Database Systems- Project

Lab 4: B+ Tree Index

Chirag Shivakumar 1004996, Devanshi Joshi 1005162, Suthar Prachi Jayesh 1005421

# Design Decisions made during the Project

## Familiarization with B+ Tree Index

In this lab, we were tasked with creating a B+Tree data structure and performing operations like searching, inserting, and deleting keys. The B+Tree is a popular data structure in database systems for indexing data. This report discusses the design decisions made during the implementation of the B+Tree. There are four different types of pages to implement the B+Tree: internal pages, leaf pages, header pages, and a root page. This decision was made to simplify the B+Tree implementation and to provide a clear separation between the various types of pages. The BTreePage abstract class was also designed to hold code that is shared by both leaf and internal pages.

## Exercise 1- Searching

The implementation of the findLeafPage() function in BTreeFile.java involved a recursive search through internal nodes using the BTreeInternalPage.iterator() function to iterate through key values and access child page IDs. The team also dealt with a null key value and used BTreeFile.getPage() to retrieve pages with an extra argument to track dirty pages. Internal pages were fetched with READ\_ONLY permission, while the leaf page was fetched with permission passed to the function as an argument.

## Exercise 2- Inserting

The splitLeafPage() and splitInternalPage() methods were designed to split leaf and internal pages when they became full and could no longer accept new entries.

### Splitting Leaf Pages

The page must be split in order to insert a tuple into a full leaf page. The splitLeafPage() method is implemented by obtaining the parent page with an empty slot and a new page using the getParentWithEmptySlots() method. The new entry is added to the parent page, and half of the tuples from the previous page are transferred to the new page via the iterator() and reverseIterator() methods. Furthermore, the parent pointers of the new pages are updated, as are the sibling pointers of any leaf pages that were split. Finally, as indicated by the provided key field, the method returns the page into which the new tuple should be inserted.

### Splitting Internal Pages

When an internal node becomes full, it is split, and the key is pushed up to the parent page. The splitInternalPage() method is used to implement this operation, which is similar to but not identical to splitLeafPage(). Instead of copying the key, it is pushed up to the parent page, and the parent pointers of all the children that were moved are updated using the BTreeInternalPage class's updateParentPointers() method. Any newly created pages and any pages modified due to new pointers or data are added to the set of dirty pages.

## Exercise 3 and 4- Deleting

In order to implement B+Tree deletions in BTreeFile.java, several functions were created to redistribute tuples/entries, merge pages, and update parent nodes. The design decisions made for these functions prioritized the tree's balance while minimizing space waste. The first three functions (stealFromLeafPage(), stealFromLeftInternalPage(), and stealFromRightInternalPage()) redistributed tuples/entries between pages, updated parent nodes, and kept the tree's structure. The last two functions (mergeLeafPages() and mergeInternalPages()) merged pages and ensured that deleted pages could be reused.

BTreeFile.getPage() was used to encapsulate the process of fetching pages and keeping the list of dirty pages up to date in order to ensure efficient implementation. This strategy ensured that pages were fetched and updated as quickly as possible, and that changes were made in memory before being flushed to disk. It was confirmed that the implementation was successful by passing the unit tests in BTreeFileDeleteTest.java and the system tests in systemtest/BTreeFileDeleteTest.java.

## Additional Exercises- Transactions

SimpleDB requires the implementation of locking to ensure that transactions are executed correctly and without deadlock, in addition to preventing phantom tuples. The BTreeDeadlockTest.java tests were used to ensure that locking was implemented correctly within the B+ tree code.

Strict two-phase locking was implemented at the page level to prevent dirty reads and to ensure that write locks were held until the transaction was completed. This design decision ensures that transactions are correctly executed and do not overwrite changes made by previous transactions. Locks should always be acquired in the same order, regardless of the transaction that requests them, to avoid deadlocks. This was accomplished by acquiring locks in the order of the page numbers to prevent transactions from becoming deadlocked.

# Non-Trivial parts of code

### Exercise 2- Splitting of the leaf and internal pages

#### BTreeFile.splitLeafPage()

The function divides existing pages in a B-tree data structure in two and updates the parent page and sibling pointers to accommodate new entries. It also copies the middle key into the parent page and splits it recursively as needed to accommodate the new entry. Finally, it updates all of the affected leaf pages' sibling pointers and returns the page into which a tuple with the given key field should be inserted.

A method splitLeafPage is defined in the given code, which splits a B+ tree leaf page by adding a new page to the right of the existing page and moving half of the tuples to the new page. The middle key is copied up into the parent page, and the parent is split recursively to accommodate the new entry. To get the parent node with empty slots, use the getParentWithEmptySlots() method. All of the affected leaf pages' sibling pointers are updated, and the page into which a tuple with the given key field should be inserted is returned.

#### BTreeFile.splitInternalPage()

The purpose of this method is to split an internal page to make room for new entries and then split its parent page recursively as needed to accommodate a new entry. The transaction ID, a map of dirty pages, the internal page to split, and the key field of the entry to be inserted after the split are all passed to the method. The method returns the internal page where the new entry should go.

The splitInternalPage method accepts a transaction ID, a map of dirty pages, the internal page to split, and a field to insert into the tree once the split is finished. The method works by creating a new empty internal page, copying half of the entries from the previous page, and pushing the middle key up to the parent page. The parent page is recursively split to accommodate the new entry, and the child pointers on the parent entry should point to the two internal pages that result from the split.

## Exercise 3- Redistributing pages

The code consists of three methods that are used to redistribute entries between pages in a B+ tree implementation.

1. The first method is used to redistribute tuples between a leaf page and its sibling.
2. The second method is used to redistribute entries between two internal pages when the left sibling has more entries than the current page.
3. The third method is to borrow an entry from a page's right sibling when it has fewer entries than the minimum allowed.

The Iterators are used by the methods to move entries between pages, update parent pointers and keys, and flag pages as dirty. The updateParentPointers method is also used to update the parent pointers of all children who have been moved on the current page.

# Justify any changes made to the API- None

# Any missing or incomplete elements in your code- None