Minisql Design Report

3210105944 黄锦骏

一、框架设计

实验目的：

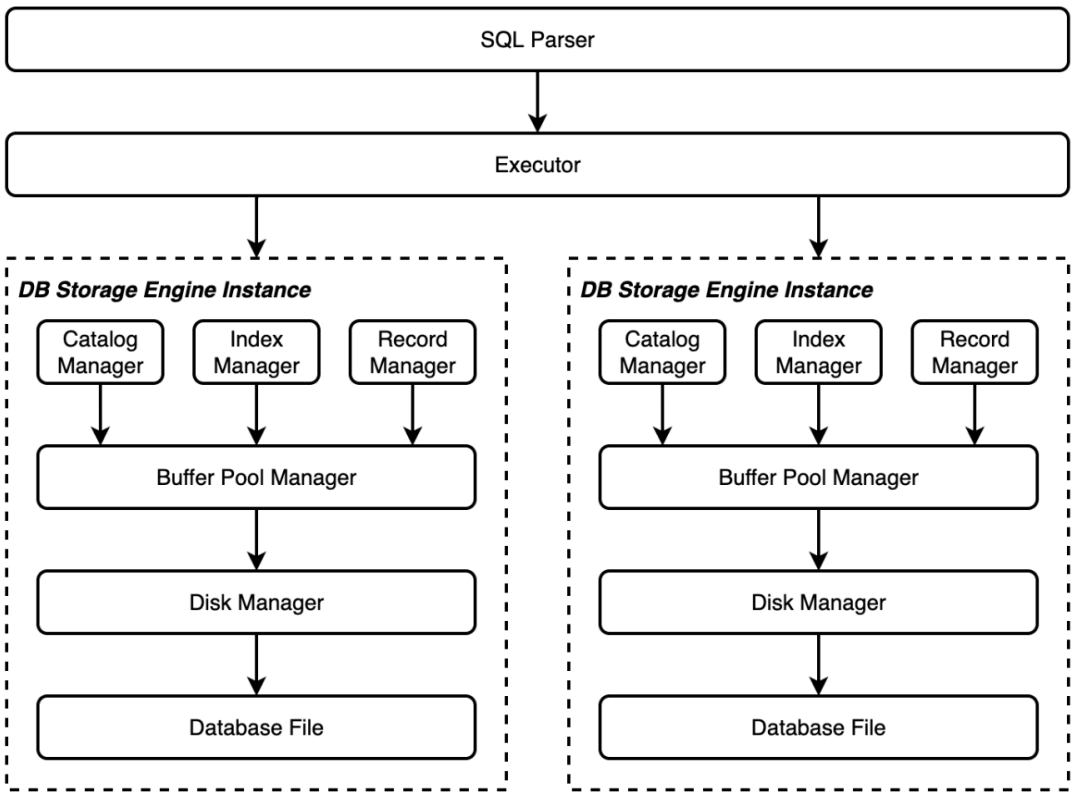
1. 设计并实现一个精简型单用户SQL引擎MiniSQL，允许用户通过字符界面输入SQL语句实现基本的增删改查操作，并能够通过索引来优化性能。
2. 通过对MiniSQL的设计与实现，提高学生的系统编程能力，加深对数据库管理系统底层设计的理解。

实验需求：

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。

系统架构：

- 系统架构如图所示。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）



- 系统模块概述：
  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，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。
     - 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. Buffer Pool Manager
     - Buffer Manager 负责缓冲区的管理，主要功能包括：
       - 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；

         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；

         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；

         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。
  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。
  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。
  5. Catalog Manager
     - Catalog Manager 负责管理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。
     - 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。

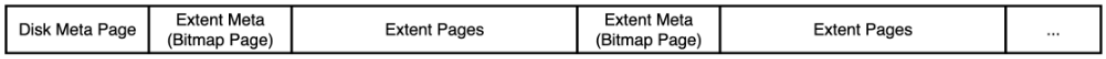
6. Planner and Executor
    - Planner（执行计划生成器）的主要功能是根据解释器（Parser）生成的语法树，通过 Catalog Manager 提供的信息检查语法树中的信息是否正确，如表、列是否存在，谓词的值类型是否与column类型对应等等，随后将这些词语转换成可以理解的各种 c++ 类。解析完成后，Planner根据改写语法树后生成的Statement结构，生成对应的Plannode，并将Plannode交由Executor进行执行。
    - Executor（执行器）的主要功能是遍历Planner生成的计划树，将树上的 PlanNode 替换成对应的 Executor，并调用 Record Manager、Index Manager 和 Catalog Manager 提供的相应接口进行执行。Executor采用的是火山模型，提供迭代器接口，每次调用时会返回一个元组和相应的 RID，直到执行完成。
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。

二、模块设计及实现

1. Disk Manager

1.1 模块介绍

- 功能介绍：Disk Manager负责数据库文件中数据页page的分配和回收，以及数据页中数据的读取和写出。
- 结构介绍：由Meta page, Bitmap page, Extent pages三部分组成，其在内存中的组织可以参考下图。其中一个Bitmap page和Extent pages组成一个extent



    - Meta page: 作为Disk Manager的元数据，它记录了Disk中全部已经分配的page number，extent number和每个extent中已经使用过的page数量
    - Bitmap Page: 在一个Extent中，标记该Extent中一段连续数据页的分配情况。其中记录了当前extent中已经分配的page number，以及该extent中的next_free_page_，并用一个char数组来标记该page是否已经分配过，每一个byte用于标记八个page.
    - Extent Page：具体存储数据的page，具有page_id_属性，为buffer_pool_manager所访问的逻辑地址，pin_count属性用于记录有多少线程正在访问该page，is_dirty用于标记该page是否有数据，一个page latch，和一个data数组用于存储真实数据

1.2 具体实现细节

- Bitmap Page:
    - Bitmap Page构造函数，用于将大小为PAGE_SIZE的char数组内数据赋值给Bitmap Page中的各个属性。添加了判断Bitmap Page为空和满的两个函数IsEmpty和IsFull
    - AllocatePage：首先需要判断Bitmap_page是否已经分配完全，然后根据传入的offset计算出是哪个byte的哪一位。计算出后进行将该位标记为1。next_free_page从当前page_offset开始遍历直到发现为空
    - DeAlloacatePage: 首先判断传入的page_offset所对应的page是否free, 再进行标记更新
- DiskManager:

- AllocatePage：实际上调用了Bitmap_page的allocatepage方法来分配page. 首先从物理内存中读取bitmap_page，如果Bitmap_page未满，就使用该bitmap_page进行分配。否则读取一个新的Bitmap_page，每次分配后要更新meta_page的元数据，最后要将bitmap_page重新写入到内存中去
- DeAllocatePage: 同理，根据输入的logical_page_id计算出一个extent中的物理页page_index和extent的index(Bitmap_page_index)，再实际调用bitmap_page.DeAllocatePage释放page并更新元数据.特别注意当释放page后如果bitmap_page为空，则需要将meta_page中的extent数目减1
- 数据页中数据的读入和写出由WritePhysicalPage和ReadPhysicalPage实现
- MapPageId: 实现的是逻辑id到物理id的映射

```
physical_id = logical_page_id + 2 + logical_page_id / BITMAP_SIZE
```

2. BufferPool Manager

2.1 模块介绍

- 功能介绍：BufferPool模块主要负责将磁盘中的数据页从内存中来回移动到磁盘，实现了类似于缓存的操作
- 结构介绍：
  - 属性元素：

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

  - 功能函数：
    - `BufferPoolManager::FetchPage(page_id)`：根据逻辑页号获取对应的数据页，如果该数据页不在内存中，则需要从磁盘中进行读取；
    - `BufferPoolManager::NewPage(&page_id)`：分配一个新的数据页，并将逻辑页号于 `page_id` 中返回；
    - `BufferPoolManager::UnpinPage(page_id, is_dirty)`：取消固定一个数据页；
    - `BufferPoolManager::FlushPage(page_id)`：将数据页转储到磁盘中；
    - `BufferPoolManager::DeletePage(page_id)`：释放一个数据页；
    - `BufferPoolManager::FlushAllPages()`：将所有的页面都转储到磁盘中。

2.2 具体实现细节

- LRU替换算法：即最近最少使用算法，是一种内存数据淘汰策略，使用常见是当内存不足时，需要淘汰最近最少使用的数据。具体实现采用双向链表和Hashmap的方式实现

```
private:
  // add your own private member variables here
  std::list<frame_id_t> LRU_list; // a double linked list to store the page in the LRU list
  std::unordered_set<frame_id_t> LRU_map; //a hashmap to represent whether a frame_id_t is in the LRU_LIST
  size_t max_pages;
```

LRU_list用于存储当前在LRU_list中的数据页，Hashmap LRU_map用于标记某个数据页当前是否在LRU_list中(便于快速查找)，max_pages是LRU_list中最大的page数目上限

  - 具体实现思想：每当有空闲的数据页时(即没有占用数据页的进程)那么就将该数据页加入到LRU_list的头部，并在map中标记该数据页已经在LRU_list中。当数据页被占用后，就应从

LRU_List中删除。当需要替换时就返回LRU_list中头部的元素，因为每次插入到头部可以保证头部元素是最近访问过的

- BufferPool Manager: 依次介绍每个模块的算法设计思路
  - FetchPage：
    1. Search the page table for the requested page (P).
       1.1 If P exists, pin it and return it immediately.
       1.2 If P does not exist, find a replacement page (R) from either the free list or the replacer.Note that pages are always found from the free list first.
    2. If R is dirty, write it back to the disk.
    3. Delete R from the page table and insert P.
    4. Update P's metadata, read in the page content from disk, and then return a pointer to P.
  - NewPage：
    1. If all the pages in the buffer pool are pinned, return nullptr.
    2. Pick a victim page P from either the free list or the replacer. Always pick from the free list first.
    3. Update P's metadata, zero out memory and add P to the page table.
    4. Set the page ID output parameter. Return a pointer to P.
  - UnpinPage
    1. Search the page_table_ to check whether the page is in the buffer pool.
    2. Check whether the page is pinned or not. If it is pinned, it can't be unpinned again.
    3. If the page is pinned, unpin it and update the metadata.
    4. Check whether its pin number is 0 after unpinning.
  - FlushPage
    1. Check whether the page is in the buffer pool.
    2. Write the page into the disk
  - DeletePag
    1. Search the page table for the requested page (P).
       1.1 If P does not exist, return true.
       1.2. If P exists, but has a non-zero pin-count, return false. Someone is using the page.
    2. Otherwise, P can be deleted. Remove P from the page table, reset its metadata and return it to the free list.

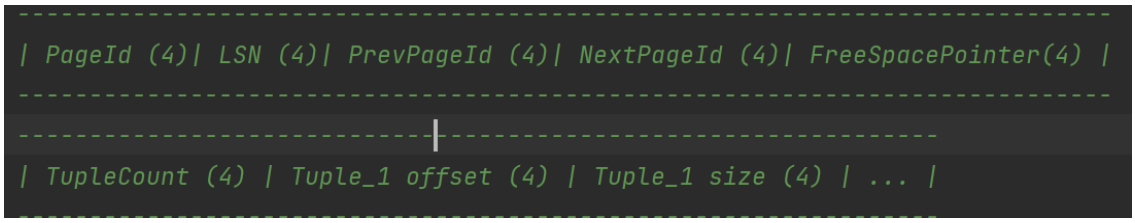
3. Record Manager:

3.1 模块介绍

- 总功能介绍：Record Manager模块主要用于负责管理数据库的记录，支持记录的插入，删除和查找操作。对外提供相应的接口
- 结构介绍：Record Manager的核心是通过堆表TableHeap来管理记录，TableHeap由table_page的双向链表构成，table_page是物理上实质存储记录(Row)的地方，所以TableHeap中是通过Row的唯一标识RowId来找到Row所属的table_page，再通过table_page中的具体实现来进行Row的插入，更新和删除. TableHeap通过实现的TableIterator来进行数据的访问

3.1.1 Table_page:

- 模块构成：table_page的具体实现分为两个部分，table_Page外部作为双向链表的连接部分，内部tuple的插入，更新，删除部分，在删除部分分为打上逻辑上的DeletedFlag标记标识删除和物理上的实质删除，
- 物理组织：table_page作为一个数据页，大小仍然为PAGE_SIZE，物理上由table_page_header，free_space和insert_tuples所构成。table_page_header结构如下



访问数据通过FreeSpacePointer以及偏移的计算来访问，每一个tuple有自己的slot_number用于访问自身的offset和size，再通过自身的offset访问该Page中的具体存储地址.

table_page的所有操作由框架给出

3.1.2 table_heap

- 模块构成：table_heap由table_page的双向链表构成，包括创建堆表，对数据的插入，删除，更新，查询。成员变量包括一个用于page管理的buffer_pool_manager，堆表中第一个first_page_id以及整张表的结构schema_
- 功能函数：
  - `TableHeap:InsertTuple(&row, *txn)`：向堆表中插入一条记录，插入记录后生成的 `RowId` 需要通过 `row` 对象返回（即 `row.rid_`）；
  - `TableHeap:UpdateTuple(&new_row, &rid, *txn)`：将 `RowId` 为 `rid` 的记录 `old_row` 替换成新的记录 `new_row`，并将 `new_row` 的 `RowId` 通过 `new_row.rid_` 返回；
  - `TableHeap:ApplyDelete(&rid, *txn)`：从物理意义上删除这条记录；
  - `TableHeap:GetTuple(*row, *txn)`：获取 `RowId` 为 `row->rid_` 的记录；
  - `TableHeap:FreeHeap()`：销毁整个 `TableHeap` 并释放这些数据页；
  - `TableHeap::Begin()`：获取堆表的首迭代器；
  - `TableHeap::End()`：获取堆表的尾迭代器；
  - `TableIterator` 类中的成员操作符
    - `TableIterator::operator++()`：移动到下一条记录，通过 `++iter` 调用；
    - `TableIterator::operator++(int)`：移动到下一条记录，通过 `iter++` 调用；

3.1.3 table_iterator

- 模块构成：table_iterator实现了对于table_heap的访问，实现了对于迭代器来说常见的++，->等基本运算符，成员变量包括指针row_用于指示当前行，table_heap_用于访问当前row_所在的table_heap，事务指针txn(在获取row内容时使用)
- 功能函数：
  - `TableIterator::operator==(TableIterator &itr)`
  - `TableIterator::operator!=(const TableIterator &itr)`
  - `Row &TableIterator::operator*()`
  - `Row *TableIterator::operator->()`
  - `TableIterator &TableIterator::operator=(const TableIterator &itr)`
  - `TableIterator TableIterator::operator++(int)`
  - `TableIterator &TableIterator::operator++()`

3.1.4 record instances

- 模块描述：此部分主要是数据库中的具体记录，包括column, Schema, Field, Row，Schema描述了一个数据表或者索引的结构，Column用于描述数据表中某一列的定义属性，Row用于描述数据表中某一行的数据，Field对应一条记录(一个row)里某一个字段的数据信息
- 模块实现：主体部分由框架给出，个人主要实现了四种对象的序列化和反序列化操作.

3.2 实现细节

3.2.1 Serialize and DisSerialize

- 对于Field对象：由于他只是某一个row里一个字段，所以在序列化和反序列化时我们仅用根据field中存储的数据类型(char, int float)将Union中的相应数据存入到内存或者从内存中读取到对象.
- 对于Row对象：由于一个Row中有一个field数组，而实际上field的具体数据可能为空(对应数据表中的NULL)，因此我们需要一个位图来标识field中的数据是否为空。因此我们首先将field的数目存入到内存中(用于获取Bitmap)，再创建一个Bitmap并存入到内存中，最后才将field中的具体数据存入内存。反序列化时，首先读出field_count，再读出bitmap，最后调用field中实现的反序列化操作
- 对于Column和Schema对象：Column仅用将所存储的各种属性存入内存，Schema也仅用将存储的Column数组和is_manage标识，magic_num存入内存即可.

3.2.2 table_iterator:

- 成员属性定义：

```
TableHeap *table_heap; //The table heap pointer
Row *row; //Traverse the each row in the table
Transaction *txn; //Used for GetTuple operation
```

- 后置++操作：首先创建当前Iterator的副本，根据iterator中的row获取所在页面，在通过梭子啊页面获取下一条记录。获取下一条记录可能有三种情况，第一种是直接获取到下一条记录，第二种是当前记录是该页面的最后一条记录，需要获取下一个页面的第一条记录，第三种是该记录是tableheap里的最后一条记录，需要将Iterator设置为nullptr

3.2.3 table_heap

- 成员属性定义：

```
BufferPoolManager *buffer_pool_manager_;  //内存池管理器
page_id_t first_page_id_; //数据表中第一个tablepage的Id
Schema *schema_; //整张数据表的结构
LogManager *log_manager_;
LockManager *lock_manager_;
```

- InsertTable():

  首先判断堆表中是否存在page，如果不存在则通过内存池分配一个新的page，然后在这个page里进行插入，插入后对这个page进行Unpin释放

- UpdateTable():

  首先根据输入的rid在堆表中进行页面获取，检查堆表中是否存在该page，然后将旧的row内容读出到old_row内容，接着调用page的UpdateTuple操作进行更新。更新分为三种情况，一种是直接更新成功，一种是返回错误并返回内存不够的message信息，此时我们开辟一个新的页面，将更新的内容插入到新的页面中，并对原来的row进行标记删除，等待合适的时机进行删除

- Applydelete():

首先根据rid获取该row所在的页面，调用page的ApplyDelete进行删除

- GetTuple()：

  首先根据rid获取该row所在的页面，调用page的Gettuple()进行row内容的获取。而在page的底层实现中，

- Begin()：

  从数据页中获取到first_row，将其和this指针作为TableIterator返回

4. Index Manager

4.1 模块介绍

- 功能介绍：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。具体实现的只有B+tree的部分，B+ tree index的操作部分由框架给出
- B+ tree index：B+tree的上层接口，利用B+ tree作为底层数据结构的索引，具体操作包括InsertEntry, RemoveEntry以及Scankey还有迭代器的获取，均调用了底层B+tree的操作

B+ tree page

- B+ tree page分为两个部分，分别为leaf page和internal page，leaf_page存储的是RowId + Key，InternalPage存储的是PageId + key。索引的名字和root_page页号，由Index_root_page统一管理

```
* Format (size in byte):
*  --------------------------------------------------------------------
* | RecordCount (4) | Index_1 id (4) | Index_1 root_id (4) | ... |
*  --------------------------------------------------------------------
*/
```

4.2 具体实现细节

- 本模块中，具体实现了B+树的插入，删除，叶子节点的遍历等操作，具体函数包括分裂，合并，重新分配等等，其中涉及到的函数较多，此处不一一讲解。仅仅讲解插入和删除以及遍历的具体操作过程：
  - 具体函数包括：Init, KeyIndex, Insert, InsertIntoleaf, InsertIntoParent, StartNewTree, Remove, MoveHalfTo, CopyNFrom, LookUp, RemoveandDeleteRecord，MoveAllTo, MoveFirstToEndOf, CopyLastFrom, FindLeafPage, GetValue, UpdateRootPageId, AdjustRoot,Split, Coalesce, Redistribute, CoalesceOrRedistribute等
- Insert: 首先要判断B+树是否为空，若为空，则需要创建一棵新的B+树，并且要把根节点的信息更新到INDEX_ROOT_PAGE中。若不为空，则需要判断索引的unique，即从B+树中GetValue，随后直接调用insert_into_leaf，首先要寻找到叶子节点，插入到叶子节点。然后判断叶子节点是否需要分裂，如果数量满了就调用insert_into_parent，在这个函数中进行递归split的过程。涉及到根节点要特别处理，需要更新Index_Root_Page中的根节点信息
- Remove：首先要判断B+树是否为空。若B+树不为空，则直接寻找到叶子节点进行删除。然后对这个叶子节点进行判断，调用CoalesceOrRedistribute函数判断它是否需要合并以及重分配并执行相应操作，再对它的parent进行递归判断。涉及到根节点要特别处理，需要更新Index_Root_Page中的根节点信息
- 遍历：Index_iterator是本模块提供的接口，我使用了如下的几个属性来设计Iterator

```
private:
 page_id_t current_page_id{INVALID_PAGE_ID};
 LeafPage *page{nullptr};
 int item_index{0};
 BufferPoolManager *buffer_pool_manager{nullptr};
```

其中current_page_id是当前iterator所在的page_id，page是其中的数据，item_index是该数据在page中的数据下标。当进行++时，就对item_index和page进行更新。用page + item_index来寻找具体的数据

5. Catalog Manager

5.1 模块介绍

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。

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。

5.2 具体实现细节

Catalog Manager利用元信息来对table和index进行管理，元信息包括**CatalogMeta, IndexMetadata和TableMetadata**。CatalogMeta记录了table_id和对应table_heap所在page_id，以及index_id和其B+ tree所在page_id。TableMeta记录了table的id, name, root_page_id以及schema信息，IndexMeta记录了index的id,name, index所在table_id， 以及index中key和原始schema中的key的下标对应。

具体实现的函数包括三者的序列化和反序列化，以及三者的初始化，核心在于CatalogManager中的CreateTable, CreateIndex, LoadTable, LoadIndex, DropTable, DropIndex, FlushCatalogMetaPage函数。`CatalogManager`能够在数据库实例（`DBStorageEngine`）初次创建时（`init = true`）初始化元数据；并在后续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，构建`TableInfo`和`IndexInfo`信息置于内存中。此外，`CatalogManager`类还需要对上层模块提供对指定数据表的操作方式，如`CreateTable`、`GetTable`、`GetTables`、`DropTable`、`GetTableIndexes`；对上层模块提供对指定索引的操作方式，如`CreateIndex`、`GetIndex`、`DropIndex`。

- CreateTable: 首先检查table是否存在，不存在则创建新的table_heap, table_info,table_metadata并加入到catalog_meta中去（TableInfo就是利用table_metadata和table_heap创建的具体表，包含了该表的元信息和具体数据）
- CreateIndex: 首先检查table是否存在，再检查相应table上的index是否存在，不存在则创建新的index_info和index_metadata, table_metadata并加入到catalog_meta中去
- LoadTable: 首先从catalog_metadata拿到table_id和对应table数据所在的page_id，从table数据所在的page拿到有关table的所有信息进行重建table即可。此时数据的重建仅仅是table_heap中指向第一条数据的page的指针的复制
- LoadIndex: 首先从catalog_metadata中拿到index_id和对应index信息所在的page_id，此时和LoadTable不同的是，IndexMetadata仅仅包含了index_id，index_name，table_name和key_map，并不包含具体索引的信息。具体索引的信息由Index_Root_Page来维护，所以我们需要从IndexMetadata中获取index_id，再利用index_id从Index_Root_page中获取具体索引所在的

page。再遍历具体索引数据所在的leaf_page进行重建索引。

```
while(tree_page->IsLeafPage() != true){
  buffer_pool_manager_->UnpinPage(tree_page->GetPageId(), true);
  tree_page = reinterpret_cast<InternalPage *>(buffer_pool_manager_->FetchPage(tree_page->ValueAt(0))->GetData());
}
LeafPage *leaf_page = reinterpret_cast<LeafPage *>(tree_page);
while(1){
    for(int i = 0; i < leaf_page->GetSize(); i++){
      b_tree_index->GetKeyManager().DeserializeToKey(leaf_page->KeyAt(i), row, index_info->GetIndexKeySchema());
      rid = leaf_page->ValueAt(i);
      b_tree_index->InsertEntry(row, rid, nullptr);
    }
    buffer_pool_manager_->UnpinPage(leaf_page->GetPageId(), true);
    if(leaf_page->GetNextPageId() == INVALID_PAGE_ID)
      break;
    else
      leaf_page = reinterpret_cast<LeafPage *>(buffer_pool_manager_->FetchPage(leaf_page->GetNextPageId())->GetData());
}
```

- DropTable: 首先检查table是否存在，若存在则首先Drop该表上的所有索引，此后再将table_info删除，删除table_info的同时释放了相应的table_heap和table_metadata的内存
- DropIndex: 首先检查索引是否存在，然后遍历所有的索引，调用索引自身的RemoveEntry进行删除，最后释放Index_info内存，同时释放了index_meatadata和B+ tree的内存
- FlushCatalogMetaPage: 具体包括catalog的元信息，table的元信息，index的元信息三者都序列化到catalog_page中去

```
dberr_t CatalogManager::FlushCatalogMetaPage() const {
  Page * catalog_page = buffer_pool_manager_->FetchPage(CATALOG_META_PAGE_ID);
  if(catalog_page == nullptr){
//    LOG(ERROR) << "Failed to fetch the catalog meta page in the FlushCatalogMetaPage" << endl;
    return DB_FAILED;
  }
  char * data = catalog_page->GetData();
  //Serialize the catalog meta data
  catalog_meta_->SerializeTo(data);
  data += catalog_meta_->GetSerializedSize();
  buffer_pool_manager_->UnpinPage(CATALOG_META_PAGE_ID, true);
//  buffer_pool_manager_->FlushPage(CATALOG_META_PAGE_ID);

  //Serialize the table meta data
  for(auto table_iter : catalog_meta_->table_meta_pages_){
    Page * table_page = buffer_pool_manager_->FetchPage(table_iter.second);
    data = table_page->GetData();
    tables_.at(table_iter.first)->GetTableMetadata()->SerializeTo(data);
//    data += tables_.at(table_iter.first)->GetTableMetadata()->GetSerializedSize();
    buffer_pool_manager_->UnpinPage(table_iter.second, true);
  }
  //Serialize the index meta data
  for(auto index_iter : catalog_meta_->index_meta_pages_){
    Page * index_page = buffer_pool_manager_->FetchPage(index_iter.second);
    data = index_page->GetData();
    indexes_.at(index_iter.first)->GetIndexMetadata()->SerializeTo(data);
//    data += indexes_.at(index_iter.first)->GetIndexMetadata()->GetSerializedSize();
    buffer_pool_manager_->UnpinPage(index_iter.second, true);
  }
  return DB_SUCCESS;
}
```

6. Planner Manager

6.1 模块介绍

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。

6.2 具体实现细节

此部分由框架提供，本身未作任何实现。框架实现部分具体包括语法解释器，利用语法解释器生成语法树以及计划生成。具体 流程是，在Parser模块调用`yyparse()`完成SQL语句解析后，将会得到语法树的根结点`pSyntaxNode`。将语法树根结点传入`ExecuteEngine`（定义于`src/include/executor/execute_engine.h`）后，`ExecuteEngine`将会根据语法树根结点的类型，决定是否需要传入`Planner`生成执行计划。

`Statement`中的函数`SyntaxTree2Statement`将解析语法树，并将各种Identifier转化为可以理解的表达式，存储在Statement结构中。Planner再根据Statement，生成对应的执行计划

语法树数据结构：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。`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（如在`kNodeString`类型的结点中，`val_`将用于存储该字符串的字面量）。

```
/**
 * Syntax node definition used in abstract syntax tree.
 */
struct SyntaxNode {
  int id_;    /** node id for allocated syntax node, used for debug */
  SyntaxNodeType type_; /** syntax node type */
  int line_no_; /** line number of this syntax node appears in sql */
  int col_no_;  /** column number of this syntax node appears in sql */
  struct SyntaxNode *child_;  /** children of this syntax node */
  struct SyntaxNode *next_;   /** siblings of this syntax node, linked by a single linked list */
  char *val_; /** attribute value of this syntax node, use deep copy */
};
typedef struct SyntaxNode *pSyntaxNode;
```

7. Executor Manager

7.1 模块介绍

该模块实现了基于火山模型的Executor，该executor会遍历查询计划树，将树上的 PlanNode 替换成对应的 Executor，随后调用 Record Manager、Index Manager 和 Catalog Manager 提供的相应接口进行执行，并将执行结果返回给上层模块

7.2 具体实现细节

该模块实现了如下sql语句的对应executor

```
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));
drop table t1;
create index idx1 on t1(a, b) using btree;
drop index idx1;
show indexes;
select id, name from t1;
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 where id = 1 and amount = 2.33;
update t1 set a = 1, b = "ccc" where b = 2.33;
quit;
execfile "a.txt";
```

其中，除了表中数据查询，数据更新，数据删除，数据插入操作较为复杂，需要plan生成具体查询计划外，其它操作不用通过Planner生成查询计划，它们被声明为`private`类型的成员，所有的执行过程对上层模块隐藏，上层模块只需要调用`ExecuteEngine::execute()`并传入语法树结点即可无感知地获取到执行结果。所以这里着重介绍查询，更新，删除和插入操作对应Executor的实现。

- Scan：分为SeqScan和IndexScan，前者对表格中的数据进行顺序查找，后者利用条件语句中属性的索引进行查找，若查询条件中有属性具有索引，则会生成IndexScan的执行计划。具体实现介绍如下：
  - SeqScan：从SeqScanPlanNode中拿出CatalogManager，从CatalogManager中拿出table的信息，然后再判断是否有条件(Predicate)，没有就直接遍历表，将所有数据加入到Result_set中。有的话，在遍历时调用相应AbstractExpressionRef的Evaluate递归判断所有条件即可
  - IndexScan：IndexScan首先需要从IndexScanPlanNode中拿到table的信息，然后需要遍历表达式树获取条件。条件节点分为四类，LogicExpression，ComparisonExpression，ColumnValueExpression和ConstantValueExpression。LogicExpresssion记录了谓词语句之间的连接符号。关于其余三者，ComparisonExpression，CompaColumnValueExpression和ConstantValueExpression。ComparisonExpression记录了条件语句的符号，后两者中前者保存了有索引的具体属性在原Schema中的列号，后者保存了需要比较的值。由于默认只有and连接，我们只需用map存取Column和Constant里的信息，以及Comparison的比较符号，对每个有索引的属性做一次Scan，将每次Scan的结果作一次Intersect，就可以得到索引列的所有结果。但是考虑到可能还有条件属性没有索引，对于这种情况，我们仍旧需要遍历Intersect的结果，在图中调用Predicate的Evaluate语句进行递归的条件判断，从而得到最终的Result_set。Predicate树的一个可能示例如下

    ```
    LogicType::And(2)
    Constant(ret: kTypeInt, 0) Comparision(ret: kTypeInt, 2)
    LogicType::Or(child 2) Column(row 0 col 3 child 0)
    Constant(kTypeFloat,0) Comparision(ret:kTypeInt, 2)
    LogicType::and(child 2) Column(row 0 col 2 child 0)
    Constant(ret: kTypeInt, 0) Comparision(child 2)
    Comparision(2) Column(0 1 0)
    Column(0, 0) Constant(0)
    ```

- Insert: Insert操作需要同时更新索引和table的内容，由SeqScan Executor提供需要插入的数据。我们通过SeqScan Executor提供的数据，进行数据插入。首先要利用索引，检查索引列是否有重复数据项，然后先对table_heap进行更新，再对index进行更新

- Update: Update操作首先检查索引列的重复，再进行Update。对于表格，直接调用底层函数UpdateTuple，对于索引，需要先Remove再进行Insert。
- Delete：和Update操作十分类似，首先检查存在性，再进行数据项和索引项的删除。

三、测试以及功能验证

模块测试

- Disk_manager: 利用给出的BitmapTest和DiskMangaerTest进行测试。BitmapTest对Bitmap的Allocate和Deallocate功能进行了测试。DiskManagerTest对DiskManager的Allocate和DeAllocate的功能正确性进行了测试
- Buffer_manager: 利用给出的BufferPoolManagerTest进行测试。该测试中，对BufferManager的接口NewPage，FetchPage, FlushPage, UnpinPage进行了测试
- Record_manger: 利用给出的TupleTest和Table_heap test进行了测试，TupleTest中对Field和Column的对象进行了序列化和反序列化的测试，本人补充了Row的序列化和反序列化测试。Table_heap_test中对Table_heap的接口insertTuple, GetTuple和Table Iterator进行了测试，本人补充了Update的测试。
- Index_manager：利用给出的BPlusTreeTests, BPlusTreeIndexTests和IndexIteratorTest三者对该模块进行了测试。第一个测试了B+ tree的插入和删除操作，第二个测试了利用B+ tree作为底层数据结构的索引操作，包括Insert, Scan。第三个测试了B+ tree的插入和删除操作，以及各个操作后的叶子遍历操作。
- Catalog_manager: 利用给出的CatalogTest进行验证，该测试中，测试了CatalogMeta的序列化和反序列化，Table的Create和Load, Index的Create和Load.
- Exeucor: 给出了三条sql语句的测试，主要还是通过直接输入sql语句进行验证，具体参考如下的验收流程

    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：

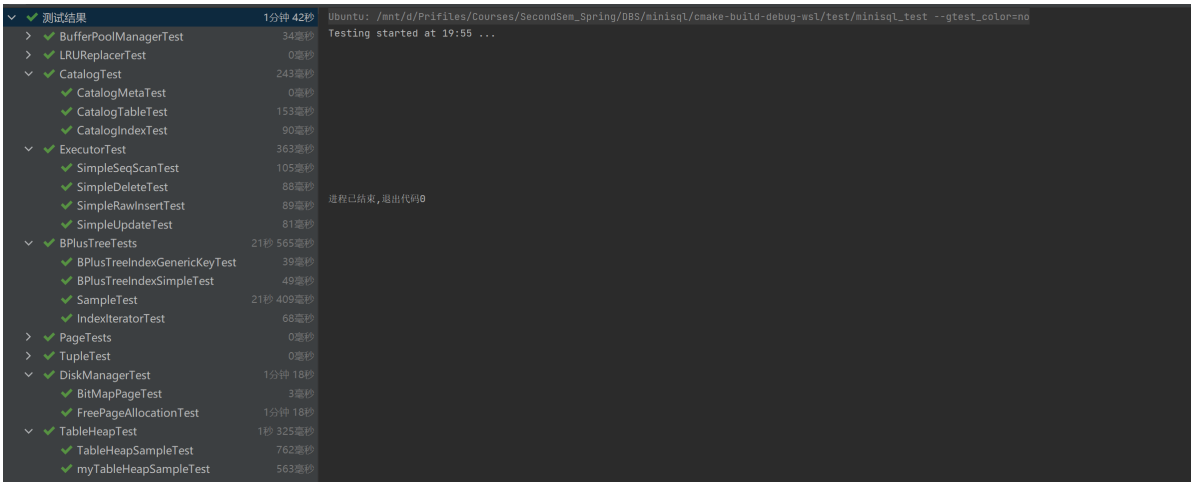
```
create table account(
  id int,
  name char(16) unique,
  balance float,
  primary key(id)
);

-- Note: 在实现中自动为UNIQUE列建立B+树索引的情况下，
--       这里的NAME列不加UNIQUE约束，UNIQUE约束将另行考察。
--           （NAME列创建索引的时候，不需要限制只有UNIQUE列才能建立索引）
```

    3. 考察SQL执行以及数据插入操作：
        1. 执行数据库文件`sql.txt`，向表中插入100000条记录（分10次插入，每次插入10000条，至少插入30000条）
            1. 参考SQL数据，由脚本自动生成：验收数据.zip
            2. 批量执行时，所有sql执行完显示总的执行时间
        1. 执行全表扫描`select * from account`，验证插入的数据是否正确（要求输出查询到100000条记录）
    4. 考察点查询操作：
        1. `select * from account where id = ?`

    2. `select * from account where balance = ?`
    3. `select * from account where name = "name56789"`，此处记录执行时间t_1
    4. `select * from account where id <> ?`
    5. `select * from account where balance <> ?`
    6. `select * from account where name <> ?`
5. 考察多条件查询与投影操作：
    1. `select id, name from account where balance >= ? and balance < ?`
    2. `select name, balance from account where balance > ? and id <= ?`
    3. `select * from account where id < 12515000 and name > "name14500"`
    4. `select * from account where id < 12500200 and name < "name00100"`，此处记录执行时间t_5
6. 考察唯一约束：
    1. `insert into account values(?, ?, ?)`，提示PRIMARY KEY约束冲突或UNIQUE约束冲突
7. 考察索引的创建删除操作、记录的删除操作以及索引的效果：
    1. `create index idx01 on account(name)`
    2. `select * from account where name = "name56789"`，此处记录执行时间t_2，要求$t_2 < t_1$
    3. `select * from account where name = "name45678"`，此处记录执行时间t_3
    4. `select * from account where id < 12500200 and name < "name00100"`，此处记录执行时间t_6，比较t_5和t_6
    5. `delete from account where name = "name45678"`
    6. `insert into account values(?, "name45678", ?)`
    7. `drop index idx01`
    8. 执行(c)的语句，此处记录执行时间t_4，要求$t_3 < t_4$
8. 考察更新操作：
    1. `update account set id = ?, balance = ? where name = "name56789";`并通过`select`操作验证记录被更新
9. 考察删除操作：
    1. `delete from account where balance = ?`，并通过`select`操作验证记录被删除
    2. `delete from account`，并通过`select`操作验证全表被删除
    3. `drop table account`，并通过`show tables`验证该表被删除

所有测试通过结果如下：



功能验证

- 数据库创建，切换，展示以及删除

```
squhuang@LAPTOP-JJGT7A0G:~/minisql/build/bin$ ./main
minisql > create database db0;
create database db1;
create database db2;
use db0;
show databases;query OK(0.0212sec)
minisql > query OK(0.0194sec)
minisql >
query OK(0.0229sec)
minisql > Database changed
minisql >
+--------------------+
| Database           |
+--------------------+
| db0                |
| db1                |
| db2                |
+--------------------+
I20230621 20:38:28.740600 54055 execute_engine.cpp:366] 3 rows in set (3.1e-05sec)
```

```
minisql > drop database db0;
query OK(0.0207sec)
minisql > show databases;

+--------------------+
| Database           |
+--------------------+
| db1                |
| db2                |
+--------------------+
I20230621 20:39:00.655232 54055 execute_engine.cpp:366] 2 rows in set (7.8e-05sec)
```

- 表创建，展示，删除

```
minisql > create table account(
  id int,
  name char(16) unique,
  balance float,
  primary key(id)
);
Query OK, 0 rows affected (0.000446sec)
minisql > create table person (
        height float unique,
        pid int,
        name char(32),
        identity char(128) unique,
        age int unique,
        primary key(pid)
);
Query OK, 0 rows affected (0.000371sec)
minisql > show tables;
+--------------------+
| db1                |
+--------------------+
| person             |
| account            |
+--------------------+
2 row in set(0.0002 sec).
```

```
minisql > show tables;
+--------------------+
| db1                |
+--------------------+
| person             |
| account            |
+--------------------+
2 row in set(0.0002 sec).
minisql > drop table perso
Query OK. (0.000sec)
minisql > show tables;
+--------------------+
| db1                |
+--------------------+
| account            |
+--------------------+
1 row in set(0.0001 sec).
```

- 索引创建，展示，删除

```
minisql > show indexes;
+-------------------------+
| Indexes_in_account      |
+-------------------------+
| index_name              |
| index_id                |
+-------------------------+

Query OK. (0.000198sec)
minisql > create index idx01 on account(name);
Query OK, 0 rows affected (95.1sec)
minisql > show indexes;
+-------------------------+
| Indexes_in_account      |
+-------------------------+
| idx01                   |
| index_name              |
| index_id                |
+-------------------------+

Query OK. (9.3e-05sec)
minisql > drop index idx01;
Query OK. (65.7sec)
minisql > show indexes;
+-------------------------+
| Indexes_in_account      |
+-------------------------+
| index_name              |
| index_id                |
+-------------------------+

Query OK. (8.5e-05sec)
```

- 记录插入，查询，删除，更新
  - 运行`execfile "account00.txt"..."execfile "account09.txt`插入10w条数据，并进行查询展示。

```
| 12599971 | name99971 | 210.339996 |
| 12599972 | name99972 | 787.130005 |
| 12599973 | name99973 | 220.149994 |
| 12599974 | name99974 | 28.320000  |
| 12599975 | name99975 | 901.989990 |
| 12599976 | name99976 | 160.229996 |
| 12599977 | name99977 | 609.859985 |
| 12599978 | name99978 | 870.059998 |
| 12599979 | name99979 | 126.879997 |
| 12599980 | name99980 | 416.450012 |
| 12599981 | name99981 | 204.250000 |
| 12599982 | name99982 | 707.140015 |
| 12599983 | name99983 | 600.219971 |
| 12599984 | name99984 | 431.040009 |
| 12599985 | name99985 | 658.559998 |
| 12599986 | name99986 | 576.140015 |
| 12599987 | name99987 | 702.179993 |
| 12599988 | name99988 | 185.309998 |
| 12599989 | name99989 | 965.760010 |
| 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(0.7840 sec).
```

- 进行范围查询：`select id, name from account where balance >= 500 and balance < 9999`

```
| 12599953 | name99953 |
| 12599955 | name99955 |
| 12599957 | name99957 |
| 12599958 | name99958 |
| 12599960 | name99960 |
| 12599961 | name99961 |
| 12599962 | name99962 |
| 12599963 | name99963 |
| 12599964 | name99964 |
| 12599966 | name99966 |
| 12599967 | name99967 |
| 12599969 | name99969 |
| 12599970 | name99970 |
| 12599972 | name99972 |
| 12599975 | name99975 |
| 12599977 | name99977 |
| 12599978 | name99978 |
| 12599982 | name99982 |
| 12599983 | name99983 |
| 12599985 | name99985 |
| 12599986 | name99986 |
| 12599987 | name99987 |
| 12599989 | name99989 |
| 12599992 | name99992 |
| 12599993 | name99993 |
| 12599995 | name99995 |
+----------+-----------+
50177 row in set(0.4530 sec).
```

- 更新后查询

```
minisql >
  select * from account where name = "name08988";
+----------+-----------+------------+
| id       | name      | balance    |
+----------+-----------+------------+
| 12508988 | name08988 | 209.740005 |
+----------+-----------+------------+
1 row in set(0.0020 sec).
minisql > update account set id = 1, balance = 3 where name = "name08988";
Query OK, 1 row affected(0.2160 sec).
minisql >   select * from account where name = "name08988";
+----+-----------+----------+
| id | name      | balance  |
+----+-----------+----------+
| 1  | name08988 | 3.000000 |
+----+-----------+----------+
1 row in set(0.0020 sec).
```

- 删除后查询

```
minisql >  delete from account where name = "name08988";
Query OK, 1 row affected(0.2060 sec).
minisql > select * from account where name = "name08988";
Empty set(0.0010 sec).
```

- quit后数据持久化，quit后依旧有数据

```
minisql > quit;
Bye.
squhuang@LAPTOP-JJGT7A0G:~/minisql/build/bin$ ./main
minisql > show databases;

+--------------------+
| Database           |
+--------------------+
| db0                |
| db1                |
| db2                |
+--------------------+
```