Project Three Design Report

Yu Xin, Feifan Wu, Emma He

In this project, we implement a standard B+ tree data structure to store indices of relations. At first, we implement the insert method to build the B+ tree efficiently. Then we implement the search functionality and use it in range checking efficiently. Since this B+ tree is built on pages from bufMgr class, we keep pages pinned for the shortest time needed. We also design additional tests to test the correctness, reliability, reflexivity, and efficiency of our implementations. Additionally, we provide two designs if duplicate is allowed.

To build a B+ tree is to insert the index of a relation one by one. A standard effective B+ tree requires that insert takes O(H) time (H is the height of the tree) and at least half of the capacity is filled up for each node, except for root. Our implementations fulfill these requirements as follows.

The insert functionality is implemented by calling a recursive insert method to traverse the B+ tree from the top (root) to the bottom. When the tree node to be inserted is found, we check if this tree node has enough space. If so, traverse the list of keys in this node to insert the record and return null. Otherwise, split this node, insert the record, create a new root if needed, and return a pointer pointing to the mid-pair. After the insertion is completed, we check the return value. If the return value is null, return null. Otherwise, if current node has space, insert the mid-pair and return null; otherwise, repeat the split process.

When we split one node, we move the right half of the key pair to the newly created node. It ensures that the number of key pairs in all individual node, except for the root node, is at least half of its capacity. This implementation is compatible with the definition of the standard B+ tree.

Let the height of the tree be H, size of the sample be N, and number of keys stored in node/page be t. The only process related to the sample size, N, is to traverse the B+ tree from the top (root) to the bottom to find the node to insert. And the number of nodes to be traversed is approximately the height of this tree, which is H. Once the node is found, a linear search is needed to find the specific location. Our implementation goes over at most t keys only once. Therefore, to insert one record takes O(t + H) time. This is compatible with the time complexity of standard B+ tree insert. Hence, the performance of this implementation is efficient in the sense of time complexity.

Additionally, we test the run time performance by inserting one million data. After repeated testing, the approximate average time to insert one million data into the B+ tree structure is around 20 seconds. Hence, the run time performance of the implementations is also efficient.

Aside from building the B+ tree using insert method, range checking is another important functionality of this project. There are three methods involved: startScan, scanNext, and endScan.

In the startScan method, we traverse the B+ tree from the top (root) to the bottom to find the record to start scanning. When such record is found, we update three private fields to keep track of useful information to locate this record. Then in the scanNext method, we use these private fields to retrieve this record, update these fields, and retrieve the next record all in constant time.

For the scan functionality, startScan takes O(t + H) time, scanNext takes constant time, and endScan also takes constant time. One range search only requires one traversal. Let the range to be searched be V. Hence, one range search takes O(t + H + V) time, which is compatible with the requirement of B+ tree search functionality, since V is independent of the size of relation. Therefore, our implementation of the range search is efficient.

Implementing the above two main functionalities is based on pages from bufMgr class. Keeping a large number of pages pinned at the same time will negatively impact the performance. In our implementation, we keep as few pages pinned as possible.

The basic rule of pinning and unpinning one page is that we unpin one page as soon as we finish manipulating its content. To insert into or read from one node, we only keep one page pinned. If split and new root is needed, we keep three pages pinned: current node, the newly created right sibling to the current node, and the new root. Otherwise (if split is needed but new root is not needed), we keep two pages pinned. For the former two cases, info from all three nodes are needed at the same time.

To implement the insert functionality in recursion, we keep pages pinned along the traversal. Therefore, the number page we pinned for one traversal is equal to the height of the tree, H. The worst case of one traversal requires splitting the node and creating a new root. Hence, in one traversal, the number of page we pinned is at most (H + 2).

To insert the index one million data, at most 4 pages are pinned at the same time. (To store indices for one million data requires a B+ tree with three levels, hence H is 3. And indices for one million data did not use up all leaves which indicates that a new root is not required when H is 3.) Similarly, for the basic tests, at most 3 pages are pinned at the same time. In either case, the number of pages pinned at the same time is relatively small for the sample data. Hence, the number of pages pinned at the same time in the buffer pool is reasonably small which would not affect the overall performance.

After implementing the B+ tree, it is important to test for the correctness, reliability, reflexivity, and efficiency. We designed three types of tests as follows.

We test for the correctness of splitting node using two tests: to test if the root would split if no split is needed; to test if the root would split into one non-leaf node and two leaf nodes when insertion into a full root. The former checks the correctness of the condition for splitting, and the latter tests the one of the complicated processes involving splitting leaf, inserting into leaf, creating new root, updating header page information, creating mid-pair to move up, and inserting into non-leaf.

We test for the correctness, reliability, and efficiency by inserting different size of relation. One extreme case is to test on an empty tree to check the functionality of the constructor. Inserting different size of data and using different order is to test the correctness and reliability in different complicated cases. Among these insertions, we include a large data test, where one million data is inserted. To insert one million data using CreateRelationForward takes up to 8-9 minutes, wheareas inserting one million data into B+ tree takes around 20 seconds. This comparison demonstrates the efficiency of B+ tree.

Additionally, we test for the reflexivity by inserting both positive and negative keys at the same time. This test is to check if our implementation could accommodate a wider range of cases.

In this project, our implementation of the B+ tree is correct and efficient. However, our implementation does not consider the case of duplicate key. To allow duplicate key, we will redesign our implementation in the following two ways:

If the number of duplication of the same key is not high, we will borrow chained bucket from hash table to the rid array in the leaf node. This implementation will benefit data set whose duplication rate of all keys and the number of duplication of one key are not high. And it is simple to implement that each index of rid array stores two pieces of information: rid and pointer to next rid. Furthermore, this implementation will not take up unnecessary memory, because different chained bucket will share one or two pages. Since we do not expect long chained bucket with this implementation, the time complexity for insert and range search will be O(t + H + L), in which L stands for the length of the longest chained bucket.

If storing the chained bucket is costly, we will use overflow page to store all rid records of one individual key and change rid array to pointer array in which each pointer points to the overflow page corresponding to one individual key. Simply changing the rid array to pointer array will not alter the structure and length of arrays in the leaf node. Let the largest number of rid stored in one overflow page be R. The time complexity for insert and range search will be O(t + H + R).

The above two approaches have pros and cons. One important difference between them is that the second approach uses the space of one pointer to store one rid per record, while the first approach uses the space of two pointers to store one rid and one pointer to the next rid per record. If the number of duplication of the same key is not high, using the second approach would leave a lot of page space unused, since one whole page will be assigned to every distinct individual key. Otherwise, if the number of rid corresponding one key could take up more than half the space of one page, using chained bucket would cost more space than overflow page. Hence, the trade-off between these two approaches is whether doubling the number of duplications of the same key is larger than the capacity of one page or not.