B+ TREE IMPLEMENTATION

CSE 5331-001 Team 15

This project implements inserts and naïve delete of the index file organization for the database management system using MINIBASE

Inshaad Merchant
Araohat Kokate
Aindrila Bhattacharya

TABLE OF CONTENTS

Overall Status **2**

Insert implementation 2

Delete implementation 2

File Description **3**

Division of Labor **3**

Logical Errors **3**

Overall Status

Insert implementation

The implementation of the insert was done with a few different possibilities:

1. If there was no tree: the key would be inserted as a new leaf node. The previous page pointer and the next page pointer would point to INVALID_PAGE and the header would only contain the leaf node as a root.
2. If there was a pre-existing tree: This would require insertion, in which case we move to the _insert function:
    a. If the node to be inserted is a leaf node, it checks for space in the existing tree structure for the size of the key and its data and insert the node into that space. If no space is found, we begin the process of having a split to add a new entry and then recursively move the values up. We gather all the records in an ArrayList and sort them. The first half of the sorted elements are placed on the old leaf page and the second half in the new leaf page. The pointers of the two leaf pages are also fixed accordingly. The new leaf page is returned to insert to propagate upwards.
    b. If the node to be inserted is an index node, we check for space and if space is found, we insert and return null (thus indicating that no further splits are to be made recursively). If no space is found, we begin the process of splitting to add the new node. All the entries plus the upEntry are added to an ArrayList and sorted. The first half of the entries are put into the old node and the second half are put into the new node; the middle entry is given to upEntry to propagate upwards. The prev pointer of newIndexPage is set to that of upEntry.

Delete implementation

We implemented the NaiveDelete() method to remove a specific (key,rid) from our B+ Tree without performing any merging or redistribution of nodes. First, we locate the relevant leaf page by calling findRunStart(key, startrid). If this page doesn't exist, we return false, indicating that the key was not found. Next, we examine the current entry in the leaf page. If it's null, we move on to the next page by unpinning the current page and pinning the following one. Once we have a valid entry, we compare its key with our target key. If they match, we check whether its data is LeafData and confirm the rid matches the one we want to delete. If everything aligns, we delete the entry from the leaf page, mark the page as dirty (because we made changes), and return true to show the deletion was successful. If

we exhaust all entries across the leaf pages without finding a match, we return false to indicate that the key was not found.

File Description

No additional files were created for our project. All the files we have used were provided with the initial bundle.

Division of Labor

We conducted a virtual meeting first in which we went over the document and read the instructions carefully about what we have to do in this assignment. We also divided the three functions among ourselves as to who would work on which function. We also conducted one in-person meeting to help each other setup our development environment on omega and begin coding. We used discord to share updates on our code and track progress.

1. **_insert ()** function – Inshaad Merchant
2. **Insert () function** - Araohat Kokate
3. **Naïve Delete () function** – Aindrila Bhattacharya
4. **Report and documentation** – All

By coding together, we were able to help each other with the debugging of the program, and that made it easier to find flaws in the code.

Logical Errors

1. **Incomplete Splitting Logic in _insert() :**

What Happened: Initially, our B+-tree never gained more than one level, no matter how many inserts we performed. The newly created leaf or index pages ended up empty, and the old pages remained full. Essentially, the split code in _insert() either never ran or ran incorrectly.

Why It Happened (Diagnosis): We realized that although we recognized a “no space” condition, we only created a second page (leaf or index) but never moved existing records over to the new page. Instead, the old page was left untouched except for the new record, so the tree structure never expanded.

How We Fixed It: We followed the professor’s pseudocode to:

Gather all existing records (plus the new record) into a temporary list. Clear the old page by deleting all its entries. Redistribute about half the entries back to the old page, the other half to the new page. Promote the "middle" key upward as upEntry for index splits (or use the first key on the new leaf for leaf splits). Return that upEntry to the parent call. After implementing this logic, the B+-tree started splitting properly and building additional index levels as expected.

2. **KeyNotMatchException in the Comparator:**

What Happened: When sorting our KeyDataEntry objects—especially in the newly added "split" code—calls to BT.keyCompare(…) threw KeyNotMatchException. But Java's Comparator.compare() method cannot throw new checked exceptions, so the code refused to compile.

Why It Happened (Diagnosis): Because KeyNotMatchException is a checked exception, we either had to catch it inside the comparator or remove it from the comparator's signature. Java does not allow Comparator.compare() to declare additional checked exceptions.

How We Fixed It: We wrapped the call to BT.keyCompare() in a try/catch, then rethrew any caught KeyNotMatchException as an unchecked RuntimeException:

```
allEntries.sort(new Comparator<KeyDataEntry>() {
  public int compare(KeyDataEntry o1, KeyDataEntry o2) {
    try {
      return BT.keyCompare(o1.key, o2.key);
    } catch (KeyNotMatchException e) {
      throw new RuntimeException(e);
    }
  }
});
```