2 RECORD MANAGER

2.1 实验概述

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。

2.2 记录与模式

在实现通过堆表来管理记录之前，先做一个小的热身项目，这是一个有关数据的序列化和反序列化操作的任务。为了能够持久化存储上面提到的`Row`、`Field`、`Schema`和`Column`对象，我们需要提供一种能够将这些对象序列化成字节流（`char*`）的方法，以写入数据页中。与之相对，为了能够从磁盘中恢复这些对象，我们同样需要能够提供一种反序列化的方法，从数据页的`char*`类型的字节流中反序列化出我们需要的对象。总而言之，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。

```
// 逻辑对象
class A {
    int id;
    char *name;
};

// 以下是序列化和反序列化的伪代码描述
void SerializeA(char *buf, A &a) {
    // 将id写入到buf中，占用4个字节，并将buf向后推4个字节
    WriteIntToBuffer(&buf, a.id, 4);
    WriteIntToBuffer(&buf, strlen(a.name), 4);
    WriteStrToBuffer(&buf, a.name, strlen(a.name));
}

void DeserializeA(char *buf, A *&a) {
    a = new A();
    // 从buf中读4字节，写入到id中，并将buf向后推4个字节
```

```
    a->id = ReadIntFromBuffer(&buf, 4);
    // 获取name的长度len
    auto len = ReadIntFromBuffer(&buf, 4);
    a->name = new char[len];
    // 从buf中读取len个字节拷贝到A.name中，并将buf向后推len个字节
    ReadStrFromBuffer(&buf, a->name, len);
}
```

为了确保我们的数据能够正确存储，我们在上述提到的`Schema`和`Column`对象中都引入了魔数`MAGIC_NUM`，它在序列化时被写入到字节流的头部并在反序列化中被读出以验证我们在反序列化时生成的对象是否符合预期。

在本节中，你需要完善`Row`、`Schema`和`Column`对象各自的`SerializeTo`、`DeserializeFrom`和`GetSerializedSize`方法，具体以何种方式进行序列化（即需要序列化类中的哪些数据）由你自行决定，我们在测试代码中只会验证序列化前后的对象是否匹配。为了避免同学们对这块内容毫无头绪，我们保留了`Field`类型对象的序列化和反序列化操作，用于提供参考。

在本节中你需要完成如下函数：

- `Row::SerializeTo(*buf, *schema)`
- `Row::DeserializeFrom(*buf, *schema)`
- `Row::GetSerializedSize(*schema)`
- `Column::SerializeTo(*buf)`
- `Column::DeserializeFrom(*buf, *&column)`
- `Column::GetSerializedSize()`
- `Schema::SerializeTo(*buf)`
- `Schema::DeserializeFrom(*buf, *&schema)`
- `Schema::GetSerializedSize()`

其中，`SerializeTo`和`DeserializeFrom`函数的返回值为`uint32_t`类型，它表示在序列化和反序列化过程中`buf`指针向前推进了多少个字节。

对于`Row`类型对象的序列化，对于为`null`的`Field`，可以通过位图的方式标记(即 *Null Bitmaps*)；对于`Row`类型对象的反序列化，在反序列化每一个`Field`时，反序列化出来的`Field`的内存都由该`Row`对象维护。对于`Column`和`Schema`类型对象的反序列化，分配后新生成的对象于参数`column`和`schema`中返回，以下是一个简单的例子：

```
uint32_t Column::DeserializeFrom(char *buf,
                                 Column *&column){
  if (column != nullptr) {
    LOG(WARNING) << "Pointer to column is not null in column deserialize."
<< std::endl;
  }
  /* deserialize field from buf */
    ......

  /* allocate object */
  if (type == kTypeChar) {
    column = new Column(column_name, type, col_len, col_ind, nullable, unique);
  } else {
    column = new Column(column_name, type, col_ind, nullable, unique);
```

```
  }
  return ofs;
}
```

此外，在序列化和反序列化中可以用到一些宏定义在`src/include/common/macros.h`中，可根据实际需要使用：

```
#define MACH_WRITE_TO(Type, Buf, Data) \
           do { \
              *reinterpret_cast<Type *>(Buf) = (Data); \
           } while (0)
#define MACH_WRITE_UINT32(Buf, Data) MACH_WRITE_TO(uint32_t, (Buf), (Data))
#define MACH_WRITE_INT32(Buf, Data) MACH_WRITE_TO(int32_t, (Buf), (Data))
#define MACH_WRITE_STRING(Buf, Str)      \
           do {                                        \
              memcpy(Buf, Str.c_str(), Str.length()); \
           } while (0)

#define MACH_READ_FROM(Type, Buf) (*reinterpret_cast<const Type *>(Buf))
#define MACH_READ_UINT32(Buf) MACH_READ_FROM(uint32_t, (Buf))
#define MACH_READ_INT32(Buf) MACH_READ_FROM(int32_t, (Buf))

#define MACH_STR_SERIALIZED_SIZE(Str) (4 + Str.length())
```

2.3 通过堆表管理记录

2.3.1 RowId

对于数据表中的每一行记录，都有一个唯一标识符`RowId`（`src/include/common/rowid.h`）与之对应。`RowId`同时具有逻辑和物理意义，在物理意义上，它是一个64位整数，是每行记录的唯一标识；而在逻辑意义上，它的高32位存储的是该`RowId`对应记录所在数据页的`page_id`，低32位存储的是该`RowId`在`page_id`对应的数据页中对应的是第几条记录（详见#2.3.2）。`RowId`的作用主要体现在两个方面：一是在索引中，叶结点中存储的键值对是索引键`Key`到`RowId`的映射，通过索引键`Key`，沿着索引查找，我们能够得到该索引键对应记录的`RowId`，也就能够在堆表中定位到该记录；二是在堆表中，借助`RowId`中存储的逻辑信息（`page_id`和`slot_num`），可以快速地定位到其对应的记录位于物理文件的哪个位置。

2.3.2 堆表

堆表（`TableHeap`，相关定义在`src/include/storage/table_heap.h`）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。堆表中的记录通过`RowId`进行定位。`RowId`记录了该行记录所在的`page_id`和`slot_num`，其中`slot_num`用于定位记录在这个数据页中的下标位置。

堆表中的每个数据页（与课本中的`Slotted-page Structure`给出的结构基本一致，见下图，能够支持存储不定长的记录）都由表头（Table Page Header）、空闲空间（Free Space）和已经插入的数据（Inserted Tuples）三部分组成，与之相关的代码位于`src/include/page/table_page.h`中，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。具体的实现细节请自行参考实现代码。

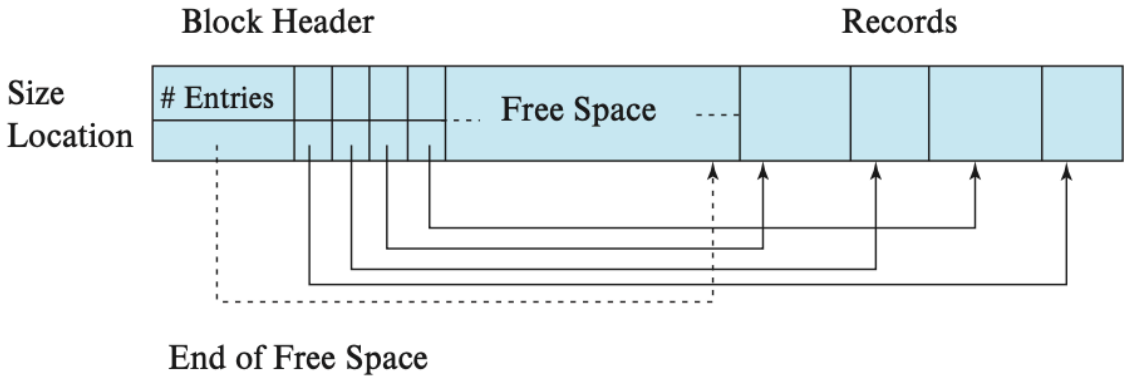


Figure 13.6 Slotted-page structure.

当向堆表中插入一条记录时，一种简单的做法是，沿着`TablePage`构成的链表依次查找，直到找到第一个能够容纳该记录的`TablePage`（*First Fit* 策略）。当需要从堆表中删除指定`RowId`对应的记录时，框架中提供了一种逻辑删除的方案，即通过打上Delete Mask来标记记录被删除，在之后某个时间段再从物理意义上真正删除该记录（本节中需要完成的任务之一）。对于更新操作，需要分两种情况进行考虑，一种是`TablePage`能够容纳下更新后的数据，另一种则是`TablePage`不能够容纳下更新后的数据，前者直接在数据页中进行更新即可，后者的实现方式留给同学们自行思考。此外，在堆表中还需要实现迭代器`TableIterator`(`src/include/storage/table_iterator.h`)，以便上层模块遍历堆表中的所有记录。

更新：在`TablePage::UpdateTuple`函数中，返回的是`bool`类型的结果，其中返回`true`表示更新成功，返回`false`表示更新失败。但更新失败可能由多种原因造成，只用一个`false`无法区分更新失败的原因。可以采取以下两种做法：（1）更改返回值为`int`类型；（2）参数列表中增加一个参数表示返回状态；（不太理解的同学可以移步评论区看细节）

综上，在本节中，你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，具体需要实现的函数如下：

- `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；
  - ……

提示：一个使用迭代器的例子

```cpp
for (auto iter = table_heap.Begin(); iter != table_heap.End(); iter++) {
    Row &row = *iter;
    /* do some things */
}
```

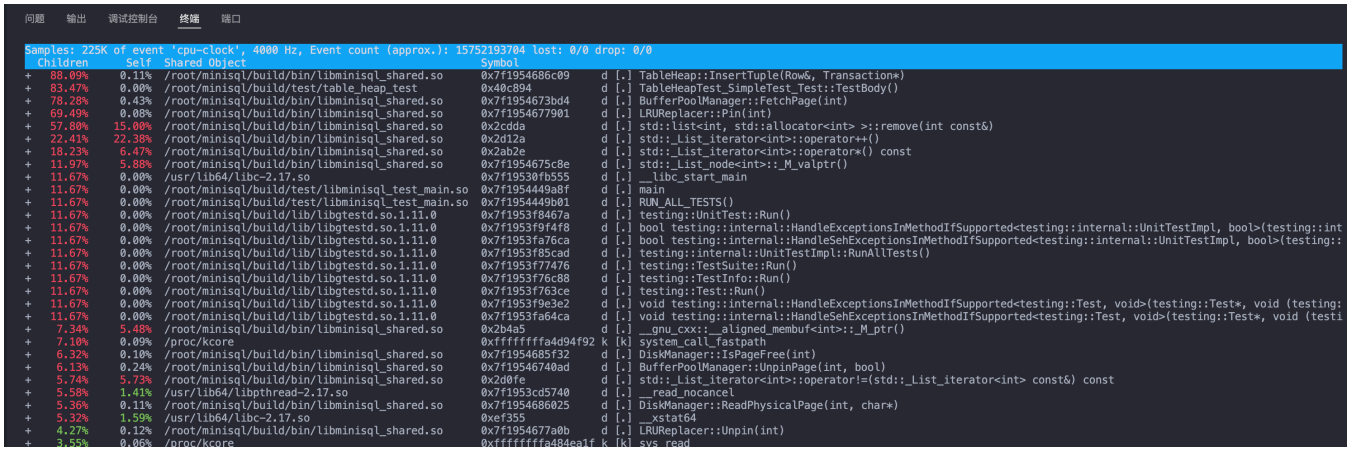
Bonus: 优化堆表（`TableHeap`）以及数据页（`TablePage`）的实现，通过使用额外的空间记录一些元信息来加速`Row`的插入、查找和删除操作。

2.4 模块相关代码

- `src/include/record/row.h`
- `src/record/row.cpp`
- `src/include/record/schema.h`
- `src/record/schema.cpp`
- `src/include/record/column.h`
- `src/record/column.cpp`
- `src/include/storage/table_iterator.h`
- `src/storage/table_iterator.cpp`
- `src/include/storage/table_heap.h`
- `src/storage/table_heap.cpp`
- `test/record/tuple_test.cpp`
- `test/storage/table_heap_test.cpp`

2.5 开发提示

1. 推荐在**夏学期第3周前**完成本模块的设计。
2. `Linux`系统下可以使用性能测试工具`perf`来剖析性能，找到运行热点（即运行时开销较大的函数或指令）。一个基本的例子：`perf top -a -g -p <进程PID>`，可以看到在插入大量数据时`InsertTuple`是一个性能热点，为此可以通过优化插入算法来提升系统的整体性能。对性能调优感兴趣的同学可以自行上网学习有关`perf`工具的具体用法。



1. 对于大型c++项目，因为指针的存在，如果程序中存在内存申请后没有释放就会造成内存泄漏，从而产生严重的问题。因此，在申请内存并把它交给一个指针后，需要思考这个指针指向的内存何时会被释放掉。使用智能指针可以帮助你进行内存管理，如果不想用智能指针，那么new和delete必须配对使用。可以采用valgrind和Asan等内存检查工具，帮助你检查内存泄漏等问题（可参考 配置valgrind和asin来检查内存泄漏）。如果使用Asan，配置好后可打开CMakeLists.txt中的-fsanitize=address 编译开关。
2. 在插入大量记录时，如果运行缓慢，可以使用`release`模式编译代码，即在编译时指定`CMAKE_BUILD_TYPE=Release`，但`release`模式下可能会丢失一些调试信息。

```
mkdir build
cd build
cmake .. -DCMAKE_BUILD_TYPE=Release
```

2.6 评论

有关 WSL2 使用 perf ，可以参考这篇文章：https://stackoverflow.com/questions/60237123/is-there-any-method-to-run-perf-under-wsl

补充一下，这里同样也可以下载 perf ，解压 make 之后就可以直接使用了

文档提到TablePage::UpdateTuple需要区分false的原因，那么TableHeap:UpdateTuple是否也需要区分false的原因呢?
如果不区分false的原因，那么当TablePage不能够容纳下更新后的数据时，TableHeap:UpdateTuple应该delete和insert
如果区分false的原因，直接return即可

YingChengJun2022-05-11 13:03

对的，如果TablePage中需要区分的话，那么TableHeap中也需要区分。

[![fakeyyy](https://mdn.alipayobjects.com/huamei_0prmtq/afts/img/A*khrYRYi6VN0AAAAAAAAAAA

fakeyyyYingChengJun2022-05-11 14:38

假设函数A调用TableHeap:UpdateTuple，但是失败了且是由于TablePage不能够容纳下更新后的数据，那么A应该delete和insert。也就是说，TableHeap:UpdateTuple的语义不包括delete和insert，是吗?

YingChengJunfakeyyy2022-05-11 14:52

想了一下，TableHeap:UpdateTuple的语义还是应该包括DEL和INS的。
这里我的想法是，TableHeap::UpdateTuple从设计上来说应该是对上层调用者（一般来说就是执行器了）屏蔽实现细节的，也就是说上层调用者不应该知道UpdateTuple的过程中这条记录是否由于空间不够而被挪到了其它的TablePage中。只要调用接口的行为不是非法的（比如传入一个不存在的RowID），那么TableHeap::UpdateTuple返回的结果就应该返回TRUE。
如果TableHeap::UpdateTuple在调用TablePage:UpdateTuple的过程中，发现数据页不够了，那这个时候TableHeap应该自己做一次DEL+INS操作，如果能够正确执行，那么就向上层返回TRUE。
所以这里需要解决的问题在于TableHeap::UpdateTuple怎么知道TablePage:UpdateTuple返回FALSE时，到底是因为调用接口的行为不合法还是说是TablePage的空间不能够容纳更新后的数据。

[![fakeyyy](https://mdn.alipayobjects.com/huamei_0prmtq/afts/img/A*khrYRYi6VN0AAAAAAAAAAA

fakeyyyYingChengJun2022-05-11 15:18

嗯嗯，本来也是这样想的。
但是看到了table_heap.h中对UpdateTuple的注释：if the new tuple is too large to fit in the old page, return false (will delete and insert)，所以就产生了疑惑，按照注释来说，都return false了，那DEL和INS当然是上层来做了。
感谢解答！

- 👍 9

BrucecaiYingChengJun2022-05-11 20:12

是不是可以理解为TabelPage::UpdateTuple需要返回一个值给它的上一层TableHeap::UpdateTuple能够区分是因为什么原因导致的不合法，而在TableHeap::UpdateTuple只需要返回True和False就可以了。

YingChengJunBrucecai2022-05-11 22:57

是的，这样是对现有接口改动最小的做法

VatineDJC2022-05-13 17:19

请问一下row序列化函数里的schema参数是用来做什么的

YingChengJun2022-05-13 20:40

这个参数可以不用到，Row里面的信息结合Schema信息可以打ASSERT做一些调试验证

if you2022-05-14 21:05

请问本文档中Row::GetSerializedSize()的参数是漏写了吗

YingChengJun2022-05-17 11:29

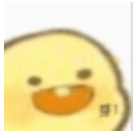
已更正

应乔松2022-05-17 14:42

●TableHeap:UpdateTuple(&new_row, &rid, *txn)要求将new_row的RowId通过new_row.rid*返回；但是在框架内部TableHeap:UpdateTuple的new_row 是一个常引用，这样无法修改new_row.rid*的值。请问是在内部做强制类型转换，还是我对函数的理解有问题呢?

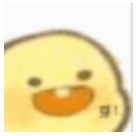
YingChengJun2022-05-17 15:21

这里是之前设计的时候没有考虑周全，目前先做一个强转或者把函数的const参数去掉都可以。



NieNieNie2022-05-22 16:55

能请教一下，对于class Row，如果GetSerializedSize的时候只传入参数schema，如何保证GetSerailizedSize的大小和SerializeTo时的序列大小以及DeserializedFrom时的反序列大小保持一致呢?
因为值为null的field是不会序列化的，以此来节省空间，但是schema并没有包含真实的数据，所以GetSerializedSize不知道哪些field是null，它算出的值不能和序列化的实际大小保持一致。



NieNieNie2022-05-22 17:00

文档里说 "对于Row类型对象的序列化，可以通过位图的方式标记为null的Field(即Null Bitmaps

)"，但如果采用位图的方式，并且之后不存值为null的记录，感觉就会出现上面提到的问题

YingChengJun2022-05-22 19:50

前面的评论区也提到了schema的作用。
row在序列化/获取写入长度时，使用里面的field进行计算，而不是使用schema进行计算。

oneko2022-05-23 15:58

在运行table_heap_test时，将row_nums从1000改成100000时，插入到69000左右时，bufferpool中所有的数据页均被固定，导致无法NewPage()，后续返回空值，ASSERT_EQ(row_nums, row_values.size());报错。 由于插入时只有FetchPage修改了pin_count，在检查所有Unpin的调用正确后仍有问题。助教建议修改bufferpool的size，将位于config.h中的DEFAULT_BUFFER_POOL_SIZE从1024修改为2048后可以通过大样例测试。此外助教建议debug时：“buffer pool满的话，最好加一个LOG输出，或者ASSERT直接把程序挂掉，这样容易及时发现问题”。

HarryRookie2023-06-06 21:51IP 属地浙江

TableIterator::operator==以及TableIterator::operator!=在头文件中声明为inline函数，内联函数应在头文件中定义，而不是在.cpp中，否则可能会造成链接错误：inline function 'bool TableIterator::operator==' used but never defined以及`undefined reference to`TableIterator::operator==(TableIterator const&) const' `