WIKI

이름: 문준영

학과: 컴퓨터소프트웨어학부

학번: 2022006135

Design

명세에서 요구하는 것은 Join을 수행할 시 "시간 제약"이 있고 "메모리 제한"이 있다는 것이다. 이를 충족하기 위해 배웠던 알고리즘 중 무엇을 사용하여 Join을 구현할 지 고민하였다. 먼저 Nested-loop-join이다. 이 알고리즘은 너무 brute force한 알고리즘이기 때문에 제한 시간을 맞추치 못할 것 같아 포기하였다. 또한, 만들어진 .db파일은 모두 B+ tree 구조로 되어 있다. 이말인 즉슨, 메모리상에서 연속적으로 분포하는 것이 아닌 data 자체가 indexing 되어 있는, 즉, B+ tree file structure를 이루고 있다. 때문에 indexing을 활용하는 알고리즘을 생각해 봤지만 이 역시 그다지 효율적이라 느끼지 못했다.

그러던 중 가장 주의깊게 보게 된 점은 **leaf node에서의 정렬**이다. Leaf node가 정렬되어있기 때문에 leaf node를 순차적으로 훑으면서 가장 빠른 비교 알고리즘을 사용하면 좋을 것이라고 생각하였다. 때문에 Merge Join을 사용하도록 구현할 것이다.

고려사항 중 한 가지 더 있는 것은 메모리 사용량이다. 메모리 사용량이 1GB로 한정되어 있기 때문에 이를 최대한 활용하여야 할 것이다. 구현된 B+ tree에서 한 node의 크기는 4096 byte, 즉, 4KB이다. 이를 1GB를 이로 나누어보면 대략 26만이 나온다. 때문에 한 테이블에서 최대 26만의 node(block)이 올라갈 수 있는 것이다. 때문에 한 테이블 당 13만개의 block을 최대로 옮겨 seek time을 최소화 하려고 한다.

마지막으로 고려해야 될 것은 원래의 B+ tree 구현된 코드를 두 개의 파일을 열 수 있도록 변경해야 한다. 이를 위해서는 두 개의 hp가 필요하다. 그리고 모든 operation에 table_id를 주어 어떤 테이블에 대해서 해당 operation을 수행하는 지를 명시해 줘야 한다.

위를 토대로 구현한 결과를 보도록 하자.

Implementation

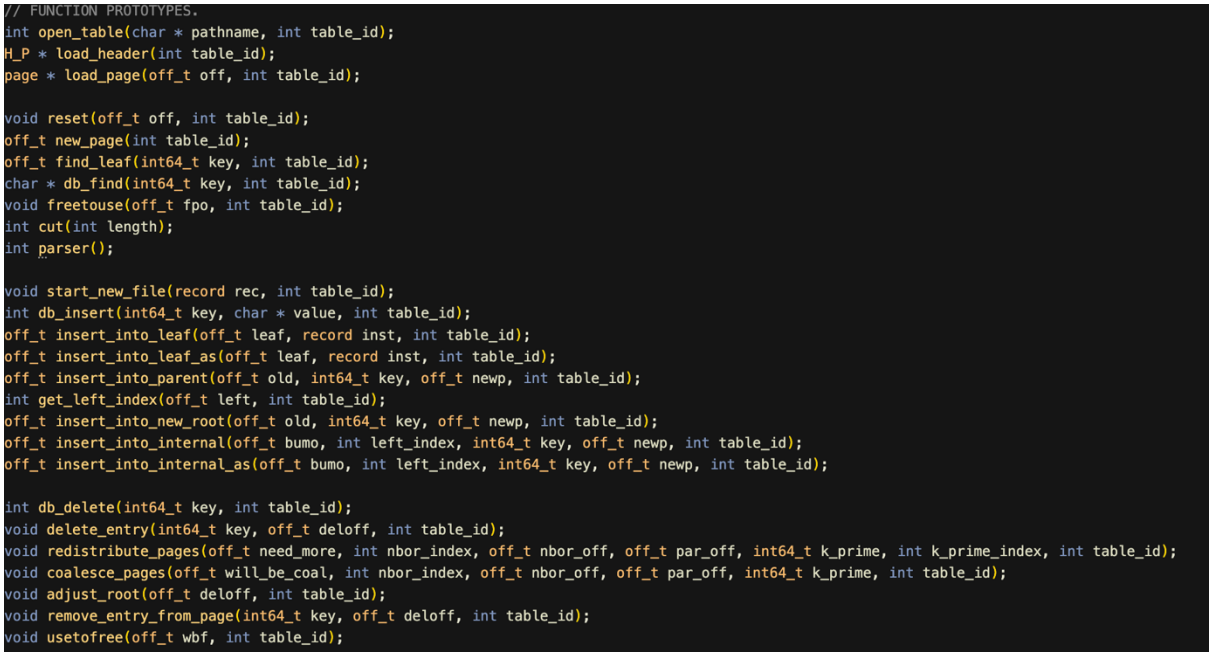
먼저, 두 개의 file/table을 여는 과정을 위해 다음과 같이 두 개의 header page, root page, file descriptor를 만들었다.

```
H_P * hp[2];

page * rt[2] = {NULL, NULL}; //root is declared as global

int fd[2] = {-1, -1}; //fd is declared as global
```

또한, 모든 함수에 대해서 인지로 int table_id를 주어 어떤 테이블에 대해서 해당 operation을 수행하는시 명시되도록 하였다.



```
// FUNCTION PROTOTYPES.
int open_table(char * pathname, int table_id);
H_P * load_header(int table_id);
page * load_page(off_t off, int table_id);

void reset(off_t off, int table_id);
off_t new_page(int table_id);
off_t find_leaf(int64_t key, int table_id);
char * db_find(int64_t key, int table_id);
void freetouse(off_t fpo, int table_id);
int cut(int length);
int parser();

void start_new_file(record rec, int table_id);
int db_insert(int64_t key, char * value, int table_id);
off_t insert_into_leaf(off_t leaf, record inst, int table_id);
off_t insert_into_leaf_as(off_t leaf, record inst, int table_id);
off_t insert_into_parent(off_t old, int64_t key, off_t newp, int table_id);
int get_left_index(off_t left, int table_id);
off_t insert_into_new_root(off_t old, int64_t key, off_t newp, int table_id);
off_t insert_into_internal(off_t bumo, int left_index, int64_t key, off_t newp, int table_id);
off_t insert_into_internal_as(off_t bumo, int left_index, int64_t key, off_t newp, int table_id);

int db_delete(int64_t key, int table_id);
void delete_entry(int64_t key, off_t deloff, int table_id);
void redistribute_pages(off_t need_more, int nbor_index, off_t nbor_off, off_t par_off, int64_t k_prime, int k_prime_index, int table_id);
void coalesce_pages(off_t will_be_coal, int nbor_index, off_t nbor_off, off_t par_off, int64_t k_prime, int table_id);
void adjust_root(off_t deloff, int table_id);
void remove_entry_from_page(int64_t key, off_t deloff, int table_id);
void usetofree(off_t wbf, int table_id);
```

이로써 두 개의 파일을 열 수 있는 환경을 마련하였다.

이제 본격적으로 Join의 구현과정을 보도록 하자.

처음에 고민한 바는 **어떻게 해야 최대로 많은 Block을 가져올 수 있을까** 였다. 각 테이블당 대략 13만개의 block을 가져오도록 하고 싶은 것이다. 하지만 이내 이가 불가능 하다는 것을 깨달았다. 그 이유인 즉슨, **각 테이블의 구조가 B+ tree로 이루어져 있기 때문에 leaf node끼리 Disk 상에서 서로 연속적으로 위치하고 있지 않을 가능성이 매우 크기 때문이다.** 때문에 연속적으로 데이터를 읽기는 포기하고 merge join을 그냥 구현하기

로 하였다.

```
void db_join(){
    //각 table에서 첫 leaf node 찾기
    off_t leaf_offset_0 = find_leaf(MIN_KEY, 0);
    off_t leaf_offset_1 = find_leaf(MIN_KEY, 1);

    page * page_0 = NULL;
    page * page_1 = NULL;

    int i_0 = 0;
    int i_1 = 0;
```

먼저 기본적으로 각 테이블에서 가장 왼쪽 leaf node를 찾는다. (MIN_KEY 값은 int64의 최솟값으로 설정하였다.)

```
    //merge join
    while(leaf_offset_0 != 0 && leaf_offset_1 != 0){
        //페이지 끝까지 돌았을 경우 다음 leaf node load
        if(page_0 == NULL){
            page_0 = load_page(leaf_offset_0, 0);
            i_0 = 0;
        }
        if(page_1 == NULL){
            page_1 = load_page(leaf_offset_1, 1);
            i_1 = 0;
        }

        int64_t key_0 = page_0->records[i_0].key;
        int64_t key_1 = page_1->records[i_1].key;
```

```
if(key_0 < key_1){
    //table_0의 key가 더 작으면 다음 키로 이동
    i_0++;
    //index최대치를 넘어가면
    if(i_0 >= page_0->num_of_keys){
        leaf_offset_0 = page_0->next_offset;
        free(page_0);
        page_0 = NULL;
    }
}
else if(key_0 > key_1){
    //table_1의 key가 더 작으면 다음 키로 이동
    i_1++;
    if(i_1 >= page_1->num_of_keys){
        leaf_offset_1 = page_1->next_offset;
        free(page_1);
        page_1 = NULL;
    }
}
else{
    //key가 서로 같은 경우
    char * value_0 = page_0->records[i_0].value;
    char * value_1 = page_1->records[i_1].value;
    printf("%ld,%s,%s\n", key_0, value_0, value_1);
    i_0++;
    i_1++;
    if(i_0 >= page_0->num_of_keys){
        leaf_offset_0 = page_0->next_offset;
        free(page_0);
        page_0 = NULL;
    }
    if(i_1 >= page_1->num_of_keys){
        leaf_offset_1 = page_1->next_offset;
        free(page_1);
        page_1 = NULL;
    }
```

본격적으로 merge join 을 수행한다. 각 테이블의 leaf node를 필요시 load 하고 각 key을 비교한다.

Key 값을 비교해 일치하지 않으면 더 큰 key값을 가진 table의 index를 증가시켜 준다. 만약 index가 해당 node의 범위를 초과한 경우 다음 leaf node로 이동하도록 한다.

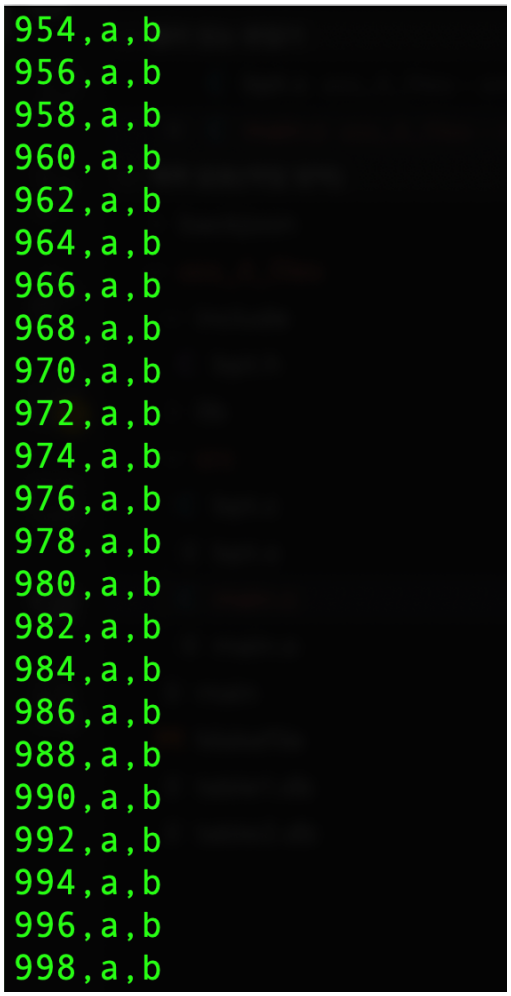
```
}
//남은 page가 있다면 free 해주기
if(page_0 != NULL){
    free(page_0);
}
if(page_1 != NULL){
    free(page_1);
}
```

마지막으로 while문을 나온 후 마지막 leaf node를 free해주면서 merge Join을 마친다.

Result

```
for(int i = 1; i < 1000; i++){
    db_insert(i, "a", 0);
}
for(int i = 1; i < 1000; i++){
    db_insert(2*i, "b", 1);
}
```

이 테스트 케이스로 결과를 보여주도록 하겠다. Table1에는 1부터 999까지의 key가 "a"의 value를 가지면서 삽입된 상태일 것이고, table2 에는 2부터 1998까지의 모든 짝수가 key로 들어갈 것이고 "b"라는 value가 이와 함께 들어갈 것이다. Join의 결과물은 1부터 999사이의 모든 짝수들일 것이다.



결과는 예상한 바와 같이 나왔다.

Trouble Shooting

본 과제에서 가장 고민을 많이 했던 부분은 앞서 말했던 바와 같이 여러 leaf node를 연속적으로 memory에 load 할 수 있는지 였다. 하지만 안타깝게도 연속된 디스크 공간에 Data가 저장된 것이 아니기 때문에 불가능 하다는 것을 깨달았다. 때문에 각 leaf node를 필요할 때 마다 load하여 meerge join을 수행하였다.