Database System 2020-2
Final Report

Class Code (ITE2038-11801)
2018007356
강 응 찬

Table of Contents

전체 layered architecture 는 다음과 같이 구성된다.

- ❖ Initialization part(Init_db/init_table/open_table)
- ❖ Index layer(db_insert, db_find, db_update, db_find, db_delete..)
- ❖ Buffer layer(buffer_read, buffer_write, close_table, shutdown_db..)
- ❖ Disk space manage layer()

- ❖ initialization part(Init_db/init_table/open_table)
  - ➢ int_db()
    - ▪ 사용자로부터 buffer 의 사이즈를 입력 받아 frame 의 개수를 설정한다.
      - •
```
frame = malloc(sizeof(struct buffer_ctr) * num_buf);
```
    - ▪ buffer 동작에 필요한 값들을 초기화 해주는 작업을 진행한다.
      - •
```
for(int i = 0; i < buffer_num; i++){
    frame[i].frame_id = i;
    frame[i].is_dirty = false;
    frame[i].pin_count = 0;
    frame[i].pagenum = 1;    // header is 0  , to avoid this
    frame[i].table_id = 0;
    frame[i].data = malloc(sizeof(char) * PAGE_SIZE);
    for(int j =0; j < PAGE_SIZE; j++)
    {frame[i].data[j] = '\0';}
    frame[i].next = NULL;
    pin_queue[i] = -1;   // frame id is from 0,
}
```
  - ➢ init_table()
    - ▪ 새 파일이 생성될 때, header 와 root, free 영역을 미리 정한다.
      - • header 는 파일의 0 ~ 4096
      - • root 는 파일의 4097 ~ 8192
      - • free 는 그 다음 영역의 전부로 한다. 즉 free page 를 요구하면 root page 바로 다음부터 4096byte 만큼씩 올려준다.
  - ➢ open_table()
    - ▪ Table 을 open 하기 전에 먼저. file descripter 와 pathname 혹은 table name 을 저장하기 위한 array type 의 변수를 전역변수로 정한다.
  - ➢ 사용자로부터 파일 이름을 임력받으면 open()함수를 호출하여 존재하는 파일인지 새 파일인지를 체크하고
    - ▪ 새 파일일 경우 미리 만들어 놓은 table_name[]전역변수에 pathname 을 저장하고, file_descripter[]에도 fd 를 저장하고나서 table_id 를 리턴한다.
    - ▪ 기존 파일일 경우 받은 fd 를 가지고 table_name[]변수를 호출하여 fd 와 pathname 을 비교하면서 맞는 값이 나올 때 table_id 를 리턴한다.

Index layer 는 buffer layer 의 함수들을 호출하면서 데이터를 buffer 에 저장하고 실행한다..

❖ Index layer(db_insert, db_find, db_delete..)

- db_insert(int table_id, pagenum_t key, char* value)
  - mapping_header()라는 함수를 호출하여 해당 파일의 헤더페이지 주소를 리턴받는다.
  - get_frame_id()라는 함수를 호출하여 해당 파일의 해당 pagenum 이 buffer 에 있는 경우 그 frame id 를 리턴하고 없을 경우 빈 frame 을 리턴하여 거기에 header page 를 담는다.
  - header page 에서 root page 주소를 얻어 다시 위의 과정을 반복해서 root page 를 buffer 로 올린다.
  - root page 에서부터 내려가면서 leaf_page 가 나올 때까지 위 과정을 반복하고 나서 find_leaf_in_buffer()함수를 호출하여 해당 key 가 있는 위치를 리턴한다.
  - 이 과정에 pin_count 를 계속 올려준다.
  - buffer_write()함수를 호출하여 해당 key 에 value 를 입력하고 그 page 를 dirty 표시를 한 후 key 값을 증가시키고 pin_count 를 낮춘다.
- db_find(int table_id, pagenum_t key, char* ret_val)
  - insert_db()에서 했던 과정을 반복하여 leaf-page 의 key 가 있는 위치를 찾는다.
  - buffer_read()함수를 호출하여 그 key 에 있는 value 를 읽어온다.
- db_delete(int table_id, pagenum_t key)
  - leaf-page 를 찾을 때까지 과정을 반복하고 delete_in_frame()함수를 호출하여 해당 key 가 가리키는 value 를 삭제한다.
  - dirty 표시를 하고 num_key 를 감소시킨다.
  - num_key 를 체크하고 0 이면 merge_page()함수를 호출한다.

buffer layer 는 index layer 와 disk space manager 와 모두 communication 을 한다.

❖ buffer layer(buffer_read, buffer_write, close_table, shutdown_db..)

- buffer 에서 value 를 읽어오는 작업은 buffer_read()함수, value 를 쓰는 작업은 buffer_write()을 쓴다.
- Index 에서 요구한 page 가 buffer 에 없는 경우는 frame_read_disk()함수에 해당 page 를 넘겨서 pread()함수를 이용하여 직접 disk 에서 읽어오고 해당 page 를 담은 frame id 를 enqueue 한다.(frame 을 넣는 것이 아니라 frame number 를 enqueue)
- close_table()을 진행할 경우 queue 에 있는 모든 frame number 를 하나씩 꺼내서(dequeue) 해당 frame 이 담고 있는 page 중에 dirtycheck 가 되었 있는 page 를 file_write_disk()함수를 이용하여 disk 에 저장한다.

- Disk 에 저장하기 전에 num_key 를 체크하여 merge 나 split 이 필요한 상황이라면 해당 함수를 호출한다.

➢ shutdown_db()는 table_id 를 담고 있는 전역변수를 decrease 하면서 0 이 될 때까지 close_table 을 반복한다.

❖ disk space manage layer()

➢ file_alloc_page(void)

- 디스크에서 빈 페이지를 찾아서 리턴해주는 역할을 하는 함수다.
- 빈페이지 주소는 header page 의 free_page 영역에 있고, 그 page number 를 리턴해준다.
- 그리고 그 page 바로 다음 page number 를 다시 header page 에 저장한다.

➢ file_write_page()

- key 를 string type 으로 변경시킨 후, pwrite()함수를 호출하여 해당 page 에 key <value>형태로 저장한다.
- 저장한 후 sort_leaf_page()함수를 호출하여 sorting 을 진행한다.
- num_key 를 증가시키고 num_key 가 leaf_order 보다 큰 지를 체크
- 큰 경우 insert_parent()함수를 호출하여 split 을 진행한다.

➢ file_read_page()

- search_key()함수를 호출한다.
- 위 함수는 해당 page 를 binary search 하여 key 의 위치를 offset 으로 넘긴다.
- 그럼 그 page 에 offset 위치를 찾아서 pread()함수를 용하여 값을 읽어온다.

➢ Insert_parent()

- split 이 필요한 경우 호출하는 함수다.
- 일단 free_page 를 할당 받은 후, split_page()함수를 호출하여 old page 의 데이터의 절반을 새로 할당받은 free_page 로 넘기고 old page 의 sibling 에 new_page 의 주소를 넣는다..
- 만일 split 이 필요한 leaf page 가 parent 가 없는 경우, insert_new_paren()함수를 호출하여 새로운 parent 를 만든다.
- parent 가 이미 존재하는 경우, new page 의 leftmost record id 를 parent 로 copy 해서 올려 보낸 후 sorting 을 진행한다.
- parent 내의 num_key 를 체크하여 split 이 필요한 상황인지 체크하고 만일 필요하면 또 insert_paretn()를 호출한다.
- split 이 필요하지 않을 때까지 recursive 로 insert_parent()함수를 호출한다.

➢ merge_page()

- 일단 parent page 가 있는 경우 parent 를 file_free_page()함수를 호출하여 제거한다.
- 해당 page 의 sibling page 에서 데이터를 전부 읽어서 왼쪽 page 로 넘기는 작업을 sibling page 가 더는 없을 때까지 오른쪽으로 한 page 씩 넘기면서 실행한다.

```
while(atoi(sibling)){  // has sibling and has also parent
    page_t sib_page = mapping_page(atoll(sibling));
    pread(fd, data_buffer, sizeof(data_buffer), sib_page.pagenum + sib_page.DATA);
    pwrite(fd, data_buffer, sizeof(data_buffer), merge.pagenum + merge.DATA);
    pread(fd, sibling, sizeof(sibling), (sib_page.pagenum + sib_page.right_sibling));
}
```

➢ 그 외에 key 를 가지고 page number 를 찾는 get_page_number(), internal_page 에만 쓰기를 진행하는 write_internal_page(), root page 부터 search 하여 leaf page 를 찾아서 return 하는 find_leaf_page()함수가 있다.

❖ Concurrency Control Implementation

Concurrency control 은 index_layer 의 db_find()와 db_update()에서 구현되었다.

➢ db_find() & db_update()

- 첫 순서로 lock_mode 를 0 혹은 1 로 한다.
- Lock mode 가 0 이라는 것은 share_mode, 1 은 exclusive mode.
- 다음 buffer 를 잠근다.

```
pthread_mutex_lock(&buffer_mutex);    // lock the buffer
```

- buffer 에서 해당 페이지를 찾은 다음 buffer 를 unlock 하고 다시 그 page 를 잠근다.

```
pthread_mutex_unlock(&buffer_mutex);    // unlock the buffer
pthread_mutex_lock(&page_mutex[index]);   // lock the page in buffer
```

- lock acquire 함수를 호출하여 해당 record 에 대해 lock 을 걸고 page lock 은 푼다.
- 이때 table_id, key, trx_id, lock_mode 를 lock_acquire 함수로 넘겨준다.
- Hash 함수를 이용해서 인자로 받은 table 의 key 를 가지고 있는 header 를 받는다.
- lock_acquire 함수는 두 개의 sub 함수(share_mode, exclusive_mode)를 가지고 있다. Lock_mode 를 체크하고 1 이면 exclusive_mode 함수를 호출, 0 이면 share_mode 함수를 호출한다.
- share_mode 함수는 모든 접근을 허용하므로 함수 안에서 따로 mutex 를 호출하지 않는다.
- exclusive_mode 함수는 해당 record 에 대한 mutex 를 호출하고 작업을 시작한다. `pthread_mutex_lock(&lock_table_latch[table_id][key]);`
- header 가 가리키는 linked_list 에 하나의 trx 라도 있으면 대기하게 된다.

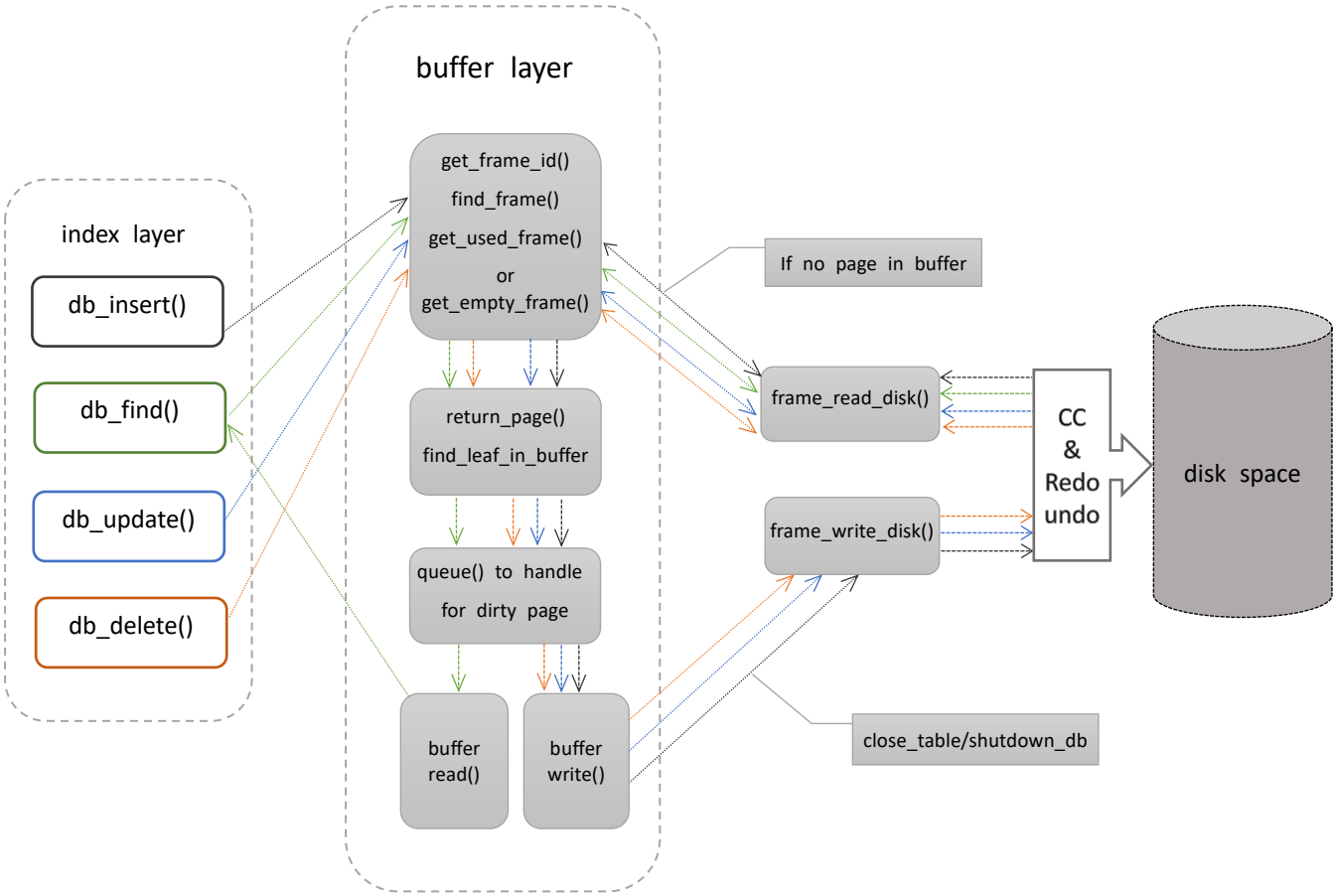
```
pthread_cond_wait(&cond[table_id][key], &lock_table_latch[table_id][key]);
```

- db_find()는 항상 share_mode 함수만 호출하고 db_update()는 항상 exclusive_mode 함수만 호출한다.
- trx_commit 요청이 들어올 경우 lock_release 함수를 호출하여 자원을 회수하고 list 에서 버린다.
- lock_release 에서는 먼저 들어온 object 가 가지고 있는 sentinel 영역에서 header 의 주소를 추출한다. `lock_t * header = lock_obj->sentinel;`
- header 를 이용하여 해당 object 의 mode 를 알아내고 cond_signal 을 보낼지 말지를 결정한다.
- 이전 trx 와 다음 trx 를 이어주고 인자로 들어온 trx 는 지운다.

```
lock_obj->prev = lock_obj->next;
lock_obj->next->prev = lock_obj->prev;
```

❖ Crash-Recovery Implementation & In-depth Analysis

crash-recovery 는 디자인은 짜 놓았지만 구현과정에서 디버깅을 끝내지 못해 완성에 실패하였습니다. 제가 시도했던 디자인은 log-buffer 에 기록되는 순서와 그 데이터를 stable log 로 옮기는 순서를 잘 지키기 위해 queue 를 이용하였습니다. 즉 log buffer 에 logging 할 경우 log_enqueue 로 진행되고 그것이 stable log 로 빠져나갈 때는 log_dequeue 를 사용하였습니다. 그러나 완성하지 못했고 따라서 저의 code 를 바탕으로 crash-recovery 와 depth analysis 를 설명할 수 있는 기회를 가지지 못했습니다. 아래에는 overall layered architecture 와 layer 간 관계 그리고 제가 구현하려고 했던 crash recovery 의 diagram 입니다.



Overall layered architecture & crash recovery diagram