Design report

Constructor:

For the B-tree constructor we chose to initialize an empty non-leaf root as well as a single empty leaf child. These decision slightly increase the space usage of very small trees, but help to simplify insertions and splitting of nodes. We also use INT_MAX to dente an empty key and 0 for empty values. INT_MAX allows the B-TREE logic to know that all keys are on the left of missing nodes and since page numbers are always at least 1, 0 is a good null value. The construcor also insert the keys into the relation one at a time. This runs the insert method below N times where n is the number keys to be inserted.

The constructor leaves no pages pinned after running.

Insert:

Our insert method preforms a recursive call to a recursive insert method. This recursive method recurs down to the leaf where it inserts the new key, value pair. If a split if required the method preforms it and return the key& value of the right side of the split to the caller. When a recirsive calls returns a pair it is inserts at the appropriateposition.

Insert keeps no pages pinned after it has finished running.

Insert has at most $H + 1$ pages pinned while inserting where H is height of the tree. We did the because we need to read those pages in order to loacte the key location, and it it more efficient to keep the keys in the buffer so that we do not have to re-read pages in the case that we have cascading inserts up the tree. Because B-Tree's do not have very large heights we thought this was a reasonable about of storage to expect the buffer to provide.

This method typically requires H IO's where H is the height of the tree. It can be slightly worse (still $\mathcal{O}(H)$) in cases with many splits, but there cases are infrequent.

Scan:

When a scan is started by start scan, the b-tree traversed down to find the first valid key. If no valid key is found it throws an exception. It then stores the page and location where the key was found. scanNext continues on the page start scan found and increments the stored pointers, if the next key is valid.

startScan will have 1 page pinned at any given time and when it returns it will leave one page pinned(the page with the first entry of the desired search). Throughout the entire scan process, from start scan until the last entry is found via scanNext, on page is kept pinned. We chose to keep one page pinned because we ecpect a sequential scan to refer to the same page in cosecutive calls of scanNext and it is more effieicent to store that page than to have to re-read it hundreds of time.

This method prefroms on scan down the tree to find the first node an then is a lears can through leaves form there. Thus, the method take $H + N$ IO's where H is the hight of the tree and N is the number of leaf nodes containg entries in the range.

Test report

We impemented the follwing tests:

manySearchKeyTest:

This test simply creates an index on a relation of integes [0,5000)

It then scans over the range [-1000,6000]

This test ensures that a scan over the entire b-tree runs correctly and thus, that all leaf nodes are properly connected. It also tests the edges of starting and ending a scan outside of the bounds of the relation.

testBuildIndexOnExisting:

This test creates an index on a relation, deletes it and the creates a new index on the same relation. It then runs several scans to test the performance of the new index.

This is to test the functionality that allows an index to re-open an old index that was created prior.

largeSizeRelation:

This test creates a relation with 1,000,00 entries in a insert order. It then constructs a b-tree on these entries.

This test ensures that root node is split and tests to ensure that leaf splits, interanl node splits, and root splits all function correctly.

size3000RelationSparse:

The creats a realtion with 3000 number randomly chose form the range [0,5000]. It then preforms various scans. This test ensures that the b-tree prefroms well on non-consecutive keys.