Project2 MileStone 2

2017030055 윤상현

이번 Milestone 2 에서는 기존의 B+ 트리 코드에서 Tree Modification 이 발생할 때마다 생기는 Disk I/O 로 인한 overhead들을 개선시키는 방법에 대하여 설명하겠습니다. 기존 B+ 트리의 경우,

사전에 정의한 Internal 과 leaf의 최소 key 개수, 최대 key 개수를 기준으로 이를 어긋나는 input 이 들어오는 순간 즉각적으로 split, Merge, redistribution 등의 tree modification 연산이 발생하게 되고, 이는 tree 구조를 변경하므로 disk에 변경된 tree 정보를 기입하거나 받아오며 Disk I/O를 발생시킵니다. Disk에 필요한 data를 write 하거나, 필요한 data를 read하는 것은 disk의 head를 필요한 위치까지 옮기는 physical 적인 작업이 필요하고, 이는 in-memory 상황에서의 I/O에 비해 막대한 시간이 소요됩니다. 이러한 Disk I/O를 절감시키기 위한 가장 확실하고 직관적인 방식은 Cache memory 등 in-memory를 활용하여 다시 쓰일 가능성이 높은 data들을 LRU와 같은 방식으로 선별한 뒤, 불필요한 disk I/O를 줄이는 방법 등이 존재하지만, 이번 Milestone 에서는 이러한 방식을 배제하고 순수하게 B+ 트리 자체의 알고리즘에 대한 개선 사항에 대해서만 작성하겠습니다.

Design & Implementation

Design

저는 B+ 트리의 tree Modification에서 발생하는 disk I/O를 줄이는 방식을 크게 2가지로 분류하였습니다.

1) 1번의 Tree Modification에 발생하는 disk I/O 횟수를 줄이기.
2) Tree Modification 횟수를 최소화하기

먼저 1번 방식, 즉 1번의 Tree Modification에 발생하는 disk I/O 횟수를 줄이기 방식의 경우, 앞서 설명한 Cache memory등을 활용하는 방식이 있으나, 이는 tree 자체에 대한 improvement와는 결이 다르다고 생각되어 배제했습니다.

따라서 제가 선택한 것은 2번 방식, 즉 문제가 되는 tree Modification 자체의 횟수를 최소화하는 것입니다. 제가 처음 생각한 것은 internal, leaf 노드의 최소 key 개수를 감소시키는 것입니다.

Tree modification 중 key delete 시 발생하는 redistribution과 Merge의 경우, 최소 key 개수 이하로 내려가게 만드는 delete 연산시에 수행되며, 많은 disk I/O 가 발생합니다.

따라서 최소 key 개수를 감소시킨다면, delete 연산 수행 시 tree modification을 발생시키는 조건이 완화되어 disk I/O가 감소하게 될 것입니다.

하지만 이 방식의 경우, tree balancing이 자주 발생하지 않게 되고, 결국 tree 높이의 증가로 이어질 수 있는 tradeoff가 생길 수 있습니다.

따라서 제가 실제로 적용한 방식은 logical delete를 사용하여 Merge 연산을 가능한 delay 시키는 것입니다. 우선 제가 logical delete를 사용할 영역은 오로지 Leaf Node(page)로 국한됩니다.

그 이유는 어차피 모든 실제 insert, delete는 leaf에서 시작해서 그 부모 노드를 타고 올라가며 적용되기에, 애초에 leaf 부터 logical하게만 지워주고 physical하게 지우지 않는다면, 상위의 internal 노드 안에 존재하는 key들 역시 지워지지 않기 때문입니다.

자세한 설명은 제가 구현한 코드와 함께 이야기해보겠습니다.

Implementations

우선 기존의 record 구조체에 변경사항이 있습니다. 변경사항은 아래와 같습니다.

```
typedef struct record{
    int64_t key;
    short logical_del;
    char value[120];
}record;
```

기존의 record에 logical_del이라는 short 타입의 변수가 추가되었는데, 이는 해당 record가 logical하게는 지워진 상태라는 것을 의미합니다.

아래는 on-disk B+ 트리 코드 중 실제로 leaf에서 해당 key를 가진 record를 지우는 remove_entry_from_page() 함수와 제가 추가한 사항입니다.

```
void remove_entry_from_page(int64_t key, off_t deloff) {

    int i = 0;
    page * lp = load_page(deloff);
    if (lp->is_leaf) {
        int check = lp->is_leaf ? cut(LEAF_MAX) : cut(INTERNAL_MAX);
        //printf("remove leaf key %ld\n", key);
        while (lp->records[i].key != key)
            i++;


            if(lp->num_of_keys-1  < check  ){ //when this deletion make tree modification, just logically delete it!   mod2
                    //printf("logical del!\n");

                    lp->records[i].logical_del=1;

                    pwrite(fd, lp, sizeof(page), deloff);
                    //write_into_cache(lp, deloff);
                    return;
            }
```

만약 delete 연산을 수행하는 중 해당 delete로 인하여 최소 key 개수 이하의 key를 가지게 되어 tree modification이 필요하다면 해당 key를 보유한 leaf 노드에서 해당 key를 logical 하게만 delete 하고 실제 delete는 수행하지 않습니다.

logical delete의 방식은 record 구조체에 key 와 value 외 logical delete 라는 0 or 1을 가지는 short type을 하나 선언하여 해당 변수가 1이라면 아직 해당 key가 physical 하게 지워지진 않아 여전히 record에 남아있지만 logical 하게 지워진 채 남아 있다는 뜻입니다.

따라서 해당 리코드는 find를 통해 찾아지지 않습니다.

아래는 db_find 내의 변경사항입니다.

```
 if(p->records[i].logical_del==1){ //logical del key search mod2

 // printf("logically del key is searched!\n");
      free(p);
    return NULL;
  }
```

직전에 말했듯이, logical하게 지워진 key들은 아직 tree에 남아있지만, 지워진 key들이기에 find에서 찾아지면 안 됩니다.

따라서 logical하게 지워진 key를 search하게 될 경우, 해당 key가 없는 것으로 판단하고 NULL을 return하여 찾지 못했다는 것을 표시합니다.

이렇게 logical하개만 지운다면 결국 실제 delete는 전혀 수행되지 않기에 insert 하는 key가 늘어날수록 delete로 키를 지우더라도 tree의 높이가 점점 증가하고, 각 노드에서 적절한 key를 찾아 다음 노드로 이동하는 search 작업역시 탐색사간이 증가하게 됩니다.

따라서 한 번씩 logical하게 지운 key 들을 실제로 delete하는 작업을 수행해야 하는데, 이를 언제 수행할지가 다음 쟁점이었습니다.

제가 선택한 방식은 insert할 때 실제 delete를 수행하는 것입니다.

아래는 제가 수정한 insert 내부입니다.

```
off_t leaf = find_leaf(key);
page * leafp = load_page(leaf);

 if ( (leafp->num_of_keys) >= LEAF_MAX){ //if split occur situation, physically delete all logical del before insert occur  mod2


   for(int i=0;i<leafp->num_of_keys;i++){
           if(leafp->records[i].logical_del==1)
                   delete_entry(leafp->records[i].key, leaf); //call del_entry for every logically deleted key.
                   //del_entry func will physically delete keys. but if one more physical delete occur merge, then stop .
  }

}
```

만약 leaf 노드에 어떤 key를 insert할때, 해당 노드가 최대 키 개수를 초과해 가득 차 있다면, 해당 leaf 노드에서 여태까지 logical하게 지웠던 key들을 모두 찾고, 전부 physical하게 제거합니다.

이를 통해 logical Delete를 수행하더라도, tree modification을 줄이기 위한 logical delete로 인해서 split이라는 새로운 tree modification이 trigger 되지 않습니다.

Test&Trade-off

제가 구현한 방식은 delete 와 insert 연산이 골고루 발생하는 상황에서 tree modification을 최소화합니다. 하지만 이에 대한 tradeoff 역시 존재합니다.

만약 delete연산만 계속해서 발생하는 경우, tree는 실제로는 가지고 있을 필요가 없는 데이터를 계속해서 보유하게 되고, 이로 인하여 유효하지 못한 데이터 역시 탐색해야 하기에 특정 key를 find 하는 시간이 감소하며, 이는 insert와 delete 연산의 속도 자체에 영향을 주게 됩니다.

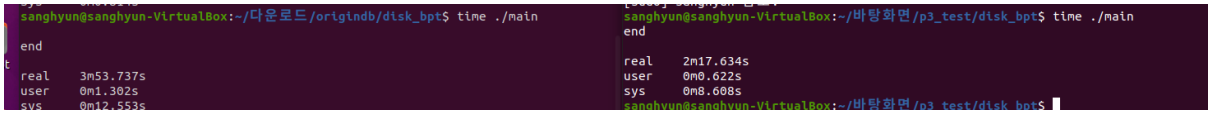
이를 확인하기 위한 test case로 아래와 같은 main 문을 수행해보았습니다.

```c
int main(){
    int64_t input;
    char instruction;
    char buf[120];
    char *result;
    open_table("test.db");



    for(int j=0;j<10;j++){
            for(int i=0;i<10000;i++){

                buf[0]='1';
                db_insert(i, buf);

            }
            for(int i=0;i<10000;i++){
                if(i%2==0)
                        db_delete(i);
            }
    }
    printf("end\n");
     return EXIT_SUCCESS;
     printf("\n");
```

이 test에서는 insert와 delete가 한쪽에 치우치지 않고 일정하게 발생하는 case를 확인합니다.

Insert와 delete를 1만번씩 순차적으로 실행하고, 이를 10번 반복하였습니다.

제가 적용한 improvement는 delete에 비중이 지나치게 많다면 tree가 축소되지 않아 속도가 느려지는 문제가 발생하는데, 이 test와 같이 insert 와 delete가 균등하게 반복될 경우는 insert에서 logical하게 지워진 값들을 그때 그때 physical하게 지워주기에 이런 문제없이 향상된 결과가 나오게 됩니다.



위 사진은 해당 main문을 time을 통해 시간을 측정한 값입니다. 왼쪽이 기존의 on-disk b+ 트리 코드이고, 오른쪽은 제가 향상시킨 on-disk b+트리 코드입니다.

위 결과에서 알 수 있듯이, 제가 구현한 improvement는 확실히 insert와 delete가 한쪽으로 치우치지 않는 상황에선 성능 향상이 확인되었습니다. 왼쪽은 3분 53.737초 정도로 확인되었는데 반하여 오른쪽은 2분 17.634초로 약 1분 30초, 혹은 1.6배 정도 성능이 향상되었습니다.

하지만 만약 delete 연산만 계속해서 수행해야하는 케이스의 경우, 기존의 코드가 제가 향상시킨 코드에 비해 뛰어난 성능을 보여주는 것을 확인하며 제가 작성한 코드의 tradeoff를 체크할 수 있었습니다.

마지막으로 제가 수행한 improvement를 요약하자면 delete애서 tree Modification이 발생할 때는 logical하게 delete하고, 이렇게 logical 하게만 지워진 값들이 insert에서 split을 야기하게 한다면 실제로 제거하는 방식의 improvement를 적용하였습니다.

Trouble shooting

우선 제가 처음 겪었던 trouble은 Cache memory에 관한 내용입니다. 처음엔 단순히 disk I/O자체를 줄이는 최적의 방법이 Cache memory였기에 이를 사용하려 했지만, 이는 B+ tree에 대한 이해도와 무관한, 별개의 방식이었기에 새로운 방식을 적용하였습니다.

추가로 제가 겪었던 가장 큰 문제는 바로 성능 향상에 대한 문제점이었습니다.

제가 생각한 improvement의 방식은 여러가지가 있었지만, 어느 한쪽을 improve 시키면 반드시 그에 따른 tradeoff가 확인되어 완벽한 improvement를 찾는 것에 어려움이 있었습니다.

하지만 방법을 연구하던 중, B+ 트리라는 정체성 속에서 B+트리 자체에서 개선점을 찾는 다면, 어떠한 예외 상황 없이 모든 상황에서 더욱 뛰어난 완벽한 상위호환의 버전을 완성하는 것은 불가능하다는 생각을 하였습니다.

이에 따라 제가 생각한 방법들의 Pros & Cons를 모두 정리하였고, 어떤 방식의 개선점이 상대적으로 적은 risk로 큰 return을 얻을 수 있는지 생각해보며 문제를 해결할 수 있었습니다.

Reference

https://ko.wikipedia.org/wiki/B%2B_%ED%8A%B8%EB%A6%AC

https://velog.io/@emplam27/%EC%9E%90%EB%A3%8C%EA%B5%AC%EC%A1%B0-%EA%B7%B8%EB%A6%BC%EC%9C%BC%EB%A1%9C-%EC%95%8C%EC%95%84%EB%B3%B4%EB%8A%94-B-Plus-Tree