# 数据库系统原理与实现

姓名：洪宇
学号：2022080901022

## 实验一

### 设计

首先需要说明 project 1 的整体设计。

在磁盘上的存储采用的是一表一文件，但由于整个项目并没有实际上落盘，因此基本可以忽略；在内存中的存储以块（block）为单位，每一个块是在内存中的一块连续的空间。为什么每个块必须是一块连续的空间呢？这是因为，在实际的应用中，需要将内存中的记录刷到磁盘上，而记录在磁盘上的存储便是以块这一具有连续存储空间的单位组织起来的。本实验主要涉及到的是数据块（data block）和超块（super block）。

#### 数据块

数据块是真正存储记录的块，其结构可以（从逻辑上）自上而下分为头部（header）、数据区、空闲区（free space）和尾部（trailer）。在它们所存储的数据中，本实验需要关注的有：

#### 头部

`unsigned short type`：
块的类型。数据块对应的宏为 `const unsigned short BLOCK_TYPE_DATA = 2`。

`unsigned short freespace`：
指向空闲区起始位置的指针，即从块的起始地址到空闲区起始地址的偏移量。随着插入记录的增多，数据区也会不断向下扩展，同时尾部因为存储了指向数据区中各记录的指针，所以也会向上扩展，因此空闲区会受到两个方向的挤压。因此，空闲区大小便等于块大小减去空闲区起始地址偏移量再减去尾部大小，即 `BLOCK_SIZE - getTrailerSize() - be16toh(header->freespace)`。

`unsigned short freesize`：
逻辑上可用空间的大小。为什么是“逻辑上”呢？它和 free space 的大小有什么区别呢？

逻辑上的空闲空间大小和实际上的大小是不等的。这是因为，在删去一条记录时，仅将该记录的删除位（tombstone）置1，因此虽然逻辑空闲空间大小会加上该记录的长度，但在内存中该区域实际上仍未被释放。仅当需要插入记录，而逻辑空间足够但物理空间不够时，才会对数据区进行一次扫描，将所有删除位为1的记录释放，并同时将后面的记录向前移动。因此，`freespace` 所指向的空闲区是物理可用空间的大小，而 `freesize` 是逻辑可用空间的大小。

两者的关系为：

$$free\ size = free\ space + \sum_{tombstone(i)=1} sizeof(Record[i])$$

`unsigned int self`：
本块 id。在进行索引时需要找到特定的块，因此每个块需要有能唯一标识的 id。

`unsigned int next`：
下一个数据块的 id。在 project 1 中，数据块被组织为两个链表，分别为数据链和空闲（idle）链。其中，数据链的元素是存储了有效记录的块，因此 `next` 便指向了在数据链中的下一个块；空闲链的元素则是在内存中分配了空间、但没有被使用的块，每当需要使用一个新的块时，先从空闲链中获得，如果空闲链为空再在内存中分配新空间。

数据链和空闲链的头指针都存储在超块的头部中。

`unsigned short slots`：
尾部中 `slot` 的总数，即数据区中记录的数量。每个 `slot` 是一个结构体，包含了记录大小及记录在相对于块起始地址的偏移量。

在查找记录时，需要先在 `slots` 数组中查找对应的 `slot`，再根据其中的偏移量字段来获得记录。而为了获得指向 `slots` 数组所在内存区的指针，便需要根据 `slot` 总数来计算，即先通过块首地址加块大小定位到块末地址，再减去 `slots` 乘以 `sizeof(Slot)`。

#### 超块

在实验一中，超块最重要的作用便是提供数据链和空闲链的头指针。虽然还有时间戳、校验位、总块数等字段，但都和本实验没有很强的关联，因此不再赘述。

## 实现

在实验一中，数据块是由一个简单的单链表组织的，因此 CRUD 基本都在同一个块中进行。原有代码中已经给出了搜索（`searchRecord`）和插入（`insertRecord`）的实现，因此只需要实现删除（`removeRecord`）和更新（`updateRecord`）。

#### 删除

函数签名为：

```
bool DataBlock::removeRecord(std::vector<struct iovec>& iov);
```

#### iovec

`iovec` 是对一块内存区的描述，包含指向该内存区的指针及该内存区的大小。以插入记录为例，表示一条记录的类 `Record` 有一个成员字段 `buffer_`，其指向该记录在内存中的起始地

址；插入时，会先将一个记录的 `buffer_` 指向某一个块的空闲处，随后调用 `allocate` 来在块中为新记录分配空间，接着调用 `record.set()` 来把 `iov` 所指向的内存区的值拷贝到块中。

值得注意的一点是，在分配新记录的空间时，可能出现物理空间不够的情况（即虽然 `freesize` 够但 free space 不够），此时需要回收所有删除位为1的记录并扩充空闲区。这一过程已经在 `allocate` 中完成了，因此无需再实现。

**确定主键**

在 `removeRecord` 中，首先需要确定记录的第几个字段是主键，以及主键的数据类型，这样才可以根据主键来搜索块中的记录。诸如第几个字段是主键这样的元数据存储在类 `Table` 中，而每张表下的各个块均有一个指向 `Table` 对象的指针 `table_` 。这实际上是所有 CRUD 函数共有的起始操作：先根据 `table_` 来确定主键在记录中的下标 `keyIdx`，接着根据 `keyIdx` 来确定记录中该字段（即主键）的数据类型。

**定位记录**

在删除记录前需要先定位到记录。针对不同数据类型，搜索方式也有所区别，因此上面确定的主键数据类型就起到作用了：

```
// 确定该记录对应的 slot 下标
unsigned short index =
    keyType->search(buffer_, keyIdx, iov[keyIdx].iov_base, iov[keyIdx].iov_len);
if (index >= getSlots()) return false; // 记录不存在
```

翻看 `DataType` 中 `search` 的源码可知其返回的是 lower bound，即大于等于搜索值的第一个元素所在的下标。在 project 2 中会看到这其实是一个有些 tricky 的设计，在此先埋个伏笔。

注意到即使搜到的 `index` 是合法的，目标记录也仍有可能不存在。因此需要：

1. 获得该块中指向 `slots` 的指针；
2. 通过 `Record` 类来获得该 `slot` 指向的块中记录；
3. 通过 `memcmp` 比较块中记录和目标记录是否相等；
4. 若相等，则：
    i. 将记录的删除位置1；
    ii. 挤压 `slots` 数组；
    iii. 修改 `slot` 数目及 `freesize` 大小。
5. 否则，返回 `false` 。

**获得块中记录**

```
Slot *slots = getSlotsPointer();
Record record;
```

```
record.attach(
    buffer_ + be16toh(slots[index].offset), be16toh(slots[index].length));
```

这是本实验中初始化 `Record` 的标准做法。`buffer_` 是每个 `Block` 类的成员，指向该块在内存中的起始地址；通过加上 `slot` 所指示的块中偏移量，便可定位到记录位置。这里有个非常容易出错的要点：因为本实验考虑了字节序转换的问题，所以需要在主机序（`h`）和大端序（`be`）间转换。具体而言：

- 块中的数据均以大端序存储；
- **在把数据从块中取出并进行操作前，需要先用 `betoh` 转换为主机序；**
- **在把数据写入块中前，需要先用 `htobe` 转换为大端序。**

接下来通过 `record.getByIndex()` 来获取 `record` 中指定下标的字段。但这里有个比较棘手的地方：`getByIndex` 接受一块内存区的地址作为实参，从而把记录字段的值拷贝到该内存区中；同时，还需要传入该内存区的大小。那么问题来了：因为主键的数据类型是不定的，因此内存区大小也是不定的——这意味着我需要根据 `keyType` 来动态地初始化该内存区！

为了解决该问题，我的实现是：

1. 根据 `keyType` 获得主键数据类型的字节数；
2. 通过 `std::vector` 和字节数来定义一个 `char` 数组；
3. 将该数组的首地址和字节数传入 `getByIndex`。

```
inline size_t getKeyBytes(DataType *keyType) { return keyType->size; }

size_t keySize = getKeyBytes(keyType);
std::vector<char> tmpKey(keySize);
unsigned int tmpKeyLen = (unsigned int) iov[keyIdx].iov_len;

record.getByIndex((char *) &tmpKey[0], &tmpKeyLen, keyIdx);
```

这个技巧在 project 2 中会被广泛使用。因为在初始化 `iov` 时，也需要先申请一块内存，然后把 `iov` 指向该内存，并赋给它该内存的大小。无独有偶，这也会引出动态获取主键数据类型对应字节数的问题，因此我也是通过动态申请一块 `char` 数组来解决的。

**删除记录**

如果记录存在，便可以删除。在源码中已经有专门用于删除一条块中记录的 `deallocate` 函数，它接收记录在 `slots` 数组中的下标作为参数：

```
deallocate(index);
```

该函数已经实现了设置删除位、挤压 `slots` 数组等善后操作，至此 `removeRecord` 便完成了。

**更新**

`updateRecord` 同样只需传入一个 `iov` 作为实参。更新记录时，并非搜索到记录之后原地修改，而是先将原记录删除再插入一个新记录。乍一看这似乎很没效率，但注意到：

1. 块中数据区内的记录是连续排列的；
2. 可能有变长字段（VARCHAR）。

因此，假如有变长字段的话，原地修改便可能导致后面的字段全部需要移动！所以统一先删除、再插入才是更好的做法。那么只需要先调用 `removeRecord`，再根据返回值判断是否删除成功（即原记录是否存在），若成功再调用 `insertRecord`。

但这个实现实际上并不完整。因为可能有变长字段，所以在删除之后再插入时，仍有可能块中的空闲区不足！好在 `insertRecord` 返回值包含两部分，即是否插入成功及对应在 `slots` 数组中的下标。因此实现思路为：

1. 根据 `insertRecord` 的返回值判断是否需要分裂；
2. 若是，则进行分裂：
    i. 根据新记录应插入的下标，确定分裂位置；
    ii. 申请一个新的 `Block`；
    iii. 由分裂位置，移动部分记录到新 `Block` 中
    iv. 判断新记录应插入到旧的还是新的 `Block`，进行插入
    v. 维护数据链，即将旧 `Block` 的 `next` 指向新 `Block`。

分裂逻辑中的第1到3步被我封装到函数 `split` 中，因为在 project 2 的 `insert` 中仍可复用。

**申请新块**

在此有必要单独说明一下申请新块的过程，因为这涉及到本项目对缓冲区的实现。前面提到，一个块是一块在内存中的连续空间，而这些块则是由一个全局缓冲区 `kBuffer` 管理的。`kBuffer` 需要存储一个从（表名，块 id）到 `Block` 的映射；每次要使用一个块时，都必须向 `kBuffer` “借”，用完之后则需要“还”。这有点像在并发场景下给一个块加锁——在同一时刻，只允许一个用户拥有对一个块的使用权。为了实现这一点，`kBuffer` 会为借出的每个块提供一个描述符 `BufDesp`，其最重要的功能便是维护一个计数器，当借出时计数器加一，释放块时计数器减一，因此只有当计数器为0时才允许借出。

因为向缓冲区借块的操作非常频繁，所以我封装进了函数 `attachBuffer` 中。使用一个块（包括数据块和超块）的流程如下：

1. 声明该 `Block` 对象；
2. 设置该对象的 `table_` 指针；
3. 由该块的 `table_` 指向的表名及该块的 id，调用 `kBuffer.borrow()` 来获取 `BufDesp` 描述符；

4. 将该块的首地址 `buffer_` 置为描述符指向的缓冲区；
5. 块使用结束后，调用 `kBuffer.releaseBuf()` 来释放该块。

## 测试

project 1 的测试都放在了 `TEST_CASE("BlockTest", "[p1]")` 中，不同的 `SECTION` 对应了对不同函数的单元测试。要明确说明的是：**不同的 `TEST_CASE` 间有依赖关系**！这意味着，在前一个 `TEST_CASE` 中对内存里某些块作出的修改，在之后的 `TEST_CASE` 里也会保留。正因如此，不同的单元测试文件如果同时运行，将发生彼此之间的纠缠。以下几个测试文件是必要的：

- `bufferTest.cc`：初始化 `kBuffer`
- `fileTest.cc`：初始化其他全局变量，打开、写入数据库文件等
- `schemaTest.cc`：创建一张名为“table”的表，并填充元数据

好在 Catch2 提供了给 `TEST_CASE` 打标签的功能，于是给以上三个测试文件中的测试用例均打上 `[p1]` 的标签，要想只运行有 `[p1]` 标签的用例，只需：

1. 进入 `build` 文件夹
2. `.\make`
3. `.\bin\utest [p1]`

### 删除

主要分为以下三种情况：

- 块为空（无记录）
- 记录存在
- 记录不存在

### 初始化

各个 `SECTION` 的初始化都比较类似，基本步骤为：

1. 通过 `table.open("table")` 打开名为 `table` 的表（该表已经在 `schemaTest` 中进行了初始化）；
2. 向 `kBuffer` 借一个数据块；
3. 通过 `findDataType` 获得对应字段的 `DataType` 对象，该对象用于针对不同的数据类型采用不同的字节序转换函数（实质差异在于字节数不同）。在测试用例中，主键类型为 `BIGINT`，对应 C++ 数据类型为 `long long`；
4. 声明几个局部变量，随后声明一个 `iovec` 数组 `iov`，分别指向这些变量。

### 测试块为空

因为要频繁为 `iov` 赋值，所以将该操作封装在了 `setIov` 中。通过修改 `iov` 指向的内存的值再将 `iov` 传入 CRUD 函数中，便可搜索、插入、删除或修改对应的记录。

令数据块变量名为 `data`，调用 `data.clear()` 便可将该块清空，随后再通过 `REQUIRE(!data.removeRecord(iov));` 来验证删除失败。

注：在测试用例中，还分布了许多辅助检查，比如 `REQUIRE(bd);` 检查是否借到了数据块等。为了突出重点，辅助检查在报告中都不再赘述。

**测试记录存在**

首先用 `REQUIRE(data.insertRecord(iov).first);` 来插入记录，并验证记录插入成功；接着再 `REQUIRE(data.removeRecord(iov));` 来验证删除成功即可。

**测试记录不存在**

这里主要验证 `removeRecord` 中的 `memcmp` 是否起了作用，即当要删除的主键值在块的最大主键和最小主键之间、但该主键值并不存在时，是否能正确返回删除失败。因此，我首先插入了主键值为2、4、5的记录，再将 `iov` 的主键字段指向值为3的内存，再测试是否删除失败。

**说明**

因为单元测试的效果仅为测试用例是否通过的字样，因此将在后文中统一给出截图。当然，读者也可以按前文中指定 `[p1]` 标签的方法来验证。

**更新**

对应 `SECTION("update")` 中的代码。测试用例主要分为三部分：

- 更新后无分裂
- 记录不存在
- 更新后需要分裂出新块

初始化类似，以后都不再赘述。记录共3个字段，分别为 `BIGINT`（主键）、`CHAR`、`VARCHAR`。`CHAR` 是一个固定长度的 `char` 数组，而由于 `VARCHAR` 对应的内存区大小可变，所以用一个 `char *` 指针作为中介，再将该指针赋值给 `iov[2]`。

**无分裂**

初始化 `iov` 各元素后先插入，接着再用 `strcpy_s` 修改 `iov` 指向的第二块内存（`CHAR`），随后用 `REQUIRE(data.updateRecord(iov));` 来更新并验证返回值。为了真正证明成功修改，还需要进行以下操作：

1. 初始化一个 `Record` 对象。因为这是块中的第一个记录，所以该对象所指向的内存地址相对于块首地址的偏移量由 `slots[0].offset` 给出；
2. 通过 `record.getByIndex()` 获取块中记录的值，存到局部变量中；

3. 将局部变量内的值与 `iov` 的对应元素进行对比。比如，为了检查第二个字段，需使用 `strcmp`。

注意，该过程因为涉及到频繁地写入块和从块中取值，所以有很多字节序转换的过程！所有相关操作遵循前文给出的转换规则即可，以后都不再赘述。

**记录不存在**

将 `iov` 指向的主键值设为一个不存在与块中记录的值，随后 `REQUIRE(!data.updateRecord(iov));` 验证更新失败即可。

**有分裂**

1. 首先在一个 `while` 循环中不断插入记录，直到 `insertRecord` 返回值的第一部分为 `false`，即插入失败——表明再插入就要分裂了；
2. 将 `iov` 对应 `VARCHAR` 的元素指向一个非常长的字符串（在源码中由宏 `LONG_ADDR` 定义；
3. 通过 `REQUIRE(!data.getNext());` 检查此时该块在单链表中尚未有后继节点；
4. 调用 `REQUIRE(data.updateRecord(iov));` 来更新最后一条记录；
5. 借 `data.getNext()` 对应的块，检查其不为空，说明分裂出了新块；
6. 因为在前面插入时主键是递增的，因此假如记录被成功移动到新块，则被修改的新记录一定是新块中的最后一条记录（因为主键较小的记录都留在旧块中）；
7. 将 `record` 指向的地址置为新块最后一条记录对应的地址，随后调用 `getByIndex` 来检查与 `LONG_ADDR` 的值是否相等。

注意：为了检查是否与 `LONG_ADDR` 相等，需要新开一个数组来接受 `getByIndex` 的拷贝，因为在之前所用到的局部变量中，要么是被 `iov[2]` 元素指向的内存，则其值本来就是 `LONG_ADDR`；要么与 `LONG_ADDR` 的字节数不匹配。`iov` 及其指向的内存是本实验中比较容易出错的地方，如果不记清楚 `iov` 此时指向哪块内存的话便很可能导致逻辑错误。

**测试结果**

```
========== Build: 1 succeeded, 0 failed, 3 up-to-date, 0 skipped ==========
========== Build completed at 14:32 and took 02.805 seconds ==========
PS C:\Users\yu\Projects\db\build> .\bin\utest [p1]
Filters: [p1]
===============================================================================
All tests passed (72 assertions in 4 test cases)
```