DB Assignment 4 Wiki

2020081958 송재휘

1. Design

이번 과제에서 주어진 Natural Join 기능을 구현하기 위해 저는 join 알고리즘을 정해야 했습니다. 저희가 구현한 B+Tree의 특성 상 search key인 key 값에 중복이 없고, 이미 sorting 되어 있으며, data 값 또한 노드 안에서 순차적으로 저장되어 있는 구조이기 때문에 (별도의 leaf node의 경우는 모름) 따로 sorting overhead가 필요하지 않고, join의 block transfer이 가장 적은 merge join을 활용해 Natural join을 구현해기로 결정했습니다. Merge Join은 수업시간에 배운 이론에 의하면 한번에 여러 페이지(=블록 크기)를 읽어 올수록 block seek이 줄어들기 때문에, 저 또한 과제 명세에 주어진 1GB의 메모리 제한을 침범하지 않는 선에서 최대한 많은 페이지를 한번에 가져올 수 있도록 디자인했습니다. 1GB 중 424MB는 join의 연산을 비롯해 각 테이블의 첫번째 leaf_offset을 찾고, 헤더 페이지, fd, rt 등을 관리하기 위한 메모리로 빼두었고, 600MB를 테이블의 페이지를 사전에 올려두기 위한 메모리 용량으로 설정했습니다. 하지만, 실제 테스트 환경에서의 변수를 고려해 safe margin의 개념으로 600MB에 0.85를 곱해 약 510MB을 페이지를 사전에 올려두기 위한 메모리 용량으로 최종 설정했습니다. 이를 4KB의 leaf page 크기로 나눈 결과 약 130,560 page를 한번에 올릴 수 있었습니다. 물론 join에 참여하는 table이 2개이기 때문에 table 당 65,280 개의 leaf page가 한번에 로드될 수 있는 제한을 설정했습니다. 하지만, 구현 과정에서 실제 B+Tree의 leaf page의 사용량을 고려했을 때 각 테이블 당 65,280개의 leaf page를 올리더라도, B+Tree는 31개 레코드 field 중 절반인 16개 만을 보장하기 때문에, 최악의 경우 65280 개의 페이지를 로드하기 위한 메모리 공간으로 사용하더라도 실제로는 그 반절인 32640개의 페이지의 성능에 미치지 못할 수 있겠다는 결론에 도달했습니다. 그로 인해 저는 해당 메모리 사용의 효율성을 극대화하고, 성능을 최대한 올리고자 page 단위 대신 record 단위의 저장 방식을 사용했습니다. 기존에 계산한 각 테이블 당 올릴 수 있는 leaf page 개수인 65280에 각 페이지가 꽉 차 있을 때를 가정한 record의 개수인 31을 곱해 2,023,680을 한번에 메모리에 올려 처리할 전체 record의 수로 결정했습니다. 이렇게 함으로써 메모리를 더욱 효율적으로 사용할 수 있도록 했습니다. 또한 이번 과제의 경우 두 테이블을 열 수 있어야 하기 때문에 두 테이블에 대한 insert, delete, find, join 기능을 지원하기 위해 s라는 switch 기능을 main.c에 추가로 구현했습니다. S를 입력하고, 바꾸고자 하는 table의 번호를 누르면 fd, hp, rt가 해당 테이블의 정보로 변경 돼, 테이블 두 개에 대한 확장을 가능하도록 설계했습니다. 실제 merge join의 구현 과정은 Implement 파트에서 설명하겠습니다.

2. Implement

이번 과제를 구현한 순서대로 설명하겠습니다.

1) main.c에서 두개의 table을 열도록 수정 (각각에 대한 insert, delete, find, join이 가능하도록)

main.c에서 두 개의 table.db 파일을 지원하기 위해선 기존 B+Tree 프로그램에서 전역 변수의 형태로 사용하는 fd, hp, rt를 동적으로 변경해줘야 했습니다. 이를 위해 bpt.c의 open_table함수에서 기존에 성공했을 때 0을 리턴하고, 실패했을 때 -1을 리턴하던 것을 실패했을 땐 동일하게 -1을 리턴하고, 성공했을 땐 할당된 fd 값을 리턴하도록 수정했습니다.

```
if (fd > 0) {
       //printf("New File created\n");
       hp->fpo = 0;
       hp->num_of_pages = 1;
       hp->rpo = 0;
       pwrite(fd, hp, sizeof(H_P), 0);
       free(hp);
       hp = load_header(0);
       return fd;
   }
   fd = open(pathname, O_RDWR|O_SYNC);
   if (fd > 0) {
       //printf("Read Existed File\n");
       if (sizeof(H_P) > pread(fd, hp, sizeof(H_P), 0)) {
           return -1;
       }
       off_t r_o = hp->rpo;
       rt = load_page(r_o);
       return fd;
   }
```

이후, main.c에서 각각 테이블을 open_table을 사용해 열 때, table1_fd, table2_fd라는 변수에 할당된 fd 값을 저장하도록 했습니다.

```
int table1_fd = open_table("table1.db");
int table2_fd = open_table("table2.db");
```

또한 기존 insert, delete find를 지원하기 위해 s라는 switch case를 추가해 각 기능을 적용할 table을 선택할 수 있도록 가능을 추가했습니다.

```
case 's':
            print_current_file(table1_fd, table2_fd);
            printf("Enter 1 or 2 to switch table (1:table1.db, 2:table2.db) : ");
            scanf("%d", &table_num);
            switch_table(table_num, table1_fd, table2_fd);
            break;
```

이를 통해, s를 입력하게 되면 현재의 table정보가 나오고 바꾸고자 하는 테이블의 변호를 입력해 insert, delete, find 등의 기능을 수행하고자 하는 table의 fd, hp, rt값을 로드할 수 있습

니다. 위의 print_current_file함수는 현재 참조 중인 table의 정보를 출력하도록 bpt.c에 만들었습니다.

```
void print_current_file(int table1_fd, int table2_fd) {
    int table_no = (fd == table1_fd)? 1 : 2;
    printf("Curent Table Number : %d\n", table_no);
}
```

또한, switch_table 함수가 바로 함수의 전역 변수를 새로운 테이블에 맞게 수정해주는 함수로, 전달된 인자의 table_num(1 또는 2)에 해당 하는 값으로 fd, hp, rt 정보를 재설정합니다. 이때 기존 hp, rt가 존재한 경우 free를 통해 메모리 할당을 해제합니다.

```
void switch_table(int table_num, int table1_fd, int table2_fd) {
    if (table_num == 1) {
        fd = table1_fd;
        free(hp);
        hp = load_header(0);
        if (rt != NULL) {
            free(rt);
            rt = NULL;
        }
        if (hp->rpo != 0) {
            rt = load_page(hp->rpo);
        }
    }
    else if (table_num == 2) {
        fd = table2_fd;
        free(hp);
        hp = load_header(0);
        if (rt != NULL) {
            free(rt);
            rt = NULL;
        }
        if (hp->rpo != 0) {
            rt = load_page(hp->rpo);
        }
    }
    else {
        printf("Inserted non-exist table number! Try again.\n ");
    }
}
```

이렇게 하고 나면 s 기능을 통해 동적으로 수정할 테이블을 설정할 수 있게 되고, insert, delete, find를 원할 시 s를 통해 target table을 설정하고 수행할 수 있습니다. 이후 join의 경우는 명세에 주어진 코드와 동일하지만, 함수 내에서 각 테이블의 fd, hp, rt로 전환하는 데 사용하기 위해 인자로 table1_fd와 table2_fd를 함께 넘겼습니다.

```
case 'j':
            db_join(table1_fd, table2_fd);
            break;
```

2) db_join 구현

명세에 정의된 방식으로 1GB의 메모리 할당량을 넘지 않으면서 join의 실행시간을 최대한 줄이는 DB join을 구현하기 위해 이론 시간에 배운 Merge join의 방식으로 natural join을 구현했습니다. 이때 위키의 design 파트에서 설명한대로, block seek의 가능성을 최대한 줄이기 위해 600MB에 safety margin 0.85를 곱해 정한 510MB에 해당하는 130,560 개의 page를 memory에 저장해두도록 Limit을 설정했습니다. 이렇게 정한 이유는 해당 프로그램을 돌리는 과정에서 추가로 load해야 하는 page가 발생할 수 있고, 또 추가로 어떤 비용이 들지 몰라 위와 같이 설정했습니다. 이렇게 되면 각 테이블 당 65,280개의 페이지가 메모리에 올라갈 수 있게 되는데, 원래 이렇게 페이지 단위로 구현을 하던 중 leaf page의 특성 상 record의 사용량에 대한 제한이 반 밖에 존재하지 않는데 이렇게 되면 65,280개의 페이지를 올려도 이론 상 최악의 경우 32,640개의 페이지를 올리는 효과가 날 것이라 판단되어 성능을 최적화 시키기 위해 page가 아닌 record 단위로 저장하도록 설계했습니다. 이렇게 결정한 페이지당 record의 수는 65,280에 LEAF MAX인 31을 곱한 값인 2,023,680이고, 위 값을 bpt.h에 정의했습니다.

```
#define PAGE_SIZE 4096
#define MAX_PAGE_PER_FILE 65280 // 600 MB * 0.85 split by 2 (for safety margin)
#define TOTAL_RECORD_SIZE 2023680  // MAX_PAGE_PER_FILE * 31 (LEAF_MAX)
```

이후, 실제 db_join 함수의 구현 로직에 대해 설명드리겠습니다. Db_join에서 제가 구현한 흐름은 다음과 같습니다. 우선, 각각의 table에서 순차적으로 TOTAL_RECORD_SIZE만큼의 레코드를 동적할당을 통해 생성한 record 배열에 저장하도록 했습니다. Join 시 출력의 순서 관계를 보장하기 위해 함수에 들어와 현재 설정된 fd의 값을 확인하도록 했습니다. 만약 현재 fd 값이 table1_fd값이 아니라면 switch_table 함수를 통해 첫번째 테이블의 fd값으로 설정하도록 했습니다. 이후, 구현한 find_first_leaf함수를 통해 해당 테이블의 leftmost leaf page의 offset을 찾도록 했습니다.

```
if (fd != table1_fd) {
        switch_table(1, table1_fd, table2_fd);
    }

    // find first leaf offset of table 1
    table1_leaf_offset = find_first_leaf();
```

find_first_leaf함수의 경우, rt부터 시작해 가장 왼쪽 pointer인 next_offest 필드를 쭉 따라가도록 설계 했습니다.

```
// find first leaf page offset
off_t find_first_leaf() {
    int i = 0;
    page * p;
    off_t loc = hp->rpo;

    if (rt == NULL) {
        //printf("Empty tree.\n");
```

```
        return 0;
    }
    p = load_page(loc);

    while (!p->is_leaf) {
        loc = p->next_offset;

        free(p);
        p = load_page(loc);
    }

    free(p);
    return loc;
}
```

이후, 만약 이렇게 구현 leaf offset 값이 0이라면 tree가 빈 것이기 때문에 곧바로 함수를 종료합니다. 그게 아닐 경우, 실제 레코드 저장을 위해 table1_records, table2_records라는 이름의 레코드 배열을 동적할당으로 선언했습니다. 이때 크기는 앞서 계산한 510MB를 2로 나누고, 이를 record 크기인 128 바이트로 나눈 크기와 같습니다.

```
record * table1_records = (record *)malloc(sizeof(record) * TOTAL_RECORD_SIZE);
record * table2_records = (record *)malloc(sizeof(record) * TOTAL_RECORD_SIZE);
```

이후, table1의 레코드를 read_page_records를 통해 읽어서 table1_records에 저장하도록 했습니다. 또한 해당 함수에서는 table1_len을 반환하는데 이는 현재 동적할당 된 배열에 존재하는 table1_record의 레코드 수를 의미합니다.

```
// read leaf pages of table1 until reading TOTAL_RECORD_SIZE(2023680) records
    table1_len = read_page_records(&table1_leaf_offset, table1_records);
```

read_page_records에서는 인자로 전달받은, table1_leaf_offest을 옆의 right sibling page로 변경하면서 TOTAL_RECORD_SIZE만큼 읽기 전까지 반복합니다. 각 leaf page의 records를 순차적으로 읽어 배열에 복사합니다. 마지막 조건의 경우 만약 record를 계속 추가하다가 현재 더 넣을 수 있는 record의 수가 31보다 작다면, 새로운 page를 다 못 읽을 수도 있기 때문에 반복문을 종료하도록 했습니다. 이후 해당 페이지는 현재까지 읽은 records의 비교가 끝난 후 다시 읽는 과정에서 다시 메모리로 load 되어 배열에 저장됩니다. 이 값은 table1_leaf_offset 변수에 저장되어 있습니다.

```
int read_page_records(off_t * leaf_offset, record * records) {
    int len = 0, j;
    page * current_page;

    while (*leaf_offset != 0 && len < TOTAL_RECORD_SIZE) {
        current_page = load_page(*leaf_offset);

        // load page records to memory
        for (j = 0; j < current_page->num_of_keys; j++) {
            records[len++] = current_page->records[j];
        }
```

```
        // change leaf offset to right sibling node
        *leaf_offset = current_page->next_offset;
        free(current_page);

        // if there is no room for extra page
        if (TOTAL_RECORD_SIZE - len < 31) {
            break;
        }
    }
    return len;
}
```

이렇게 되면 table1의 records는 다 읽은 것이고, 이제 table2를 읽기 위해 현재의 전역 변수인 fd, hp, rt값을 table2의 것으로 변경합니다. 이후, 동일한 과정을 거쳐 table2의 레코드를 읽게 됩니다.

```
// switch fd, hp, rt for table 2
    switch_table(2, table1_fd, table2_fd);

    // find first leaf page offset of table 2
    table2_leaf_offset = find_first_leaf();

    if (table2_leaf_offset == 0) {
        printf("Empty Tree!\n");
        return;
    }

    // read leaf pages of table2 until reading TOTAL_RECORD_SIZE(2023680) records
    table2_len = read_page_records(&table2_leaf_offset, table2_records);
```

이렇게 두 table의 레코드를 읽어서 메모리에 저장한 후, 실질적인 merge join 알고리즘을 실행합니다. 인자로 받은 table1_len과 table2_len이 0보다 클 경우, 이는 읽을 레코드가 남았다는 의미이기 때문에 0이 될 때까지 반복문을 돌게 됩니다. 반복문에서는 table1_records의 table1_idx, table2_records의 table2_idx를 초기값 0부터 해서 merge join을 수행합니다. 만약 각각의 인덱스의 key값이 같은 경우, 해당 레코드를 key, value1, value2 형식으로 출력합니다. 이후, 여기선 중복이 없기 때문에 각각의 index를 1씩 증가시킵니다. 만약 table1_record의 현재 인덱스에 해당하는 key값이 작은 경우 table1_idx를 1 증가 시키고, 반대의 경우 table2_idx의 값을 1 증가시킵니다. 이를 통해 같은 key를 가진 records를 찾아 natural join을 수행합니다.

```
// merge join approach
    while (table1_len > 0 && table2_len > 0) {
        // if the key value is same
        if (table1_records[table1_idx].key == table2_records[table2_idx].key) {
            printf("%ld,%s,%s\n", table1_records[table1_idx].key,
table1_records[table1_idx].value, table2_records[table2_idx].value);
            table1_idx++;
            table2_idx++;
```

```
        }
        // if the key value of table 1 is smaller than that of table 2
        else if (table1_records[table1_idx].key < table2_records[table2_idx].key) {
            table1_idx++;
        }
        // if the key value of table 2 is smaller than that of table 1
        else{
            table2_idx++;
        }
```

이렇게 반복문을 돌다가 table1_records의 값을 다 읽은 경우, table1_records를 다시 처음부터 채워줍니다. 반대의 경우 table2_records를 다시 처음부터 채워주게 됩니다. 이후, 더 이상 table2_records를 채우지 못할 경우 join은 끝나게 됩니다.

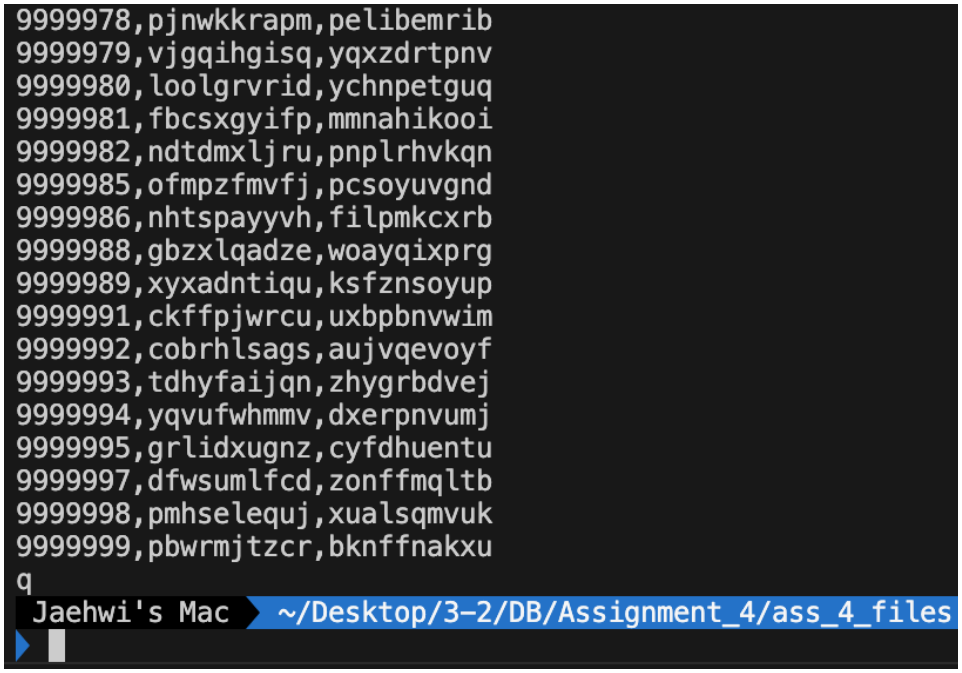
```
// read new page records from table 1
        if (table1_idx >= table1_len) {
            if (fd != table1_fd) {
                switch_table(1, table1_fd, table2_fd);
            }

            table1_idx = 0;

            // read leaf pages of table1 until reading TOTAL_RECORD_SIZE(2023680)
records
            table1_len = read_page_records(&table1_leaf_offset, table1_records);
        }
        // read new page records from table 2
        if (table2_idx >= table2_len) {
            if (fd != table2_fd) {
                switch_table(2, table1_fd, table2_fd);
            }

            table2_idx = 0;

            // read leaf pages of table2 until reading TOTAL_RECORD_SIZE(2023680)
records
            table2_len = read_page_records(&table2_leaf_offset, table2_records);
        }
```

이렇게 join이 끝나면, 동적할동 된 table1_records와 table2_records를 free해주고 함수를 종료합니다.

```
free(table1_records);
free(table2_records);
```

3. Result

이렇게 구현한 DB_JOIN의 실행 결과는 다음과 같습니다. 실제 테스트를 위해 1000만개의 records를 insert를 사용해 각각 table1.db, table2.db 파일을 생성하는 코드를 짜 생성했습니다. 각각의 레코드는 1~10,000,000까지의 key와 random한 문자열을 value로 하도록 생성했습니다. 이때 첫번째, table1.db에서 의도적으로 9,999,983, 9,999,984, 9,999,987, 9,999,990, 9,999,996, 10,000,000의 key 값에 해당하는 레코드를 삭제했습니다. 이후 테스트한 join의 결과는 다음과 같습니다.



총 9,999,994개에 대한 출력이 나오기 때문에 출력 정보는 위와 같이 최소화 했습니다. 출력을 확인한 결과, 앞서 일부로 삭제했던 9,999,983, 9,999,984, 9,999,987, 9,999,990, 9,999,996, 10,000,000의 key 값에 해당하는 레코드들이 생략된 채 적절한 Natural Join이 이루어지는 것을 확인할 수 있었습니다. 로컬 머신에서 테스트한 결과 약 40초 정도의 소요시간이 발생했습니다. 메모리 정보를 추적해본 결과 해당 ./main을 실행시키기 위해 530MB를 사용한다는 것을 확인했습니다. 1GB의 제한조건이 있기 때문에 이 TOTAL_RECORDS_SIZE 값을 조금 더 증가시킬까 고민을 했지만, 채점 환경이 다르기 때문에 여러 변수를 고려해 그대로 유지했습니다. 실행 패턴을 본 결과 설정한 매 2,023,680 레코드마다 경계로 레코드를 읽어오기 위한 대기가 발생하는 것을 확인했습니다.

4. Trouble Shooting

이번 과제를 수행하면서, 가장 큰 시간을 사용했던 부분은 처음 open_table을 통해 두 개의 db파일을 컨트롤 하는 것과 사용할 메모리 용량을 결정하고 구현하는 과정이었습니다. 처음 open_table을 수정하기 위해 fd, hp, rt를 개별 table에 대한 구조체로 묶어 해당 값을 동적으로 변경해줄 계획이었지만, 그렇게 할 경우, 업데이트가 반영되지 않을 뿐만 아니라, 반영하기 위해선 기존의 코드 165곳을 수정해야 했기 때문에 고민을 한 끝에 간단한 방식으로 전역 변수를 통제하는 방법을 고안해 냈습니다. 이렇게 하고 나서, 컴파일을 해보았을 때 insert가 실행되지 않는 문제가 발생했는데, switch_table에 rt가 NULL인 경우에만 free를 해주는 조건문을 추가했더니 위와 같은 문제가 해결되었습니다. 또한, join을 수행하는 과정에서 1GB 조건에서 최대한의 성능을 발휘하기 위해 올릴 record 수를 결정하는 데 많은 시간을 사용했습니다. 연산 과정에서 사용되는 메모리 양을 잘 모르고, 환경이 다른 곳에서 채점을 진행하기 때문에 최대한 보수적으로 로드하는 페이지 수를 설정했고, page 대신 레코드를 저장함으로써 같은 메모리 용량으로 최대한 많은 페이지를 로드하도록 설계하기 위해 노력했습니다.