr_t ExecuteEngine::ExecuteShowDatabases`
    - 打印`db_`中已有的数据库
- `dberr_t ExecuteEngine::ExecuteUseDatabase`
    - 找到database并把它作为`current_db_`

Tables相关函数

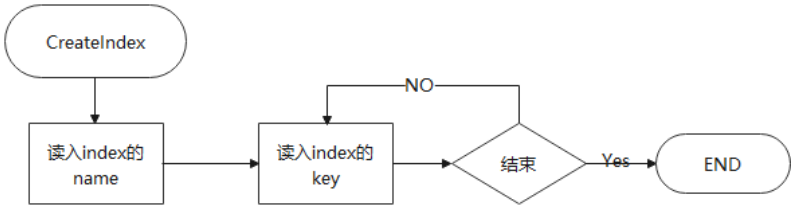
- `ExecuteEngine::ExecuteShowTables(*ast, *context)`
    - 在`Current Database`中的`catalog_mgr_`中的函数`GetTables()`得到当前`use`的数据库中的表名，输出。
- `ExecuteEngine::ExecuteCreateTable(*ast, *context)`
    - 遍历语法树将column和type的信息收集，形成一个列的`vector`
    - 注意`primary key`，该列不能有重复的元素
    - 在`Current database`中的`catalog_mgr_`中调用`CreateTable()`来新建一个table



- `ExecuteEngine::ExecuteDropTable(*ast, *context)`
    - 从语法树中得到需要被drop的table name
    - 调用`current database`中的`catalog_mgr_`中的`DropTable()`函数来drop

Index相关函数

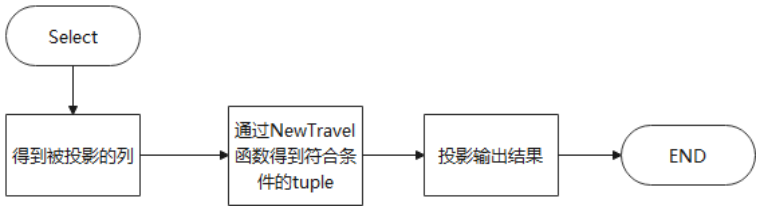
- `ExecuteEngine::ExecuteShowIndexes(*ast, *context)`
    - 用`catalog_mgr_`中的`GetTableIndexes()`来得到表的索引并且打印
- `ExecuteEngine::ExecuteCreateIndex(*ast, *context)`



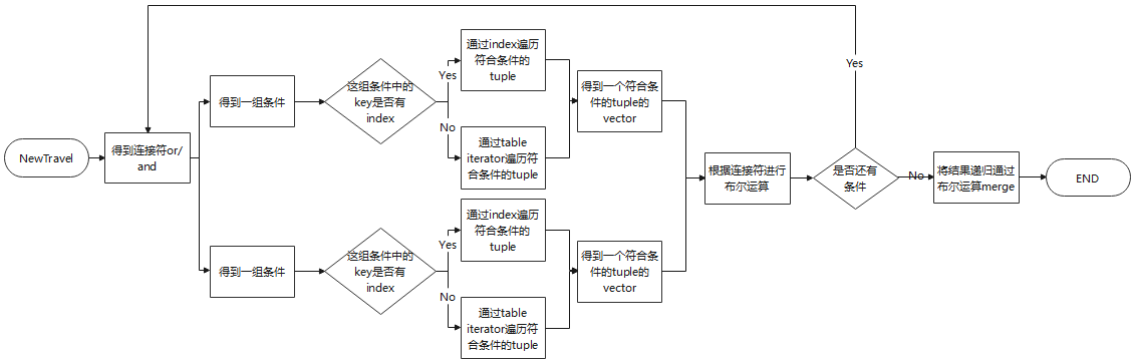
  - 检查是否在唯一键上建立索引，如果不是输出提示语
  - 调用 `current database` 中的 `catalog_mgr_` 中的 `CreateIndex()` 函数
- `ExecuteEngine::ExecuteDropIndex(*ast, *context)`
  - 在现有的tables中寻找同名的index
  - 调用 `current database` 中的 `catalog_mgr_` 中的 `DropIndex()` 函数

增删查改相关函数

- `ExecuteEngine::ExecuteSelect(*ast, *context)`



  `NewTravel()` 函数是以语法树条件部分的根节点为参数，通过筛选条件后进行布尔运算，得到符合条件的元组。(在insert、delete、update中也需要用这个方法得到符合条件的tuple)



- `ExecuteEngine::ExecuteInsert(*ast, *context)`
  - 调用 `current database` 中的 `catalog_mgr_` 中的 `GetTable()` 函数找到要被插入的表
  - 通过语法树得到需要被插入的一条记录的type以及内容（即上文讲的 `NewTravel()` 函数），与该表的column做检查
  - 检查unique以及primary key，是否有冲突，如果有，输入提示语，插入失败
  - 调用 `current table` 中的 `GetTableHeap()` 中的 `InsertTuple()` 函数插入一条记录
  - 如果被修改的列上有index则更新index
- `ExecuteEngine::ExecuteDelete(*ast, *context)`
  - 调用 `current database` 中的 `catalog_mgr_` 中的 `GetTable()` 函数找到要被删除记录的表
  - 通过类似于 `select` 中的方法来找到需要被删除的 `vector<RowId>` 然后通过迭代器删除
  - 如果被修改的列上有index则更新index
- `ExecuteEngine::ExecuteUpdate(*ast, *context)`
  - 与 `select`、`insert`、`delete` 中原理一致
  - 更新也需判断unique和primary key等约束条件
  - 如果被修改的列上有index则更新index

执行文件

- `ExecuteEngine::ExecuteExecfile(*ast, *context)`
  - 与`main.cpp`中的内容相同，将文件中的每一行指令解析语法树后通过`Execute()`执行

结束

```
dberr_t ExecuteEngine::ExecuteQuit(pSyntaxNode ast, ExecuteContext *context) {
#ifdef ENABLE_EXECUTE_DEBUG
  LOG(INFO) << "ExecuteQuit" << std::endl;
#endif
  ASSERT(ast->type_ == kNodeQuit, "Unexpected node type.");
  context->flag_quit_ = true;
  return DB_SUCCESS;
}
```

第4章 测试结果与展示

项目最终成功实现了一个精简型MiniSQL，在项目中我们通过插入30000条数据和进行相关SQL操作，来检查是否需要实现的功能成功实现。下面是测试结果的具体的展示：

数据库和表的创建

插入数据库

- ```
  minisql > create database db0;
  cost time: 0.045836
  minisql > create database db1;
  cost time: 0.017084
  minisql > show tables;
  cost time: 7.6e-05
  minisql > show databases;
  Database:
  db1
  db0
  cost time: 0.000198
  ```

创建数据表

正确创建：

- ```
  minisql > create table t1(a int, b char(20) unique, c float, primary key(a, c));
  cost time: 0.002038
  ```
- ```
  minisql > create table student(
    sno char(8),
    sage int,
    sab float unique,
    primary key (sno, sab)
  );
  cost time: 0.000555
  ```

错误创建1：字符串长度<=0

- ```
  minisql > create table t1(a int, b char(0) unique, c float, primary key(a, c));
  字符串长度<=0
  ```

错误创建2： 类型不同

- ```
  minisql > create table t1(a int, b char(-5) unique, c float, primary key(a, c));
  字符长度不是整数
  cost time: 0.000125
  minisql > create table t1(a int, b char(3.69) unique, c float, primary key(a, c));
  字符长度不是整数
  cost time: 0.000191
  minisql > create table t1(a int, b char(-0.69) unique, c float, primary key(a, c));
  字符长度不是整数
  cost time: 0.000152
  ```
  ```

插入数据

正确插入：

- ```
  minisql > insert into t1 values(1, "aa", 1.2);
  cost time: 0.000254
  ```

错误插入1：类型错误

- ```
  minisql > insert into t1 values(1, 1, 2);
  Wrong Type!
  cost time: 0.000157
  ```

错误插入2：输入错误

- ```
  minisql > insert into t1 values(2);
  Wrong Input!
  cost time: 0.000199
  ```

select操作检查

select *操作：

- ```
  minisql > create database db0;
  cost time: 0.00844
  minisql > use db0;
  cost time: 3.6e-05
  minisql > create table t1(a int, b char(20) unique, c float, primary key(a, c));
  cost time: 0.001325
  minisql > insert into t1 values(1, "dsdghrdr", 2);
  cost time: 0.000604
  minisql > insert into t1 values(2, "pdoppppp", 3);
  cost time: 0.000413
  minisql > insert into t1 values(4, "aaaaaa", 9);
  cost time: 0.000326
  minisql > insert into t1 values(5, "uisodod", 7);
  cost time: 0.000336
  minisql > insert into t1 values(8, "uisdlgsa", 8);
  cost time: 0.000383
  minisql > select * from t1;
  1 dsdghrdr 2.000000
  -------------
  2 pdoppppp 3.000000
  -------------
  4 aaaaaa 9.000000
  -------------
  5 uisodod 7.000000
  -------------
  8 uisdlgsa 8.000000
  -------------
  cost time: 0.001166
  ```

单条件select：

- ```
  minisql > select a from t1 where c > 7;
  5
  cost time: 0.000918
  minisql > select c from t1 where a = 2;
  6.000000
  cost time: 0.00061
  ```

多条件select：

- ```
  minisql > select a from t1 where b = "ii" or c = 9;
  2
  5
  cost time: 0.001038
  minisql > select a from t1 where b <> "ii" and c = 9;
  5
  cost time: 0.000623
  ```

null条件select：

- ```
  minisql > select a from t1 where b not null;
  1
  2
  5
  cost time: 0.000543
  ```

唯一约束检查
  ```

primarykey重复插入：

- ```
  minisql > insert into t1 values(1, "sd", 1.2);
  对于primary key列，不应该插入重复的元组
  cost time: 0.000465
  ```

索引不在唯一键上的创建：

- ```
  minisql > create index idx1 on t1(a);
  只能在唯一键上建立索引
  cost time: 0.000115
  ```

索引的创建与删除

索引的创建

- ```
  minisql > create index id1 on t1(b);
  cost time: 0.000797
  minisql > show indexes;
  t1 b
  ```

索引的删除

- ```
  minisql > drop index id1;
  cost time: 0.000153
  minisql > show indexes;
  t1 c
  ```

Update操作检查

正确update

- ```
  minisql > update t1 set b = "bbbbb" where a =2;
  cost time: 0.000184
  ```
- ```
  minisql > select * from t1;
  1 aaaU 1.100000
  -------------
  2 bbbbb 3.000000
  -------------
  ```

错误update1：与primary key冲突

```
minisql > update t1 set a = 1 where a = 2;
cost time: 0.000405
minisql > select * from tq;
cost time: 1.1e-05
minisql > select * from t1;
1 aaaU 1.100000
-------------
1 bbbbb 3.000000
-------------
cost time: 0.000166
minisql > update t1 set c = 1.1 where b = "bbbbb";
conflict with primary key!
cost time: 0.000218
```

错误update2：与unique冲突

```
minisql > update t1 set b = "aaa" where c = 3;
对于Unique列，不应该插入重复的元组
cost time: 0.000148
```

删除操作

数据库的删除

```
minisql > drop database db0
;
cost time: 0.004111
minisql > show databases;
Database:
No database.
cost time: 2.8e-05
```

数据表的删除

```
minisql > show tables;
t
cost time: 7.7e-05
minisql > drop table t;
cost time: 0.000147
minisql > show tables;
cost time: 9e-06
```

数据的删除

```
minisql > delete from t1 where b = "bbbbb" and c = 3;
cost time: 0.000309
minisql > select * from t1;
1 aaaU 1.100000
-------------
cost time: 0.000204
```

Quit

- 

第5章 总结与展望

通过对MiniSQL的精简版的设计，我们更加了解数据库系统底层的具体设计流程，对数据库系统中磁盘的具体管理细节及存储的数据组成，缓冲区的实现细节和优先替换算法LRU策略，负责数据表中数据管理的record manager的具体操作细节，优化数据库的b+树索引实现，管理数据库模式信息的catalog，和最终的接口执行器都有了深刻的理解，在实现的过程中，我们面对挑战，不仅仅是每个模块的成功实现，更多的是将五个模块的成功连接，其中涉及许多不同模块的函数的调用，这一部分内容比较繁杂，以至于在其中我们耗费了许多时间。

在实现MiniSQL的过程中，我们还仍有许多不足，许多bonus我们并没有来得及实现，在数据库的性能优化方面也做的不够，在表的管理方面，我们采用的是共享表空间的方式，该方式虽然便于管理，但对大数据的操作存在着空间浪费等问题，如何优化表的管理是未来我们可以思考的事情，还有怎么更进一步的加快MiniSQL的操作过程也有待思考。

总结来看，在这次项目设计中，我们对数据库的相关知识更加了解，同时也第一次对实现一个大型项目的过程了有了更切实的体会。虽然仍有更多有待进步，但目前已经实现了一个功能较为完整的MiniSQL。这次项目的设计我们每一位组员都受益匪浅。

第6章 工作分配

蒋思超：

`DISK AND BUFFER POOL MANAGER` 1.2 1.3部分及相关报告

`RECORD MANAGER` 部分及相关报告

`INDEX MANAGER` 部分及相关报告

王晨雨：

`DISK AND BUFFER POOL MANAGER` 1.4部分及相关报告

`SQL EXECUTOR` 部分及相关报告

王婕：

`DISK AND BUFFER POOL MANAGER` 1.5部分及相关报告

`CATALOG MANAGER` 部分及相关报告