Design Report

Yuanhang Wang, Jiujiu Pan, Jimin Lim

1. Algorithm/implementation choice:

For insertion, we first traverse through the tree from root to leaf to find the correct leaf node to insert the key-rid pair. Then, simply insert the key-rid pair in the leaf. If the leaf is full, then split the leaf: create a new page before insertion, move half of data in the old page to the new page, push up the greatest key of the old page, and then insert the new key-rid pair in one of the two pages depending on whether it is greater than or less than the push-up key.

Pushing up key is an insertion to a non-leaf node (internal node), and the process is almost the same as insertion to the non-leaf node.

Edge cases: a) if the node to split is root, then we split the root node and update the root page number and info in the meta page. Updating information is handled by splitRoot() function, and the rest operations are handled in splitAndInsert (leaf split) and splitInternal (non-leaf split) methods.

For the implementation of rangeSearch, we first use starScan to loop from the root, and compare the lowValInt with each of its keyArray element. When we find the first element i >(or >=) than the lowValInt, then we went to the ith child of the root to go to next level until we reach the first leafNode that > (or >=) than the lowValInt, then we set this page as currentPageData , the whole process for finding the first leafNode takes O(logN \* fanout). then we start from the currentPageData to check each of its key in the scan range. When we reach the end of the node, we go to the right sibling and check whether the first key of right sibling greater than the highValInt. If it is, it means it is impossible to have more record in the range so we finish the scan process. This part takes at most O(range + fanout).

1. How often do you keep pages pinned?

If there is no modification to the page, we unpin the page immediately after it is read to a Page pointer. Otherwise, we flush (unpin) the page after all modifications are done.

1. How efficient is your implementation?

For insertion, we only traverse the tree once to find the correct leaf to insert in.

When pushing up keys, we traverse the tree from root once to find the parent node to push-up since there is no other way to find the parent node of a node.

For scan, we only traverse from the root to leaf node once to find the left most leaf contain the low bound value and only traverse each leaf node may contain in-range key once to retrieve the record.

1. Additional design choice:
2. We add a field called size in the leaf and non-leaf struct (page). Reason: there’s no other way to get access to the size of a node. One alternative way is to set all empty slots in the pages with a identifier, say -1 or 0. However in this way every time we want to access the size we have to traverse through all keys in the page and see where the first “-1” show up, which is not time-efficient. The drawback is that a page can store fewer pairs, which I think is a tradeoff between speed and space.
3. Duplicate keys:

Basically, the design won’t change much. One thing is that we try to insert the duplicates into the node that already has a duplicate. This means when splitting we try to avoid pushing up the duplicate keys (although there might be more keys in one node than in another node).

And the allowance for duplication key will not affect the correctness for scanning since starScan will pin the leftmost page which contain the lowValInt and keep scanning toward right until scan all the in-range value or reach the last leaf node.

We have some new tests with larger number of keys in the main.cpp btw.

We also add extra scan test(test7)to check whether scan stops after reach the last leaf node and whether return the correct result for the same low and high range bound value in different operator cases