CS 564 Database Management Systems

Kirthanaa Raghuraman

March 26, 2017

Assignment 3 : BTree Index Manager

Design Report

Overall Implementation

- **Implementation:**
  - I have implemented the Btree with the given skeleton code and added a few template helper methods.
  - The initialization is done in the constructor - I check if the index file for the given relation already exists. If it exists, I read the IndexMetaInfo page and populate the index parameters. If it doesn't, I create one, allocate and initialize the index and root node page, scan the relation file and insert records.
  - The insertEntry() function calls another helper function which handles the recursion. The helper function is written as a template function. It traverses the index by fetching the non leaf and leaf pages. Once at the leaf page, it inserts the entry - if the leaf node is full, it is split (which is handled by another helper function). If the node is split, the new key and the page number of the newly created leaf node is passed by reference to previous levels of the recursion. The split causes more inserts/splits in non-leaf nodes depending on their capacity - this is propagated until the root node. If the root node is split, a new one is created, the pointers and index pages are updated.
  - The startScan() starts from the root and traverses the index checking for the scan condition. It traverses till the leaf node that could contain the key and updates the necessary book keeping information. The scanNext() function traverses the leaf node identified by the startScan() function. It identifies the records which satisfy the query and outputs them. It traverses till
    - the key is out of the query range
    - the page number of a rid is 0
    - the right sibling page number is 0.
  - The endScan() effectively ends the scan and cleans up unnecessary pages in the buffer pool. The destructor also does similar stuff - clears buffer pool and deallocates any objects allocated.
- **Efficiency of the implementation:**
  - *Insertion:* The time complexity of insertion in my implementation is $O(d\ log_d\ n)$, where d if the order of the tree and n is the number of nodes in the tree.
  - *Scanning:* The cost of searching in my implementation of BTree is *Time for traversing down the tree to the leaf node + Time for searching in the leaf node.* The time complexity for reaching the leaf node is $O(log_F\ m)$, where m is the number of nodes in the tree and F is the fanout factor. But at the leaf nodes, I traverse the entries from the left to right and the complexity is O(N) at the leaf node, where N is the number of key - record id pairs that can be stored in one leaf node.
- **Pinning/Unpinning pages:**
  I pin the pages whenever I need to fetch a node into memory. I then assign and cast it to the proper class object and unpin the page from memory. This prevents the buffer pool from filling up unnecessarily.
- **Duplicate Keys:**
  - My implementation for insertion and scanning will not change much if duplicate keys come into the picture.
  - The logic for finding which leaf node to insert a record/ finding which leaf node could contain a key will be the same.
  - *Insertion Logic:* During insertion, at the leaf node, instead of directly inserting at the leaf node, if the key already exists, a new page needs to be allocated and the record ids having the same key will be stored on that page. This requires additional book keeping - if a key has multiple records and the number of records which have the same key.
  - *Scanning logic:* During scanning, similar to insertion, there will not be change in logic except at the leaf nodes. At leaf nodes, instead of directly fetching the record ids, we need to fetch additional page from the disk and return all the record ids that match a particular key.