#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，用于存储记录和索引键，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。

Bonus: 对于`float`类型的数据，在代码中用常规的`float`类型的变量存储会造成精度丢失问题。可以通过修改`src/include/record/types.h`中`float`类型的相关定义以及`src/include/record/field.h`、`src/record/column.cpp`中有关`float`类型数据的存储来解决精度丢失的问题。

#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。

一个简单的序列化例子 C++ 复制代码

```cpp
// 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`Row`、`Field`、`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()`
- `Column::SerializeTo(*buf)`
- `Column::DeserializeFrom(*buf, *&column, *heap)`
- `Column::GetSerializedSize()`
- `Schema::SerializeTo(*buf)`
- `Schema::DeserializeFrom(*buf, *&schema, *heap)`
- `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，可以通过位图的方式标记为`null`的`Field`（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，需要将自身的`heap_`作为参数传入到`Field`类型的`Deserialize`函数中，这也意味着所有反序列化出来的`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，将使用`MemHeap`类型的对象`heap`来分配空间，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：

```cpp
uint32_t Column::DeserializeFrom(char *buf,
                                 Column *&column,
                                 MemHeap *heap){
  if (column != nullptr) {
    LOG(WARNING) << "Pointer to column is not null in column deserialize."                  << std::endl;
  }
  // can be replaced by:
  //    ALLOC_P(heap, Column)(column_name, type, col_ind, nullable, unique);
  void *mem = heap->Allocate(sizeof(Column));
  column = new(mem)Column(column_name, type, col_ind, nullable, unique);
  /* deserialize field from buf */
  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：

```cpp
#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())

#define ALLOC(Heap, Type) new(Heap.Allocate(sizeof(Type)))Type
#define ALLOC_P(Heap, Type) new(Heap->Allocate(sizeof(Type)))Type
#define ALLOC_COLUMN(Heap) ALLOC(Heap, Column)
```

Bonus: `MemHeap` 通常用于内存分配和回收的管理，它能在析构时自动释放分配的内存空间，在一定程度上能够避免内存泄漏的问题。同时，合理的内存分配方式能够避免多次频繁调用 `malloc` 对性能造成的影响。本实验框架中仅给出了 `SimpleMemHeap` 用于简单的内存分配和回收。若实现一种新的内存分配和管理方式（继承 `MemHeap` 类实现其分配和回收函数）提高内存分配和回收的性能可以获得一定的加分。

#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。

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。

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, *schema)`：向堆表中插入一条记录，插入记录后生成的 `RowId` 需要通过 `row` 进行返回；
- `TableHeap:UpdateTuple(&new_row, *old_row, *schema)`：将 `RowId` 为 `old_row->row_id` 的记录修改为 `new_row`，并将旧的记录拷贝至 `old_row` 对象中，同时在 `new_row` 中传出新纪录 `row_id`；
- `TableHeap:ApplyDelete(&rid)`：从物理意义上删除这条记录；
- `TableHeap:GetTuple(*row, *schema)`：获取 `RowId` 为 `row->row_id` 的记录；
- `TableHeap:GetFirstTupleRid(*first_rid)`：获取第一条记录的 `RowId`
- `TableHeap:GetNextTupleRid(&cur_rid, *next_rid)`：获取 `RowId` 为 `cur_rid` 对应记录的下一条记录的 `RowId`；
- `TableHeap::Begin()`：获取堆表的首迭代器；
- `TableHeap::End()`：获取堆表的尾迭代器；
- `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

C++ | 复制代码

```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。

3. 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。

Release模式下编译命令 Bash 复制代码

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

#2.6 诚信守则

1. 请勿从其它组或在网络上找到的其它来源中复制源代码，一经发现抄袭，成绩为 `0` ；
2. 请勿将代码发布到公共Github存储库上。

57a7c4859acf.png&title=%232%20RECORD%20MANAGER%20%C2%B7%20%E8%AF%AD%E9%9B%80%20%7C