CSE-4331/5331-001

DBMS Project- 1

B+ Tree Implementation

Prof. Abhishek Santra

2/20/2024

Team 15

Muna Bhattarai

Abhishek Patel

Jeet Sheth

Overview of the Project

The project is about B+Tree index implementation over heap file system. We have implemented the following functions:

Insert(): to insert value at root node of B+Tree Index

To commence, an evaluation of the key length is performed to ascertain its compliance with permissible dimensions. This entails a comparison between the maximum key length specified in the header and the current key length. Should the length fall within acceptable parameters, the process advances to the insertion phase within the B+Tree Index. Conversely, a deviation beyond the permissible length precipitates the issuance of a KeyTooLongException.

After the key length verification, an assessment of the header's status is undertaken to determine the presence of an initialized tree. A null header indicates an absence of pages within the B+Tree, thereby necessitating the creation and pinning of a new leaf page. Given the page's novelty, adequate space for record insertion is guaranteed. The newly created page is then assigned to the next and previous page references, designated as INVALID_PAGE, to reflect its solitary status within the tree. Following the unpinning of the page, the header is updated to reference this new insertion point.

Conversely, a non-null header signifies the existence of a pre-established tree. In this scenario, navigation to the appropriate leaf page for record insertion is required. This is facilitated by invoking the _insert() function, supplied with the key, RID (Record Identifier), and root header as arguments. The function endeavors to integrate the record into the existing B+Tree structure. A non-null return value from the _insert() function indicates a split propagation to the root, necessitating the creation of a new index page to serve as the root. This procedure involves the generation of a new index page, the reassignment of the left pointer to reference the current root, the insertion of the returned KeyDataEntry as a record, and the update of the root pointer to this new index page, thereby ensuring the tree's structural integrity and navigability.

_insert(): to traverse and find a suitable location to insert a new record in B+ Tree Index

The operation commences with the acceptance of three parameters: key, RID (Record Identifier), and PageId. Initially, the designated page is pinned, and a BTSortedPage instance (referred to as curPage) is instantiated to encapsulate the newly created page. Given the ambiguity regarding the page's eventual classification as either an index or leaf node, a BTSortedPage is utilized as a generic container pending determination of the page type. This determination is facilitated through the invocation of the GetType() method from the BTSorted class.

The process bifurcates based on the node type identified, encompassing two distinct scenarios: Leaf Page and Index Page.

Leaf Page Scenario:

If `curPage.GetType()` equates to `NodeType.LEAF`, an instance of BTLeafPage is instantiated using the PageId supplied as an argument. The availability of space within this leaf page is ascertained through the `Available_space()` method. This value is juxtaposed with the requisite space for the key, as determined by `BT.GetKeyDatalength`, which factors in both the key and

node type. Should the leaf node possess adequate space, the record is seamlessly inserted. Subsequently, the page is unpinned, and a null value is returned, signifying the absence of a requirement for further action.

In scenarios devoid of sufficient space, a split operation is necessitated. The current page is pinned, and a new BTLeafPage instance is created and similarly pinned. This new leaf page's next pointer is aligned with the current leaf page's NextPageId, while its previous pointer references the current leaf page. Additionally, the current leaf page's next pointer is adjusted to reflect the ID of the new leaf page. Attention is then directed towards managing the subsequent page (if present) in the sequence. Should the new leaf page's next pointer not be null, the adjacent right page's pointer is recalibrated to acknowledge the new leaf page. This adjustment involves pinning the right page, instantiating a BTLeafPage for it, setting its previous page to the new page's ID, and subsequently unpinning it.

The redistribution of records between the current and new leaf pages is executed with meticulous care to ensure balance. The `getSlotCount()` method facilitates the enumeration of records, enabling iterative processing. Records are transferred from the current to the new leaf page until a point of equilibrium is reached regarding available space. This reallocation process is further refined by the creation of a `KeyDataEntry` variable, `finalEntry`, which serves as a repository for the pivotal record that delineates the boundary between the two pages. The insertion of the new record into the appropriate leaf page is dictated by a comparison of the key values, utilizing the `BT.keyCompare` function. The destination for the new record—either the current or new leaf page—is determined based on this comparison, with an emphasis on maintaining the order.

Following the insertion, both leaf pages are marked as dirty to signal modifications, necessitating persistence of changes. The operation culminates in the return of a `KeyDataEntry` comprising the key of the first record in the new leaf page and the PageId of the new leaf page. This return signifies the propagation of the split to the parent node, indicating a structural modification within the tree necessitating upward adjustment.

Index Page Scenario:

If `curPage.GetType()` does not correspond to `NodeType.LEAF`, it signifies the presence of an index page. Subsequently, an instance of BTIndexPage is instantiated using the provided PageId. The process of insertion into an index page involves an evaluation of the available space within the index node, juxtaposed against the requisite space for accommodating the key and corresponding pointers. This determination is facilitated by the `Available_space()` method. Should the index page possess sufficient space, the new entry is seamlessly integrated. Conversely, if the available space is insufficient to accommodate the new entry, a split operation becomes imperative. The process entails the creation of a new BTIndexPage instance, which is similarly pinned. The subsequent steps involve redistributing the entries between the current and new index pages to ensure balance.

An iterative process is undertaken to transfer entries from the current index page to the new index page until a point of equilibrium is attained. During this redistribution process, the entries are reorganized to maintain the hierarchical structure of the B+Tree. Upon completion of the split operation, a pivotal entry representing the boundary between the two index pages is identified.

This entry serves as a point of reference for determining the appropriate insertion location for subsequent entries. Following the redistribution and reorganization, the parent index page pointers are adjusted to reflect the structural modifications. This ensures the integrity and navigability of the B+Tree. Finally, both index pages are marked as dirty to signify the persistence of modifications. The operation concludes with the return of a `KeyDataEntry` containing the key of the pivotal entry and the PageId of the new index page. This return value signifies the propagation of the split to the parent node, indicative of structural adjustments necessitating upward traversal within the tree hierarchy.

NaiveDelete(): to delete the record from the B+tree index

The procedure commences by utilizing the `findRunStart` function, which accepts the key and RID as arguments and returns a BTLeafPage expected to contain the target key. A boolean variable is initialized to true to facilitate an infinite loop. Within this loop, the leaf page is scrutinized to ascertain its nullity. A null leaf page denotes the absence of any page containing the specified key. In such instances, a notification "No Instance of Record found." is printed, signaling the inability to locate the target record. Conversely, if the leaf page is not null, indicating the discovery of a page containing the target value, it is pinned for further manipulation. Subsequently, the target value is deleted from the page using the `delEntry()` method, which takes a KeyDataEntry as an argument. The return value of this operation, indicating the success or failure of the deletion, is stored in the previously defined boolean variable. Should the boolean variable evaluate to true, signifying the successful deletion of the entry, the page is unpinned, and the message "Instance of Record are deleted." is printed. Conversely, if the boolean variable evaluates to false, indicating the absence of the target entry, the page is still unpinned, and the message "All Instances of Record are deleted." is printed. Subsequently, the loop is terminated, and true is returned to indicate the completion of the deletion process.

Extra Files Addes:
No extra files were added to this project. Changes were only made to BTTree.java file.

Logical Errors:

1. Deleting left side of the pages that does not exists
   For instance, we have the record from 20 to 30. When prompted to delete the record less than 20, the program runs and prints deletion was done if the record existed. This is because the pointer goes to the left pages and searches all the records and since the key is less than the value in the index node it will return the previous page pointer and tries to delete if the record exists.

Work Distribution:

The project was done in a team. In total 23-hour 45 min was spent by each student together while working on the project. The team met at a designated location and worked together rather than splitting tasks among themselves.

Code Review and Understanding	5 hours
Working on insert and _insert function	6 hours
Fixing _insert bugs	4 hours
Working on delete	2 hours 45 min
Testing	2 hours 30 mins
Document	3 hours 30 mins