Assignment 3 Design Report

Project Group 20:
Jacob Dillie, Austin Lee, Ben Milas

Comp Sci 564 - Spring 2022

Test Report

Compiling, testing, and cleaning can be achieved with the command make test. Note that all tests will take a few minutes to execute due to the large number of insertions. (The given test file, once initialized, should run in a few seconds.)

Additional test methods can be found in the file src/main.cpp. test4, test5, and test6 check additional insertion orders so that edge cases when splitting are properly handled. test7 creates a relation on a pre-existing index that is being re-opened which tests the correctness of our constructor. test8, test9, and test10 check for boundary scan cases across various forms of indexing. These tests are primarily used to find keys that are out of range of the input. Our final relation test, test11, checks that that insertions are properly handled for a relation with sufficiently many records to split an internal node. For this reason, the main file takes a few minutes to execute. The given errorTests method also runs to ensure that the program correctly detects erroneous input.

Notes on Public Method Implementation

The constructor and destructor follow the documentation.

The insertEntry method makes an initial call to recInsertEntry at the root and everything is handled there.

The startScan method traverses to the rightmost child with a key below the scanning range and sets the current page and key to the first record within the range. The page correponding to the node containing this key is left pinned so that it can be read.

The scanNext method sets the currentPageNum and nextEntry values based

on the location of the following key in the index. If these values correspond to key that is out of range when this method is called, we must have reached the end of the range, so an exception is thrown. Note that this method may unpin the current page and use the next sibling if it reaches the end. When the next entry is scanned, it should always have a pin count of 1 so that it can be immediately unpinned once we are done with it.

The endScan method uses a try-catch block to generalize for all cases of a terminated scan. In some cases, the current page will be pinned; however, this is not always true. Implementation otherwise follows the documentation.

Helper Methods

The recInsertEntry Method serves as the traversal component of insertion and implements logic for the current level. It inserts and splits as necessary and returns a PageKeyPair to indicate whether a pivot must be added to the parent node. In a recursive call, the current page is left pinned so that we do not lose data; if a non-null PageKeyPair, this method places the Key and Page into the current node. Once we are done at this level, the current page is unpinned. Recursive calls always initially begin at the root of the tree.

The addEntry method adds a Key and PageId/RecordId to the correponding arrays of a given node. It uses a boolean parameter to distinguish whether it is inserting into an internal node or a leaf. This method is used both by addEntryAndSplit for copying entries and recInsertEntry

The addEntryAndSplit method copies the right half of the entries in a specified node and places them into a right sibling. Upon completion, it inserts the given Key and PageId/RecordId. This method is only called by recInsertEntry when it detects that the node is full to ensure that the minimum number of splits occur.

The checkRange method is only used when scanning. It examines a key and returns a boolean tuple by reference according to the scanning parameters. Low indicates if the key is below the scanning range and high indicates if key is above the scanning range. Note that a key can be low, high, or neither.

Additional Private Members and Fields

The private PageId initRootPageNum keeps track of the very first root page in the tree (prior to splits). This assists with implementing logic in several meth-

ods. In particular, it is used in traversal methods (insertion and scanning) to determine if we are starting at a leaf. It also helps us set level fields correctly when splitting internal nodes.

The private boolean first was used in the scanning methods to determine if the page being examined is the first page with a valid entry. This tells us whether or not the previous page was pinned and if the current page is pinned.

A field numEntries was added to both node structures as this allowed us to determine the number of keys present in constant time. In both node types, keys were stored in arrays, so this allowed for efficient parsing. The value of numEntries for a given node is updated during each insertion and split.

Efficiency of Operations

The time complexities of the two main procedures, Insert and Scan, depend mostly on the implementation of locate; however, locate will occur in $\mathcal{O}(\log_f N)$ time as we traverse down the tree for the leftmost matching key. Note that we follow the standard conventions from lecture where f represents fanout and N represents the number of leaves. In addition to locating, additional operations needed for each will carry an impact on the overall runtime.

For insertion, we must first locate the leaf that is leftmost among those entirely greater than the key we are inserting. From this point, we traverse leftward along the leaf until we find the correct place of insertion (possibly moving to index 0 of the right sibling). After the point of insertion is chosen, we must shift all keys to the left of the insertion point over by one index. Further, if we exceed the capacity at any level, we must split the node, but in the worst case, we could split at every level. Shifting and copying of entries at a leaf level can occur in $\mathcal{O}(d)$ time, where d is the capacity of a leaf and $\mathcal{O}(f)$ time for an internal Node. Therefore, the worst case time complexity for insertion is $\mathcal{O}(\log_f N) + \mathcal{O}(d) + \mathcal{O}(\log_f f) = \mathcal{O}(\log_f(Nf))$, the combined time of location and shifting/adding keys.

For scanning, we are even more efficient than locate. For the initial scan, we are faced with an $\mathcal{O}(\log_f N)$ time complexity for finding the first entry, but after that, we can find the next page in $\mathcal{O}(1)$ time. After the initial page identifier is found, we examine the next one by making four comparisons to determine if it is in range, unpin the page if applicable and then pin. Afterwards, we get the index of the next entry and this requires that we examine if the current page id is in the max index of the current leaf. When we are ending the scan, we simply set scanExecuting to false and unpin the current page, so this also occurs in constant time. In the worst case; however, all pages are within range

so the entire scan can be $\mathcal{O}(N)$ time.

In summary, we have the following worst case time complexities:

- Insertion: $\mathcal{O}(\log_f(Nf))$
- Initial Scan: $\mathcal{O}(\log_f(N))$
- Scan next element: $\mathcal{O}(1)$
- Scan entire range: $\mathcal{O}(N)$