Database Systems

(ITE2038)

2024 B+ tree implementation assignment

컴퓨터소프트웨어학부

2022066953

임정원

1. Overview

해당 과제는 B+ Tree를 직접 구현하는 것이다. B+ Tree는 B Tree의 변형 구조로 데이터베이스나 파일 시스템에서 효율적인 데이터 검색과 정렬된 데이터 접근을 위해 자주 사용되는 자료 구조이다. 이진 탐색 트리와 달리 여러 개의 자식을 가질 수 있는 균형 트리로 모든 데이터를 leaf node에 저장하며, non leaf node, 즉 internal node는 검색에 필요한 키만 저장한다.

B+ Tree의 주요 특징 중 하나는 leaf node가 linked list 구조를 통해 데이터의 범위 검색이 매우 빠르다는 점이다. 이는 대량의 데이터를 다룰 때 트리의 깊이를 얕게 유지하여 디스크 접근을 최소화하는 데 매우 유리하다. Non leaf node가 저장하는 키 값들은 데이터가 저장된 leaf node의 경로를 나타내고 있고 이를 통해 트리의 높이를 줄여 디스크 I/O를 줄여준다.

이 과제에서는 이러한 B+ Tree의 구조를 직접 구현하고 가장 기본적인 기능인 삽입, 탐색, 삭제 연산을 수행할 수 있도록 하는 것이 목표이다. 이러한 기능을 유지하면서 B+ Tree의 구조를 유지하는 것이 중요한 요건이다.

1. Requirements

해당 과제의 요구사항은 다음과 같다.

* 윈도우 혹은 맥 os에서 C++, Python, Java 중 하나를 택하여 구현해야 한다.
* B+ Tree의 index는 하나의 file에 저장되어야 한다. 해당 파일에는 인덱스에 대한 모든 정보와 인덱스 노드들이 포함되어 있어야 한다.
* 단일 키 검색(Search), 범위에 해당되는 키 검색(Range Search), 키 삽입(Insertion), 키 삭제(Deletion) 기능을 제공해야 한다.
* Node에 저장된 키들은 모두 오름차순으로 정렬되어 있어야 한다.
* 기존의 B+ Tree에 관련된 미리 구현된 함수를 사용해서는 안 된다.

1. Implemetation Details

해당 과제는 Mac os에서 python을 통해 구현하였다

main 함수 내에서는 Command line을 통해 어떠한 작업을 실행해야하는지 구분할 수 있도록 argparse 모듈을 사용하여 처리하였다. 옵션은 총 총 6가지로 이루어져 있으며 -c를 통해 index 파일을 생성하고 b 값을 저장한다. -i를 통해 key-value쌍을 삽입하고 이를 index 파일에 저장한다. -p를 통해 B+ Tree구조를 출력할 수 있도록 하였다. 명세에는 print와 관련된 기능이 들어가 있지 않지만 구현을 하는 과정에서 편의성을 위해 삽입하였다. -s를 통해 특정 key를 검색하는 명령을 수행하고 -r를 통해 주어진 범위를 포함한 범위 내에 있는 키들을 검색하는 기능을 수행한다. 마지막으로 -d를 통해 delete 파일 내에 주어진 키를 삭제하는 명령을 수행한다.

Node의 기본 구조는 다음과 같다.

**class** **BPTreeNode**:

**def** \_\_init\_\_(self, leaf=**True**, b=**None**):

self.leaf = leaf

self.keys = []

self.children = []

self.b = b

self.next\_leaf = **None** *# Linked list of leaf nodes*

self.parent = **None** *# Pointer to the parent node*

**class** **DuplicateKeyError**(**Exception**):

*# Duplicate key error*

**Pass**

* Leaf node 여부: 해당 플래그를 통해 노드가 leaf node인지 internal node인지를 구분하였다. Leaf node일 경우 data의 key-value가 쌍으로 저장되어야하고 linked list 형태를 유지하여야 한다.
* Key 배열: 각 node에 저장된 key값을 저장하는 배열이다. 기본적으로 모든 key값들은 오름차순으로 정렬된 상태로 저장되어야한다.
* Children 배열: non leaf node의 경우 children node에 대한 pointer를 저장하는 배열이 필요하다. Leaf node는 자식을 가지지 않으므로 비어있다.
* b: b는 command line에서 주어지는 값으로 각 노드의 자식의 최대 개수를 나타낸다. 따라서 b-1개의 키가 저장될 수 있으며 이를 초과하였다면 split이 일어나야한다.
* Next\_leaf: leaf node간의 연결을 나타내기 위해 포인터를 추가하였다. 각 leaf node가 다음 leaf node를 가리키고 있으며 이를 통해 효율적인 범위 탐색이 가능하다.
* Parent: 해당 포인터는 자신의 부모 node를 가리킨다. Split이 일어나거나 merge가 일어나는 등 다양한 연산이 일어날 때 트리 구조를 유지하기 위해 사용되는 포인터이다.
* 마지막 class는 중복 키가 삽입되면 예외 처리를 하기 위해 추가하였다. 해당 과제에서는 중복 키 삽입이 이루어지지 않는 조건이 추가되었지만 testcase를 돌릴 때 중복 키가 들어오는 경우를 처리하기 위해 해당 class를 삽입하였다.

Tree 구조에 대한 class 정의이다.

**class** **BPlusTree**:

**def** \_\_init\_\_(self, b):

self.root = BPTreeNode(leaf=**True**, b=b)

self.b = b

* 초기 설정을 한다.
* 처음에는 root node가 leaf node이므로 leaf=True로 설정하고 b의 값에 따라 node의 크기가 달라진다.

다음은 다양한 연산을 수행하기 위해 작성한 함수들에 대한 구체적인 설명이다.

먼저 **삽입**을 하는 연산에 관련된 함수이다.

* insert 함수

**def** insert(self, key, value):

root = self.root

*# Insert key-value pair into the leaf node*

self.insert\_in\_leaf\_node(root, key, value)

*# If root is full, split the root*

**if** len(root.keys) > self.b - 1:

new\_root = BPTreeNode(leaf=**False**, b=self.b)

new\_root.children.append(root)

root.parent = new\_root

self.split\_child(new\_root, 0)

self.root = new\_root

삽입은 root에서 시작하여 적절한 leaf node까지 내려가면서 진행이 된다. leaf node에 삽입을 진행해야 하므로 insert\_in\_leaf\_node 메서드로 적절한 node를 찾고 오름차순으로 정렬될 수 있게끔 유지한다. root node의 용량을 확인하여 최대 저장될 수 있는 key의 개수를 초과하면 node를 분할하는 기능을 수행할 수 있도록 추가하였다.

* insert\_in\_leaf\_node 함수

**def** insert\_in\_leaf\_node(self, node, key, value):

**if** node.leaf:

*# Use binary search for leaf node*

pos = self.binary\_search(node.keys, key, is\_leaf=**True**)

*# Insert the key-value pair at the correct position*

node.keys.insert(pos, (key, value))

**else**:

*# Use binary search for internal node*

pos = self.binary\_search(node.keys, key, is\_leaf=**False**)

*# Recursive call to insert the key into the correct child*

self.insert\_in\_leaf\_node(node.children[pos], key, value)

*# If child node is full after insertion, split the node*

**if** len(node.children[pos].keys) > node.children[pos].b - 1:

self.split\_child(node, pos)

node.children[pos].parent = node

node.children[pos + 1].parent = node

해당 함수는 B+ Tree의 leaf node에 key-value 값을 삽입하거나 적절한 자식 노드로 삽입을 진행하는 역할을 한다. leaf node에만 key-value값이 저장되어야 하므로 이를 구분하여 적절한 값을 삽입할 수 있도록 하였다. 삽입할 때는 binary\_search를 진행하여 node의 keys 배열해서 key를 삽입할 위치를 찾는다. 이의 시간 복잡도는 O(log n)으로 정렬된 배열에서 적은 시간을 소모하며 삽입할 위치를 빠르게 찾을 수 있다는 장점이 있다. child node가 삽입을 진행한 후에 가득찼다면 노드 분할을 진행한다. 중간키를 상위 노드로 올리고 중간키를 사이에 두고 두 개의 하위 노드는 왼쪽 자식 노드와 오른쪽 자식 노드로 분할된다. 그리고 분할된 자식 노드들은 parent 포인터를 update해야한다.

* split\_child 함수

**def** split\_child(self, node\_parent, index):

*# Node to be split (left node)*

left\_child = node\_parent.children[index]

*# New node (right node)*

right\_child = BPTreeNode(leaf=left\_child.leaf, b=self.b)

right\_child.parent = node\_parent

**if** **not** left\_child.leaf:

*# Internal node split*

*# Make room for the new key in the parent node*

node\_parent.keys.insert(index, left\_child.keys[self.b // 2])

node\_parent.children.insert(index + 1, right\_child)

*# Split keys between the left and right children*

right\_child.keys = left\_child.keys[self.b // 2 + 1:]

left\_child.keys = left\_child.keys[:self.b // 2]

*# Split child pointers between the left and right children*

right\_child.children = left\_child.children[self.b // 2 + 1:]

left\_child.children = left\_child.children[:self.b // 2 + 1]

*# Set parent pointers for child nodes*

**for** child **in** right\_child.children:

child.parent = right\_child

**for** child **in** left\_child.children:

child.parent = left\_child

*#print(f"Internal node split. Left node: {left\_child.keys}, Right node: {right\_child.keys}, Parent: {node\_parent.keys}")*

**else**:

*# Leaf node split*

*# Make room for the new key in the parent node*

node\_parent.keys.insert(index, left\_child.keys[self.b // 2][0])

node\_parent.children.insert(index + 1, right\_child)

*# Split keys between the left and right children*

right\_child.keys = left\_child.keys[self.b // 2:]

left\_child.keys = left\_child.keys[:self.b // 2]

*# Update next\_leaf pointers*

right\_child.next\_leaf = left\_child.next\_leaf

left\_child.next\_leaf = right\_child

*#print(f"Leaf node split. Left node: {left\_child.keys}, Right node: {right\_child.keys}, Parent: {node\_parent.keys}")*

해당 함수는 B+ Tree에서 child node가 가득 찼을 때 해당 노드를 분할하는 역할을 한다. 이는 leaf node 여부에 따라 다른 로직으로 구현하였다. split을 진행야해야하는 node를 left\_child로 지정하고 right\_child를 새롭게 생성한 노드로 칭하였다. 분할 후 해당 노드는 당연히 오른쪽 자식이 될 노드이다. internal node일 때는 중간키를 부모 노드로 올리고 자식 노드를 절반으로 나누어 부모에 추가한다. leaf node는 중간키를 부모 노드로 올리고 두 leaf node를 나눈 뒤 leaf node간의 연결 리스트 형태를 유지하기 위해서 pointer를 갱신해 주어야한다.

다음은 탐색(Search)에 관련된 함수이다.

* search 함수

**def** search(self, key):

*# Search for the key in the B+ tree*

current\_node = self.root

*# Print the traversal path*

**while** **not** current\_node.leaf:

print(",".join(str(k) **for** k **in** current\_node.keys))

*# Move until a leaf node is reached*

i = 0

**while** i < len(current\_node.keys) **and** key >= current\_node.keys[i]:

i += 1

current\_node = current\_node.children[i]

*# Search for the key in the leaf node and print the value*

**for** k, v **in** current\_node.keys:

**if** k == key:

print(v)

**return**

*# Failed to find the key*

print("NOT FOUND")

탐색은 트리의 root부터 시작해서 internal node를 따라 내려가며 leaf node에 도달하면 해당 node에 키가 존재하는지 확인한다. 만약 키를 찾으면 해당 키의 값이 출력되고 키가 없으면 “NOT FOUND”를 출력한다. internal node를 탐색할 때는 트리 탐색 경로를 출력한다. 탐색 연산은 트리의 높이에 따라 O(log n) 시간 복잡도를 가진다. 트리가 균형 구조를 유지하기 때문에 효율적인 검색이 가능하다.

다음은 범위 탐색에 관련된 함수이다.

* range\_search 함수

**def** range\_search(self, start\_key, end\_key):

*# Search for keys in the given range*

current\_node = self.root

*# Traverse to the leaf node*

**while** **not** current\_node.leaf:

i = 0

**while** i < len(current\_node.keys) **and** start\_key >= current\_node.keys[i]:

i += 1

current\_node = current\_node.children[i]

*# Search for keys in the range and print the key-value pairs*

**while** current\_node **is** **not** **None**:

**for** k, v **in** current\_node.keys:

**if** start\_key <= k <= end\_key:

print(f"**{**k**}**,**{**v**}**")

**elif** k > end\_key:

**return**

current\_node = current\_node.next\_leaf

위의 함수는 주어진 범위 내의 키와 값을 검색하는 기능을 수행한다. start\_key부터 end\_key 범위 내에 존재하는 모든 key-value 쌍을 leaf node에서 찾아 출력한다. B+ Tree의 leaf node는 연결 리스트 구조로 되어있기 때문에 start\_key의 위치를 찾으면 쉽게 범위 검색을 효율적으로 수행할 수 있다.

root에서부터 탐색을 시작한다. 이는 current\_node가 leaf node가 아닐 때까지 반복한다. 현재 노드의 키들과 start\_key를 비교하여 적절한 자식 노드를 선택한다. 해당 leaf node에 도달하면 해당 노드에서 시작하여 범위 내의 키들을 출력한다. 만약 범위가 현재 leaf node를 넘어서는 경우에는 연결된 다음 leaf node로 이동하여 연속적인 범위 검색을 수행한다. 그리고 범위의 끝인 end\_key를 초과하는 키가 나오면 탐색을 종료한다.

해당 함수는 O(log n)의 시간 복잡도로 leaf node에 도달하고 그 이후에는 leaf node간의 연결 리스트를 따라 범위 내 키의 개수 m, O(m) 시간에 키를 탐색할 수 있다.

다음은 삭제(Delete)에 관련된 함수이다.

* delete 함수

**def** delete(self, key):

*# Delete the key from the B+ tree*

current\_node = self.root

node\_parent = **None**

parent\_index = -1

*# Search for the leaf node containing the key*

**while** **not** current\_node.leaf:

node\_parent = current\_node

i = 0

**while** i < len(current\_node.keys) **and** key >= current\_node.keys[i]:

i += 1

parent\_index = i

current\_node = current\_node.children[i]

**for** i, (k, v) **in** enumerate(current\_node.keys):

**if** k == key:

**if** i > 0:

*# Case 1: Non-first key in leaf node*

**del** current\_node.keys[i]

self.balance\_after\_deletion(current\_node, key)

**else**:

*# Case 2: First key in leaf node*

*# Delete the key from the leaf node*

self.delete\_key\_from\_leaf\_node(key, current\_node, node\_parent, parent\_index)

*# Delete the key from the internal nodes*

self.delete\_key\_from\_internal\_node(self.root, key)

**return**

*# If the key is not found in the leaf node, print that it doesn't exist*

print(f"Key **{**key**}** not found in the tree.")

delete는 많은 경우를 따져야하고 트리의 구조를 유지해야하기 때문에 가장 많은 노력이 들어갔다. 해당 함수는 트리 구조에서 주어진 키를 찾아서 제거하고 균형이 깨졌다면 균형을 유지하는 병합 또는 재배치 작업을 진행한다.

먼저 루트 노드에서 시작해서 삭제할 키가 있는 leaf node를 찾기 위해 탐색을 시작한다. 부모 노드와 현재 노드의 인덱스를 추적하기 위해서 node\_parent, parent\_index를 초기화한다. leaf node가 될 때까지 반복하여 자식 노드로 내려가고 현재 노드에서 삭제할 키가 있는 자식 노드의 인덱스를 찾는다. leaf node에 도달하면 모든 key값을 확인하며 삭제할 key가 존재하는지 확인한다. 이를 발견하면 두 가지의 경우로 나눠서 진행한다.

Case 1. leaf node에서 첫 번째가 아닌 키를 삭제한다.

i가 0보다 크다는 것은 leaf node에서 첫 번째 키가 아니라는 것을 뜻한다. 이와 같은 경우라면 해당 키를 삭제하고 트리의 균형을 유지하기 위한 balance\_after\_deletion 함수를 실행한다.

Case 2. leaf node에서 첫 번째에 있는 키를 삭제한다.

이 때는 해당 키를 바로 삭제하기 보다 delete\_key\_from\_leaf\_node 함수를 실행한다. 이를 진행하는 이유는 첫 번째 키를 삭제하는 경우는 위에 있는 internal node에 영향을 미칠 수 있어 추가 처리가 필요하기 때문이다. 가장 처음에 있는 leaf node를 제외하고는 leaf node에 있는 첫 번째 키는 internal node에 존재한다. 이후 delete\_key\_from\_internal\_node를 통해 internal node에 해당 키가 존재하는지를 확인하고 그 키 또한 같이 삭제해준다.

current\_node.keys에서 키를 찾기 못한 경우는 해당 키를 찾지 못했다고 출력한다.

* delete\_key\_from\_internal\_node

Trouble Shooting

* 처음에 구현을 할 때는 시간 복잡도를 깊게 생각히지 않고 빠르고 조금 더 편한 방식으로 구현하였다. 따라서 삽입할 위치를 찾을 때 while문을 통해 key가 들어갈 적절한 위치를 찾고 삽입하도록 구현하였다. 구현을 완료한 후 여러 testcase들을 돌려보면서 insert를 진행할 때 시간이 생각보다 오래걸리는 점을 발견하였다. 적은 수의 data를 삽입할 때는 체감하지 못하였지만 1,000,000개의 data 삽입을 여러번 진행하면서 시간을 출력해 보았을 때 7초에 가까이 되는 시간이 걸리는 것을 확인하고 정렬된 배열에서 삽입된 위치를 찾기 위해서는 이진 탐색이 필수적이구나라는 사실을 알 수 있었다. 이진 탐색을 구현하여 삽입 함수를 다시 수정한 결과 0,5초 정도 시간이 줄어드는 것을 확인할 수 있었다.
* 가장 기본적인 사실이지만 다시금 깨닫게 된 부분이 있었다. 백만개의 숫자를 insert하고 delete하는 과정을 수행할 때 만 개를 삭제해 보았는데 시간이 정말 오래 걸렸었다. 거의 1시간에 준하는 시간이 걸려서 잘되나 test하기도 쉽지 않았고 이를 어떻게 해야하는지 굉장히 고민을 많이 했다. 알고 보니 당연히 delete 할 때마다 index.dat 파일에 삭제된 결과를 쓰도록 구현을 해놨었다. 운영체제 시간에 I/O는 정말 시간이 오래걸린다는 것을 배웠었지만 직접 코딩을 하면서 체감을 해보지 않았기 때문에 그 사실을 간과하고 있었던 것 같다. 최종적으로 삭제를 다하였을 때 마지막에만 파일에 결과를 작성하도록 수정한 결과 0.3초 만에 만개의 delete가 수행되는 것을 보고 파일 I/O는 내가 생각했던 것보다 훨씬 시간이 많이 걸린다는 사실을 배웠다. 앞으로는 이와 같은 실수를 하지 않을 것이다.