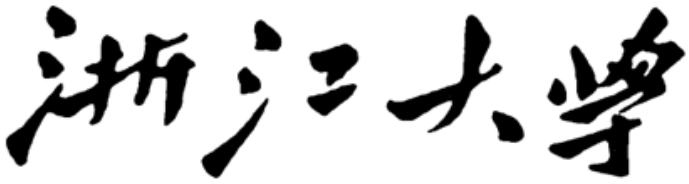


数据库系统实验报告

项目名称: MiniSQL

姓　　名: 李谞远

学　　号: 3220102970

电子邮箱: 497410282@qq.com

联系电话: 15268453576

指导老师: 苗晓晔

日　　期: 2024-06-06

目录

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，提高学生的系统编程能力，加深对数据库管理系统底层设计的理解。

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。

3 实验框架和环境配置

本实验基于 CMU-15445 BusTub 框架，并做了一些修改和扩展。

3.1 编译环境

- `gcc` & `g++`：9.4.0 (Linux)，使用 `gcc --version` 和 `g++ --version` 查看
- `cmake`：3.16.3 (Linux)，使用 `cmake --version` 查看
- `gdb`：9.2 (Linux)，使用 `gdb --version` 查看

3.2 开发环境

使用 VS Code 作为 IDE，在 WSL 环境下开发，WSL 使用 Ubuntu 子系统，版本为 20.04.6。

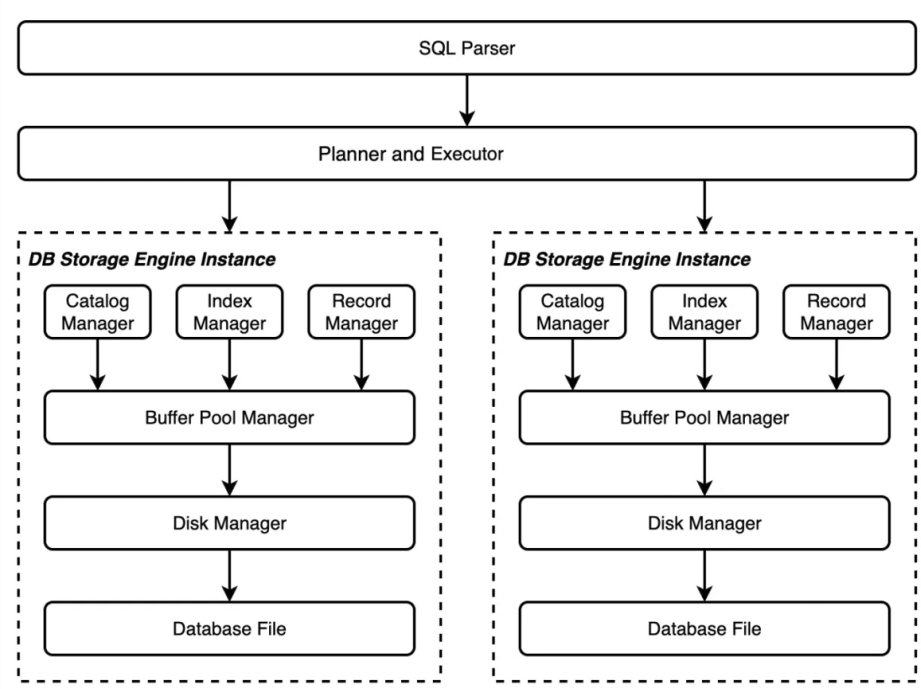
3.3 工程目录

- `src`：与 MiniSQL 工程相关的头文件和源代码。`src/include` 中为 MiniSQL 各个子模块的头文件，`src/buffer`、`src/record`、`src/index`、`src/catalog` 等目录为 MiniSQL 各个子模块的源代码。
- `test`：与测试用例相关的源代码和头文件。
- `thirdparty`：第三方库，包括日志模块 `glog` 和测试模块 `gtest`。

4 系统架构

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）。



5 模块描述

实现了除 Lock Manager 外的其余所有模块，列举如下：

- Disk Manager
- Buffer Pool Manager
- Record Manager
- Index Manager
- Catalog Manager
- Planner and Executor
- Recovery Manager

以下是对各模块的功能和实现的详细描述。

5.1 Disk Manager

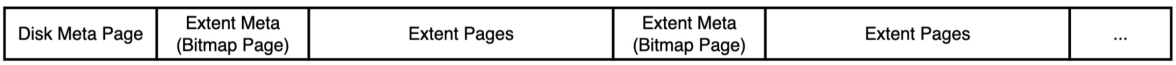
5.1.1 位图页的实现

位图是一种数据结构，位图中每一个比特（Bit）对应一个页的分配情况，用于标记改数据页是否空闲。位图页的大小为 4096 字节，即一个数据页的大小。位图页由元信息和具体的位图信息组成，元信息包括位图已经分配的页数和下一个空闲页的位置。由于一个字节有八位，因此位图信息可以保存八倍字节数的数据页分配信息。

此部分实现了分配一个空闲页，回收一个已分配的页和判断页是否空闲三个函数，基本通过位运算来完成，由于代码十分简单，因此不在此处赘述。

5.1.2 磁盘数据页管理

由于一个位图页约能管理 2^{15} 个数据页，这些数据页只能储存约 128MB 的信息，因此磁盘管理数据页采用嵌套的方式，即把一个位图页和它能管理的数据页看作一个分区，然后加一个磁盘元信息页，来维护分区个数，总分配页数等信息，这样可以管理的数据页就大大增加了，几乎为原先的 1000 倍，如下图所示：



此外，该部分存在的问题是，由于位图页的存在，真正存储数据的数据页是不连续的，因此当上层传入逻辑页号获取数据页时，需额外实现一个逻辑页号到物理页号的映射，使得上层对 `Disk Manager` 的页分配是无感知的。因为每个位图页能维护的数据页个数是固定的，因此该映射仅需要一个简单的除法运算即可。

该部分也实现了分配、回收和空闲判断三个函数，传入参数为逻辑页号。我们先定位到对应的分区，然后对元数据页进行更新，同时，从磁盘中读出位图页，也进行相应的更新。另外，我们维护一个 `next_free_extent_`，用来快捷地找到可以分配的分区，避免了每次从头遍历的开销。

5.2 Buffer Pool Manager

5.2.1 LRU 替换策略

在实现缓冲池前，我们先实现 `Replacer`，其作用是跟踪 `Buffer Pool Manager` 中数据页的分配情况，以及在 `Buffer Pool Manager` 没有空闲页时，决定替换哪一个空闲页。

LRU 全称“最近最久未使用的置换算法”，LRU 算法的核心思想是，最近最闲，意味着未来可能也是最闲，最没有价值。具体来说，在代码中，我们维护一个 `lru_list_`，顺序存放 `Buffer Pool Manager` 可以被替换的页的位置，每当一个页被访问，我们将其从 `Replacer` 中移除（因为页在使用时不允许被替换），当最后一个进程结束对它的访问时，我们再将其加入列表的头，每次需要替换页面时，我们取出列表尾部的元素，就是最久没有被访问的位置，并将该位置提供给上层作替换用。

5.2.2 CLOCK 替换策略（Bonus）

CLOCK 替换策略是有别于 LRU 的另一种替换策略，其通过一个环状链表来维护类似于 LRU 方法的 list，在该环状链表中，每一个位置保存两个值，分别代表该位置是否可替换（没有进程访问），以及一个访问标记。当一个位置被加入 `Replacer` 时，其访问位初始为 1。

CLOCK 替换策略维护一个指针，每当需要替换页时，该指针循环遍历链表，如果遇到一个位置可替换，则判断其访问位，若访问位为 0，就返回该位置，如果访问位为 1，就将访问位置 0。该替换策略保证了每个页新被加入 `Replacer` 时，不会在下一轮立刻被替换，而是在第二次再被选中时被替换出去。

5.2.3 缓冲池管理

由于直接从硬盘中读写数据页的效率是非常低的，因此我们需要一个缓冲池，Buffer Pool Manager 负责从 Disk Manager 中获取数据页并将它们存储在内存中，从而加快对频繁使用的数据页的访问。

在 Buffer Pool Manager 中，初始所有页都是空闲的，我们用一个 free_list_ 来保存，如果 Buffer Pool Manager 还没满，我们每从磁盘中访问或是新建一个页，都直接从 free_list_ 中获取位置，如果 free_list_ 已经为空，此时就需要由 Replacer 来决定哪个页被换出去。

要注意的点有两个：

- 当页面正在被使用时，该页不允许被替换，我们对每个页维护一个 PinCount，表示当前有多少进程在使用该页，仅当该值为 0 时，才能加入 Replacer 作为可替换的选择。

- 对脏页要进行处理，每当要发生替换，要判断原先位置是否是脏页（即进程是否对其进行修改），如果是，那么要先将其写回磁盘，再进行替换。脏页在进程 Unpin 时，通过页的 is_dirty_ 变量来进行标记。

5.3 Record Manager

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。与记录（Record）相关的概念有以下几个：

- 列（Column）：在 src/include/record/column.h 中被定义，用于定义和表示数据表中的某一个字段，即包含了这个字段的字段名、字段类型、是否唯一等等；

- 模式（Schema）：在 src/include/record/schema.h 中被定义，用于表示一个数据表或是一个索引的结构。一个 Schema 由一个或多个的 Column 构成；

- 域（Field）：在 src/include/record/field.h 中被定义，它对应于一条记录中某一个字段的数据信息，如存储数据的数据类型，是否是空，存储数据的值等等；

- 行（Row）：在 src/include/record/row.h 中被定义，与元组的概念等价，用于存储记录或索引键，一个 Row 由一个或多个 Field 构成。

此外，与数据类型相关的定义和实现位于 src/include/record/types.h 中。

5.3.1 序列化与反序列化

序列化和反序列化操作实际上是将数据库系统中的对象（包括记录、索引、目录等）进行内外存格式转化的过程，前者将内存中的逻辑数据（即对象）通过一定的方式，转换成便于在文件中存储的物理数据，后者则从存储的物理数据中恢复出逻辑数据，两者的目的都是为了实现数据的持久化。

在本节中，实现了对 Row、Schema 和 Column 对象的序列化、反序列化和序列化大小三个功能的实现。序列化的方法十分简单，对于定长的数据，如 int，u_int 等，我们直接根据其长度，将其写入内存中；对于不定长的数据，为了方便其反序列化，我们在写入内存时，先将其长度写入，这样在反序列化时，我们可以先读出该数据的长度，再根据长度读取该数据。

5.3.2 堆表管理记录

对于数据表中的每条记录，都有一个 RowId，保存了 page_id 和 slot_num，通过该值可以方便的定位到某条记录在堆表中位于什么位置。

堆表（TableHeap）是一种将记录以无序堆的形式进行组织的数据结构，不同的数据页（TablePage）之间通过双向链表连接。堆表中的每个数据页采用分槽页的形式（见下图，能够支持存储不定长的记录），分槽页由表头（Table Page Header）、空闲空间（Free Space）和已经插入的数据（Inserted Tuples）三部分组成；表头在页中从左往右扩展，记录了 PrevPageId、NextPageId、FreeSpacePointer 以及每条记录在 TablePage 中的偏移和长度；插入的记录在页中从右向左扩展，每次插入记录时会将 FreeSpacePointer 的位置向左移动。

分槽页的形式，使得数据页的空间能够尽可能的被利用，并方便了数据的更新。下面将对堆表中几个重要的操作进行介绍：

- InsertTuple：在堆表中插入一条记录。由于堆表是链表的形式，因此插入的方法是，从第一页开始，遍历整个堆表，直到访问到第一个放得下该记录的数据页，将记录插入。如果访问到堆表末尾还没有插入，则新建数据页并插入。

- MarkDelete：对堆表中的一条记录标记为删除。由于需要回滚，因此记录不直接从堆表中删除，而是标记为删除。

- `UpdateTuple`：更新堆表中的一条记录。更新记录时，会遇到两种情况：一种是记录变小了，直接在原处进行更新；另一种是记录变大了，并且在原处无法存下，直接调用 `InsertTuple` 函数，重新插入该记录。

- `FreeTableHeap`：析构整个堆表，释放空间。通过遍历链表，依次释放每个数据页的空间，来保证整个堆表的空间被充分释放。

5.3.3 堆表迭代器

所实现的堆表迭代器中，维护了 `table_heap_` 和 `row_`，由于 `TablePage` 中已经实现好了 `GetFirstTupleRid` 和 `GetNextTupleRid`，迭代器的 ++ 操作，几乎只需要调用这两个函数就可以完成。在迭代器的 Begin 函数中，需要注意的是，数据页不一定有记录，需要找到第一个有记录的数据页。

5.4 Index Manager

`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。

由于通过遍历堆表的方式来查找一条记录是十分低效的。为了能够快速定位到某条记录而无需搜索数据表中的每一条记录，我们在上一个实验的基础上实现了一个索引，这能够为快速随机查找和高效访问有序记录提供基础。索引有很多种实现方式，如 B+ 树索引，Hash 索引等等。本次 MiniSQL 选择的索引是 B+ 树索引。

5.4.1 B+ 树数据页

B+ 树中的每个结点（Node）都对应一个数据页，用于存储 B+ 树结点中的数据。因此在本节中，实现了以下三种类型的 B+ 树结点数据页：

- `BPlusTreePage`：以下两类节点的父类，保存了一些两类节点共享的数据，比如节点类型，节点中保存的键值对数量、节点的 `page_id` 等等。

- `BPlusTreeInternalPage`：中间节点，不存储实际的数据，按顺序存储 m 个键和 $m+1$ 个指针（值），指针指向子节点的 `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。

- `BPlusTreeLeafPage`：叶节点，存储实际的数据，按顺序存储 m 个键和 m 个值，值实际上存储 `RowId` 的值，可以直接对应到堆表中的记录。

数据页中实现了若干方法，比如一些键值对的移动，以方便上层的实现。最重要的功能是二分查找，因为 B+ 树内的记录是有序的，因此可以二分查找，从而在插入和查询时快速定位到对应的节点处，保证了 B+ 树维护索引的时间复杂度。

5.4.2 B+ 树索引

本实验所设计的 B+ 树只能支持 `Unique Key`，这也意味着，当尝试向 B+ 树插入一个重复的 Key-Value 键值对时，将不能执行插入操作并返回 `false` 状态。

本实验框架中已经为我们实现好了一个 B+ 树索引类，以我们实现的 B+ 树类作为容器，以及，一个键值比较器 `KeyManager` 来实现上层的索引插入（`InsertEntry`），扫描（`ScanKey`）和删除（`RemoveEntry`）。

B+ 树是 B+ 树索引的核心，通过实现大量的方法，来使得 B+ 树在插入、删除时能够保持树的结构，以及在查找时能够快速定位到对应的位置。我们维护了一个额外的页称为 `IndexRootsPage`，用来保存和维护所有的 B+ 树根节点，并可以通过 `index_id` 来快速获取对应的 B+ 树根节点。

B+ 树的实现中，包含的比较重要的函数描述如下：

- `Init`：B+ 树的初始化。在这部分中，我们尝试从 `IndexRootsPage` 中获取对应的根节点，并确定 B+ 树的节点的最小最大键值对数量。键值对数量的最大值由 `KeySize` 决定，也就是一个页能够存储的最多键值对的数量。这里需要注意的是，我们需要将计算得到的最大键值对数量减一，来保证插入时不会超过 `PageSize`。

- `Destroy`：B+ 树的析构，类似于堆表的析构，我们需要递归地遍历整个 B+ 树，依次释放每个数据页的空间，保证 B+ 树的空间被充分释放。

- `Insert`：往 B+ 树中插入一个键值对。分两种情况，一种是 B+ 树初始为空，我们需要新建根节点；另一种是 B+ 树不为空，我们需要找到对应的叶子插入，对插入位置使用的也是二分查找。在插入时，我们需要保证 B+ 树的结构，即 B+ 树每个节点的键值对数量合法，因此如果插入叶子后超出了最大键值对数量，我们需要递归地进行分裂操作。

- `Split`：分裂一个 B+ 树节点。也分两种情况，一种是分裂的节点是根节点，我们需要新建一个根节点；另一种是分裂的节点不是根节点，我们需要将分裂的

键值对插入到父节点中，并递归地分裂父节点（如果需要）。分裂的过程是，我们将节点中的键值对分成两部分，一部分保留，一部分移动到新建的节点中，同时，分裂时我们需要更新各类指针，比如指向父节点指针等。

- `Remove`：删除 B+ 树中的一个键值对。删除操作也分两种情况，一种是删除后节点的键值对数量不满足最小值，我们需要递归地进行调整；另一种是删除后节点的键值对数量满足最小值，我们直接删除即可。

- `CoalesceOrRedistribute`：对 B+ 树的节点进行调整，具体为从兄弟取一个键值对或合并当前节点和兄弟节点。首先要对特殊情况进行判断，即对根节点进行特殊处理，如果根节点删空，则要从 `IndexRootsPage` 中删除根节点；如果根节点只剩一个子节点，则要将子节点设为新的根节点。否则我们根据兄弟节点的键值对数量是否足够，来决定是取一个键值对还是合并节点。取兄弟节点时，我们根据当前节点在父节点中的位置，来决定是取左兄弟还是右兄弟，代码中实现的方式是，如果是父节点中最左边的节点，就取右兄弟，否则取左兄弟。

  `Coalesce` 操作，由于合并了相邻节点，父亲节点的键值对数量会减少，因此我们需要递归地调整父亲节点，对父亲节点调用 `CoalesceOrRedistribute`。`Redistribute` 操作，由于取了兄弟节点的一个键值对，父亲节点的键值对数量不变，并且调用前就已经保证了兄弟节点的键值对数量足够，因此不需要递归调整。为了方便起见，这两个函数对中间节点和叶子节点实现了不同的操作，以方便函数调用。

5.4.3 B+ 树索引迭代器

由于 B+ 树的叶子节点是有序的，由链表的形式进行维护，因此我们可以通过 B+ 树的迭代器来实现对 B+ 树的顺序扫描。B+ 树的迭代器维护了 `page_id_` 和 `item_index_`，每次迭代器 ++ 操作，我们先判断是否到达了叶子节点的末尾，如果是，我们需要找到下一个叶子节点，否则，我们直接将 `item_index_` 加一即可。

5.5 Catalog Manager

`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、修改和删除后被持久化到数据库文件中。此外，Catalog Manager 还为上层的执行器提供公共接口以供执行器获取目录信息并生成执行计划。

5.5.1 目录元信息

数据库中定义的表和索引在内存中以 TableInfo 和 IndexInfo 的形式表现，其维护了与之对应的表或索引的元信息和操作对象。以 IndexInfo 为例，它包含了这个索引定义时的元信息 meta_data_，该索引对应的表信息 table_info_，该索引的模式信息 key_schema_ 和索引操作对象 index_。

简单来说，由 Catalog Manager 维护表和索引的一些元信息页，初始化时，即通过这些元信息页，实现重新打开数据库实例时，从数据库文件中加载所有的表和索引信息。

5.5.2 表和索引的管理

Catalog Manager 主要通过映射容器 unordered_map 来维护表和索引的信息。另外，Catalog Manager 还实现了大量接口供上层的 Executor 调用，比如创建表、删除表、创建索引、删除索引等等。这些操作的实现，主要是通过对元信息页的操作，来实现对表和索引的创建和删除。下面将对 Catalog Manager 中几个重要的操作进行介绍：

- CreateTable：创建一个表。在创建表时，我们需要先判断表是否已经存在，如果存在，返回错误；否则，我们需要创建表的元信息页，和表对应的堆表，然后利用这两者构造 table_info_。另外，建表时需要对主键和 unique 列建立索引，即调用 CreateIndex 函数。。
- CreateIndex：创建一个索引。在创建索引时，我们需要先判断索引是否已经存在，如果存在，返回错误；否则，我们需要创建索引的元信息页，然后利用索引的元信息页构造 index_info_。另外，因为建立索引时，数据库中可能已经有数据，因此我们需要对数据库中已有的数据进行索引的构建，即插入索引，因为我们 B+ 树的限制，我们不允许对有重复元素的属性建立索引。

5.6 Planner and Executor

本实验主要包括 Planner 和 Executor 两部分。Planner 的主要功能是将解释器（Parser）生成的语法树，改写成数据库可以理解的数据结构。在这个过程中，我们会将所有 SQL 语句中的标识符（Identifier）解析成没有歧义的实体，即各种 C++ 的类，并通过 Catalog Manager 提供的信息生成执行计划。Executor 遍历查询计划树，将树上的 PlanNode 替换成对应的 Executor，随后调用下层提供的相应接口进行执行。

5.6.1 Parser 生成语法树

在本实验中，Parser 模块已经由框架给出，功能是将输入的 SQL 语句解析成语法树。以下是语法树（结点）的数据结构定义，每个结点都包含了一个唯一标识符 id_，唯一标识符在调用 CreateSyntaxNode 函数时生成（框架中已经给出实现）。type_ 表示语法树结点的类型，line_no_ 和 col_no_ 表示该语法树结点对应的是 SQL 语句的第几行第几列，child_ 和 next_ 分别表示该结点的子结点和兄弟结点，val_ 用作一些额外信息的存储，比如表名、索引名等。

Parser 模块支持的 SQL 语句目前如下，一些语义上的错误（如 8-10 行），在 ExecuteEngine 中进行了处理：

```
create database db0;
drop database db0;
show databases;
use db0;
show tables;
create table t1(a int, b char(20) unique, c float, primary key(a, c));
create table t1(a int, b char(0) unique, c float, primary key(a, c));
create table t1(a int, b char(-5) unique, c float, primary key(a, c));
create table t1(a int, b char(3.69) unique, c float, primary key(a, c));
create table t1(a int, b char(-0.69) unique, c float, primary key(a, c));
create table student(
    sno char(8),
    sage int,
    sab float unique,
    primary key (sno, sab)
);
drop table t1;
create index idx1 on t1(a, b);
-- "btree" can be replaced with other index types
create index idx1 on t1(a, b) using btree;
drop index idx1;
show indexes;
select * from t1;
```

```
select id, name from t1;
select * from t1 where id = 1;
-- note: use left association
select * from t1 where id = 1 and name = "str";
select * from t1 where id = 1 and name = "str" or age is null and bb not null;
insert into t1 values(1, "aaa", null, 2.33);
delete from t1;
delete from t1 where id = 1 and amount = 2.33;
update t1 set c = 3;
update t1 set a = 1, b = "ccc" where b = 2.33;
quit;
execfile "a.txt";
```

5.6.2 Planner 生成执行计划

对于简单的语句例如 show databases，drop table 等，生成的语法树也非常简单，仅由一两个节点组成，这些操作由 ExecuteEngine 直接进行处理，不需要生成执行计划。对于复杂的语句，生成的语法树需传入 Planner 生成执行计划，并交由 Executor 进行执行。Planner 需要先遍历语法树，调用 Catalog Manager 检查语法树中的信息是否正确，如表、列是否存在，谓词的值类型是否与 column 类型对应等等，随后将这些词语抽象成相应的表达式，即可以理解的各种 C++ 类。解析完成后，Planner 根据改写语法树后生成的可以理解的 Statement 结构，生成对应的 Plannode，并将 Plannode 交由 Executor 进行执行。本模块的代码已经由框架给出。

5.6.3 Executor 执行计划

在拿到 Planner 生成的具体的查询计划后，就可以生成真正执行查询计划的一系列算子了。生成算子的步骤很简单，遍历查询计划树，将树上的 PlanNode 替换成对应的 Executor。

在本实验中，算子的执行模型为经典的火山模型（Iterator Model），执行引擎会将整个 SQL 构建成一个 Operator 树，查询树自顶向下的调用接口，数据则自底向上的被拉取处理。每一种操作会抽象为一个 Operator，每个算子都有 Init 和 Next 两个方法。Init 对算子进行初始化工作，Next 则是向下层算子请求下一条数据。当 Next 返回 false 时，则代表下层算子已经没有剩余数据，迭代结束。

本模块实现了五个算子，分别为 seqscan，insert，update，delete，indexscan。同时，在本模块中，只支持单列索引的扫描，当 Planner 检测到 select 的谓词中的列上存在索引，而且索引只包含该列时，会生成 IndexScanPlan，其他情况则生成 SeqScanPlan。本模块的代码同样由框架给出。

5.6.4 Executor 直接执行

该部分实现了对于简单操作的直接执行，编写该部分代码的过程是：首先运行 main 函数启动 MiniSQL，然后输入对应函数的 SQL 语句，由 Parser 解析 SQL 语句，生成语法树，我们将语法树以相应格式进行输出，放入可视化网站，根据观察结果，编写对应的执行代码。

在本模块中，实现了部分 SQL 语句的错误检查，比如插入语句，会对插入列名是否重复、主键是否存在等进行检查。同时对输出的结果表格形式进行了规范化，使得输出结果美观。代码较为简单，此处不再赘述。

5.7 Recovery Manager

Recovery Manager 负责管理和维护数据恢复的过程，包括：

- 日志结构的定义
- 检查点 CheckPoint 的定义
- 执行 Redo、Undo 等操作，处理插入、删除、更新，事务的开始、提交、回滚等日志，将数据库恢复到宕机之前的状态

5.7.1 数据恢复

出于实现复杂度的考虑，同时为了避免各模块耦合太强，Recovery Manager 模块被单独拆了出来。另外为了减少重复的内容，我们不重复实现日志的序列化和反序列化操作，只实现了一个纯内存的数据恢复模块。

此模块首先通过一个 LocRec 类来保存日志记录的操作类型、日志序号、对应的事务编号、上一个日志序号（维护一个日志记录的链表）和一些必要的操作信息。定义了 CheckPoint 检查点，包含当前数据库一个完整的状态，该结构由框架给出。在 RecoveryManager 类中，由我们实现了 Redo、Undo 等方法，用于日志的重做和撤销操作，重做和撤销均通过遍历日志序列（链表）来实现，代码较为简单，此处不再赘述。

5.7.2 思考题

如果不将此模块独立出来，真正做到数据库在任何时候断电都能恢复，同时支持事务的回滚，Recovery Manager 和 CheckPoint 机制应该怎样设计呢？

答：真正实现数据恢复，需要实现 ARIES 算法，需要添加 Log Buffer 和 Dirty Page Table。日志缓冲区也位于内存中，由于直接将日志写入磁盘，消耗的 IO 时间较长，因此需要内存中的日志缓冲区来加速。其写操作主要发生在如下三种情况：在事务提交时，要将该事务的所有日志记录从日志缓冲区写入到磁盘上的日志文件中；当日志缓冲区达到一定容量时，要将缓冲区中的日志记录刷新到磁盘，以腾出空间容纳新的日志记录；设置一个定期刷新间隔，定期将日志写入磁盘中。日志缓冲区能够避免日志逐条写入，大大增加了 IO 效率。另外一个需要添加的结构是脏页表，每个页需要维护 page_id、page_LSN、rec_LSN 等信息，用于记录页的修改情况，以及用于日志的重做和撤销操作。在进行各类修改操作时，各个模块都需要对所修改的页，维护上述各类 LSN 的信息，以便于日志的重做和撤销操作。

Recovery Manager 也需要进行修改，目前日志记录的都是物理操作，需要添加逻辑操作，以完善日志的重做和撤销。

CheckPoint 除了维护必要的活跃事务，还需要保存脏页表，以便于 ARIES 算法在分析时选取 redo_LSN 的值，提高恢复的效率。

5.8 Lock Manager（Bonus）

本次实验中，实现了 Lock Manager 模块，从而实现并发的查询，Lock Manager 负责追踪发放给事务的锁，并依据隔离级别适当地授予和释放共享和独占锁。

5.8.1 事务管理器

数据库系统中，事务管理器（Transaction Manager）是负责处理所有与事务相关操作的组件。它是维护数据库 ACID 属性（原子性、一致性、隔离性、持久性）的关键组件，确保了数据库系统中的事务能够安全、一致且高效地执行。事务管理器主要负责事务的边界控制、并发控制、恢复管理和故障处理。

在本次实验中，TxnManager 已经由框架实现，主要负责事务的边界控制、并发控制、故障处理。出于实现复杂度的考虑，同时为了避免各模块耦合太强，前面模块的问题导致后面模块完全无法完成，我们将 TxnManager 模块单独拆了出来。TxnManager 支持 Begin()、Commit()、Abort() 等方法。因为 TxnManager 模块独立，我们在 Commit()、Abort() 方法中不需要做其他事情（本来需要维护事务中的写、删除集合，结合 Recovery 模块回滚）。同时框架提供了 Txn 类, 里面通过参数控制事务的隔离级别：READ_UNCOMMITED、READ_COMMITTED 和 REPEATABLE_READ。Lock Manager 负责检查事务的隔离级别，任何失败的锁操作都将导致事务中止，并

同时抛出异常，此时 TxnManager 将捕获该异常并回滚。

5.8.2 锁管理器

本次项目实现了一个用于管理数据库事务锁的锁管理器 Lock Manager，以及一个死锁检测机制。锁管理器的主要功能包括处理事务对数据记录的共享锁（Shared Lock）和独占锁（Exclusive Lock）的请求，并确保在请求冲突时正确地阻塞和唤醒事务。死锁检测机制则在后台运行，定期检查事务等待图中的循环，并中止导致死锁的事务。

锁管理器负责处理事务对数据记录的锁请求，并维护当前活动事务所持有的锁。其主要功能包括：

- 共享锁请求（LockShared）：事务请求获取某数据记录的共享锁。如果该记录被其他事务持有独占锁，当前事务将被阻塞，直到独占锁被释放。
- 独占锁请求（LockExclusive）：事务请求获取某数据记录的独占锁。如果该记录被其他事务持有共享锁或独占锁，当前事务将被阻塞，直到所有冲突的锁被释放。
- 锁升级请求（LockUpgrade）：事务请求将已持有的共享锁升级为独占锁。如果该记录被其他事务持有共享锁或独占锁，当前事务将被阻塞，直到所有冲突的锁被释放。
- 释放锁（Unlock）：事务释放某数据记录上的锁，并根据事务的状态更新其两阶段锁 (2PL) 阶段。释放锁后，会唤醒被阻塞的其他事务。
- 锁准备（LockPrepare）：在事务请求锁之前，检查事务的状态是否符合预期，并在锁表中创建对应的数据记录和锁请求队列。
- 检查中止（CheckAbort）：检查事务的状态是否为中止，如果是，则执行相应的操作，如移除锁请求并抛出异常。

死锁检测机制通过后台线程定期运行，构建事务等待图并检测其中的循环。其主要功能包括：

- 添加边（AddEdge）：在事务等待图中添加一条边，表示一个事务在等待另一个事务释放锁。
- 移除边（RemoveEdge）：从事务等待图中移除一条边。

- 循环检测 (`HasCycle`)：使用深度优先搜索 (DFS) 算法检测事务等待图中的循环。如果找到循环，则返回最年轻的事务 ID 并中止该事务。
- 循环检测运行 (`RunCycleDetection`)：后台检测线程定时运行，后台线程在每次唤醒时即时构建图表，而不是维护一个图表，然后对该图表进行循环检测。等待图在每次线程唤醒时构建和销毁。

5.8.3 思考题

本模块中，为了简化实验难度，我们将 `Lock Manager` 模块独立出来。如果不独立出来，做到并发查询期间根据指定的隔离级别进行事务的边界控制，考虑模块 3 中 B+ 树并发修改的情况，需要怎么设计？

答：要支持 B+ 树结构中的并发修改，需要添加更细粒度的锁机制，比如说，在 B+ 树的内部节点和叶子节点上使用意向锁，分为意向共享锁 (IS)、意向独占锁 (IX) 和共享意向独占锁 (SIX)。要支持更高程度的并发，以及防止幻读，需要在索引记录之间加锁。这些间隙锁保护不存在的记录，以确保可重复读隔离级别的正确性。

6 功能测试

6.1 模块功能测试

6.1.1 Buffer Pool Manager Test

对 `Buffer Pool Manager` 运行测试，通过：

```
[----------] 1 test from BufferPoolManagerTest
[ RUN      ] BufferPoolManagerTest.BinaryDataTest
[       OK ] BufferPoolManagerTest.BinaryDataTest (1 ms)
[----------] 1 test from BufferPoolManagerTest (1 ms total)
```

6.1.2 Replacer Test

对 LRU 和 CLOCK 两种策略的 `Replacer` 运行测试，其中 CLOCK 的测试为自己编写的，通过：

```
[----------] 1 test from CLOCKReplacerTest
[ RUN      ] CLOCKReplacerTest.SampleTest
[       OK ] CLOCKReplacerTest.SampleTest (0 ms)
[----------] 1 test from CLOCKReplacerTest (0 ms total)

[----------] 1 test from LRUReplacerTest
[ RUN      ] LRUReplacerTest.SampleTest
[       OK ] LRUReplacerTest.SampleTest (0 ms)
[----------] 1 test from LRUReplacerTest (0 ms total)
```

6.1.3 Disk Manager Test

对 `Disk Manager` 运行测试，通过：

```
[----------] 2 tests from DiskManagerTest
[ RUN      ] DiskManagerTest.BitMapPageTest
[       OK ] DiskManagerTest.BitMapPageTest (3 ms)
[ RUN      ] DiskManagerTest.FreePageAllocationTest
[       OK ] DiskManagerTest.FreePageAllocationTest (135 ms)
[----------] 2 tests from DiskManagerTest (139 ms total)
```

6.1.4 Tuple Test

对 Field、Row、Column 和 Schema 运行测试，其中 Column 和 Schema 的测试为自己编写的，通过：

```
[----------] 4 tests from TupleTest
[ RUN      ] TupleTest.FieldTest
[       OK ] TupleTest.FieldTest (0 ms)
[ RUN      ] TupleTest.RowTest
[       OK ] TupleTest.RowTest (0 ms)
[ RUN      ] TupleTest.ColumnTest
[       OK ] TupleTest.ColumnTest (0 ms)
[ RUN      ] TupleTest.SchemaTest
[       OK ] TupleTest.SchemaTest (0 ms)
[----------] 4 tests from TupleTest (0 ms total)
```

6.1.5 Table Heap Test

对 Table Heap 运行测试，其中自己添加了对元组的删除、更新的测试，以及对迭代器的测试，通过：

```
[----------] 2 tests from TableHeapTest
[ RUN      ] TableHeapTest.TableHeapSampleTest
[       OK ] TableHeapTest.TableHeapSampleTest (656 ms)
[ RUN      ] TableHeapTest.TableIteratorTest
[       OK ] TableHeapTest.TableIteratorTest (418 ms)
[----------] 2 tests from TableHeapTest (1075 ms total)
```

6.1.6 B Plus Tree Test

对 B Plus Tree 运行测试，通过：

```
[----------] 1 test from BPlusTreeTests
[ RUN      ] BPlusTreeTests.SampleTest
[       OK ] BPlusTreeTests.SampleTest (94 ms)
[----------] 1 test from BPlusTreeTests (94 ms total)
```

6.1.7 Index Iterator Test

对 Index Iterator 运行测试，通过：

```
[----------] 1 test from BPlusTreeTests
[ RUN      ] BPlusTreeTests.IndexIteratorTest
[       OK ] BPlusTreeTests.IndexIteratorTest (19 ms)
[----------] 1 test from BPlusTreeTests (19 ms total)
```

6.1.8 Catalog Test

对 Catalog 运行测试，通过：

```
[----------] 3 tests from CatalogTest
[ RUN      ] CatalogTest.CatalogMetaTest
[       OK ] CatalogTest.CatalogMetaTest (0 ms)
[ RUN      ] CatalogTest.CatalogTableTest
[       OK ] CatalogTest.CatalogTableTest (29 ms)
[ RUN      ] CatalogTest.CatalogIndexTest
[       OK ] CatalogTest.CatalogIndexTest (24 ms)
[----------] 3 tests from CatalogTest (54 ms total)
```

6.1.9 Executor Test

对 Executor 运行测试，通过：

```
[----------] 4 tests from ExecutorTest
[ RUN      ] ExecutorTest.SimpleSeqScanTest
[       OK ] ExecutorTest.SimpleSeqScanTest (37 ms)
[ RUN      ] ExecutorTest.SimpleDeleteTest
[       OK ] ExecutorTest.SimpleDeleteTest (41 ms)
[ RUN      ] ExecutorTest.SimpleRawInsertTest
[       OK ] ExecutorTest.SimpleRawInsertTest (25 ms)
[ RUN      ] ExecutorTest.SimpleUpdateTest
[       OK ] ExecutorTest.SimpleUpdateTest (27 ms)
[----------] 4 tests from ExecutorTest (132 ms total)
```

6.1.10 Recovery Manager Test

对 Recovery Manager 运行测试，通过：

```
[----------] 1 test from RecoveryManagerTest
[ RUN      ] RecoveryManagerTest.RecoveryTest
[       OK ] RecoveryManagerTest.RecoveryTest (0 ms)
[----------] 1 test from RecoveryManagerTest (0 ms total)
```

6.1.11 Lock Manager Test

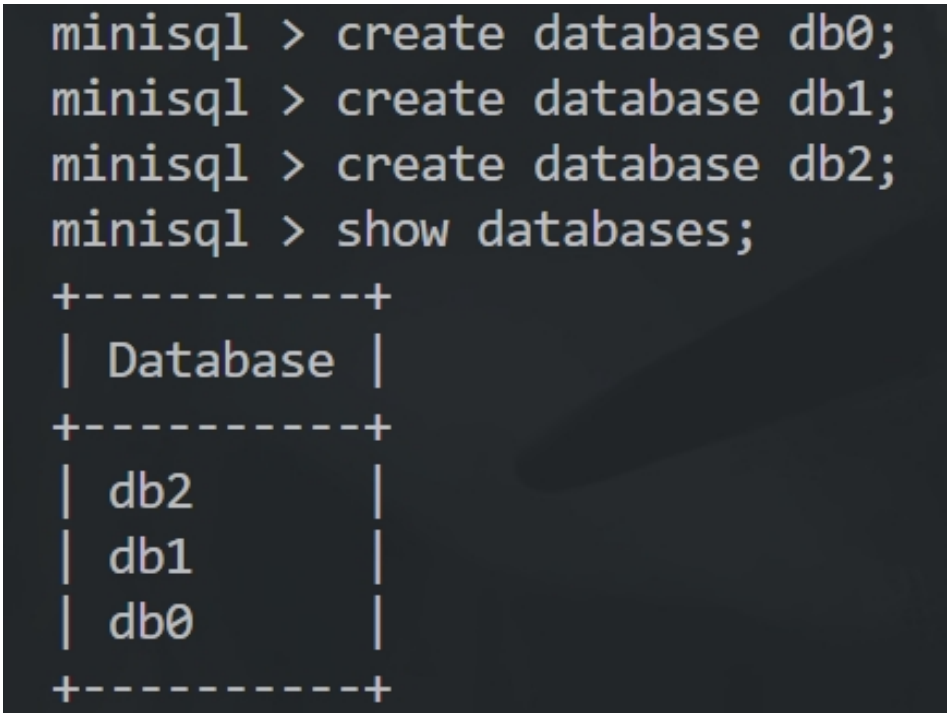
对 Lock Manager 运行测试，通过：

```
[----------] 10 tests from LockManagerTest
[ RUN      ] LockManagerTest.SLockInReadUncommittedTest
[       OK ] LockManagerTest.SLockInReadUncommittedTest (0 ms)
[ RUN      ] LockManagerTest.TwoPhaseLockingTest
[       OK ] LockManagerTest.TwoPhaseLockingTest (0 ms)
[ RUN      ] LockManagerTest.UpgradeLockInShrinkingPhase
[       OK ] LockManagerTest.UpgradeLockInShrinkingPhase (0 ms)
[ RUN      ] LockManagerTest.UpgradeConflictTest
[       OK ] LockManagerTest.UpgradeConflictTest (100 ms)
[ RUN      ] LockManagerTest.UpgradeTest
[       OK ] LockManagerTest.UpgradeTest (0 ms)
[ RUN      ] LockManagerTest.UpgradeAfterAbortTest
[       OK ] LockManagerTest.UpgradeAfterAbortTest (100 ms)
[ RUN      ] LockManagerTest.BasicCycleTest1
[       OK ] LockManagerTest.BasicCycleTest1 (0 ms)
[ RUN      ] LockManagerTest.BasicCycleTest2
[       OK ] LockManagerTest.BasicCycleTest2 (0 ms)
[ RUN      ] LockManagerTest.DeadlockDetectionTest1
[       OK ] LockManagerTest.DeadlockDetectionTest1 (1600 ms)
[ RUN      ] LockManagerTest.DeadlockDetectionTest2
[       OK ] LockManagerTest.DeadlockDetectionTest2 (1601 ms)
[----------] 10 tests from LockManagerTest (3403 ms total)
```

6.2 整体系统演示

由于截图限制，此处仅展示部分功能测试的截图。

1. 创建数据库 db0、db1、db2，并列出所有的数据库：



2. 在 db0 数据库上创建数据表 account，表的定义如下：

```
create table account(
    id int,
    name char(16) unique,
    balance float,
    primary key(id)
);
```

```
minisql > use db0;
Database changed
minisql > create table account(
  id int,
  name char(16) unique,
  balance float,
  primary key(id)
);
Table account is created successfully
```

3. 执行 `execfile "./data/account0x.txt"`，分十次插入十万条数据，并执行全表扫描 `select * from account`，验证插入数据：

```
| 12599990 | name99990 | 1.430000   |
| 12599991 | name99991 | 332.899994 |
| 12599992 | name99992 | 598.010010 |
| 12599993 | name99993 | 587.750000 |
| 12599994 | name99994 | 314.859985 |
| 12599995 | name99995 | 970.500000 |
| 12599996 | name99996 | 243.809998 |
| 12599997 | name99997 | 477.829987 |
| 12599998 | name99998 | 449.160004 |
| 12599999 | name99999 | 303.079987 |
+----------+-----------+------------+
100000 row in set(0.3970 sec).
```

4. 考察点查询，执行 `select * from account where id = ?`:

```
| 12599992 | name99992 | 598.010010 |
| 12599993 | name99993 | 587.750000 |
| 12599994 | name99994 | 314.859985 |
| 12599995 | name99995 | 970.500000 |
| 12599996 | name99996 | 243.809998 |
| 12599997 | name99997 | 477.829987 |
| 12599998 | name99998 | 449.160004 |
| 12599999 | name99999 | 303.079987 |
+----------+-----------+------------+
100000 row in set(0.3970 sec).
minisql > select * from account where id = 12599995;
+----------+-----------+------------+
| id       | name      | balance    |
+----------+-----------+------------+
| 12599995 | name99995 | 970.500000 |
+----------+-----------+------------+
1 row in set(0.0070 sec).
```

5. 考察多条件查询与投影操作，执行 `select id, name from account where balance >= ? and balance < ?`：

```
minisql > select id, name from account where balance >= 199.8 and balance < 200;
+----------+-----------+
| id       | name      |
+----------+-----------+
| 12500708 | name00708 |
| 12501835 | name01835 |
| 12515537 | name15537 |
| 12522509 | name22509 |
| 12530584 | name30584 |
| 12534270 | name34270 |
| 12535780 | name35780 |
| 12537045 | name37045 |
| 12544234 | name44234 |
| 12551155 | name51155 |
| 12553459 | name53459 |
| 12553681 | name53681 |
| 12554878 | name54878 |
| 12568324 | name68324 |
| 12570798 | name70798 |
| 12571730 | name71730 |
| 12577418 | name77418 |
| 12593305 | name93305 |
+----------+-----------+
18 row in set(0.1910 sec).
```

6. 考察唯一约束，执行 `insert into account values(?, ?, ?)`，提示 UNIQUE 约束冲突：

```
minisql > insert into account values(123456, "name00000", 114.514);
Duplicate key found in index account_name_UNIQUE
Query OK, 0 row affected(0.0010 sec).
```

7. 考察索引的创建、删除和效果，此处先删除了 unique 索引，以便展现索引的效果。先在建立索引前进行查询，然后在执行 `create index idx01 on account(name)` 后，再次进行查询，可以看到索引查询比顺序查询速度大幅提升：

```
minisql > drop index account_name_UNIQUE;
Index account_name_UNIQUE is dropped successfully
minisql > select * from account where name = "name45678";
+----------+-----------+-----------+
| id       | name      | balance   |
+----------+-----------+-----------+
| 12545678 | name45678 | 61.500000 |
+----------+-----------+-----------+
1 row in set(0.2050 sec).
minisql > create index idx01 on account(name);
Index idx01 is created successfully
minisql > select * from account where name = "name45678";
+----------+-----------+-----------+
| id       | name      | balance   |
+----------+-----------+-----------+
| 12545678 | name45678 | 61.500000 |
+----------+-----------+-----------+
1 row in set(0.0000 sec).
```

8. 考察更新操作，执行 `update account set id = 114, balance = 514 where name = "name56789"`，并通过 `select` 操作进行验证：

```
minisql > update account set id = 114, balance = 514 where name = "name56789";
Query OK, 1 row affected(0.1880 sec).
minisql > select * from account where name = "name56789";
+-----+-----------+------------+
| id  | name      | balance    |
+-----+-----------+------------+
| 114 | name56789 | 514.000000 |
+-----+-----------+------------+
1 row in set(0.0000 sec).
```

9. 考察删除操作，分别对记录和表进行删除，通过 `select` 操作进行验证：

```
minisql > delete from account where balance = 200;
Query OK, 1 row affected(0.1820 sec).
minisql > select * from account where balance = 200;
Empty set(0.1780 sec).
minisql > delete from account;
Query OK, 99995 row affected(4.1660 sec).
minisql > select * from account;
Empty set(0.0050 sec).
minisql > drop table account;
Table account is dropped successfully
minisql > show tables;
Empty set (0.00 sec)
```

7 建设性意见

在 MiniSQL 实验过程中，我指出了两个框架存在的问题，一是 `insert_executor` 中，插入没有考虑索引的约束（因为我们实现的 B+ 树索引只支持 `Unique Key`），二是 `insert_executor` 的 `Next` 函数，没有为 `insert_rid` 赋值，导致插入 B+ 树时只有 `value` 没有 `key`。

另外，`catalog` 部分没有注释，如果添加适当的注释，会让框架变得更可读。