Project Report on

B+ TREE IMPLEMENTATION

(CSE 5331- PROJECT 1)

Submitted by: Team 07

Shreya Manishkumar Patel (1002026812)

Sanket Rajendrakumar More (1001952737)

Mrunmai Nitin Magar (1002092125)

TABLE OF CONTENTS

OVERALL STATUS:

Implementation of insert() method:

- We have implemented the insert method to facilitate the insertion of keys and Record IDs (RIDs) into the B+ tree structure.
- We first check if the B+ tree has no root yet by examining the root page ID stored in the header.
- If the root page ID indicates that the tree has no root (i.e., equals INVALID_PAGE), we proceed to create a new leaf page and set it as the root of the B+ tree.
- We create a new leaf page (BTLeafPage) and obtain its page ID. Then, we pin the page, ensuring it remains in memory, and set its next and previous page pointers to INVALID_PAGE to signify it as the root.
- Next, we insert the provided key and RID into the newly created root page.
- Once the insertion is complete, we unpin the root page, ensuring it can be flushed to disk if needed, and update the header to reflect the new root page ID.
- If the B+ tree already has a root page, indicating an existing structure, we proceed to insert the key and RID into the existing tree.
- We invoke the _insert method recursively to handle the insertion operation within the B+ tree structure.
- In case a new root entry is returned from the _insert method, indicating that the tree structure needs to be adjusted, we create a new index page (BTIndexPage).
- We insert the new root entry into the index page and set its previous page pointer to point to the current root page ID.
- Finally, we unpin the index page, ensuring it is flushed to disk if necessary, and update the header to reflect the new root page ID.

Implementation of _insert() method:

- We implemented the _insert method to handle the insertion of keys and Record IDs (RIDs) into the B+ tree structure.
- Upon invocation, the method operates on the specified current page, determined by the provided currentPageId.
- If the current page is identified as a leaf node, the method inserts the key and RID directly into the leaf page if sufficient space is available.
- In cases where the leaf page lacks space, the method splits the page, redistributing records between the old and new leaf pages to accommodate the new entry.
- Once the insertion process is completed, the method returns a KeyDataEntry containing the key and the PageId of the new leaf page, facilitating potential adjustments in the tree structure.
- If the current page is an index node, the method recursively navigates to the appropriate child page for insertion.

- Upon return from the recursive call, the method handles index page splitting if necessary, redistributing keys between the old and new index pages to maintain balance.
- Finally, the method returns a KeyDataEntry containing the key and the PageId of the new index page, facilitating potential structural adjustments.
- In cases where the current page type is neither leaf nor index, the method throws an InsertException to indicate an unexpected condition.
- Overall, the _insert method plays a crucial role in maintaining the integrity and balance of the B+ tree structure during key insertion operations.

Implementation of NaiveDelete() method:

- In this functionality, we have incorporated the naiveDelete method, which accepts the parameters <key, recordID>.
- This method is responsible for deleting a key and its corresponding record identifier (RID) from a B+ Tree leaf node.
- We begin by finding the starting leaf page (BTLeafPage) where we expect the given key to be located using the findRunStart method.
- If the leaf page is not found (null), we return false, indicating that the key to be deleted was not found in the B+ Tree.
- We then iterate through the leaf pages and entries to locate the target key for deletion and once we find the key, we attempt to delete the corresponding entry using the delEntry method of the leaf page.
- If the deletion is successful, we return true, indicating successful deletion.
- If the key is not found, we move to the next page and continue the search.
- After completing the deletion process, we unpin the leaf page and return false, indicating that the key was not found for deletion.

SUMMARY:

- We implemented page management functionalities to handle the creation, deletion, pinning, and unpinning of pages within the B+ Tree structure. This ensured efficient memory utilization and minimized disk I/O operations.
- We designed classes to manage different types of nodes in the B+ Tree, including leaf nodes and index nodes. These classes encapsulated functionalities such as key insertion, deletion, splitting, and merging, ensuring consistency and integrity of the B+ Tree structure.
- To handle cases where nodes become full during insertion, we implemented splitting algorithm. When a node exceeds its capacity, it is split into two nodes, and the appropriate keys are redistributed between them to maintain balance.

FILE DESCRIPTION:

We did not create any additional files for the project.
The files provided initially were utilized consistently throughout the project.

DIVISION OF LABOUR:

- Sanket : _insert(index), insert(leaf)
- Shreya: _insert(leaf), naiveDelete()
- Mrunmai: naiveDelete(), report

We spent around 1 week to complete this project.

ENCOUNTERED ERRORS:

1. Naïve Deletion –
   In the NaiveDelete() method, while attempting to delete a node that was already absent in the tree, we encountered issues while printing the message that "The record you entered doesn't exist".

2. The code worked well in Eclipse IDE, but when we ran it on the OMEGA server, the insert operation failed, and the tree remained empty. We suspect there might be compatibility issues between the code and the server.

3. Distribution of data:
   Initially, we encountered a challenge while working on splitting the index and distributing the data to keep the page half-filled. Even though we tried to understand the logic behind the execution, it took us a while to get it completely and implement it into the code.