MyJQL

FDU 2024 – Introduction to Database Systems – Project II

2024.5

Introduction

- MyJQL是一个类似redis的键值数据库，键值类型均为变长字符串；
- key上建立B-树索引，以提高访问效率；
- 具有get，set，del三种操作。

key	value
aa	adfasdfasdfasdfads
aab	sdfadfasdfasdfasdfasdf
abc	adfasdfsadfasdfafasdfasdf
bbbb	dfsadfasdfadf
cccccc	dfasdf
def	sdfasdfasdfasdfasdf

An Intuitive Practice

- 数据保存于文本文件，启动时全部读入，退出时全部写出。

key	value
aa	adfasdfasdfasdfads
aab	sdfadfasdfasdfasdfasdf
abc	adfasdfsadfasdfafasdfasdf
bbbb	dfsadfasdfadf
cccccc	dfasdf
def	sdfasdfasdfasdfasdf

xxx.txt

---

aa [sep] adfasdfasdfasdfads [sep]
aab [sep] sdfadfasdfasdfasdfasdf [sep]
abc [sep] adfasdfsadfasdfafasdfasdf [sep]
bbbb [sep] dfsadfasdfadf [sep]
cccccc [sep] dfasdf [sep]
def [sep] sdfasdfasdfasdfasdf [sep]

- 数据量比较大时，这样做显然不佳。

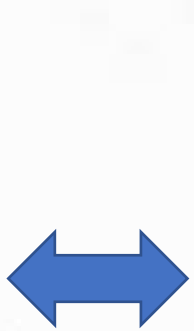
A Different Practice

- 数据量比较大时，这样做.....

Logical View (1)

- 分两张表，键值分别为指向另一张表中的字符串的指针。

key	value
aa	adfasdfasdfasdfads
aab	sdfadfasdfasdfasdfasdf
abc	adfasdfsadfasdfafasdfasdf
bbbb	dfsadfasdfadf
cccccc	dfasdf
def	sdfasdfasdfasdfasdf



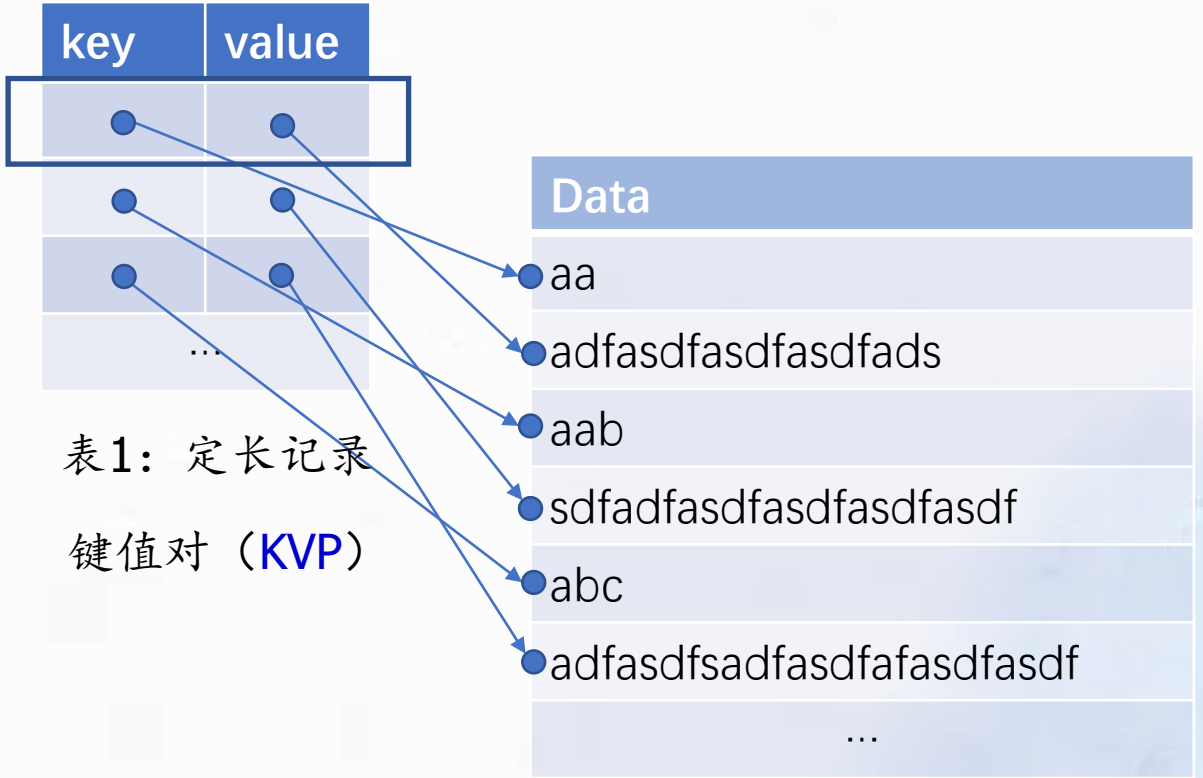


表1：定长记录

键值对（KVP）

表2’：变长记录

Logical View (2)

表2′可以进一步变化

- 长字符串切分成字符串块（Chunk），以链表形式保存。
- 字符串块最大不能超过一页的大小。

Data
aa
adfasdfasdfasdfads
aab
sdfadfasdfasdfasdfasdf
abc
…

表2′：变长记录

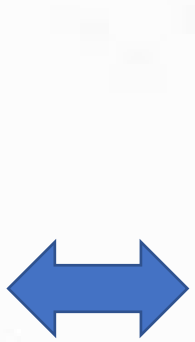
Data	Next
aa	
adfasdfasd	●
fasdfads	
aab	
sdfadfasdf	●
asdfasdfas	●
df	
abc	
…	

表2：变长记录

Logical View (3)

- 即：

key	value
aa	adfasdfasdfasdfads
aab	sdfadfasdfasdfasdfasdf
abc	adfasdfsadfasdfafasdfasdf
bbbb	dfsadfasdfadf
cccccc	dfasdf
def	sdfasdfasdfasdfasdf



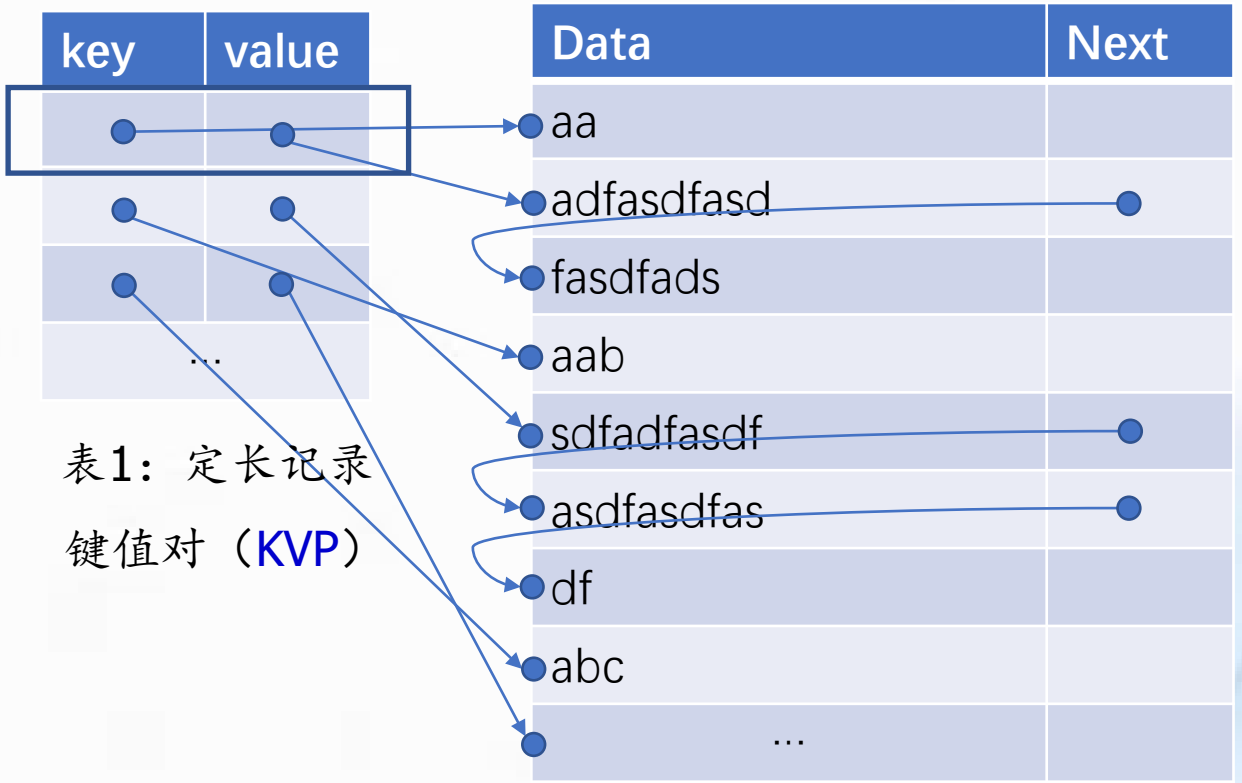
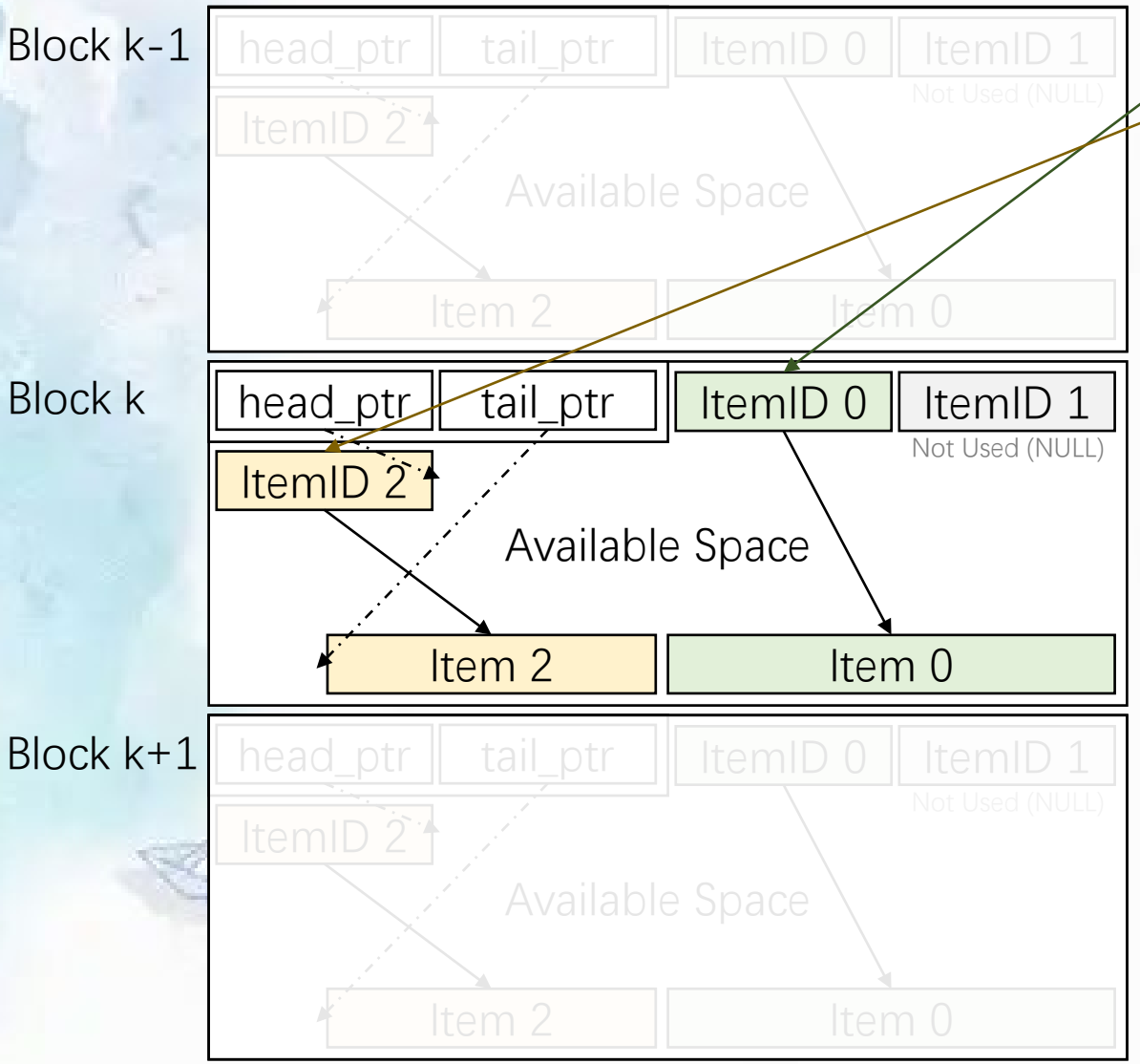


表1：定长记录

键值对（KVP）

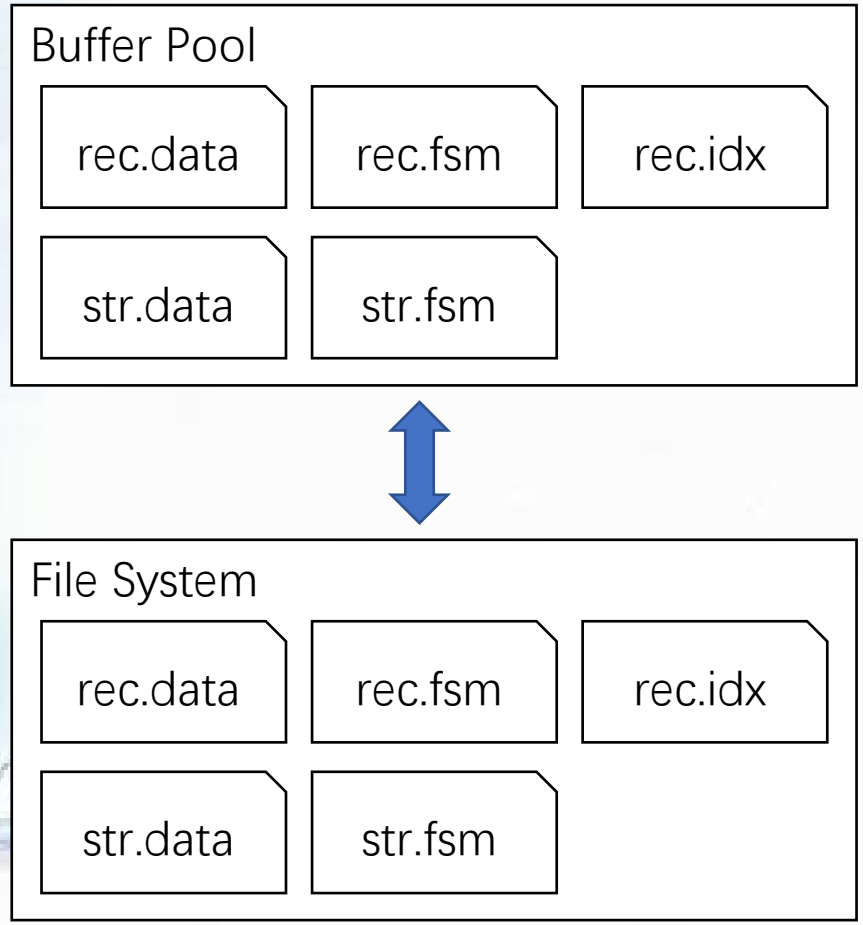
表2：变长记录

Physical View



- 一个数据文件（*.data，即存储数据的文件，而非索引等其他数据结构的文件）由若干个块（Block）构成。
- 每个块由头（Header，包含head_ptr和tail_ptr等字段），若干ItemID和Item构成。
- ItemID（定长）从头部开始分配，作为标识，指向尾部的Item（变长），Item保存具体数据（KVP或字符串块）。
- head_ptr至tail_ptr指向的空间为可用空间。

File & IO



- 所有文件IO操作通过缓冲池完成。
- 每个文件有独立的缓冲池。
- *.data：保存具体数据。
- *.fsm：维护空闲空间信息。
- *.idx：保存B-树索引。

Overview

- 文件列表：
  - file_io.h
  - buffer_pool.h
  - block.h
  - hash_map.h
  - table.h
  - str.h
  - b_tree.h
  - myjql.h

file_io.h

- file_io.c已给，若非必要，无须修改。
- 重要宏：
- PAGE_SIZE
  - 页大小，规定为128。
  - 为了方便调试，建议在引入B-树前使用64，引入B-树后使用128。
- 重要函数：
- FileIOResult read_page(Page *page, const FileInfo *file, off_t addr);
  - 读入位于addr的page。
- FileIOResult write_page(const Page *page, FileInfo *file, off_t addr);
  - 将page写出至addr，若addr==文件大小，则追加新的页。

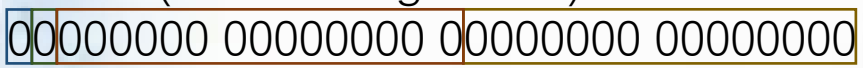
参考链接：MySQL的缓冲池管理

buffer_pool.h

- BufferPool数据结构请自行设计，缓冲池内需要缓冲一定量的页，可以固定缓冲池大小，也可以动态申请。
- void init_buffer_pool(const char *filename, BufferPool *pool);
  - 打开filename，并关联pool为其缓冲池。
- void close_buffer_pool(BufferPool *pool);
  - 关闭缓冲池，将缓冲的页写回文件。
  - 注意：无需考虑程序异常终止，测试保证程序一定会正常退出。
- Page *get_page(BufferPool *pool, off_t addr);
  - 获取地址为addr的页，并锁定（保证该页不会被意外换出）。
- void release(BufferPool *pool, off_t addr);
  - 释放地址为addr的页，该页之后可以被换出。
- 提示：文件读写调用file_io.h中的读写函数。

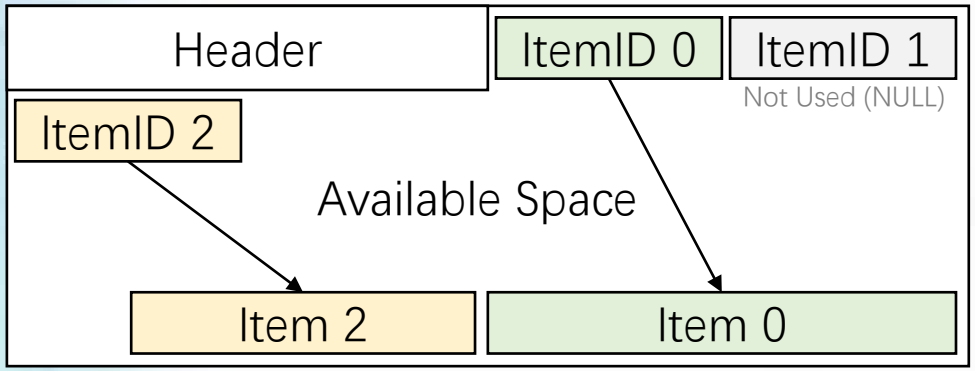
block.h

ItemID (32-bit unsigned int):



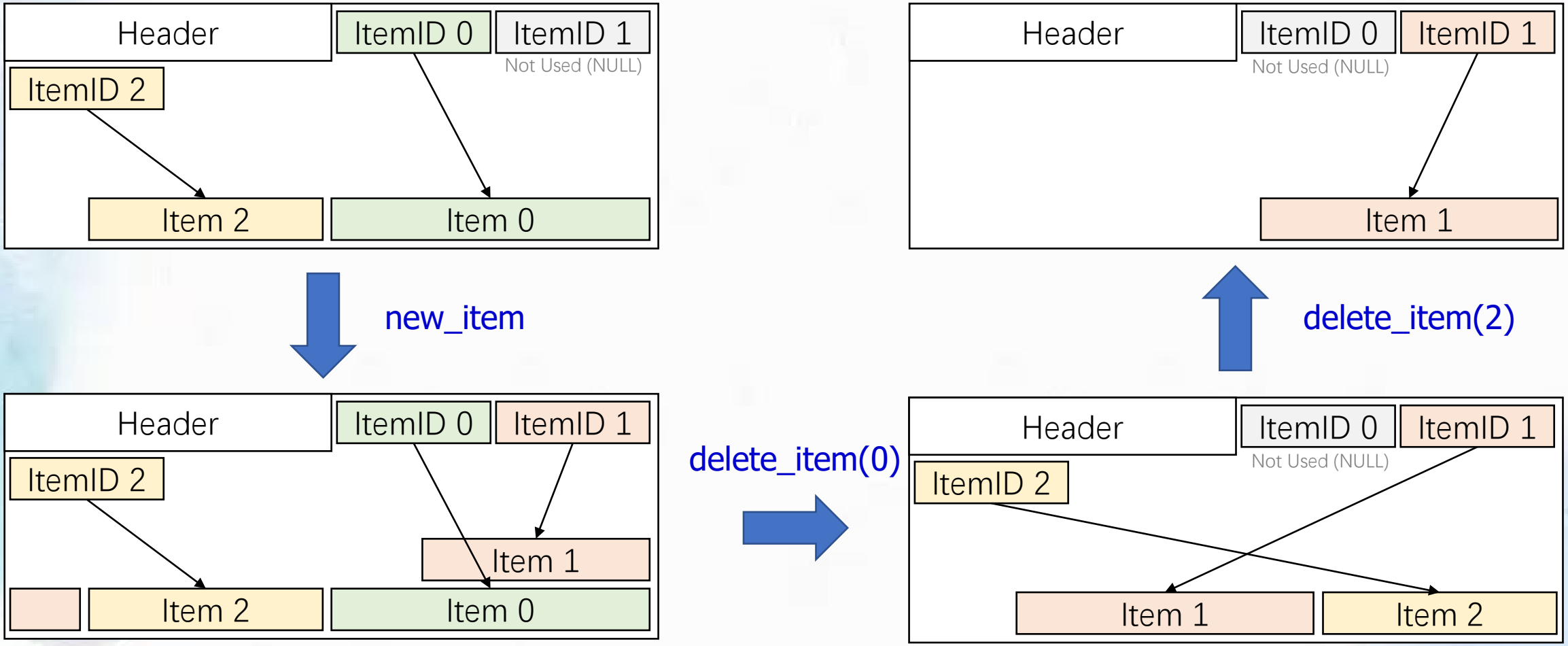
Unused bit　Offset: Item的起始地址　Size: Item的大小

Availability
0: 已使用，指向具体Item
1: 空闲可用，空指针



- ItemPtr get_item(Block *block, short idx);
  - 返回block第idx项Item的起始地址。
- short new_item(Block *block, ItemPtr item, short item_size);
  - 向block插入大小为item_size，起始地址为item的Item，返回插入后的项号（idx）。
- void delete_item(Block *block, short idx);
  - 删除block第idx项Item。

block.h

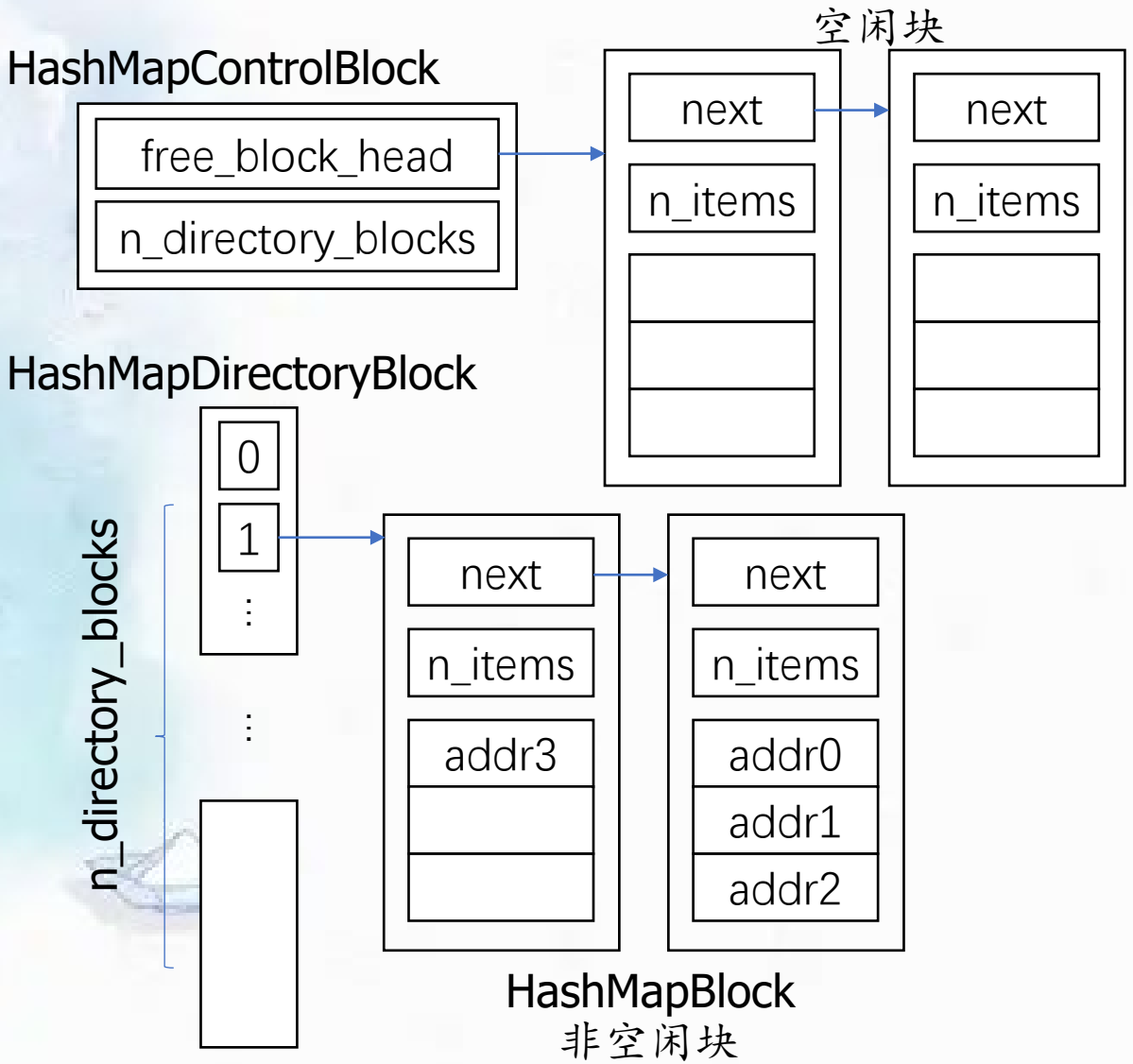

Header
ItemID 0
ItemID 1
Not Used (NULL)
ItemID 2
Item 2
Item 0
new_item
Header
ItemID 0
ItemID 1
Not Used (NULL)
Item 1
delete_item(2)
Header
ItemID 0
ItemID 1
ItemID 2
Item 1
Item 2
Item 0
delete_item(0)
Header
ItemID 0
ItemID 1
Not Used (NULL)
ItemID 2
Item 1
Item 2

hash_map.h

- 管理块的空闲空间。
- 重要函数：
- void hash_table_insert(BufferPool *pool, short size, off_t addr);
  - 标记地址为addr的块有size的空闲空间。
- off_t hash_table_pop_lower_bound(BufferPool *pool, short size);
  - 返回至少包含size空闲空间的块地址。
- void hash_table_pop(BufferPool *pool, short size, off_t addr);
  - 删除地址为addr的块的记录，该块有size的空闲空间。

参考链接：操作系统文件存储空间管理（存储空间的划分与初始化）

hash_map.h



- *.fsm文件由若干块构成。
- 第0块是控制块（HashMapControlBlock）。
- free_block_head块保存空闲块链表头指针。
- 第1~n_directory_blocks块是哈希表目录块（HashMapDirectoryBlock），构成哈希表目录，按空闲空间大小分类保存（*.data文件中）块的地址。
- 其余均为哈希表块（HashMapBlock），以链表形式组织，保存哈希表块链表头指针。

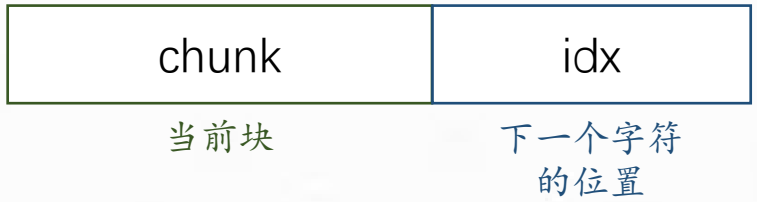
table.h

- 保存一个表需要有两个文件，数据文件（*.data）和空闲空间映射文件（Free Space Map，*.fsm，这里用哈希表实现）。
- 重要函数：
- void table_read(Table *table, RID rid, ItemPtr dest);
  - 根据rid，将数据读入dest，需要确保dest拥有适当的大小。
- RID table_insert(Table *table, ItemPtr src, short size);
  - 插入大小为size，起始地址为src的Item，返回rid。
- void table_delete(Table *table, RID rid);
  - 根据rid，删除相应Item。

str.h

StringChunk:
存储于文件

StringRecord:
存储于内存



- int has_next_char(StringRecord *record);
  - 判断record是否还有下一个字符。
- char next_char(Table *table, StringRecord *record);
  - 获取record的下一个字符。

- void read_string(Table *table, RID rid, StringRecord *record);
  - 根据rid，读取字符串至record。
- RID write_string(Table *table, const char *data, off_t size);
  - 将长度为size的字符串data写入table表，并返回rid。
- void delete_string(Table *table, RID rid);
  - 根据rid，删除对应字符串。

b_tree.h

- B-树索引存储于*.idx文件，其组织方式与哈希表文件（*.fsm）类似。
- 第0块为控制块，包含根节点地址与空闲块链表头指针。
- 其余块为B-树节点块。
- 重要函数：
- RID b_tree_search(BufferPool *pool, void *key, size_t size, b_tree_ptr_row_cmp_t cmp); /* 搜索 */
- RID b_tree_insert(BufferPool *pool, RID rid, b_tree_row_row_cmp_t cmp); /* 插入 */
- void b_tree_delete(BufferPool *pool, RID rid, b_tree_row_row_cmp_t cmp); /* 删除 */

参考链接：B-树、B+树及其持久化

myjql.h

- size_t myjql_get(const char *key, size_t key_len, char *value, size_t max_size); /* get */
- void myjql_set(const char *key, size_t key_len, const char *value, size_t value_len); /* set */
- void myjql_del(const char *key, size_t key_len); /* del */

Grading

- 满分：10分
- 4组测试~~（每组测试2分，共8分）~~
  - 前三组测试代码公开，最后一组测试代码非公开
  - 1 test_hash_map 2分
  - 2 test_str 2分
  - 3 test_b_tree 3分
  - 4 [?] 3分
- 实验报告：对缓冲池、块管理等组件进行有效改进（2分附加分，加到满分10分为止）
  - 有效改进是指：与原始实现相比，改进后的实现，在随机数据/某些极端情况下的性能有显著提升（平均耗时/空间占用情况）

Q. & A.