MJ Sun (ms969)
Kevin Chen (kfc35)

B+ Tree Documentation

Implementation Strategy

`BTreeFile::Insert`: When inserting into the B+ Tree, if a node becomes full, it needs to be split into two pages with the same balance and the splitting propagates up the tree. For implementing insert, we grouped the insertion into couple cases.

If the tree is completely empty, the insertion creates a new leaf page as the root and inserts the key into the tree.

If the tree has a root node, then we traverse down the index pages until we find the leaf node to insert into. If the leaf node has room for insertion, we simply insert. If it doesn't, then splitting is needed. Splitting of leaf pages is performed the way that is presented in the write up. We move key value list pairs from the old page to the new page until we reach where the new key is to be inserted. If at this point, the new page has more room than the old page, insert the new key and value along with any existing values associated with the new key into the new page, and while new page still has more room than the old page, continue moving key value list pairs from the old page to the new page. If instead, the old page has more space than the new page, we insert the new key into the old page, and move key value list pairs from the new page back onto the old page until both page have about the same space.

If a leaf has been split, the index needs to be updated. We keep track of all the index pages visited when we traversed to the leaf node, so we go in reverse order up the tree through every visited index and update them as needed. If insertion of values into the index doesn't cause the page to become full, simply insert. If insertion causes splitting, then a new page is created and the splitting commences. Index splitting is similar to leaf splitting except we don't need to worry about moving all the values associated with a key. While checking balance of the two index pages during splitting, we take into account that one key value pair will end up propagating up the index split and not be included in either of the two pages that are being balanced. If the splitting propagates up to the root node, a new root node is created and the special condition is taken care of.

`BTreeFile::DestroyFile`: Destroy file is implemented recursively. It traverses down the tree and frees every page in the tree. It then frees the header page and deletes the file from the database by calling `DeleteFileEntry`.

BTreeFileScan: Our BTreeFileScan keeps track of its own pointers to the current key and record. It uses a PageKVScan object to traverse the nodes. After DeleteCurrent is called, the cursor is pointed to the record after the deleted entry.

Problems Encountered
None

Assumptions
Right after DeleteCurrent in BTreeFileScan, the cursor points at the record after the deleted record.

DeleteCurrent calls are always followed by a GetNext call.

Test Cases
We've added 2 more test cases to the existing interactive test program.

Test 5 tests the buffer pool. It does a series of operation on a BTreeFile. Afterwards, it tests if there are lingering pinned pages in the buffer pool. Our implementation passes this test.

Test 6 tests DeleteCurrent according to our above-mentioned assumption. It contains tests for duplicate cases as well. Our implementation passes this test.

Known bugs
None