Design Report

Project 3: B+ Tree Index Manager

Avichal Rakesh • Shyamal Anadkat • Bryce Sprecher

This project implements a B+ tree index on an integer type search key in a relation. The notion of an index is to maximize performance and reduce IO time. Our implementation of a B+ Tree Index is based on certain assumptions laid out in the project guidelines. First, we assume all records in the file have the same length. This allows for any given attribute to have a consistent offset in the record. Second, our B+ tree supports single-attribute indexing and not composite attribute indices. Third, our implementation builds an index using only integer data types as the search key. The existing implementation can be extended and made generic for other data types as well. Finally, we assume that we never insert duplicate data entries in the index.

How would our design change if we were to allow duplicate keys in the B+ tree ?

+ We could make use of overflow pages to house duplicate keys in the B+ tree. For example, the leaf node could be a pointer to page which contains an additional data structure.
+ Start scan finds the leftmost child leaf node that fits the specified range, so implementation shouldn't be affected by duplicate key values. Any duplicate keys will be accessible through the right sibling via pointers in leaf nodes. Scan next should not be affected by duplicate keys. It iterates through all the keys in the leaf node and any record key pair conforming to the search criteria are returned, so duplicate values can be returned sequentially

Implementation Notes and Design Decisions

- **Notes on the design of constructor**:
    - When the index constructor is called, a check occurs to determine if an index for the specified attribute already exists. If such an index file already exists then it is loaded.
    - If an index file on the attribute does not exist the constructor will create a new index file using the insert entry function. The name of the resulting index file is based off of the attribute used to create it.
    - If the index file exists, we validate the meta-info to make sure they match.

- We also have a rootIsLeaf boolean for the special initial case when the root is the leaf node in the B+ tree.
- We make sure we unpin the required pages : header page and root page.

- **Efficiency**
  - Timing : it takes us about **3.438768s** to create B+ index on relation size of **500k**

- **Notes on the design of insertEntry:**
  inserting Entries into B+ Index Tree is broken down to 7 functions:
  - insertLeafEntry :
    - Function containing the high level logic for adding to a leaf, and splitting the leaf if necessary.

  - splitLeafNode:
    - Function containing the specific logic of splitting leaves to ensure 50% occupancy in both new leaves

  - insertToLeaf:
    - Function containing specific logic to add a record to a non-full leaf.

  - insertNonLeafEntry:
    - Function to recursively traverse the tree and add to appropriate leaf.
    - This Function is also responsible for propagating splits back up if needed.

  - splitNonLeafNode:
    - Function containing specific logic to split a full non-leaf node.

  - insertToNonLeaf:
    - Function containing specific logic to add a key to a non-leaf node

  - insertEntry:
    - This is the public function that triggers an insert.
    - This function also handles splitting of roots. If the root was split, a new root is created and all relevant metadata is updated.

+ **Comments on unpinning pages**
    - Effort was made to ensure that all unused pages are unpinned as soon as possible.
    - To keep the logic and workflow simple, a simple rule was followed: The function that pinned a page will also unpin that page.
    - All unnecessary pages are unpinned as soon as their work is complete.

+ **Notes on the design of start-scan, scan-next, end-scan**
    - Unpinning pages as frequently as we can
    - Start scan finds the leftmost child leaf node that fits the specified range as well as setting up class global variables for the other scanning methods. It iterates through the key value array until the lower boundary value of the search is contained in the corresponding page at that index. Once the first leaf containing the lower boundary is found it is assigned to a global variable that the scan next function will use.
    - Scan next iterates through all the keys in the leaf node and any record key pair conforming to the search criteria are returned. The scan will stop when either the upper value boundary is reached or there are no more leaf nodes in the index. The record is returned via the parameter that was passed to scan next.

+ **Test Cases included**
    - Stress relation test (relation size of 500k)
    - Testing opening/closing of index file
    - Testing empty relation
    - Error tests
    - Add-on int-scan tests