B+ Tree Documentation

Ethan Rubinson (ebr45) **Daniel Garay (dg488)**

Implementation Strategy

BTreeFile::~BTreeFile()

When the destructor for the B+ tree is called it is assumed that the only remaining pinned page is the header page. As such, this method simply frees the *dbname* array and unpins the header page assuming it's valid.

BTreeFile::DestroyFile()

To destroy the B+ tree we free all pages and delete the entire index file. To do so, we traverse through the tree in a similar manner as the *Search* and *DumpStatistics* functions and do the following:

- For each sub-tree, navigate throughout all the index records and delete their corresponding leaf pages
- Once the leaf pages are freed delete the index pages themselves
- Finally, once all of the pages (except the header page) have been freed, we delete the delete the database file.

BTreeFile::SplitLeafNode(...)

To split a leaf node we first initialize a new leaf page and move all of the entries from the full page into the new page. Then, we re-link the leaf node chain to account for the newly created page. Once this is done we move the entries back into the new page, starting from the entry with the lowest key in the new page, until both leaf nodes have approximately the same amount of available space left. While this is occurring we simultaneously check to see if we should insert the new entry as well (if the key of the record we are about to transfer is greater than the new key) and do so if this is the case. When this process finishes we check to see if the new key was inserted into the old page. If it wasn't, we insert it now into the new page. Finally, we return the key of the first entry in the new page as well as the new page's ID so that *Insert* can update the proper index node(s).

BTreeFile::SplitIndexNode(...)

This function follows the same procedure as *SplitLeafNode* except for that the new index nodes are not linked together like the leaf nodes are, and that the first

entry of the new index node is deleted (since it will be a duplicate entry when it is propagated up the B+ tree) before the key of said entry is returned along with the page ID of the new leaf node.

BTreeFile::Insert(…)

This function serves to insert a unique key and its corresponding record into the B+ tree. To do so it implements the following algorithm:

- If there is no existing root node:
  - Create a leaf root node and insert the entry
- If there is an existing root node:
  - If the root node is a leaf:
    - If there is room to insert:
      - Insert the entry
    - If there is not room to insert:
      - Split the root leaf node
      - Wrap the two new leafs with an index node
      - Insert an entry with the key and record returned from *SplitLeafNode* into the new index node
  - If the root node is an index:
    - Traverse through the tree to find the proper leaf node to insert the new entry
    - If the leaf node we ended up at has room to insert:
      - Insert the entry
    - If there is not room to insert the entry in the leaf node:
      - Split the leaf node
      - If there is room in its parent index node to insert an entry with the key and record returned from *SplitLeafNode* into the new index node:
        - Insert this entry into the parent index node
      - If there is not room in the parent index node:
        - Split the index node
        - ** If this split node was the root index node of the B+ tree, wrap it and the new index node in another index node and insert the entry with key and record returned from *SplitIndexNode* into this new root index.
        - Otherwise, if there is room in its parent index node to insert an entry with the key and record returned from *SplitLeafNode* into the new index node, do so
        - If neither of these conditions hold, split the parent index node and repeat **

BTreeFile::Delete(...)

This function serves to delete an entry with the designated key and record from the B+ tree. As implementing redistribution and merge on *Delete* was not required, the entry is simply deleted from its leaf page (found by traversing the tree) if it exists. This allows there to be empty leaf pages which is accounted for in the *BTFileScan::GetNext(...)* function. Also, no entries in the index nodes are removed or modified on *Delete*.

BTreeFile::OpenScan(...)

This function initializes a *BTreeFileScan* with the low and high search keys as well as the page ID of the first leaf node that would contain the low key. This page may actually be empty if there is no matching entry for the low key.

BTreeFileScan::~BTreeFileScan(...)

No cleanup is needed when destructing the *BTreeFileScan* object. Any pinned pages are unpinned after each iteration of *GetNext(...)*.

BTreeFileScan::Init(...)

Initializes the *BTreeFileScan* object setting its private members

BTreeFileScan::GetNext(...)

This function returns the next value found in the range [low-key, high-key]. To do so it performs the following:

- If the scan is finished, return DONE
- Otherwise, if the scan has not been started:
    - Find the first entry that would satisfy the low-key requirement, following the leaf node chain if necessary. (Accounts for empty pages)
    - If a valid entry is found:
        - Return it as the first found value and mark the scan as started
    - Otherwise, mark the scan as finished
- Otherwise, if the scan has been started:
    - Get the next entry that would satisfy the range of keys to search. It follows the leaf node chain if necessary.
    - If a valid entry is found, return it
    - Otherwise, mark the scan as finished

Assumptions

- Handling merge and/or redistribution for underflow is not required (as specified on Piazza). As such, test case #4 does not pass.
- There are no duplicate key entries (as specified in the write up)

Test Case Review

- All tests in the automated test suite (except test #4) pass without error
- Manual testing was performed with the manual test suite to test large numbers of inserts as well as corner cases arising in *Delete*. No problems were detected here either
- *DumpStatistics* returns reasonable data for all use cases.

Known Bugs

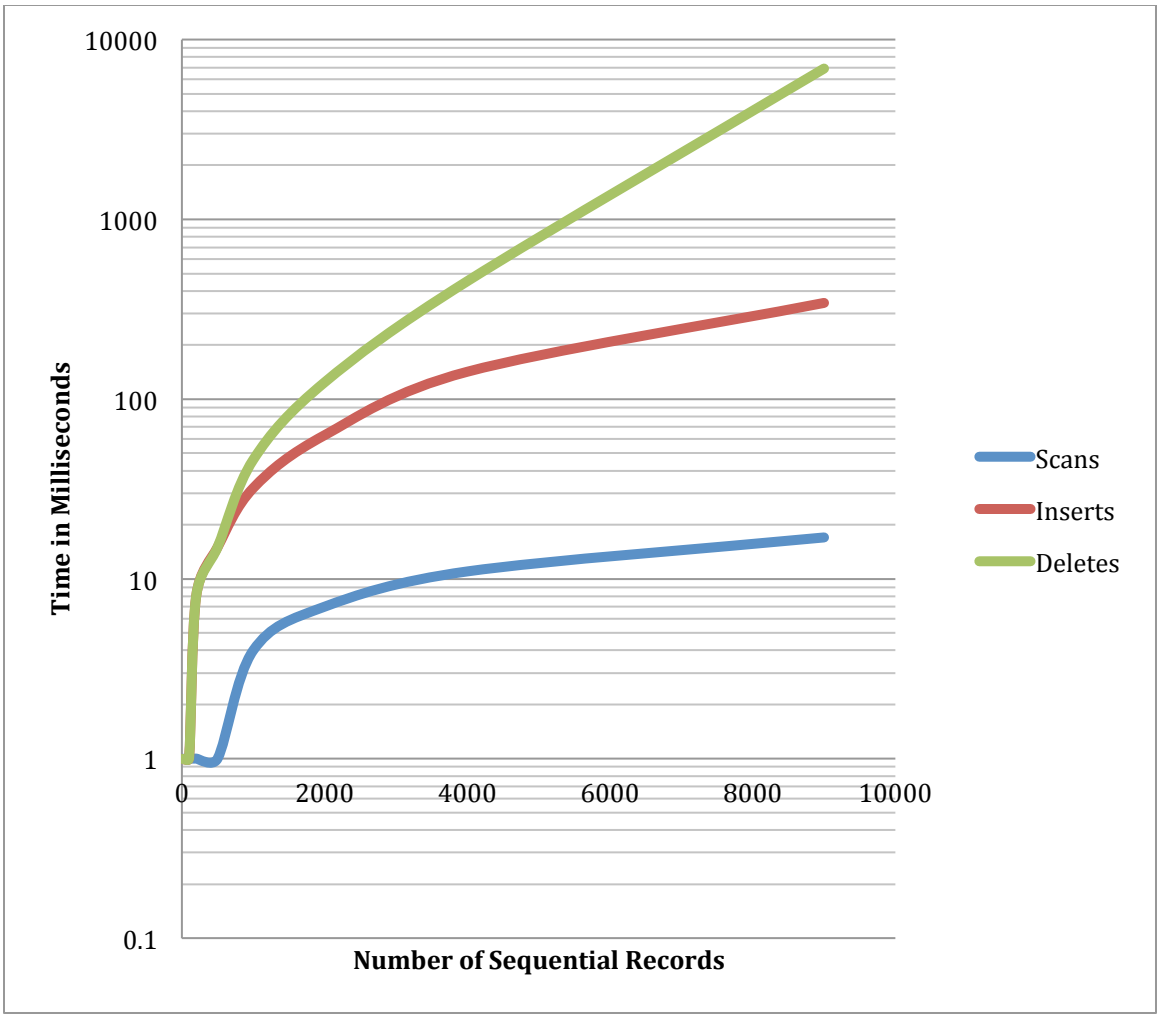
- None

Performance Evaluation [Extra Credit]

- Utilizing the manual test suite and testing the insertion and scan commands for various numbers of sequential records, we took the following measurements and outlined the results in the graph below.
- All printing to standard output was disabled during these tests

For each point, the time required is the average of 3 timed runs for the command on a certain number of records

KEY RANGES	INSERTS	SCANS	DELETES
0-10	<1 ms	<1 ms	<1 ms
0-100	<1 ms	<1 ms	<1 ms
0-200	8 ms	<1 ms	8 ms
0-500	15 ms	<1 ms	15 ms
0-1000	32 ms	4 ms	46 ms
0-2000	63 ms	7 ms	124 ms
0-4000	142 ms	11 ms	453 ms
0-9000	343 ms	17 ms	6905 ms


10000
1000
100
10
1
0.1
Time in Milliseconds
0
2000
4000
6000
8000
10000
Number of Sequential Records
Scans
Inserts
Deletes