B+tree Index Design Report

Project team members:

- Sahib Singh Pandori
  - 9071742556
  - pandori@wisc.edu
- Haylee Jane Monteiro
  - 9071694690
  - monteiro@wisc.edu
- Sreejita Dutta
  - 9075301680
  - sdutta7@wisc.edu

Implementation Design:

- ***insertEntry***
  This first reads the root page and uses the keyArray and corresponding pageNoArray values stored there to find which next node should be traversed to. The b+tree is traversed in this manner to find the parent of the leaf node into which the new entry is to be added. A stack is used to keep track of the path used to reach the level 1 non-leaf node. It uses the parent to reach the appropriate leaf node and calls insertKeyInLeafNode to try to insert the entry in the leaf node. If successful, all the pages in the stack are unpinned. If not, the leaf node is split using splitLeafNode and the first value of the new node created is copied to be insert into the parent. This key is inserted into the parent using insertKeyInNonLeafNode, doing a split if necessary (using spiltNonLeafNode). This is repeatedly done until all the parent pages in the stack are exhaused. At this point, a new root is created and the value of the key from the middle of the old root(that was split) is copied to the new root.
- ***splitLeafNode***
  Creates a new leaf node and initializes it with the default values. The second half of the current node is then copied to the new node (and cleared from the current node). Then, the new key-rid pair is inserted using insertKeyInLeafNode into the corresponding node and the page ID of the new leaf node is returned.
- ***splitNonLeafNode***
  Creates a new non-leaf node and initializes it with the default values. A key value and page ID array is then created to store all the keys and page IDs in a sorted order along with the new key and page ID. The first half of this array is then copied to the old leaf node and the second half is copied to the new leaf node while excluding just the middle key value. This middle key value is then passed by reference to be copied up to the parent. The page ID of the new non-leaf node is returned.

- ***insertKeyInLeafNode***
  Inserts the key in the leaf node at its correct position and shifts the rest of the array to the right by 1 index. If the insert was successful it returns true, but if the leaf node didn't contain space for another key then it returns false (without modigying the node).
- ***insertKeyInNonLeafNode***
  Inserts the key in the non-leaf node at its correct position and shifts the rest of the array to the right by 1 index. If the insert was successful it returns true, but if the non-leaf node didn't contain space for another key then it returns false (without modifying the node).
- ***clearLeafNodeAtIdx***
  Invalidates the entry in a leaf node at a particular index. The page number and slot number of the record pointed to by the index are cleared. While the key is set to -1, it has no significance to the actual validity of the entry. The reason behind this is mentioned later in the report.
- ***clearNonLeafNodeAtIdx***
  Invalidates the entry in a non-leaf node at a particular index. The page number pointed to by the key is set to the INVALID_PAGE value. This is used when a non-leaf node is split and the additional slots need to be invalidated. It is also used during insertion when there is no empty non-leaf node found, and a new root needs to be created.
- ***startScan***
  This method validates the scan parameters, sets up the variables for the scan, and then sets the state of the scan execution to denote that a scan is in progress. It makes a call to getFirstParent to find the parent of the first leaf node that needs to be scanned for this particular range scan.
- ***getFirstParent***
  Looks for the parent node of the first leaf node that needs to be evaluated for the scan. The tree is traversed using the key by comparing it to the lower scan limit. When the appropriate parent node has been found, the currentPageNum is set to the first page node whose values are to be assessed. The search for the page is done sequentially for all the pages in the non-leaf node, looking at their corresponding key values. When the correct leaf page is found, binary search is used on the keys of the entries that are in that page, to find the first entry that should be considered in the scan. Binary search is used so as to improve the efficiency of the scan.
- ***scanNext***
  This method scans the leaf nodes for the next record that satisfies the scan condition. If all the entries in a leaf page have been evaluated, the rightSibPageNo field of the current node is used to go to the next leaf node without having to travel up and then down a level.

Implementation Choices:

- Pages in the buffer manager are unpinned as soon as they are used for whatever was needed. This was done to avoid having too many pinned pages on the buffer, so that processing b+trees with larger number of pages would be easier. It also made it easier

to debug the cause of any PagePinnedExceptions when flushing a file.
The only case when pages are not unpinned immediately is when a scan is executing. Instead, a page is kept pinned until all the records on the page have been read (or if the scan is ended), so that the currentPage being read does not get replaced by another page in the buffer. This does not overuse the buffer since there will only be one page being pinned and read at a time.

- This implementation of a b+tree index is quite efficient. Since all the entries in the tree are stored in sorted order, traversal becomes quicker. Also, each leaf node points to the next one through the rightSibPageNo value, so during an index scan, we can jump from one leaf node to the next without having to travel one level higher in the tree, and then back down to the next leaf page.
  An optimization we did to make the scan efficient is to use binary search to find the value of nextEntry when starting the scan on a particular leaf node. This makes the search for the first value in the b+tree that matches the search condition be logarithmic time instead of linear time.
- In order to check which entries in a non-leaf node are invalid, the value in the pageNoArray for the entry was checked for equality with Page::INVALID_NUMBER. The same is done when checking the validity of an entry in a leaf node, where the page number of the record associated with an entry is assessed. This was done since a key could have any integer value, and there is no dedicated integer value that could be used to denote and invalid entry.

An implementation with duplicate keys:

- The simplification that no two entries with the same key are ever inserted into the index makes the implementation easier. With this assumption, it is easy to have all the entries in the leaf nodes of the b+tree sorted, and have all the entries uniform. It is also easier to determine what entries go in the non-leaf nodes and which side of the non-leaf node should be scanned in order to find a particular entry, since it can only be on one side. Splitting a leaf node with this current design would be complicated if duplicate keys were permitted since having half filled leaf nodes could make things messy. Consider the case where [1 2 2 3] needs to be split to insert another 3. The 2 key will present in two different incompletely filled leaf nodes, [1 2 _ _] [2 3 3 _]. If another 2 were to be inserted instead, it would make it difficult to determine which leaf node the 2 should be inserted into.
- A change in the design of the b+tree would be needed in order to accomodate for allowing duplicate keys in the index. A possible solution would be to modify what a leaf node contains. Instead of having a key map to a RecordId, we could have a key map to a list of RecordIds. In this way, all the records that correspond to a particular key would be accessible through the list. The ridArray in the leaf node would now be of type List<RecodId>. This is analogous to a bucketed hash table which handles collisions.

Additional Tests:

- 6 new tests were added to see how the b+tree index performs with a larger relation size
    - test4: Tests createRelationForward with relationSize 100000
    - test5: Tests createRelationBackward with relationSize 100000
    - test6: Tests createRelationRandom with relationSize 100000
    - test7: Tests createRelationForward with relationSize 1000000
    - test8: Tests createRelationBackward with relationSize 1000000
    - test9: Tests createRelationRandom with relationSize 1000000
- 3 new range tests added to intTests()
    - Both upper and lower bounds set too high
    - Both upper and lower bounds set too low
    - Lower bound set too low and upper bound set too high